Towards Scalable Synchronization on Multi-Cores

THÈSE N^{O} 7246 (2016)

PRÉSENTÉE LE 21 OCTOBRE 2016

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE PROGRAMMATION DISTRIBUÉE
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Vasileios TRIGONAKIS

acceptée sur proposition du jury:

Prof. J. R. Larus, président du jury
Prof. R. Guerraoui, directeur de thèse
Dr T. Harris, rapporteur
Dr G. Muller, rapporteur
Prof. W. Zwaenepoel, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2016

“*We can only see a short distance ahead,*
but we can see plenty there that needs to be done.”
— Alan Turing

To my parents,
Eirini and Charalampos

Acknowledgements

"*The whole is greater than the sum of its parts.*"
— Aristotle

To date, my education (i.e., diploma, M.Sc., and Ph.D.) has lasted for 13 years. I could not possibly be here and sustain all the pressure (and of course the financial expenses) without the help and support of my family, Eirini (my mother), Charalampos (my father), and Eleni (my sister). I want to deeply thank them for being there for me throughout these years.

In my experience, a successful Ph.D. thesis in the area called "systems" (i.e., with a focus on software systems) requires either 10 years of solo work, or 5–6 years of fruitful collaborations. I was lucky enough to belong in the latter category and to have the chance to collaborate with many amazing people in producing the research that is included in this dissertation. This is actually the main reason why in the main body of this dissertation I use "we" instead of "I."

First and foremost, I would like to thank my advisor, Rachid Guerraoui. Naturally, without him this dissertation would not exist. Rachid is among the most clever and optimistic people I have ever met. To him, I owe three of the most important attributes which I gained during my Ph.D.: (i) being laconic, (ii) optimism, and (iii) my "marketing skills." The first one can be simply explained by his single-sentence answers to several paragraphs long e-mails. My optimism towards research and my marketing skills can be summarized by my current belief that *there are no "bad" research results, but there are definitely bad ways to present those results.* In other words, Rachid taught me that if the research work is solid, people will always be interested to read about it.

As I mentioned earlier, my work is a result of several fruitful collaborations. I first want to thank Tudor David, with whom we traveled, partied, but also wrote the first two papers of my thesis. Similarly, Javier Picorel has been a close friend throughout the five plus years of my Ph.D. After a couple of years of friendship, we did not resist and decided to collaborate on a very cool project (Chapter 5), where we combined my software with his hardware expertise. On that same project, I also had the chance to collaborate with Babak Falsafi, who I deeply appreciate for his advice on how to present and improve my research results.

Additionally, I wish to thank Tim Harris, who offered me the great opportunity to join Oracle Labs for a three-months internship in Cambridge and who also was one of the five members of my Ph.D. defense jury. Albeit the short duration of that internship, we were able to finalize the project and, with some additional work here at EPFL, to write a paper (Chapter 7). For that same project, I want to thank my friend and collaborator Georgios Chatzopoulos for his help.

Furthermore, I must thank the three other committee members for my private Ph.D. defense, namely Gilles Muller, Willy Zwaenepoel, and James Larus. Our discussions during the defense, as well as their written comments, helped me improve the quality of my dissertation.

Naturally, as a Ph.D. student my main/only interest was to solve problems. Still, I had zero interest in solving problems unrelated to research, such as traveling expenses and other bureaucratic stuff, or installing and configuring processors and software. Luckily, I did not really have to handle any of these distractions due to the help of the two secretaries of our lab, Kristine Verhamme and France Faille, and our system administrator Fabien Salvi. Similarly, I want to thank Peva Blanchard for translating my thesis' abstract to French.

Of course, many more people contributed in a way or another to this Ph.D. I would like to thank my colleagues and my friends (many people belong to both groups) for making my everyday life simpler and happier. Without David, Javier, Iraklis, Nadia, Christina, Iris, Matt, George, Tudor, Adi, Matej, Karolos, ... my life during these five years would have been unbearable. In particular, with Iraklis Psaroudakis, Javier Picorel, and David Kozhaya we started the Ph.D. in the same year. I am lucky and grateful to have them as my friends, as they are the people with whom I "cried" about the difficulties of the Ph.D., I celebrated happy moments, etc. Similarly, my "non-Lausannois" friends were always there for me, to listen to my problems over Skype and to spend awesome vacations together. Last but not least, I want to give special thanks to my girlfriend Bojana Paunovic, who arrived in my life almost a year ago and made this past complicated year of my life significantly more pleasant.

Lausanne, EPFL, 29 September 2016 Vasileios Trigonakis

Preface

This dissertation presents part of the Ph.D. work performed under the supervision of Prof. Rachid Guerraoui at EPFL in Switzerland. The main results of this thesis appeared originally in the following conference publications (*author names are in alphabetical order*).

1. Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. "*Everything you always wanted to know about synchronization but were afraid to ask.*" Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP). ACM, 2013. (Chapter 4)
2. Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. "*Asynchronized concurrency: The secret to scaling concurrent search data structures.*" Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, 2015. (Chapter 6)
3. Rachid Guerraoui, and Vasileios Trigonakis. "*Optimistic concurrency with OPTIK.*" Proceedings of the Twenty-First ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM, 2016. (Chapter 6)
4. Babak Falsafi, Rachid Guerraoui, Javier Picorel and Vasileios Trigonakis. "*Unlocking energy.*" Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC). USENIX, 2016. (Chapter 5)
5. George Chatzopoulos, Rachid Guerraoui, Tim Harris, and Vasileios Trigonakis. "*Abstracting multi-core topologies with MCTOP.*" Under submission. (Chapter 7)

Besides the above-mentioned publications that constitute the backbone of this thesis, I further worked on the following conference publications (*author names are in alphabetical order*):

1. Vincent Gramoli, Rachid Guerraoui, and Vasileios Trigonakis. "*TM^2C: A software transactional memory for many-cores.*" Proceedings of the Seventh European Conference on Computer Systems (EuroSys). ACM, 2012.
2. Jelena Antic, Georgios Chatzopoulos, Rachid Guerraoui, and Vasileios Trigonakis. "*Locking made easy.*" Proceedings of the Seventeenth Annual Middleware Conference (Middleware). ACM, 2016.

Abstract

The shift of commodity hardware from single- to multi-core processors in the early 2000s compelled software developers to take advantage of the available parallelism of multi-cores. Unfortunately, only few—so-called embarrassingly parallel—applications can leverage this available parallelism in a straightforward manner. The remaining—non-embarrassingly parallel—applications require that their processes coordinate their possibly interleaved executions to ensure overall correctness—they require *synchronization*. Synchronization is achieved by constraining or even prohibiting parallel execution. Thus, per Amdahl's law, synchronization limits software scalability.

In this dissertation, we explore how to minimize the effects of synchronization on software scalability. We show that scalability of synchronization is mainly a property of the underlying hardware. This means that synchronization directly hampers the cross-platform performance portability of concurrent software. Nevertheless, we can achieve portability without sacrificing performance, by creating design patterns and abstractions, which implicitly leverage hardware details without exposing them to software developers.

We first perform an exhaustive analysis of the performance behavior of synchronization on several modern platforms. This analysis clearly shows that the performance and scalability of synchronization are highly dependent on the characteristics of the underlying platform. We then focus on lock-based synchronization and analyze the energy/performance tradeoffs of various waiting techniques. We show that the performance and the energy efficiency of locks go hand in hand on modern `x86` multi-cores. This correlation is again due to the characteristics of the hardware that does not provide practical tools for reducing the power consumption of locks without sacrificing throughput.

We then propose two approaches for developing portable and scalable concurrent software, hence hiding the limitations that the underlying multi-cores impose. First, we introduce OPTIK, a new practical design pattern for designing and implementing fast and scalable concurrent data structures. We illustrate the power of our OPTIK pattern by devising five new algorithms and by optimizing four state-of-the-art algorithms for linked lists, skip lists, hash tables, and queues. Second, we introduce MCTOP, a multi-core topology abstraction which includes low-level information, such as memory bandwidths. MCTOP enables developers to accurately and portably define high-level optimization policies. We illustrate several such policies through four examples, including automated backoff schemes for locks, and illustrate the performance and portability of these policies on five platforms.

Keywords: multi-cores, concurrency, synchronization, locking, message passing, concurrent data structures, scalability, energy efficiency

Résumé

L'évolution des architectures matérielles des processeurs mono-coeur aux multi-coeurs, au début des années 2000, a incité les développeurs de logiciels à profiter du parallélisme fourni par ces processeurs multi-coeurs. Cependant, très peu d'applications (dites trivialement parallélisables) peuvent bénéficier de ce parallélisme d'une manière simple et directe. Les autres applications (non trivialement parallélisables) demandent à ce que les processus se coordonnent au cours de leurs exécutions entrelacées afin de garantir l'exactitude de leur calcul : ces applications nécessitent de la *synchronisation*. La synchronisation des processus est obtenue en contraignant, voire même en empêchant certaines exécutions parallèles. En conséquence, conformément à la loi d'Amdahl, la synchronisation limite la scalabilité logicielle.

Dans ce mémoire, nous examinons les moyens de minimiser les effets de synchronisation sur la scalabilité. Nous montrons que la relation entre scalabilité et synchronisation dépend en grande partie de l'architecture matérielle sous-jacente. Dit autrement, la synchronisation affecte directement la portabilité de la performance des programmes concurrents à travers les différentes plate-formes matérielles. Néanmoins, il est possible de garantir la portabilité sans sacrifier les performances, en forgeant des schémas de conception (design patterns) et des abstractions, qui tirent profit implicitement des détails matériels sans les exposer aux développeurs.

Nous présentons d'abord une analyse exhaustive des performances de synchronisation sur différentes plateformes récentes. Cette analyse montre clairement que la performance et la scalabilité de la synchronisation dépendent fortement des caractéristiques de la plateforme sous-jacente. Nous nous concentrons ensuite sur la synchronisation à base de verrous (lock-based) et analysons les compromis entre consommation énergétique et performance que présentent les diverse techniques d'attente. Nous montrons que la performance et l'efficacité énergétique évolue conjointement sur les récentes architectures multi-coeurs `x86`. Cette corrélation est due, encore une fois, aux caractéristiques du matériel qui ne fournit pas d'outils pratiques permettant de réduire la consommation énergétique des verrous sans affecter le débit.

Nous proposons deux approches pour le développement de logiciel concurrent qui soit à la fois portable et scalable, masquant ainsi les limitations que l'architecture multi-coeurs sous-javente impose. En premier lieu, nous présentons OPTIK, un nouveau schéma de conception (design pattern) pratique permettant de forger et d'implémenter des structures de données concurrentes à la fois rapide et scalable. Nous illustrons la pertinence de notre schéma OPTIK en présentant cinq nouveaux algorithmes, et en améliorant quatre des plus récents algorithmes, pour les listes chaînées, les listes à enjambement (skip list), les tables de hachage, et les files. En second lieu, nous présentons MCTOP, une abstraction de la topologie multi-coeurs qui offre des informations bas-niveau, telles que la bande-passante mémoire. MCTOP permet au développeur de définir, de manière précise et portable, des politiques d'optimisation haut-niveau. Nous illustrons plusieurs de ces politiques à travers quatre examples, dont des schémas de retrait automatique pour les verrous, et analysons la performance et la portabilité de ces politiques sur cinq plateformes.

Mots-clés : multi-coeurs, concurrence, synchronisation, verrou, passage de message, struture de données concurrente, scalabilité, efficacité énergétique.

Contents

List of Figures

List of Tables

1 Introduction

Moore's law [157] and Dennard's scaling rule [53] are the two principles that used to govern processor design for almost four decades. Moore's law suggests that the number of transistors that can be inexpensively placed on an integrated circuit can double approximately every two years. Dennard's rule claims that every technology generation brings a doubled integration capacity of transistors that are 40% faster in the same power envelope as the previous generation. While Moore's law is still effective [114], Dennard's scaling rule's life came to an end in the early 2000s, mainly due to sub-threshold voltage leakage and heat dissipation limitations [26]. The end of Dennard's rule signaled the transition from single- to multi-core processors. Today, embedded, desktop, and server processors are all multi-cores. As the name suggests, a multi-core processor includes several CPU cores that can execute software threads in parallel.

This shift of commodity hardware from single- to multi-cores also triggered a transition towards the need for software parallelism. In other words, in order to improve performance of a software application on multi-cores, developers have to employ parallelism for utilizing the multiple cores of the underlying machines. As it was simply put in 2005 by Herb Sutter, "The free lunch is over [203]," meaning that developers have to explicitly harvest the available hardware parallelism in their software.

For a certain type of applications, called *embarrassingly parallel*, extracting parallelism is relatively straightforward. For example, data parallel problems require applying certain computations on large amounts of data (e.g., MapReduce-type of computation [52]). The most important characteristic of embarrassingly parallel applications is that they do not include accesses to shared state.

In contrast, non-embarrassingly parallel software, such as operating systems and databases, must access shared state. For instance, modern operating-system schedulers include shared thread queues. Similarly, most components of database systems, such as indexes and the transaction manager, are also shared. Concurrent processes have to coordinate their accesses to this shared state in order to ensure correctness—they must *synchronize*. For example, consider the simple code for decrementing an unsigned counter by one if the counter is greater than zero: `if (c > 0) { c = c - 1 }`. If `c` is a shared counter and two processes execute this code without synchronization, it can happen that both processes simultaneously read `c = 1` and decide to decrease the value of `c`, potentially resulting to the incorrect state `c = -1`. The goal of synchronization is to disallow such erroneous executions.

1.1 Synchronization Primer

Synchronization stems from the Greek words "syn" (*with*) and "chronos" (*time*) and denotes the act of coordinating the execution of a set of processes. As the only purpose of synchronization is to ensure correctness, synchronization can be seen as mere overhead (i.e., it does not contribute to the forward progress of the high-level application computation). Typically, synchronization is achieved by limiting or even prohibiting parallelism. Thus, per Amdahl's law [8, 106], synchronization hinders software scalability. In practice, synchronization often imposes scalability bottlenecks in concurrent systems [17, 28, 29, 33, 35, 42, 43, 44, 136, 145], which can be notoriously difficult to remove (e.g., the global locks in the Linux kernel [21, 134]).

1.1.1 Synchronization Techniques

Synchronization on top of coherent shared memory can be implemented in various ways. Still, regardless of the implementation specifics, all synchronization approaches are bound to rely on the low-level primitives that are available on cache-coherent multi-cores, such as loads, stores, memory barriers, and compare-and-swap. Additionally, all synchronization techniques must adhere to the memory consistency model of the processor, which dictates how hardware can re-order the memory operations of a core [201].

In most practical systems, synchronization is implemented with *locks* (i.e., *mutual exclusion*). As the purpose of locks is to serialize the accesses to shared resources, locks directly hinder scalability. Additionally, locks are often cited for several correctness and performance issues, such as deadlocks, priority inversion, and lock convoying [41, 103]. Still, locks are used in essentially every modern concurrent software system, mainly due to their simplicity compared to other synchronization techniques.

Alternatively, there is a trend towards *lock-free* designs for increasing parallelism and alleviating the problems associated with locks in concurrent systems [27, 54, 144]. However, lock-free programming is cumbersome and is thus impractical for widespread use. Instead, data sharing is commonly "hidden" behind the interface of *concurrent data structures*, such as linked lists and hash tables. Concurrent data structures offer certain benefits: (i) they are a higher-level abstraction than locks, (ii) they are usually designed by concurrency experts, and (iii) they can be implemented using lock-free or lock-based techniques.

Of course, there are more approaches to synchronization. Two other prominent ones are *message passing* and *transactional memory*. With message passing, processes communicate through messages, although they might execute on a shared memory multi-core. The main benefit of message passing stems from the explicit inter-process communication, which allows for better control of process and data placement [17]. Message passing is the main synchronization mechanism in various programming languages such as Erlang, Go, or D.

Finally, *transactional memory* (TM) [102, 197] provides the transactional construct to programmers. The programmer has to wrap her sequential code within the delimiters of a transaction. The TM runtime is then responsible for synchronizing concurrent transactions. Although TM has gained momentum due to the recent introduction of a restricted version of TM in Intel processors [113], TM has not been widely-used in practice due to its limited availability.

1.1.2 Towards Scalable Synchronization

As synchronization can been seen as a mere execution overhead, a synchronization scheme is said to scale if its performance does not degrade as the number of processes increases. For example, acquiring a lock should ideally take the same amount of time regardless of the number of processes sharing that lock.

There is an immense amount of work regarding synchronization (e.g., [1, 10, 17, 24, 27, 28, 33, 43, 46, 56, 76, 136, 139, 145, 149, 150, 187, 193, 195, 209, 214]). This prior work mainly involves the design of new algorithms [1, 46, 56, 136, 149], fixing synchronization-related issues in specific systems [28, 33, 187, 193, 209], or analyzing specific aspects of synchronization [10, 24, 150]. As a result, the conclusions of existing work cannot be properly generalized to understand the root cause of synchronization problems in new platforms, workloads, or systems. In most cases, when software systems face scalability issues, it is not clear if these issues are due to the underlying hardware, to the synchronization algorithm itself, to its usage of specific atomic operations, to the application context, or to the workload.

Additionally, prior work has focused on traditional performance metrics, namely *throughput* and *latency*. These have been the main metrics for measuring the efficiency of computing systems for several decades. However, this state of affairs started changing in the past few years as *energy* has become a very important factor [16]. Reducing the power consumption of systems is considered crucial today [61, 92]. Synchronization is a very appealing target for saving energy, mainly because synchronization is a building block for software and typically translates to processes waiting for each other. Consequently, we need to analyze and understand the energy efficiency trade-offs in synchronization.

In this dissertation, we take an in-depth approach to analyzing synchronization both in terms of performance and energy efficiency, focusing on lock-based synchronization. In summary, we show that *scalability of synchronization is mainly a property of the underlying hardware. This means that synchronization directly hampers the cross-platform performance portability of concurrent software. Nevertheless, we can achieve portability without sacrificing performance, by creating design patterns and abstractions, which implicitly leverage hardware details without exposing them to software developers.*

In what follows, we first explain why and how hardware dictates the scalability of synchronization (Section 1.2) and then show how we can potentially hide these hardware intricacies to achieve portable scalability (Section 1.3).

1.2 Hardware Dictates the Scalability of Synchronization

Part II of this dissertation (Chapters 4 and 5) includes two analyses of synchronization: The first analysis focuses on the performance of synchronization on various multi-core processors, while the second analysis revolves around the energy efficiency of locks.

Chapter 4 presents an exhaustive study of synchronization. We span multiple layers, from hardware cache-coherence protocols up to high-level concurrent software. We do so on different types of architectures, from single-socket—uniform and non-uniform—to multi-socket—directory and broadcast-based—multi-cores. We draw a set of observations that, roughly speaking, imply that scalability of synchronization is mainly a property of the hardware. In brief, we show that non-uniformity of memory accesses is the main inhibitor of scalability. In particular, synchronizing across sockets of multi-socket processors is detrimental for scalability. Unfortunately, we also observe that trying to limit synchronization within a socket is not always feasible, again due to limitations imposed by the underlying hardware.

Chapter 5 presents the first study of the energy efficiency of lock-based synchronization on `x86` multi-cores. Intuitively, locks are a natural place for improving the energy efficiency of software. First, as we mentioned earlier, concurrent systems are mainstream and when their threads synchronize, they typically do it with locks. Second, locks are well-defined abstractions, hence changing the algorithm implementing them can be achieved without modifying the software system. Third, some locking strategies consume more power than others, thus the strategy choice can make a difference. Last but not least, as we show in Chapter 5, improving the energy efficiency of locks goes hand in hand with improving their throughput.

We make our case for this throughput/energy-efficiency correlation through a series of observations obtained from an exhaustive analysis of the energy efficiency of locks on two processors and six software systems. Essentially, these observations show that modern hardware does not provide adequate tools for reducing the power consumption of locks without destroying throughput. Naturally, without the ability to affect power consumption, we have to continue focusing on throughput optimizations in order to also improve energy efficiency.

Overall, both of these synchronization studies show that the underlying multi-core hardware largely dictates the scalability of synchronization that can be achieved by software.

1.3 Hiding Hardware Limitations to Achieve Portable Scalability

Multi-core hardware dictating the behavior of synchronization is very bad news for the portability of concurrent systems. Fine-tuning software for every single platform is intractable. In Part III of this dissertation (Chapters 6 and 7), we introduce two approaches that can lead to both portability and scalability of synchronization (and of concurrent software).

Chapter 6 introduces OPTIK, a new practical design pattern for designing and implementing fast and scalable concurrent data structures. OPTIK relies on the commonly-used technique of version numbers for detecting conflicting concurrent operations. We show how to implement the OPTIK pattern using the novel concept of OPTIK locks. These locks extend the traditional lock interface for efficiently implementing the OPTIK pattern. Existing state-of-the-art lock-based data structures acquire the lock and then check for conflicts (e.g., [97, 105]). In contrast, with OPTIK locks, we merge the lock acquisition with the detection of conflicting concurrency in a single atomic step, similarly to lock-free algorithms. We illustrate the power of our OPTIK pattern and its implementation by introducing five new algorithms and by optimizing four state-of-the-art algorithms for linked lists, skip lists, hash tables, and queues. Our results show that concurrent data structures built using OPTIK are more scalable than the state of the art.

Chapter 7 introduces `libmctop`, a library that relies on the determinism of cache-coherence protocols for inferring the basic topology of multi-cores using only latency measurements. `libmctop` then augments this basic representation with low-level information, such as memory bandwidths, to deliver the MCTOP topology abstraction. MCTOP enables developers to accurately and portably define high-level performance policies. For example, using MCTOP, we can easily define policies such as “use the two closest cores,” or “use two sockets with maximum communication bandwidth.” These MCTOP policies utilize low-level characteristics of multi-cores, such as latencies and bandwidth, without exposing them to the developer, resulting in portable software optimizations. We illustrate several such policies through four examples: (i) automatic backoff schemes for locks, (ii-iii) thread placement in OpenMP and the Metis MapReduce library, as well as (iv) a topology-aware mergesort algorithm. We illustrate the performance benefits and the portability of these policies across five processors from Intel, AMD, and Oracle, with minimal development effort.

Overall, these two approaches prove that we can achieve portable scalability of concurrent software. To this end, we need to create design patterns, abstractions, or other high-level constructs which encapsulate synchronization and other hardware characteristics without exposing low-level details to software developers.

1.4 Contributions

As the number of cores and the complexity of multi-core systems keeps increasing, designing and implementing scalable synchronization becomes ever more challenging. This dissertation offers (i) a better understanding of the behavior of synchronization on modern multi-cores, and (ii) two approaches to deliver portable scalability of synchronization (and of concurrent software). In detail, this dissertation makes the following intellectual contributions:

1. A clear understanding of how hardware limits and dictates the performance and energy efficiency of synchronization, with ramifications on both software and hardware design.
2. OPTIK: A design pattern for devising scalable concurrent data structures.
3. MCTOP: An abstraction of multi-core topologies and a policy-based approach towards portable optimizations.

Furthermore, this dissertation contributes in providing several novel algorithms and synchronization libraries (implementing both our new and existing state-of-the-art algorithms):

1. SSYNC: A cross-platform synchronization suite; SSYNC works on `x86`, `SPARC`, and Tilera processors. SSYNC is available at `http://lpd.epfl.ch/site/ssync`.
2. LOCKIN: A locking library with more than 10 state-of-the-art lock algorithm implementations. LOCKIN includes MUTEXEE, our novel variant of pthread mutex lock. LOCKIN is available at `http://lpd.epfl.ch/site/lockin`.
3. OPTIK: A concurrent data structure library. OPTIK includes the implementation of OPTIK locks (i.e., our extension of the traditional lock interface for efficiently implementing the OPTIK pattern) and the five new data structure algorithms we design with OPTIK. OPTIK is available at `http://lpd.epfl.ch/site/optik`.
4. CLHT: A very fast and scalable concurrent hash table. CLHT is a part of ASCYLIB and is available at `http://lpd.epfl.ch/site/ascylib`.
5. `libmctop`: A library for designing portable software based on our MCTOP multi-core abstraction. `libmctop` includes the implementation of MCTOP-ALG (i.e., an algorithm for inferring the topology of any multi-core solely based on cache-coherence measurements). `libmctop` will soon be available at `http://lpd.epfl.ch/site/mctop`.

1.5 Thesis Roadmap

The rest of the dissertation is organized in three parts as follows:

Part I
- Chapter 2 introduces some background regarding shared-memory synchronization and describes the platforms used for the experiments throughout this dissertation.
- Chapter 3 discusses existing related work.

Part II
- Chapter 4 includes an exhaustive analysis of synchronization on four multi-cores.
- Chapter 5 presents the first study of the energy-efficiency trade-offs of lock-based synchronization on modern `x86` multi-cores.

Part III
- Chapter 6 introduces OPTIK, a novel design pattern for designing and implementing scalable optimistic concurrent data structures.
- Chapter 7 presents `libmctop`, a novel library that enables portable optimizations of concurrent software by abstracting multi-core topologies.

- Chapter 8 concludes this dissertation and discusses potential avenues for future work.

Part I

Preliminaries

2 Background and Target Platforms

In this chapter, we provide the necessary background related to the main chapters of this thesis. We start by describing important characteristics of modern multi-cores that affect the design and implementation of concurrent software. We continue by introducing the basic concepts that are used in synchronization and in the design of concurrent data structures. We conclude this chapter by describing the eight platforms that we use in our experiments.

2.1 Characteristics of Modern Multi-Core Processors

From Single- to Multi-Core Processors. In the early 2000s, processor manufacturers suddenly switched from single- to *multi-core* processors for commodity hardware. This switch was motivated by technological reasons related to power consumption and heat dissipation of microprocessors [25]. Nowadays, embedded, desktop, and server processors are multi-cores. As the name suggests, a multi-core processor includes several CPU cores that can execute software threads in parallel.

This shift of commodity hardware from single- to multi-cores also triggered a transition towards the need for concurrent programming. In other words, on multi-cores, in order to improve performance of a software application, one needs to use concurrent programming to utilize the multiple cores of the underlying machine. As it was simply put in 2005 by Herb Sutter, "The free lunch is over [203]," meaning that developers must employ concurrency in their software in order to harvest the available hardware parallelism. On single-core processors, technology scaling used to ensure that approximately every two years processors would become almost twice as fast as the processors of the previous generation. Consequently, every single–threaded piece of software was also "becoming twice as fast" every two years—motivating the name *free lunch*.

Cache Coherence. Traditionally, even on single-core processors, the gap between CPU core performance and the memory latency has been increasing for the past decades [100]. This problem is mitigated by multiple levels of caches. On single-core processors, caching is "straightforward," given that there is a single core with multiple level of caches, hence the core can freely cache and update data in the caches.

In contrast, multi-core processors bring the need for *cache coherence* between the (private) caches of different cores. Cache coherence is the hardware protocol responsible for maintaining the consistency of data in caches. The cache-coherence protocol implements the two fundamental operations of an architecture: *load* (read) and *store* (write). In addition to these two operations, cache coherence typically also provides other, more sophisticated *atomic operations*: compare-and-swap, fetch-and-increment, atomic swap, etc. Atomic operations are essential for synchronization as many synchronization problems are impossible with just load and store operations [14].

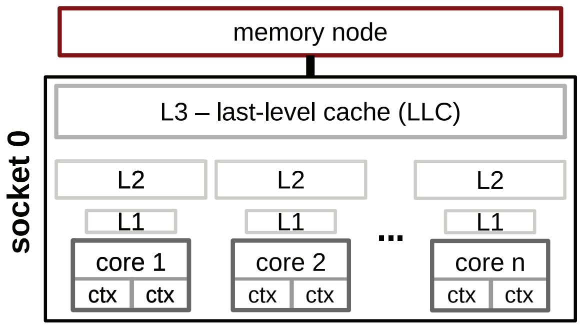


Figure 2.1 – Typical configuration of a single-socket multi-core processor.

Figure 2.1 depicts a typical configuration of one multi-core *socket* (i.e., one processor chip/die). For performance reasons, each core has two levels of private caches. Intuitively, if core 1 intends to modify a *cache line* of data[1] that is cached in the private caches of core 2, core 1 must inform core 2's caches about the update in order to avoid data inconsistencies. These situations are handled by the cache-coherence protocol of the processor.

Modern processors implement the *MESI* coherence protocol [174], or variants of MESI. With MESI, each cache line can be in one of the following four states:

- **M**odified: This is the only and the latest copy of this block of data in the cache hierarchy.[2] This block of data is stale in main memory.
- **E**xclusive: This is the only copy of data in the cache hierarchy. This block of data is valid in main memory.
- **S**hared: This is one of possibly many copies of this block of data (i.e., other cores might hold copies in their caches). This block of data is valid in main memory.
- **I**nvalid: This block of data holds invalid data that cannot be used by the core.

[1] A cache line, also known as a *cache block*, is the granularity of data transfer and coherence on modern processors. The size of a cache line is typically 64 bytes on modern multi-cores.

[2] Multiple copies of this modified cache line might exist in the caches of the same core (e.g., L1 and L2), depending if caches are inclusive or exclusive [100].

As we describe in Chapter 4, cache coherence can be viewed as synchronization at the hardware level and dictates how fast two threads can communicate in software.

Coherence is usually implemented by either *snooping* or using a *directory* [201]. With snooping, the individual caches monitor any traffic on the addresses they hold in order to ensure coherence. A directory keeps approximate or precise information of which caches hold copies of a memory location. Any operation that requires coherence has to consult the directory, enforce consistency, and update the directory.

Simultaneous Multi-Threading (SMT). The multiple cores of a multi-core can be used to take advantage of the *thread-level parallelism* of software—different threads can, in parallel, perform work on different cores. Still, many workloads include irregular memory accesses, resulting in poor utilization of the core's resources due to memory stalls.[3] In order to alleviate this problem, and at the same time deliver even higher thread-level parallelism, many modern processors (e.g., Intel `x86` and `SPARC`) include simultaneous multi-threading (SMT) [212]. With SMT, each CPU core contains more than one *hardware context*, which share many of the resources of the core, such as the caches and the pipelines. Typically, when a hardware context is stalled, another context is scheduled by the hardware.

Operating systems, such as Linux and Solaris, expose hardware contexts as the scheduling unit for software. A developer can *pin* software threads to specific hardware contexts, so that threads execute only on these hardware contexts. Note that although the OS essentially exposes hardware contexts as cores, executing two threads on the hardware contexts of the same or of different cores can make a huge performance difference due to the heavy resource sharing of hardware contexts of the same core. In this thesis, the term *thread* refers to software threads, while the term *hardware context* refers to either a hardware context of an SMT-enabled processor, or a core of a non-SMT processor.

Dynamic Voltage and Frequency Scaling (DVFS). DVFS is a commonly-used power-management technique that adjusts the frequency and the voltage of the CPU to save power. Most modern processors expose control registers to adjust the desired frequency and voltage points. For instance, Linux uses the `freqcpu` driver to adjust these control registers based on CPU utilization. When the driver detects that the processor utilization increases, the frequency and voltage of the CPU are increased accordingly. When the processor is mostly idle, the frequency and voltage are scaled down to save power. For a comprehensive explanation of how DVFS operates on modern multi-cores, we refer the reader to [213]. In Chapter 5, we consider DVFS for optimizing the energy efficiency of synchronization.

[3] Note that the same problem is attacked by *out-of-order execution* [100], which re-orders instructions of a single execution context in order to take advantage of *instruction-level parallelism* and hide long-latency instructions.

Non-Uniform Memory Access (NUMA). Modern server processors are typically packaged in *multi-socket* configurations. Multi-socket multi-cores, also known as *multi-processors*, consist of several multi-cores which are interconnected with fast network (e.g., Intel's QuickPath interconnect [112], or AMD's HyperTransport [45]). A typical 4-socket configuration is shown in Figure 2.2, where each socket is directly connected to all other sockets and to a *memory node*. Multi-socket servers typically provide fully-coherent globally shared memory (i.e., threads have coherent accesses to all memory nodes, local or not). The cross-socket interconnection network serves two main purposes: (i) transferring memory across sockets, and (ii) propagating cross-socket coherence messages.

Accessing memory from the local node of a socket is faster than accessing remote memory, thus multi-socket processors are said to offer *non-uniform memory accesses* (NUMA). As we show in Chapter 4, NUMA effects play an important role in the performance and scalability of synchronization. Additionally, in Chapter 7, we introduce a multi-core topology abstraction that can simplify programming on NUMA architectures.

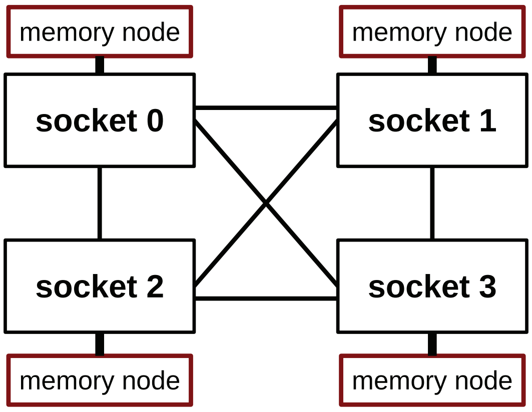


Figure 2.2 – Typical configuration of a multi-socket multi-core processor.

2.2 Synchronization

Synchronization denotes the act of coordinating the execution of a set of processes. In concurrent systems where processes share data, synchronization is necessary for correctness [14]. Essentially, synchronization does not contribute any useful work to concurrent software, hence, it can be considered as mere overhead. Therefore, as we show in this thesis, synchronization must be reduced to the bare minimum.

As we mentioned earlier, cache-coherence protocols essentially implement synchronization at the hardware level. Although these protocols are not directly exposed to software, as we illustrate throughout this thesis, they play a crucial role on the performance and scalability of concurrent software. Essentially, all software synchronization techniques directly build on top of this hardware synchronization (as they make use of loads, stores, and atomic operations). In particular, in Chapter 7, we show that the characteristics of cache-coherence protocols are so explicit that they can be used to infer the topology of any multi-core.

In software, synchronization can be implemented in various ways. In what follows, we describe the two most prominent synchronization techniques, which we consider in this dissertation.

Locking. *Locking* is by far the most commonly-used approach to synchronization. Practically all modern software system employ locks in their design and implementation. Prominent examples include operating systems (e.g., Linux [72, 73]), databases (e.g., MySQL [168]), and key-value stores (e.g., Memcached [70], RocksDB [64]).

The main reason behind the popularity of locking is that offers an intuitive abstraction. Locks ensure *mutual exclusion*; only the holder/owner of the lock can proceed with its execution. Executions that are protected by locks are known as *critical sections*. The remaining threads wait until the holder releases the lock. This waiting is implemented with either *sleeping* (*blocking*), or *busy waiting* (*spinning*) [172].

With sleeping, the thread is put in a per-lock wait queue and the hardware context is released to the OS. When the lock is released, the OS might wake up the thread (triggered by the release function). With busy waiting, threads remain active, polling the lock in a spin-wait loop. Sleeping is employed by the pthread mutex lock (MUTEX). On Linux, MUTEX builds on top of `futex` system calls, which allow a thread to wait for a value change on an address. MUTEX might first perform busy waiting for a limited amount of time and if the lock cannot be acquired, the thread makes the `futex` call.

The lock algorithms which employ busy waiting are called *spinlocks* [10, 103]. In simple spinlock algorithms (e.g., TAS, TTAS, TICKET) processes spin on a common memory location until they acquire the lock. Simple spinlocks are generally considered to scale poorly because they involve high contention on a single cache line [10], an issue which is addressed by queue-based spinlocks [46, 149]. Queue-based locks remove the single cache line bottleneck of simple spinlocks by generating a queue of nodes, so that each process spins on a unique memory location.

Spinlocks mostly differ in their busy-waiting implementation. For example, TAS spins with an atomic operation, continuously trying to acquire the lock (*global spinning*). In contrast, all other spinlocks (e.g., TTAS, TICKET, MCS, CLH, HCLH) spin with a load until the lock becomes free and only then try to acquire the lock with an atomic operation (*local spinning*). To acquire a *queue-based lock* (e.g., MCS, CLH), a thread adds an entry to a queue and spins until the previous holder hands over the lock. Hierarchical locks [56, 139] (e.g., HCLH, HTICKET) are tailored to today's NUMA architectures by using node-local data structures and minimizing accesses to remote data. Table 2.1 describes the lock algorithms that we consider in this thesis.

Name	Waiting	Short description
MUTEX [87]	sleeping	The standard pthread mutex lock algorithm. Threads (might) spin for a while before releasing their hardware context to the OS (if they do not manage to acquire the lock during this busy waiting period).
MUTEXEE [66]	sleeping	Our optimized MUTEX lock algorithm, presented in Section 5.5.1. MUTEXEE tries to avoid the frequent sleep invocations of MUTEX, using spinning both in lock and unlock functions.
TAS [10]	global spinning	Test-and-set lock. Threads busy wait on the lock with atomic operations (e.g., compare-and-swap) until they manage to acquire the lock.
TTAS [10]	local spinning	Test-and-test-and-set lock. Threads busy wait by polling (loading) the lock value until the lock becomes free. Once the lock is free, threads perform atomic operations to try to acquire it. If unsuccessful, they return again to local spinning.
TICKET [149]	local spinning	Contains two counters: *ticket* and *current*. To acquire the lock, threads atomically fetch and increase the ticket counter. If the acquired ticket equals the current counter, the thread has acquired the lock, otherwise, the thread spins until its ticket becomes equal to counter.
ARRAY [103]	queue based	Includes an array of flags. To acquire the lock, threads atomically fetch and increment a counter in order to find which slots in the array to use. Only one flag can be free at a time. To release the lock, the lock owner transfers the free flag to the next slot.
MCS [149]	queue based	Threads create a linked queue of lock requests by appending their local node with an atomic swap to the tail of the lock. Threads spin on their local queue nodes and release the lock by following the successor node of their local node.
CLH [46, 140]	queue based	CLH is similar to MCS. However, threads "inherit" and spin on their predecessor node, not on their own local node. Essentially, the queue with CLH is implicit (every thread knows of just two nodes).
HCLH [139]	hierarchical queue	A hierarchical version of CLH. Threads create implicit per-NUMA-node queues and splice them into a global queue with a compare-and-swap.
HTICKET [49, 56]	hierarchical local	A hierarchical version of TICKET. When a thread acquires the top-level lock (across nodes), it fetches several tickets. These tickets are used for transferring the lock among threads of the same node. When no local thread intends to acquire the lock or all tickets are consumed, the lock is released to other nodes.

Table 2.1 – Short description of various lock algorithms.

Message Passing. An alternative to locks that we consider in this thesis is to partition the system resources between processes. In this view, synchronization is achieved through *message passing*, which is either provided by the hardware or implemented in software [13]. Software implementations are generally built over cache-coherence protocols and impose a single-writer and a single-reader for the used cache lines. The main goal of using message passing on a shared-memory multi-core system is to achieve explicit inter-process communication, which is easier to control and debug than implicit communication over shared memory. Additionally, message-passing-based software systems can be easily ported and deployed as distributed systems and can thus easily leverage new technologies such as RDMA.

In principle, synchronization on top of shared memory or through message passing are duals of each other—i.e., concurrent software built with shared memory has a counterpart with message passing, and vice versa [13, 126]. Still, as we show in Chapter 4, modern multi-cores favor shared memory unless there is extremely high contention.

2.3 Concurrent Data Structures

Data structures allow for efficient storage and retrieval of data elements. These elements are typically identified by unique keys. In particular, *search data structures* (e.g., lists, hash tables) include three main operations: (i) *search*, for searching for an element with a given key, (ii) *insert*, for inserting a new element in the structure if the key is not already there, and (iii) *delete*, for deleting an existing element. Other data structures, such as queues, offer a different interface. Queues are first-in first-out (FIFO) structures with two main operations: (i) *enqueue*, to place an element at the head of the queue, and (ii) *dequeue*, to remove the current tail element (if any).

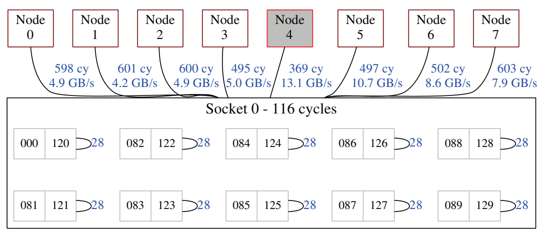
Concurrent data structures (CDSs) can be simultaneously accessed by multiple threads through their interface. The consistency of CDSs is typically measured with respect to linearizability [104]. Linearizable CDS algorithms are commonly classified based on the progress guarantees they offer. It is common to distinguish between *blocking* [103], *lock-free*, and *wait-free* [101] algorithms. Blocking algorithms typically rely on *locking* and might block, waiting for a lock to be released. Lock-free algorithms are non-blocking in the sense that (i) they do not use locks, and (ii) they ensure that at least one process in a system can make progress. Finally, wait-free algorithms, in addition to being non-blocking, guarantee that every process in a system eventually makes progress. In practice, wait-free algorithms are slower than their lock-based and lock-free counterparts and are thus not very commonly used [48]. In Chapter 6, we introduce several novel CDS algorithms, including lock-free and lock-based designs.

Most state-of-the-art CDS algorithms are *optimistic*, regardless if they are lock-based or lock-free. They are optimistic in the sense that they first optimistically perform some work, without synchronizing with other threads, and then synchronize to validate the consistency of the optimistic work and to modify the state of the structure. Lock-based algorithms perform validation in critical sections (i.e., in the execution parts which are protected by locks), while lock-free algorithms use atomic operations, such as compare-and-swap, to simultaneously validate and update the target nodes of the data structure.

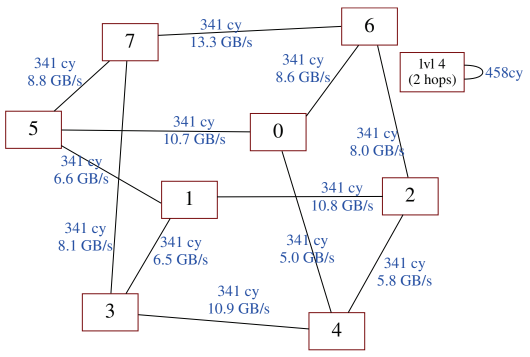
2.4 Target Platforms

In what follows, we describe the eight multi-core processors that we employ in collecting the experimental results of this thesis. Each chapter utilizes a different subset of these multi-cores, depending on the requirements of the chapter (i.e., what the chapter aims to illustrate). The topology representations that we depict in the figures of this chapter are automatically generated by `libmctop`, a tool that we introduce in Chapter 7 (we do not include graphs for the single-socket processors).

Opteron. The 48-core AMD Opteron (we use *Opteron* as an example for `libmctop` in Chapter 7–Figure 7.1) contains four Opteron 6172 multi-chip modules (MCMs). Each MCM has two 6-core dies, for a total of 8 memory nodes. It operates at 2.1 GHz and has 64 KB, 512 KB, and 5 MB (per die) L1, L2, and LLC data caches, respectively.



(a) Intra-socket topology of a socket.



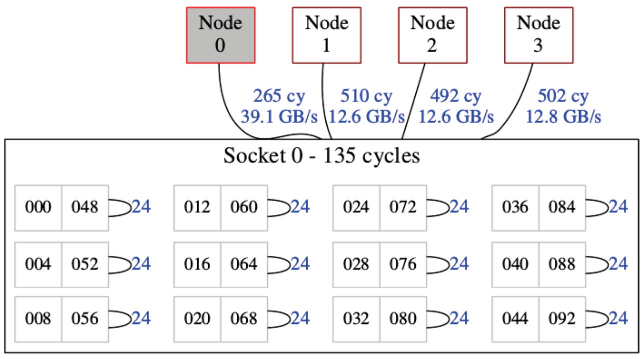
(b) Cross-socket topology.

Figure 2.3 – Topology representation of an 8-socket Intel Xeon processor—*Westmere*.

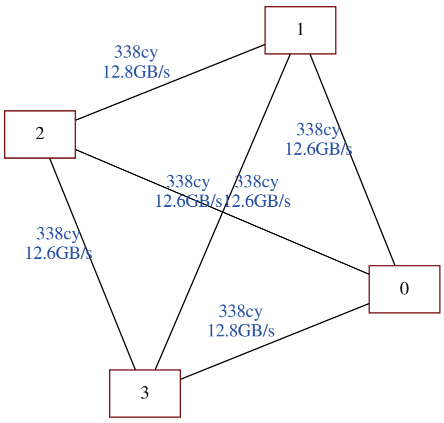
Westmere. The 80-core Intel Xeon (Figure 2.3) consists of eight sockets of Intel Xeon Westmere E7-8867L 10-cores (two hardware contexts per core). Westmere operates at 1.1-2.1 GHz and has 32 KB, 256 KB, and 30 MB (per die) L1, L2, and LLC data caches, respectively.

Haswell. The 48-core (two hardware contexts core) Intel Xeon (Figure 2.4) comprises four Haswell E7-4830 v3 sockets. Haswell operates at 1.2-2.7 GHz and has 32 KB, 256 KB, and 30 MB (per die) L1, L2, and LLC data caches, respectively.

Ivy. The 20-core Intel Xeon (we use *Ivy* as an example for `libmctop` in Chapter 7–Figure 7.4) consists of two sockets of Ivy Bridge E5-2680 v2 10-core (20 hardware contexts). *Ivy* runs at 1.2-2.8 GHz and includes 32 KB, 256 KB, and 25 MB (per die) L1, L2, and LLC, respectively.



(a) Intra-socket topology of a socket.



(b) Cross-socket topology.

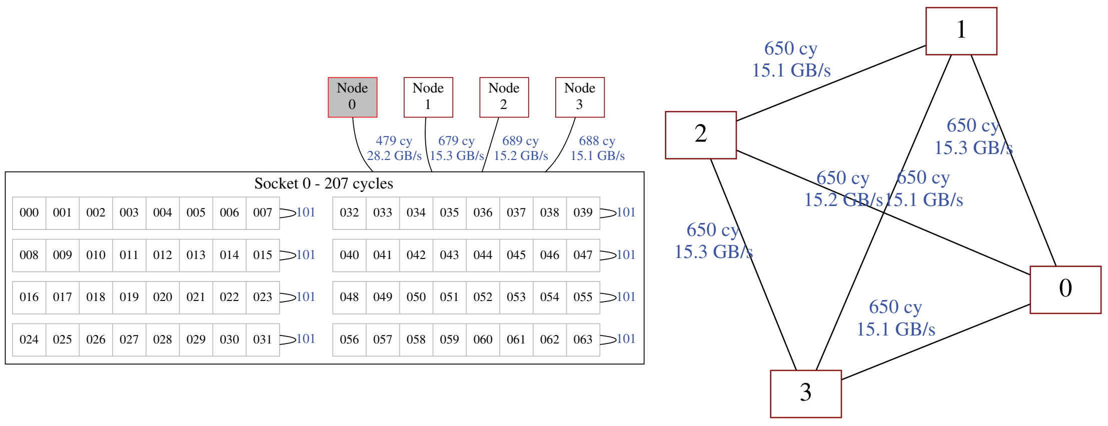
Figure 2.4 – Topology representation of a 4-socket Intel Xeon processor—*Haswell*.

Ivy-desktop The 4-core Intel Core i7 is a desktop Ivy Bridge 3770K processor (8 hardware contexts). *Ivy-desktop* runs at 1.6-3.5 GHz and includes 32 KB, 256 KB, and 8 MB (per die) L1, L2, and LLC, respectively.

SPARC-T2. The Sun Niagara 2 is a single-die processor (SUN UltraSPARC-T2) that incorporates 8 cores clocked at 1.2 GHz. It is based on the chip multi-threading architecture; it provides 8 hardware contexts per core, totaling 64 hardware threads. Each L1 cache (8 KB) is shared among the 8 hardware contexts of a core, while the shared LLC is 4 MB.

SPARC-T44 The Oracle SPARC T4-4 (Figure 2.5) consists of four SPARC T4 sockets with eight cores per socket and a total of 256 hardware contexts (eight hardware contexts per core). *SPARC-T44* operates at 3 GHz and has 16 KB, 256 KB, and 4 MB (per die) L1, L2, and LLC data caches, respectively.

Tilera. The Tilera TILE-Gx36 [205] is a 36-core chip multi-processor. It clocks at 1.2 GHz and has 32 KB, 256 KB, and 9 MB 3 L1, L2, and L3 data caches, respectively.



(a) Intra-socket topology of a socket.

(b) Cross-socket topology.

Figure 2.5 – Topology representation of a 4-socket Oracle processor—*SPARC-T44*.

3 Related Work

In this chapter, we discuss the work that is the most related to the topics covered by this dissertation. We focus on synchronization based on locks and message passing (Section 3.1). Still, we describe alternative synchronization techniques for optimistic concurrency in the context of concurrent data structures (Section 3.2). Finally, we describe the implications of synchronization on the scalability of software systems (Section 3.3).

3.1 Scaling Synchronization on Multi-Cores

Leveraging Cache Coherence. Cache-coherence protocols guarantee the consistency of data across the multiple caches in multi-cores and are thus also responsible for transferring data within the memory hierarchy. Given that today's multi-cores provide communication latencies with large non-uniformity, analyzing and leveraging cache coherence is essential for the scalability of synchronization primitives. The characteristics of cache-coherence protocols on x86 multi-sockets have been investigated by a number of studies [90, 156], whose focus is on bandwidth limitations. The cache-coherence latencies are measured from the point of view of loading data. We extend these studies by also measuring stores and atomic operations—which are essential for most synchronization constructs [14]—and analyzing the impact on higher-level concurrent software.

Molka et al. [156] consider the effect of the AMD and Intel memory hierarchy characteristics on various workloads from SPEC OMP2001 and conclude that throughput is mainly dictated by memory limitations. The results are thus of limited relevance for systems involving high contention. Moses et al. [160] use simulations to show that increasing non-uniformity entails a decrease in the performance of the TTAS lock under high contention. However, the conclusions are limited to spinlocks and one specific hardware model. We generalize and quantify such observations on commonly used architectures and synchronization schemes, while also analyzing their implications.

Scaling Lock-Based Synchronization. Lock-based synchronization has been thoroughly analyzed in the past. Many studies [3, 10, 117, 136, 149] point out scalability problems due to excessive coherence traffic with traditional spinlocks and propose alternatives. Note that some of these studies pre-date multi-core processors with MESI cache coherence. Instead, they address concerns regarding caching of lines in shared mode at multiple processors. In more detail, Agarwal and Cherian [3] propose various adaptive backoff mechanisms for reducing coherence traffic of spinlocks. In Chapter 7 we design “educated” lock backoff mechanisms that build on top of the cache-coherence latencies of multi-cores. Mellor-Crumney et al. [149] and Anderson [10] introduce and test several alternatives to simple spinlocks, such as queue locks. Their evaluation is performed on large scale multi-processors, on which the memory latency distribution is significantly different than on today’s multi-cores. Throughout this thesis, we thoroughly evaluate queue-based spinlocks, such as MCS [149] and CLH [46, 140], on modern multi-core processors.

Goodman et al. [81] propose various architectural primitives for implementing synchronization, based on the idea of synchronization bits that offer mutual exclusion. Kägi et al. [117] build on the idea of synchronization bits and propose the design of a hardware-based queue-based lock for minimizing coherence traffic. Naturally, these techniques require hardware modifications and are not available on commercial processors.

Lim and Agarwal [130] propose reactive locks, an adaptive synchronization scheme that switches between different protocols and waiting strategies. The idea of reactive locks is 100% aligned to the findings of this thesis, however, (i) the authors report modest performance benefits, and (ii) reactive locks must be implemented and evaluated on modern multi-cores.

More recently, Radovic and Hagersten [184, 185] were the first to propose hierarchical locks, tailored for NUMA architectures. Hierarchical locks trade short-term fairness for performance, by letting threads of a socket exchange the lock within the socket (for some fixed number of lock handovers) before releasing the lock to threads of other sockets. Based on this idea, many other hierarchical locks have been designed. Luchangco et. al [139] study a NUMA-aware hierarchical CLH lock (HCLH) and compare its performance with a number of well-known locks. Dice et al. [56] propose lock cohorting, a generic technique for designing hierarchical locks. Similarly, Chabbi et al. [38] introduce a hierarchical MCS lock that supports deep hierarchies. Our analysis in Chapter 4 shows that hierarchical locks can be beneficial in the presence of (i) large non-uniformity and (ii) under very-high lock contention, so that the lock can indeed remain busy within a single socket. Additionally, in Chapter 5, we show how reducing fairness can result in significant energy efficiency benefits.

Inspired by our analysis of synchronization in Chapter 4, Guiroux et al. [89] performed an extensive analysis of 27 lock algorithms on `x86` multi-cores on applications from PARSEC, Phoenix, and SPLASH2 suites. To be able to easily modify lock algorithms in theses applications, they develop an interpolation library that builds on our CLHT hash table (Section 6.2).

Their results corroborate our findings that “every lock algorithm has its fifteen minutes of fame” and that the underlying hardware platform plays a big role on the performance of locking.

Other lock-related studies focus on the Linux kernel [29] and conclude that the default ticket lock implementation causes important performance bottlenecks in the OS on a multi-core. Performance is improved in a number of different scenarios by replacing the ticket locks with complex locks. We confirm that plain spinlocks do not scale across sockets and present some optimizations that alleviate the issue.

Similarly to our MUTEXEE lock (Section 5.5.1), Solaris’ mutex locks offer the option of “adaptive unlock,” where the lock owner does not wake up any threads if the lock can be handed over in user space [154]. Moreshet et al. [158] share some preliminary results suggesting that transactional memory can be more energy efficient than locks. Wamhoff et al. [213] evaluate the overheads of using DVFS in locks and show how to improve performance by boosting the lock owner. Our work extends prior synchronization work with a complete study of the energy efficiency of lock-based synchronization.

Spin-Then-Sleep Trade-Off in Locks. The spin-then-sleep strategy was first proposed by Ousterhout [172] in order to avoid wasteful context switches in case processes are likely to wait for only a short amount of time. Various studies [24, 118, 129] analyze this trade-off and show that just spinning or sleeping is typically suboptimal. Franke et al. [74] recognize the need for fast user-space locking and describe the first implementation of `futex` in Linux. Our MUTEXEE lock (Section 5.5.1) uses the `futex` system calls for putting threads to sleep.

Johnson et al. [116] advocate for decoupling the lock-contention strategy from thread scheduling. At first glance, our MUTEXEE lock might look similar to their load-control lock (LC). LC builds on top of TP-MCS [96], an MCS lock with support for timeouts, allowing threads to be dequeued from the lock-waiting queue. However, the LC and MUTEXEE have some notable differences. LC relies on a global view of the system for load control (threads are put to sleep when the system load is high), while MUTEXEE performs per-lock load control. LC’s global load control can result in “unlucky” locks having their few waiting threads sleep for at least 100 ms, although there is low lock contention—sleeping threads are not woken up by a lock release, but only because of a decrease in load or 100 ms timeout (we could say that LC is unfair with 100 ms “granularity”). Finally, in contrast to MUTEXEE, LC might waste energy, because on low system load, no thread is blocked, even if the waiting times are hundreds of ms.

Scaling Message-Passing-Based Synchronization. As we mentioned earlier, synchronization on top of shared memory or through message passing are in principle duals of each other [13, 126]. A number of efforts (e.g., Barrelfish [17], fos [214]) point out the difficulty of scaling traditional shared-memory operating systems on multi-cores. This difficulty stems from the fact that, generation after generation, multi-cores tend to offer a larger number of cores and more heterogeneous resources. To achieve scalability, these systems avoid resource sharing altogether by using message passing (often implemented on top of shared memory).

In our prior work [84], we introduced TM^{2}C, the first software transactional memory for non-coherent multi-core processors. TM^{2}C builds on top of a distributed lock service and offers starvation-free transactions. Analyzing the portability of TM^{2}C on various multi-core processors gave us the inspiration for the synchronization analysis of Chapter 4.

Various techniques have been proposed in order to improve the performance of highly-contended locks, especially on multi-socket processors. For example, flat combining [99] is an approach in which a thread can execute critical sections on behalf of others. With flat combining, an operation translates to a message to a dedicated server thread that performs the operation on locally-held data without employing any synchronization. The immediate benefits are that (i) the server threads perform unsynchronized accesses to (likely) locally cached data, and (ii) requests are serialized by message passing, hence avoiding the single contention point of memory contention with locks.

Fatourou and Kallimanis [68] propose three optimized implementations of flat combining with the goal of minimizing coherence traffic to improve performance. The main idea of their solution is to piggyback thread requests to the server in the queue node of an MCS lock. That way, executing an operation boils down to "acquiring a lock." RCL [136, 137] replaces the "lock, execute, and unlock" pattern with remote procedure calls to a dedicated server core. For highly-contended critical sections this approach hides the contention behind messages and enables the server to locally access the protected data.

Similarly, Petrovic et al. [177] devise server-based and combining algorithms tailored for hardware message passing available on platforms such as the *Tilera* (see Section 2.1). They use these two synchronization approaches to design queue and stack algorithms and show that for these highly-contended data structures their solutions significantly outperform their shared memory counterparts. Overall, the scope of flat-combining solutions is limited to high contention and a large number of cores.

Our results regarding message-passing-based synchronization in Chapter 4 corroborate prior work and confirm that message passing can provide significant scalability benefits over locking under very high contention. Nevertheless, we show that when contention is low, message passing is significantly slower than solutions that build directly on top of shared memory.

3.2 Scaling Concurrent Data Structures on Multi-Cores

This section mainly discusses the work that is related to Chapter 6, our OPTIK design pattern for concurrent data structures. Essentially, variants of the OPTIK pattern can be found wherever optimistic concurrency is used (e.g., databases, distributed systems). In the following, we highlight work that is the most related to OPTIK.

Designing Concurrent Data Structures (CDSs). There has been a large amount of work on designing efficient and scalable CDSs, for linked lists [93, 97, 151, 183], hash tables [127, 151], skip lists [75, 183, 202], binary search trees [30, 57, 60, 161], queues [153, 159, 177, 210], and stacks [98, 177, 206]. Every new CDS design typically introduces a new technique for detecting and handling concurrency. In Chapter 6, we introduce OPTIK, a generic design pattern that provides a way of detecting conflicting concurrency via version numbers in different CDSs.

Gramoli et al. [85] utilize version numbers to design a concurrent linked list which reduces synchronization over the lazy linked list [97] and is similar to our OPTIK-based fine-grained linked list. Our BST-TK binary-search-tree algorithm is the first occurrence of the OPTIK pattern. In this thesis, we further generalize the usage of version numbers to a design pattern and show how to use it in various CDSs.

OPTIK is largely inspired by our "asynchronized concurrency" (ASCY) paradigm [51]. ASCY consists of four complementary programming patterns that call for the design of concurrent search data structures to resemble that of their sequential counterparts. ASCY is a result of an extensive performance analysis of several algorithms on four multi-core processors. Similarly to our ASCY work, Gramoli [83] analyzed lock-free, lock-based, and transaction-based concurrent data structure algorithms on multi-cores. One of his main conclusions is that lock-free designs are more scalable than lock-based ones. Our results with ASCY and OPTIK clearly show that lock-based algorithms are as scalable as their lock-free counterparts (and are usually faster).

Alistarh et al. [6] recently proved that, even in the presence of high contention for some memory locations, lock-free concurrent algorithms are wait-free under stochastic scheduling of shared memory accesses. Similarly, David and Guerraoui [48] empirically showed that lock-based concurrent search data structures, such as lists, practically behave as wait-free algorithms. Essentially, they show that lock-related issues (e.g., priority inversions, lock convoying) do not manifest in search data structures, thus the simplicity and high performance of lock-based designs should be preferred over wait-free algorithms. Our experience with OPTIK corroborates their findings and proves that we can easily and efficiently design concurrent data structures.

Optimistic Concurrency Techniques. Several concepts and tools have been proposed for designing and implementing optimistic concurrency.

Read-copy update (RCU) [146] is a technique that was introduced in the Linux kernel for easily designing CDSs with (i) wait-free reads and (ii) memory reclamation. Nevertheless, RCU targets read-mostly workloads. Relativistic programming [208] extends RCU to support infrequent, but expensive operations, such as hash-table resize. Arbel and Attiya [11] extend RCU to better support concurrent updates. Still, their binary-search-tree design is slower than other state-of-the-art trees, especially on write-intensive workloads. Predicate RCU (PRCU) [12] reduces the granularity of waiting in RCU. PRCU offers a trade-off between the amount of work that search operations must do and the amount of waiting in updates. RLU [145] improves the usability of RCU by offering concurrency of reads with multiple writers. With OPTIK, we decouple memory reclamation from concurrency control, thus we are able to achieve designs that incur none of the aforementioned overheads of RCU.

Transactional memory offers the concept of transactions for implementing synchronization. Software transactional memory (STM) [197] implements transactions in software. STM can be used in the design of CDSs, but due to the instrumentation overheads of STMs, the resulting implementations are typically slower than their lock-free or lock-based counterparts [36]. Hardware transactional memory (HTM) [102] implements transactions in hardware and thus avoids the instrumentation and the metadata overhead of STMs. Unfortunately, HTMs are currently neither ubiquitous nor robust enough to be extensively used by CDS designers.

Speculative lock elision [186, 190] aims at reducing the overhead of locking when concurrent critical sections do not actually conflict. A thread might elide a lock, meaning that threads optimistically execute their critical sections without acquiring that lock. If a true data conflict appears, then the thread rolls back and executes the critical section normally. The main goal of lock elision is to enable writing concurrent applications with coarse-grained locking that perform well. In contrast, OPTIK's main goal is to enable the design of high-performance CDSs in a methodical way. As we discussed earlier, flat combining [99] is another technique that appears promising for optimizing coarse-grained lock-based CDSs (e.g., queues). Unlike OPTIK, flat combining is not suitable for highly-concurrent data structures, such as hash tables.

Sequence locks (seqlocks) [125] resemble OPTIK locks as they include a lock and a version number. With seqlocks, readers ensure that they read consistent data by double checking the version number. However, unlike OPTIK, seqlocks assume distinct readers/writers and keep the lock and the version separately. In fact, OPTIK locks can be used in implementing the seqlock functionality.

Version Numbers in Concurrency. Optimistic concurrency control was introduced for optimizing database transactions [123] in 1981. It relied on transaction numbers for detecting conflicting concurrency. In concurrent programming, many STM systems (e.g, TL2 [55], TinySTM [69], NOrec [47], SpecTM [58]) rely on version numbers for validating the optimistic results of transactions. Version numbers have also been employed in distributed transactions (e.g., [4, 59]) for detecting conflicts. To the best of our knowledge, we are the first to extend the traditional lock interface, with OPTIK locks, so that we merge validation with locking.

3.3 Scaling Software Systems on Multi-Cores

Scaling System Performance. In order to improve OS scalability on multi-cores, a number of approaches deviate from traditional kernel designs. The OS is typically restructured to either improve locality (e.g., Tornado [76]), limit sharing (e.g., Corey [27]), or avoid resource sharing altogether by using message passing (e.g., Barrelfish [17], fos [214]). Boyd-Wickizer et al. [28] aim at verifying whether these scalability issues are indeed inherent to the Linux kernel design. The authors show how optimizing, using various concurrent programming techniques, removes several scalability issues from both the kernel and the applications. By doing so, they conclude that it is not necessary to give up the traditional kernel structure just yet. Lozi et al. [138] analyze the behavior of the Linux scheduler and show that certain bugs can result in significant performance degradation of synchronization-heavy applications.

Clements et al. [43] link commutative interfaces to the existence of scalable implementations. In essence, they argue that commutative operations can lead to cache conflict-free implementations, that are inherently scalable from the memory-system point of view. This conclusion can help developers avoid unnecessary synchronization that can hinder system scalability.

Numerous key-value stores, such as Memcached [70], RocksDB [64], LevelDB [82], SILT [131] or Masstree [143] are based on concurrent data structures. In some cases, these structures have been shown to be scalability bottlenecks, as for example in Memcached [28, 67, 163]. Fan et al. [67] achieve a 3-fold performance increase over the traditional Memcached, mainly by optimizing its hash table's synchronization. Similarly, Lim et al. [132] get significant scalability improvements on key-value stores, largely due to synchronization optimizations. Golan-Gueta et al. [80] propose an architecture for log-structured data stores with optimized synchronization. They redesign LevelDB and show up to 2.5x higher throughput than state-of-the-art log-structured systems.

Our results confirm these papers' observation that synchronization can be an important bottleneck. We go a step further: We study the roots of scalability problem and observe that many of the priorly reported issues are in fact hardware-related and would manifest differently on different platforms.

Scaling System Energy Efficiency. There are many hardware techniques for reducing the energy footprint of systems. Hardware techniques for reducing energy consumption include clock gating [128], power gating [179], as well as voltage and frequency scaling [71, 189].

Additionally, there is a body of work that points out the importance of energy-efficient software. For instance, Linux has rules to manage frequency and voltage settings [173]. Further work proposes OS facilities for managing and estimating power [188, 198, 199, 219, 221]. Other frameworks approximate loops and functions to reduce energy [15, 62, 192]. Moreover, compiler-based [216, 218] and decoupled access-execute DVFS [122] frameworks trade off performance for energy. In servers, consolidation [19, 39] collocates workloads on a subset of servers, and fast transitioning between active-to-idle power states allows for low idle power [147, 148]. Basically, those techniques require changes in hardware, installing new schedulers or runtime systems, or even rebuilding the entire system.

Psaroudakis et al. [181] achieve up to 4x energy-efficiency improvements in database analytical workloads, using hardware models for power-aware scheduling. Similarly, Tsirogiannis et al. [211] analyze a DB system and conclude that the most energy-efficient point is also the best performing one. Our POLY conjecture (Chapter 5) is a similar result for locks. Nevertheless, while they evaluate various DB configurations, we study the spin vs. sleep trade-off. To the best of our knowledge, this is the first work to consider the energy trade-offs of lock-based synchronization on modern multi-cores.

Optimizing Systems for Multi-Cores. As corroborated by a large amount of work in operating systems [17, 18, 18, 27, 28, 76, 176, 220], databases [77, 115, 178, 182, 222], programming languages [78, 165], parallel runtimes [2, 23, 94, 141], key-value stores [20, 132], and synchronization [34, 37, 38, 56], system developers need to optimize software for the target platform to achieve good performance. Below, we discuss selected examples of multi-core optimizations.

Giceva et al. [77] explore the efficient deployment of database query plans on multi-cores for improving database performance. Psaroudakis et al. [182] describe how data placement and access patterns can significantly affect performance in databases. In the same vein, Gidra et al. [78, 79] improve the performance of Java's garbage collection by mainly optimizing memory placement and removing memory bottlenecks.

In synchronization, optimizing for the underlying platform is a one-way road (as we clearly show in Chapter 4). To this end, the various hierarchical lock algorithms and techniques [37, 38, 56] that we discussed earlier are designed to be topology aware.

`libmctop` (Chapter 7) is built based on the realization that multi-core optimizations are necessary for good system performance. With `libmctop`, we offer the MCTOP topology abstraction that enables portable optimizations.

Tools for Multi-Cores. Developing concurrent software is an onerous task. To simplify programming, software developers rely on tools and libraries that offer thread pinning, memory placement, etc. Accordingly, libraries with similar functionality to our `libmctop` (Chapter 7) already exist. The most prominent ones are `libnuma` [120], `liblgrp` [167] and `hwloc` [31]. Similarly to `libmctop`, all three provide some form of topology abstraction, as well as APIs for thread and memory placement. In contrast to `libmctop`, all three libraries rely on the OS for the topology of the machine (which, as we show, can lead to inaccuracies). They also lack the low-level measurements that the enriched MCTOP abstraction offers.

Additionally, `libnuma` and `liblgrp` offer relative "distances" between resources. These depend on the OS and can be very inaccurate, as we notice during our experiments. Both `libnuma` and `liblgrp` are also OS-specific (`libnuma` works on Linux, while `liblgrp` on Solaris). `hwloc` is portable across platforms (i.e., it can load the topology from various different operating systems), but is also missing the detailed latency and bandwidth measurements of MCTOP, which, as we show, are crucial to optimizing software. `hwloc` also offers an API that can be used across platforms. Unfortunately, it focuses mainly on locality and the available cache hierarchies of the platforms. In contrast, with MCTOP, we have both the portable abstraction of the topology, as well as the enriched measurements which can be used either directly or indirectly to optimize software across platforms.

LIKWID [207] is a set of command-line tools that visualize the thread and cache topology of a multi-core, as well as control the thread affinities of an application. LIKWID relies on the operating system for its topology (currently it supports only Linux) and focuses mainly on performance counter measurements.

There also exist tools for collecting measurements similar to the ones in MCTOP. These include Intel's performance counter monitor [215], which can be used to measure the memory bandwidth from the memory controllers on an Intel platform.

As we show in Chapter 7, `libmctop` contains all the necessary components (i.e., topology and low-level measurement abstractions) to achieve portable optimizations on multi-cores.

Part II

Analyzing Synchronization

As the number of cores and the complexity of multi-core systems keeps increasing, designing, implementing, debugging, and optimizing synchronization becomes more and more challenging. Additionally, emerging performance metrics, such as energy consumption, contribute further to this difficulty. Today, software developers must not only optimize their systems for throughput and latency, but also for energy efficiency. In this part, we analyze synchronization in terms of throughput, latency, and energy efficiency, in order to assist developers with designing and implementing scalable synchronization on modern multi-cores.

4 A Performance Analysis of Synchronization on Multi-Cores[1]

In this chapter, we present an exhaustive study of synchronization in terms of traditional performance metrics, such as throughput and latency. Our goal with this study is to offer a better understanding of how synchronization behaves on modern multi-core processors. We span multiple layers, from hardware cache-coherence protocols up to high-level concurrent software. We do so on different types of architectures, from single-socket—uniform and non-uniform—to multi-socket—directory and broadcast-based—multi-cores. We draw a set of observations which, roughly speaking, imply that scalability of synchronization is mainly a property of the hardware.

4.1 Introduction

Scaling software systems to multi-core architectures is one of the most important challenges in computing today. A major impediment to scalability is synchronization. As we discussed in Chapter 3, in the past few decades, a large body of work has been devoted to the design, implementation, evaluation, and application of synchronization schemes. Yet, the designer of a concurrent system still has little indication, a priori, of whether a given synchronization scheme will scale on a given modern multi-core architecture and, a posteriori, about exactly why a given scheme did, or did not, scale.

One of the reasons for this state of affairs is that there are no results on modern architectures connecting the low-level details of the underlying hardware (e.g., the cache-coherence protocol) with synchronization at the software level. Recent work that evaluated the scalability of synchronization on modern hardware (e.g., [28, 136]) was typically put in a specific application and architecture context, making the evaluation hard to generalize. In most cases, when scalability issues are faced, it is not clear if they are due to the underlying hardware, to the synchronization algorithm itself, to its usage of specific atomic operations, to the application context, or to the workload.

[1] Appeared in: Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. "*Everything you always wanted to know about synchronization but were afraid to ask*." SOSP 2013.

Of course, getting the complete picture of how synchronization schemes behave, in every single context, is very difficult. Nevertheless, in an attempt to shed some light on such a picture, we present the most exhaustive study of synchronization on multi-cores to date. Our analysis seeks completeness in two directions (Figure 4.1).

depth	breadth
concurrent software hash table, Memcached	**single-socket** uniform (Sun Niagara 2) non-uniform (Tilera TILE-Gx36)
primitives locks, message passing	
atomic operations CAS, FAI, TAS, SWAP	**multi-socket** directory-based (AMD Opteron) broadcast-based (Intel Xeon)
cache coherence loads, stores	

Figure 4.1 – Methodology of our performance analysis of synchronization.

1. We consider multiple synchronization layers, from basic multi-core hardware up to complex concurrent software. First, we dissect the latencies of cache-coherence protocols. Then, we study the performance of various atomic operations, such as compare-and-swap, test-and-set, fetch-and-increment. Next, we proceed with locking and message passing techniques. Finally, we examine a concurrent hash table and an in-memory key-value store (Memcached).
2. We vary a set of important architectural attributes to better understand their effect on synchronization. We explore both single-socket (chip multi-processor) and multi-socket (multi-processor) multi-cores. In the former category, we consider uniform (e.g., Sun Niagara 2—*SPARC-T2*) and non-uniform (e.g, Tilera TILE-Gx36—*Tilera*) designs. In the latter category, we consider platforms that implement coherence based on a directory (e.g., AMD Opteron—*Opteron*) or broadcast (e.g., Intel Xeon—*Westmere*).

We focus our analysis on traditional performance metrics—i.e., throughput and latency. Our set of experiments, of what we believe constitute the most commonly–used synchronization schemes and hardware architectures today, induces the following set of observations.

Crossing sockets is a killer. The latency of performing any operation on a cache line (e.g., a store or a compare-and-swap) simply does not scale across sockets. Our results indicate an increase from 2 to 7.5 times compared to intra-socket latencies, even under no contention. These differences amplify with contention at all synchronization layers and suggest that cross-socket sharing should be avoided.

Sharing within a socket is necessary but not sufficient. If threads are not explicitly placed on the same socket, the operating system might try to load balance them across sockets, inducing expensive communication. However, surprisingly, even with explicit placement within the same socket, an incomplete cache directory, combined with a non-inclusive last-level cache (LLC), might still induce cross-socket communication. On *Opteron* for instance, this phenomenon entails a 3-fold increase compared to the actual intra-socket latencies. We discuss one way to alleviate this problem by circumventing certain access patterns.

Intra-socket (non-)uniformity does matter. Within a socket, the fact that the distance from the cores to the LLC is the same, or differs among cores, even only slightly, impacts the scalability of synchronization. For instance, under high contention, *SPARC-T2* (uniform) enables approximately 1.7 times higher scalability than *Tilera* (non-uniform) for all locking schemes. The developer of a concurrent system should thus be aware that highly-contended data pose a higher threat in the presence of even the slightest non-uniformity, such as non-uniformity inside a socket.

Loads and stores can be as expensive as atomic operations. In the context of synchronization, where memory operations are often accompanied with memory fences, loads and stores are generally not significantly cheaper than atomic operations with higher consensus numbers [101]. Even without fences, on data that are not locally cached, a compare-and-swap is roughly only 1.35 (on *Opteron*) and 1.15 (on *Westmere*) times more expensive than a load.

Message passing shines when contention is very high. Structuring an application with message passing reduces sharing and proves beneficial when a large number of threads contend for a few data. However, under low contention and/or a small number of cores, locks perform better on higher-layer concurrent testbeds (i.e., a hash table) even when message passing is provided in hardware (e.g., *Tilera*). This suggests the exclusive use of message passing for optimizing certain highly contended parts of a system.

Every locking scheme has its fifteen minutes of fame. None of the nine locking schemes we consider consistently outperforms any other one, on all target architectures or workloads. Strictly speaking, to seek optimality, a lock algorithm should thus be selected based on the hardware platform and the expected workload.

Simple locks are powerful. Overall, an efficient implementation of a ticket lock is the best performing synchronization scheme in most low contention workloads. Even under rather high contention, the ticket lock performs comparably to more complex locks, in particular within a socket. Consequently, given their small memory footprint, ticket locks should be preferred, unless it is sure that a specific lock will be very highly contended.

A high-level ramification of many of these observations is that the scalability of synchronization appears, first and above all, to be a property of the hardware, in the following sense. Basically, in order to be able to scale, synchronization should better be confined to a single socket, ideally a uniform one. On certain platforms (e.g., *Opteron*), this is simply impossible due to the hardware. Within a socket, sophisticated synchronization schemes are generally not worthwhile. Even if, strictly speaking, no size fits all, a proper implementation of a simple ticket lock seems enough.

In summary, the main contribution of this chapter is the most exhaustive study of synchronization to date. Results of this study can be used to help predict the cost of a synchronization scheme, explain its behavior, design better schemes, as well as possibly improve future hardware design. SSYNC, the cross-platform synchronization suite we built to perform the study is, we believe, a contribution of independent interest. SSYNC abstracts various lock algorithms behind a common interface: It not only includes many state-of-the-art algorithms, but also provides platform specific optimizations with substantial performance improvements. SSYNC also contains a library that abstracts message passing on various platforms, and a set of microbenchmarks for measuring the latencies of the cache-coherence protocols, the locks, and the message passing. SSYNC is available at `http://lpd.epfl.ch/site/ssync`.

The rest of the chapter is organized as follows. We present more details about our target platforms in Section 4.2. We describe SSYNC in Section 4.3. We present our analyses of synchronization from the hardware and software perspectives, in Sections 4.4 and 4.5, respectively. Finally, in Section 4.6, we conclude this chapter.

4.2 Target Platforms in Detail

This section describes in detail the four platforms considered in our experiments. Each is representative of a specific type of multi-core architecture. In Section 2.1, we provide a higher-level overview of these platforms.

In this chapter, we consider two large-scale multi-socket multi-cores, henceforth called the *multi-sockets*, and two large-scale chip multi-processors (CMPs), henceforth called the *single-sockets*. The multi-sockets are the 4-socket AMD Opteron (*Opteron*) and the 8-socket Intel Xeon (*Westmere*), whereas the CMPs are the 8-core Sun Niagara 2 (*SPARC-T2*) and the 36-core Tilera TILE-Gx36 (*Tilera*). The characteristics of the four platforms are detailed in Table 4.1.

Name	***Opteron***	***Westmere***	***SPARC-T2***	***Tilera***
System	AMD Magny Cours	Intel Westmere-EX	SUN SPARC-T5120	Tilera TILE-Gx36
Processors	4× Opteron 6172	8× Xeon E7-8867L	UltraSPARC-T2	TILE-Gx CPU
# Cores	48	80 (SMT disabled)	8 (64 contexts)	36
Core Clock	2.1 GHz	2.13 GHz	1.2 GHz	1.2 GHz
DVFS	-	disabled	-	-
L1 Cache	64/64 KB I/D	32/32 KB I/D	16/8 KB I/D	32/32 KB I/D
L2 Cache	512 KB	256 KB		256 KB
Last-level Cache	2× 6 MB (per die)	30 MB (shared)	4 MB (shared)	9 MB (distributed)
Interconnect	6.4 GT/s HyperTransport (HT) 3.0	6.4 GT/s QuickPath Interconnect (QPI)	Niagara2 Crossbar	Tilera iMesh
Memory #Channels / #Nodes	128 GB DDR3-1333 4 per socket / 8	192 GB DDR3-1067 4 per socket / 8	32 GB FB-DIMM-400 8 / 1	16 GB DDR3-800 4 / 2
OS / Kernel	Ubuntu 12.04.2/3.4.2	Red Hat EL 6.3/2.6.32	Solaris 10 u7	Tilera EL 6.3/2.6.40

Table 4.1 – Details about the hardware and the OS characteristics of the target platforms.

All platforms have a single die per socket, aside from the Opteron, that has two. Given that these two dies are actually organized in a 2-socket topology, we use the term *socket* to refer to a single die for simplicity.

4.2.1 Multi-Socket – Directory-Based: *Opteron*

The 48-core AMD Opteron contains four multi-chip modules (MCMs). Each MCM has two 6-core dies with independent memory controllers. Hence, the system comprises, overall, eight memory nodes. The topology of the system is depicted in Figure 7.1b. The maximum distance between two dies is two hops. The dies of an MCM are situated at a 1-hop distance, but they share more bandwidth than two dies of different MCMs.

The caches of *Opteron* are write-back and non-inclusive [45]. In other words, every new cache fill goes in the L1 cache, but not in the L2/L3 which are victim caches, filled by the evictions of the L1 and the L2 respectively. Nevertheless, the hierarchy is not strictly exclusive; on an LLC hit the data is pulled in the L1 but may or may not be removed from the LLC (decided by the hardware [7]). *Opteron* uses the MOESI protocol for cache coherence. 'O' stands for the *owned* state, which indicates that this cache line has been modified (by the owner) but there might be more shared copies on other cores. This state allows a core to load a modified line of another core without the need to invalidate the modified line. The modified cache line simply changes to *owned* and the new core receives the line in shared state. Cache coherence is implemented with a broadcast-based protocol, assisted by what is called the HyperTransport Assist (also known as the probe filter) [45]. The probe filter is, essentially, a directory residing in the LLC.[2] An entry in the directory holds the owner of the cache line, which can be used to directly probe or invalidate the copy in the local caches of the owner core.

4.2.2 Multi-Socket – Broadcast-Based: *Westmere*

The 80-core Intel Xeon consists of eight sockets of 10-cores.[3] These form a twisted hypercube, as depicted in Figure 2.3b maximizing the distance between two nodes to two hops. *Westmere* uses inclusive caches [111]—every new cache-line fill occurs in all the three levels of the hierarchy. The LLC is write-back; the data is written to the memory only upon an eviction of a modified line due to space or coherence. Within a socket, *Westmere* implements coherence by snooping. Across sockets, it broadcasts snoop requests to the other sockets. Within the socket, the LLC keeps track of which cores might have a copy of a cache line. Additionally, *Westmere* extends the MESI protocol with the *forward* state [112]. This state is a special form of the shared state and indicates the only cache that will respond to a load request for that line (thus reducing bandwidth usage).

[2] Typically, the probe filter occupies 1 MB of the LLC.
[3] SMT is disabled for the experiments in this chapter.

4.2.3 Single-Socket – Uniform: *SPARC-T2*

The Sun Niagara 2 is a single-die processor that incorporates 8 cores. It is based on the chip multi-threading architecture; it provides 8 hardware threads per core, totaling 64 hardware threads. Each L1 cache is shared among the 8 hardware threads of a core and is write-through to the LLC. The 4 MB LLC (L2) cache is divided into eight banks of 512 KB each and is shared by the 8 cores. The 8 cores communicate with the shared LLC through a crossbar [162], which means that each core is equidistant from the LLC (*uniform*). The cache-coherence implementation is directory-based and uses duplicate tags [155] (i.e., the LLC cache holds a directory of all the L1 lines).

4.2.4 Single-Socket – Non-Uniform: *Tilera*

The Tilera TILE-Gx36 [205] is a 36-core chip multi-processor. The cores, also called tiles, are allocated on a 2-dimensional mesh and are connected with Tilera's iMesh on-chip network. iMesh handles the coherence of the data and also provides hardware message passing to the applications. *Tilera* implements the dynamic distributed cache technology [205]. All L2 caches are accessible by every core on the chip, thus, the L2s are used as a 9 MB coherent LLC. The hardware uses a distributed directory to implement coherence. Each cache line has a *home tile* (i.e., the actual L2 cache where the data reside if cached by the distributed LLC). Consider, for example, the case of core x loading an address homed on tile y. If the data is not cached in the local L1 and L2 caches of x, a request for the data is sent to the L2 cache of y, which plays the role of the LLC for this address. Clearly, the latency of accessing the LLC depends on the distance between x and y, hence *Tilera* is a *non-uniform* cache architecture.

4.3 SSYNC: A Cross-Platform Synchronization Library

SSYNC is our cross-platform synchronization suite; it works on `x86_64`, `SPARC`, and Tilera processors. SSYNC contains `libslock`, a library that abstracts lock algorithms behind a common interface and `libssmp`, a library with fine-tuned implementations of message passing for each of the four platforms. SSYNC also includes microbenchmarks for measuring the latencies of cache coherence, the locks, and message passing, as well as a cache efficient hash table (`ssht`).

4.3.1 Libraries

libslock. This library contains a common interface and optimized implementations of a number of widely used locks (see Table 2.1 for a short description of the algorithms). `libslock` includes three simple spinlocks, namely TAS, TTAS with exponential backoff [10, 103], and TICKET. The queue locks are the MCS lock and the CLH lock. We also employ an array-based lock (ARRAY). `libslock` also contains hierarchical locks, such as HCLH and the hierarchical ticket lock (HTICKET).[4] Finally, `libslock` abstracts the pthread mutex interface (MUTEX). `libslock` also contains a cross-platform interface for atomic instructions and other architecture dependent operations, such as fences, thread and memory placement functions.

libssmp. `libssmp` is our implementation of message passing over cache coherence (similar to the one in Barrelfish [17]).[5] It uses cache line-sized buffers (messages) in order to complete message transmissions with single cache-line transfers. Each buffer is one-directional and includes a byte flag to designate whether the buffer is empty or contains a message. For client-server communication, `libssmp` implements functions for receiving from any other, or from a specific subset of the threads. Even though the design of `libssmp` is identical on all platforms, we leverage the results of Section 4.4 to tailor `libssmp` to the specifics of each platform individually.

4.3.2 Microbenchmarks and Concurrent Software

ccbench. `ccbench` is a tool for measuring the cost of operations on a cache line, depending on the line's MESI state and placement in the system. `ccbench` brings the cache line in the desired state and then accesses it from either a local or a remote core. `ccbench` supports 30 cases, such as store on modified and test-and-set on shared lines.

Stress Tests. SSYNC provides tests for the primitives in `libslock` and `libssmp`. These tests can be used to measure the primitives' latency or throughput under various conditions (e.g., number and placement of threads, level of contention).

Hash Table (ssht). `ssht` is a concurrent hash table that exports three operations: `put`, `get`, and `remove`. It is designed to place the data as efficiently as possible in the caches in order to (i) allow for efficient prefetching and (ii) avoid false sharing. `ssht` can be configured to use any of the locks of `libslock` or the message passing of `libssmp`.

4.4 Hardware-Level Analysis

In this section, we report on the latencies incurred by the hardware cache-coherence protocols and discuss how to reduce them in certain cases. These latencies constitute a good estimation of the cost of sharing a cache line in a multi-core platform and have a significant impact on the scalability of any synchronization scheme. We use `ccbench` to measure basic operations such as load and store, as well as compare-and-swap (CAS), fetch-and-increment (FAI), test-and-set (TAS), and swap (SWAP).

[4] In fact, based on the results of Section 4.4 and without being aware of [56], we designed and implemented the HTICKET algorithm.

[5] On *Tilera*, it is an interface to the hardware message passing.

4.4.1 Local Accesses

Table 4.2 contains the latencies for accessing the local caches of a core. In the context of synchronization, the values for the LLCs are worth highlighting. On *Westmere*, the 44 cycles is the local latency to the LLC, but also corresponds to the fastest communication between two cores on the same socket. The LLC plays the same role for the single-socket platforms, however, it is directly accessible by all the cores of the system. On *Opteron*, the non-inclusive LLC holds both data and the cache directory, so the 40 cycles is the latency to both. However, the LLC is filled with data only upon an eviction from the L2, hence the access to the directory is more relevant to synchronization.

	Opteron	*Westmere*	*SPARC-T2*	*Tilera*
L1	3	5	3	2
L2	15	11		11
LLC	40	44	24	45
RAM	136	355	176	118

Table 4.2 – Local caches and memory latencies (cycles).

4.4.2 Remote Accesses

Table 4.3 contains the latencies to load, store, or perform an atomic operation on a cache line based on the cache line's previous state and location. Notice that the accesses to an invalid line are accesses to the main memory. In the following, we discuss all cache-coherence states, except for the invalid. We do not explicitly measure the effects of the forward state of *Westmere*: There is no direct way to bring a cache line to this state. The effects of forward state are included in the load from shared case.

Loads. On *Opteron*, a load has basically the same latency regardless of the previous state of the line; essentially, the steps taken by the cache-coherence protocol are always the same. Interestingly, although the two dies of an MCM are tightly coupled, the benefits are rather small. The latencies between two dies in an MCM and two dies that are simply directly connected differ by roughly 12 cycles. One extra hop adds an additional overhead of 80 cycles. Overall, an access over two hops is approximately 3 times more expensive than an access within a die.

The results in Table 4.3 represent the best-case scenario for *Opteron*: At least one of the involved cores resides on the memory node of the directory. If the directory is remote to both cores, the latencies increase proportionally to the distance. In the worst case, where two cores are on different nodes, both 2-hops away from the directory, the latencies are 312 cycles. Even worse, even if both cores reside on the same node, they still have to access the remote directory, wasting any locality benefits.

System	*Opteron*				*Westmere*			*SPARC-T2*		*Tilera*	
Hops / State	same die	same MCM	one hop	two hops	same die	one hop	two hops	same core	other core	one hop	max hops
loads											
Modified	81	161	172	252	109	289	400	3	24	45	65
Owned	83	163	175	254	-	-	-	-	-	-	-
Exclusive	83	163	175	253	92	273	383	3	24	45	65
Shared	83	164	176	254	44	223	334	3	24	45	65
Invalid	136	237	247	327	355	492	601	176	176	118	162
stores											
Modified	83	172	191	273	115	320	431	24	24	57	77
Owned	244	255	286	291	-	-	-	-	-	-	-
Exclusive	83	171	191	271	115	315	425	24	24	57	77
Shared	246	255	286	296	116	318	428	24	24	86	106
atomic operations: CAS (C), FAI (F), TAS (T), SWAP (S)											
Operation	all	all	all	all	all	all	all	C/F/T/S	C/F/T/S	C/F/T/S	C/F/T/S
Modified	110	197	216	296	120	324	430	71/108/64/95	66/99/55/90	77/51/70/63	98/71/89/84
Shared	272	283	312	332	113	312	423	76/99/67/93	66/99/55/90	124/82/121/95	142/102/141/115

Table 4.3 – Latencies (cycles) of the cache coherence to load/store/CAS/FAI/TAS/SWAP a cache line depending on the MESI state and the distance. The values are the average of 10000 repetitions with < 3% standard deviation.

In contrast, *Westmere* does not have the locality issues of *Opteron*. If the data is present within the socket, a load can be completed locally due to the inclusive LLC. Loading from the shared state is particularly interesting, because the LLC can directly serve the data without needing to probe the local caches of the previous owner (unlike the modified and exclusive states). However, the overhead of going off-socket on *Westmere* is very high. For instance, loading from the shared state is 7.5 times more expensive over two hops than loading within the socket.

Unlike the large variability on multi-sockets, the results are more stable on the single-sockets. On *SPARC-T2*, a load costs an L1 or L2 access, depending on whether the two threads reside on the same core. On *Tilera*, the LLC is distributed, hence the latencies depend on the distance of the requesting core from the home tile of the cache line. The cost for two adjacent cores is 45 cycles, whereas for the two most remote cores,[6] it is 20 cycles higher (2 cycles per hop).

Stores. On *Opteron*, both loads and stores on a modified or an exclusive cache line have similar latencies (no write-back to memory). However, a store on a shared line is different.[7] Every store on a shared or owned cache line incurs a broadcast invalidation to all nodes. This happens because the cache directory is incomplete (it does not keep track of the sharers) and does not in any way detect whether sharing is limited within the node.[8] Therefore, even if all sharers reside on the same node, a store needs to pay the overhead of a broadcast, thus increasing the cost from around 83 to 244 cycles. Obviously, the problem is aggravated if the

6 10 hops distance on the 6-by-6 2-dimensional mesh of *Tilera*.

7 For the store on shared test, we place two different sharers on the indicated distance from a third core that performs the store.

8 On *Westmere*, the inclusive LLC is able to detect if there is sharing solely within the socket.

directory is not local to any of the cores involved in the store. Finally, the scenario of storing on a cache line shared by all 48 cores costs 296 cycles.

Again, *Westmere* has the advantage of locality; *Westmere* locally completes an operation that involves solely cores of a single node. In general, stores behave similarly regardless of the previous state of the cache line. Finally, storing on a cache line shared by all 80 cores on *Westmere* costs 445 cycles.

Similarly to a load, the results for a store exhibit much lower variability on the single-sockets. A store on *SPARC-T2* has essentially the latency of the L2, regardless of the previous state of the cache line and the number of sharers. On *Tilera*, stores on a shared line are a bit more expensive due to the invalidation of the cache lines of the sharers. The cost of a store reaches a maximum of 200 cycles when all 36 cores share that line.

Atomic Operations. On the two multi-sockets, CAS, TAS, FAI, and SWAP have essentially the same latencies. These latencies are similar to a store followed by a memory barrier. On the single-sockets, some operations clearly have different hardware implementations. For instance, on *Tilera*, the FAI operation is faster than the others. Another interesting point is the latencies for performing an operation when the line is shared by all the cores of the system. On all platforms, the latencies follow the exact same trends as a store in the same scenario.

Implications. The latencies of cache coherence reveal some important issues that should be addressed in order to implement efficient synchronization. Cross-socket communication is 2 to 7.5 times more expensive than intra-socket communication. The problem on the broadcast-based design of *Westmere* is larger than on *Opteron*. However, within the socket, the inclusive LLC of *Westmere* provides strong locality, which in turn translates into efficient intra-socket synchronization. In terms of locality, the incomplete directory of *Opteron* is problematic in two ways. First, a read-write pattern of sharing, causes stores on owned and shared cache lines to exhibit the latency of a cross-socket operation, even if all sharers reside on the same socket. We thus expect intra-socket synchronization to behave similarly to the cross-socket. Second, the location of the directory is crucial: If the cores that use some memory are remote to the directory, they pay the remote access overhead. To achieve good synchronization performance, the data has to originate from the local memory node (or to be migrated to the local one). Overall, an *Opteron* MCM should be treated as a two-node platform.

The single-sockets exhibit quite a different behavior: They both use their LLCs for sharing. The latencies (to the LLC) on *SPARC-T2* are uniform (i.e., they are affected by neither the distance nor the number of the involved cores). We expect this uniformity to translate to synchronization that is not prone to contention. The non-uniform *Tilera* is affected both by the distance and the number of involved cores, therefore we expect scalability to be affected by contention. Regarding the atomic operations, both single-sockets have faster implementations for some of the operations (see Table 4.3). These should be preferred to achieve the best synchronization performance.

4.4.3 Enforcing Locality

A store to a shared or owned cache line on *Opteron* induces an unnecessary broadcast of invalidations, even if all the involved cores reside on the same node (see Table 4.3). This results in a 3-fold increase of the latency of the store operation. In fact, to avoid this issue, we propose to explicitly maintain the cache line to the modified state. This can be easily achieved by calling the `prefetchw` x86 instruction before any load reference to that line. Of course, this optimization should be used with care because it disallows two cores to simultaneously hold a copy of the line.

To illustrate the potential of this optimization, we engineer an efficient implementation of a ticket lock. As we describe in Table 2.1, a ticket lock consists of two counters: the *ticket* and the *current*. To acquire the lock, a thread atomically fetches and increases the *ticket* counter—it obtains a ticket *tick*. If *tick* equals the *current* counter, the thread has acquired the lock, otherwise, the thread spins until this becomes true. To release the lock, the thread increases the value of the *current* counter.

A particularly appealing characteristic of the ticket lock is the fact that the *ticket*, subtracted by the *current* counter, is the number of threads queued before the current thread. Accordingly, it is intuitive to spin with a backoff proportional to the number of threads queued in front [149]. We use this backoff technique with and without the `prefetchw` optimization and compare the results with a non-optimized implementation of the ticket lock. Figure 4.2 depicts the latencies for acquiring and immediately releasing a single lock. Obviously, the non-optimized version scales terribly, delivering a latency of 720K cycles on 48 cores. In contrast, the versions with the proportional backoff scale significantly better. The `prefetchw` gives an extra performance boost, performing up to 2 times better on 48 cores.

SSYNC uses the aforementioned optimization wherever possible. For example, our message passing implementation on *Opteron* with the `prefetchw` technique is up to 2.5 times faster than without it.

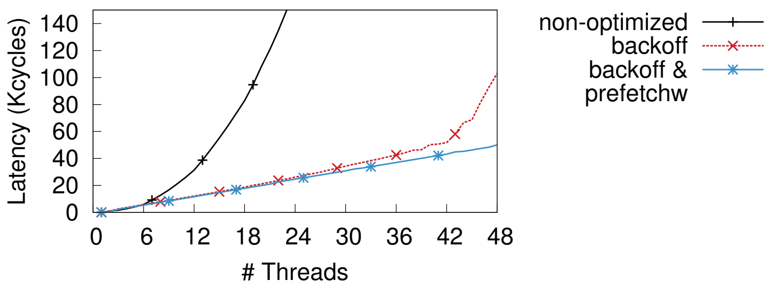


Figure 4.2 – Latency of acquiring different implementations of a ticket lock on *Opteron*.

4.4.4 Stressing Atomic Operations

In this test, we stress the various atomic operations of our four platforms. Each thread repeatedly tries to perform an atomic operation on a single shared location. For FAI, SWAP, and CAS_FAI these calls are always eventually successful (i.e., they write to the target memory), whereas for TAS and CAS they are not. CAS_FAI implements a FAI operation based on CAS. CAS_FAI enables us to highlight both the costs of spinning until CAS is successful and the benefits of having a FAI instruction supported by the hardware. After completing a call, the thread pauses for a sufficient number of cycles to prevent the same thread from completing consecutive operations locally (long runs [153]). This delay is proportional to the maximum latency across the involved cores and does not affect the total throughput in a way other than the intended.

On the multi-sockets, we allocate threads on the same socket and continue on the next socket once all cores of the previous one have been used. On *SPARC-T2*, we divide threads evenly among the eight physical cores. On all platforms, we ensure that each thread allocates its local data from the local node. We repeat each experiment five times and show the average value.

Figure 4.3 shows the results of this experiment. The multi-sockets exhibit a very steep decrease in the throughput once the location is accessed by more than one core. The latency of the operations increases from approximately 20 to 120 cycles. In contrast, the single-sockets generally show an increase in the throughput on the first few cores. This can be attributed to the cost of the local parts of the benchmark (e.g., a while loop) that consume time comparable to the latency of the operations. For more than six cores, however, the results stabilize (with a few exceptions).

Both *Opteron* and *Westmere* exhibit a stable throughput close to 20 Mops/s within a socket, which drops once there are cores on a second socket. Not surprisingly (see Table 4.3), the drop on *Westmere* is larger than on *Opteron*. The throughput on these platforms is dictated by the cache-coherence latencies, given that an atomic operation actually brings the data in its local cache. In contrast, on the single-sockets the throughput converges to a maximum value and exhibits no subsequent decrease. Some further interesting points worth highlighting

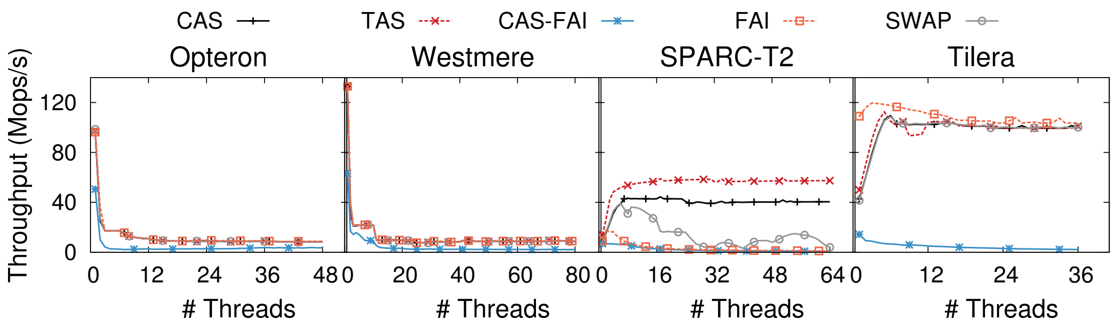


Figure 4.3 – Throughput of different atomic operations on a single memory location.

are as follows. First, *SPARC-T2* (SPARC architecture) does not provide an atomic increment or swap instruction. Their implementations are based on CAS, therefore the behavior of FAI and CAS_FAI are practically identical. SWAP shows some fluctuations on *SPARC-T2*, which we believe are caused by the scheduling of the hardware threads. However, SPARC provides a hardware TAS implementation that proves to be highly efficient. Likewise, the FAI implementation on *Tilera* slightly outperforms the other operations.

Implications. Both multi-sockets have a very fast single-thread performance, that drops on two or more cores and decreases further when there is cross-socket communication. Contrarily, both single-sockets have a lower single-thread throughput, but scale to a maximum value, that is subsequently maintained regardless of the number of cores. This behavior indicates that globally stressing a cache line with atomic operations will introduce performance bottlenecks on the multi-sockets, while being somewhat less of a problem on the single-sockets. Finally, a system designer should take advantage of the best performing atomic operations available on each platform, like the TAS on *SPARC-T2*.

4.5 Software-Level Analysis

This section describes the software-oriented part of this study. We start by analyzing the behavior of locks under different levels of contention and continue with message passing. We use the same methodology as in Section 4.4.4 in the experiments of this section as well. In addition, the globally shared data is allocated from the first participating memory node. We finally report on our findings on higher-level concurrent software and we discuss the implications of the results of this chapter on other synchronization approaches such as combiners and lock-free programming.

4.5.1 Locks

We evaluate the locks in SSYNC under various degrees of contention on our platforms. We evaluate (i) no contention, (ii) under extreme and very low contention (within each platform), and (iii) under intermediate contention (across platforms).

Uncontested Locking

In this experiment we measure the latency to acquire a lock based on the location of the previous owner. Although in a number of cases acquiring a lock does involve contention, a large portion of acquisitions in applications are uncontested, hence they have a similar behavior to this experiment.

Initially, we place a single thread that repeatedly acquires and releases the lock. We then add a second thread, as close as possible to the first one, and pin it further in subsequent

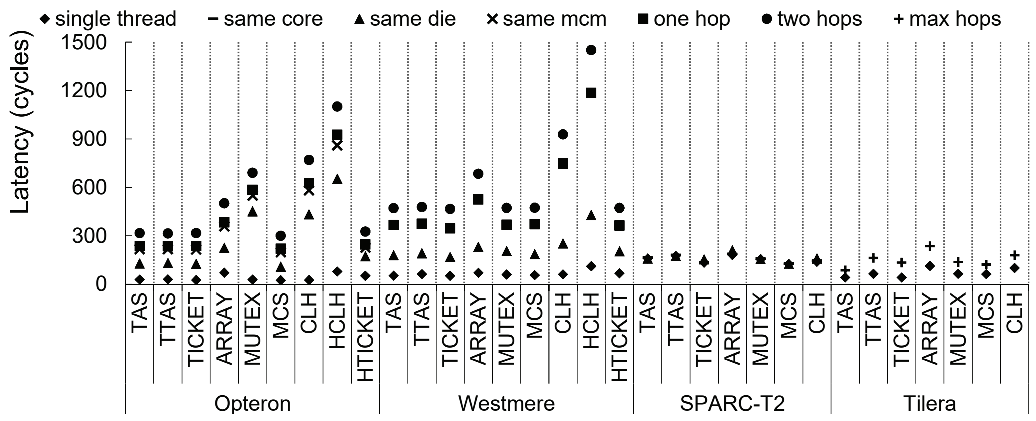


Figure 4.4 – Uncontested lock acquisition latency based on the location of the previous owner of the lock.

runs. Figure 4.4 contains the latencies of the different locks when the previous owner is at various distances. Latencies suffer important increases on the multi-sockets as the second thread moves further from the first. In general, acquisitions that need to transfer data across sockets have a high cost. Remote acquisitions can be up to 12.5 and 11 times more expensive than local ones on *Opteron* and *Westmere* respectively. In contrast, due to the shared and distributed LLCs, *SPARC-T2* and *Tilera* suffer no and slight performance decrease, respectively, as the location of the second thread changes. The latencies of the locks are in accordance with the cache-coherence latencies presented in Table 4.3.

Moreover, the differences in the latencies are significantly larger between locks on the multi-sockets than on the single-sockets, making the choice of the lock algorithm in an uncontested scenario paramount to performance. More precisely, while simple spinlocks (i.e., TAS, TTAS, TICKET) closely follow the cache-coherence latencies, more complex locks generally introduce some additional overhead.

Implications. Using a lock, even if no contention is involved, is up to one order of magnitude more expensive when crossing sockets. The 350-450 cycles on a multi-socket, and the roughly 200 cycles on a single-socket, are not negligible, especially if the critical sections are short. Moreover, the penalties induced when crossing sockets in terms of latency tend to be higher for complex locks than for simple locks. Therefore, regardless of the platform, simple locks should be preferred, when contention is very low.

Lock Algorithm Behavior

We study the behavior of locks under extreme and very low contention. On the one hand, highly contended locks are often the main scalability bottleneck of software systems. On the other hand, many systems use locking strategies, such as fine-grained locks, that induce low contention. Therefore, good performance in these two scenarios is essential. We measure

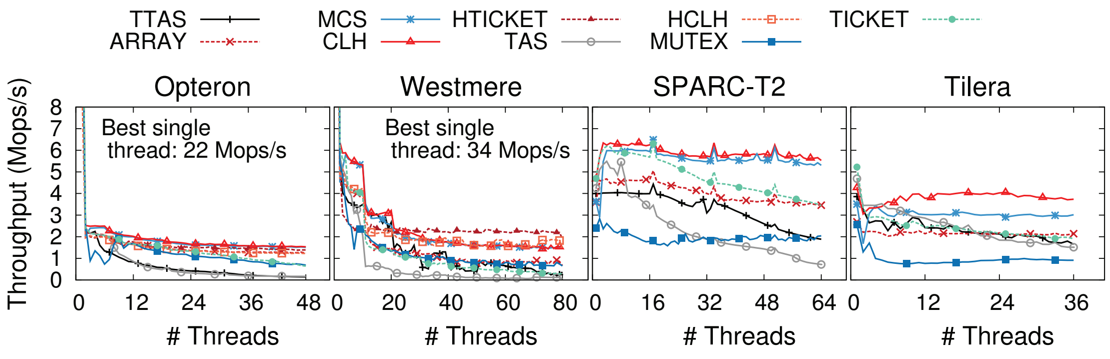


Figure 4.5 – Throughput of different lock algorithms using a single lock.

the total throughput of lock acquisitions that can be performed using each lock algorithm. Each thread acquires a randomly-chosen lock, reads and writes one corresponding cache line of data, and releases the lock. Similarly to the atomic operations stress test (Section 4.4.4) in the single-lock experiment, a thread pauses after it releases the lock, in order to ensure that the release becomes visible to the other cores before retrying to acquire the lock. Given the uniform structure of the platforms, we do not use hierarchical locks on the single-sockets.

Extreme Contention. Figure 4.5 depicts the results of the maximum contention experiment (one lock). As we mentioned earlier, our microbenchmark employs sufficient delays after releasing the lock so that we avoid long runs. As we described in Section 4.4, *Westmere* exhibits very strong intra-socket locality. Accordingly, hierarchical locks (i.e., HTICKET and HCLH) perform the best. Although there is a very big drop from one to two cores on the multi-sockets, within the socket both *Opteron* and *Westmere* manage to keep a rather stable performance. However, once a second socket is involved the throughput decreases again.

Not surprisingly, the CLH and the MCS locks are the most resilient to contention. They both guarantee that a single thread is spinning on each cache line and use the globally shared data only to enqueue for acquiring the lock. TICKET proves to be the best simple spinlock on this workload. Overall, the throughput on two or more cores on the multi-sockets is an order of magnitude lower than the single-core performance. In contrast, the single-sockets maintain a comparable performance on multiple cores.

Very Low Contention. The very low contention results (512 locks) are shown in Figure 4.6. Once again, one can observe the strong intra-socket locality of *Westmere*. In general, simple locks match or even outperform the more complex queue locks. While on *Westmere* the differences between locks become insignificant for a large number of cores, it is generally TICKET that performs the best on *Opteron*, *SPARC-T2*, and *Tilera*. On a low-contention scenario it is thus difficult to justify the memory requirements that complex lock algorithms have. It should be noted that, aside from the acquisitions and releases, the load and the store on the protected data also contribute to the lack of scalability of multi-sockets, for the reasons pointed out in Section 4.4.

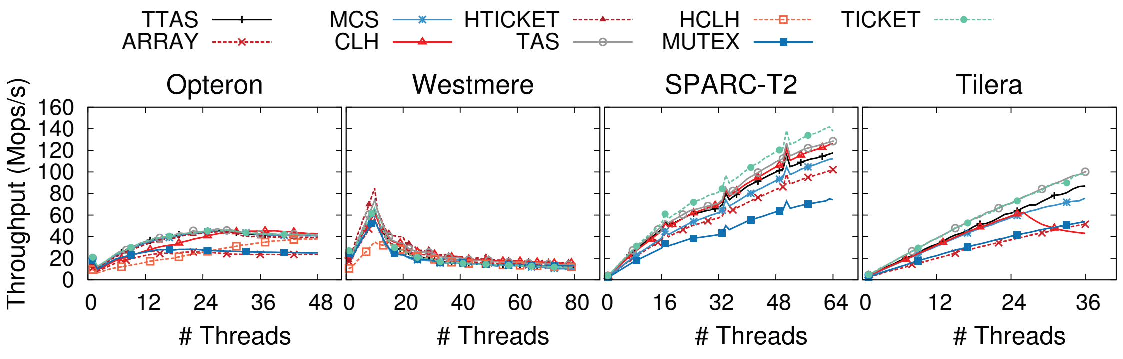


Figure 4.6 – Throughput of different lock algorithms using 512 locks.

Implications. None of the locks is consistently the best on all platforms. Moreover, no lock is consistently the best within a platform. While complex locks are generally the best under extreme contention, simple locks perform better under low contention. Under high contention, hierarchical locks should be used on multi-sockets with strong intra-socket locality, such as *Westmere*. *Opteron*, due to the previously discussed locality issues, and the single-sockets favor queue locks. In case of low contention, simple locks are better than complex implementations within a socket. Under extreme contention, while not as good as more complex locks, a ticket lock can avoid performance collapse within a socket. On *Westmere*, the best performance is achieved when all threads run on the same socket, both for high and for low contention. Therefore, synchronization between sockets should be limited to the absolute minimum on such platforms. Finally, we observe that when each core is dedicated to a single thread there is no scenario in which pthread mutexes perform the best. Mutexes are however useful when threads contend for a core. Therefore, unless multiple threads run on the same core, alternative implementations should be preferred. In Section 5.5.1, we introduce MUTEXEE, an optimized implementation of pthread mutex.

Cross-Platform Lock Behavior

In this experiment, we compare lock behavior under various degrees of contention across architectures. In order to have a straightforward cross-platform comparison, we run the tests on up to 36 cores. Having already explored the lock behavior of different algorithms, we only report the highest throughput achieved by any of the locks on each platform. We vary the contention by running experiments with 4, 16, 32, and 128 locks, thus examining high, intermediate, and low degrees of contention.

The results are shown in Figure 4.7. In all cases, the differences between the single and multi-sockets are noticeable. Under high contention, single-sockets prevent performance collapse from one thread to two or more, whereas in the lower contention cases these platforms scale well. As noted in Section 4.4, stores and atomic operations are affected by contention on *Tilera*, resulting in slightly less scalability than on *SPARC-T2*: On high contention workloads, the uniformity of *SPARC-T2* delivers up to 1.7 times more scalability than *Tilera* (i.e., the rate at which performance increases). In contrast, multi-sockets exhibit a significantly lower

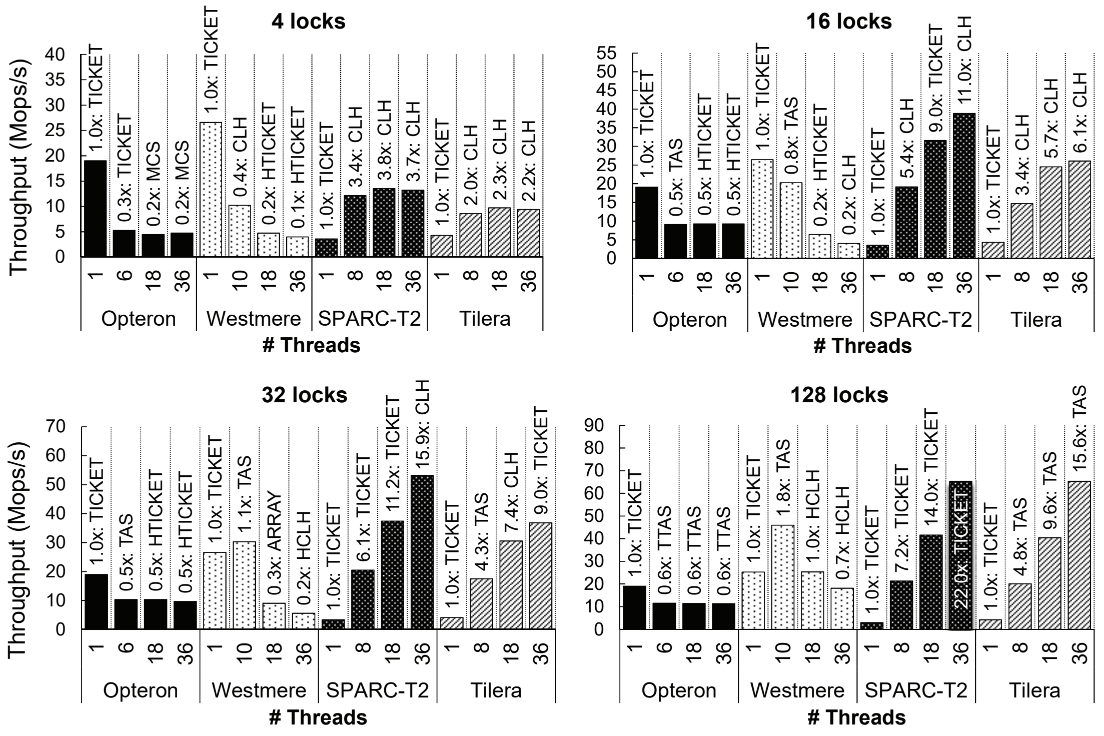


Figure 4.7 – Throughput and scalability of locks depending on the number of locks. The "X : Y" labels on top of each bar indicate the best-performing lock (Y) and the scalability over the single-thread execution (X).

throughput for high contention, when compared to single-thread performance. Multi-sockets provide limited scalability even on the low contention scenarios. The direct cause of this contrasting behavior is the higher latencies for the cache-coherence transitions on multi-sockets, as well as the differences in the throughput of the atomic operations. It is worth noticing that *Westmere* scales well when all the threads are within a socket. Performance, however, severely degrades even with one thread on a remote socket. In contrast, *Opteron* shows poor scalability regardless of the number of threads. The reason for this difference is the limited locality of *Opteron* we discussed in Section 4.4.

Implications. There is a significant difference in scalability trends between multi and single-sockets across various degrees of contention. Moreover, even a small degree of non-uniformity can have an impact on scalability. As contention drops, simple locks should be used in order to achieve high throughput on all architectures. Overall, we argue that synchronization-intensive systems should favor platforms that provide locality (i.e., they can prevent cross-socket communication).

4.5.2 Message Passing

We evaluate the message passing implementations of SSYNC. To capture the most prominent communication patterns of a message passing application, we evaluate both one-to-one and client-server communication. The size of a message is 64 bytes (a cache line).

One-to-One Communication. Figure 4.8 depicts the latencies of two cores that exchange one-way and round-trip messages. As expected, *Tilera*'s hardware message passing performs the best. Not surprisingly, a one-way message over cache coherence costs roughly twice the latency of transferring a cache line. Once a core x receives a message, it brings the receive buffer (i.e., a cache line) to its local caches. Consequently, the second core y has to fetch the buffer (first cache-line transfer) in order to write a new message. Afterwards, x has to re-fetch the buffer (second transfer) to get the message. Accordingly, the round-trip case takes approximately four times the cost of a cache-line transfer. The reasoning is exactly the same with one-way messages, but applies to both ways: Send and then receive.

Client-Server Communication. Figure 4.9 depicts the one-way and round-trip throughput for a client-server pattern with a single server. Again, the hardware message passing of *Tilera* performs the best. With 35 clients, one server delivers up to 16 Mops/s (round-trip) on *Tilera* (less on the other platforms). In this benchmark, the server does not perform any computation between messages, therefore the 16 Mops constitutes an upper bound on the performance of a single server. It is interesting to note that if we reduce the size of the message to a single word, the throughput on *Tilera* is 27 Mops/s for round-trip and more than 100 Mops/s for one-way messages, respectively.

Two additional observations are worth mentioning. First, *Westmere* performs very well within a socket, especially for one-way messages. The inclusive LLC cache plays the role of the

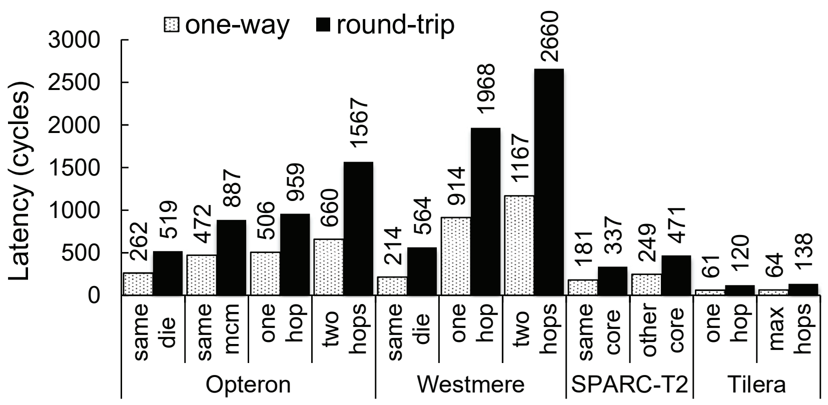


Figure 4.8 – One-to-one communication latencies of message passing depending on the distance between the two cores.

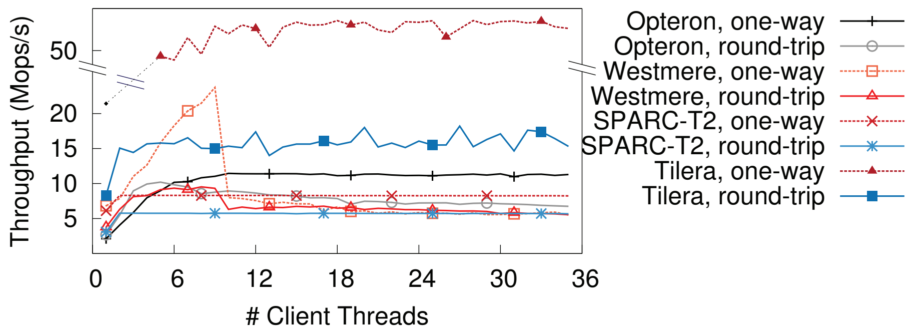


Figure 4.9 – Total throughput of client-server communication.

buffer for exchanging messages. However, even with a single client on a remote socket, the throughput drops from 25 to 8 Mops/s. The second point is that as the number of cores increases, the round-trip throughput becomes higher than the one-way on *Westmere*. We also observe this effect on *Opteron*, once the length of the local computation of the server increases (not shown in the graph). This happens because the request-response model enables the server to efficiently prefetch the incoming messages. On one-way messages, the clients keep trying to send messages, saturating the incoming queues of the server. This leads to the clients busy-waiting on cache lines that already contain a message. Therefore, even if the server prefetches a message, the client will soon bring the cache line to its own caches (or transform it to shared), making the consequent operations of the server more expensive.

Implications. Message passing can achieve latencies similar to transferring a cache line from one core to another. This behavior is essentially not affected by contention, because each pair of cores uses individual cache lines for communication. These observations apply both to one-to-one and client-server communication. However, a single server has a rather low upper bound on the throughput it can achieve, even when not executing any computation. In a sense, we have to trade performance for scalability.

4.5.3 Hash Table (`ssht`)

We evaluate `ssht` (i.e., the concurrent hash table implementation of SSYNC) under low (512 buckets) and high (12 buckets) contention, as well as short (12 elements) and long (48 elements) buckets. We use 80% `get`, 10% `put`, and 10% `remove` operations, so as to keep the size of the hash table constant. We configure `ssht` so that each bucket is protected by a single lock, the keys are 64 bit integers, and the payload size is 64 bytes. The trends on scalability pertain on other configurations as well. Finally, we configure the message passing (mp) version to use (i) one server per three cores[9] and (ii) round-trip operations (i.e., all operations block, waiting for a response from the server). It should be noted that dedicating

[9] This configuration achieves the highest throughput.

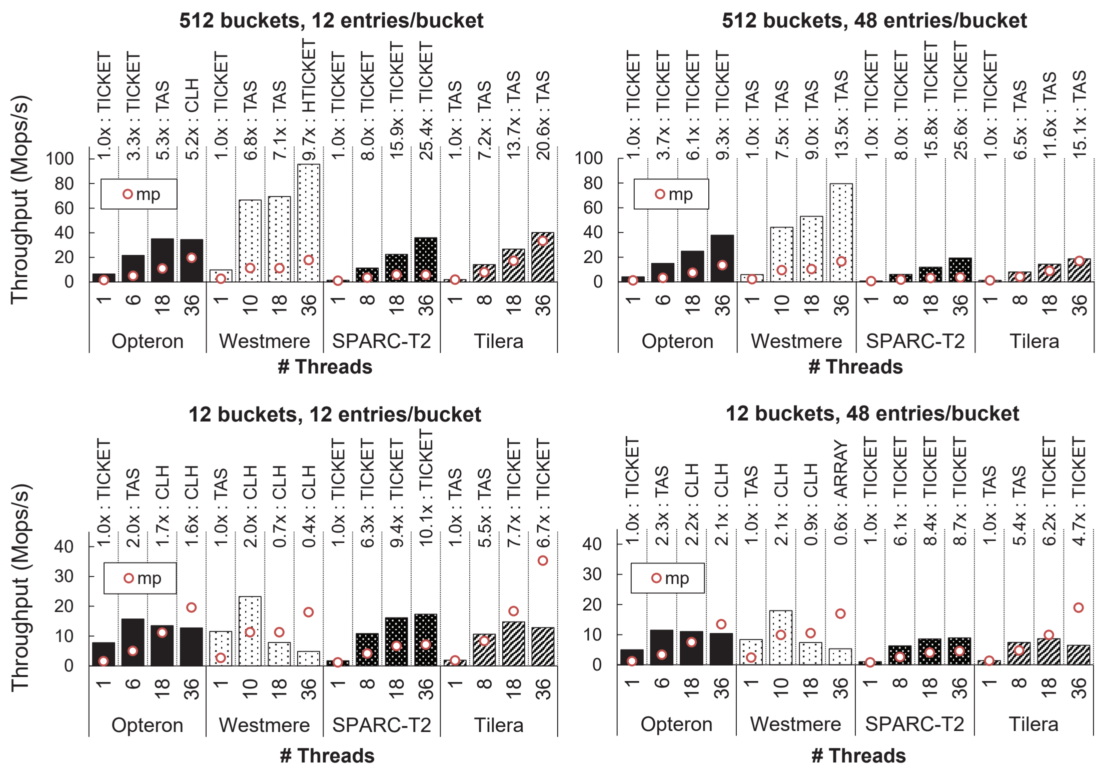


Figure 4.10 – Throughput and scalability of the hash table (`ssht`) on different configurations. The "X : Y" labels on top of each bar indicate the best-performing lock (Y) and the scalability over the single-thread execution (X).

some threads as servers reduces the contention induced on the shared data of the application. Figure 4.10 depicts the results on the four target platforms on the aforementioned scenarios.[10]

Low Contention. Increasing the length of the critical sections increases the scalability of the lock-based `ssht` on all platforms, except for *Tilera*. The multi-sockets benefit from the efficient prefetching of the data of a bucket. All three systems benefit from the lower single-thread performance, which leads to higher scalability ratios. On *Tilera*, the local data contend with the shared data for the L2 cache space, reducing scalability. On this workload, the message passing implementation is strictly slower than the lock-based ones, even on *Tilera*. It is interesting to note that *Westmere* scales slightly even outside the 10 cores of a socket, thus delivering the highest throughput among all platforms. Finally, the best performance in this scenario is achieved by simple spinlocks.

High Contention. The results are radically different for high contention. First, the message passing version not only outperforms the lock-based ones on three out of the four platforms (for high core counts), but it also delivers by far the highest throughput. The hardware threads of *SPARC-T2* do not favor client-server solutions; the servers are delayed due to the sharing

[10] The single-thread throughput for message passing is actually a result of a one server / one client execution.

of the core's resources with other threads. However, with locks, *SPARC-T2* achieves a 10-fold performance increase on 36 threads, which is the best scalability among the lock-based versions and approaches the optimal 12-fold scalability. It is worth mentioning that if we do not explicitly pin the threads on cores, the multi-sockets deliver 4 to 6 times lower maximum throughput on this workload.

Summary. These experiments illustrate two major points. First, increasing the length of a critical section can partially hide the costs of synchronization under low contention. This, of course, assumes that the data accessed in the critical section are mostly being read (so they can be shared) and follow a specific access pattern (so they can be prefetched). Second, the results illustrate how message passing can provide better scalability and performance than locking under extreme contention.

4.5.4 Key-Value Store (Memcached)

Memcached (v. 1.4.15) [70] is an in-memory key-value store, based on a hash table. Memcached's hash table has a large number of buckets and is protected by fine-grain locks. However, during certain rebalancing and maintenance tasks, Memcached dynamically switches to a global lock for short periods of time. Since we are interested in the effect of synchronization on performance and scalability, we replace the default pthread mutexes that protect the hash table, as well as the global locks, with the interface provided by `libslock`. In order to stress Memcached, we use the *memslap* tool from the *libmemcached* library [5] (v. 1.0.15). We deploy *memslap* on a remote server and use its default configuration. We use 500 client threads and run a get-only and a set-only tests.

Get. The *get* test does not cause any switches to global locks. Due to the essentially non-existent contention, the lock algorithm has little effect in this test. In fact, even completely removing the locks of the hash table does not result in any performance difference. This indicates that scalability is limited by bottlenecks other than synchronization.

Set. A write-intensive workload however stresses a number of global locks, which introduces contention. In the *set* test the differences in lock behavior translate in a difference in the performance of the application as well. Figure 4.11 shows the throughput on various platforms using different locks. We do not present the results on more than 18 cores, since none of the platforms scales further. Changing the MUTEX to TICKET, MCS, or TAS locks achieves speedups between 29% and 50% on three of the four platforms.[11] Moreover, the cache-coherence implementation of *Opteron* proves again problematic. Due to the periodic accesses to global locks, the previously presented issues strongly manifest, resulting in a maximum speedup of 3.9. On *Westmere*, the throughput increases while all threads are running within a socket, after which it starts to decrease. Finally, thread scheduling has an important impact on

[11] The bottleneck on *SPARC-T2* is due to network and OS issues.

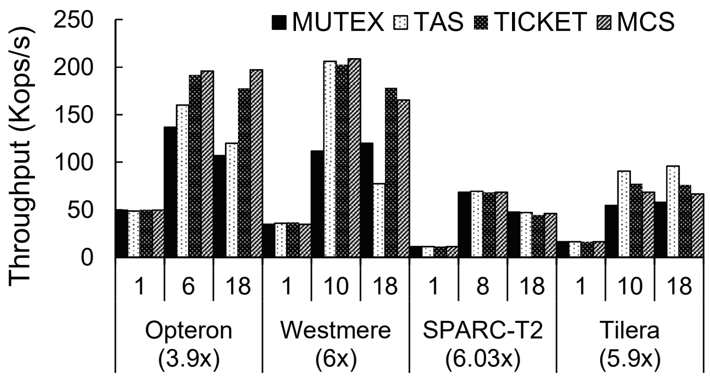


Figure 4.11 – Throughput of Memcached using a set-only test. The maximum speed-up vs. single thread is indicated under the platform names.

performance. Not allocating threads to the appropriate cores decreases performance by 20% on the multi-sockets.

Summary. Even in an application where the main limitations to performance are networking and the main memory, when contention is involved, the impact of the cache coherence and synchronization primitives is still important. When there is no contention and the data is either prefetched or read from the main memory, synchronization is less of an issue.

4.5.5 Discussion: Beyond Locks and Message Passing

In this chapter we focused on analyzing lock and message-passing-based synchronization. Arguably, these are two most commonly deployed synchronization approaches. Still, other synchronization techniques have been shown to be promising for optimizing certain coordination patterns (e.g., highly-contended critical sections). In what follows, we discuss how the results of this chapter correlate to lock-free and combiner-based synchronization.

Lock-Free Synchronization. As we discuss in Chapters 1 and 2, lock-free programming avoids using locks by directly employing hardware synchronization primitives, such as compare-and-swap. As such, the scalability of lock-free designs directly depends on the performance and scalability of atomic operations, which we thoroughly analyze in Section 4.4. For instance, crossing sockets significantly increases the latencies of atomic operations, therefore, crossing sockets will inevitably be problematic for lock-free synchronization.

Additionally, in Chapter 6, we summarize the results of our analysis of lock-free and lock-based synchronization in concurrent search data structures [51]. Our results clearly indicate that both lock-free and lock-based data structures follow the exact same scalability trends, which are of course dictated by the underlying hardware/workload combination. Regardless of the design, achieving scalability requires to minimize synchronization.

Combiner-Based Synchronization. As we discuss in Chapter 3, there is a large body of recent work on combiner-based approaches [68, 99, 136, 137, 177]. Combiners promise to improve the scalability of highly-contented critical sections by letting "server" threads to execute critical sections on behalf of other threads. In our discussion, we focus on RCL which has been shown to be the best performing combiner on several configurations [136, 137].

RCL replaces the "lock, execute, and unlock" pattern with remote procedure calls to a dedicated server core. The RCL approach hides the contention behind messages and enables the server to locally access the protected data. RCL is designed for improving the scalability of highly-contended locks. Our message-passing concurrent hash-table of Section 4.3.2 essentially implement the RCL approach.

Intuitively, for no/low contention critical sections RCL still has to pay the cost of a round-trip message for synchronization. In contrast, with locks, synchronization is typically achieved with at most one cache-line transfer. If the lock is already present in the requesting core's L1 cache (or if it is successfully prefetched), then there is no cache-line transfer. This behavior is reflected in the hash-table results (Figure 4.10): Locks are faster than message passing under low contention.

However, for highly-contended critical sections (which is the target of RCL), we indeed see that RCL can deliver much better scalability than lock-based synchronization. As we show in Figure 4.9, client-server communication delivers stable throughput regardless of the contention levels (i.e., the number of requesting threads). For RCL, this behavior translates into critical sections that do not suffer from congestion collapse as the number of threads increases.

Overall, our results corroborate that on multi-socket multi-cores, under very high contention, combiner-based approaches deliver better scalability than traditional lock algorithms.

4.6 Conclusions

In this chapter, we dissected the cost of synchronization and studied its scalability along different directions. Our analysis extended from basic hardware synchronization protocols and primitives all the way to complex concurrent software. We also considered different representative hardware architectures. The results of our experiments and our cross-platform synchronization suite, SSYNC, can be used to evaluate the potential for scaling synchronization on different platforms and to develop concurrent applications and systems. In fact, the remainder of this dissertation is heavily inspired by the synchronization study of this chapter.

Our experimentation induced various observations about synchronization on multi-cores. The first obvious one is that crossing sockets significantly impacts synchronization, regardless of the layer (e.g., cache coherence, atomic operations, locks). Synchronization scales much better within a single socket, irrespective of the contention level. Systems with heavy sharing should reduce cross-socket synchronization to the minimum. As we pointed out, this is not always possible (e.g., on the multi-socket *Opteron*), for hardware can still induce cross-socket traffic, even if sharing is explicitly restricted within a socket. Message passing can be viewed as a way to reduce sharing as it enforces partitioning of the shared data. However, it comes at the cost of lower performance (than locks) on a few cores or low contention.

Another observation is that non-uniformity affects scalability even within a single-socket multi-core—synchronization on *SPARC-T2* scales better than on *Tilera*. Consequently, even on a single-socket multi-core such as the *Tilera*, a system should reduce the amount of highly-contended data to avoid performance degradation (due to the hardware).

We also noticed that each of the nine state-of-the-art lock algorithms that we evaluated performs the best on at least one workload/platform combination. Nevertheless, if we reduce the context of synchronization to a single socket (either one socket of a multi-socket, or a single-socket multi-core), then our results indicate that simple spinlocks should be preferred over more complex locks. Complex locks (e.g., queue-based locks) have a lower uncontested performance, a larger memory footprint, and only outperform simple spinlocks under relatively high contention.

On a high-level, we showed that, roughly speaking, scalability of synchronization is largely a property of the underlying multi-core. Different platforms offer different cache-coherence implementations, atomic operations, non-uniformity profiles, intra-socket locality, etc. Consequently, optimizing synchronization of a concurrent system to the maximum is not portable, because it requires platform-specific fine-tuning.

5 An Energy Efficiency Analysis of Locking on Multi-Cores[1]

In Chapter 4, we analyzed synchronization in terms of throughput and latency. Nevertheless, the past few years energy has also become a very important factor in computing. In this chapter, we present an extensive study of the energy/performance trade-offs of lock-based synchronization on modern x86 hardware. Locks are a natural place for improving the energy efficiency of software systems. Intuitively, some locking strategies consume more power than others, thus the strategy choice can have a significant effect in concurrent systems. In summary, this chapter illustrates that improving the energy efficiency of locks goes hand in hand with improving their throughput. The main reason behind this result is that current hardware does not provide adequate tools for reducing the power consumption of synchronization without negatively impacting throughput.

5.1 Introduction

For several decades, the main metric to measure the efficiency of computing systems has been *throughput*. This state of affairs started changing in the past few years as *energy* has become a very important factor [16]. Reducing the power consumption of systems is considered crucial today [61, 92]. Various studies estimate that datacenters have contributed over 2% of the total US electricity usage in 2010 [121], and project that the energy footprint of datacenters will double by 2020 [164].

We argue that *optimizing lock-based synchronization* is an effective approach to saving energy in concurrent software. The rationale is the following. First, concurrent systems are now mainstream and need to synchronize their activities. In most cases, synchronization is achieved through locking. Hence, designing locking schemes that reduce energy consumption can affect many software systems. Second, locks are well-defined abstractions and one can usually replace the lock implementation without any modification to the rest of the system. Third, the choice of the locking scheme can have a significant effect on energy consumption. Indeed,

[1] Appeared in: Babak Falsafi, Rachid Guerraoui, Javier Picorel and Vasileios Trigonakis. "*Unlocking energy.*" USENIX ATC 2016.

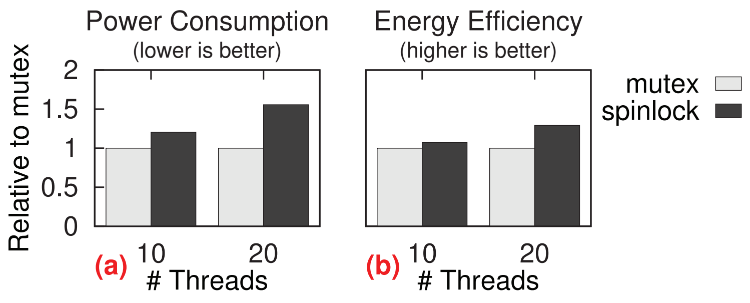


Figure 5.1 – Power consumption and energy efficiency of `CopyOnWriteArrayList` with mutex and spinlock lock algorithms.

the main consequence of synchronization is having some threads *wait* for one another—an opportunity for saving energy.

To illustrate this opportunity, consider the average power consumption of two versions of a `java.util.concurrent.CopyOnWriteArrayList` [166] stress test over a long-running execution—Figure 5.1(a). The two versions differ in how the lock handles contention: Mutexes use *sleeping*, while spinlocks employ *busy waiting*. With sleeping, the waiting thread is put to sleep by the OS until the lock is released. With busy waiting, the thread remains active, polling the lock until the lock is finally released. Choosing sleeping as the *waiting strategy* brings up to 33% benefits on power. Hence, as we pointed out, the choice of locking strategy can have a significant effect on power consumption.

Accordingly, privileging sleeping with mutex locks seems like the systematic way to go. This choice, however, is not as simple as it looks. What really matters is not only the power consumption, but the amount of energy consumed for performing some work, namely *energy efficiency*. In the Figure 5.1 example, although the spinlock version consumes 50% more power than mutex, it delivers 25% higher energy efficiency (Figure 5.1(b)) for it achieves twice the throughput. Hence, indeed, locking is a natural place to look for saving energy. Yet, choosing the best lock algorithm is not straightforward.

To finalize the argument that optimizing locks is a good approach to improve the energy efficiency of systems, we need locks that not only reduce power, but also do not hurt throughput. Is that even possible?

We show that the answer to this question is positive. We argue for the POLY[2] conjecture: *Energy efficiency and throughput go hand in hand in the context of lock algorithms.* POLY suggests that we can optimize locks to improve energy efficiency without degrading throughput; the two go hand in hand. Consequently, we can apply prior throughput-oriented research on lock algorithms almost as is in the design of energy-efficient locks as well.

[2] POLY stands for "Pareto optimality in locks for energy efficiency."

We argue for our POLY conjecture through a thorough analysis of the energy efficiency of locks on two modern Intel processors and six software systems (i.e., Memcached, MySQL, SQLite, RocksDB, HamsterDB, and Kyoto Kabinet). We conduct our analysis in three layers. We start by analyzing the hardware and software artifacts available for synchronization (e.g., pausing instructions, the Linux `futex` system calls). Then, we evaluate optimized variants of lock algorithms in terms of throughput and energy efficiency. Finally, we apply our results to the six software systems. We derive from our analysis the following observations that underlie POLY:

Busy waiting inherently hurts power consumption. With busy waiting, the underlying hardware context remains active. On Intel machines, for example, it is not practically feasible to reduce the power consumption of busy waiting. First, there is no power-friendly pause instruction to be used in busy-wait loops. The conventional way of reducing the cost of these loops, namely the `x86` `pause` instruction, actually increases power consumption. Second, the `monitor`/`mwait` instructions require kernel-level privileges, thus using them in user space incurs high overheads. Third, traditional DVFS techniques for decreasing the voltage and frequency of the cores (hence lowering their power consumption) are too coarse-grained and too slow to use. Consequently, the power consumption of busy waiting can simply not be reduced. The only way is to look into sleeping.

Sleeping can indeed save power. Our Xeon (*Ivy* in Section 2.1) server has approximately 55 Watts idle power and a max total power consumption of 206 Watts. Once a hardware context is active, it draws power, regardless of the type of work it executes. We can save this power if threads are put to sleep while waiting behind a busy lock. The OS can then put the core(s) in one of the low-power idle states [110]. Furthermore, when there are more software threads than hardware contexts in a system, sleeping is the only way to go in locks, because busy waiting kills throughput.

However, going to sleep hurts energy efficiency. The `futex` system call implements sleeping in Linux and is used by pthread mutex locks. In most realistic scenarios, the `futex`-call overheads offset the energy benefits of sleeping over busy waiting, if any, resulting in worse energy efficiency. Additionally, the spin-then-sleep policy of mutex is not tuned to account for these overheads. The mutex spins for up to a few hundred cycles before employing `futex`, while waking up a sleeping thread takes at least 7000 cycles. As a result, it is common that a thread makes the costly `futex` call to sleep, only to be immediately woken up, thus wasting both time and energy. We design MUTEXEE, an optimized version of mutex that takes the `futex` overheads into account.

Thus, some threads have to go to sleep for long. An unfair lock can put threads to sleep for long durations in the presence of high contention. Doing so results in lower power consumption, as fewer threads (hardware contexts) are active during the execution. In addition, lower fairness brings (i) better throughput, as the contention on the lock is decreased, and (ii) higher tail latencies, as the latency distribution of acquiring the lock might include some large values.

Overall, on current hardware, every power trade-off is also a throughput and a latency trade-off (motivating the name POLY): (i) sleeping vs. busy waiting, (ii) busy waiting with vs. without DVFS or `monitor/mwait`, and (iii) low vs. high fairness. The main reason for these trade-offs is that hardware does not provide adequate tools for reducing the power consumption in locks without destroying throughput.

Interestingly, in our quest to substantiate POLY, we optimize state-of-the-art locking techniques to increase the energy efficiency of our considered systems. We improve the energy efficiency or our target systems by 33% on average, driven by a 31% increase in throughput. These improvements are either due to completely avoiding sleeping using spinlocks, or due to reducing the frequency of sleep/wake-up invocations using our new MUTEXEE scheme.

We conduct our analysis on two modern Intel platforms as they provide tools (i.e., RAPL interface [111]) for accurately measuring the energy consumption of the processor. Still, we believe that POLY holds on most modern multi-cores. On the one hand, without explicit hardware support, busy waiting on any multi-core exhibits similar behavior. On the other hand, `futex` implementations are alike regardless of the underlying platform, thus the overheads of sleeping will always be significant. However, should the hardware provide adequate tools for fine-grained energy optimizations in software, POLY might need to be revised. We discuss the topic further in Section 5.7.

In summary, the main contributions of this chapter are:

- An extensive analysis of the energy efficiency of locks. The results of this analysis can be used to optimize lock algorithms for energy efficiency.
- The POLY conjecture, stating that we can simply, yet effectively optimize lock-based synchronization to improve the energy efficiency of software systems.
- Our lock libraries and benchmarks, available at: `http://lpd.epfl.ch/site/lockin`.
- MUTEXEE, an improved variant of pthread mutex lock. MUTEXEE delivers on average 28% higher energy efficiency than mutex on six modern systems.

It is worth noting that POLY might not seem surprising to a portion of the multi-core community. Yet, we believe it is important to clearly state POLY and quantify through a thorough analysis the reasons why it is valid on current hardware. As we discuss in Section 5.7, our results have important software and hardware ramifications.

The rest of the chapter is organized as follows. In Section 5.2, we describe our experimental methodology. We provide some extended details regarding our target platforms in Section 5.3 and explore techniques for reducing the power of synchronization in Section 5.4. We analyze in Section 5.5 the energy efficiency of locks and we use our results to improve various software systems in Section 5.6. We conclude the chapter in Section 5.7.

5.2 Methodology

Lock-Based Synchronization. As we describe in Chapter 2, locks ensure mutual exclusion; only the holder of the lock can proceed with its execution. The remaining threads wait until the holder releases the lock. This waiting is implemented with either *sleeping* (*blocking*), or *busy waiting* (*spinning*) [172].

In this chapter, we analyze the energy-efficiency trade-offs between sleeping and spinning. We use the pthread mutex lock (MUTEX) as our baseline sleeping lock and we consider various spinlock algorithms (i.e., TAS, TTAS, TICKET, MCS, CLH) for representing busy waiting—see Table 2.1 for a description of these algorithms.

Energy Efficiency of Software. *Energy efficiency* represents the amount of work produced for a fixed amount of energy and can be defined as *throughput per power* (TPP, $\#operation/Joule$). Higher TPP represents a more energy-efficient execution. We use the terms energy efficiency and TPP interchangeably. Alternatively, energy efficiency can be defined as the energy spent on a single operation, namely *energy per operation* (EPO, $Joule/operation$). Note that TPP = 1/EPO.

Experimental Methodology. We prefer TPP over EPO because both throughput and TPP are “higher-is-better” metrics. Recent Intel processors include the RAPL [111] interface for accurately measuring energy consumption. RAPL provides counters for measuring the cores’, package, and DRAM energy. We use these energy measurements to calculate average power. Our microbenchmark results are the median of 11 repetitions of 10 seconds. When we vary the number of threads, we first use the cores within a socket, then the cores of the second socket, and finally, the remaining (second, third, etc.) hardware threads of each core.

5.3 Power Consumption of Target Platforms

In this chapter, we use two modern Intel x86 platforms, namely *Ivy* and *Ivy-desktop* of Section 2.1. We focus on these two platforms as they both offer hardware counters for accurately measuring their energy consumption. For brevity, we only present the experimental results of our *Ivy* server. Note that the results on *Ivy-desktop* are in accordance with the ones on *Ivy*. We first provide more details about our two target platforms and then estimate their maximum power consumption.

Platforms. *Ivy* runs on frequencies scaling from 1.2 to 2.8 GHz due to DVFS and uses the Linux kernel 3.13 and glibc 2.13. *Ivy-desktop* runs on frequencies scaling from 1.6 to 3.5 GHz due to DVFS and runs the Linux kernel 3.2 and glibc 2.15. We disable Intel Turbo Boost [111].

5.3.1 Estimating Maximum Power Consumption

We estimate the maximum power that *Ivy* can consume, using a memory-intensive benchmark that consists of threads sequentially accessing large chunks of memory from their local node.

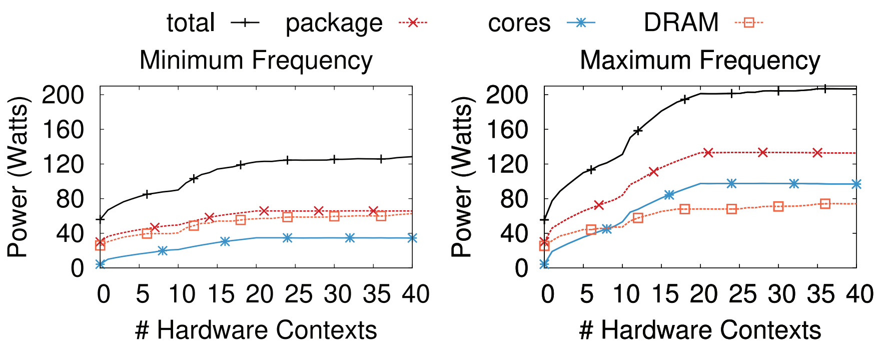


Figure 5.2 – Power-consumption breakdown on *Ivy*.

Figure 5.2 depicts the total power and the power of different components on *Ivy*, depending on the number of active hardware contexts and the voltage-frequency (VF) setting.

Idle Power Consumption. The 0-thread points represent the idle power consumption, which accounts for the static power in cores and caches, and DRAM background power, and is the power that is consumed when all cores are inactive.[3] In both min and max frequency settings the total idle power is 55.5 Watts as the VF setting only affects the active power.

Power of Active Cores. Activating the first core of a socket is more expensive than activating any subsequent due to the activation of the uncore package components. In particular, it costs 6.4 and 13.6 Watts in package power on the min and max VF settings, respectively. The second core costs 2.3 and 5.6 Watts. We perform more experiments (not shown in the graphs) with data sets that fit in L1, L2, and LLC. The results show that the package power is not vastly reduced on any of these workloads, indicating that once a core is active, the core consumes a certain amount of power that cannot be avoided.

Attribution of Power to Cores, Package, and Memory. Notice the breakdown of total power to package/core[4] and DRAM power. DRAM power has a smaller dynamic range than package and core power. On the max VF setting, DRAM power ranges from 25 to 74 Watts, while the range of package power is from 30 to 132 Watts, and core power from 4 up to 96 Watts.

Implications. The power consumption of *Ivy* ranges from 55 up to 206 Watts. Out of the 206 Watts, 74 Watts are spent on the DRAM memory. Locks are typically transferred within the processor by the cache-coherence protocol, thus limiting the opportunities for reducing power to package power (30-132 Watts). Additionally, once a core is active, the core draws power, regardless of the type of work performed. Consequently, the opportunity for reducing power consumption in software is relatively low and mostly has to do with (i) using fewer cores, by, for example, putting threads to sleep, or (ii) reducing the frequency of a core.

[3] Still, the OS briefly enables a few cores during the measurements.
[4] The package power includes the core power.

5.4 Reducing Power Consumption in Synchronization

In this section, we evaluate the costs of busy waiting and sleeping, and examine different ways of reducing them.

5.4.1 Power: The Price of Busy Waiting

We measure the total power consumption of the three main waiting techniques (i.e., sleeping, global spinning, and local spinning—see Section 5.2) when all threads are waiting for a lock that is never released. Figure 5.3 shows the power consumption and the cycles per instruction (CPI). CPI represents the average number of CPU cycles that an instruction takes to execute. CPI is typically used to show how efficiently does some software execute on hardware (*Ivy* is able to deliver as low as 0.25 cycles per instruction).

Two main points stand out. First, in this extreme scenario, sleeping is very efficient because the waiting threads do not consume any CPU resources. In this benchmark, as nothing else executes on the processor, the OS puts all cores to low-power states. Second, local spinning consumes up to 3% more power than global spinning. This behavior is explained by the CPI graph: Global spinning performs atomic operations on the shared memory address of the lock, resulting in a very high CPI (up to 530 cycles). In local spinning, every thread executes an L1 load each cycle, whereas, in global spinning, storing over coherence occurs once the atomic operation is performed, each 530 cycles on average.

5.4.2 Reducing the Price of Busy Waiting

We reduce the power consumption of busy waiting in different ways: (i) we examine various ways of pausing in spin-wait loops, (ii) we employ DVFS, and (iii) we use `monitor/mwait` to "block" the waiting threads.

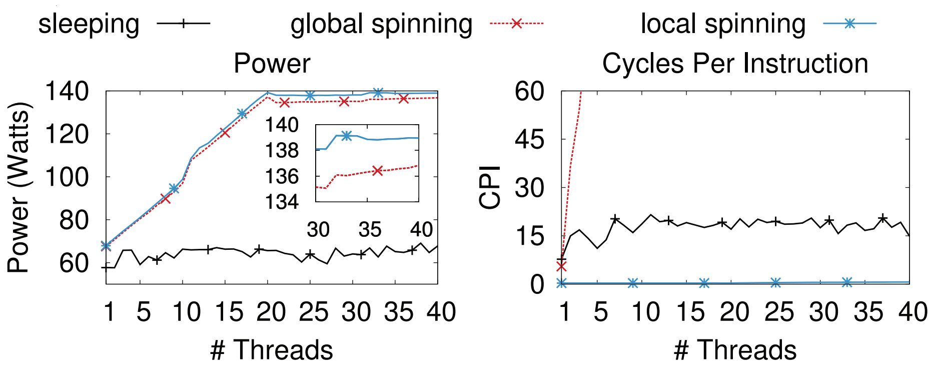


Figure 5.3 – Power consumption and CPI while waiting.

Pausing Techniques. Busy waiting with local spinning is power hungry, because threads execute with low CPI. Essentially, it is one of the rare cases in computing where the efficient execution of a piece of software on hardware is a problem. Hence, to reduce the power consumption of busy waiting, we must increase the spin-loop's CPI. We take several approaches to this end (Figure 5.4).

Any instruction, that the out-of-order core can hide, cannot reduce the power of the spin loop. For example, using the `nop` instruction to add a "bubble" in the pipeline of the core decreases CPI decreases from 0.33 to 0.25, followed by a slight increase in power consumption compared to the empty loop (not show in the graph). According to Intel's Software Developer's Manual [111], "*Inserting a pause instruction in a spin-wait loop greatly reduces the processor's power consumption.*" A `pause` (*local-pause*) increases CPI to 4.6. However, not only does it not "greatly reduce" power, but it even increases the power consumption by up to 4%. We speculate that one of the reasons for this increase in power is that `pause` gives a hint to the core to prioritize the other hardware context.

In general, the reason behind the very low CPI of local spinning is the aggressive execution mechanisms of modern processors that allow instructions to execute both speculatively and out of the program order. This results in one out of three of the retired operations being a memory load (the other two are a test and a conditional jump). Without appropriate pausing, the spin loop retires one memory load per cycle.

A way to avoid the speculative execution of the load is to insert a full, or a load, memory barrier. That way, the loads only execute once the previous load retires and the instructions that depend on it, test and jump, are stalled as well. The results (*local-mbar*) show that the barrier reduces the power consumption of local spinning to the point that becomes less expensive than global spinning (*global*). Additionally, *local-mbar* consumes up to 7% less power than *local-pause*. It is worth noting that *local-mbar* consumes less power than *local-pause* even for low thread counts (e.g., 5% on 10 threads). In the rest of the chapter, we use a memory barrier for pausing in spin loops.

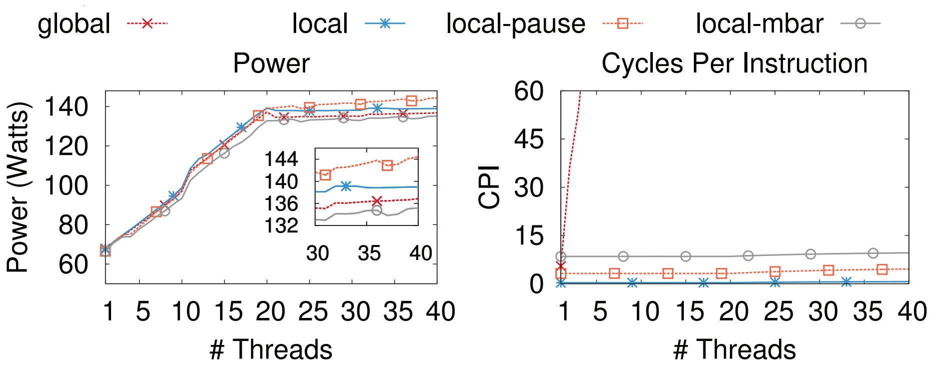


Figure 5.4 – Power consumption and CPI while spinning.

Dynamic Voltage and Frequency Scaling (DVFS). An intuitive way of lowering the power consumption of an active core is to reduce the voltage-frequency (VF) point via DVFS (see Section 5.3). Figure 5.5 shows that spinning on *VF-min* consumes up to 1.7x less power than on the *VF-max* setting. Still, DVFS is currently impractical for dynamically reducing power in busy waiting.

First, to trigger the VF change with DVFS, we need to write on a certain per-hardware context file of the `/sys/devices` directory (more details about DVFS can be found in [213]). Hence, the VF-switch operation is slow: We measure that it takes 5300 cycles on *Ivy*. If DVFS is used while busy waiting, this overhead will be on the critical path when the lock is acquired and the thread must switch back to the maximum VF point.
Second, both hardware contexts of a physical core share the same VF setting—the higher of the two. If a context lowers its VF setting, the action will have no effect unless the second context has the same or lower VF setting. Consequently, using DVFS with SMT is tricky, and as the *DVFS-normal* line shows, the power consumption drops only when both hardware contexts lower their VF points.

Monitor/mwait. The `monitor/mwait` [111] instructions allow a hardware context to block and enter an implementation-dependent optimized state while waiting for a store event. In detail, `monitor` allows a thread to declare a memory range to monitor. The hardware thread then uses `mwait` to enter an optimized state until a store is performed within the address range. Essentially, `mwait` implements sleeping in hardware and can be used in spin-wait loops: The hardware sleeps, yet the thread does not release its context.

These instructions require kernel privileges. We develop a virtual device and overload its *file_operations* functions to allow a user program to declare and wait on an address, similar to [9]. A thread can wake up others with a user-level store. However, threads pay the user-to-kernel switch and system-call overheads for waiting.

Figure 5.5 includes the power of busy waiting with `monitor/mwait`. These instructions can reduce power consumption over conventional spinning up to 1.5x. However, similarly to DVFS, using `monitor/mwait` has two shortcomings. First, `monitor/mwait` can be only used

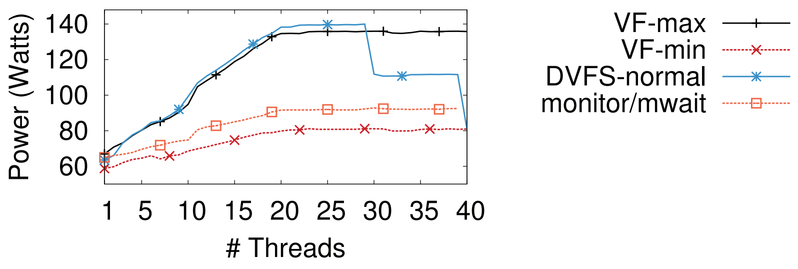


Figure 5.5 – Power consumption of busy waiting using DVFS and `monitor/mwait`.

in kernel space. The overloaded file operation takes roughly 700 cycles. The best case wake-up latency from `mwait`, with just one core “sleeping,” is 1600 cycles. In comparison, “waking up” a locally-spinning thread takes two cache-line transfers (i.e., 280 cycles). Second, programming with `monitor/mwait` on Intel processors can be elaborate and limiting. The `mwait` instruction blocks the hardware context until the thread is awaken. In oversubscribed environments (i.e., more threads than hardware contexts), `monitor/mwait` will likely exacerbate the “livelock” issues of spinlocks (see Section 5.6). Blocked threads might occupy most hardware contexts, thus preventing other threads from performing useful work.

Implications. Busy waiting drains a lot of power because cores execute at full speed. Neither of the two platforms provides sufficient tools for reducing power consumption in a systematic way. Pausing techniques, such as `pause`, can even increase the power of busy waiting. Techniques that can significantly reduce power, such as DVFS and `monitor/mwait`, are not designed for user-space usage as they require expensive kernel operations. Hence, sleeping is currently the only practical way of reducing the power consumption in locks.

5.4.3 Latency: The Price of Sleeping

In Linux, sleeping is implemented with `futex` system calls. A `futex`-sleep call puts the thread to sleep on a given memory address. A `futex` wake-up call awakes the first N threads sleeping on an address ($N = 1$ in locks). The `futex` calls are protected by kernel locks. In particular, the kernel holds a hash table (array) of locks and `futex` operations calculate the particular lock to use by hashing the address. Given that the array is large (approximately $256 * \#cores$ locks), the probability of false contention is low. However, operations on the same address (same MUTEX) do contend on kernel level.

We use a microbenchmark where two threads run in lock-step execution (synchronized at each round with barriers)—Figure 5.6. One thread makes `futex`-sleep calls, while the second thread makes wake-up calls on the same `futex`, after waiting for some time. A `futex`-sleep call (i.e., enqueuing behind the lock and descheduling the thread) takes around 2100 cycles (estimated as the required delay between sleep and wake-up calls for the wake-up calls to

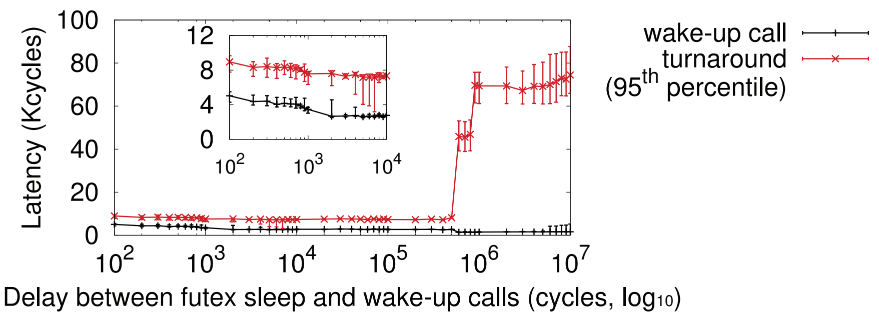


Figure 5.6 – Latency of different `futex` operations.

almost always find the other thread sleeping). This sleep latency is not necessarily on the critical path: The thread sleeps because the lock is occupied. However, the latency to wake up a thread and the one for the woken-up thread to be ready to execute are on the critical path. Figure 5.6 contains the wake-up call and the turnaround latencies, depending on the delay between the invocation of the sleep and the wake-up calls. The turnaround latency is the time from the wake-up invocation until the woken-up thread is running.[5]

The turnaround time is at least 7000 cycles and is higher than the wake-up call latency. Apart from the approximately 2700 cycles of the wake-up call, the woken-up thread requires at least 4000 more cycles before executing. Concretely, once the wake-up call finishes, the woken-up thread pays the cost of idle-to-active switching and the cost of scheduling.[6]

Figure 5.6 further includes two interesting points. First, for low delays between the two calls, the wake-up call is more expensive as it waits behind a kernel lock for the completion of the sleep call. Second, when the delay between the calls is very large (>600K cycles), the turnaround latency explodes, because the hardware context sleeps in a deeper idle state [135].

Finally, the results in Figure 5.6 use just two threads and thus represent the best-case latencies, with minimal or no contention at the kernel level. With more threads, a wake-up invocation is likely to contend with `futex` sleep calls, all serialized using a single kernel lock.

Implications. `futex` operations have high latencies and consume energy, as a non-negligible number of instructions are executed. Handing over a lock with a `futex` wake-up call requires at least 7000 cycles. Even on rather lengthy critical sections (e.g., 10000 cycles), this latency is prohibitive; it almost doubles the execution time of the critical section. In this case, the energy benefits of sleeping will not easily compensate the performance losses. In short critical sections, invoking `futex` calls will have detrimental effects on performance.

5.4.4 Reducing the Price of Sleeping

Sleeping can save energy on long waiting duration. We estimate when sleeping reduces power consumption with two threads:

Period between wake-up calls (cycles)	1024	2048	4096	8192
Power (Watts)	72.03	69.18	68.75	68.02

The first thread sleeps on a location, while the second periodically wakes up the first thread. We vary the period between the wake-up invocations, which essentially represents the critical-section duration in locks. The results confirm that if a thread is woken up more frequently than the `futex`-sleep latency, power consumption is not reduced. The thread goes to sleep only to be immediately woken up by a concurrent wake-up call. When these "sleep misses" happen, we lose performance without any power reduction. Once the delay becomes larger than the sleep latency (i.e., approximately 2100 cycles on *Ivy*), we start observing power reductions.

[5] The wake-up call latency is directly measured in our microbenchmark, while the turnaround time is estimated

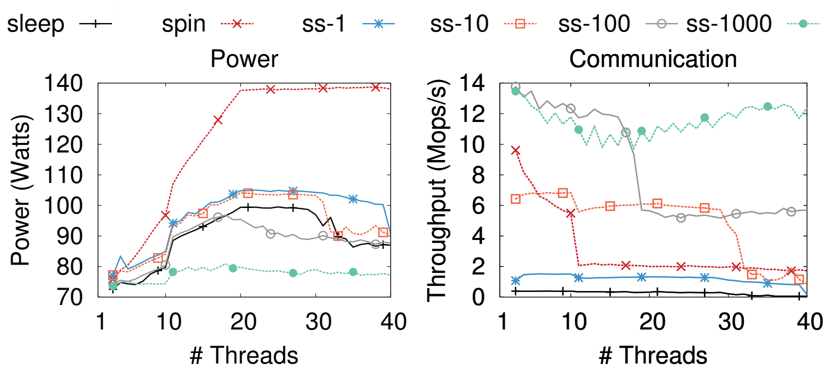


Figure 5.7 – Power consumption and communication throughput of sleeping, spinning, and spin-then-sleep for various Ts.[8]

Reducing Fairness. We show two problems with `futex`-based sleeping: (i) high turnaround latencies, and (ii) frequent sleeps and wake ups do not reduce power. To fix both problems simultaneously, we recognize the following trade-off: We can let some threads sleep for long periods, while the rest coordinate with busy waiting. If the communication is mostly done via busy waiting, we almost remove the `futex` wake-up calls from the critical path. Additionally, we let threads sleep for long periods, a requirement for reducing power consumption.

This optimization comes at the expense of fairness. The longer a thread sleeps while some others progress, the more unfair the lock becomes. We experiment with the extreme case where only two threads communicate via busy waiting, while the rest sleep. Each active thread has a “quota” T of busy-waiting repetitions, after which it wakes up another thread to take its turn. Figure 5.7 shows the power and the communication rate (similar to a lock handover) of sleeping, busy waiting, and spin-then-sleep (*ss*-T) with various Ts on a single `futex`. T is the ratio of busy-waiting over `futex` handovers.

Figure 5.7 clearly shows that the more unfair an execution—large Ts, the better the energy efficiency. First, larger Ts result in lower power, because the sleep and wake-up `futex` calls become infrequent, hence sleeping threads sleep for a long duration. For example, on 10 threads with $T = 1000$, threads sleep for about 2M cycles. In comparison, with only sleeping, the sleep duration is less than 90000 cycles. Second, spin handovers face minimal contention, as only two threads attempt to “acquire” the cache line. Consequently, because most handovers (99.9%) happen with spinning, the latency is very low, resulting in high throughput.

Implications. Frequent `futex` calls will hurt the energy efficiency of a lock. A way around this problem is to reduce lock fairness in the face of high contention, by letting only a few threads use the lock as a spinlock, while the remaining threads are asleep.

as the duration of the sleep call, reduced by the delay between the sleep and wake-up calls.

[6] When the core is active due to multiprogramming, the turnaround latency only includes the scheduling delays.

[8] The performance collapse of *spin* is due to contention, while of *ss*-10 and *ss*-100 due to the high idle-to-active switching costs (see Figure 5.6).

5.5 Energy Efficiency of Locks

We evaluate the behavior of various locks in terms of energy efficiency and throughput, relying on the results of Section 5.4. We first introduce MUTEXEE, an optimized version of MUTEX.

5.5.1 MUTEXEE: An Optimized MUTEX Lock

In Section 5.4, we analyze the overhead of `futex` calls. Additionally, we show how we can trade fairness for energy efficiency. MUTEX does not explicitly take these trade-offs into account, although it is an unfair lock.

In particular, MUTEX by default attempts to acquire the lock once before employing `futex`. MUTEX can be configured (with the `PTHREAD_MUTEX_ADAPTIVE_NP` initialization attribute) to perform up to 100 acquire attempts before sleeping with `futex`.[9] Still, threads spin up to a few hundred cycles on the lock before sleeping with `futex` (the exact duration depends on the contention on the cache line of the lock). This behavior can result in very poor performance for critical sections of up to 4000 cycles. In brief, threads are put to sleep, although the queuing time behind the lock is less than the `futex`-sleep latency. Additionally, to release a lock, MUTEX first sets the lock to "free" in user space and then wakes up one sleeping thread (if any). However, a third concurrent thread can acquire the lock before the newly awaken thread T_{aw} is ready to execute. T_{aw} will then find the lock occupied and sleep again, thus wasting energy, creating unnecessary contention, and breaking lock fairness.

To fix these two shortcomings, we design an optimized version of MUTEX, called MUTEXEE. Table 5.1 details how MUTEXEE differs from the traditional MUTEX. The "wait in user space" step of unlock requires further explanation. MUTEXEE, after releasing the lock in user space, but before invoking `futex`, waits for a short period to detect whether the lock is acquired by another thread in user space. In such case, the unlock operation returns without invoking `futex`. The waiting duration must be proportional to the maximum coherence latency of the processor (e.g., 384 cycles on *Ivy*).

[9] For brevity, in our graphs we show the default MUTEX configuration (i.e., without `PTHREAD_MUTEX_ADAPTIVE_NP`). We choose the default MUTEX version because: (i) it is the default in our systems (Section 5.6), and (ii) we thoroughly compare the two versions and conclude that for most configurations MUTEX is slightly faster without the adaptive attribute.

	MUTEX	**MUTEXEE**
lock	for up to ~ **1000 cycles** spin with `pause`	for up to ~ **8000 cycles** spin with `mfence`
	if still busy, sleep with `futex`	
unlock	release in user space (`lock->locked = 0`)	
		wait in user space
	wake up a thread with `futex`	

Table 5.1 – Differences between MUTEX and MUTEXEE.

Moreover, MUTEXEE operates in one of two modes: (i) *spin*, with ~ 8000 cycles of spinning in the lock function and ~ 384 in unlock, and (ii) *mutex*, with ~ 256 cycles in lock and ~ 128 in unlock (used to avoid useless spinning). MUTEXEE keeps track of statistics regarding how many handovers occur with busy waiting and with `futex`. Based on those statistics, MUTEXEE periodically decides on which mode to operate in: If the `futex`-to-busy-waiting handovers ratio is high (>30%), MUTEXEE uses mutex, otherwise it remains in spin mode.

Our design sensitivity analysis for MUTEXEE (not shown in the graphs) highlights three main points. First, spinning for more than 4000 cycles is crucial for throughput: MUTEXEE with 500 cycles spin behaves similarly to MUTEX. Second, the "wait in user space" functionality is crucial for power consumption (and improves throughput): If we remove it, MUTEXEE consumes similar power to MUTEX. Finally, the spin and mutex modes of MUTEXEE can save power on lengthy critical sections.

Fine-Tuning MUTEXEE. The default configuration parameters of MUTEXEE should be suitable for most `x86` processors. Still, these parameters are based on the latencies of the various events that happen in a `futex`-based lock, such as the latency of sleeping or waking up. Accordingly, in order to allow developers to fine-tune MUTEXEE for a platform, we provide a script which runs the necessary microbenchmarks and reports the configuration parameters that can be used for that platform.

Comparing MUTEXEE to MUTEX. Figure 5.8 depicts the ratios of throughput and energy efficiency of MUTEXEE over MUTEX on various configurations on a single lock. MUTEXEE indeed fixes the problematic behavior of MUTEX for critical sections of up to 4000 cycles. While MUTEX continuously puts threads to sleep and wakes them up shortly after, MUTEXEE lets the threads sleep for larger periods and keeps most lock handovers `futex` free. Of course, the latter behavior of MUTEXEE results in lower fairness as shown in Figure 5.9. Up to 4000 cycles, MUTEXEE achieves much lower 95th percentile latencies than MUTEX, because most lock handovers are fast with busy waiting. However, the price of this behavior is a few extremely

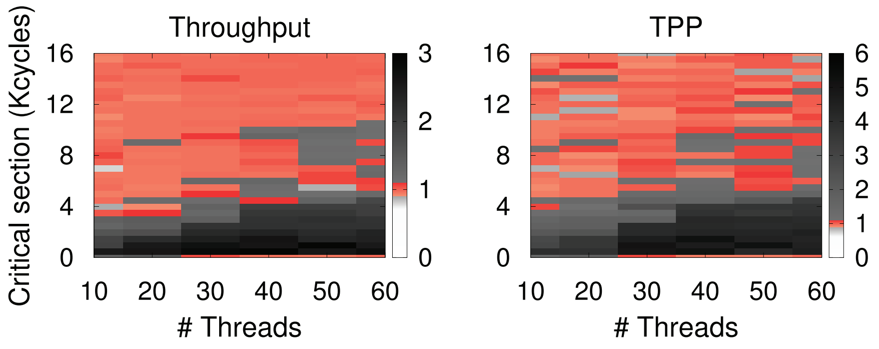


Figure 5.8 – Throughput and TPP ratios of MUTEXEE over MUTEX on various configurations with a single lock.

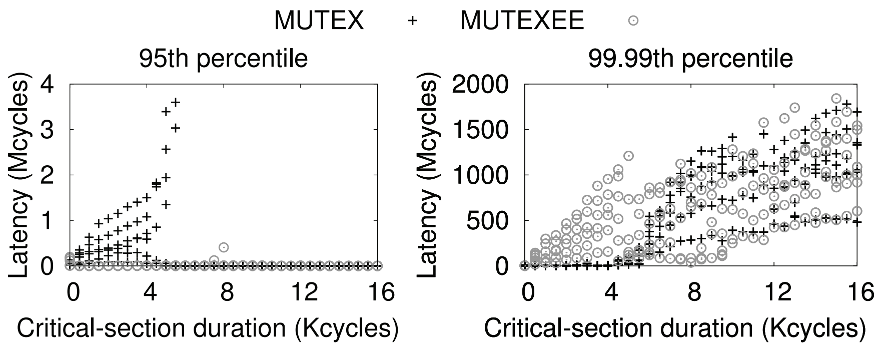


Figure 5.9 – 95/99.99th percentile latency of a MUTEX and a MUTEXEE on various configurations.

high latencies as shown in the 99.99th percentile graph. These values are caused by the long-sleeping threads and represent the trade-off between lock fairness and energy efficiency. As the critical section size increases, the behavior of the two locks converges: Both locks are highly unfair as they allow very high tail latencies (the main reason for this unfairness is that, as we show in Figure 5.6, waking up with `futex` takes a lot of time, hence the just woken up threads find the lock occupied by another thread that acquired the lock in the meantime).

Reducing MUTEXEE's Tail Latencies. MUTEXEE purposefully reduces the number of `futex` invocations by handing the lock over in user space whenever possible. Therefore, it might let some threads sleep while the rest keep the lock busy, resulting in high tail latencies. A straightforward way to limit these tail latencies, so that threads are not allowed to remain "indefinitely" asleep, is to use a timeout for the `futex` sleep call. Once a thread is woken up due to a timeout, the thread spins until it acquires the lock, without the possibility to sleep again. Of course, one can design more elaborate variants of this protocol. Controlling this timeout essentially controls the maximum latency of the lock (given that the sleep duration is significantly larger than the critical sections protected by that lock).

Figure 5.10 depicts the relative performance of MUTEXEE without over with timeouts for a single lock with 2000 cycles critical sections. For an 8 μs timeout, MUTEXEE delivers up to 14x

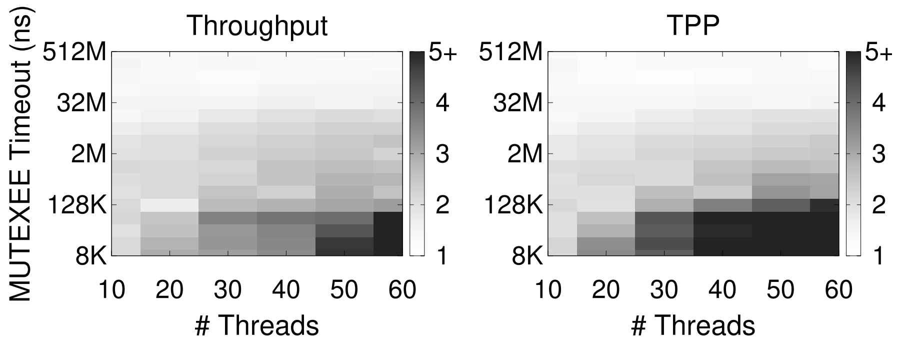


Figure 5.10 – Throughput and TPP ratios of MUTEXEE without over with timeouts.

lower throughput and 24x lower TPP than without timeouts. Threads are continuously sleeping and waking up with `futex` calls, thus significantly reducing throughput and increasing power consumption compared to MUTEXEE. In general, for timeouts shorter than 16-32 ms, both throughput and TPP suffer, representing the clear trade-off between fairness and performance.

For example, with 20 threads, MUTEXEE with a 4 ms timeout compares to the rest as follows:

Lock	**Throughput**	**TPP**	**Max Latency**
	Kacq/s	Kacq/Joule	Mcycles
MUTEX	317	4.0	2.0
MUTEXEE	855	10.9	206.5
MUTEXEE timeout	474	6.5	12.0

Depending on the application, the developer can decide whether to use timeouts and choose the timeout duration for MUTEXEE. For brevity, in the rest of the chapter, we use MUTEXEE without timeouts. As we show in Section 5.6, we do not observe significant tail-latency increases due to MUTEXEE in real systems.

5.5.2 Evaluating Lock Algorithms

We evaluate various lock algorithms under different contention levels in terms of throughput and energy efficiency (TPP).

Uncontested Locking. As we have mentioned again earlier, it is common in systems that a lock is mostly used by a single thread and both the acquire and the release operations are almost always uncontested. Table 5.2 includes the throughput (Macq/s) and the TPP (Kacq/Joule) of various lock algorithms when a thread continuously acquires and releases a single lock. We use short critical sections of 100 cycles.

The trends in throughput and TPP are identical as there is no contention. The locks perform inversely to their complexity. The simple spinlocks (TAS, TTAS, and TICKET) acquire and release the lock with just a few instructions. MUTEX performs several sanity checks and also has to handle the case of some threads sleeping when a lock is released. MUTEXEE is also more complex than simple spinlocks due to its periodic adaptation. The queue-based lock, MCS, is even more complex, because threads must find and access per-thread queue nodes.

	MUTEX	TAS	TTAS	TICKET	MCS	MUTEXEE
Throughput	11.88	16.88	16.98	16.97	12.04	13.32
TPP	174.31	248.14	249.41	249.24	176.72	195.48

Table 5.2 – Single-threaded lock throughput and TPP.

Contention – Single (Global) Lock. We experiment with a single lock accessed by a varying number of threads. This experiment captures the behavior of highly-contended coarse-grained locks. We use a fixed critical section of 1000 cycles.

Figure 5.11 contains the throughput and the TPP results. On 40 threads, MUTEX delivers 73% lower TPP than TICKET: 63% less throughput and 5.8% more power. The throughput difference is due to (i) the global spinning of MUTEX, and (ii) the `futex` calls, even if they are infrequent. The power consumption difference is mainly because of the pausing technique. MUTEX spins with `pause`, while TICKET uses a memory barrier. With `pause` instead of a barrier, TICKET consumes 4 Watts more.

Moreover, MUTEXEE maintains the contention levels and the frequency of `futex` calls low, regardless of the number of threads. This results in stable throughput and TPP because neither contention, nor the number of active hardware contexts increases with the number of threads. This behavior comes at the expense of high tail latency: On 40 threads, MUTEXEE has an 80x higher 99.9th percentile latency than MUTEX.

Regarding spinlocks, TAS is the worst in this workload. This behavior is due to the stress on the lock, which makes the release of TAS very expensive. Moreover, for up to 40 threads, the queue-based lock (MCS) delivers the best throughput and TPP. Queue-based locks are designed to avoid the burst of requests on a single cache line when the lock is released. On more than 40 threads, fairness shows its teeth. As *Ivy* has 40 hardware threads, there is oversubscription of threads to cores. TICKET and MCS, the two fair locks, suffer the most: If the thread that is the next to acquire the lock is not scheduled, the lock remains free until that thread is scheduled.

Finally, throughput and TPP are directly correlated: The higher the throughput, the higher the energy efficiency. Still, MUTEXEE delivers higher TPP by achieving both better throughput and lower power than the rest.

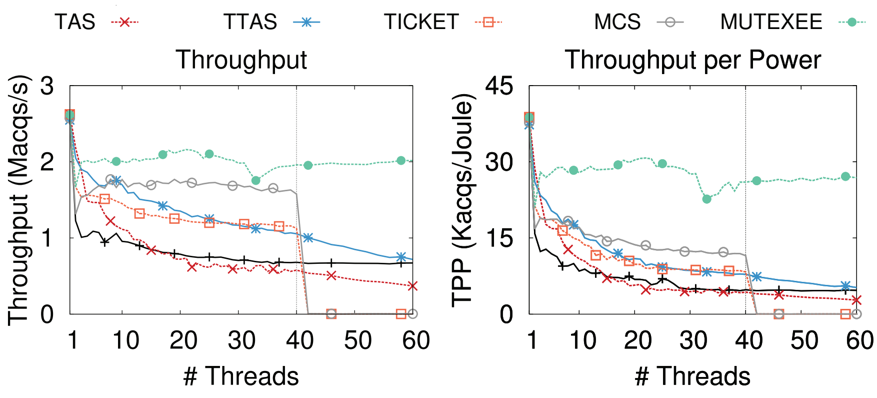


Figure 5.11 – Throughput and energy efficiency of a single (global) lock.

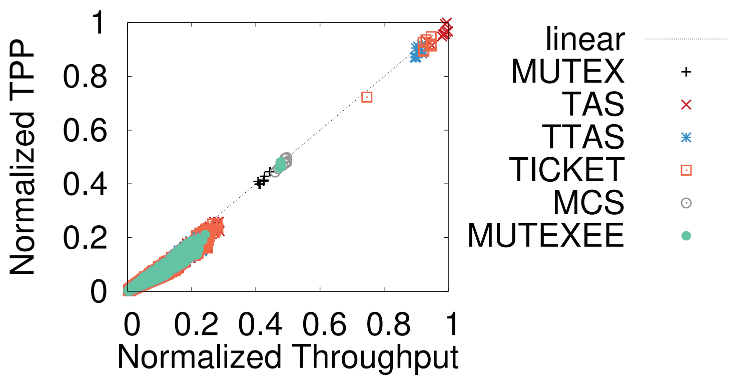


Figure 5.12 – Correlation of throughput with energy efficiency on various contention levels.

Variable Contention. Figure 5.12 plots the correlation of throughput with TPP on a diverse set of configurations. We vary the number of threads from 1 to 16, the size of critical section from 0 to 8000 cycles, and the number of locks from 1 to 512. At every iteration within a configuration, each thread selects one of the locks at random. The results are normalized to the overall maximum throughput and TPP, respectively.

Most data points fall on, or very close to, the linear line. In other words, most executions have almost one-to-one correlation of throughput with TPP. The bottom-left cluster of values represents highly-contended points. On high contention, there is a trend below the linear line, which represents executions where throughput is relatively higher compared to energy efficiency. These results are expected, as on very high contention sleeping can save power compared to busy waiting, but still, busy waiting might result in higher throughput.

If we zoom into the per-configuration best throughput and TPP, the correlation of the two is even more profound. On 85% of the 2084 configurations, the lock with the best throughput achieves the best energy efficiency as well. On the remaining 15%, the highest throughput is on average 8% better than the throughput of the highest TPP lock, while the highest TPP is 5% better than the TPP of the highest throughput lock.

Finally, MUTEXEE delivers much higher throughput and TPP than MUTEX; on average, 25% and 32% higher throughput and TPP, respectively. MUTEX is better than MUTEXEE in just 4% of the configurations (by 9% on average, both in terms of throughput and TPP).

5.5.3 Implications

The POLY conjecture states that energy efficiency and throughput go hand in hand in locks. Our evaluation of POLY with six state-of-the-art locks on various contention levels shows that, with a few exceptions, POLY is indeed valid. The exceptions to POLY are high contention scenarios, where sleeping is able to reduce power, but still results in slightly lower throughput than busy waiting on the contended locks.

For low contention levels, energy efficiency depends only on throughput, as there are no opportunities for saving energy. In these scenarios, even infrequent `futex` calls reduce both throughput and energy efficiency.

For high contention, sleeping can reduce power consumption. However, the frequent `futex` calls of MUTEX hinder the potential energy-efficiency benefits due to throughput degradation. MUTEXEE is able to reduce the frequency of `futex` calls either by avoiding the ones that are purposeless, or by reducing fairness. MUTEXEE achieves both higher throughput and lower power than spinlocks or MUTEX for high contention levels.

Our POLY conjecture also highlights that the energy efficiency of lock-based synchronization is largely dictated by the underlying multi-core hardware. First, the main reason behind POLY is that hardware does not provide adequate tools for reducing the power consumption of busy waiting without destroying throughput. Second, as energy efficiency mostly depends on throughput, our results of Chapter 4 directly apply for the energy efficiency of (lock-based) synchronization as well.

5.6 Energy Efficiency of Lock-Based Systems

In this section, we modify the locks of various concurrent systems to improve their energy efficiency. We choose the set of systems so that they use the pthread library in diverse ways, such as using mutexes or reader-writer locks, building on top of mutexes, or relying on conditionals. Note that we do not modify anything else other than the pthread locks and conditionals in these systems.

Table 5.3 contains the description and the different configurations of the six systems that we evaluate. All benchmarks use a dataset size of approximately 10 GB (in memory), except for the MySQL SSD configuration that uses 100 GB. We set the number of threads for each system according to its throughput scalability.

5.6.1 Results

Figures 5.13 and 5.14 show the throughput and the energy efficiency (TPP) of the target systems with different locks. For brevity, we show results with MUTEX, TICKET, and MUTEXEE. The remaining local-spinning locks are similar to TICKET (TAS is less efficient—see Section 5.5).

Throughput and Energy Efficiency. In 16 out of the 17 experiments, avoiding the overheads of MUTEX improves energy efficiency from 2% to 184%. On average, changing MUTEX for either TICKET or MUTEXEE improves throughput by 31% and TPP by 33%. The results include three distinct trends.

HamsterDB [191] Version: 2.1.7 # Threads: 4	An embedded key-value store. We run three tests with random reads and writes, varying the read-to-write ratio from 10% (WT), 50% (WT/RD), to 90% (RD).
Kyoto [124] Version: 1.2.76 # Threads: 4	An embedded NoSQL store. We stress Kyoto with a mix of operations for three database versions (CACHE, HT DB, B-TREE).
Memcached [70] Version: 1.4.22 # Threads: 8	An in-memory cache. We evaluate Memcached using a Twitter-like workload [133]. We vary the get-to-set ratio from 10% (WT), 50% (WT/RD), to 90% (RD). The server and the clients run on separate sockets.
MySQL [168] Version: 5.6.19	An RDBMS. We use Facebook's LinkBench and tuning guidelines [63] for an in-memory (MEM) and an SSD-drive (SSD) configurations.
RocksDB [64] Version: 3.3.0 # Threads: 12	A persistent embedded store. We use the benchmark suite and guidelines of Facebook for an in-memory configuration [65]. We run 3 tests with random reads and writes, varying the read-to-write ratio from 10% (WT), 50% (WT/RD), to 90% (RD).
SQLite [204] Version: 3.8.5	A relational DB engine. We use TPC-C with 100 warehouses varying the number of concurrent connections (i.e., 8, 32, and 64).

Table 5.3 – Software systems and configurations.

First, in some systems/configurations (i.e., Memcached and HamsterDB) sleeping can "kill" throughput. For instance, on the SET workload on Memcached, MUTEXEE allows for a few sleep invocations, resulting in lower throughput than TICKET.

Second, in some systems/configurations (i.e., MySQL and RocksDB) MUTEX is less of a problem. Both of these systems build more complex synchronization patterns on top of MUTEX. MySQL handles most low-level synchronization with custom-designed locks. Similarly, RocksDB em-

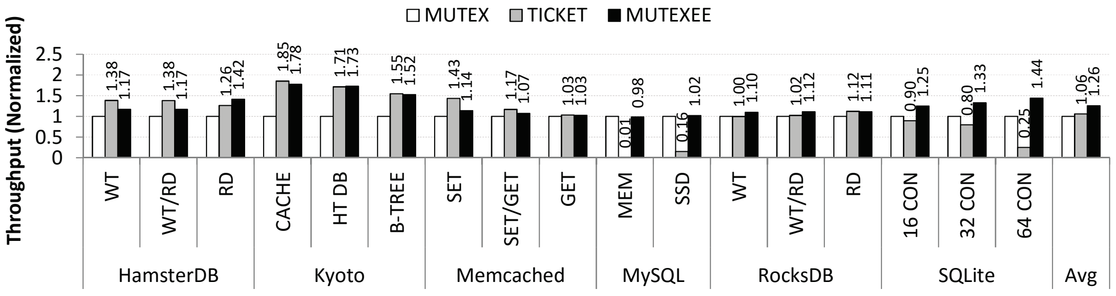


Figure 5.13 – Normalized (to MUTEX) throughput of various systems. (Higher is better)

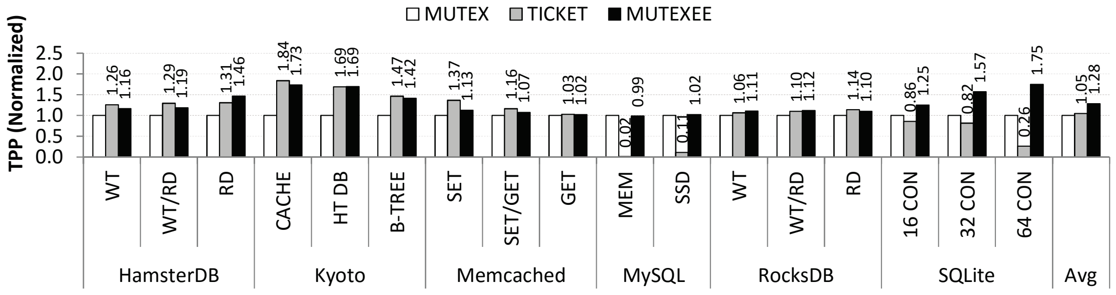


Figure 5.14 – Normalized (to MUTEX) energy efficiency of various systems. (Higher is better)

ploys a write queue where threads enqueue their operations (i.e., a combiner-based approach—see Chapter 3) and mostly relies on a conditional variable. Therefore, altering MUTEX with another algorithm does not make a big difference.

Finally, in MySQL and SQLite sleeping is necessary. Both these systems oversubscribe threads to cores, thus spinlocks, such as TICKET, result in very low throughput. A spinning thread can occupy the context of a thread that could do useful work. Additionally, on the SSD, TICKET consumes 40% more power than the other two, as it keeps all cores active. The fairness of TICKET exacerbates the problems of busy waiting in the presence of thread oversubscription: TTAS (not shown in the graph) has roughly 6x higher throughput than TICKET, but it is still much slower than MUTEX and MUTEXEE.

Overall, in five out of the six systems, the energy-efficiency improvements are mostly driven by the increased throughput. SQLite is the only system where the lock plays a significant role in terms of both throughput and power consumption. With MUTEXEE, SQLite consumes 15% and 18% less power than with MUTEX with 32 and 64 connections, respectively.

Tail Latency. MUTEXEE can become more unfair than MUTEX (see Section 5.5). Figure 5.15 includes the QoS of four systems in terms of tail latency. For most configurations, the results are intuitive: Better throughput comes with a lower tail latency. However, there are a few configurations that are worth analyzing.

First, MUTEXEE's unfairness appears in the RD configuration of HamsterDB, resulting in almost 20x higher tail latency than MUTEX, but also in 46% higher TPP. Second, TICKET has high tail latencies on all oversubscribed executions as a result of low performance.

Finally, MUTEXEE on SQLite achieves better throughput and lower power than MUTEX, without increasing tail latencies. TPC-C transactions on SQLite have latencies in the scale of tens of ms. Each transaction consists of multiple accesses to shared data protected by various locks. MUTEXEE does indeed increase the tail latency of individual locks, but these latencies are in the scale of hundreds of μs and do not appear in the transaction latencies. However,

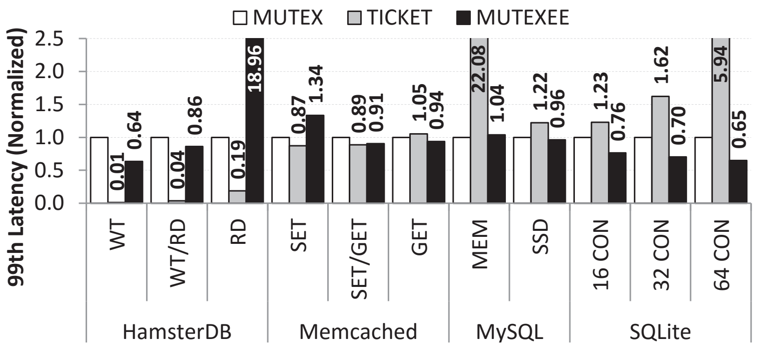


Figure 5.15 – Normalized (to MUTEX) tail latency of various systems. (Lower is better)

this low-level unfairness brings huge contention reductions. For instance, on 64 CON, the SQLite server with MUTEX puts threads to sleep for 472 μs on average, compared to 913 μs with MUTEXEE. The result is that with MUTEX, SQLite spends more than 40% of the CPU time on the `_raw_spin_lock` function of the kernel due to contention on `futex` calls. In contrast, MUTEXEE spends just 4% of the time on kernel locks, and 21% on the user-space lock functions.

Implications. Changing MUTEX in six modern systems results in 33% higher energy efficiency, driven by a 31% increase in throughput on average. Clearly, the POLY conjecture (i.e., throughput and energy efficiency go hand in hand in locks) holds in software systems and implies that we can continue business as usual: To optimize a system for energy efficiency, we can still optimize the system's locks for throughput.

Additionally, we show that MUTEX locks must be redesigned to take the latency overheads of `futex` calls into account. MUTEXEE, our optimized implementation of MUTEX, achieves 26% higher throughput and 28% better energy efficiency than MUTEX. Furthermore, the unfairness of MUTEXEE might not be a major issue in real systems: MUTEXEE can lead to high tail latencies only under extreme contention scenarios, that must be avoided in well engineered systems.

In conclusion, we see that optimizing lock-based synchronization is a good candidate for improving the energy efficiency of real systems. We can modify the locks with minimal effort, without affecting the behavior of other system components, and, more importantly, without degrading throughput.

5.7 Conclusions

In this chapter, we thoroughly analyzed the energy/performance trade-offs in lock-based synchronization in order to improve the energy efficiency of systems. Our results support the POLY conjecture: Energy efficiency and throughput go hand in hand in lock algorithms. POLY has important software and hardware ramifications.

For software, POLY conveys the ability to improve the energy efficiency of systems in an simple and systematic way, without hindering throughput. We indeed improved the energy efficiency of six popular software systems by 33% on average, driven by a 31% increase in throughput, These improvements are mainly due to MUTEXEE, our redesigned version of pthread mutex lock, that builds on the results of our analysis.

We considered the energy-efficiency trade-offs of lock-based synchronization. Nevertheless, most of our results directly or indirectly apply to other forms of synchronization. In particular, any type of waiting can be either implemented with sleeping (via `futex`, signals, interprocess interrupts, etc.) or busy waiting. For instance, thread barriers, conditional variables, and reader-writer locks essentially offer the same trade-offs as mutually exclusive locks. Similarly, lock-free synchronization frequently resolves to backoff techniques, which again falls under the same performance/energy trade-off described in this chapter.

For hardware, POLY highlights the lack of adequate tools for reducing the power consumption of synchronization, without significantly degrading throughput. We performed our analysis on two modern Intel platforms that are representative of a large portion of the processors used nowadays. We argue that our results apply in most multi-core processors, because without explicit hardware support for synchronization, the power behavior of both busy waiting and sleeping will be similar regardless of the underlying hardware.

Overall, similarly to Chapter 4, we conclude that multi-core hardware largely dictates the energy efficiency of (lock-based) synchronization. Current hardware only enables developers to optimize the throughput of synchronization: As we showed, any effort to reduce power consumption is either futile or significantly degrades throughput. Consequently, hardware (i) does not allow for power-related optimizations, and (ii) dictates the scalability of synchronization in terms of throughput and latency (per Chapter 4).

Part III

Scaling Synchronization

In Part II, we empirically showed that scalability of synchronization, in terms of throughput, latency, and energy, is mainly dictated by the underlying hardware. This finding entails that the performance portability of concurrent software across platforms is significantly hampered by synchronization: Software developers have to fine-tune synchronization for the specifics of the underlying multi-core. In this part, we show that it is still feasible to design portable and scalable concurrent software by hiding the intricacies of multi-cores behind design patterns and abstractions.

6 Designing Concurrent Data Structures with OPTIK[1]

An effective approach to abstracting synchronization away from software developers is to encapsulate data sharing with concurrent data structures. Designing and implementing fast, scalable, and portable concurrent data structures is far from trivial. This chapter introduces OPTIK, a new practical design pattern for designing and implementing portable and scalable concurrent data structures. OPTIK relies on the commonly-used technique of version numbers for detecting conflicting concurrent operations. We illustrate the power of our OPTIK pattern and its implementation by introducing five new algorithms and by optimizing four state-of-the-art algorithms for linked lists, skip lists, hash tables, and queues. Our results show that concurrent data structures built using OPTIK are more scalable than the state of the art.

6.1 Introduction

Building concurrent data structures (CDSs) in a *pessimistic* manner is easy, but typically does not lead to good performance. For example, one can design a linked list protected by a global lock in a few minutes, but inevitably, this list will be non-scalable. Accordingly, *optimistic concurrency* is deployed in every state-of-the-art data structure algorithm (e.g., lists [93, 105], hash tables [151, 196], trees [30, 161], queues [153, 159]). With optimistic concurrency, operations perform some non-synchronized work, before employing synchronization (i) for validating this optimistic work, and (ii) for possibly modifying the data structure. Performing non-synchronized work allows concurrent threads to execute truly in parallel (modern hardware is very good at executing read-only code).

Nevertheless, optimistic concurrency additionally introduces the need for validating the non-synchronized parts of the operation in order to detect conflicting concurrent operations. Validating this optimistic work is far from being trivial. Every new scalable CDS algorithm introduces a new neat technique for efficiently handling validation. Concrete examples are

[1] Appeared in: (i) Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. "*Asynchronized concurrency: The secret to scaling concurrent search data structures.*" ASPLOS 2015, (ii) Rachid Guerraoui, and Vasileios Trigonakis. "*Optimistic concurrency with OPTIK.*" PPoPP 2016.

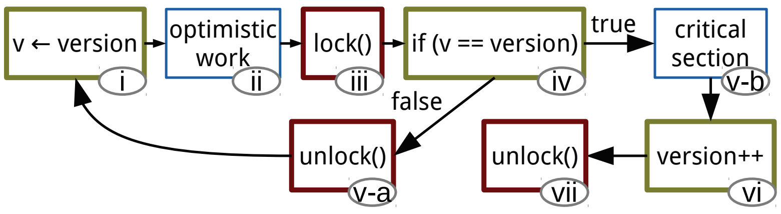


Figure 6.1 – The OPTIK pattern (high-level view).

the linked list by Tim Harris [93] that marks the least significant bit of a pointer to indicate deletions, as well as the binary search tree by Natarajan et al. [161] that marks edges instead of nodes to minimize the number of stores. These techniques are great, but are very specific to the corresponding data structure and are thus hardly generalizable to other structures.

We begin this chapter by showcasing the complexities of designing lock-based and lock-free optimistic concurrent data structures. In detail, we use the results of Chapter 4 and the ideas that we develop for asynchronized concurrency [51] to design CLHT, a concurrent hash table that places each hash-table bucket on a single cache line and performs in-place updates so that operations complete with at most one cache-line transfer. CLHT outperforms state-of-the-art hash tables in virtually every scenario. CLHT showcases that (i) we can use the results of Part II of this thesis to design scalable synchronization, and (ii) that ad-hoc validation in optimistic concurrency is indeed a very difficult problem.

Ideally, general-purpose design patterns could assist developers in creating efficient CDSs. Design patterns are commonplace in software engineering as they allow for easy and efficient solutions to recurring problems. In concurrent programming, commonly-used software constructs such as locks, semaphores, monitors, and thread pools can be viewed as design patterns. However, these patterns are very low level. Higher-level patterns are required for systematically designing and implementing efficient CDSs. Of course, as we discuss in Chapter 3, there are various techniques for simplifying the design of optimistic CDSs that could be viewed as high-level design patterns. However, all existing patterns trade programming simplicity off for performance.

In this chapter, we introduce OPTIK,[2] a new pattern for devising and implementing fast and scalable concurrent data structures. OPTIK relies on version numbers for detecting concurrency. A version number is coupled with a lock that protects a set of data (e.g., one list node). The version number has the same granularity as the lock, thus we can devise both coarse- and fine-grained algorithms with OPTIK. An optimistic operation, such as an insertion in a hash-table bucket, uses the version number in the following steps (Figure 6.1): (i) it locally stores the current value of the version in order to later use it for validation, (ii) it performs some optimistic, non-synchronized work, (iii) it grabs the corresponding lock, (iv) it validates that the version number has not changed, (v-a) if validation fails, it releases the lock and restarts

[2] The name OPTIK stands for "*optimistic concurrency with ticket locks*," as our first implementation of OPTIK locks builds on top of ticket locks [50, 51].

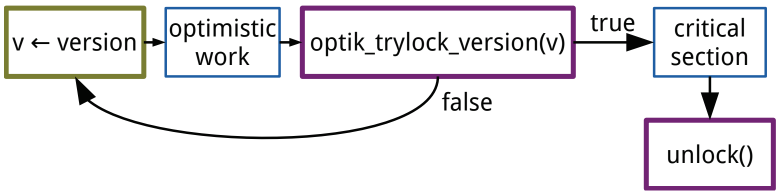


Figure 6.2 – The OPTIK pattern implemented with OPTIK locks.

the operation, otherwise (v-b) it performs the critical-section work, and then (vi) it increments the version number to indicate to other threads that the protected data has been modified, and finally, (vii) it releases the lock.

Intuitively, the validation in step (v) can fail because the version number has been incremented between steps (i) and (iii). This alteration indicates that a concurrent thread completed a modification on the protected data, rendering the optimistically accessed data inconsistent. Naturally, the reader might wonder about (a) the genuineness of OPTIK, and (b) why it has not been recognized in the past as a pattern for designing CDSs. Version numbers have been extensively used in databases [123], STMs [47, 55], and distributed systems [4, 59]. However, we are the first to recognize that the underlying idea can be expressed in a general way that offers a fast technique for detecting concurrency in CDSs. We argue that the main reason why the OPTIK pattern has not appeared in the past is the lack of an efficient implementation.

Consider the steps taken in Figure 6.1. To detect concurrency with versions, we must include the "overhead steps" (i), (iv), and (vi). To make things even worse, if validation in step (iv) fails, the thread has acquired the lock, possibly after contending for it, just to fail the validation and restart. To the best of our knowledge, most existing state-of-the-art lock-based algorithms, such as the linked-list by Heller et al. [97] and the skip list by Herlihy et al. [105], include exclusively the overhead for step (iv), namely for validating that the optimistic results are still consistent. Consequently, implementing the OPTIK pattern as described above, would not only include the same overheads as existing algorithms, but also the ones for keeping track of and incrementing the version numbers.

We solve the aforementioned limitations of the OPTIK pattern by introducing the OPTIK-lock abstraction that merges locking with validation. OPTIK locks rely on the simple observation that existing lock algorithms, such as ticket locks, employ version numbers in their implementation. Accordingly, we design the OPTIK-lock abstraction that offers an extended interface to traditional locks. In particular, OPTIK locks offer the `optik_trylock_version(lock, targetv)` function that acquires the lock iff (a) the `lock` is free, and (b) the current version in the `lock` is the same as the `targetv` version. We concretely implement the OPTIK-lock abstraction on top of ticket and versioned locks. As the unlock function of ticket locks simply increments the version, we can also merge unlocking with incrementing the version number.

Accordingly, as we show in this chapter, we can efficiently implement the OPTIK pattern using OPTIK locks (Figure 6.2). The resulting implementation guarantees that if the lock is acquired, then the critical section will be performed. Therefore, we are able to reduce contention behind the lock and to avoid the wasted work of waiting for the lock only to fail the validation. Locking and validation are performed with a single compare-and-swap. In a sense, OPTIK locks bring lock-based algorithms closer to their lock-free counterparts, where validation and the actual modifications are performed in single steps with atomic operations.

We illustrate the effectiveness of OPTIK by (a) designing new algorithms and by (b) optimizing existing state-of-the-art ones for linked lists, hash tables, skip lists, and queues. In particular, we design two new linked list algorithms, one based on global and one on fine-grained locks, and we introduce the concept of *node caching* for speeding up list traversals. Based on these lists, we design two corresponding hash tables. Additionally, we design a new concurrent array map and use it in a hash table, and we employ OPTIK locks in optimizing existing hash tables. Furthermore, we use OPTIK locks to simplify validation in the optimistic skip-list algorithm [105] and we design a novel, simple skip-list algorithm based on the OPTIK pattern. Finally, we design three variants of the classic Michael-Scott queues [153] and we also introduce the concept of *victim queues* for reducing enqueue contention. Our OPTIK-lock library, together with the data structures we design and optimize with OPTIK are available at `http://lpd.epfl.ch/site/optik`.

The main contributions of this chapter are as follows:

- We identify the OPTIK design pattern that can be used to easily design and optimize concurrent data structures.
- We introduce OPTIK locks that offer an efficient implementation of the OPTIK pattern.
- We design six new highly-efficient data-structure algorithms (CLHT and five based on OPTIK) and optimize four existing state-of-the-art algorithms.

We focus in this chapter on using OPTIK in CDSs. Nevertheless, we could imagine using OPTIK, instead of the classic lock interface, wherever a lock can be used. The only requirement is that the critical section must include a read-only prefix that can be optimistically performed before acquiring and validating the OPTIK lock. Of course, we do not claim that OPTIK is a silver bullet for all concurrency problems, but rather that it is an efficient design pattern for various use cases. For example, OPTIK locks are not very suitable for protecting large chunks of data that can be independently updated (e.g., the next pointers of a node of a large skip list). In these cases, OPTIK can lead to false validation failures due to updates on unrelated data (e.g., Section 6.5.3). Additionally, an OPTIK lock comprises a single memory location, thus, as every lock algorithm, it can become a scalability bottleneck if heavily stressed (e.g., Section 6.5.5).

The rest of the chapter is organized as follows. In Section 6.2, we introduce a new concurrent hash table and illustrate the difficulties correlated to optimistic concurrency in data structures. We describe the OPTIK pattern/lock and use them in two concrete examples in Sections 6.3 and 6.4, respectively. We then illustrate how to use OPTIK in designing and optimizing various CDSs in Section 6.5. We conclude the chapter in Section 6.6.

6.2 Optimistic Concurrency in Cache-Line Hash Table (CLHT)

In Part II, we clearly show that scalability of synchronization is impacted by the underlying hardware. This result raises the immediate question of how can we design portably scalable concurrent software. *Asynchronized concurrency* (ASCY) [51] answers this question for concurrent search data structures (CSDSs), such as lists and hash tables. ASCY comprises four guidelines on how to design CSDSs that are portably scalable (i.e., scale across workloads, performance metrics, and hardware platforms). The four ASCY guidelines are as follows:

ASCY$_1$: The search operation should not involve any waiting, retries, or stores.
ASCY$_2$: The parse phase of an update operation[3] should not perform any stores other than for cleaning-up purposes and should not involve any waiting, or retries.
ASCY$_3$: An update operation whose parse is unsuccessful (i.e., the element not found in case of a removal, the element already present in case of an insertion) should not perform any stores, besides those used for cleaning-up in the parse phase.
ASCY$_4$: The number and region of memory stores in a successful update should be close to those of a standard sequential implementation.

Essentially, ASCY suggests that CSDS designs must resemble their sequential counterparts in order to reduce synchronization to the minimum. In this section, we employ ASCY, as well as the results of Chapter 4, on the design of a new hash table algorithm, namely CLHT. For brevity, we only present the high-level ideas of CLHT and refer the reader to [50] for further details.

CLHT capture the basic idea behind the results of Chapter 4: Cache-line transfers degrade scalability, hence avoid cache-line transfers as much as possible. To this end, CLHT uses cache-line-sized buckets and, of course, follows the four ASCY patterns. As a cache-line block is the granularity of cache-coherence protocols, CLHT ensures that most operations are completed with *at most* one cache-line transfer. CLHT uses the 8 words of a cache line as:

`concurrency`	k_1	k_2	k_3	v_1	v_2	v_3	`next`

The first word is used for concurrency-control; the next six are the key/value pairs; the last is a pointer that can be used to link buckets. Updates synchronize based on the `concurrency` word and do in-place modifications of the key/value pairs of the bucket. To support in-place updates, the basic idea behind CLHT is that a search/parse does not simply traverse the keys, but obtains an *atomic snapshot* of each key/value pair:

```
val_t val = bucket->val[i];
if (bucket->key[i] == key && bucket->val[i] == val)
    /* atomic snapshot of key/value */
```

For an atomic snapshot to be possible, the memory allocator of the values must guarantee that the same address cannot appear twice during the lifespan of an operation. Additionally, the implementation has to handle possible compiler and CPU re-orderings (not shown in the

[3] The parse phase of an update operation refers to the traversal of the set towards the vicinity of the target node.

pseudo-code). CLHT supports operations with keys up to 64-bits. To support longer keys, the 64-bit keys in CLHT can be used as a first filter. The operation has to compare the full key, that is stored separately, only if there is a match with the 64-bit filter. This technique has already been shown to work well in practice [67].

We design and implement two variants of CLHT, lock-based (CLHT-LB) and lock-free (CLHT-LF).

CLHT-LB. The lock-based variant of CLHT uses the `concurrency` word as a lock. Search operations traverse the key/value pairs and return the value if a match is found. Updates (i.e., insertions and deletions) first perform a search to check whether the operation is at all feasible (recall ASCY$_3$) and if so, they grab the lock, recheck if the operation is feasible, apply the update, and release the lock. If there is not enough space for an insertion, the operation either links a new bucket by using the `next` pointer, or resizes the hash table.

CLHT-LF. The lock-free variant of CLHT is more elaborate than the lock-based, because key/value-pair insertions have to appear atomic. With locks, we implement atomicity by allowing for a single concurrent writer per bucket. However, without locks, several updates can concurrently alter the same key or value. Even worse, if concurrent insertions on the same bucket do not synchronize, there is no way to avoid duplicate keys on different slots.

In order to solve these complications, we devise the `snapshot_t` object. `snapshot_t` handles a word (8 bytes) as an array of bytes (*map*) with a *version* number:

```
struct snapshot_t {
  uint32_t version;  /* a 4-bytes integer */
  uint8_t  map[4];   /* an array of 4 bytes */
};
```

Naturally, `snapshot_t` occupies the `concurrency` word of a bucket. `snapshot_t` provides an interface to atomically get or set the value of an index in the map. The `version` number is used to enable sets/gets to do atomic changes with respect to the other spots in the map. In short, atomicity is implemented by reading the value of the `snapshot_t` object before the atomic section and by using the version number to get/set the target index in the map using a CAS on the whole object. For instance, if another concurrent insertion has already been completed, the current operation will fail the CAS, because the version number will be different. We then use the fields of the map as flags that indicate whether a given key/value pair is valid, invalid, or is being inserted. Note that, due to the 32-bit long version number, if a thread reads the version number and then "sleeps" for 2^{32} lock acquisitions, the version number could overflow, resulting in a potentially incorrect validation.

Evaluation. We compare CLHT to a concurrent hash table comprising per-bucket pointer reversal lists by Pugh [183] (*pugh*), one of the best performing hash tables (as we showed in [51]). In contrast to the linked-based hash tables, CLHT performs in-place updates, thus avoiding memory allocation and garbage collection of hash-table nodes. Nevertheless, for fairness, we use memory allocated values.

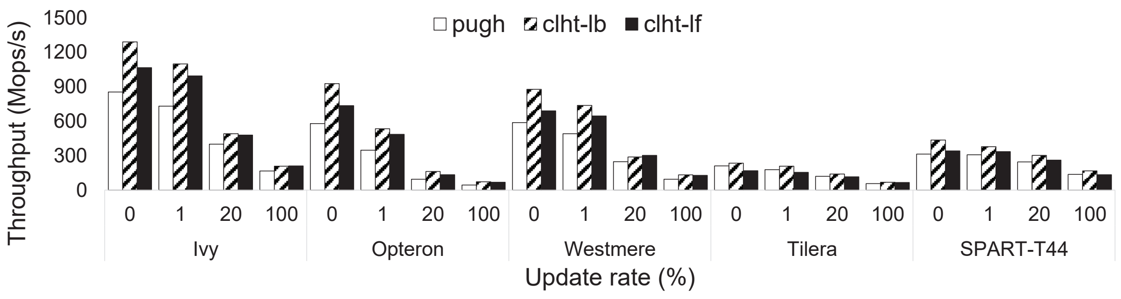


Figure 6.3 – CLHT with 4096 elements on 20 threads for various update rates.

Figure 6.3 includes the results. Noticeably, *clht-lb* and *clht-lf* outperform *pugh* by 23% and 13% on average, respectively. CLHT's design significantly reduces the number of cache-line transfers. For example, on the *Opteron* for 20% updates, *clht-lb* requires 4.06 cycles per instruction, *clht-lf* 4.24, while *pugh* operates with 6.57. Interestingly, *clht-lb* is consistently better than *clht-lf* on 20 threads. On more threads (e.g., 40), however, *clht-lf* often outperforms *clht-lb*.

6.2.1 Discussion

The CLHT algorithm clearly relies on elaborate, context specific concurrency/validation techniques: (i) bucket traversals proceed by acquiring atomic snapshots of each key/value pair, (ii) CLHT-LB traverses the bucket twice in order to validate the optimistic results after locking, (iii) CLHT-LF relies on the `snapshot_t` object for concurrency control. Evidently, these techniques combined together deliver a fast and scalable concurrent hash table. However, the aforementioned techniques are quite specific to the context of CLHT and cannot be easily generalized and reused in other algorithms. In the rest of this chapter, we detail the OPTIK pattern, which offers a generic approach to designing state-of-the-art concurrent data structures.

6.3 OPTIK

In this section, we detail the OPTIK pattern, we present the OPTIK-lock abstraction, and we describe two concrete implementations of the OPTIK-lock abstraction. We also then discuss practical considerations regarding implementing and using OPTIK, such as lock nesting, memory barriers, and memory reclamation.

6.3.1 The OPTIK Pattern

As we point out in Section 6.1, the OPTIK pattern relies on version numbers to detect potentially conflicting concurrency (see Figure 6.1). As Figure 6.4 shows, this version number is coupled with a lock and shares the same granularity as that lock (i.e., it protects the same data). The version number is incremented upon every successful critical section that modifies the shared

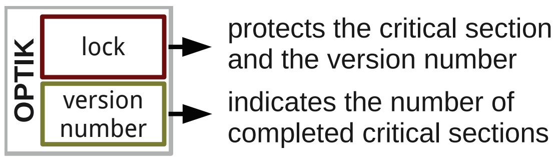

OPTIK
lock
version number
protects the critical section and the version number
indicates the number of completed critical sections

Figure 6.4 – The basic building block of the OPTIK pattern.

protected state. Thus, intuitively, we can detect whether there were concurrent modifications on the protected state if we observe a version change.

Accordingly, with OPTIK we can implement some sort of a transaction (we discuss this resemblance with transactions below), where we read the version number before starting the optimistic part of the transaction. Then, whenever we want to modify the protected data, we acquire the lock and check whether the version number is still the same. If that is the case, then no other thread could have completed a concurrent operation. Otherwise, we know that at least one thread has concurrently committed a modification.

Because the version number has the same granularity as the corresponding lock, we might have false conflicts. For example, in a linked list protected by a global lock (see Section 6.5.1), every committed modification conflicts with any concurrent one, although they might operate on completely unrelated parts of the list. In practice, in most cases we can control the granularity of the lock, hence the granularity of the version number.[4]

The OPTIK pattern has three main strengths. First, it offers a concrete way of "thinking" about optimistic concurrency, similar to STMs. With an STM, the designer makes use of transactions, but then it is up to the STM runtime to optimistically execute and coordinate these transactions. In contrast, with OPTIK, the designer must explicitly delimit the optimistic and the synchronized parts of an operation. Still, she does not need to rely on ad-hoc techniques, such as marking pointers, for validating the optimistic results. Second, the OPTIK pattern has a concrete, fast implementation based on OPTIK locks. If the pattern is appropriately employed, the resulting CDS will be efficient and scalable (as we show in Section 6.5). Third, in our experience (see Sections 6.4 and 6.5), OPTIK-based CDSs are simpler and easier to prove correct than the state of the art. In many OPTIK-based CDSs, the linearization point of an insertion or deletion is the actual write that makes a node physically linked or unlinked.

OPTIK vs. STM Transactions. The OPTIK pattern can be viewed as a transaction. OPTIK shares some common characteristics with traditional STM transactions, especially those that defer synchronization to the commit phase (e.g., [47, 55, 69]). First, they are both explicitly delimited (i.e., we know where the transaction begins and where it ends). Second, they both include an optimistic phase. Finally, the optimistic phase is followed by a validation/commit phase where conflicting concurrency is typically detected. If there are conflicts, then both OPTIK and STM transactions are restarted, otherwise they commit their modifications. For

[4] Skip lists are somewhat of an exception to this rule (see Section 6.5.3).

instance, OPTIK transactions are very similar to the ones of NOrec STM [47]. NOrec employs a global lock that is further used as a version number for validation, in a way similar to OPTIK.

However, in contrast with STMs, OPTIK does not offer isolation or atomicity guarantees. STM transactions are typically *opaque* [88] (i.e., they are serializable and they disallow even non-committed transactions from accessing inconsistent state). OPTIK allows transactions to access the intermediate results of other ongoing transactions. Additionally, STM transactions typically provide all-or-nothing semantics (i.e., *atomicity*). With OPTIK, a transaction can partially complete and then restart. The atomicity control is fully up to the programmer. Precisely because of this lack of guarantees, OPTIK can operate with zero instrumentation overhead and with minimal synchronization.

6.3.2 The OPTIK-Lock Abstraction

The OPTIK-lock abstraction merges locking with version-number validation in a single atomic step.[5] By doing so, we can implement the OPTIK pattern without the extravagant overhead of locking and then failing the validation (compare Figures 6.1 and 6.2). OPTIK locks extend the traditional lock interface with various functions. The most important ones are listed and explained below:

- `optik_trylock_version(lock, targetv)` [non-blocking]:
  acquires the `lock` iff the `lock` is free and the version of the `lock` is the same as in `targetv`. Returns a boolean indicating whether the `lock` was acquired.
- `optik_lock_version(lock, targetv)` [blocking]:
  acquires the `lock` and returns a boolean that shows whether the version that was acquired is the same as in `targetv`.
- `optik_unlock(lock)` [non-blocking]:
  increments the `lock`'s version number and releases the `lock`.
- `optik_revert(lock)` [non-blocking]:
  reverts the version of the `lock` to the one before acquiring the `lock`. It can be used to release the `lock` when no modifications were performed in the critical section.
- `optik_get_version(lock)` [non-blocking]:
  returns the current version of the `lock`.
- `optik_get_version_wait(lock)` [blocking]:
  waits until the `lock` is free and returns its current free version.
- `optik_is_same_version(v0, v1)` [non-blocking]:
  returns a boolean on whether versions `v0` and `v1` are the same.
- `optik_is_locked(v)` [non-blocking]:
  returns a boolean on whether version `v` is locked.

[5] Of course, it is up to the corresponding implementation of the abstraction to guarantee this single-step locking and validation.

We provide two implementations of the OPTIK-lock abstraction, one on top of ticket and one on top of versioned locks. For brevity, we detail the versioned-lock-based implementation (as it is simpler than the one on top of ticket locks) and discuss the additional functionality that OPTIK locks on top of ticket locks offer. In principle, the OPTIK-lock abstraction can be implemented on top of more lock algorithms. Nevertheless, `optik_trylock_version` is in the heart of the OPTIK pattern, thus we argue that every OPTIK-lock implementation must provide atomic (i.e., single compare-and-swap) locking and validation. Such an implementation requires base lock algorithms which incorporate version numbers.

OPTIK Locks Using Versioned Locks

An OPTIK lock (`optik_t`) is just an 8-byte unsigned counter (`uint64_t` in C). An odd value for the counter indicates that the lock is *locked*, while an even value means *unlocked*. The acquire function tries, until successful, to compare-and-swap (CAS) the current (even) value v with $v + 1$. The release function simply increments the counter value. Figure 6.5 includes the concrete implementation of the OPTIK abstraction on top of versioned locks. We briefly discuss this implementation.

First, `optik_trylock_version`, the most important OPTIK function, returns false (lines 6-7) if the lock is already locked or if the current lock version is not the same as the target version (`targetv`). The former check is necessary for correctness, otherwise the operation might try to erroneously CAS an odd value to an even one. The latter check is an optimization for avoiding unnecessary CAS invocations.

Similarly, `optik_lock_version` spins while the lock is locked and the tries to acquire the lock with a CAS. The unlock and revert functions increment and decrement the lock value, respectively, to indicate that the lock is now free and that a modification was (not) performed. The `optik_is_locked` function simply checks whether the given version is an odd number. The `get_version` and `get_version_wait` functions return the current version of the lock. The latter spins while the lock is locked and only then returns the version number. Finally, `optik_is_same_version` compares whether two version numbers are equivalent.

OPTIK Locks Using Ticket Locks

Ticket locks have a number of very unique properties. First, although they typically occupy just 8-bytes:

```
struct ticket_t { uint32_t ticket, current; };
```

they are fair. To acquire the lock, the thread grabs a ticket `t` with an atomic fetch-and-increment and waits until `t==lock->current`. To unlock, the owner simply increments the lock's `current` field.

```
typedef volatile uint64_t optik_t;
#define OPTIK_INIT     0
#define OPTIK_LOCKED   0x1LL //odd values -> locked

int optik_trylock_version(optik_t* l, optik_t targetv) {
  if (optik_is_locked(targetv) || *l != targetv)
      return false;
  return CAS(l, targetv, targetv + 1) == targetv;
}

int optik_lock_version(optik_t* lock, optik_t targetv) {
  optik_t ol_cur;
  do {
    do {
      ol_cur = *lock;
    } while (optik_is_locked(ol_cur));
  } while (CAS(lock, ol_cur, ol_cur + 1) != ol_cur);
  return ol_cur == targetv;
}

void optik_unlock(optik_t* lock) {
  *lock++; // mem-release
}

void optik_revert(optik_t* lock) {
  *lock--; // mem-release
}

int optik_is_locked(optik_t v) {
  return (v & OPTIK_LOCKED);
}

optik_t optik_get_version(optik_t* lock) {
  return *lock; // mem-acquire
}

optik_t optik_get_version_wait(optik_t* lock) {
  do {
    optik_t olv = *lock; // mem-acquire
    if (!optik_is_locked(olv))
      return olv;
  } while (1);
}

int optik_is_same_version(optik_t v1, optik_t v2) {
  return v1 == v2;
}
```

Figure 6.5 – Code for OPTIK locks on top of versioned locks.

Additionally, as we explain in Section 4.4.3, ticket locks show the amount of queuing behind the lock. We can use this information to implement efficient backoff schemes and to take decisions depending on the levels of contention (see Section 6.5.5 for an example). Based on these properties of ticket locks, OPTIK locks offer the `optik_num_queued` and the `optik_lock[_version]_backoff` extensions. The former returns the number of threads waiting for the lock, while the latter implements waiting with backoff that is proportional to the distance of the thread from acquiring the lock.

A shortcoming of OPTIK on top of ticket locks compared to the implementation over versioned locks is the 32-bits long version number of the former. If a thread stores the version number and then "sleeps" for 2^{32} lock acquisitions, then the version number could overflow, resulting in a potentially incorrect validation.[6] In contrast, OPTIK locks on top of versioned locks require 2^{63} acquisitions while the thread is sleeping (two increments per acquisition).

The OPTIK Pattern with and without OPTIK Locks

We illustrate the necessity of OPTIK locks with an experiment. We compare the throughput of a single OPTIK lock with the throughput of implementing version validation without OPTIK locks. As we explain earlier, to validate the version number without OPTIK locks the thread must always acquire the lock. We implement this behavior using 8 bytes; 4 bytes for a test-and-test-and-set (TTAS) lock and 4 bytes for the version number. The version number is validated and incremented while holding the lock.

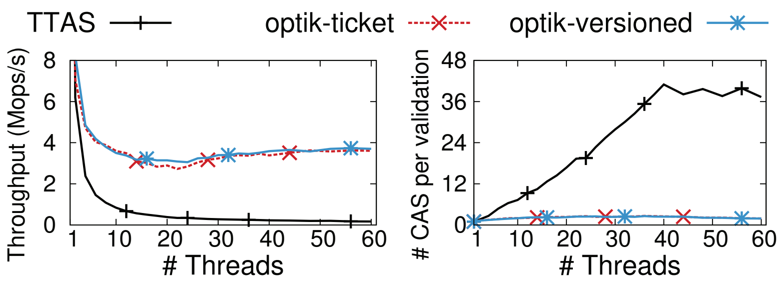


Figure 6.6 – Locking and validation with and without OPTIK locks.

Figure 6.6 depicts the validated lock-acquisition throughput with and without OPTIK locks, as well as the average number of CAS operations that are executed per successful validation on *Ivy*—an Intel Xeon server (see Section 2.1 for platform details and Section 6.5 for our experimental settings). The two OPTIK-lock implementations behave identically and deliver significantly higher throughput than validating with normal locking. OPTIK locks are more than 10 times faster than TTAS on average, explained by the number of CAS invocations per validation that grows significantly with TTAS due to lock contention.[7] As we explain earlier, without OPTIK locks the threads might wait behind the lock to later fail the validation.

[6] If the lock delivers 100M acquires/s, which is almost impossible on modern hardware (see Figure 4.3), the thread must sleep for ~40s for the overflow to happen.

[7] If we use a test-and-set lock instead of a TTAS, the number of CAS per validation "explodes."

6.3.3 Practical Considerations

OPTIK with Transactional Memory. OPTIK locks can be implemented using TM in a straightforward manner. In brief, `optik_trylock_version` can start a transaction and check if the version has been modified in order to possibly restart the operation. `optik_unlock` needs to increment the version number and commit the transaction. Given that `optik_unlock` writes on the lock, we do not expect TM to bring significant benefits to OPTIK.

OPTIK with Lock Nesting. The OPTIK pattern offers the "read then lock-validate" functionality for a single OPTIK lock. Lock nesting (i.e., acquiring and holding more than one lock at a time) requires acquiring the locks one after the other. Therefore, although the validation of an earlier lock succeeds, the validation of a later one might fail. For example, it might happen that:

$$\texttt{optik\_trylock\_version}(\texttt{l}_1,\ \texttt{v}_1) \rightarrow \texttt{true};$$
$$\texttt{optik\_trylock\_version}(\texttt{l}_2,\ \texttt{v}_2) \rightarrow \texttt{false}.$$

Depending on the semantics of the algorithm, failing the second `optik_trylock_version` can have different outcomes. For example, on the delete operation of a linked list (see Section 6.4.2 for details), failing the second trylock results in restarting the whole operation after reverting the first lock. On our novel OPTIK-based skip-list algorithm (see Section 6.5.3), we perform incremental insertions: Once the OPTIK lock for a skip-list level is acquired, the new node is linked to that level. If a subsequent trylock fails, the operation is restarted, but the locks for the already inserted levels are not reacquired.

OPTIK and Memory Fences. As we show in Figure 6.5, implementing OPTIK locks requires certain memory ordering guarantees when loading and storing on the shared word of the lock. In short, loading the version number (e.g., in `optik_get_version`) requires *acquire* semantics: No other memory access of the same thread can be reordered before this load. Similarly, storing on the memory of the lock (e.g., in `optik_revert`) requires *release* semantics: No other memory access of the same thread can be reordered after this store. Notice that on `x86` architectures the implementation of these memory-ordering semantics does not require any memory fences.

OPTIK and Memory Reclamation. OPTIK decouples concurrency control from memory reclamation. Accordingly, OPTIK can be used with practically any memory-reclamation scheme, such as hazard pointers [152], RCU [146], quiescent states [93, 95]. Our concurrent data structure implementations with OPTIK use *ssmem*,[8] a simple memory allocator with quiescent-based memory reclamation.

[8] *ssmem* is available at `https://github.com/LPD-EPFL/ssmem`.

6.4 Concrete OPTIK Examples

We illustrate in detail how to use the OPTIK pattern on two examples: (i) a map structure (abstract data type), and (ii) a novel concurrent linked-list algorithm.

6.4.1 OPTIK-Based Array Map

A map contains key-value pairs and exports the three main operations of search data structures, namely *search*, *insert*, and *delete* (see Section 2.3). We implement the map as a fixed-sized array, hence, insertions that do not find an empty spot return false (we do not employ array resizing for simplicity). In Section 6.5.2, we use our map design in a concurrent hash table.

We first briefly describe a lock-based array map that protects every operation with a global lock and then show how to optimize this array map using the OPTIK pattern.

Lock-Based Map. The design of a pessimistic, lock-based array map is straightforward: All three operations grab the lock and then traverse the array. If search or delete operations find the target key while traversing, they complete the operation (i.e., read the value of the key-value pair and, for deletions only, delete the key), unlock the lock, and return. If insertions find the key while traversing, they release the lock and return false. If they do not, they insert the new key-value pair in a free spot (if any), release the lock, and return true. If no spot is empty, insertions return false.

OPTIK-Based Map. We use the OPTIK pattern/lock to introduce optimism in the pessimistic lock-based map. Intuitively, search operations, as well as updates that return false, do not modify the data structure. Therefore, ideally, they must complete without locking. Of course, the actual insertions or deletions in the map have to synchronize for correctness.

The OPTIK pattern splits an operation into three main phases: (i) optimistic, non-synchronized (read-only) work, (ii) validation and locking, and (iii) pessimistic, synchronized (write-mostly) work. We transform the map operations to follow these phases. Figure 6.6 contains the code for the concurrent OPTIK-based array map. In what follows, we describe the code step by step.

Delete. The delete operation (Figure 6.6(a)) follows the three phases of OPTIK. It first stores the current OPTIK version number (line 9) and traverses the array without synchronization (lines 10-18), looking for the target key (line 11). If the key is not found in the array, it just returns `NULL` without ever locking (line 19). If the key is found in line 11, then the operation tries to acquire the lock using `optik_trylock_version` with the version that was earlier stored. If the validation is successful, it deletes the key, releases the lock, and returns the value (lines 13-16). If `optik_trylock_version` fails, the operation is restarted (line 12).

Insert. Insertions (Figure 6.6(b)) follow very similar logic with deletions. If the key is found while traversing the array, the operation returns false without ever acquiring the lock. If not, it tries to acquire the lock with `optik_trylock_version` and, if successful, it performs the insertion (if there is a free array spot).

```c
typedef struct {            typedef struct {
  key_t key;                  key_val_t* array;
  val_t val;                  size_t size;
} key_val_t;                  optik_t* lock;
                            } map_t;

val_t optik_map_delete(map_t* map, key_t key) {
 restart:
  optik_t vn = optik_get_version(map->lock);
  for (int i = 0; i < map->size; i++) {
    if (map->array[i].key == key) {
      if (!optik_trylock_version(map->lock, vn)) { goto restart; }
      map->array[i].key = NULL;
      val_t val = map->array[i].val;
      optik_unlock(map->lock);
      return val;
    }
  }
  return NULL;
}
```

(a) Delete operation of OPTIK-based concurrent map.

```c
int optik_map_insert(map_t* map, key, val) {
 restart:
  optik_t vn = optik_get_version(map->lock);
  int free_idx = -1;
  for (int i = 0; i < map->size; i++) {
    key_t curr_key = map->array[i].key;
    if (curr_key == key) { return false; }
    else if (curr_key == 0) { free_idx = i; }
  }

  if (!optik_trylock_version(map->lock, vn)) { goto restart; }
  int res = false;
  if (free_idx >= 0) {
    map->array[free_idx].key = key;
    map->array[free_idx].val = val;
    res = true;
  }
  optik_unlock(map->lock);
  return res;
}
```

(b) Insert operation of OPTIK-based concurrent map.

```
 1 val_t optik_map_search(map_t* map, key_t key) {
 2  restart:
 3   optik_t vn = optik_get_version_wait(map->lock);
 4   for (int i = 0; i < map->size; i++) {
 5     if (map->array[i].key == key) {
 6       val_t val = map->array[i].val;
 7       optik_t vnc = optik_get_version(map->lock);
 8       if (optik_same_version(vn, vnc))
 9         return val;
10       goto restart;
11     }
12   }
13   return NULL;
14 }
```

(c) Search operation of OPTIK-based concurrent map.

Figure 6.6 – An OPTIK-based concurrent array map data structure.

Search. We want the search operation to not acquire the lock, otherwise, the total throughput of the map will be dictated by the maximum lock throughput. Nevertheless, we must guarantee the atomicity of reading key-value pairs. In other words, we have to ensure that between matching an array key with the target key and reading the value, there was no concurrent modification on this key-value pair.

We achieve this guarantee using the OPTIK version number. The search operation (Figure 6.6(c)) reads the version number in the beginning of the operation (line 3), like update operations do. This time, however, we employ the `optik_get_version_wait` function that blocks until the lock is free. Once the key is matched (line 5), we read the corresponding value and check whether the version has changed (lines 6-8). If it did change, then the operation is restarted, otherwise the value is returned. The reason for acquiring an unlocked version in line 3 is that we need to ensure that the search operation was not concurrent with any update operations on the same key during the execution of lines 5-6.

We could decrease the "granularity" of the version number for search operations, by reading the version before line 5. We would still be able to acquire atomic snapshots of the key-value pairs. However, doing so puts a lot of stress on the cache line of the OPTIK lock, resulting in lower performance than the design in Figure 6.6. (Actually, we can devise various schemes for validating the key-value snapshot using the version number.)

In terms of correctness, successful updates are serialized behind the lock. Successful searches are trivially correct, as they complete iff there were no concurrent modifications. Unsuccessful search operations can be linearized so that they never observe the target key. Similarly, unsuccessful updates can be linearized so that they do (not) observe the target key.

Lock-Based vs. OPTIK-Based Array Map. We compare the two map implementations on two workloads on an *Ivy* (see Section 2.1 see for platform details and Section 6.5 for our experimental settings). Figure 6.7 depicts the results, where MCS represents the lock-based map protected by an MCS lock. On both the small and the large maps, the OPTIK version (*optik*) is faster than the lock-based one. *optik* has two main benefits compared to MCS. First, search operations (80% of the workload) do not acquire the lock. Second, unsuccessful updates (~10% of the operations) also do not need to synchronize.

If we exclude the results on multiprogramming (i.e., more threads than hardware contexts), where MCS suffers, *optik* is on average 4.7 and 1.4 times faster than MCS on the small and the large map, respectively. On the small workload, since there are just four spots in the map array, many operations fail (e.g., deletions do not find the key they are looking for). For example, on 10 threads, only 25% of the updates are successful, resulting in 5% total effective updates. Overall, the results can be largely explained by the latency distributions. *optik* significantly reduces the latencies for search operations and for unsuccessful deletions. The reduction is less profound on unsuccessful insertions, as a portion of those failures is due to insufficient space in the array. In these cases, both *optik* and MCS acquire and release the lock before returning false. Additionally, the effects of failing `optik_trylock_version` and restarting are visible on the tail latencies of successful updates.

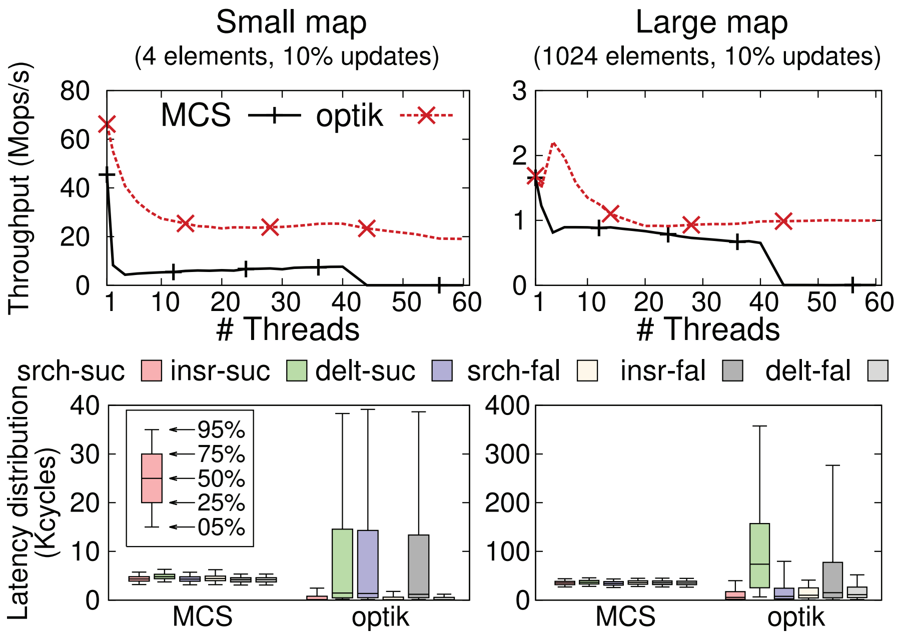


Figure 6.7 – Lock-based vs. OPTIK-based map. The latency-distribution results use 10 threads.

6.4.2 OPTIK-Based Linked List

The main idea behind a sorted OPTIK-based linked list is to keep track of the necessary version numbers while traversing the list. In a sense, similarly to hand-over-hand locking [103], the OPTIK-based list performs hand-over-hand version tracking. Figure 6.8 includes the code of our implementation. We defer the evaluation of our list to Section 6.5.

Delete. The delete operation (Figure 6.8(a)) is the most complex operation of the OPTIK-based linked list, because it requires locking two nodes; the one being deleted and its predecessor node. Traversing the list (lines 11-14) keeps track of these two version numbers that are later used for locking with `optik_trylock_version` (lines 17-20). If locking the predecessor node fails, the operation is restarted, otherwise the node to be deleted is locked. If this latter `optik_trylock_version` fails, the predecessor's OPTIK lock is reverted, instead of unlocked, in order to avoid false conflicts with other concurrent operations. Notice that due to OPTIK, (i) no *deleted* flag is required (as in [97]), and (ii) the OPTIK lock of the deleted node is never released, which prohibits updates from reusing this node. Essentially, the linearization point of a deletion is the actual write on the `pred->next` pointer in line 21.

```
 1 typedef struct node {        typedef struct {
 2   key_t key; val_t val;        node_t* head;
 3   optik_t lock;              } ll_t;
 4   struct node* next;
 5 } node_t;

 7 val_t optik_ll_delete(ll_t* list, key_t key) {
 8  restart:
 9   node_t *pred, *cur = list->head;
10   optik_t predv = curv = optik_get_version(&cur->lock);
11   do {
12     pred = cur; predv = curv; cur = cur->next;
13     curv = optik_get_version(&cur->lock);
14   } while (cur->key < key);
15   if (cur->key != key) { return NULL; }

17   if (!optik_trylock_version(&pred->lock, predv)) { goto restart; }
18   if (!optik_trylock_version(&cur->lock, curv)) {
19     optik_revert(&pred->lock); goto restart;
20   }
21   pred->next = cur->next;
22   val_t result = cur->val;
23   optik_unlock(&pred->lock);
24   node_gc_free(cur);
25   return result;
26 }
```

(a) Delete operation of OPTIK-based concurrent linked list.

```
 1 int optik_ll_insert(ll_t* list, key, val) {
 2  restart:
 3   node_t *pred, *cur = list->head;
 4   optik_t predv = curv = OPTIK_INIT;
 5   do {
 6     curv = optik_get_version(&cur->lock);
 7     pred = cur; predv = curv; cur = cur->next;
 8   } while (cur->key < key);
 9   if (cur->key == key) { return false; }

11   if (!optik_trylock_version(&pred->lock, predv)) { goto restart; }
12   node_t* newnode = new_node(key, val, cur);
13   pred->next = newnode;
14   optik_unlock(&pred->lock);
15   return true;
16 }
```

(b) Insert operation of OPTIK-based concurrent linked list.

```
val_t optik_ll_search(ll_t* list, key_t key) {
  node_t* cur = list->head;
  while (cur->key < key) { cur = cur->next; }
  if (cur->key == key)  { return cur->val; }
  return NULL;
}
```

(c) Search operation of OPTIK-based concurrent linked list.

Figure 6.8 – An OPTIK-based linked-list data structure.

Insert. Inserting in the OPTIK-based linked list (Figure 6.8(b)) requires locking and validating only the predecessor node (line 11). This OPTIK lock ensures that there are no concurrently completed modifications on the predecessor node `p` or on `p->next`.

Search. The search operation (Figure 6.8(c)) of the OPTIK-based linked list is completely oblivious to concurrency. We can support this 100% sequential search design because the linearization points of updates are the actual stores on the predecessor node.

6.5 OPTIK in Concurrent Data Structures

In this section, we illustrate the power and usefulness of OPTIK for optimizing and designing concurrent data structures (CDSs) (i.e., linked lists, hash tables, skip lists, and queues). In contrast to Section 6.4, we keep the CDS descriptions high-level for brevity. Before that, we describe the evaluation settings that we use in our experiments and the two platforms that we evaluate our data structures on.

Experimental Methodology. We evaluate various algorithms via microbenchmarks. Unless stated otherwise, all OPTIK implementations use OPTIK locks on top of versioned locks. Similarly, unless stated otherwise, non-OPTIK implementations use test-and-set locks. (Notice that for highly-contented locks, such as the locks in concurrent queues, we use MCS locks.) We take the non-OPTIK algorithm implementations from our ASCYLIB library [51] (we use the optimized versions of the algorithms). Additionally, we use the memory allocator of ASCYLIB that provides garbage collection and we use 8-byte long keys and values. Backoff schemes can significantly affect the performance of CDSs (e.g., when an operation fails and must be restarted). For fairness, all data structures use the exact same backoff function. We use exponentially increasing backoff times with up to 16k cycles maximum backoff. Furthermore, after every iteration, threads wait for a short duration, in order to avoid long runs [153].

On every run, we set the initial size of the data structure and the key range that the threads operate on. On every iteration, each thread selects a key at random within the given range. We keep the range double the initial size and the percentages of insertions and deletions the same, so that the size of the structure remains close to the initial. Because the key range is double the initial, roughly half of the update operations on search data structures return false. The update rate that we report on the graphs represents the effective percentage of updates, namely the ones that alter the data structure. For our skewed workloads, we use a zipfian distribution of keys with $a = 0.9$, where the largest keys are the most popular. Our results are the median value of 11 repetitions of 5 seconds each. We do not pin threads to cores, but let the OS do the scheduling.

For our latency measurements, we use the the per-core timestamp counter [111] for accurately measuring the duration of an operation in cycles. In detail, every thread holds an array of 16K latency measurements that, in the end of each experiment, are collected and translated to latency distribution (boxplots reporting 5^{th}, 25^{th}, 50^{th}, 75^{th}, and 95^{th} percentile latencies).

We use the *Ivy* and *Opteron* multi-core processors, described in Section 2.1.

6.5.1 OPTIK in Linked Lists

We use OPTIK in the design of concurrent (sorted) linked lists. The simplest algorithm is of course a sequential list protected by a scalable global lock, such as an MCS lock. Naturally, this algorithm does not offer any concurrency as all operations are serialized behind the lock. An easy optimization on the global-lock algorithm is to implement the search operation so that it does not acquire the lock (given that memory reclamation is properly handled). The linearization point of updates is then the actual memory writes that access the predecessor node of the one being updated.

Nevertheless, updates are fully serialized behind the global lock, resulting in low scalability. We use OPTIK to introduce optimism to the update operations. The transformation is very similar to that of the concurrent map in Section 6.4.1. Note that concurrent modifications might not be conflicting, still, using a global lock will result in false conflicts. Because of this limitation and of the high load on the global lock, this linked-list design is not expected to scale well on contended scenarios. We can resolve these limitations using fine-grained locking (see Section 6.4.2 for the design of the fine-grained OPTIK-based linked list).

Additionally, inspired by the fact that version numbers reveal whether a list node has been modified, we develop the idea of *node caching*. In short, each thread keeps track of the last accessed node after each operation, accompanied by the version number that the thread observed. This node can be subsequently used as the entry point for the next operation on the list, given that (i) it has not been deleted, and (ii) it is a correct entry point (i.e., in a sorted list, the key of the cached node is less than the target key). Of course, we must ensure that the memory of deleted nodes is not re-used while the node is still referenced by any node cache. Node-caching can be also applied on non-OPTIK algorithms, given that we can avoid the ABA problem and that we can detect whether a node is valid.

Correctness. The OPTIK-based global-lock list is trivially correct as it disallows concurrency of modifications. The linearization point of both insertions and deletions can be set to the actual write on the predecessor's next pointer. Search operations either observe the concurrent modifications in the vicinity of the target key, or not.

Evaluation. Figure 6.9 depicts the throughput of the aforementioned linked-list algorithms on various workloads. For comparison, we include the results of the lazy linked-list algorithm [97] (*lazy*), that we have shown to be very efficient [51], as well as the lock-free list by Harris [93] (*harris*). We implement the node-caching idea on the lazy list (*lazy-cache*) and on the fine-grained OPTIK-based list (*optik* and *optik-cache* in the graph). *mcs-gl-opt* represents a global-lock list protected by an MCS lock, including the non-synchronized search optimization we describe earlier.

Clearly, the node-cache optimization (*optik-cache, lazy-cache*) brings important performance benefits as it probabilistically reduces the list-traversal duration. For instance, on the large list, 49.8% of the operations make use of the node cache, while on the small list the hit rate drops

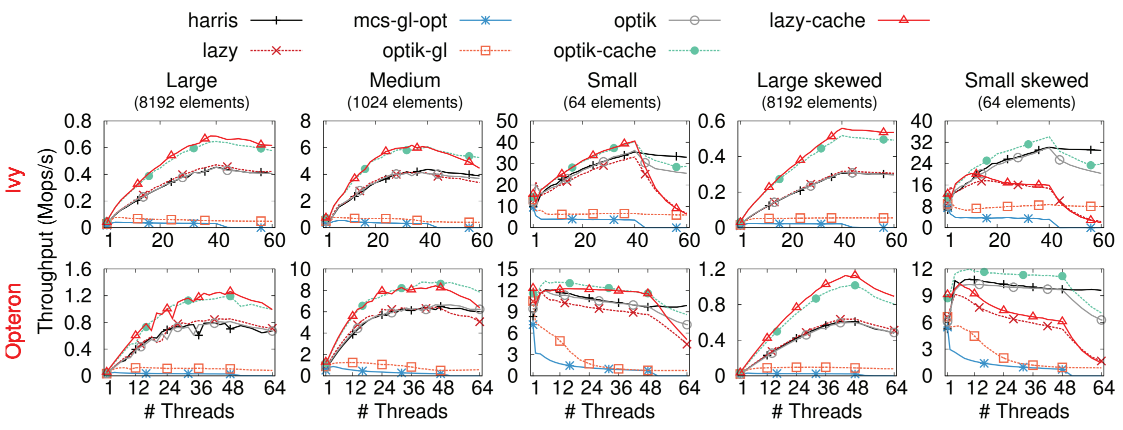


Figure 6.9 – Throughput of linked-list algorithms on *Ivy* and *Opteron* on various workloads.

to approximately 40%. On these two workloads, *optik-cache* delivers 50% and 15% higher average throughput than the version without the cache (*optik*).

Additionally, the OPTIK-based global-lock list (*optik-gl*) delivers higher throughput than *mcs-gl-opt* in all workloads. *optik-gl* mostly benefits from the fact that for 20% of the operations—the unsuccessful ones—it returns without acquiring the lock.

Finally, the fine-grained OPTIK-based list (*optik*) performs similarly to *lazy* and *harris* for the low-contention workloads (i.e., large, large-skewed, and medium). However, *optik* is more scalable than lazy on high-contention levels. On 64 elements, *optik* is on average 22% faster than *lazy*. Note that *optik* stresses the locks less than *lazy*, because the operations do not acquire the lock if they are going to fail the validation. This difference is clear on the small-skewed workload, where neither *lazy*, nor *lazy-cache* can sustain the contention of the highly-contented nodes.[9] Additionally, *optik* behaves much better than *lazy* on multiprogramming and is, on average, just 5% slower than *harris* even on the small workloads.

6.5.2 OPTIK in Hash Tables

We adapt and use the two OPTIK-based linked lists (Section 6.4.2, Section 6.5.1) in the design of two novel hash tables. Intuitively, the list protected by a global lock, resulting in per-bucket locking, is more suitable for hash tables. We also use the array map of Section 6.4.1 in the design of a third hash table.

We further use OPTIK locks to optimize existing hash tables. In a hash table, an update operation (i.e., an insertion or a deletion) might not be feasible: Delete (resp. insert) operations return false if the corresponding key is not found (resp. is found). Many hash-table algorithms (e.g., Java `ConcurrentHashMap` [127]) implement updates by directly locking the corresponding bucket, regardless if the operation is feasible. This unnecessary locking hinders scalability. In these algorithms, in order to return false without locking if an update is not feasible, we must add an extra read-only traversal of the bucket. If the operation cannot be performed, no lock is acquired and the operation simply returns false after this first traversal. Otherwise, if the operation can be performed, we must acquire the bucket lock and then re-traverse the bucket to ensure that no concurrent modification operated on the target key. Consequently, for every successful update, we have two traversals of the bucket. We can avoid the second traversal with OPTIK locks, using either `optik_lock_version` or `optik_trylock_version`. In the beginning of the operation, we keep track of the version number of the bucket and use this version in the OPTIK-lock call. If the version is validated, no concurrent modification has completed on this bucket, hence we do not need to re-traverse the bucket.

[9] The most contented node is accessed by 15% of the requests.

Correctness. The three hash tables that are based on the two OPTIK lists and the map are correct because of the correctness of these base data structures. The optimizations for avoiding double traversal with OPTIK are correct because the bucket cannot be modified without increasing the version number of the bucket lock.

Evaluation. Figure 6.10 includes the results of various hash tables. We set the number of buckets to be equal to the number of initial elements, so that initially every bucket contains on average one element. For brevity, we only show the results with medium and small-skewed sized hash tables. On the missing graphs, the behavior of the hash tables is in accordance with the results shown in Figure 6.10. Apart from the three OPTIK-based hash tables (*optik*, *optik-gl*—for per-bucket locking, and *optik-map*), we create a hash table with lazy linked lists adapted to use per-bucket locking (*lazy-gl*). Additionally, we evaluate Java's `util.concurrent.ConcurrentHashMap` [127] (*java*), as well as a modified version that avoids double parsing using `optik_trylock_version`, as we describe above. The `ConcurrentHashMap` algorithm uses lock striping: It partitions the buckets into n segments. Each segment (and its buckets) is protected by a single lock and can be individually resized. We configure n to be 128, based on Java's documentation [170] "*Ideally, you should choose a value to accommodate as many threads as will ever concurrently modify the table.*"

Optimizing *java* with OPTIK (*java-optik*) brings benefits only in the presence of (high) contention. On the large hash table (65536 elements—not shown in the graph), the improvement is just 1.9%, because there are practically no validation failures. Additionally, the second bucket pass of *java* is very fast, as the first pass brings the bucket data in the L1 cache.

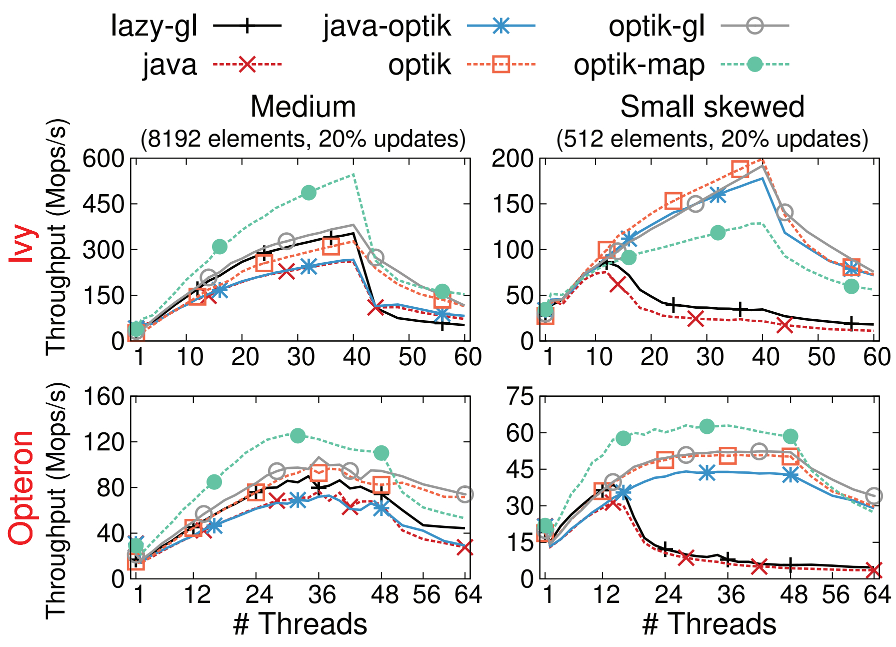


Figure 6.10 – Throughput of hash-table algorithms on *Ivy* and *Opteron* on various workloads.

Furthermore, *optik-map* does not scale well on the small workloads on *Ivy* due to the hardware. In brief, the buckets of *optik-map* are allocated in consecutive memory locations, thus occupying a few contiguous cache lines, resulting in increased hardware prefetching on *Ivy* in our experiments. For example, on 20 threads, the small hash table triggers three orders of magnitude more last-level-cache prefetches than the medium one. This inaccurate prefetching leads to low scalability due to high coherence traffic. Once the size of the hash table is large enough, *optik-map* becomes the fastest hash table on both platforms. The other hash tables do not face the aforementioned problem, because they dynamically allocate each node that is inserted in the hash table.

Regarding the remaining three hash tables, *optik-gl* is the fastest. *optik-gl* is 2-times faster than *lazy-gl* on average (31% faster on the non-skewed workloads). *optik* is on average 9% slower than *optik-gl*, as for some operations *optik* acquires two locks instead of the one lock in *optik-gl*. On the small-skewed workload, we see the power of the OPTIK pattern compared to normal locking: *optik-gl* and *optik* are both 3.7-times faster than *lazy-gl* on average. Even on the large-skewed workload (not in the graph), *lazy-gl* is on average more than 2-times slower than the OPTIK-based hash tables.

6.5.3 OPTIK in Skip Lists

In theory, OPTIK is not very suitable for skip lists. With per-node lock granularity, the same version protects all the next pointers of the node. Consequently, validating the node with OPTIK results in false conflicts. Still, using OPTIK in skip lists results in simpler designs than the existing state-of-the-art ones [75, 105]. We first simplify validation in the optimistic skip list by Herlihy et al. [105], using `optik_lock_version`. If the validation is successful, then the corresponding node has not been modified, thus we do not need to validate the optimistic results in another way. This specific skip lists checks that the node is not logically deleted and that the next pointer at the corresponding level has not been altered.

We also use OPTIK in the design of a new skip-list algorithm. As in any skip list, update operations parse the list and keep track of the predecessor and successor nodes at each level. Due to OPTIK, parsing also keeps track of the version number of each predecessor node. These version numbers are later used for validation. Once the parsing finds the spot to modify, it locks and validates the predecessor nodes and then performs the modifications. If the validation fails, the locks are released and the operation is restarted. We implement two variants of the OPTIK-based skip list. The first one, in case validation fails, performs more fine-grained validation (same one as in [105]). The second one immediately restarts the operation if an OPTIK validation fails.

Correctness. The modified Herlihy skip list maintains the correctness of the initial algorithm. Our modifications only involve reducing validation in case the `optik_lock_version` function is able to validate the previously observed version.

For brevity, we only describe the correctness sketch of the OPTIK-based skip list that immediately restarts on a trylock failure. Both insertions and deletions traverse the list and keep track of the predecessor nodes and their version at each level. As the OPTIK lock protects the whole predecessor node `p`, we do not need to keep track of the successor nodes for validating `p->next`. Insertions try to acquire the lock and perform the insertion of the new node eagerly (i.e., they do the physical linking of the node immediately after acquiring the lock of that level). If an `optik_trylock_version` call fails, the operation is restarted and, after re-parsing the list, the insertion continues from the level that failed. A flag, similar with the *fullylinked* flag in Herlihy skip list, ensures that a partially inserted node will not be concurrently deleted. Similarly, a deletion atomically sets the flag of the target node to *deleted* and unlinks the node after acquiring all predecessor locks. We can devise a variant of the algorithm where deletions proceed progressively like insertions. However, the coordination overhead between insertions and deletions on the same node surpasses the benefits of being eager.

Evaluation. Figure 6.11 compares the Herlihy skip list (*herlihy*), and the lock-free one by Fraser [75] (*fraser*), with the three lists that we describe above. For brevity, we only show the results on large-skewed and small-skewed lists. On low-contention levels (large, medium non-skewed—not shown in the graph), all algorithms behave similarly. Intuitively, all five implementations follow almost identical code paths in the absence of conflicts: Most of the time is spent traversing the list.

Using `optik_lock_version` in the Herlihy skip list (*herl-optik*) slightly affects the performance on the *Opteron*, but has a large effect on *Ivy*. In brief, the faster validation with OPTIK results in an important reduction of operation restarts. For instance, on the small-skewed

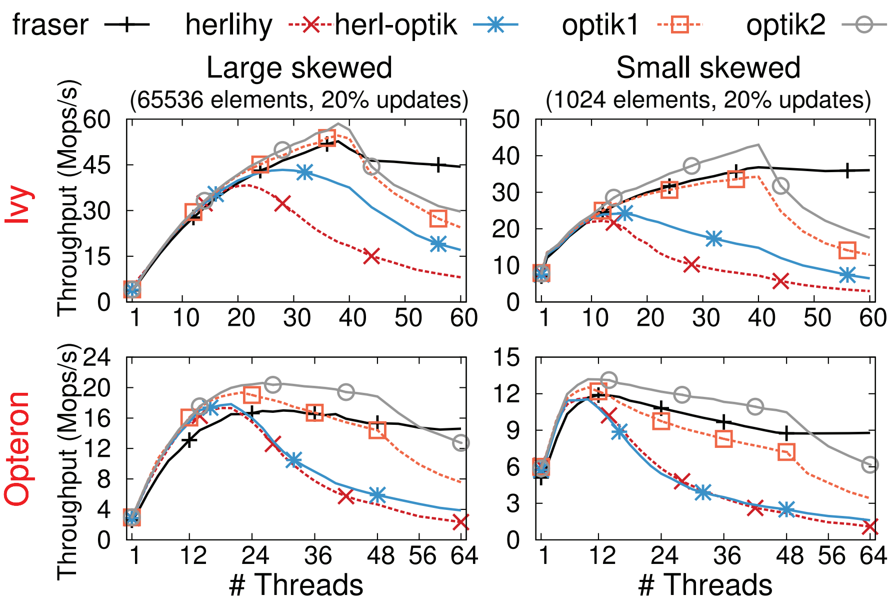


Figure 6.11 – Throughput of skip-list algorithms on *Ivy* and *Opteron* on various workloads.

workloads, on 20 threads on *Ivy*, without OPTIK 30% of update operations have to restart due to concurrency, compared to 24% with OPTIK. Contrarily, on the *Opteron* due to the overall lower throughput than *Ivy*, both *herlihy* and *herl-optik* have 50% operation restarts on 20 threads.

On skewed workloads, we also notice the benefits of using OPTIK, even though it can introduce unnecessary operation restarts. In particular, *optik2*, which is the variant that immediately restarts if there is a trylock failure, is more scalable than *optik1*, that uses `optik_lock_version` and does fine-grained validation if the version is not validated. For example, on very-high contention, on 20 threads, 40% of the operations have to restart with *optik2*, while just 20% with *fraser*. Still, *optik2* delivers 10% higher throughput than *fraser* on 20 threads. The main reason for *optik2* being more scalable than the rest is the important property of OPTIK that we have already extensively discussed: Threads fail the validation with a single atomic operation, without waiting behind the occupied lock. The other three lock-based skip lists do not include false restarts, they do however include false contention behind the per-node locks. *optik2* also benefits from (i) simpler implementation than the rest, as it does not include the fine-grained validations, and (ii) the eager node insertion. Overall, *optik2* is faster than *fraser*. However, *optik2*'s throughput significantly drops on multiprogramming, while *fraser* is able to sustain its throughput.

6.5.4 OPTIK in Binary Search Trees (BSTs)

In our asynchronized concurrency (ASCY) work [51], we observe that none of the existing lock-based BST algorithms follows all four ASCY guidelines: Most algorithms tend to acquire (on average) a large number of locks. We use OPTIK to design BST Ticket (BST-TK), an external tree, where every internal router node is protected by an OPTIK lock. The BST-TK algorithm is essentially the same as the fine-grained OPTIK-based linked list—traversing and then locking nodes is based on hand-over-hand version tracking. Overall, BST-TK acquires one lock for successful insertions and two locks for successful removals.

Correctness. The correctness sketch for BST-TK follows the same ideas as the OPTIK-based linked list. We provide a complete correctness proof of BST-TK in our technical report [50].

Evaluation. Figure 6.12 compares BST-TK to the state-of-the-art lock-free BST by Natarajan et al. [161] (*natarajan*), as well as the fastest pre-existing lock-based BST by Bronson et al. [30] (*bronson*). Clearly, *bronson* is significantly less scalable than the other two, because it includes much heavier synchronization. BST-TK and *natarajan* show similar performance, however, the OPTIK-based BST is on average 3% faster than *natarajan*.

6.5.5 OPTIK in Queues

We use OPTIK in various concurrent queue designs. First, we optimize the classic Michael-Scott queues [153] (MS-queue) using OPTIK locks. The first lock-based MS-queue variant

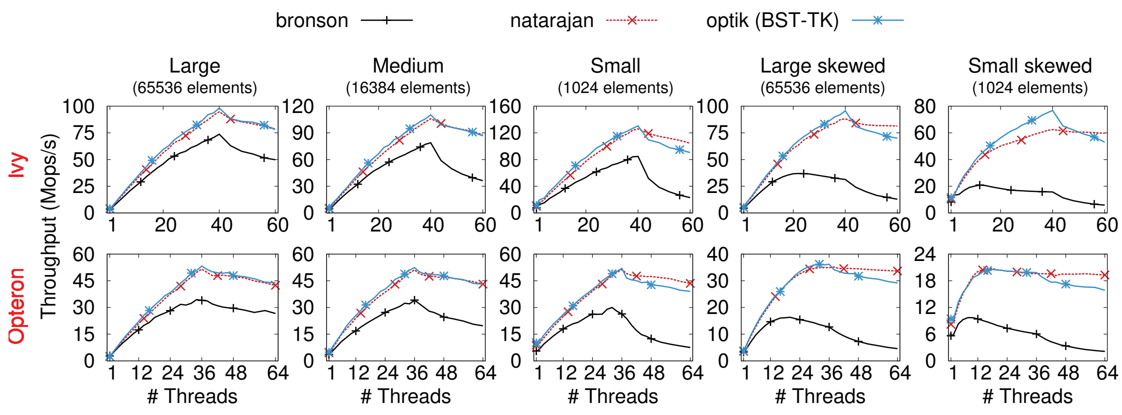


Figure 6.12 – Throughput of BST algorithms on *Ivy* and *Opteron* on various workloads.

employs the `optik_lock_version` function to optimize the `dequeue` function: The operation is optimistically prepared so that if the validation succeeds, only a single store is performed in the critical section. If the validation fails, the `dequeue` operation is prepared and performed in the critical section, as usual. The second (lock-based) variant is very similar to the first one, however, it uses `optik_trylock_version` instead of the lock function. If the validation fails, then the operation is restarted. The third variant is a lock-based/lock-free MS-queue hybrid. We use the lock-free `enqueue` implementation of the MS-queue unaltered. We opt for this approach because the enqueue operations do not offer any opportunities for optimism. For the `dequeue` function we use the OPTIK trylock implementation.

The final variant of MS-queue introduces the idea of *victim queues*. The `dequeue` function uses the same trylock implementation as the last two designs. The `enqueue` implementation utilizes the `optik_num_queued` function of OPTIK locks (on top of ticket locks—see Section 6.3). If the number of waiting nodes is large (e.g., more than two in our implementation), then the thread performs the insertion in a secondary victim queue, instead of waiting behind the lock. The first thread to put a node in the empty victim queue is responsible for linking the victim queue to the main one. The results are (i) lower contention behind the lock, and (ii) a simple victim-queue design as it does not interact with `dequeue` operations.

Correctness. The first three variants of MS-queue do not essentially affect the correctness of the original designs. The fourth design employs the victim-queue idea. Enqueue operations either wait behind the lock to normally perform their operation, or insert the element in the secondary victim queue. This secondary queue is linked to the main one, once the first thread to use it gets the lock. This same thread is also responsible for emptying the victim queue so it can be reused. Operations that utilize the victim queue have to wait until the victim queue has been emptied, thus their elements are visible in the main queue. This waiting ensures that they can be linearized properly.

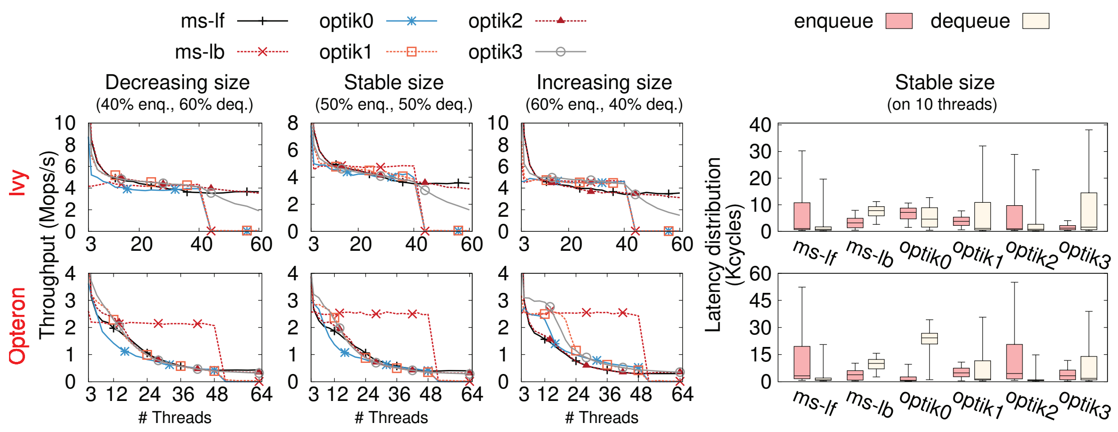


Figure 6.13 – Throughput and latency distribution of queue algorithms on *Ivy* and *Opteron* on various workloads.

Evaluation. We evaluate the lock-based (*ms-lb*) and the lock-free (*ms-lf*) MS-queues. We use *ms-lb* with MCS locks. We also evaluate the three MS-queue variants (*optik0*, *optik1*, *optik2*), as well as the one using victim queues (*optik3*). We initialize the queues with 65k elements. The results include several interesting points (Figure 6.13).

First, *ms-lb* delivers stable performance, regardless of the contention levels, due to the MCS locks. If we use any simple spinlock algorithm (e.g., test-and-set) instead of MCS, the throughput of *ms-lb* degrades as we increase contention. However, when the number of threads becomes more than the number of hardware contexts, the combination of locking and the fairness of MCS kills throughput.[10]

Second, the remaining queue algorithms do not scale and do not even keep stable performance as we increase contention, especially on *Opteron*. Unlike *ms-lb* with MCS locks, all other designs have two single points—cache lines—of contention, namely the head and the tail of the queue. *Opteron* is an 8-socket machine, thus increasing the number of threads, increases the non-uniformity as well, resulting in more expensive cache-coherence traffic (Table 4.3). Still, on both platforms, *ms-lb* is slower than the rest on less than 6-7 threads.

Third, it is worth comparing the two MS-queues with the different OPTIK-based queue implementations. *optik2* (lock-free enqueue, OPTIK-based dequeue) behaves practically the same as *ms-lf*, showing that the simple CAS validation of OPTIK locks does resemble lock-freedom. Then, the victim-queue technique of *optik3* does bring some benefits that are mostly visible on the increasing-size workload which stresses enqueues. *optik3* is on average 28% faster than *ms-lf* on this workload, while overall it is 7% faster.

Regarding *optik1*, on the one hand it contains the enqueue implementation of *ms-lb*, thus on the increasing-size workload it behaves similar to *ms-lb*. On the other hand, it uses the

[10] There are techniques, such as time-published queue-based locks [96], for alleviating this problem.

`optik_trylock_version` implementation for dequeuing, showing similar performance to *optik2* and *ms-lf*. Furthermore, *optik0* on the *Opteron* shows that using OPTIK locks with the lock/unlock interface, under high contention, is not a good idea. At the end of the day, OPTIK locks are simple spinlocks.

Finally, the latency-distribution graphs reveal the power and the weaknesses of each implementation. For example, dequeuing an element is very fast with *ms-lf*, however, enqueuing is very expensive. Similarly, enqueuing with *optik3* is fast because of the victim-queue approach, but dequeuing is slow.

6.5.6 Summary

The combination of the OPTIK pattern with OPTIK locks is a very strong concurrency tool. The resulting algorithms are simple, include minimal synchronization, and follow our ASCY patterns. We illustrate the power of OPTIK by:

- designing five new CDSs: (i) an array map with a corresponding hash table, (ii-iii) a global-lock and a fine-grained linked list with two corresponding hash tables, (iv) a concurrent BST (BST-TK), and (iv) a skip list;
- optimizing four state-of-the-art CDSs: (i) the `ConcurrentHashMap` algorithm in Java [127], (ii) the optimistic skip list by Herlihy et al. [105], and (iii-iv) both the lock-free and lock-based Michael-Scott queues [153];
- introducing two concurrency techniques: (i) node caching for list structures, and (ii) victim queues for concurrent queues.

Of course, OPTIK is not always a suitable solution. The most prominent example of such a case is stack data structures. We briefly experiment with stacks (not shown in the graphs). More precisely, we redesign the classic lock-free stack by Treiber [206] using OPTIK. The original and the OPTIK-based variants behave similarly. Still, the contention levels that can be induced on a highly parallel stack cannot be sustained by neither the “simple” OPTIK lock, nor the lock-free solution. There are ways to alleviate this problem, such as aggressive backoff mechanisms, or elimination [98]. Note that large backoff times might result in large tail latencies.

6.6 Conclusions

Concurrent data structures (CDSs) facilitate concurrent programming by abstracting data sharing behind an interface, thus removing the need for developers to implement synchronization. Consequently, a well-designed and implemented concurrent data structure offers portability and scalability of data sharing in concurrent systems, without any effort from the system developer. Nevertheless, implementing these concurrent data structure in a portably scalable manner is a daunting task. As we illustrated with our novel concurrent hash table, namely CLHT, designing scalable data structure algorithms requires devising complicated concurrency/synchronization techniques. Such techniques are of course non-ubiquitous, leaving the design of CDSs to concurrency experts only. Typically, every new scalable CDS algorithm results in a publication in one of the top concurrency conferences.

In this chapter, we simplified the design of scalable concurrent data structures by introducing the OPTIK design pattern and the underlying OPTIK locks. The OPTIK pattern offers a concrete and simple way of detecting conflicting concurrency in concurrent data structures. Therefore, it can be used to methodically design portable and scalable data structures. OPTIK locks provide a concrete and efficient implementation of the pattern. OPTIK-based algorithms are simple, include minimal synchronization, and follow our asynchronized concurrency paradigm. We illustrated the power of OPTIK by (a) designing five novel concurrent-data-structure algorithms, and by (b) optimizing four existing state-of-the-art ones.

7 Abstracting Multi-Core Topologies with MCTOP

As we highlighted earlier in this thesis, portability and efficiency are usually antagonists in multi-core computing. In order to develop efficient code, one needs to take into account the topology of the target multi-cores (e.g., for locality). This observation is valid for any aspect of concurrent programming (not only synchronization) and clearly hampers code portability. In this chapter, we show that you can achieve portable optimizations (i.e., optimize concurrent software for the underlying multi-core while maintaining portability), using MCTOP, an abstraction of multi-core topologies. MCTOP enables developers to accurately and portably define high-level performance policies. We illustrate that if these policies are designed properly, they result in portable optimizations.

7.1 Introduction

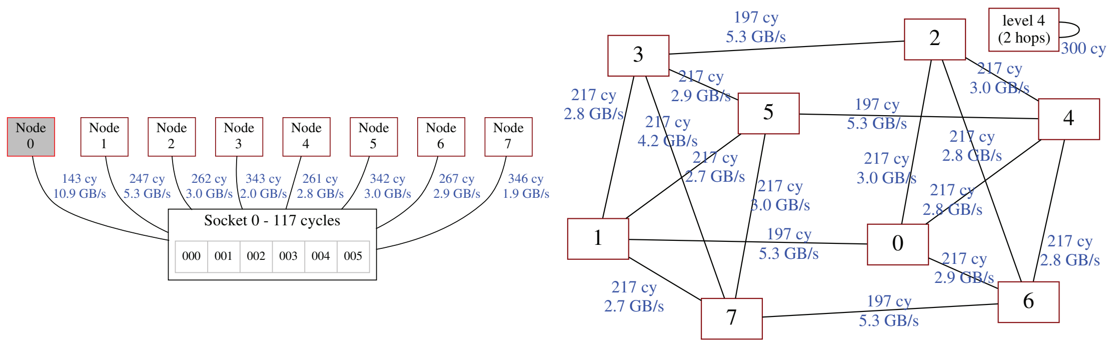
Since 2000, computing systems are becoming more diverse in terms of the numbers of threads per core, cores per socket, as well as the on-chip and off-chip interconnects. As we showed in Part II of this dissertation, this tendency complicates synchronization in concurrent systems. Apart from synchronization, this tendency generally makes concurrent programming very challenging, for developers need to fine-tune every aspect of their software for the underlying hardware in order to achieve performance (e.g., [17, 27, 28, 78]). However, optimizing for specific multi-core topologies hinders the portability of software. In fact, optimizing software for multi-cores raises two main questions: (i) how to harvest and expose the details of multi-cores in software, and (ii) how to fine-tune according to those details, while ensuring portability.

Traditionally, developers have been relying on libraries, such as `libnuma` [120] on Linux, `liblgrp` [167] on Solaris, and `hwloc` [28] for abstracting the topology of multi-cores. These libraries offer a topology representation of multi-cores as well as a companion interface for placing threads (and data). However, the provided multi-core representations are low-level and offer only the limited view of the OS. Developers do not have access to the performance characteristics of the underlying multi-core processor and still need to manually optimize their software for each platform.

We present in this chapter an easier, more portable approach to optimizing software for multi-cores. We introduce `libmctop`, a library that generates what we call MCTOP, a multi-core topology abstraction of important low-level information, such as communication latencies and memory bandwidths. Figure 7.1 depicts the visual representation of the MCTOP of *Opteron* (see Section 2.1 for more examples). Of course, a developer could directly use this low-level information to fine-tune her software for *Opteron*. For instance, she could decide to use sockets 0 and 1 as they provide minimum latency. Such optimizations—that rely on the specifics of a processor—are not portable. Instead, she could write a *policy* that uses any two sockets (if available) that minimize latency.

As we show in this chapter, MCTOP enables the design of easy, portable, and efficient optimizations using such high-level performance policies. In turn, these policies make use of the actual numbers generated by `libmctop`. Essentially, MCTOP allows developers to accurately capture high-level semantics that utilize the low-level performance details of multi-cores, thus delivering *portable optimizations*. For instance, using MCTOP, we can easily define policies such as "use one hardware context per core," "use two sockets with maximum bandwidth," or even "use as many threads as required to saturate the memory bandwidth of node *n*," or "use the maximum number of threads, in the two most remote sockets, so that each thread has at least 3 MB of LLC." If designed properly, such policies result in portable optimizations.

`libmctop` is based on MCTOP-ALG, our novel algorithm for inferring the topology of multi-cores relying on two fundamental observations: (i) *cache-coherence protocols are deterministic by design*, and (ii) *communication latencies characterize the topology*. MCTOP-ALG leverages these two observations by collecting accurate core-to-core communication latencies. These latencies are used to infer the topology of the processor. On top of this topology, `libmctop` (via plugins) collects additional low-level measurements, such as cache latencies and sizes, as well as memory latencies and bandwidths. The end result is an automatically-generated MCTOP representation of the underlying multi-core.



(a) Intra-socket topology of a socket.

(b) Cross-socket topology.

Figure 7.1 – Topology representation of an 8-socket AMD processor—*Opteron*.

We argue that MCTOP-ALG's measurement-based approach is superior to loading multi-core topologies from the underlying OS or hardware (e.g., using `CPUID`) for various reasons: (i) *portability*—collecting measurements is almost identical on any architecture or OS, unlike reading topology info from the OS or the hardware; (ii) *forward/backwards compatibility*—measurements do not depend on the OS version; (iii) *correctness*—numbers do not lie, while the OS can be misconfigured[1]; (iv) *extensibility*—independence from the information that vendors do or do not want to expose; and (v) *accuracy*—a measurement-based approach automatically collects accurate low-level measurements that we need in MCTOP.

We illustrate portable optimizations with MCTOP with four examples on five processors from Intel, AMD, and Oracle. First, we automate backing off in lock implementations using the communication latencies of MCTOP. Our optimized TAS, TTAS, and TICKET spinlocks deliver up to 39% average throughput improvements. Second, we design a topology-aware mergesort algorithm that builds an optimal cross-socket merge tree on top of MCTOP. This mergesort algorithm is 19% faster on average than the parallel sort algorithm of C++ standard library which is topology agnostic.

Furthermore, we design a thread placement library, called MCTOP-PLACE, on top of MCTOP and use it in optimizing the Metis MapReduce library and OpenMP. MCTOP-PLACE includes 12 high-level performance policies that enable thread placement with locality, bandwidth, or even power optimizations. We plug MCTOP-PLACE in Metis and achieve 17% better performance, while consuming 14% less energy in four workloads. Similarly, we extend OpenMP's thread placement functionality with runtime support for adaptation of placement policies. Consequently, our OpenMP version allows for portable, high-level, and dynamic thread placement. We evaluate Green-Marl's [108] OpenMP-based graph workloads and improve the performance of various graph analytics, such as PageRank, by 22% on average.

To summarize, the main contributions of this chapter are as follows:

1. MCTOP, a rich multi-core topology abstraction which enables portable concurrent software optimizations;
2. MCTOP-ALG, a portable algorithm for inferring the topology of multi-cores without relying on the topology information of the OS or the hardware;
3. `libmctop` and the software we build using `libmctop`, both available at `http://lpd.epfl.ch/site/mctop`.

As we discuss in Section 7.3, `libmctop` has certain limitations. We have evaluated `libmctop` only on `x86` and `SPARC` architectures, and we cannot yet guarantee the effectiveness of MCTOP-ALG on other architectures (e.g., `ARM`, `POWER`). Additionally, in order to collect accurate measurements, `libmctop` requires a offline, solo execution on the target processor for the one run that infers the topology (this means stopping all other applications for the duration of `libmctop`'s first execution).

[1] On *Opteron*, the OS has an incorrect mapping of cores to memory nodes, while MCTOP-ALG infers the correct mapping. All 48 cores are assigned to four out of eight nodes.

The rest of the chapter is organized as follows. In Section 7.2, we describe the programming interface of MCTOP. In Sections 7.3 and 7.4, we show how to create and enrich MCTOPs in a portable manner. We then describe examples of high-level policies that result in portable efficiency in Section 7.5 and use these ideas in designing a thread placement library in Section 7.6. Finally, we present practical examples and conclude the chapter in Sections 7.7 and 7.8, respectively.

7.2 The MCTOP Topology Abstraction

The first step for achieving portable optimizations is to provide a programming abstraction of multi-core topologies. That way, software can build on this abstraction and avoid using the limited view of the multi-core that is exposed by the OS. Accordingly, `libmctop` includes a portable topology abstraction, called MCTOP (shorthand for multi-core topology).

MCTOP has two important characteristics. First, MCTOP is *generic*, so that it can be used to describe any modern multi-core processor. Second, MCTOP is *extensible*, in that it supports the low-level details of multi-cores, which are necessary to achieve fine-tuning of software and portability at the same time.

In the remainder of this section, we first describe the programming interface of MCTOP, and, then, we illustrate several examples of MCTOP topologies. In Section 2.1, we have described important multi-core characteristics that affect the design of `libmctop`. In Section 7.3, we introduce a generic algorithm for harvesting MCTOP topologies of multi-cores, while in Section 7.4, we present the plugin system of `libmctop` for extending MCTOP topologies.

`libmctop` Programming Interface. MCTOP is a multi-core topology abstraction which includes the basic topology of multi-cores (e.g., how cores or sockets are interconnected), as well as low-level performance measurements (e.g., memory latencies and bandwidths). MCTOP topologies are stored in description files, which are created by `libmctop` once, as we describe later, and are then used to load the topology. Once a topology is loaded, the developer can either use `libmctop`'s programming interface to access the MCTOP topology, or visualize the topology on screen or as a graph. `libmctop` represents MCTOP topologies as a set of structures that are linked together to describe the processor. The most important structures of MCTOP are shown in Table 7.1.

These structures essentially represent graphs which are interconnected (i) vertically, in order to represent the actual hierarchical topology, and (ii) horizontally, for simplifying the traversal of all objects at each level. For instance, a `hw_context` holds pointers to its parent `hwc_group`, its parent `socket`, as well as its successor (in terms of proximity) `hw_context`. Additionally, every structure holds a pointer to additional low-level information, such as memory latencies.

`hw_context`	The lowest scheduling unit of the processor. If SMT exists, `hw_context` represents a hardware context, otherwise it represents an actual core.
`hwc_group`	A group of `hw_contexts`. This could for example be a core that contains two hardware contexts, or a group of cores that share the L2 cache. There might be multiple levels of `hwc_group` within a socket.
`socket`	A `hwc_group` with additional information about the NUMA memory nodes and the interconnection with other sockets.
`node`	A memory node with information such as capacity.
`interconnect`	The interconnection between two `sockets`. Contains info such as the communication latencies.
`mctop`	The structure that represents a processor and links everything together. Contains info about latency levels, SMT, the number of sockets and cores, etc.

Table 7.1 – The main structures of MCTOP.

We opt for a representation that uses terms that match any modern modern processor and are extensible for future designs. That way, the interface of `libmctop` uses terms which are familiar to system designers, such as:

- `mctop_get_local_node(hw_ctx)` to get the local node of a `hw_context`;
- `mctop_socket_get_cores(socket)` to get the cores of a `socket`; and
- `mctop_get_latency(id0, id1)` to get the latency between any two components.

7.2.1 Examples of MCTOP Topologies

`libmctop` can generate a simplified visual representation of the processor in order to make the topology more accessible to developers. `libmctop` uses the Graphviz [86] visualization library for graphs. `libmctop`'s visual representation includes two main graphs, depicting the intra- and the cross-socket topologies respectively. We illustrate `libmctop` on various `x86` and `SPARC` processors. In Section 2.1, we present the automatically generated graphs and provide details of each platform. In Section 7.7, we use five of these platforms (i.e., *Ivy*, *Opteron*, *Haswell*, *Westmere*, *SPARC-T44*) in our experiments.

7.3 MCTOP-ALG: Inferring Topologies

The first step for creating the enriched MCTOP representation of a processor is to generate the basic topology of the hardware. Essentially, this basic representation describes how hardware contexts are placed in sockets, how these sockets are interconnected, and how contexts are connected within each socket.

We name our algorithm that infers the basic topology of cache-coherent shared memory processors using only latency measurements MCTOP-ALG. In Section 7.4, we describe how these basic MCTOP topology abstractions are augmented with additional measurements, such as memory latencies and bandwidths, in order to enable portable optimizations. MCTOP-ALG relies on two simple, yet important observations regarding cache-coherence protocols of modern multi-core processors.

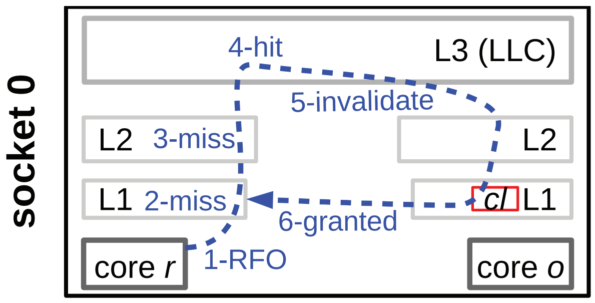


Figure 7.2 – Coherence traffic for an RFO request.

Observation 1: Cache-coherence protocols are deterministic. Cache-coherence protocols are responsible for keeping data consistent in the various caches of the multi-core. Most modern processors implement (variants of) the MESI coherence protocol [174].

Hardware cache-coherence protocols are deterministic by design. Still, non-deterministic schedules can appear, but only under contention (e.g., if multiple threads contend for a cache line, then the schedule of coherence messages is naturally not deterministic). *In the absence of contention*, hardware coherence protocols deliver deterministic schedules. In simple words, a given request type (e.g., requesting for writing), on a given multi-core, for a block of data in a specific MESI state and the same placement, always takes the same steps.

Consider the simplified example of Figure 7.2, where a cache line cl is in the modified state[2] in the caches of core o and another core r is requesting the data for writing—a request known as *request for ownership* (RFO). The RFO request for cl misses in the private caches of r. The request finds that o has the only copy of cl through the last-level cache (LLC) (or using a directory, depending on the specific implementation of MESI). Once the copy is found, an invalidation request is sent to o's private caches to discard their copy of cl, after which the RFO request is granted to r. If r is not in the same socket as o, the RFO request is propagated to the correct socket.

Overall, the coherence request takes deterministic steps. Hence, we can devise thread schedules that accurately measure the communication latency between any two hardware contexts.

Observation 2: Communication latencies characterize the topology. Multi-cores include several cache levels for minimizing latency to data. The latency of a request defines at large the *distance* between the source of the request and the placement of data. For instance, on *Ivy* (i.e., the 2-socket Intel Xeon Ivy Bridge), 4, 12, and 42 cycles are the latencies to access the three levels of caches, while 112 and 308 cycles represent the latencies to access data that are in the private caches of another core within the same socket and across sockets, respectively.

Two communicating threads can potentially detect their relative placement based on their communication latency. For example, on *Ivy*, if two threads communicate in approximately 4

[2] The modified state means that this cache line is the only fresh copy of this data and this data is stale in memory.

cycles, they *have to* reside on the same core as the L1 cache delivers this latency. In contrast, communication latency of 300+ cycles reveals that the two threads are on different sockets.

MCTOP-ALG Algorithm. MCTOP-ALG takes advantage of the aforementioned observations by collecting accurate hardware-context-to-hardware-context communication latency measurements and using them in inferring the topology of the machine. The implementation of MCTOP-ALG in `libmctop` requires three functionalities from the underlying OS: A way to read the number of available hardware contexts and the number of memory nodes, and a way to pin threads to specific contexts.

MCTOP-ALG takes the following four steps:

1. Collects context-to-context latency measurements → latency table;
2. Clusters close values into groups and normalizes the latencies accordingly → normalized latency table;
3. For each latency value l, categorizes hardware contexts into groups of contexts that communicate with latency l with each other and with the same latency with other groups → per latency level components;
4. Creates the multi-core representation by assigning roles to components → topology;

We detail below these four steps using *Ivy* as an example—Figure 7.4. We then discuss several practical considerations regarding MCTOP-ALG.

7.3.1 Context-to-Context Latencies

MCTOP-ALG uses two threads that move from hardware context to hardware context and fill up an $N \times N$ latency table, where N is the number of hardware contexts of the processor. For each data point, the two threads execute in lock step as shown in Figure 7.3. Thread y brings the data in a modified state in its local caches and then thread x measures the latency of its own access to the shared data using the timestamp counter of the core [111].

The use of an atomic operation, such as compare-and-swap (CAS), is crucial for two reasons. First, CAS includes a memory fence, hence it preclude the effects of memory consistency models [201]. Second, CAS brings the data in the modified MESI state. The modified state is

	Thread x	**Thread y**
	thread_barrier()	
		CAS(shared_line)
	thread_barrier()	
	s = rdtsc()	
	CAS(shared_line)	
lat[x][y] =	rdtsc() - s	

Figure 7.3 – Lock-step execution of MCTOP-ALG's threads.

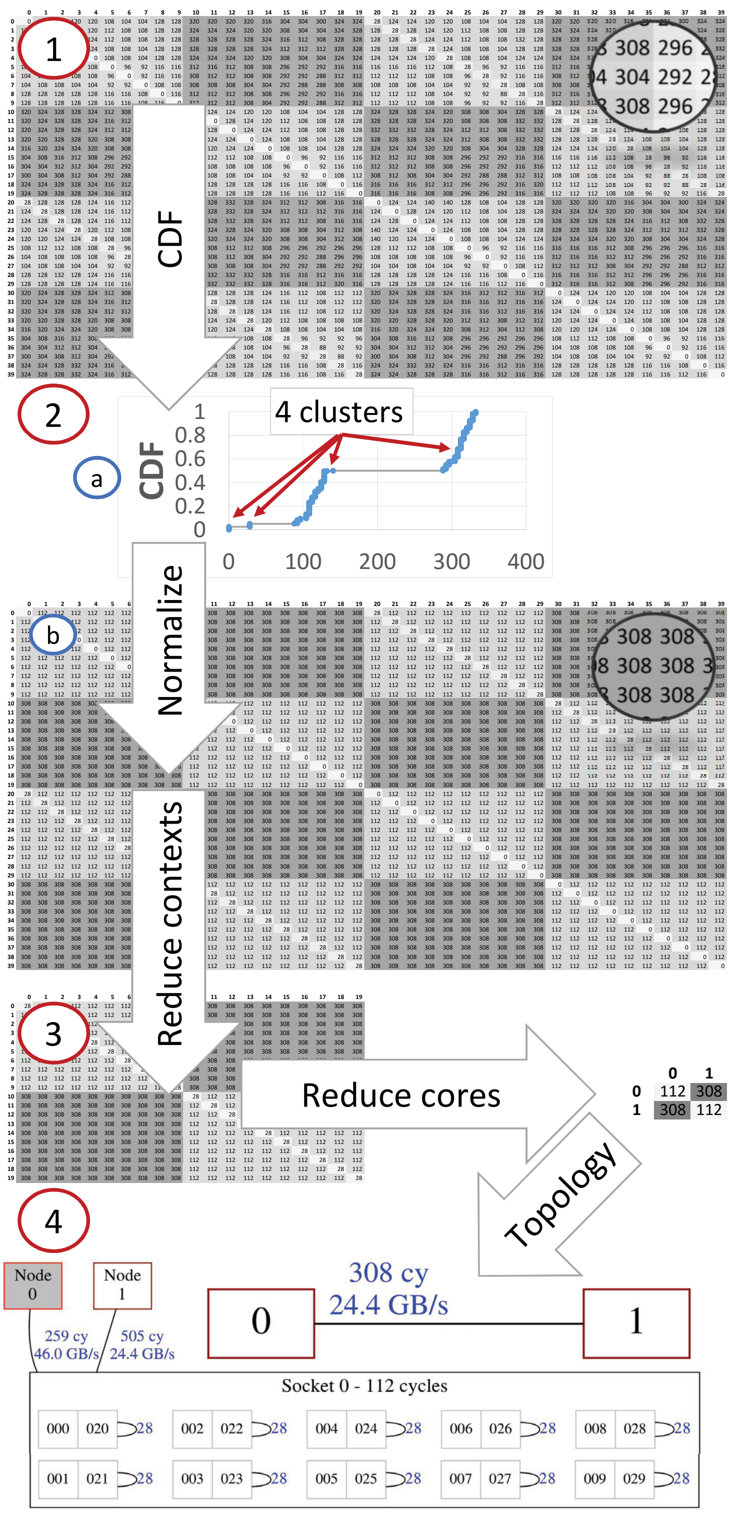


Figure 7.4 – The four steps of MCTOP-ALG: From latency measurements to *Ivy*'s MCTOP multi-core topology.

necessary for avoiding potential whole-machine communication when broadcasting invalidations for a shared cache line (e.g., in *Opteron*–see Section 4.4.2).

The outcome of this step is a latency table (Figure 7.4 ①). Note that in practice we only need to take measurements for either the upper or the lower triangular of the table as the latency measurements are symmetric.

In order to improve the accuracy of each latency measurement, the two threads repeat the lock-step execution for n times on each pair of hardware contexts ($n = 2000$ by default). These n measurements are stored in a local array. Thereafter, the median latency and the percentile standard deviation (stdev) of these latencies are calculated. If stdev is higher than a threshold (7% by default), the execution is repeated on this configuration, while the maximum allowed stdev is slightly increased. Retrying the execution for high stdevs ensures stable values, while increasing the maximum stdev on a retry ensures that the measurements complete, even if they are not very stable.

7.3.2 Latency Normalization

As the heatmap of Figure 7.4 ① shows, the relations between hardware contexts are rather clear. The white diagonal represents the individual contexts, the two light gray diagonals represent the hardware contexts of the same core, and the gray and dark-gray rectangles are the intra- and cross-socket latencies, respectively. To extract these relations, MCTOP-ALG calculates the cumulative distribution function (CDF) of the latency table values—Figure 7.4 (2a). The value clusters of CDF represent these aforementioned relations. MCTOP-ALG detects these clusters and for each cluster generates a triplet with the minimum, median, and maximum latencies.

MCTOP-ALG uses the latency clusters for normalizing the latency table (Figure 7.4 (2b)). Each value of the table is replaced with the median value of the cluster that it belongs to. Normalization is required for detecting relations among hardware contexts.

7.3.3 Component Creation

MCTOP-ALG uses the normalized latency table to extract the relations among hardware contexts for each latency level within the socket (e.g., the three first values of Figure 7.4 (2a)) and assigns them to components. We recursively define a *component* C_l of level $l > 0$ as a set of components of level $l-1$ s.t. any two components in C_l communicate with the latency of level l and have the exact same communication latencies with all the other components of level $l-1$. At level 0, with latency 0, every hardware context belongs to its own C_0 component.

Using this definition of components, MCTOP-ALG recursively groups hardware contexts together by performing classification and reduction of the latency table. For example, in Figure 7.4 ③, the first step is to group the hardware contexts (components C_0) of each core with each other and reduce the table by only keeping the components C_1 (i.e., the cores). Then, the cores of each socket are reduced to C_2 components and we end up with only the cross-socket latencies table.

The outcome is a set of components for each latency level. This assignment of hardware contexts into components describes the relations between contexts.

7.3.4 Topology Creation

In this last step, MCTOP-ALG assigns "roles" to the components of different levels according to MCTOP abstraction of Section 7.2. The result is an abstraction of the actual topology of the processor as shown in Figure 7.4 ④ (the memory measurements are described in Section 7.4). MCTOP-ALG first detects whether (via measurements—see below) the target multi-core includes SMT. If the multi-core has SMT, the components of the first non-zero latency group represent the physical cores of the processor. Similarly, MCTOP-ALG classifies as socket level the level with as many components as the the number of nodes. Every relation higher than sockets represents cross-socket connectivity.

7.3.5 Practical Considerations

Removing the Effects of DVFS. Dynamic Voltage and Frequency Scaling (DVFS) is a common hardware technique for reducing power consumption, where underutilized cores can execute at various voltage/frequency settings. Our implementation of MCTOP-ALG explicitly waits for the frequency of both cores to reach the maximum before proceeding to the lock-step execution. To achieve this, MCTOP-ALG repeatedly measures the execution time of a long-duration empty loop, until the execution time stabilizes.

Detecting SMT. `libmctop` detects if the processor has symmetric multi-threading (SMT) using the same idea with "removing the effects of DVFS." A thread first executes a spin loop solo on a core and measures the time of this execution. Then, two threads execute the same loop on two contexts with minimum latency. If these are the hardware contexts of the same core due to SMT, then the duration of the spin loop will increase.

Performance and Failures. MCTOP-ALG is designed to work solo on the target processor, as it relies on the accuracy of latency measurements. In a few experiments with `libmctop` on a utilized machine we observe that MCTOP-ALG is still often able to detect the topology, mostly depending on whether the threads of the other executing applications are pinned on specific cores. Nevertheless, the additional measurements of `libmctop` in Section 7.4 always require an idle processor.

Even on an idle processor, MCTOP-ALG might fail to infer the topology (due to a few spurious measurements). These spurious measurements are mostly because of (i) effects of DVFS, and (ii) effects of SMT, where other OS/system processes might execute on the other hardware contexts of the core that is being used by MCTOP-ALG. When MCTOP-ALG is not able to infer the topology, an error message is printed and the user must retry the execution, possibly with different settings (e.g., more repetitions). MCTOP-ALG relies on the uniform hierarchy of topologies to detect failures: If during step 3 (component creation) the reduction of the table cannot be performed, some of the measurements cannot be clustered properly.

In terms of performance, MCTOP-ALG needs ~3 seconds to infer the topology of our smallest platform (*Ivy*), while it takes 96 seconds to infer the topology of *Westmere* (160 contexts with DVFS enabled). MCTOP-ALG is more stable and faster when DVFS is disabled.

Dynamic Changes of Multi-Cores. `libmctop` does not currently support the detection of dynamic changes of the topology of a multi-core. If, after the execution of MCTOP-ALG, SMT is disabled through BIOS, or a hardware context is disabled via the OS, MCTOP-ALG must be re-executed in order to detect the new configuration.

7.4 Enriching MCTOP Topologies

The basic topology representation which is created with MCTOP-ALG includes the communication latencies of the processor by design (see Section 7.3). These latencies are sufficient for defining locality-oriented performance policies, such as "find the socket that is the closest to socket x." Although locality optimizations are very important on NUMA multi-cores, we argue that with access to further low-level information in MCTOP abstraction, we can implement a broader set of performance policies (see Section 7.5 for examples).

Therefore, MCTOP includes (i) the basic topology representation that is created by MCTOP-ALG, and (ii) a set of additional multi-core measurements. We design `libmctop` to be extensible, so that developers can write plugins to enrich MCTOP. Essentially, plugins build on top of `libmctop` and collect measurements. We have implemented four essential plugins that measure memory latencies and bandwidths, cache-related information, and power-related information (only available on modern Intel processors).

Memory Latency. To estimate the memory latency, we design a plugin that uses a single thread T. T allocates a large chunk of memory on the target memory node and creates a linked list with randomly linked cache-line-sized nodes. T then measures the time it takes to traverse the list. Because the size of the list is large and the access patterns are random, most accesses are served by the main memory. T performs this measurement for all socket-node combinations on top of MCTOP.

Memory Bandwidth. A second plugin estimates the bandwidth from sockets to nodes, using as many threads as the number of hardware contexts per socket. These threads allocate and access large chunks of memory sequentially, thus they are able to saturate the memory bandwidth. The maximum bandwidth is estimated as the summation of the per-thread bandwidths. Additionally, this plugin estimates the single-thread read bandwidth, as well as the single-thread and max write bandwidths.

Cache Latency and Size. The cache plugin estimates both the size and the latency of the various levels in the cache hierarchy. To estimate latency, the plugin uses the same technique as the memory latency measurements. The cache size estimation is based on those latency measurements (i.e., it estimates the size of each level by detecting the data size that causes latency to increase). Additionally, the plugin loads and includes the cache sizes from the operating system.

Power Consumption. The latest Intel processors include Intel's running average power limit (RAPL) [111] interface for accurately measuring the power consumption of the cores, the package, and the DRAM. We design a `libmctop` plugin that uses RAPL to gather power measurements which indicate the breakdown of power to hardware contexts. In order to estimate the maximum power consumption, we use the same memory intensive workload that we use for bandwidth measurements. We measure and include in MCTOP measurements such as: idle processor power, full power (all hardware contexts are active), power of the first hardware context, and power of the second context of one core.

7.5 Portable Optimizations with MCTOP

Optimizing a concurrent system for the underlying hardware hinders the portability to other processors. In certain cases, this lack of portability is inevitable. For example, using Intel's transactional memory [111] results in software that can only execute on specific processor models. However, for more traditional topology-oriented optimizations, such as locality or bandwidth optimizations, we can achieve *portable optimizations* (i.e., fine-tuning the system in a portable manner) on top of MCTOP. We can do so because these traditional notions are accurately defined on MCTOP. For instance, "use the cores that are the closest to core x," is a policy that can be easily, accurately, and portably defined on MCTOP. Consequently, we can define high-level performance policies on top of MCTOP that leverage, but at the same time abstract the low-level details of the topology of multi-cores.

Essentially, MCTOP provides a query engine for the topology of multi-cores. Of course, software should not rely on any assumptions regarding the underlying multi-core, but rather build on top of `libmctop`'s interface to access information in a portable manner. For instance, an algorithm that explicitly allocates memory on nodes 0 and 1 will not work on a single-node processor. Instead, the developer can use `mctop_get_num_nodes` to provision for the available resources of any multi-core.

In what follows, we highlight several examples of portable optimizations on top of MCTOP with policies. We dedicate Section 7.6 to a detailed description of thread placement policies with MCTOP. In Section 7.7, we implement and evaluate several of these examples.

Optimal Work Stealing. Work stealing [22] is a commonly used technique in parallel runtimes that aims to minimize the imbalance of work across worker threads. In brief, worker threads have access to work queues from which they dequeue and execute chunks of work. In order to avoid imbalance, workers with no work must steal work from other worker threads. Ideally, work stealing must be performed in a way that (i) reduces the overhead of accessing the non-local queue, and (ii) optimizes the locality/bandwidth of the stealer to the work chunk.
↦ **MCTOP Policies.** Assume a parallel runtime with per-thread work queues. Then, on top of MCTOP, we can easily implement the following work-stealing policy: If the local work queue is empty, steal from the queue of the hardware contexts that are the closest in term of latency. If unsuccessful, continue with the contexts that are in the next closest distance. Continue this process until either work is found, or there is not work to steal. This policy defines work stealing with optimal locality.

Optimal Reduction Trees. Many parallel algorithms and frameworks rely on the fork-join computation model. The most notable example is of course the MapReduce paradigm [52]. In the fork-join model, computation is split in chunks that are processed in parallel by multiple processes. The local results of each process are then reduced (joined) to get the final result. Intuitively, on a multi-core processor, when these local results represent a sizable amount of data, the thread and data placement of the reduction process can have a large effect on the performance of the computation.
↦ **MCTOP Policies.** We describe policies for cross-socket reduction trees that we believe are broadly applicable. The following steps assume that multiple threads can operate concurrently on reducing the same chunk of data. Within sockets, all threads of a socket cooperate on reducing the same chunks. Across sockets, we build a binary reduction tree such that (i) the final destination socket/node is the one that requires the final data, and (ii) at each level of the tree, we choose the sockets to cooperate so that we maximize the bandwidth to data. We use these policies in a sorting algorithm in Section 7.7.

Estimate Power Consumption. Power consumption and energy efficiency have gained attention in the past few years [25, 32]. MCTOP's power-related measurements on modern Intel processors can be used to estimate the power consumption of a specific execution (i.e., a fixed thread placement) before the actual execution.
↦ **MCTOP Policies.** Being able to estimate the power consumption of an execution gives us the opportunity to trade performance off for lower power. For example, a low-power policy (e.g., Section 7.6) prioritizes the use of hardware contexts that minimize power consumption.

Educated Backoffs. Waiting for a short period of time before retrying an operation, namely backing off, is an essential technique for alleviating congestion in software. For example, backoffs are used in lock implementations [3]. Estimating the correct amount of time to back off is difficult. Small backoffs miss a window for further optimization, while large backoffs could hurt performance by inducing idle periods of no work.
↦ **MCTOP Policies.** We define the granularity of backing off on top of MCTOP based on the

intuition that "messages" on multi-cores travel as fast as coherence protocols. Accordingly, we set the backoff quantum to be the maximum (or minimum, depending on the workload) latency between any threads that are involved in the execution. We use this policy in lock algorithms in Section 7.7.

7.6 Thread Placement with MCTOP

A prominent way of using `libmctop`'s MCTOP is by developing higher-level libraries that rely on performance policies. As we describe in Section 7.5, libraries such as locality-aware work queues and reduction trees can deliver practically optimal performance without exposing the details of the underlying hardware.

Another natural construction on top of MCTOP is a library that abstracts the placement of threads to hardware contexts given some placement policy. For instance, we might need to place threads so that they are as close as possible to one specific node where some data reside. We can easily implement such functionality on top of `libmctop` because of the topology abstraction and the low-level measurements that MCTOP provides.

We develop MCTOP-PLACE, a thread placement library with 12 policies and runtime support for changing policies. MCTOP-PLACE is portable: It can completely abstract the underlying multi-core topology for any platform that `libmctop` supports. MCTOP-PLACE comprises two main components: (i) the creation of individual thread placements for given configurations (i.e., policies and number of threads), and (ii) a placements pool that supports runtime modification of configurations.

MCTOP-PLACE. `libmctop` thread placement (MCTOP-PLACE) creates a mapping of threads to hardware contexts given a placement policy. Optionally, the user can provide the number of threads and the number of sockets of the placement. The basic interface of MCTOP-PLACE includes functions for: (i) initialize a new MCTOP-PLACE object with a given policy, (ii) pin a thread to the next available context of a MCTOP-PLACE object (if any), and (iii) unpin thread from the context and return the context to MCTOP-PLACE.

We implement 12 placement policies (Table 7.2). In non-SMT multi-cores, `HWC`, `HWC_CORE`, and `CORE_HWC` policies are equivalent. We believe that these policies cover the most prominent placement choices a programmer can make, such as compacting or spreading threads as much as possible. Still, if none of these policies covers a required thread placement, implementing a new policy is straightforward since the basic data structures for doing so are already in place.

Apart from the mapping of threads to hardware contexts, MCTOP-PLACE provides a plethora of additional information and function calls to leverage `libmctop`'s topology. Figure 7.5 shows an example output of `mctop_place_print` on our *Ivy* platform (see Section 2.1). MCTOP-PLACE calculates and exports details such as the number of cores that will be used, the bandwidth

Name	Short description
NONE	Threads are not pinned to hardware contexts.
SEQUENTIAL	Use the sequential OS numbering.
CON_HWC	Choose the socket with maximum local memory bandwidth. Starting from this socket, place threads as compactly as possible on all hardware contexts of this socket and then continue to the next-best connected neighboring socket (i.e., with minimum latency and maximum bandwidth).
CON_CORE_HWC	Same goal as CON_HWC. Instead of using all hardware contexts, use all unique cores of the socket before using the second hardware context. Still, fill the first socket before using the next one.
CON_CORE	Same goal as CON_HWC. Instead of using all hardware contexts, use all unique cores of all used sockets. Once all cores are used, use the second, third, etc. hardware context of each core.
BALANCE	Balanced version of the CON_* placements. Instead of first filling up a socket before using the next one, balance threads to sockets.
RR	Place threads round robin to sockets. Prioritizes the sockets with maximum bandwidth to their local memory. Uses unique cores first (RR_CORE) or all hardware contexts of the core (RR_HWC).
POWER	Place threads so that the estimated maximum power consumption is minimized. (only for Intel processors which support RAPL [111])
RR_SCALE	Same as RR_CORE, but also re-adjusts the number of threads in order to provide the number of threads per socket that is enough to saturate the memory bandwidth to their local node.

Table 7.2 – The set of policies offered by MCTOP-PLACE.

proportions of each socket according to the allocation policy, and an estimation of the maximum power consumption with and without DRAM. Additionally, once a thread is pinned, it has access to information such as its local node and its hardware context and core IDs within the socket. In Section 7.7 we use MCTOP-PLACE in various examples.

```
## MCTOP Placement   : MCTOP_PLACE_CON_HWC
# # Cores            : 15
# HW contexts (30 ) : 0 20 1 21 2 22 3 ...
# Sockets (2  )      : 20000 20001
# # HW ctx / socket : 20    10
# # Cores / socket  : 10    5
# BW proportions     : 0.655 0.345
# Max pow no DRAM    : 66.7  43.4   = 110.1 Watt
# Max pow with DRAM : 111.9 88.7   = 200.6 Watt
# Max latency        : 308   cycles
# Min bandwidth      : 24.28 GB/s
```

Figure 7.5 – Example output of MCTOP-PLACE.

MCTOP-PLACE Pool. MCTOP-PLACE is sufficient to place threads according to a single placement policy. However, software systems might require different placement policies in different execution phases. To support this functionality we build an MCTOP-PLACE pool object that offers runtime selection of placement policies. In Section 7.7 we show how we use MCTOP-PLACE's pool to extend the thread placement capabilities of OpenMP.

7.7 Examples of Portable Optimizations

We experimentally show how the performance policies on top of MCTOP achieve portable optimizations in software. Our goal for this section is to illustrate (i) the usefulness of the low-level measurements of MCTOP, (ii) the ability to optimize existing software using `libmctop`, and (iii) the portable efficiency of the resulting software.

Experimental Setup. We execute our experiments on all five platforms described in Section 7.2.1. We perform 11 runs of each experiment and present the median performance. We do not to show error bars for readability as our experiments have small variance. Whenever there is variability across runs, we discuss it in text. The duration of each of our lock experiments is 5 seconds.

7.7.1 Using Latencies to Optimize Locking

Traditional spinlock algorithms, such as ticket locks, resort to busy waiting when the lock is not free [10, 149]. While busy waiting, it can be beneficial to back off before re-accessing the shared memory location of the lock [3]. As discussed in Section 7.5, with MCTOP it is straightforward to make educated backoff decisions for such algorithms. We use MCTOP to optimize three lock algorithms: TAS, TTAS, and TICKET locks (see Table 2.1). We use as backoff quantum the maximum communication latency between any two threads involved in the execution. Different lock algorithms employ the backoff quantum in different ways. With ticket locks we set the back off to be proportional to the position of the thread in the "queue" [119, 149]. With TAS and TTAS, threads simply back off for one quantum before accessing the lock again.

Evaluation. Figure 7.6 includes the relative throughput of the three lock algorithms with and without our MCTOP-based backoff schemes. The experiment involves multiple threads

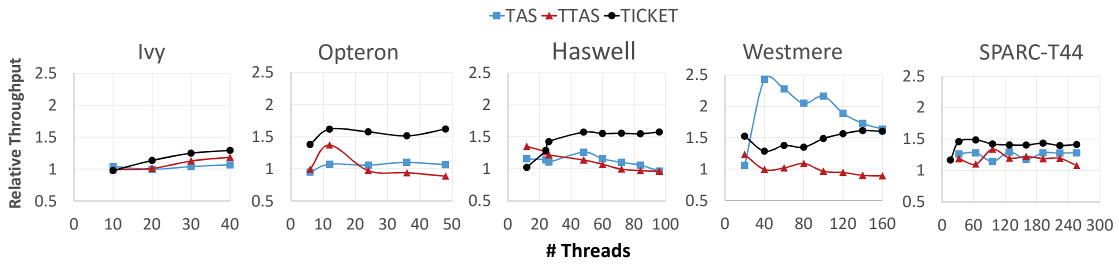


Figure 7.6 – Throughput of different lock algorithms using educated backoffs with MCTOP.

competing for the same lock, performing 1000 cycles of work in the critical section, and then releasing the lock. Threads pause after each iteration to avoid long runs [153]. On both the `x86` and the `SPARC` processors, we use the `pause` instruction for pausing [111, 169] as the baseline. On `x86`, we invoke `pause` in a loop to implement our backoff quantum.

Backing off with the "correct" backoff granularity significantly improves performance: On average, we improve the performance of TAS, TTAS, and TICKET by 25%, 8%, and 39%, respectively. These performance gains are consistent across platforms, without requiring reconfiguration or re-compilation of the applications.

Conclusion. We show how MCTOP's low-level information can be used to optimize the performance of locking algorithms. The optimization is portable across platforms, as `libmctop`'s interface provides us with the necessary latencies on each platform.

7.7.2 Using `libmctop` in Parallel Mergesort

We use `libmctop` to devise a very fast, portable mergesort algorithm. Our novelty lies in the way we perform NUMA-aware merging. The starting point for our algorithm is the parallel sort algorithm of the C++ standard library (`gnu_parallel::sort`) [200]. `gnu_parallel::sort` involves two main steps: (i) it breaks the target array into n chunks and sorts these chunks with the standard sequential quicksort algorithm (n is the number of available threads), and (ii) it iteratively performs parallel merging on the sorted chunks until the result is a single sorted array.

Our mergesort algorithm, namely `mctop_sort`, takes the same first step as `gnu_parallel::sort`. Nevertheless, `mctop_sort` merges the sorted arrays using the optimal reduction tree presented in Section 7.5.

Using SMT Cleverly. Merging two sorted arrays using traditional comparison instructions is suboptimal: The aggressive out-of-order cores are not able to predict the direction of the merge branch (i.e., which of the two arrays will give the next element). Recent projects [40, 109] show how to use SIMD instructions for efficient merging. Using 128-bit instructions, we can create a bitonic merge network that merges 8 elements at a time.

However, SIMD registers are shared across the SMT hardware contexts of a core, hence we cannot simply let all contexts perform merging. We implement a variant of `mctop_sort`, namely `mctop_sort_sse`, that bypasses this limitation. Once the sequential sorting is over, we let the first hardware context of each core use SIMD for merging, while the remaining perform traditional non-SIMD merging. To compensate for the faster merging with SIMD, threads using non-SIMD merging are assigned one-third of the data of SIMD threads.

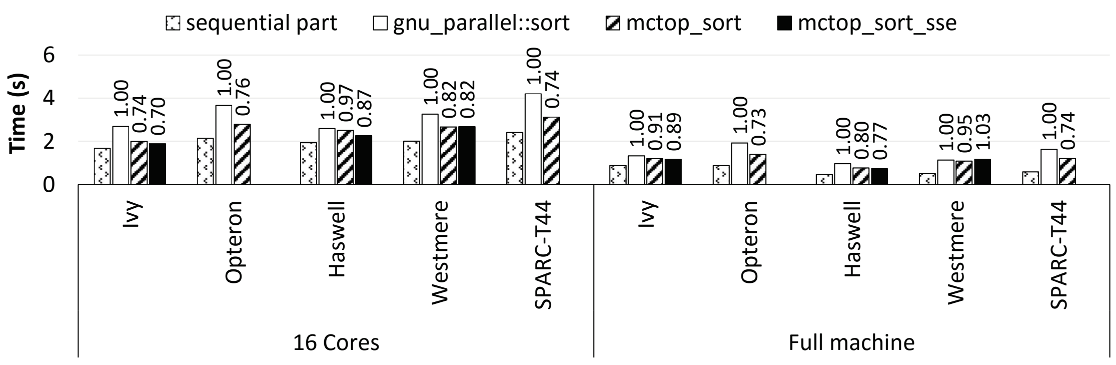


Figure 7.7 – Sorting 1GB worth of integers on various platforms. The labels on top of each bar indicate the execution time relative to `gnu_parallel::sort`.

Evaluation. Figure 7.7 includes a comparison between `mctop_sort[_sse]` and `gnu_parallel::sort`. With `mctop_sort`, threads are spread to sockets, in order to benefit from the LLC of each socket (using `RR` policy from MCTOP-PLACE). The performance of our algorithm is stable across runs, since the placement of threads and data is deterministic. In contrast, we notice big variance for `gnu_parallel::sort`, based on the thread placement of the OS scheduler.

`mctop_sort` is consistently faster than `gnu_parallel::sort` (we observe the same behavior on different data sizes—not shown in the graph), because it always chooses the optimal placement of threads. On average, `mctop_sort` is 19% faster than `gnu_parallel::sort`, which translates to 41% faster merging if we exclude the sequential part of the sorting that is the same on both algorithms. `mctop_sort_sse` delivers similar performance to `mctop_sort` on average, but it can be up to 11% faster than `mctop_sort` (on *Haswell*).

Conclusion. Building optimal merging is straightforward on MCTOP. We leverage low-level details (e.g., bandwidth and latency between nodes) of multi-cores without the need for any platform-specific optimizations.

7.7.3 Using `libmctop` to Improve Metis

We use `libmctop` to optimize the Metis MapReduce library for multi-cores [28, 142]. Metis pins worker threads to hardware contexts sequentially. The reason for this choice is that offering multiple placement policies for every platform is cumbersome, given the diverse characteristics of different platforms. Thus, we build a new version of Metis by linking `libmctop` and using the different placement policies of MCTOP-PLACE (see Section 7.6). As a result, we can choose any of the high-level placement policies offered by MCTOP-PLACE at runtime.

Evaluation. We evaluate Metis in terms of performance and energy efficiency (wherever available), using a set of four representative workloads. We identify the needs of each application, and then use MCTOP-PLACE with the placement policy that best expresses these

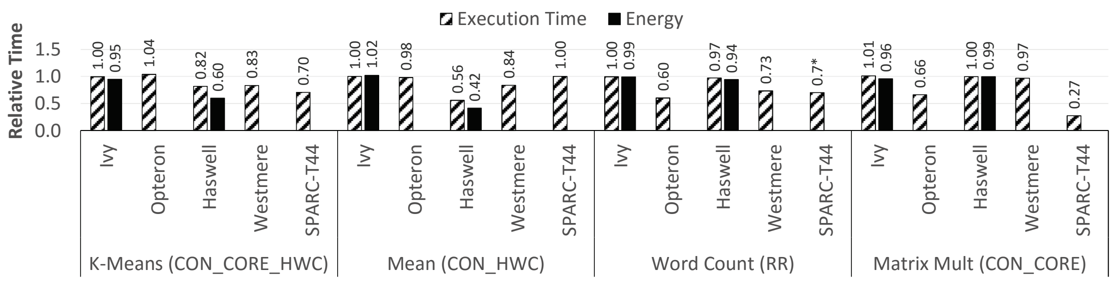


Figure 7.8 – Relative execution time and energy efficiency of Metis with versus without `libmctop`. All workloads are optimized for performance. (**SPARC-T44* for Word Count uses `CON_CORE` MCTOP-PLACE placement.)

needs. We also select the best-performance number of threads for both versions of Metis. MCTOP-enabled Metis always uses fewer or an equal number of threads than default Metis.

As Figure 7.8 reveals, different workloads have different placement needs. Thus, using the default sequential policy of Metis delivers suboptimal performance in all workloads for different platforms. Our version of Metis delivers 17% better performance, across the five platforms, with 14% less energy on the two Intel processors. It is worth noting that in one workload, namely Word Count, our *SPARC-T44* has different placement requirements than the `x86` platforms, delivering the best performance with cores of a single socket. Our performance analysis shows that Word Count has heavy memory allocation and synchronization that benefit from intra-socket locality. Finally, note that in this example we aim at performance, although we do achieve energy gains in some cases. In several Metis workloads, we can trade performance for energy using different threads placements (i.e., `POWER`)—not shown in the graphs.

Conclusion. By modifying the Metis library, we show how a complex software system can be easily take advantage of MCTOP, in order to achieve portable optimizations. General purpose frameworks, such as Metis, can get out-of-the-box benefits from using `libmctop`.

7.7.4 Using `libmctop` to Enrich OpenMP

The GNU `libgomp` OpenMP runtime [23, 180] does not pin threads to cores by default. However, `libgomp` allows users to set the available *places* of parallel threads on the available hardware contexts, as well as coarse-grain strategies for assigning parallel threads to places (e.g., keep them *close*, or *spread* them as much as possible). `libgomp` thread placement capabilities are: (i) offline—they are set through environmental variables before the execution, (ii) inflexible—placements cannot be modified at runtime and are dependent on the number of threads used during initialization, (iii) not fully portable—in many cases placements must be defined differently across platforms to achieve the same effects, (iv) not optimized—placements do not rely on latency or bandwidth numbers.

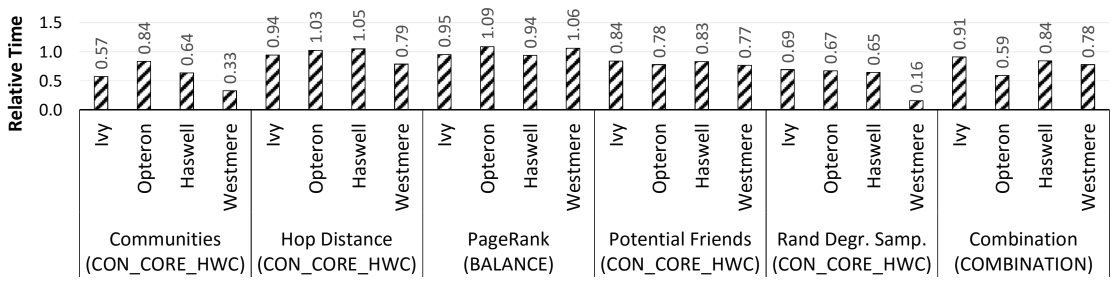


Figure 7.9 – Relative execution time of MCTOP_MP compared to default OpenMP for various workloads.

We extend the thread placement capabilities of `libgomp` (in gcc v4.9.3) using `libmctop` (and MCTOP-PLACE) in order to offer richer and higher-level placement policies. In detail, we plug MCTOP-PLACE in OpenMP and add the `omp_set_binding_policy` function to the OpenMP interface. Doing so, we enable developers to (i) choose placement policies during runtime, (ii) change placement policies between parallel regions, and (ii) leverage the high-level semantics of the MCTOP-PLACE placement policies that generate portable thread bindings.

Evaluation. We evaluate our extended OpenMP (MCTOP_MP) runtime against the vanilla `libgomp` OpenMP library on various graph algorithm workloads produced by Green-Marl [108]—Figure 7.9.[3] We use large datasets (e.g., 100M nodes with 800M edges).

We use MCTOP_MP to enable automatic thread placement policy selection, by running small parts of the workload using different policies and identifying the optimal policy for each parallel section.[4] In contrast, such online decisions are not possible with OpenMP, since it does not offer the same high-level semantics and also cannot dynamically adjust the thread placement at runtime. Overall, our MCTOP_MP version of the algorithms is on average 22% faster across platforms and workloads.

We further port MCTOP_MP's thread placements to OpenMP, in order to estimate the amount of work for reproducing these placements using the default OpenMP capabilities. We observe that in order to reproduce the exact same configurations (with possibly different number of threads per platform) we had to design one policy per-platform per-workload with OpenMP. In terms of performance—not shown in the graphs, our automatic MCTOP_MP solution delivers very similar results to OpenMP with fixed placements.

Finally, we combine two kernels (PageRank and Potential Friends) into a single application, namely Combination. With OpenMP, trying to recreate MCTOP_MP's placement proves impossible: We have to choose the optimal placement policy for either of the kernels, while the performance of the other suffers. This results in 9% lower performance for OpenMP.

[3] The available implementation of Green-Marl [107] does not support SPARC.
[4] Even if the configuration would be manually selected, the developer would need to find which placement policy matches the characteristic of each algorithm and use this policy across platforms.

Conclusion. MCTOP_MP shows that it is straightforward to offer portable optimizations through `libmctop` in software libraries such as OpenMP. MCTOP_MP offers high-level placement policies and runtime support for policy selection and adaptation—characteristics that we believe are useful to OpenMP developers.

7.8 Conclusions

Optimizing concurrent software systems for multi-cores involves fine-tuning not only synchronization, but every aspect of concurrent programming (e.g., thread and memory placement). Inevitably, performing such optimization is typically platform-specific, hence non-portable.

In this chapter, we introduced `libmctop`, a library that enables developers to optimize their software on multi-cores in a portable manner. `libmctop` is based on the MCTOP topology representation that abstracts both the topology and important low-level performance information of the processor. We showed how developers can define high-level performance policies on top of MCTOP to achieve portable optimizations. We illustrated these high-level policies on various examples, including an extended OpenMP runtime with dynamic support for thread placement policy selection based on `libmctop`.

8 Concluding Remarks

In this dissertation, we studied how to minimize the effects of synchronization on the scalability of concurrent software. To this end, we performed two analyses of synchronization on modern hardware, one involving traditional performance metrics, such as throughput and latency, and another focusing on the energy efficiency of locks. The results of these analyses indicate that scalability of synchronization is mainly a property of the underlying hardware, meaning that hardware imposes certain limitations that software cannot bypass.

Nevertheless, we further showed that although these hardware limitations cannot be bypassed by software, they can be hidden in generic and portable ways. In detail, we introduced OPTIK, a design pattern for devising scalable concurrent data structures and illustrated the effectiveness of OPTIK by designing five new algorithms. We also showed that these novel OPTIK-based algorithms are more scalable than the state of the art. Additionally, we introduced MCTOP, a multi-core topology abstraction which enables developers to portably define high-level performance policies. These policies utilize low-level characteristics of multi-cores, such as latencies and bandwidth, without exposing them to the programmer, resulting in portable software optimizations. We illustrated several such policies, such as automatic lock backoffs, and the portability of these policies across five processors.

8.1 Implications

Hardware. Modern multi-core hardware is clearly not optimized for synchronization (neither in terms of performance nor in terms of energy efficiency). Software developers either employ busy-waiting techniques, which burn the cores at full speed, or rely on thread sleeping, which is implemented in software and is thus particularly slow.

Recent efforts, such as the introduction of transactional synchronization extensions (TSX) [113] in Intel processors, attempted to simplify concurrent programming while delivering reasonable performance. However, our brief experience with TSX and related research [194, 217] indicate that programming with TSX is not straightforward, mainly because fall-back mechanisms are required in case transactions repeatedly fail to commit.[1]

As we have showed in this thesis, hardware largely dictates the expected behavior of synchronization Accordingly, based on the findings of this dissertation, we argue for several potential hardware improvements.

First, `monitor/mwait` must be exposed in user space.[2] Using these instructions properly can bring several benefits: (i) the waiting thread does not saturate the core with busy waiting, hence the other hardware thread(s) of the core can execute unobstructedly, (ii) waiting threads consume lower power, as the underlying core can enter a low-power state, (iii) waiting threads do not flood the memory subsystem with requests, hence a lock release (i.e., a memory store) can propagate faster, and (iv) `mwait` offers a performance/energy trade-off, as programmers can hint the processor how deep should be the sleep state that `mwait` uses. Of course, `monitor/mwait` is not a panacea because threads still keep hardware contexts occupied, causing significant problems, especially in oversubscribed configurations.

Second, the `x86` `pause` instruction must be redesigned to offer energy efficiency and more rich functionality. We argue that the duration of `pause` on Intel processors (i.e., a couple of cycles) is very low and does not bring any actual benefits. Ideally, the `pause` instruction should accept the pause duration as a parameter (similar to `wrpause` on recent `SPARC` processors [169]). Such functionality could bring two benefits: (i) developers can directly map backoff techniques to `pause`, and (ii) hardware can save energy proportional to the amount of time that a thread requests to pause for.

Furthermore, synchronization can benefit from other energy-related hardware advances, such as fast per-core DVFS transitions.[3] If cores can transition between voltage-frequency (VF) settings within a few cycles, then any type of busy waiting (e.g., in locks, thread barriers) can be immediately optimized for lower power consumption. Of course, this type of power optimizations should be preferably hidden behind more specialized instructions, such as `monitor/mwait` and `pause`, which we previously discussed.

Software. General purpose software systems, such as MySQL and Memcached, require portability on different platforms. Inevitably, this portability comes with the price of both wasted time and energy for synchronization. As we showed in this thesis, busy waiting wastes power and does not play well when many threads (from one or many applications) execute on the same processor, while sleeping significantly degrades performance. Additionally, we clearly showed that different synchronization algorithms perform well under different configurations.

[1] Actually, in our experience on *Haswell* (not described in this dissertation), there are cases were small transactions deterministically abort without any actual contention.

[2] Recent AMD and Oracle processors already support user-space `monitor/mwait`.

[3] As we describe in Chapter 2, DVFS stands for "dynamic voltage and frequency scaling."

Consequently, there is a need for algorithmic adaptivity in synchronization. Synchronization schemes must be able to adapt to the current configuration in order to perform well under varying platforms and workloads. Of course, implementing adaptivity is not straightforward for various reasons. (i) Which algorithms do we need to include? (ii) Which algorithms are suitable for a specific hardware platform? (iii) How frequently to adapt? and (iv) How to keep the overhead of adaptivity low?

Similarly, we optimized software using thread placement based on policies. These policies are deterministically defined on top of our MCTOP topology abstraction. For example, locality can be precisely defined on any platform as the latencies among cores can be accurately measured. However, thread and data placement is a subset of a larger, more complicated problem, namely thread scheduling. In detail, if every software application optimizes its thread placement locally (e.g., using `libmctop`), two or more applications on the same processor would probably contend for the same resources, resulting in suboptimal performance. Accordingly, we argue that OS schedulers should offer functionalities similar to `libmctop`, allowing applications to hint the scheduler about their needs. The OS scheduler can leverage the global view of the system and low-level information to take improve scheduling.

8.2 Future Research

As multi-core processors keep growing in terms of processing and memory capabilities, more scalable concurrent systems are necessary for leveraging the full potential of this hardware. Additionally, emerging memory technologies promise unprecedented software scalability. We would be very interested in taking the work of this dissertation one significant step further. Therefore, our future research directions revolve around "*How can we build software systems that scale both in terms of performance and energy efficiency?*" Below, we briefly describe three potential directions for answering this question.

Scaling Synchronization Further. Our experiences with optimizing systems (Chapters 4, 6 and 7) showed that (i) we can improve scalability with generic solutions (e.g., replacing locks), and (ii) we can significantly improve scalability with specialized solutions (e.g., our Metis optimizations). Accordingly, we would first like to further push the limits of locking. We are currently investigating how to design adaptive lock algorithms that adapt fast to the current contention levels and have very low overhead. In the future, we would also like to investigate adaptiveness to the specifics of the underlying hardware using `libmctop` (e.g., on large NUMA processors, under high contention, the adaptive lock should turn into a hierarchical lock). Additionally, we would like to further investigate how to design and apply specialized solutions (i.e., data structures, locks, and other patterns) for improving the scalability of system software. In this process, hardware transactional memory could help, as it allows for simple solutions to complex synchronization patterns which are not in the critical path (e.g., tree rebalancing).

Scaling Performance and Energy Efficiency Using Emerging Hardware. Emerging memory technologies, such as non-volatile memories and near-memory computing, open up great opportunities for improving the scalability of software systems. To achieve this scalability, we must develop new techniques and models for taking advantage of their capabilities.

In detail, non-volatile memory (NVM) provides persistent memory with latencies in between non-volatile flash storage and traditional volatile DRAM memory (NVMs are expected to be approximately 10x slower than DRAM—100x faster than flash memory). These characteristics of NVMs enable persistence without the performance costs induced by traditional techniques, such as write-ahead logging. Nevertheless, NVMs are still slower than DRAM memory, thus we cannot simply move data on NVM. We intend to work on models for exposing the persistence of NVM to the programmer. Two concrete approaches are to design persistent software transactional memory systems and concurrent data structures. These new models will still need to use DRAM for performance, but will have access to fast persistent NVM storage.

Recent advances in memory technologies have enabled the integration of low-power logic into conventional DRAM chips, reviving the old idea of near-memory processing (NMP) [91, 171, 175]. The logic inside the memory chips can leverage the abundant internal memory bandwidth and the proximity to the data to perform memory-intensive operations, minimizing the energy consumption by avoiding slow and energy-hungry off-chip communication. An interesting direction for NMP would be to offload complex access patterns that are ill-suited for modern processors—e.g., traversing graphs and linked data structures—which fully expose the memory latency. NMP raises thus the questions of (i) designing data structures that achieve the best efficiency on NMP-based systems, and (ii) how to efficiently implement operations on the limited capabilities of programmable NMPs.

Reducing Energy Consumption. Modern multi-core servers (from Intel and Oracle) include accurate energy measurements and voltage-frequency (VF) control. The granularity of these measurements and the control is still coarse grained, but hardware trends lean towards more fine-grained control. These capabilities will enable fine-grained power consumption monitoring and control. We would be interested in designing tools for accurate power profiling. We can then leverage these tools to improve the energy efficiency of software via, for example, thread scheduling. Additionally, as the latency of VF control reaches the nanosecond scale, we will be able to fine-tune energy efficiency by trading in performance for lower power. For example, spinning behind a lock can be performed in the lowest consumption state, while traversing a large linked data structure could deliver the maximum energy efficiency in an intermediate VF setting.

Bibliography

[1] José L. Abellán, Juan Fernández, and Manuel E. Acacio. GLocks: Efficient Support for Highly-Contended Locks In Many-Core CMPs. In *IPDPS*, pages 893–905. IEEE, 2011. [Page 3]

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling Parallel Programs By Work Stealing with Private Deques. In *PPOPP*, pages 219–228. ACM, 2013. [Page 26]

[3] Anant Agarwal and Mathews Cherian. Adaptive Backoff Synchronization Techniques. In *ISCA*, pages 396–406. ACM, 1989. [Pages 20, 123, and 126]

[4] Marcos Kawazoe Aguilera, Arif Merchant, Mehul A. Shah, Alistair C. Veitch, and Christos T. Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *SOSP*, pages 159–174. ACM, 2007. [Pages 24 and 83]

[5] Brian Aker. libmemcached. `http://libmemcached.org`. Accessed: 2016-07-29. [Page 51]

[6] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are Lock-Free Concurrent Algorithms Practically Wait-Free? In *STOC*, pages 714–723. ACM, 2014. [Page 23]

[7] AMD. Software Optimization Guide for AMD Family 10h and 12h Processors. `https://support.amd.com/TechDocs/40546.pdf`. Accessed 2016-07-29. [Page 35]

[8] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Spring Joint Computing Conference*, pages 483–485. AFIPS / ACM / Thomson Book Company, 1967. [Page 2]

[9] Nikos Anastopoulos and Nectarios Koziris. Facilitating Efficient Synchronization of Asymmetric Threads On Hyper-Threaded Processors. In *IPDPS*, pages 1–8. IEEE, 2008. [Page 63]

[10] Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Money Multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990. [Pages 3, 13, 14, 20, 36, and 126]

[11] Maya Arbel and Hagit Attiya. Concurrent Updates with RCU: Search Tree as an Example. In *PODC*, pages 196–205. ACM, 2014. [Page 23]

[12] Maya Arbel and Adam Morrison. Predicate RCU: an RCU for Scalable Concurrent Updates. In *PPOPP*, pages 21–30. ACM, 2015. [Page 23]

[13] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly In Message-Passing Systems. In *PODC*, pages 363–375. ACM, 1990. [Pages 14 and 21]

[14] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated. In *POPL*, pages 487–498. ACM, 2011. [Pages 10, 12, and 19]

[15] Woongki Baek and Trishul M. Chilimbi. Green: A Framework for Supporting Energy-Conscious Programming Using Controlled Approximation. In *PLDI*, pages 198–209. ACM, 2010. [Page 25]

[16] Luiz André Barroso and Urs Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12):33–37, 2007. [Pages 3 and 55]

[17] Andrew Baumann, Paul Barham, Pierre-Évariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, pages 29–44. ACM, 2009. [Pages 2, 3, 21, 24, 26, 37, and 111]

[18] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *OSDI*, pages 49–65. USENIX, 2014. [Page 26]

[19] Luca Benini, Mahmut Kandemir, and J Ramanujam. *Compilers and Operating Systems for Low Power*. Springer Science & Business Media, 2011. [Page 25]

[20] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Power and Performance Evaluation of Memcached on the TilePro64 Architecture. *Sustainable Computing: Informatics and Systems, Elsevier*, 2(2):81–90, 2012. [Page 26]

[21] Arnd Bergmann. BKL: That's All, Folks. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=4ba8216cd90560bc402f52076f64d8546e8aefcb`. Accessed: 2016-07-15. [Page 2]

[22] Robert D. Blumofe. Scheduling Multithreaded Computations By Work Stealing. In *FOCS*, pages 356–368. IEEE, 1994. [Page 123]

[23] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, 2013. Accessed: 2016-07-29. [Pages 26 and 129]

[24] Leonid B. Boguslavsky, Karim Harzallah, Alexander Y. Kreinin, Kenneth C. Sevcik, and Alexander Vainshtein. Optimal Strategies for Spinning and Blocking. *J. Parallel Distrib. Comput.*, 21(2):246–254, 1994. [Pages 3 and 21]

[25] Shekhar Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, 19(4):23–29, 1999. [Pages 9 and 123]

[26] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Commun. ACM*, 54(5):67–77, 2011. [Page 1]

[27] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai, Yang Zhang, and Zheng Zhang. Corey: An Operating System for Many Cores. In *OSDI*, pages 43–57. USENIX, 2008. [Pages 2, 3, 24, 26, and 111]

[28] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *OSDI*, pages 1–16. USENIX, 2010. [Pages 2, 3, 25, 26, 31, 111, and 128]

[29] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable Locks Are Dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012. [Pages 2 and 21]

[30] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A Practical Concurrent Binary Search Tree. In *PPOPP*, pages 257–268. ACM, 2010. [Pages 23, 81, and 106]

[31] François Broquedis, Jérôme Clet-Ortega, Stephanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *PDP*, pages 180–186. IEEE, 2010. [Page 26]

[32] David J. Brown and Charles Reams. Toward Energy-Efficient Computing. *Commun. ACM*, 53(3):50–58, 2010. [Page 123]

[33] Davidlohr Bueso. Scalability Techniques For Practical Synchronization Primitives. *Commun. ACM*, 58(1):66–74, 2015. [Pages 2 and 3]

[34] Irina Calciu, David Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. NUMA-Aware Reader-Writer Locks. In *PPOPP*, pages 157–166. ACM, 2013. [Page 26]

[35] Bryan Cantrill and Jeff Bonwick. Real-World Concurrency. *Commun. ACM*, 51(11):34–39, 2008. [Page 2]

[36] Calin Cascaval, Colin Blundell, Maged M. Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *ACM Queue*, 6(5):46–58, 2008. [Page 24]

[37] Milind Chabbi and John M. Mellor-Crummey. Contention-Conscious, Locality-Preserving Locks. In *PPOPP*, pages 1–14. ACM, 2016. [Page 26]

[38] Milind Chabbi, Michael W. Fagan, and John M. Mellor-Crummey. High Performance Locks for Multi-Level NUMA Systems. In *PPOPP*, pages 215–226. ACM, 2015. [Pages 20 and 26]

[39] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *SOSP*, pages 103–116. ACM, 2001. [Page 25]

[40] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture. *PVLDB*, 1(2):1313–1324, 2008. [Page 127]

[41] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating Systems Errors. In *SOSP*, pages 73–88. ACM, 2001. [Page 2]

[42] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable Address Spaces Using RCU Balanced Trees. In *ASPLOS*, pages 199–210. ACM, 2012. [Page 2]

[43] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert Tappan Morris, and Eddie Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *SOSP*, pages 1–17. ACM, 2013. [Pages 2, 3, and 25]

[44] Clem Cole and Russell Williams. Photoshop Scalability: Keeping it Simple. *Commun. ACM*, 53(10):32–38, 2010. [Page 2]

[45] 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. [Pages 12 and 35]

[46] Travis Craig. Building FIFO and Priority-Queuing Spin Locks From Atomic Swap. Technical report, University of Washington, Seattle, 1993. [Pages 3, 13, 14, and 20]

[47] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM By Abolishing Ownership Records. In *PPOPP*, pages 67–78. ACM, 2010. [Pages 24, 83, 88, and 89]

[48] Tudor David and Rachid Guerraoui. Concurrent Search Data Structures Can Be Blocking and Practically Wait-Free. In *SPAA*, pages 337–348. ACM, 2016. [Pages 15 and 23]

[49] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *SOSP*, pages 33–48. ACM, 2013. [Page 14]

[50] Tudor David, Rachid Guerraoui, Tong Che, and Vasileios Trigonakis. Designing ASCY-compliant Concurrent Search Data Structures. Technical report, EPFL, Lausanne, 2014. [Pages 82, 85, and 106]

[51] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *ASPLOS*, pages 631–644. ACM, 2015. [Pages 23, 52, 82, 85, 86, 100, 101, and 106]

[52] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150. USENIX, 2004. [Pages 1 and 123]

[53] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of Ion-Implanted MOSFET's With Very Small Physical Dimensions. *IEEE Solid-State Circuits*, 9(5):256–268, 1974. [Page 1]

[54] David Dice and Alex Garthwaite. Mostly Lock-Free Malloc. In *MSP/ISMM*, pages 269–280. ACM, 2002. [Page 2]

[55] David Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *DISC*, pages 194–208. Springer, 2006. [Pages 24, 83, and 88]

[56] David Dice, Virendra J. Marathe, and Nir Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *PPOPP*, pages 247–256. ACM, 2012. [Pages 3, 13, 14, 20, 26, and 37]

[57] Dana Drachsler, Martin T. Vechev, and Eran Yahav. Practical Concurrent Binary Search Trees Via Logical Ordering. In *PPOPP*, pages 343–356. ACM, 2014. [Page 23]

[58] Aleksandar Dragojevic and Tim Harris. STM in the Small: Trading Generality for Performance in Software Transactional Memory. In *EuroSys*, pages 1–14. ACM, 2012. [Page 24]

[59] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *NSDI*, pages 401–414. USENIX, 2014. [Pages 24 and 83]

[60] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck Van Breugel. Non-Blocking Binary Search Trees. In *PODC*, pages 131–140. ACM, 2010. [Page 23]

[61] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA*, pages 365–376. ACM, 2011. [Pages 3 and 55]

[62] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture Support for Disciplined Approximate Programming. In *ASPLOS*, pages 301–312. ACM, 2012. [Page 25]

[63] Facebook. Facebook LinkBench Benchmark. `https://github.com/facebook/linkbench`. Accessed: 2016-07-15. [Page 74]

[64] Facebook. RocksDB. `http://rocksdb.org`. Accessed: 2016-07-29. [Pages 13, 25, and 74]

[65] Facebook. RocksDB In-memory Workload Performance Benchmarks. https://github.com/facebook/rocksdb/wiki/RocksDB-In-Memory-Workload-Performance-Benchmarks. Accessed: 2016-07-29. [Page 74]

[66] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. Unlocking Energy. In *ATC*, pages 393–406. USENIX, 2016. [Page 14]

[67] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *NSDI*, pages 371–384. USENIX, 2013. [Pages 25 and 86]

[68] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the Combining Synchronization Technique. In *PPOPP*, pages 257–266. ACM, 2012. [Pages 22 and 53]

[69] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *PPOPP*, pages 237–246. ACM, 2008. [Pages 24 and 88]

[70] Brad Fitzpatrick. Memcached. http://www.memcached.org. Accessed: 2016-07-29. [Pages 13, 25, 51, and 74]

[71] Marc Fleischmann. LongRun Power Management. Technical report, White Paper of Transmeta Corporation, 2001. [Page 25]

[72] The Linux Foundation. Linux Kernel MCS lock. http://lxr.free-electrons.com/source/kernel/locking/mcs_spinlock.h. Accessed: 2016-07-29. [Page 13]

[73] The Linux Foundation. Linux Kernel x86 spinlock. http://lxr.free-electrons.com/source/arch/x86/include/asm/spinlock.h. Accessed: 2016-07-29. [Page 13]

[74] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *AUUG*, pages 85–98, 2002. [Page 21]

[75] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2004. [Pages 23, 104, and 105]

[76] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *OSDI*, pages 87–100. USENIX, 1999. [Pages 3, 24, and 26]

[77] Jana Giceva, Gustavo Alonso, Timothy Roscoe, and Tim Harris. Deployment of Query Plans on Multicores. *PVLDB*, 8(3):233–244, 2014. [Page 26]

[78] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. A Study of the Scalability of Stop-The-World Garbage Collectors On Multicores. In *ASPLOS*, pages 229–240. ACM, 2013. [Pages 26 and 111]

[79] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. NumaGiC: A Garbage Collector for Big Data on Big NUMA Machines. In *ASPLOS*, pages 661–673. ACM, 2015. [Page 26]

[80] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling Concurrent Log-Structured Data Stores. In *EuroSys*, pages 1–14. ACM, 2015. [Page 25]

[81] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-scale Cache-coherent Multiprocessors. In *ASPLOS*, pages 64–75. ACM, 1989. [Page 20]

[82] Google. LevelDB. `http://leveldb.org`. Accessed: 2016-07-29. [Page 25]

[83] Vincent Gramoli. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *PPOPP*, pages 1–10. ACM, 2015. [Page 23]

[84] Vincent Gramoli, Rachid Guerraoui, and Vasileios Trigonakis. TM^{2}C: A Software Transactional Memory for Many-Cores. In *EuroSys*, pages 351–364. ACM, 2012. [Page 22]

[85] Vincent Gramoli, Petr Kuznetsov, Srivatsan Ravi, and Di Shang. Brief Announcement: A Concurrency-Optimal List-Based Set. In *DISC*, 2015. [Page 23]

[86] Graphviz. Graphviz - Graph Visualization Software. `http://www.graphviz.org`. [Page 115]

[87] The Open Group. Pthread Mutex Lock. `http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_mutex_lock.html`, 1997. Accessed: 2016-07-29. [Page 14]

[88] Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In *PPOPP*, pages 175–184. ACM, 2008. [Page 89]

[89] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. Multicore Locks: The Case Is Not Closed Yet. In *ATC*, pages 649–662. USENIX, 2016. [Page 20]

[90] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *MICRO*, pages 413–422. ACM, 2009. [Page 19]

[91] Mary W. Hall, Peter M. Kogge, Jefferey G. Koller, Pedro C. Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John J. Granacki, Jay B. Brockman, Apoorv Srivastava, William C. Athas, Vincent W. Freeh, Jaewook Shin, and Joonseok Park. Mapping Irregular Applications to DIVA, a PIM-Based Data-Intensive Architecture. In *SC*, page 57. ACM, 1999. [Page 136]

[92] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward Dark Silicon in Servers. *IEEE Micro*, 31(4):6–15, 2011. [Pages 3 and 55]

[93] Tim Harris. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *DISC*, pages 300–314. Springer, 2001. [Pages 23, 81, 82, 93, and 101]

[94] Tim Harris and Stefan Kaestle. Callisto-RTS: Fine-Grain Parallel Loops. In *ATC*, pages 45–56. USENIX, 2015. [Page 26]

[95] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of Memory Reclamation for Lockless Synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, 2007. [Page 93]

[96] Bijun He, William N. Scherer III, and Michael L. Scott. Preemption Adaptivity in Time-Published Queue-Based Spin Locks. In *HiPC*, pages 7–18. Springer, 2005. [Pages 21 and 108]

[97] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. In *OPODIS*, pages 3–16. Springer, 2005. [Pages 5, 23, 83, 98, and 101]

[98] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A Scalable Lock-Free Stack Algorithm. In *SPAA*, pages 206–215. ACM, 2004. [Pages 23 and 109]

[99] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-Parallelism Tradeoff. In *SPAA*, pages 355–364. ACM, 2010. [Pages 22, 24, and 53]

[100] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012. [Pages 9, 10, and 11]

[101] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1): 124–149, 1991. [Pages 15 and 33]

[102] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, pages 289–300. ACM, 1993. [Pages 3 and 24]

[103] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised First Edition*. Elsevier, 2012. [Pages 2, 13, 14, 15, 36, and 98]

[104] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. [Page 15]

[105] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A Simple Optimistic Skiplist Algorithm. In *SIROCCO*, pages 124–138. Springer, 2007. [Pages 5, 81, 83, 84, 104, and 109]

[106] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, 2008. [Page 2]

[107] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. Green-Marl. `https://github.com/stanford-ppl/Green-Marl`. Accessed: 2016-07-29. [Page 130]

[108] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *ASPLOS*, pages 349–362. ACM, 2012. [Pages 113 and 130]

[109] Hiroshi Inoue and Kenjiro Taura. SIMD- and Cache-Friendly Algorithm for Sorting an Array of Structures. *PVLDB*, 8(11):1274–1285, 2015. [Page 127]

[110] Intel. Intel Xeon Processor E5-1600/ E5-2600/E5-4600 Product Families. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-1600-2600-vol-1-datasheet.pdf. Accessed: 2016-07-29. [Page 57]

[111] Intel. Intel 64 and IA-32 Architectures Software Developer Manuals. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. Accessed: 2016-07-29. [Pages 35, 58, 59, 62, 63, 100, 117, 122, 125, and 127]

[112] Intel. An Introduction to the Intel QuickPath Interconnect. https://www-ssl.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, 2009. Accessed: 2016-07-29. [Pages 12 and 35]

[113] Intel. Transactional Synchronization Extensions Overview. https://software.intel.com/en-us/node/524022, 2013. Accessed: 2016-07-29. [Pages 3 and 133]

[114] ITRS. 2015 International Technology Roadmap for Semiconductors (ITRS). http://www.semiconductors.org/main/2015_international_technology_roadmap_for_semiconductors_itrs. Accessed: 2016-07-26. [Page 1]

[115] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *EDBT*, pages 24–35. ACM, 2009. [Page 26]

[116] Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mowry. Decoupling Contention Management From Scheduling. In *ASPLOS*, pages 117–128. ACM, 2010. [Page 21]

[117] Alain Kägi, Doug Burger, and James R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *ISCA*, pages 170–180. ACM, 1997. [Page 20]

[118] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan S. Owicki. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *SOSP*, pages 41–55. ACM, 1991. [Page 21]

[119] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalability in the Clouds!: A Myth Or Reality? In *APSys*, pages 1–7. ACM, 2015. [Page 126]

[120] Andi Kleen. A NUMA API for Linux. *SUSE Labs white paper*, 2004. [Pages 26 and 111]

[121] Jonathan Koomey. Growth in Data Center Electricity Use 2005 to 2010. In *Analytics Press Report*, 2011. [Page 55]

[122] Konstantinos Koukos, David Black-Schaffer, Vasileios Spiliopoulos, and Stefanos Kaxiras. Towards More Efficient Execution: A Decoupled Access-Execute Approach. In *ICS*, pages 253–262. ACM, 2013. [Page 25]

[123] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981. [Pages 24 and 83]

[124] FAL Labs. Kyoto Cabinet. `http://fallabs.com/kyotocabinet`. Accessed: 2016-07-29. [Page 74]

[125] Christoph Lameter. Effective Synchronization on Linux/NUMA Systems. In *Gelato Federation Meeting*, 2005. [Page 24]

[126] Hugh C. Lauer and Roger M. Needham. On the Duality of Operating System Structures. *Operating Systems Review*, 13(2):3–19, 1979. [Pages 14 and 21]

[127] Doug Lea. Overview of Package util.concurrent Release 1.3.4. `http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html`, 2003. Accessed: 2016-07-29. [Pages 23, 102, 103, and 109]

[128] Hai Li, Swarup Bhunia, Yiran Chen, Kaushik Roy, and T. N. Vijaykumar. DCG: Deterministic Clock-Gating for Low-Power Microprocessor Design. *IEEE Trans. VLSI Syst.*, 12(3): 245–254, 2004. [Page 25]

[129] Beng-Hong Lim and Anant Agarwal. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. *ACM Trans. Comput. Syst.*, 11(3):253–294, 1993. [Page 21]

[130] Beng-Hong Lim and Anant Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *ASPLOS*, pages 25–35. ACM, 1994. [Page 20]

[131] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: a Memory-Efficient, High-Performance Key-Value Store. In *SOSP*, pages 1–13. ACM, 2011. [Page 25]

[132] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *NSDI*, pages 429–444. USENIX, 2014. [Pages 25 and 26]

[133] 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 *ISCA*, pages 36–47. ACM, 2013. [Page 74]

[134] Rick Lindsley and Dave Hansen. Bkl: One Lock to Bind Them All. In *Ottawa Linux Symposium*, pages 301–309, 2002. [Page 2]

[135] David Lo, Liqun Cheng, Rama Govindaraju, Luiz Andre Barroso, and Christos Kozyrakis. Towards Energy Proportionality for Large-Scale Latency-Critical Workloads. In *ISCA*, pages 301–312. IEEE, 2014. [Page 65]

[136] 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 *ATC*, pages 65–76. USENIX, 2012. [Pages 2, 3, 20, 22, 31, and 53]

[137] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia L. Lawall, and Gilles Muller. Fast and Portable Locking for Multicore Architectures. *ACM Trans. Comput. Syst.*, 33(4):13, 2016. [Pages 22 and 53]

[138] Jean-Pierre Lozi, Baptiste Lepers, Justin R. Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux Scheduler: A Decade of Wasted Cores. In *EuroSys*, pages 1–16. ACM, 2016. [Page 25]

[139] Victor Luchangco, Daniel Nussbaum, and Nir Shavit. A Hierarchical CLH Queue Lock. In *Euro-Par*, volume 4128, pages 801–810. Springer, 2006. [Pages 3, 13, 14, and 20]

[140] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *IPPS*, pages 165–171. IEEE, 1994. [Pages 14 and 20]

[141] Zoltan Majo and Thomas R. Gross. A Library for Portable and Composable Data Locality Optimizations for NUMA Systems. In *PPOPP*, pages 227–238. ACM, 2015. [Page 26]

[142] Yandong Mao, Robert Morris, and M Frans Kaashoek. Optimizing MapReduce for Multicore Architectures. Technical report, MIT-CSAIL-TR-2010-020, MIT, 2010. [Page 128]

[143] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys*, pages 183–196. ACM, 2012. [Page 25]

[144] Henry Massalin and Calton Pu. A Lock-Free Multiprocessor OS Kernel. *Operating Systems Review*, 26(2):8, 1992. [Page 2]

[145] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-Log-Update: A Lightweight Synchronization Mechanism For Concurrent Programming. In *SOSP*, pages 168–183, 2015. [Pages 2, 3, and 24]

[146] Paul E McKenney and John D Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *PDCS*, pages 509–518, 1998. [Pages 23 and 93]

[147] David Meisner and Thomas F. Wenisch. DreamWeaver: Architectural Support for Deep Sleep. In *ASPLOS*, pages 313–324. ACM, 2012. [Page 25]

[148] David Meisner, Brian T. Gold, and Thomas F. Wenisch. PowerNap: Eliminating Server Idle Power. In *ASPLOS*, pages 205–216. ACM, 2009. [Page 25]

[149] 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. [Pages 3, 13, 14, 20, 41, and 126]

[150] John M. Mellor-Crummey and Michael L. Scott. Synchronization without Contention. In *ASPLOS*, pages 269–278. ACM, 1991. [Page 3]

[151] Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *SPAA*, pages 73–82, 2002. [Pages 23 and 81]

[152] Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004. [Page 93]

[153] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*, pages 267–275. ACM, 1996. [Pages 23, 42, 81, 84, 100, 106, 109, and 127]

[154] Sun Microsystems. Multithreading in the Solaris Operating Environment. `http://home.mit.bme.hu/~meszaros/edu/oprendszerek/segedlet/unix/2_folyamatok_es_utemezes/solaris_multithread.pdf`. Accessed: 2016-07-29. [Page 21]

[155] Sun Microsystems. UltraSPARC T2 Supplement to the UltraSPARC Architecture. `http://www.oracle.com/technetwork/systems/opensparc/t2-13-ust2-uasuppl-draft-p-ext-1537760.html`, 2007. Accessed: 2016-07-29. [Page 36]

[156] Daniel Molka, Robert Schöne, Daniel Hackenberg, and Matthias S. Müller. Memory Performance and SPEC OpenMP Scalability on Quad-Socket x86_64 Systems. In *ICA3PP*, pages 170–181. Springer, 2011. [Page 19]

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

[158] Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. Energy Implications of Multiprocessor Synchronization. In *SPAA*, page 329. ACM, 2006. [Page 21]

[159] Adam Morrison and Yehuda Afek. Fast Concurrent Queues for x86 Processors. In *PPOPP*, pages 103–112. ACM, 2013. [Pages 23 and 81]

[160] Jaideep Moses, Ramesh Illikkal, Li Zhao, Srihari Makineni, and Don Newell. Effects of Locking and Synchronization on Future Large Scale CMP Platforms. In *CAECW Workshop, along with HPCA*, 2006. [Page 19]

[161] Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-Free Binary Search Trees. In *PPOPP*, pages 317–328. ACM, 2014. [Pages 23, 81, 82, and 106]

[162] Umesh Gajanan Nawathe, Mahmudul Hassan, King C Yen, Ashok Kumar, Aparna Ramachandran, and David Greenhill. Implementation of an 8-Core, 64-Thread, Power-Efficient SPARC Server on a Chip. *IEEE SSC*, 43(1):6–20, 2008. [Page 36]

[163] 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 *NSDI*, pages 385–398. USENIX, 2013. [Page 25]

[164] NRDC. America's Data Centers Consuming and Wasting Growing Amounts of Energy. `http://www.nrdc.org/energy/data-center-efficiency-assessment.asp`. Accessed: 2016-07-29. [Page 55]

[165] Takeshi Ogasawara. NUMA-Aware Memory Manager with Dominant-Thread-Based Copying GC. In *OOPSLA*, pages 377–390. ACM, 2009. [Page 26]

[166] Oracle. Java CopyOnWriteArrayList Data Structure. `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html`. Accessed: 2016-07-29. [Page 56]

[167] Oracle. Memory and Thread Placement Optimization Developer's Guide. `https://docs.oracle.com/cd/E26502_01/html/E35301/toc.html`. Accessed: 2016-07-15. [Pages 26 and 111]

[168] Oracle. MySQL. `http://www.mysql.com`. Accessed: 2016-07-29. [Pages 13 and 74]

[169] Oracle. SPARC T4 Supplement to the Oracle SPARC Architecture 2011. `http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/120214-t4-d04-p-ext-2305974.pdf`, 2012. Accessed: 2016-07-29. [Pages 127 and 134]

[170] Oracle. ConcurrentHashMap in Java Docs. `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html`, 2015. Accessed: 2016-07-29. [Page 103]

[171] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *ISCA*, pages 192–203. IEEE / ACM, 1998. [Page 136]

[172] John K Ousterhout. Scheduling Techniques for Concurrent Systems. In *ICDCS*, pages 22–30, 1982. [Pages 13, 21, and 59]

[173] Venkatesh Pallipadi and Alexey Starikovskiy. The Ondemand Governor. In *Ottawa Linux Symposium*, pages 223–238, 2006. [Page 25]

[174] Mark S. Papamarcos and Janak H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *ISCA*, pages 348–354. ACM, 1984. [Pages 10 and 116]

[175] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent RAM: IRAM. *IEEE Micro*, 17(2):34–44, 1997. [Page 136]

[176] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. In *OSDI*, pages 1–16. USENIX, 2014. [Page 26]

[177] Darko Petrovic, Thomas Ropars, and André Schiper. Leveraging Hardware Message Passing for Efficient Thread Synchronization. In *PPOPP*, pages 143–154. ACM, 2014. [Pages 22, 23, and 53]

[178] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pinar Tözün, and Anastasia Ailamaki. OLTP on Hardware Islands. *PVLDB*, 5(11):1447–1458, 2012. [Page 26]

[179] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and TN Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *ISLPED*, pages 90–95, 2000. [Page 25]

[180] The GNU Project. GNU libgomp. `https://gcc.gnu.org/onlinedocs/libgomp`. Accessed: 2016-07-15. [Page 129]

[181] Iraklis Psaroudakis, Thomas Kissinger, Danica Porobic, Thomas Ilsche, Erietta Liarou, Pinar Tözün, Anastasia Ailamaki, and Wolfgang Lehner. Dynamic Fine-Grained Scheduling for Energy-Efficient Main-Memory Queries. In *DaMoN*, pages 1–7. ACM, 2014. [Page 25]

[182] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-Aware Data and Task Placement. *PVLDB*, 8(12):1442–1453, 2015. [Page 26]

[183] William Pugh. Concurrent Maintenance of Skip Lists. Technical report, Report No. UMIACS-TR-90-80, University of Maryland, 1990. [Pages 23 and 86]

[184] Zoran Radovic and Erik Hagersten. Efficient Synchronization for Nonuniform Communication Architectures. In *SC*, pages 1–13. IEEE, 2002. [Page 20]

[185] Zoran Radovic and Erik Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *HPCA*, pages 241–252. IEEE, 2003. [Page 20]

[186] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *MICRO*, pages 294–305. ACM/IEEE, 2001. [Page 24]

[187] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-Core and Multiprocessor Systems. In *HPCA*, pages 13–24. IEEE, 2007. [Page 3]

[188] Haris Ribic and Yu David Liu. Energy-Efficient Work-Stealing Language Runtimes. In *ASPLOS*, pages 513–528. ACM, 2014. [Page 25]

[189] Efi Rotem, Alon Naveh, Micha Moffie, and Avi Mendelson. Analysis of Thermal Monitor Features of the Intel Pentium M Processor. In *TACS Workshop at ISCA*, 2004. [Page 25]

[190] Amitabha Roy, Steven Hand, and Tim Harris. A Runtime System for Software Lock Elision. In *EuroSys*, pages 261–274. ACM, 2009. [Page 24]

[191] Christoph Rupp. HamsterDB. `http://hamsterdb.com`. Accessed: 2016-07-29. [Page 74]

[192] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In *PLDI*, pages 164–174. ACM, 2011. [Page 25]

[193] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. In *SOSP*, pages 83–90. ACM, 1989. [Page 3]

[194] Michael L. Scott. Transactional Memory Today. *SIGACT News*, 46(2):96–104, 2015. [Page 134]

[195] Michael L. Scott and William N. Scherer III. Scalable Queue-Based Spin Locks with Timeout. In *PPOPP*, pages 44–52. ACM, 2001. [Page 3]

[196] Ori Shalev and Nir Shavit. Split-Ordered Lists: Lock-Free Extensible Hash Tables. In *PODC*, pages 102–111. ACM, 2003. [Page 81]

[197] Nir Shavit and Dan Touitou. Software Transactional Memory. In *PODC*, pages 204–213. ACM, 1995. [Pages 3 and 24]

[198] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. Power Containers: An OS Facility for Fine-Grained Power and Energy Management on Multicore Servers. In *ASPLOS*, pages 65–76. ACM, 2013. [Page 25]

[199] 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. [Page 25]

[200] Johannes Singler and Benjamin Konsik. The GNU libstdc++ Parallel Mode: Software Engineering Considerations. In *Workshop on Multicore Software Engineering*, pages 15–22. ACM, 2008. [Page 127]

[201] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011. [Pages 2, 11, and 117]

[202] Håkan Sundell and Philippas Tsigas. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. In *IPDPS*, page 84. IEEE, 2003. [Page 23]

[203] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):202–210, 2005. [Pages 1 and 9]

[204] SQLite Development Team. SQLite. `http://sqlite.org`. Accessed: 2016-07-29. [Page 74]

[205] Tilera. Tilera TILE-Gx. https://www.mellanox.com/related-docs/prod_multi_core/PB_TILE-Gx36.pdf. Accessed: 2016-07-29. [Pages 17 and 36]

[206] R Kent Treiber. Systems Programming: Coping with Parallelism. Technical report, 1986. [Pages 23 and 109]

[207] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *ICPP Workshops*, pages 207–216. IEEE, 2010. [Page 27]

[208] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, Scalable, Concurrent Hash Tables Via Relativistic Programming. In *ATC*, pages 145–158. USENIX, 2011. [Page 23]

[209] Jessica H. Tseng, Hao Yu, Shailabh Nagar, Niteesh Dubey, Hubertus Franke, Pratap Pattnaik, Hiroshi Inoue, and Toshio Nakatani. Performance Studies of Commercial Workloads on a Multi-Core System. In *IISWC*, pages 57–65. IEEE, 2007. [Page 3]

[210] Philippas Tsigas and Yi Zhang. A Simple, Fast and Scalable Non-Blocking Concurrent FIFO Queue For Shared Memory Multiprocessor Systems. In *SPAA*, pages 134–143. ACM, 2001. [Page 23]

[211] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A. Shah. Analyzing the Energy Efficiency of a Database Server. In *SIGMOD*, pages 231–242. ACM, 2010. [Page 25]

[212] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ISCA*, pages 392–403. ACM, 1995. [Page 11]

[213] Jons-Tobias Wamhoff, Stephan Diestelhorst, Christof Fetzer, Patrick Marlier, Pascal Felber, and Dave Dice. The TURBO Diaries: Application-Controlled Frequency Scaling Explained. In *ATC*, pages 193–204. USENIX, 2014. [Pages 11, 21, and 63]

[214] David Wentzlaff and Anant Agarwal. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *Operating Systems Review*, 43(2):76–85, 2009. [Pages 3, 21, and 24]

[215] Thomas Willhalm, Roman Dementiev, and Patrick Fay. Intel Performance Counter Monitor-A Better Way to Measure CPU Utilization. http://www.intel.com/software/pcm. Accessed: 2016-07-29. [Page 27]

[216] Qiang Wu, Margaret Martonosi, Douglas W. Clark, Vijay Janapa Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David M. Brooks. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In *MICRO*, pages 271–282. IEEE, 2005. [Page 25]

[217] Lingxiang Xiang and Michael L. Scott. Software Partitioning of Hardware Transactions. In *PPOPP*, pages 76–86. ACM, 2015. [Page 134]

[218] Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits. In *PLDI*, pages 49–62. ACM, 2003. [Page 25]

[219] Chao Xu, Felix Xiaozhu Lin, Yuyang Wang, and Lin Zhong. Automated OS-Level Device Runtime Power Management. In *ASPLOS*, pages 239–252. ACM, 2015. [Page 25]

[220] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling Cores, Kernels, and Operating Systems. In *OSDI*, pages 17–31. USENIX, 2014. [Page 26]

[221] Yan Zhai, Xiao Zhang, Stéphane Eranian, Lingjia Tang, and Jason Mars. HaPPy: Hyperthread-Aware Power Profiling Dynamically. In *ATC*, pages 211–217. USENIX, 2014. [Page 25]

[222] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *OSDI*, pages 465–477. USENIX, 2014. [Page 26]

Vasileios Trigonakis

Contact Information

Bat INR 312, Station 14
EPFL IC LPD
1015 Lausanne, Switzerland

Tel: 0041 216 938 121
E-mail: vasileios.trigonakis@epfl.ch
Web: people.epfl.ch/vasileios.trigonakis

Research Interests

software/hardware systems, concurrent programming, synchronization, data structures, distributed systems, transactional memory

Education

EPFL, Lausanne, Switzerland

Ph.D. Candidate, Computer Science — September 2011–October 2016

- Dissertation Title: "Towards Scalable Synchronization on Multi-Cores"
- Supervisor: Rachid Guerraoui

KTH, Stockholm, Sweden

M.Sc., Software Engineering of Distributed Systems — September 2009–August 2011

- Graduated first in my class
- Thesis: "Design of a Distributed Transactional Memory for Many-core Systems"
- Supervisor: Rachid Guerraoui | Examiner: Seif Haridi

NTUA, Athens, Greece

Dipl.-Ing., Electrical and Computer Engineering — September 2003–October 2008

- Thesis: "Design of a GIS System for Automated Filing and Presentation of Electromagnetic Field Power Measurements"
- Supervisor: Philippos Konstantinou

Publications

(author names in alphabetical order)

Unlocking Energy
Babak Falsafi, Rachid Guerraoui, Javier Picorel, Vasileios Trigonakis
USENIX ATC '16 (USENIX Annual Technical Conference)

Optimistic Concurrency with OPTIK
Rachid Guerraoui, Vasileios Trigonakis
PPoPP '16 (Symposium on Principles and Practice of Parallel Programming)

Locking Made Easy
Jelena Antic, Georgios Chatzopoulos, Rachid Guerraoui, Vasileios Trigonakis
Middleware '16 (Annual Middleware Conference)

Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures
Tudor David, Rachid Guerraoui, Vasileios Trigonakis
ASPLOS '15 (International Conference on Architectural Support for Programming Languages and Operating Systems)

Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask
Tudor David, Rachid Guerraoui, Vasileios Trigonakis
SOSP '13 (Symposium on Operating Systems Principles)

TM^{2}C: a Software Transactional Memory for Many-Cores
Vincent Gramoli, Rachid Guerraoui, Vasileios Trigonakis
EuroSys '12 (European Conference on Computer Systems)

INTERNSHIPS

Oracle Labs, Cambridge, UK

Research Intern — August 2014–November 2014

- Project: "Designing Tools and Libraries for Better Leveraging Multi-Cores"
- Supervisor: Tim Harris

EPFL, Lausanne, Switzerland

Research Intern — February 2011–August 2011

- Conducted my M.Sc. thesis

PRESENTATIONS & INVITED TALKS

Unlocking Energy
- Conference presentation at USENIX ATC, Denver, Colorado, USA, June 2016

Optimistic Concurrency with OPTIK
- Conference presentation at PPoPP, Barcelona, Spain, March 2016
- Invited talk at Cisco, Lausanne, Switzerland, November 2015

Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures
- Invited talk at MSR, Cambridge, UK, June 2015
- Conference presentation at ASPLOS, Istanbul, Turkey, April 2015

Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask
- Winter School on Hot Topics in Distributed Computing, La Plagne, France, March 2014
- Conference presentation at SOSP, Farmington, Pennsylvania, USA, November 2013
- Invited talk at MIT, Boston, Massachusetts, USA, November 2013
- Invited talk at University of Pennsylvania, Philadelphia, Pennsylvania, USA, November 2013
- EcoCloud Annual Event, Lausanne, Switzerland, June 2013

TM^{2}C: a Software Transactional Memory for Many-Cores
- School on Research Directions in Distributed Computing, Heraklion, Greece, June 2013
- Conference presentation at EuroSys, Bern, Switzerland, April 2012
- ASPLOS Doctoral Workshop, London, UK, March 2012
- Euro-TM Workshop on Distributed Transactional Memory, Lisbon, Portugal, February 2012
- Winter School on Hot Topics in Distributed Computing, La Plagne, France, March 2011 & 2012

SOFTWARE PROJECTS

Available at github.com/trigonak and github.com/LPD-EPFL:
- LOCKIN, a library with various lock algorithms optimized for energy efficiency
- ASCYLIB & OPTIK, a concurrent data structure library with over 40 implementations
- CLHT, a very efficient and scalable concurrent hash table
- ccbench, a tool for measuring the cache-coherence latencies of a processor
- libslock, a cross-platform atomic operation and lock algorithm library
- ssmp, a message passing library built on top of cache-coherent shared memory
- TM^{2}C, a software transactional memory built on top of message passing

PEER REVIEWS

- *External reviewer*, SPAA (Symposium on Parallelism in Algorithms and Architectures), 2016
- *PC member for artifact evaluation*, PPoPP (Symposium on Principles and Practice of Parallel Programming), 2015 & 2016
- *External reviewer*, IPDPS (International Parallel and Distributed Processing Symposium), 2015
- *External reviewer*, ICDCS (International Conference on Distributed Computing Systems), 2014
- *External reviewer*, ICDCN (International Conference on Distributed Computing and Networking), 2014
- *Shadow PC member*, EuroSys (European Conference on Computer Systems), 2013
- *External reviewer*, DISC (International Symposium on Distributed Computing), 2011

TEACHING & MENTORING

Teaching

- Concurrent Algorithms (CS-453), Graduate class, EPFL, 2012–2015
- Real-time Networks (COM-413), Graduate class, EPFL, 2014
- Distributed Algorithms (CS-451), Graduate class, EPFL, 2012

Mentoring

- *M.Sc. thesis*, Egeyar Bagcioglu, "Using Hardware Transactional Memory in Concurrent Data Structures," February 2016–June 2016
- *Semester project*, Daniel Vargas, "Porting ASCYLIB to C++," February 2016–June 2016
- *Semester project*, Sebastien Rouault, "Porting ASCYLIB to Go," February 2016–June 2016
- *Semester project*, Alexandru Ciprian, "gcmalloc: Memory Allocation with Garbage Collection," September 2015–January 2016
- *M.Sc. thesis*, Karolos Antoniadis, "Devising a Theory for Asynchronized Concurrency," April 2015–November 2015
- *M.Sc. thesis*, Jelena Antic, "GLS: A Generic Locking Service," February 2015–June 2015
- *Semester project*, Ivo Mihailovic, "Improving the Scalability of Memcached with CLHT," February 2015–June 2015
- *Semester project*, Nemanja Djurkic, "wlock: Doing Work Instead of Waiting in Locks," February 2015–June 2015
- *Semester project*, Egeyar Bagcioglu, "Implementing Randomized Concurrent Data Structures," February 2015–June 2015
- *Semester project*, Chengzhen Wu, "Cross-Platform Implementations of Barrier Algorithms," February 2014–June 2014
- *Semester project*, Radmila Popovic,
  - "Optimizing Memcached with CLHT," September 2015–January 2016
  - "Cross-platform Implementations of Reader-Writer Locks," February 2014–June 2014
- *Semester project*, David Cervini, "Porting TM^{2}C to Pthreads," September 2013–January 2014
- *Semester project*, Oana Balmau & Igor Zablotchi,
  - "Increasing Concurrency of RocksDB," February 2014–June 2014
  - "Concurrent Binary Search Trees on Many-Cores," September 2013–January 2014
- *Research project*, Ugur Gurel, "Designing Scalable Concurrent Hash Tables," September 2012–February 2013

LANGUAGE SKILLS

Greek: native speaker
English: fluent
French: basic communication skills

PROFESSIONAL EXPERIENCE

Pansystems S.A., Athens, Greece

Technical Consultant — July 2009

Undertook the software setup of a GIS-based system for the national intelligence service (EYP) of Greece. Designed, installed, and tuned an Oracle database 11g on a Windows server 2008 cluster of two servers with Oracle Failsafe. Installed and configured two Oracle application servers and the GIS software running on top.

Geo Information S.A., Athens, Greece

Technical Consultant — February 2009–June 2009

Directed the promotion, installation, and modification of the enterprise series of GIS products of Erdas Inc. The products use Oracle database with spatial option, Oracle application server, Oracle MapViewer, and Java SE & EE. Improved the GIS of the cities of Zografou, Giannitsa, Koropi, and Perama in Greece.

PROFESSIONAL EXPERIENCE

Ministry of Transportation and Communication, Athens, Greece

Call Services — April 2007–March 2008

Organized the appointments for the national vehicle control centers of Athens, Greece. Collaborated with a team of 16 people and, often, supervised the team.

EXTRACURRICULAR ACTIVITIES

- I love photography and football, I like cooking, and I have started liking biking
- In the past, I played football and volleyball on various (local, school, and university) teams
- Wolfram|Alpha's beta tester
- Member of Stockholm's Erlang Users' Group (2010–2011)