Understanding the performance of mutual exclusion algorithms on modern multicore machines.

Hugo Guiroux

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Hugo GUIROUX

Thèse dirigée par **Vivien QUEMA**, GRENOBLE INP
et codirigée par **Renaud LACHAIZE**, UNIV. GRENOBLE ALPES
préparée au sein du **Laboratoire Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Comprendre la performance des algorithmes d'exclusion mutuelle sur les machines multicoeurs modernes

Understanding the performance of mutual exclusion algorithms on modern multicore machines.

Thèse soutenue publiquement le **17 décembre 2018**,
devant le jury composé de :

Monsieur TIMOTHY HARRIS
INGENIEUR, AMAZON A CAMBRIDGE - ROYAUME-UNI, Rapporteur
Monsieur GAËL THOMAS
PROFESSEUR, TELECOM SUD-PARIS, Rapporteur
Monsieur ANDRZEJ DUDA
PROFESSEUR, GRENOBLE INP, Président
Monsieur PASCAL FELBER
PROFESSEUR, UNIVERSITE DE NEUCHATEL - SUISSE, Examinateur
Monsieur VIVIEN QUEMA
PROFESSEUR, GRENOBLE INP, Directeur de thèse
Monsieur RENAUD LACHAIZE
MAITRE DE CONFERENCES, UNIVERSITE GRENOBLE ALPES, Examinateur

We have arranged the whole system as a society of sequential processes whose harmonious cooperation is regulated by explicit mutual synchronization statements.
— Edsger W. Dijkstra.

To my wife, Justine...

Acknowledgements

First of all, I would like to thank the members of my Ph.D. thesis jury, Dr. Harris, Pr. Thomas, Pr. Duda and Pr. Felber, which have taken the time to examine my work and made meaningful comments and remarks before and during the defense. It was an honour for me to present my work in front of such a prestigious jury. I would also like to thank my two Ph.D. advisors, Renaud Lachaize and Vivien Quéma, for the long hours that we spent together discussing research ideas, profound scientific questions as well as non-scientific related topics. They both taught me to recognize what is a great research paper, to what standards my work should hold and how to make strong and interesting scientific work. They were here to support me during the difficult yet exciting experience that a Ph.D. can be.

Doing this Ph.D. would have been less pleasurable without the member of the ERODS team, with which I had countless discussions. More precisely, I want to thank (in alphabetical order): Maha Alsayasneh, Fabienne Boyer, Thomas Calmant, Matthieu Caneill, Jeremie Decouchant, Amadou Diarra, Didier Donsez, Christopher Ferreira, Soulaimane Guedria, Olivier Gruber, Mohamad Jaafar, Vikas Jaiman, Vincent Lamotte, Vania Marangozova-Martin, Muriel Nguimtsa, Nicolas Palix, Noel de Palma, Albin Petit, Ahmed El Rheddhane and Thomas Ropars. This thesis would not have happened without the financial support of the LIG and the University Grenoble Alpes.

During this three years, I had the chance to do an internship inside the MLE team at Oracle Labs Zurich. This experience brought me a lot and opened my mind to other interesting research topics. First and foremost, I would like to thank Vasileios Trigonakis for being a friend, helping me and advising me when I needed it. Thanks to the MLE team and the Oracle Labs: Matthias Branter, Lucas Braun, Laurent Daynès, Bastian Hossbach, Alexander Ulrich and Alexander Schubert.

A big shout out to all my friends that were here to support me during the three long years. Without you, I would not have done the Ph.D. that I did, even if you were not directly involved in my work. All the week-ends and holidays that we spent together helped me tremendously. A Ph.D. is a long journey, friends and family are the strong foundation that I needed. Thanks to Fairouz Azzoune, Carla Balbo, Perrine Blachon, Yoann Blein, Sandro Chionna, Jeanne Detroye, Aurélie Estermann, Clément Louis, Jennyfer Martinez, Simon Moura, Célia Muffato. Family is important: they provide a safe environment where anyone can talk openly about their feeling without being judged. I would like to thank both my family and my in-laws for their support

throughout the years. A special thank to my sister, who told me to not always take things too seriously and enjoy the life that I had, and to my father, who always gives me invaluable advices and who taught me to always be curious about everything.
Finally, a final word for the person that is the most important in my life: Justine. Without you, I would not have done this Ph.D., and I would not have accomplished a lot of things that happened in my life. You were always here to support me and encourage me in everything that I tried to accomplish. You always believed in me, even when I was not. Thank you for everything that you did.

Zurich, 05 January 2019 H. G.

Preface

This thesis presents the research conducted in the ERODS team at Laboratoire d'Informatique de Grenoble, to pursue a Ph.D. in Computer Science from the doctoral school "Mathématiques, Sciences et Technologies de l'Information, Informatique" of the Université Grenoble-Alpes. My research activities have been supervised by Vivien Quéma (ERODS / Grenoble INP) and Renaud Lachaize (ERODS / Université Grenoble Alpes). Some of the works presented in this thesis have been done in collaboration with members of the Distributed Programming Laboratory (LPD) of Ecole Polytechnique Fédérale de Lausanne (EPFL) [69].

This thesis studies and analyzes mutual exclusion lock algorithms on multicore applications running on NUMA multicore machines.

This work led to the following publications *(authors are in alphabetical order)*:

- Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. "Multicore Locks: The Case Is Not Closed Yet". *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.* Ed. by Ajay Gulati and Hakim Weatherspoon. USENIX Association, 2016, pp. 649–662
- Rachid Guerraoui, Hugo Guiroux, Renaud Lachaize, Vivien Quéma, and Vasileios Trigonakis. "Lock – Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems". *ACM Transaction on Computer System* (2018) *(to appear)*

Grenoble, Summer 2018 H. G.

Abstract

A plethora of optimized mutual exclusion lock algorithms have been designed over the past 25 years to mitigate performance bottlenecks related to critical sections and synchronization. Unfortunately, there is currently no broad study of the behavior of these optimized lock algorithms on realistic applications that consider different performance metrics, such as energy efficiency and tail latency. In this thesis, we perform a thorough and practical analysis, with the goal of providing software developers with enough information to achieve fast, scalable and energy-efficient synchronization in their systems. First, we provide a performance study of 28 state-of-the-art mutex lock algorithms, on 40 applications, and four different multicore machines. We not only consider throughput (traditionally the main performance metric), but also energy efficiency and tail latency, which are becoming increasingly important. Second, we present an in-depth analysis in which we summarize our findings for all the studied applications. In particular, we describe nine different lock-related performance bottlenecks, and propose six guidelines helping software developers with their choice of a lock algorithm according to the different lock properties and the application characteristics.

From our detailed analysis, we make a number of observations regarding locking algorithms and application behaviors, several of which have not been previously discovered: (i) applications not only stress the lock/unlock interface, but also the full locking API (e.g., trylocks, condition variables), (ii) the memory footprint of a lock can directly affect the application performance, (iii) for many applications, the interaction between locks and scheduling is an important application performance factor, (iv) lock tail latencies may or may not affect application tail latency, (v) no single lock is systematically the best, (vi) choosing the best lock is difficult (as it depends on many factors such as the workload and the machine), and (vii) energy efficiency and throughput go hand in hand in the context of lock algorithms. These findings highlight that locking involves more considerations than the simple "lock – unlock" interface and call for further research on designing low-memory footprint adaptive locks that fully and efficiently support the full lock interface, and consider all performance metrics.

Résumé

Une multitude d'algorithmes d'exclusion mutuelle ont été conçus au cours des vingt cinq dernières années, dans le but d'améliorer les performances liées à l'exécution de sections critiques et aux verrous. Malheureusement, il n'existe actuellement pas d'étude générale et complète au sujet du comportement de ces algorithmes d'exclusion mutuelle sur des applications réalistes (par opposition à des applications synthétiques) qui considère plusieurs métriques de performances, telles que l'efficacité énergétique ou la latence. Dans cette thèse, nous effectuons une analyse pragmatique des mécanismes d'exclusion mutuelle, dans le but de proposer aux développeurs logiciels assez d'informations pour leur permettre de concevoir et/ou d'utiliser des mécanismes rapides, qui passent à l'échelle et efficaces énergétiquement.

Premièrement, nous effectuons une étude de performances de 28 algorithmes d'exclusion mutuelle faisant partie de l'état de l'art, en considérant 40 applications et quatre machines multicoeurs différentes. Nous considérons non seulement le débit (la métrique de performance traditionnellement considérée), mais aussi l'efficacité énergétique et la latence, deux facteurs qui deviennent de plus en plus importants. Deuxièmement, nous présentons une analyse en profondeur de nos résultats. Plus particulièrement, nous décrivons neufs problèmes de performance liés aux verrous et proposons six recommandations aidant les développeurs logiciels dans le choix d'un algorithme d'exclusion mutuelle, se basant sur les caractéristiques de leur application ainsi que les propriétés des différents algorithmes.

A partir de notre analyse détaillée, nous faisons plusieurs observations relatives à l'interaction des verrous et des applications, dont plusieurs d'entre elles sont à notre connaissance originales : (i) les applications sollicitent fortement les primitives lock/unlock mais aussi l'ensemble des primitives de synchronisation liées à l'exclusion mutuelle (ex. trylocks, variables de conditions), (ii) l'empreinte mémoire d'un verrou peut directement impacter les performances de l'application, (iii) pour beaucoup d'applications, l'interaction entre les verrous et l'ordonnanceur du système d'exploitation est un facteur primordial de performance, (iv) la latence d'acquisition d'un verrou a un impact très variable sur la latence d'une application, (v) aucun verrou n'est systématiquement le meilleur, (vi) choisir le meilleur verrou est difficile, et (vii) l'efficacité énergétique et le débit vont de pair dans le contexte des algorithmes d'exclusion mutuelle.

Ces découvertes mettent en avant le fait que la synchronisation à base de verrou ne

se résume pas seulement à la simple interface “lock – unlock”. En conséquence, ces résultats appellent à plus de recherche dans le but de concevoir des algorithmes d’exclusion mutuelle avec une empreinte mémoire faible, adaptatifs et qui implémentent l’ensemble des primitives de synchronisation liées à l’exclusion mutuelle. De plus, ces algorithmes ne doivent pas seulement avoir de bonnes performances d’un point de vue du débit, mais aussi considérer la latence ainsi que l’efficacité énergétique.

Contents

Acknowledgements iii

Preface v

Abstract vii

Résumé ix

List of figures xii

List of tables xiii

1 Introduction 1
- 1.1 Multicore primer . . . 1
- 1.2 Mutual exclusion . . . 3
- 1.3 Thesis statement . . . 3
- 1.4 Contributions . . . 4
- 1.5 Outline . . . 5

2 Background 7
- 2.1 Locking . . . 7
  - 2.1.1 Synchronization primitives . . . 7
  - 2.1.2 Categorizing lock algorithms . . . 9
  - 2.1.3 Waiting policy . . . 25
- 2.2 Related work . . . 27
  - 2.2.1 Lock algorithm implementations . . . 27
  - 2.2.2 Adaptive algorithms . . . 28
  - 2.2.3 Studies of synchronization algorithms . . . 28
  - 2.2.4 Energy efficiency . . . 29
  - 2.2.5 Lock-related performance bottlenecks . . . 30

3 LiTL: A Library for Transparent Lock interposition 31
- 3.1 Design . . . 31
  - 3.1.1 General principles . . . 31
  - 3.1.2 Supporting condition variables . . . 33

3.1.3 Support for specific lock semantics . . . 34
3.2 Implementation . . . 36
3.3 Lookup overhead . . . 36
3.4 Experimental validation . . . 37
3.5 Statistical test . . . 39

4 Study **41**
4.1 Study's methodology . . . 41
4.1.1 Studied algorithms . . . 41
4.1.2 Testbed . . . 44
4.1.3 Studied applications . . . 45
4.1.4 Tuning and experimental methodology . . . 47
4.2 Study of lock throughput . . . 50
4.2.1 Preliminary observations . . . 50
4.2.2 Main questions . . . 53
4.2.3 Additional observations . . . 61
4.2.4 Effects of the lock choice on application performance . . . 63
4.3 Study of lock energy efficiency . . . 65
4.3.1 Energy-efficiency lock behavior . . . 65
4.3.2 POLY . . . 67
4.4 Study of lock tail latency . . . 73
4.4.1 How does tail latency behave when locks suffer from high levels of contention? . . . 73
4.4.2 Do fair lock algorithms improve the application tail latency? . . . 74
4.4.3 Do lock tail latencies affect application throughput? . . . 74
4.4.4 Implications . . . 75
4.5 Analysis of lock/application behavior . . . 78
4.5.1 Summary of the lock/application behavior analysis . . . 78
4.5.2 Guidelines for lock algorithms selection . . . 88

5 Conclusion **95**
5.1 Lessons learned . . . 96
5.2 Future research . . . 97

Bibliography **100**

List of Figures

1.1 A modern multicore NUMA machine. 2

2.1 Principle of the ttas lock algorithm. 10
2.2 Principle of the MCS lock algorithm. 12
2.3 Principle of a Cohort lock algorithm. 14
2.4 Principle of the RCL lock algorithm. 18
2.5 Principle of the Malth_Spin lock algorithm. 21

3.1 Pseudocode for the main wrapper functions of LiTL. 32
3.2 Performance comparison of manually implemented locks vs. LiTL. . . . 35

4.1 For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock *A* vs lock *B*: percentage of lock-sensitive applications for which lock *A* performs at least 5% better than *B*. 56
4.2 Correlation of throughput with energy efficiency (TPP) on various lock-sensitive applications with various lock algorithms and various contention levels. 68
4.3 Correlation of throughput with energy efficiency (TPP) on various lock-sensitive applications at *one node* for the different lock algorithms. . . . 69
4.4 Correlation of throughput with energy efficiency (TPP) on various lock-sensitive applications at *max nodes* for the different lock algorithms. . . 70
4.5 For each server application, the bars represent the normalized 99th tail latency (w.r.t. Pthread) and the dots execution time (lower is better) normalized (w.r.t. Pthread) of each lock algorithm. 77
4.6 Steps to follow for the application developer to chose a lock algorithm. 92

List of Tables

3.1 Detailed statistics for the performance comparison of manually implemented locks vs. LiTL. . . . 38
3.2 Percentage of lock pairs *(A, B)* where, if performance with manually implemented locks of A is worse, equals or better than B, it is also respectively worse, equals or better than B with transparently interposed locks using LiTL. . . . 38
3.3 Paired Student t-test to compare manually implemented locks vs. LiTL. 39

4.1 Lock algorithms summary. . . . 43
4.2 Hardware characteristics of the testbed platforms. . . . 44
4.3 Real-world applications considered. . . . 45
4.4 For each application, performance gain of the best vs. worst lock and relative standard deviation. . . . 52
4.5 Number of applications and number of lock performance sensitive applications. . . . 53
4.6 For each *(lock-sensitive application, lock)* pair, performance gain (in %) of *opt nodes* over *max nodes*. . . . 54
4.7 Breakdown of the *(lock-sensitive application, lock)* pairs according to their optimized number of nodes. . . . 55
4.8 Statistics on the coverage of locks on lock-sensitive applications. . . . 55
4.9 For each lock, fraction of the lock-sensitive applications for which the lock yields the best performance for three configurations: *one node*, *max nodes* and *opt nodes*. . . . 57
4.10 For each lock, at *max nodes* and at *opt nodes*, fraction of the lock-sensitive applications for which the lock is harmful. . . . 58
4.11 For each lock-sensitive application, at *max nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. . 60
4.12 For each lock-sensitive application, percentage of pairwise changes in the lock performance hierarchy when changing the number of nodes. . 62
4.13 Energy-efficiency gain of *opt nodes* over *max nodes*breakdown for lock-sensitive applications. . . . 66
4.14 Percentage of lock-sensitive applications for which *opt nodes* is lower, the same or higher for energy efficiency w.r.t. performance. . . . 66

4.15 Pearson correlation coefficient between throughput and TPP for all lock-sensitive applications. 71
4.16 Lock-sensitive application performance bottleneck(s) and lock algorithms choice advice. 79
4.17 Lock algorithm properties. 89

1 Introduction

For several decades, processor vendors improved performance by increasing CPU frequency. However, physical limitations such as current leakage causing CPU to heat up limited the increase of frequency over the years. As a consequence, to keep increasing performance, vendors switched to multicore architectures [104].

As of 2018, multicore machines with tens of cores (e.g., 64 cores) are now widespread. While previously reserved to server, multicore processors are now embedded into personal desktop, laptop and even mobiles. Moreover, processor vendors continue to increase the number of cores per socket, with socket up to 64 cores are rumored to become available before 2020 [15].

1.1 Multicore primer

Figure 1.1 illustrates a modern multicore machine. Modern multiprocessor machines are composed of several sockets, each one composed of several homogeneous cores. Each of these cores is composed of its own set of registers and components (e.g., memory unit, arithmetic logic unit – *ALU*), allowing several execution contexts running in parallel. Cores access memory through a hierarchical organization of private and shared caches. Generally, there are one (*L1*) or two (*L1* and *L2*) private caches per core; a bigger cache (the *last level cache* – *LLC*, often *L3*) is shared between the cores of a socket. Finally, data is stored in DRAM and menory accesses go through the memory controller responsible for accesses to the DRAM module where the memory is stored.

Historically, CPU frequency has increased faster than memory frequency, such that cores stall, waiting for their memory accesses to be fulfilled. A (partial) solution is to provide bigger and faster caches, yet this is not always enough, especially as the number of cores on a socket keeps increasing. To distribute the memory load, modern multicore machines are based on a NUMA (Non-Uniform Memory Accesses) architecture. A NUMA machine is composed of several NUMA nodes, each one composed of

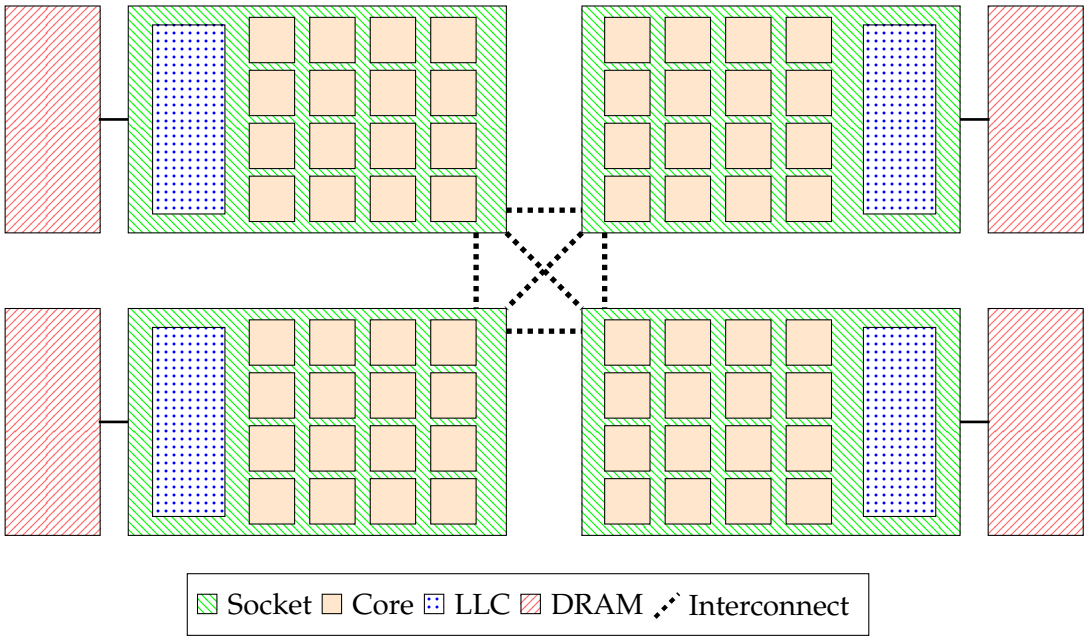


Figure 1.1 – A modern multicore NUMA machine.

one socket, one or more memory controllers, upon which one or more DRAM modules are attached. In order to allow any core to have access to the entire available memory, sockets are linked together through a high-speed interconnect (e.g., Intel QuickPath Interconnect [60] or AMD HyperTransport [28]). On such a machine, local memory accesses (i.e., a memory request is served by a cache/DRAM of the same NUMA node as the request core) and remote memory accesses (a memory request is served by a cache/DRAM of a remote NUMA node) do not have the same latency, hence the term *Non-Uniform Memory Access*. Besides, the bandwidth of remote memory accesses is more constrained than the one of local memory accesses.

Because cores rely on caches to improve memory latency, situations might arise where two cores have the same data in their respective private cache, one core modifies its own local copy and the second one does not see the modification: there is a cache coherence problem. To maintain a single, coherent view of the memory, a cache coherency protocol is used[1]. At the granularity of a cache line (generally 64 bytes on a modern machine), each core tracks the status of a given memory location and communicates to other cores when the location is accessed or modified. Maintaining this coherency is costly, because cache lines need to be exchanged between cores, and on NUMA machine this leads to costly exchanges through the interconnect.

[1]This thesis only considers ccNUMA (cache-coherent NUMA) SMP machines.

1.2 Mutual exclusion

To fully exploit the parallelism exposed by a multicore machine, software is now written from the beginning with concurrency and parallelism in mind, often leveraging frameworks and languages that help developers writing concurrent code. However, efficiently leveraging the available hardware parallelism is often challenging due to bottlenecks that impede multicore scalability. One of the main factor of poor scalability is the consequence of Amdahl's law.

Theoretically, the scalability of a program is dictated by Amdahl's law [7], which states that a program's scalability is always constrained by its sequential parts. These sequential parts are most often due to synchronization: in order to avoid inconsistent states, execution contexts running concurrently must synchronize their access to shared data. The most popular means of synchronization is mutual exclusion via locking: a critical section (i.e., the part of code that needs to be executed atomically) is protected by a lock, serializing accesses to this section. In this context, an efficient way to improve scalability is to reduce the sequential parts as much as possible. Because the time spent inside the lock algorithm protecting the critical section might unnecessarily lengthen the sequential part, studying lock algorithms has been of prime interest to the research community. Over the past 25 years, a plethora of optimized mutual exclusion (mutex) lock algorithms have been designed to mitigate these issues [91, 9, 80, 29, 76, 85, 88, 54, 75, 36, 55, 37, 43, 33, 68, 25, 38, 24, 42, 112, 34, 46, 64, 93]. Application and library developers can choose from this large set of algorithms for implementing efficient synchronization in their software. However, there is currently no detailed study to guide this puzzling choice for realistic applications.

1.3 Thesis statement

The most recent and comprehensive empirical performance evaluation on multicore synchronization [33], due to its breadth (from hardware protocols to high-level data structures), only provides a partial coverage of locking algorithms. Indeed, the aforementioned study only considers nine algorithms, does not consider hybrid waiting policies (i.e., what a thread does while waiting for a lock), omits emerging approaches (e.g., load-control mechanism) and provides a modest coverage of hierarchical locks [38, 25, 24], a recent and efficient approach for NUMA architectures. Generally, most of the observations highlighted in the existing literature are based on microbenchmarks and only consider the lock/unlock interface, ignoring other lock-related operations such as condition variables and trylocks. Besides, in the case of papers that present a new lock algorithm, the empirical observations are often focused on the specific workload characteristics for which the actual lock was designed [61, 72], or mostly based on microbenchmarks [38, 34]. Finally, existing analyses focus on traditional performance metrics (mainly throughput) and do not cover other metrics, such as energy efficiency

and tail latency, which are becoming increasingly important.

In this thesis, we perform a thorough and practical analysis of synchronization, with the goal of providing software developers with enough information to design fast, scalable and energy-efficient synchronization in their systems.

1.4 Contributions

The contributions of this thesis are threefold.

The first contribution is a broad performance study (Sections 4.2 to 4.4) on Linux/x86 (i.e., the Linux operating system running on AMD/Intel x86 64-bit processors) of 28 state-of-the-art mutual exclusion lock algorithms on a set of 40 realistic and diverse applications: PARSEC, Phoenix, SPLASH2 benchmark suites, MySQL, Kyoto Cabinet, Memcached, RocksDB, SQLite, upscaledb and an SSL proxy. Among these 40 applications, we determine that performance varies according to the choice of a lock for roughly 60% of them, and perform our in-depth study on this subset of applications. We believe this set of applications to be representative of real-world applications: we consider applications that not only stress the classic lock/unlock interface to different extents, but also exhibit different usage patterns of condition variables, trylocks, barriers and that use different number of locks (i.e., from one global lock to thousands of locks). We consider four different multicore machines and three different metrics: throughput, tail latency and energy efficiency. In our quest to understand the behavior of locking, when choosing the per-configuration best lock, we improve on average application throughput by 90%, energy efficiency by 110% and tail latency 12× with respect to the default POSIX mutex lock (note that, in many cases, different locks optimize different metrics). As we show in this thesis, choosing a well performing lock is difficult, as this choice depends on many different parameters: the workload, the underlying hardware, the degree of parallelism, the number of locks, how they are used, the lock-related interfaces that the application stresses (e.g., lock/unlock, trylock, condition variables), the interaction between the application and the scheduler, and the performance metric(s) considered.

Our second contribution is an in-depth analysis of the different types of lock-related performance bottlenecks that manifest in the studied applications. In particular, we describe nine different lock-related performance bottlenecks. Based on the insights of this analysis, we propose six guidelines for helping software developers with their choice of lock algorithms according to the different lock properties and the application characteristics. More precisely, by answering to a few questions about the considered application (e.g., *more threads than cores? blocking syscalls?*) and by looking at a few lock-related metrics (e.g., the number of allocated locks, the number of threads concurrently trying to acquire a lock), the developer is able to understand easily and quickly which

lock algorithm(s) to choose or to avoid for his specific use case.

Our third contribution is LiTL[2], an open-source, POSIX compliant [57], low-overhead library that allows transparent interposition of Pthread mutex lock operations and support for mainstream features like condition variables. Indeed, to conduct our study, manually modifying all the applications in order to retrofit the studied lock algorithms would have been a daunting task. Moreover, using a meta-library that allows plugging different lock algorithms under a common API (such as liblock [72] or libslock [33]) would not have solved the problem, as this still requires a substantial re-engineering effort for each application. In addition, such meta-libraries provide no or limited support for important features like Pthread condition variables, used within many applications. Our approach is a pragmatic one: similarly to what is done by previous works on memory allocators [16, 3, 67, 47], we argue that transparently switching (i.e., without modifying the application) lock algorithms (resp. memory allocators) is an efficient and pragmatic solution.

1.5 Outline

This thesis is organized as follows. Chapter 2 introduces our work, gives some background on locking and lock algorithms and discusses related works. Chapter 3 presents LiTL, the library we developed to evaluate the lock algorithms on the studied applications. Chapter 4 studies the throughput, energy efficiency and tail latency of lock algorithms, and performs a detailed analysis explaining, for each of the studied applications, which types of locks work well/poorly and why.

Finally, Chapter 5 concludes this thesis and discusses future research directions that we believe are worth investigating.

[2]LiTL: Library for Transparent Lock interposition.

2 Background

This Chapter provides the necessary background related to the other chapters of this thesis. We first give a primer of synchronization and locking, as well as describe existing lock algorithms in Section 2.1, then discuss the related work in Section 2.2.

2.1 Locking

All modern lock algorithms rely on hardware atomic instructions to ensure that a critical section is executed in mutual exclusion. To provide atomicity, the processor relies on the cache-coherence protocol of the machine to implement an atomic read-modify-write operation on a memory address. Previous work [33] demonstrated that lock algorithm performance is mainly a property of the hardware, i.e., a lock algorithm must take into account the characteristics of the underlying machine. The design of a lock algorithm is thus a careful choice of data structures, lock acquisition/release policies and (potential) load-control mechanisms.

Section 2.1.1 introduces the locking API. Section 2.1.2 proposes a classification of the lock algorithms into five categories. Section 2.1.3 discusses the various waiting policies.

2.1.1 Synchronization primitives

Locking is by far the most commonly-used approach to synchronization. Practically all modern software systems employ locks in their design and implementation. The main reason behind the popularity of locking is that it offers an intuitive abstraction. Locks ensure *mutual exclusion*; only the lock holder can proceed with its execution. Executions that are protected by locks are known as *critical sections*. Mutual exclusion is a way to synchronize concurrent accesses to the critical section, i.e., threads synchronize/coordinate to avoid one thread entering the critical section before the other left

it. In addition, *condition variables* allow threads to cooperate within a critical section by introducing a happened-before relationship between them, e.g., for a synchronized queue protected by a lock, inserting an element inside the queue happens before removing the element.

Mutual exclusion

Lock/unlock. Upon entering the critical section, a thread must acquire the lock via the `lock` operation. This operation is *blocking*, i.e., a thread trying to acquire a lock instance already held waits until the instance becomes available. When the lock holder exits the critical section, it must call the `unlock` operation, to explicitly release the lock. How to acquire a lock, what to do while waiting for the lock, and how to release the lock are choices made by a lock algorithm.

Trylock. If a lock is busy, a thread may do other work instead of blocking. In this case, it can use the non-blocking `trylock` operation. This operation has a return code to indicate if the lock is acquired. What a thread does when the `trylock` does not acquire the lock is up to the software developer, not the lock algorithm. We observe that developers frequently use `trylock` to implement busy-waiting, in order to avoid being descheduled (the policy used by the Pthread lock algorithm while waiting for a lock) if the lock is already held. This action is useful when the application developer knows that the critical section protected by the lock is short, and thus that there is a high chance for a thread to obtain the lock quickly. If the `trylock` acquires the lock, the lock holder must call `unlock` to release the lock.

Conditions variables

Threads often rely on condition variables to receive notifications when an event happens (e.g., when data is put inside a queue). A thread that wants to wait on a condition variable calls `wait` while holding a lock. As a consequence, the thread releases the lock and blocks[1]. When the condition is fulfilled, another thread calls `signal` or `broadcast` to wake any or all blocked threads, respectively. Upon wake-up (and before exiting from `wait`), a thread needs to acquire the lock again in order to re-enter the critical section. Efficiently implementing condition variables on top of locks is non-trivial (see Section 3.1.2).

[1] Releasing the lock and blocking is atomic, to avoid loosing a signal and being blocked indefinitely.

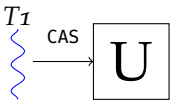
2.1.2 Categorizing lock algorithms

The body of existing work on optimized lock algorithms for multicore architectures is rich and diverse and can be split into the following five categories. The first two categories (competitive and direct handoff succession) are based on the succession policy [34] of the lock algorithm, i.e., how lock ownership is transferred at unlock-time. These two categories are mutually exclusive. The three other categories regroup algorithms that either compose algorithms from the first two categories (hierarchical approaches), change how critical sections are executed (delegation-based approaches), or improve existing locks with load-control mechanisms. Note that overall these categories overlap: a given algorithm can fall into several categories.

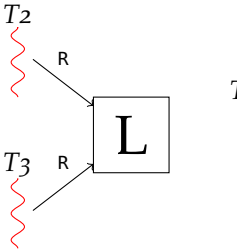
1) Competitive succession

Some algorithms rely on a competitive succession policy, where the lock holder sets the lock to an available state, and all competing threads might try to acquire it concurrently, all executing an atomic instruction on the same memory address. Such algorithms generally stress the cache-coherence protocol as they trigger cache line invalidations at unlock-time to all cores waiting for the lock, while ultimately only one core succeeds in acquiring it. Competitive succession algorithms might allow *barging*, i.e., "arriving threads can barge in front of other waiting threads" [34], which, for applications sensitive to the fairness between threads (e.g., latency sensitive applications), might lead to unfairness and starvation. Examples of algorithms using a competitive succession policy are simple spinlock [95], Backoff spinlock [9, 80], test and test-and-set (ttas) lock [9], Mutexee lock [42] and standard Pthread mutex locks[2] [68, 57]. Figure 2.1 illustrates the ttas lock algorithm.

[2]Throughout this manuscript, we refer to the GNU C Library implementation of the Pthread mutex lock algorithm.

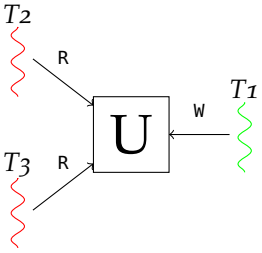


(a) To acquire the lock (the white rectangular box), which is a memory location having the state **U**nlocked (e.g., its content is 0), thread $T1$ executes an atomic instruction (`CAS`, *compare-and-set*) to change the lock state from **U**nlocked to **L**ocked (e.g., changing its content from 0 to 1). Note that $T1$ is in blue to illustrate the fact that it is *outside* the critical section (CS) before executing the `CAS` instruction.

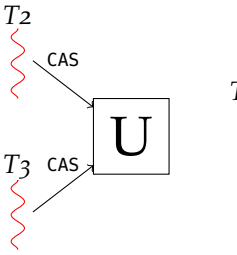


(b) $T1$ has the lock and is inside the CS (a thread inside the CS is in green). $T2$ and $T3$ both try to acquire the lock. However $T1$ already holds the lock, thus they wait for the lock to become **U**nlocked by looping on a memory load (**R**ead) instruction on the location of the lock. A red thread is a thread waiting for a lock.

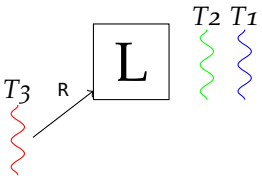
Figure 2.1 – Principle of the ttas lock algorithm.



(c) Upon the exit of the CS, $T1$ releases the lock with a memory store (**W**rite), writing **U**nlocked inside the memory location. Note that on some architectures (e.g., ARM [11]), a memory barrier might be necessary before the write to avoid memory instructions inside the CS to be reordered with instructions outside the CS.



(d) The lock is **U**nlocked, and $T1$ is outside the CS. Because both $T2$ and $T3$ were waiting for a change of the lock status, they both issue a `CAS` instruction concurrently, trying to acquire the lock. Due to the semantics of the `CAS` instruction, only one of the two threads acquires the lock and enters the CS.



(e) $T2$ acquired the lock and is inside the CS. $T3$ still waits for the lock, and will acquire it when $T2$ releases it, unless $T1$ tries to acquire the lock in the meantime. In such a case, for a ttas lock, because arriving threads can barge in front of other waiting threads, $T3$ will be in competition with $T1$ to acquire the lock after $T2$ releases it.

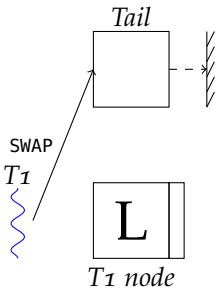
Figure 2.1 – Principle of the ttas lock algorithm (Cont.).

2) Direct handoff succession

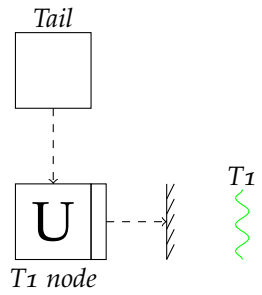
Direct handoff locks (also known as queue-based locks) are lock algorithms in which the unlock operation identifies a waiting successor and then passes ownership to that thread [34]. As the successor of the current lock holder is known, it allows each waiting thread to wait on a non-globally shared memory address (one per waiting thread). Then, the lock holder passes ownership with the help of this private memory address, thus avoiding cache line invalidations to all the other competing cores (contrary to the competitive succession policy). This approach is also known to yield better fairness. Besides, this approach generally gives better throughput under contention compared to simpler locks like spinlock: with direct handoff locks, each thread spins on its own local variable, avoiding to send cache line invalidations to all other spinning cores when the lock is acquired/released (contrary to locks based on a global variable). Examples of direct handoff lock algorithms are: MCS [80, 95], CLH [29, 76, 95].

Some algorithms do use a globally shared memory address but still use a direct handoff succession policy. For example, the Ticket lock [91] repeatedly reads a single memory address in a non-atomic fashion, waiting for its turn to come. The Partitioned Ticket lock [36] uses a hybrid solution, where the same memory address can be observed by a subset of the competing threads.

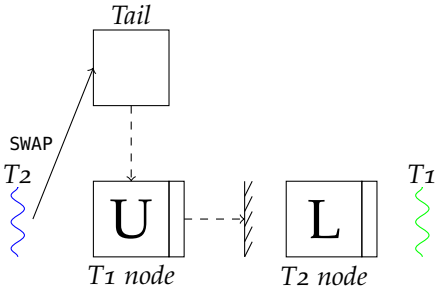
Figure 2.2 illustrates the MCS lock algorithm.



(a) The MCS lock algorithm organizes waiting threads in a single linked list, keeping track of the tail of the list with a pointer. At the beginning, the linked list is empty, i.e., *Tail* points to *nil*. $T1$ wants to acquire the lock, thus it tries to enqueue itself for the lock. $T1$ *node* (for the linked list) contains a memory location on which $T1$ spins while waiting for the lock and by default is **L**ocked, and a pointer to the next element (i.e., the next thread node) of the waiting list. To acquire the lock, $T1$ uses an atomic `SWAP` instruction, which atomically sets *Tails* to point to $T1$ *node* and retrieves the previous value of *Tail* (here *nil*).

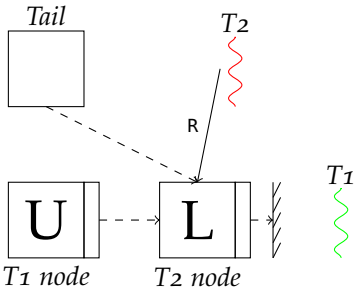


(b) After the *SWAP*, *Tail* points to $T1$ *node*. Because *Tail* was previously *nil*, there is no other thread waiting for the lock nor inside the CS, thus $T1$ enters the CS without waiting ($T1$ memory location is **U**nlocked as $T1$ does not wait).

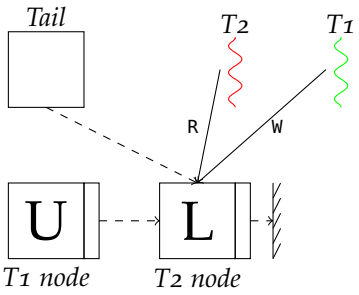


(c) Another thread $T2$ also wants to acquire the lock, thus it atomically uses `SWAP` to update the *Tail* pointer so that it points to $T2$ *node*.

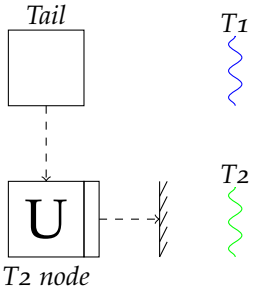
Figure 2.2 – Principle of the MCS lock algorithm.



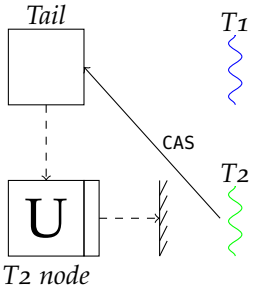
(d) After the SWAP, because *Tail* was not previously *nil* (it pointed to *T*1 *node*), *T*2 sets its node to be the successor of the previous *Tail*, i.e., *T*2 *node* is the successor of *T*1 *node*. It also waits (*T*2 is in red) for its memory location (that is **L**ocked) to become **U**nlocked before entering the CS. Note that contrary to a ttas lock (see fig. 2.1), each thread waiting for a MCS lock waits on its private memory location, avoiding cache line invalidations sent to all cores when the lock state changes (with MCS, cache lines are exchanged only between the cores of the unblocked thread and the lock holder).



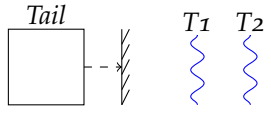
(e) Upon CS exit, *T*1 sees that it has a successor, thus *T*1 releases the lock by changing the memory location of *T*2 *node* to **U**nlocked.



(f) *T*2 memory location is **U**nlocked, thus *T*2 enters the CS.



(g) Upon CS exit, *T*2 sees that it has no successor, thus it tries to release the lock by setting the *Tail* pointer back to *nil* using a CAS atomic instruction The CAS is necessary to ensure that both checking that *Tail* points to *T*2 *node* and changing *Tail* execute atomically. Otherwise, another thread might try to acquire the lock while *T*2 releases it (i.e., the other thread switches *Tail* to point to its node and sets itself as the successor of *T*2).



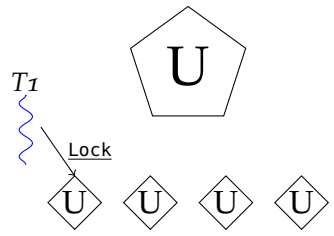
(h) *T*2 successfully sets *Tail* to *nil*, both *T*1 and *T*2 are outside the CS.

Figure 2.2 – Principle of the MCS lock algorithm (Cont.).

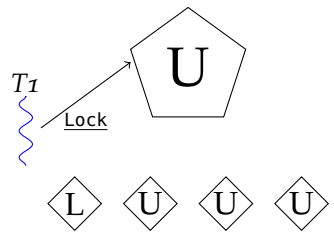
3) Hierarchical approaches

These approaches aim at providing scalable performance on NUMA machines, by attempting to reduce the rate of lock migrations (i.e., cache line transfers between last level caches), which are known to be costly between NUMA nodes. A hierarchical lock tends to give the lock to a thread running on the same NUMA node as the thread holding the lock, improving throughput, at the expense of fairness between threads. This category includes HBO [88], HCLH [75], FC-MCS [37], HMCS [25] and the algorithms that stem from the *lock cohorting* framework [38]. A cohort lock is based on a combination of two lock algorithms (similar or different): one used for the global lock and one used for the local locks (there is one local lock per NUMA node); in the usual C-L_A-L_B notation, L_A and L_B respectively correspond to the global and the node-level lock algorithms. The list notably includes C-BO-MCS, C-PTL-TKT and C-TKT-TKT (also known as HTicket [33]). The *BO*, *PTL* and *TKT* acronyms respectively correspond to Backoff lock, Partitioned Ticket lock, and standard Ticket lock.

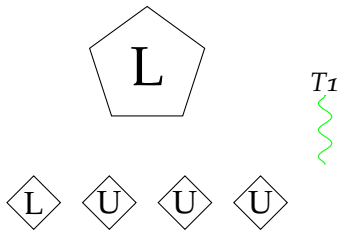
Figure 2.3 illustrates a cohort lock algorithm.



(a) A cohort lock algorithm is a hierarchical lock composed of two different lock algorithms: a global one (represented by a pentagon) and a local one (represented by a rotated square). There are one local lock per NUMA node (here 4) and one global lock. Each lock composing a cohort lock starts in a **U**lnocked state. To acquire the cohort lock, $T1$ first locks the local lock attached to the NUMA node on which $T1$ runs (here the left-most one).

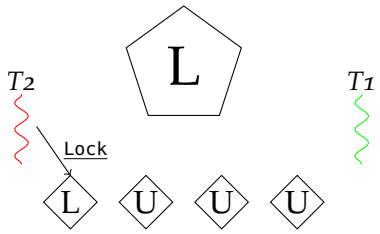


(b) $T1$ is the lock holder of its local lock and it tries to acquire the global lock.

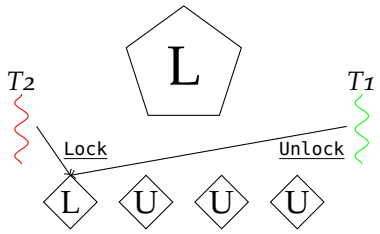


(c) $T1$ has both a local lock and the global lock, it enters the CS.

Figure 2.3 – Principle of a Cohort lock algorithm.

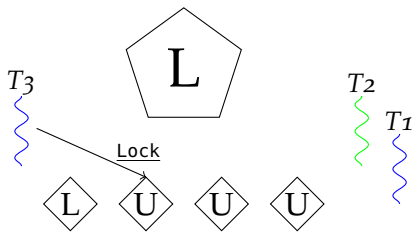


(d) $T2$ also wants to acquire the cohort lock. Because it runs on the same NUMA node as $T1$, $T2$ tries to acquire the same local lock as $T1$. The lock is already taken by $T1$, thus $T2$ waits for the lock to become available.

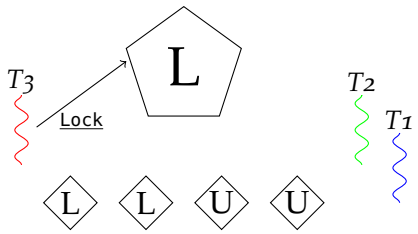


(e) Upon exit of the CS, $T1$ sees that there is a waiting thread on its local lock, thus it unlocks the local lock, which will be acquired by $T2$. Because $T1$ and $T2$ have the same local lock, $T1$ does not unlock the global lock, but "gives" it to $T2$ (there is no special action here, the global lock "belongs" to a NUMA node, not to a specific thread).

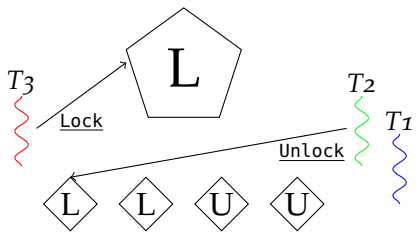
Note that not all lock algorithms can be used for the global/local locks: (i) for the local lock algorithm, it must be possible to know if there is another waiting thread (also known as *cohort detection*), and (ii) the global lock algorithm must support the scenario where the thread locking the lock is not the same as the thread unlocking it (also known as *thread-oblivousness*).



(f) A new thread $T3$ wants to enter the CS. $T3$ does not run on the same node as $T1$ and $T2$, thus it tries to acquire its local lock (the second one on the left) and succeeds.

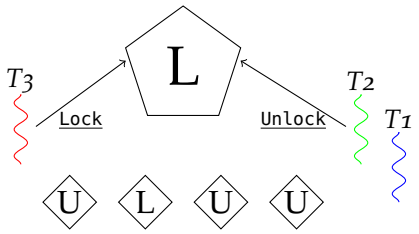


(g) $T3$ sees that its NUMA node is not the holder of the global lock, thus it waits for the global lock to be released. Here we see that in the worst case, there are at most $N-1$ waiting threads on the global lock (where N is the number of NUMA nodes).

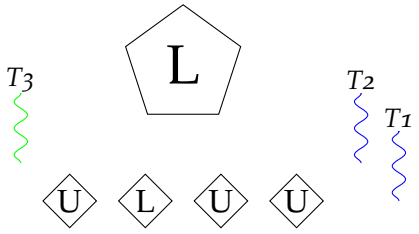


(h) Upon CS exit, $T2$ releases its local lock.

Figure 2.3 – Principle of a Cohort lock algorithm (Cont.).



(i) Because there is no other waiting thread on the local lock of $T2$, it releases the global lock, which belongs to $T2$'s NUMA node.



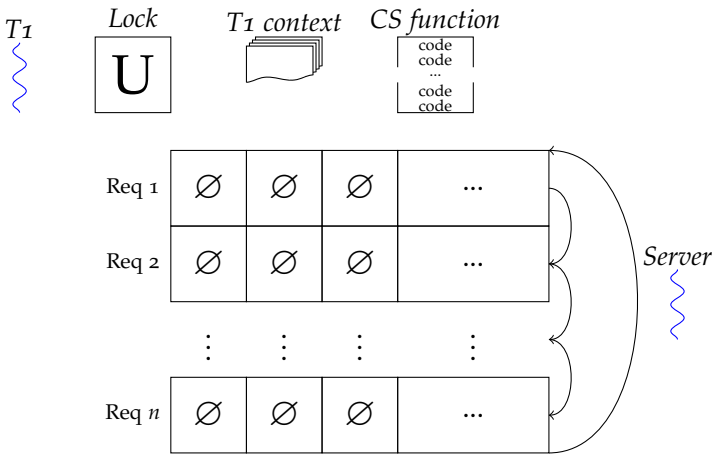
(j) $T3$ acquires the global lock. Because it has both a local lock and the global one, it enters the CS. If there is no other thread that tries to acquire the cohort lock, upon CS exit, $T3$ will first unlock its local lock, then the global one (similarly to the previous step).

Figure 2.3 – Principle of a Cohort lock algorithm (Cont.).

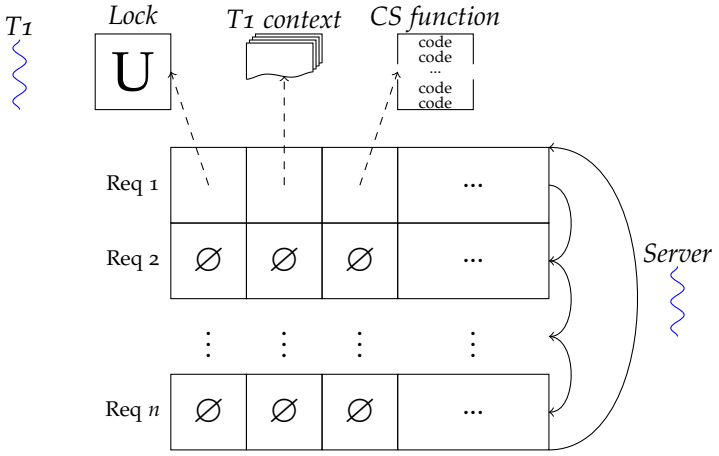
4) Delegation-based approaches

Delegation-based lock algorithms are locks in which it is (sometimes or always) necessary for a thread to delegate the execution of a critical section to another thread. The typical benefits expected from such approaches are improved cache locality for the operations within the critical section and better resilience under very high lock contention. This category includes Oyama [85], Flat Combining [55], RCL [72], FFWD [93], CC-Synch [43] and DSM-Synch [43]. Contrary to other algorithms, delegation-based algorithms require critical sections to be expressed as a form of closure (e.g., a function), which is not compatible with the lock/unlock API presented earlier (§2.1.1).

Figure 2.4 illustrates the RCL lock algorithm.

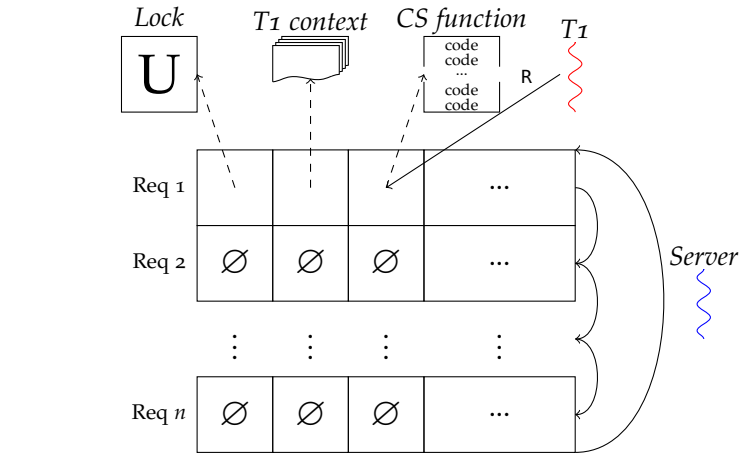


(a) A RCL lock delegates the execution of the CS to a remote thread (the *Server*). The *Server* monitors an array of requests (one slot per thread), where each request is composed of a pointer to the *lock* protecting the CS, a pointer to a structure called the *context*, which contains the environment (e.g., variables) with which the CS must be executed, and a pointer to the *CS function*. Each request entry is padded to fit a cache line to avoid false sharing. Note that one of the limitations of RCL is that the CS must be expressed as a function taking a list of arguments (the *context*), which is not the initial semantics of a POSIX mutex lock. As a consequence, RCL is not transparent w.r.t. the application. The *Server* continuously scans the array for the slot whose *CS function* field is not *nil*, which indicates that a thread wants a CS to be executed.
On this figure, *T*1 wants to execute a CS protected by a lock (by default **U**nlocked).

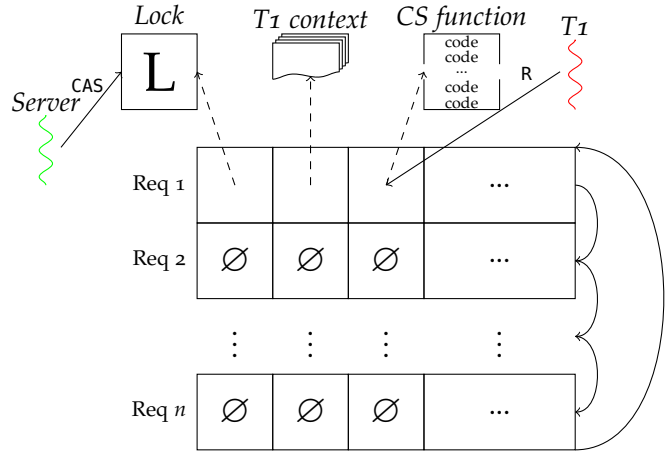


(b) After having set up its context, *T*1 writes the *lock*, *context* and *CS function* addresses to their respective pointer fields.

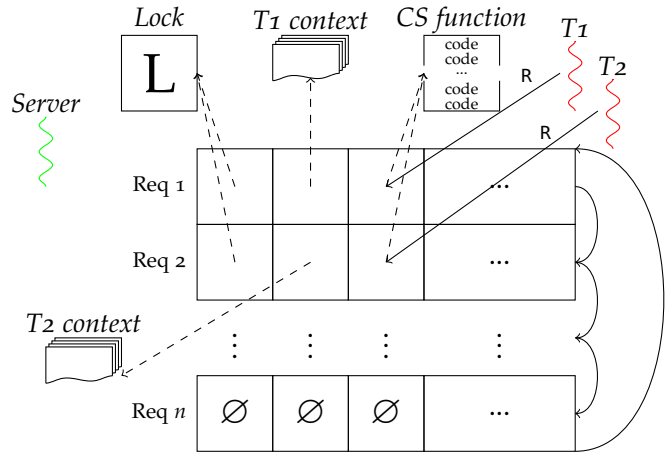
Figure 2.4 – Principle of the RCL lock algorithm.



(c) $T1$ waits for its request to be fulfiled (i.e., waiting for the CS to be executed on its behalf) by waiting for the *CS function* pointer to become *nil*.
Note that the *Server* serializes all CS executions, even if they acquire different locks. However, when two requests use the same lock, only the *Server* acquires the lock (avoiding cache line transfers), the requesting threads wait on their private *CS function* field.

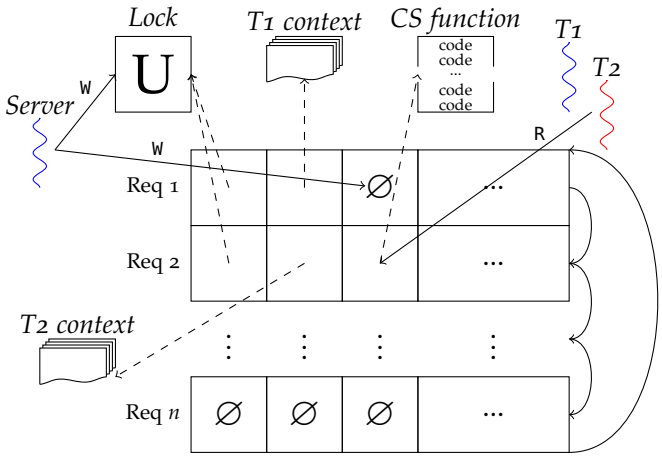


(d) The *Server* sees a new request (*CS function* of $T1$ is not *nil*) and first tries to acquire the lock pointed by the *lock* pointer field. RCL still tries to take the lock atomically in the case of a concurrent thread that did not delegate its CS execution to the *Server*. The *Server* succeeds and enters the CS.

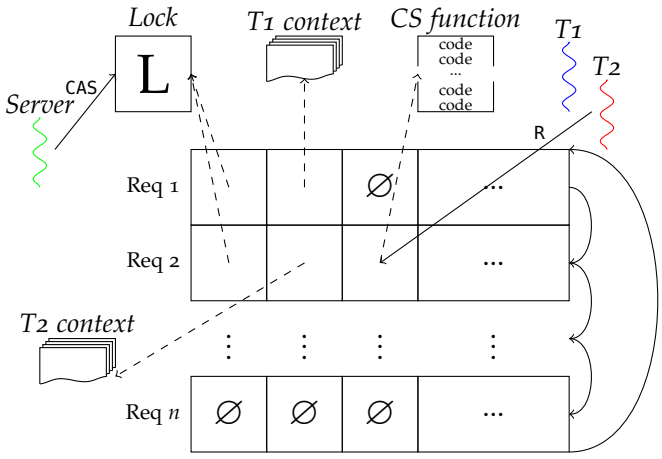


(e) While the *Server* is executing the CS of $T1$, another thread $T2$ requests the execution of the same CS protected by the same lock. It sets up its context, its request fields, and waits on the *CS function* pointer for its request to be fulfilled.

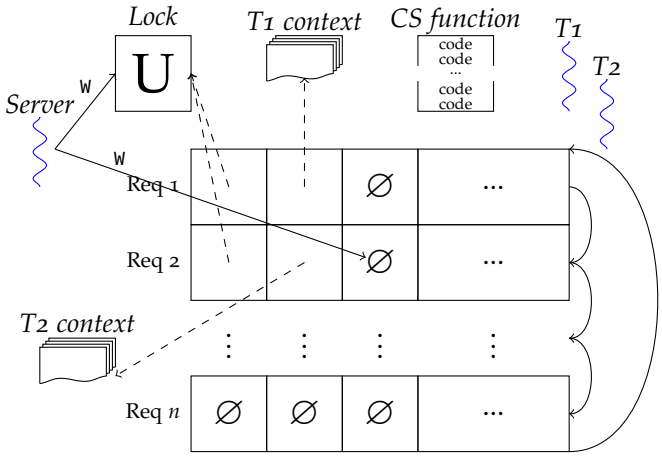
Figure 2.4 – Principle of the RCL lock algorithm (Cont.).



(f) The *Server* finished handling the request of *T*1: it writes *nil* inside *T*1 *CS function* pointer, which unblock *T*1. The *Server* also unlocks the lock.



(g) The *Server* proceeds to the next request and sees *T*2's one. As a consequence, it atomically acquires the lock, and enters the CS.



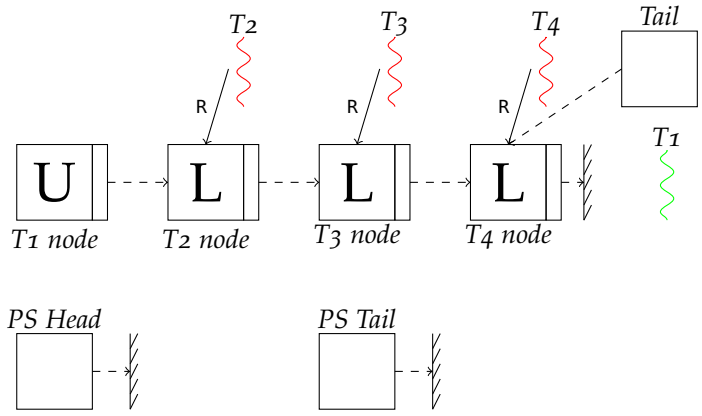
(h) Finally, the *Server* finishes *T*2's request, and writes *nil* to *T*2's *CS function* pointer to unblock *T*2.

Figure 2.4 – Principle of the RCL lock algorithm (Cont.).

5) Load-control mechanisms

This category includes lock algorithms implementing mechanisms that detect situations in which a lock needs to adapt itself. For example, GLS [10] and SANL [112] switches between different lock algorithms to cope with changing levels of contention (i.e., how many threads concurrently attempt to acquire a lock). Other algorithms such as AHMCS[3] [24] and so-called *Malthusian algorithms* like Malth_Spin and Malth_STP[4] [34] adapt the locking scheme (i.e., how a lock is acquired/released) depending on the contention level. Finally, some algorithms aim to avoid lock-related pathological behaviors (e.g., preemption of the lock holder to execute a thread waiting for the lock): MCS-TimePub[5] [54] and LC [61] are two examples of such locks.

Figure 2.5 illustrates the Malth_Spin lock algorithm.



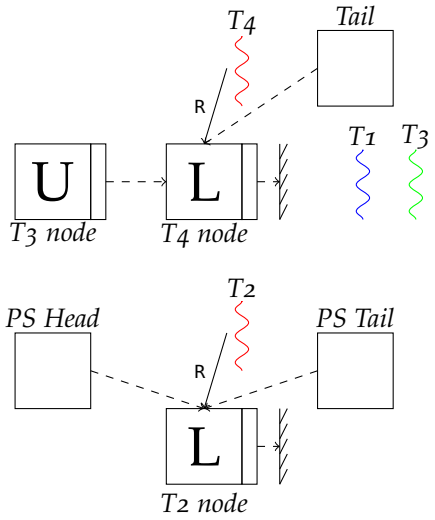
(a) A Malth_Spin lock algorithm is a modification of the MCS lock algorithm. The idea of Malth_Spin is to move the surplus of waiting threads into a passive set (*PS* hereafter), where threads in this set do not compete to acquire the lock. Malth_spin eventually converges towards the situation where there is at most one thread in the CS, one thread waiting for the lock, and all the other waiting threads are in the *PS* or outside the CS. The *PS* is maintained as a double linked list, where the head and the tail of this list are tracked with *PS head* and *PS tail* pointers. All modifications to the MCS algorithm are made inside the `unlock` operation, just before releasing the lock.
In this figure, we start with a MCS lock with three waiting threads ($T2$, $T3$, $T4$) and one thread inside the CS ($T1$). There is no thread in the *PS* yet, i.e., both *PS head* and *PS tail* point to *nil*.

Figure 2.5 – Principle of the Malth_Spin lock algorithm.

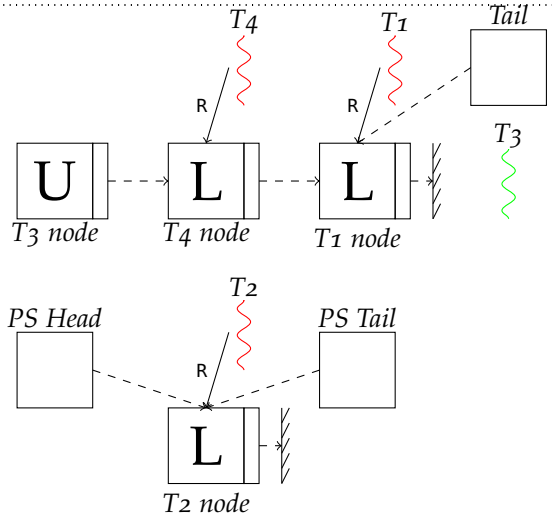
[3]The original AHMCS paper [24] presents multiple versions of AHMCS. In this article, the version *without* hardware transactional memory of AHMCS is considered.

[4]Malth_Spin and Malth_STP correspond to MCSCR-S and MCSCR-STP respectively in the terminology of Dave Dice [34]; still we do not use the latter names to avoid confusion with other MCS locks.

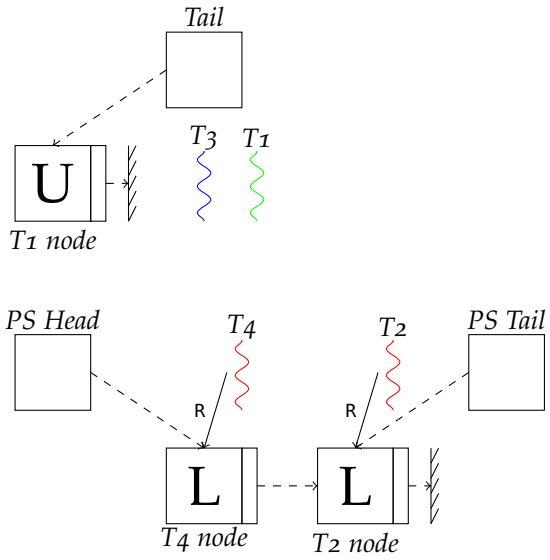
[5]MCS-TimePub is mostly known as MCS-TP. Still, we use MC-TimePub to avoid confusion with MCS_STP.



(b) Upon CS exit, $T1$ sees that there is more than one waiting thread, i.e., $T1$ node has more than one successor. As a consequence, $T1$ puts its successor $T2$ in the *PS*, which will wait spinning on its node lock (in the Malth_STP variant, threads are descheduled while waiting on the *PS*). Then $T1$ gives the lock to $T3$, which enters the CS. Note that *PS head* and *PS tail* pointers are updated while $T1$ still holds the lock, thus there is no need of atomic instructions/specific care to safely update them.

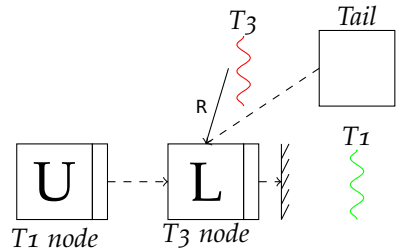


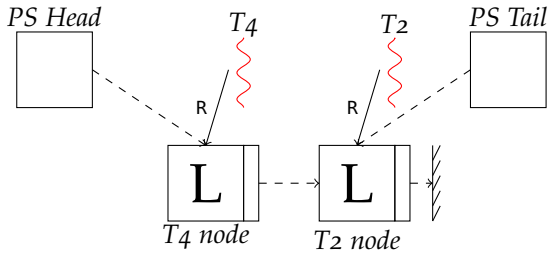
(c) While $T3$ is still inside the CS, $T1$ tries to acquire the lock again, thus it enqueues itself after $T4$ (this is part of the MCS algorithm, not Malth_Spin).



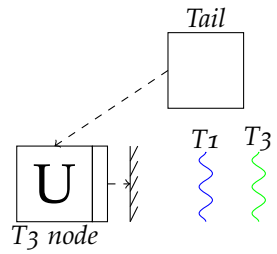
(d) Upon CS exit, $T3$ sees that there is more than one thread waiting on the lock, thus it moves $T4$ to the *PS* and gives the lock to $T1$ by switching the state of the memory location of $T1$'s node to **U**nlocked. $T1$ enters the CS.

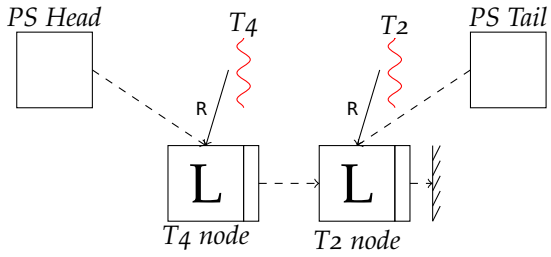
Figure 2.5 – Principle of the Malth_Spin lock algorithm (Cont.).



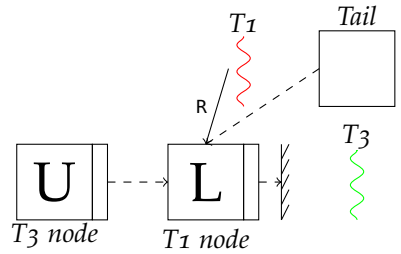


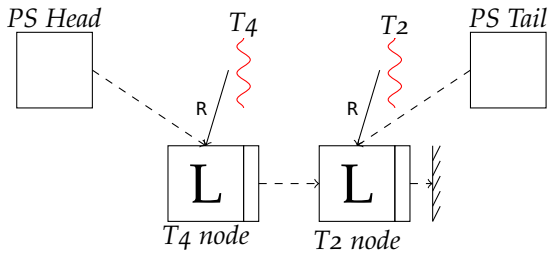
(e) While $T1$ is still inside the CS, $T3$ tries to acquire the lock again.





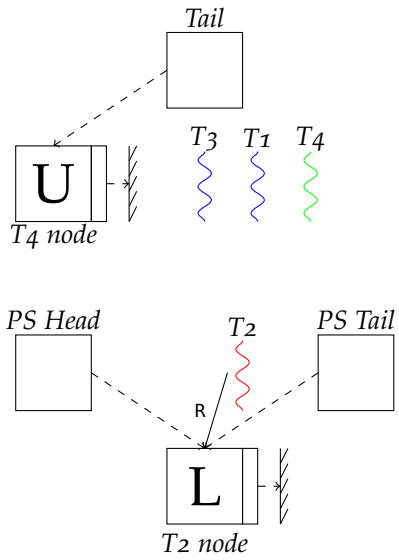
(f) Upon CS exit, $T1$ sees that there is only one thread waiting for the lock, thus it switches the memory location of $T3$'s node to **U**nlocked. $T3$ enters the CS.



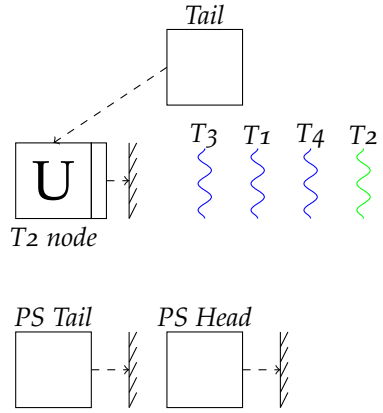


(g) While $T3$ is still inside the CS, $T1$ tries to acquire the lock again. Note that the algorithm has reached its stable state: the four previous steps can be repeated ad infinitum.

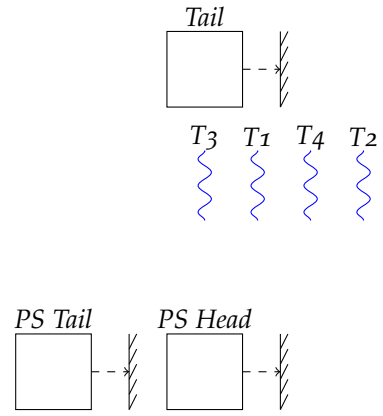
Figure 2.5 – Principle of the Malth_Spin lock algorithm (Cont.).



(h) This time, upon CS exit, $T1$ sees that there is no other waiting thread. Because Malth_Spin is work conserving, it will never leave the lock **U**nlocked while there is a waiting thread (even if it is in the *PS*). Thus, $T1$ removes $T4$ from the *PS* (by updating *PS head*) and switches the memory location of $T4$'s node to **U**nlocked. $T4$ enters the CS.



(i) Again, $T4$ sees that there is no other waiting thread, thus removes $T2$ from the *PS* and switches $T2$'s node memory location to **U**nlocked.



(j) All threads are outside the CS, and there is no more thread in the *PS*.

Figure 2.5 – Principle of the Malth_Spin lock algorithm (Cont.).

2.1.3 Waiting policy

An important design dimension of lock algorithms is the *waiting policy* used when a thread cannot immediately obtain a requested lock [34]. There are three main approaches.

Spinning

The most straightforward solution for waiting is to continuously check the status of the lock until it becomes available. However, such a policy might waste energy, and the time spent waiting on a core might prevent other descheduled threads from progressing. Processors provide special instructions to inform the CPU microarchitecture when a thread is spinning. For example, x86 CPUs offer the `PAUSE` instruction[6] that is specifically designed to avoid branch-misprediction, and which informs the core that it can release shared pipeline resources to sibling hyperthreads [34].

In case of a failed lock acquisition attempt, different lock algorithms can use different (and possibly combine several) techniques to lower the number of simultaneous acquisitions attempts and the energy consumption while waiting. Using a fixed or randomized backoff (i.e., a thread avoids attempting to acquire the lock for some time) lowers the number of concurrent atomic instructions, thus the cache-coherence traffic. Hardware facilities can also be used to lower the frequency of the waiting thread's core (DVFS [106]), or to notify the core that it can enter in an idle state to save power (via the `MONITOR/MWAIT` instructions [42][7]). Finally, a thread can voluntarily surrender its core in a polite fashion by calling `sched_yield` or `sleep`.

Immediate parking

With immediate parking[8], a thread waiting for an already held lock immediately blocks until the thread gets a chance to obtain the lock[9]. This waiting policy requires kernel support (via the `futex` syscall on Linux) to inform the scheduler that the thread is waiting for a lock, so that it does not try to schedule the thread until the lock is made available. At unlock-time, the lock holder is then responsible to inform the scheduler that the lock is available.

[6] The `MFENCE` instruction can also be used and is known to yield lower energy consumption than the `PAUSE` instruction on certain Intel processors [42].

[7] On the x86 platform, `MONITOR/MWAIT` are privileged instructions, which are accessible for locks running in privilegied mode, or via a kernel module [8].

[8] In the remainder of this manuscript, we use *blocking* and *(immediate) parking* interchangeably.

[9] Some locks use timeouts to bound the time a thread spends in the blocked state in order to improve responsiveness.

Hybrid approaches

The motivation behind hybrid approaches is that different waiting policies have different costs. For example, the *spin-then-park* policy is a hybrid approach using a fixed or adaptive spinning threshold [63]. It tries to mitigate the cost of parking as the block and unblock operations are expensive (both in terms of energy and performance). The spinning threshold is generally equal to the time of a round-trip context switch. Other techniques mix different spinning policies, such as backoff and `sched_yield` [33]. Finally, more complex combinations can be implemented: some algorithms [103, 42] trade fairness for throughput by avoiding to unpark a thread at unlock-time if there is another one currently spinning (also known as *adaptive unlock*).

The choice of the waiting policy is mostly orthogonal to the lock design but, in practice, policies other than pure spinning are only considered for certain types of locks: the direct handoff locks (from categories 2, 3 and 5 above), Mutexee and the standard Pthread mutex locks. However, this choice directly affects both the energy efficiency and the performance of a lock: Falsafi et al. [42] found that pure spinning inherently hurts power consumption, and that there is no practical way to reduce the power consumption of pure spinning. They found that blocking can indeed save power, because when a thread blocks, the kernel can then put the core(s) in one of the low-power idle states [6, 59]. However, the process of blocking is costly, because the cost of the blocking and unblocking operations is high on Linux. Switching continuously between blocking and unblocking can hurt energy efficiency, sometimes even more than using pure spinning policies. Thus, there is an energy-efficiency tradeoff between spinning and parking. Note that we use hereafter the expression *parking policy* to encompass both *immediate parking* and hybrid *spin-then-park* waiting policies.

In this Section, we provided background on locking, proposed a categorization of the existing lock algorithms and discussed the importance of the waiting policy of a lock algorithm. Designing correct and efficient lock algorithms is hard: the number of design choices (e.g. NUMA-aware, succession policy, waiting policy) is large and these choices imply trade-offs. The next Section discusses work around lock algorithm implementations and their effect on performance.

2.2 Related work

This thesis aims to understand the performance of existing lock algorithms on modern multicore machines. In this Section, we discuss the large body of work studying the different aspects of lock algorithms. More precisely, Section 2.2.1 presents work studying the implementation of lock algorithms, and previous approaches to transparently replace lock algorithms inside applications. Section 2.2.2 discusses the possibility to dynamically adapt lock synchronization at run-time. Section 2.2.3 considers previous studies of multicore lock algorithms. Section 2.2.4 covers existing works that highlight the importance of energy efficiency for both applications and lock algorithms. Finally, Section 2.2.5 discusses lock-related performance bottlenecks.

2.2.1 Lock algorithm implementations

The design and implementation of the LiTL lock library (presented in Chapter 3) borrows code and ideas from previous open-source toolkits that provide application developers with a set of optimized implementations for some of the most-established lock algorithms: Concurrency Kit [4], liblock [73, 71, 72], libslock [33] and lockin [42, 10]. All of these toolkits require potentially tedious source code modifications in the target applications, even in the case of algorithms that have been specifically designed to lower this burden [12, 95, 107]. Moreover, among the above works, virtually none of them provides a simple and generic solution for supporting Pthread condition variables. One noticeable exception is lockin [42, 10], which only requires including a header inside the source code of the application and recompile it linked against a specific shared library. lockin also proposes a condition variable algorithm; still the proposed algorithm does not circumvent the "thundering-herd" effect for all lock algorithms (see Section 3.1.2). The authors of liblock [72] proposed an approach to support condition variables; still we discovered that it suffers from liveness hazards due to a race condition (more details in Section 3.1.2). Indeed, when a thread T calls `pthread_cond_wait`, it is not guaranteed that the two steps (releasing the lock and blocking the thread) are always executed atomically. Thus, a wake-up notification issued by another thread might get interleaved between the two steps and T might remain indefinitely blocked.

Several research works have leveraged library interposition to compare different locking algorithms on legacy applications (e.g., Johnson et al. [61] and Dice et al. [38]). However, to the best of our knowledge, they have not publicly documented the design challenges to support arbitrary application patterns (e.g., condition variables), nor disclosed the corresponding source code and the overhead of their interposition library has not been discussed.

2.2.2 Adaptive algorithms

Previous works discuss the possibility to dynamically adapt lock synchronization at run-time. One way is to dynamically switch between lock algorithms depending on the contention level. The work by Lim et al. [70] considers switching among three lock algorithms (TTAS, MCS and a delegation-based one), depending on the level of contention on the lock instance. SANL [112] switches between local and remote (i.e., delegation-based) locking schemes. As explained in Section 2.1.2, delegation-based algorithms require critical sections to be expressed as a form of closure, which is incompatible with our transparent approach (i.e., without source code modification). More recently, Antic et al. [10] proposed GLS, a solution that dynamically switches among three lock algorithms (Ticket, MCS, Pthread mutex), using Ticket at low contention levels, MCS at high contention levels, and Pthread when it detects overthreading (i.e., more threads than cores). While these approaches confirm our observations that there is no one-size-fit-all locking algorithm (more details in Section 4.5), their goal is to make locking easy for a developer, not to choose the best lock algorithm in all cases. Indeed, they only switch among a few different lock algorithms, whereas, as we will show with our study, there are more lock algorithms to consider, making the choice more complex. None of the solutions considers some of the bottlenecks that we observed, like trylock contention, the lock handover effect and bottlenecks related to the memory footprint of a lock instance (§4.5). For example, all solutions embed all the different lock data structures into a unique one, inflating the memory layout of a lock instance: some applications (e.g., dedup) using thousands of lock instances that are good with a classical low memory footprint Ticket algorithm might not be good with the Ticket version of GLS, even if GLS never uses lock algorithms other than Ticket.

A second solution is to monitor the load pattern of the application to detect situations that are subject to pathological behavior. Load control (LC) [61] is a runtime solution, which dynamically reduces the number of threads trying to acquire the lock at the same time, to avoid pathological issues (e.g., lock convoy). LC requires kernel modifications on Linux to measure load accurately and with high resolution ($\sim 100\mu s$). This approach is thus incompatible with our work, where we focus on lock algorithms that do not require code modifications. As we will show later, our work highlights the need for low-memory, complete interface (i.e., lock, trylock, and condition variables), fully adaptive (i.e., from spinlocks all the way to complex HMCS locks) lock algorithms. Yet, none of the existing solutions answer this need.

2.2.3 Studies of synchronization algorithms

Several studies have compared the performance of different multicore lock algorithms, from a theoretical angle and/or based on experimental results [9, 80, 65, 19, 95, 33, 71, 38]. Our study (see Chapter 4) encompasses significantly more lock algorithms

and waiting policies. Moreover, the bulk of these prior studies is mainly focused on characterization microbenchmarks, while we focus instead on workloads designed to mimic real applications. Two noticeable exceptions are the work from Boyd-Wickizer et al. [19] and Lozi et al. [72]; still they do not consider the same context as our study. The former is focused on kernel-level locking bottlenecks, and the latter is focused on applications in which only one or a few heavily contended critical sections have been rewritten/optimized (after a profiling phase). For all these reasons, we make observations that are significantly different from the ones based on all the above-mentioned studies.

Some related work discusses the choice of synchronization paradigms and lock algorithms [78, 79, 77]. The proposed guidelines are often a subset of our proposed guidelines in Section 4.5.2: because these works only study a smaller set of applications and lock algorithms, they generally do not cover all the cases we observed.

Other synchronization-related studies have a different scope and focus on concurrent data structures, possibly based on other facilities than locks. Gramoli [48] studies different concurrent data structures on micro-benchmarks with multiple synchronization techniques. David el al. [31, 32] evaluate theoretical and practical progress properties of concurrent search data structures. Brown et al. [20] study the performance of hardware transactional memory with microbenchmarks on modern NUMA multicore machines. Finally, Calciu et al. [21] study the tradeoff between message passing and shared memory synchronization on multicore machines. Similarly to us, they advocate that software should be designed to be largely independent of the choice of low-level communication mechanism.

2.2.4 Energy efficiency

Improving energy efficiency in systems and applications has been thoroughly studied in the past. For example, previous works describe user-level [109, 99, 98, 82, 92, 110] and kernel [86] facilities that both manage and predict power consumption. Prior works propose trading performance and/or precision for energy. For example, programming models [13, 94] allow developers to approximate loops to decrease power consumption. Compiler techniques [109, 108] and hardware mechanisms [66] trade off performance for energy. To the best of our knowledge, the work by Falsafi et al. [42] is the only one studying the energy efficiency of lock algorithms. In this thesis, we confirm their findings and validate their POLY[10] conjecture on significantly more lock algorithms and applications (see Section 4.3).

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

2.2.5 Lock-related performance bottlenecks

Some tools have been proposed to facilitate the identification of locking bottlenecks in applications [105, 87, 30, 72, 10]. These tools are useful to identify which lock instances suffer from contention; still they do not help a software developer to choose a lock algorithm for an application. The proposed tools are orthogonal to our work. We note that, among them, the profilers based on library interposition could be stacked on top of LiTL.

Finally, lock-related performance bottlenecks have been previously analyzed. For example, many studies [2, 62, 33, 34] point out scalability problems due to excessive cache-coherence traffic with traditional spinlocks. Scheduling issues like the lock holder preemption problem have been well studied [65, 34] and some solutions try to mitigate it [65, 54]. Nonetheless, we discovered lock-related issues that, to the best of our knowledge, have not been described before (§4.5). Moreover, we are the first to analyze the impact of lock algorithms on such a large panel of applications, and to discuss in depth and summarize the many different bottlenecks they exhibit.

SyncPerf [5] is a recent profiler detecting previously undiscussed lock-related performance bottlenecks. Similarly to us, the authors of SyncPerf discover that trylocks contention and uncontended lock acquisitions are two bottlenecks affecting application performance. While this tool is a must-have in the system performance analysis tool belt, it only considers the Pthread mutex lock, and thus fails at detecting some lock-related performance bottlenecks. Indeed, as we will show in this manuscript, many applications benefit from using other locks than Pthread, and these other locks suffer from bottlenecks unseen with Pthread (e.g., scheduling issues, memory consumption).

3 LiTL: A Library for Transparent Lock interposition

In this Chapter, we present the LiTL library, an open-source, POSIX compliant, low-overhead library that allows transparent interposition of Pthread mutex lock operations and support for mainstream features like condition variables. The primary motivation for using LiTL is to be able to evaluate multiple lock algorithms on a large number of applications without modifying the source code or recompiling the applications. However, the advantages of a using a shared library to "wrap" the lock algorithm implementation are broader, and introduce other opportunities such as building adaptive lock algorithms that can be switched at run-time, using multiple lock algorithm for different lock instances inside the same application, or providing condition variables to custom locks that are workload specifics. We first describe the design of LiTL in Section 3.1, discuss its implementation in Section 3.2, evaluate some elementary costs introduced by LiTL in Section 3.3, and experimentally assess its performance in Section 3.4.

3.1 Design

We describe the general design principles of LiTL, how it supports condition variables, and how it can easily be extended to support specific lock semantics. The pseudo-code of the main wrapper functions of the LiTL library is depicted in Figure 3.1.

3.1.1 General principles

The primary role of LiTL is to maintain a mapping between an instance of the standard Pthread lock (`pthread_mutex_t`) and an instance of the chosen optimized lock type (e.g., MCS_Spin). This mapping is maintained in an external data structure (see details in §3.2), rather than using an "in-place" modification of the `pthread_mutex_t` structure. This choice is motivated by two main reasons. First, for applications that rely on condition variables, we need to maintain a standard `pthread_mutex_t` lock instance (as

```
// Return values and error checks omitted for simplicity.

pthread_mutex_lock(pthread_mutex_t *m) {
    optimized_mutex_t *om = get_optimized_mutex(m);
    if (om == null) {
        om = create_and_store_optimized_mutex(m);
    }
    optimized_mutex_lock(om);
    real_pthread_mutex_lock(m); // Acquiring the "real" mutex in order
                                // to support condition variables.
                                // Note that there is virtually no
                                // contention on this mutex.
}

pthread_mutex_unlock(pthread_mutex_t *m) {
    optimized_mutex_t *om = get_optimized_mutex(m);
    optimized_mutex_unlock(om);
    real_pthread_mutex_unlock(m);
}

pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m) {
    optimized_mutex_t *om = get_optimized_mutex(m);
    optimized_mutex_unlock(om);
    real_pthread_cond_wait(c, m);
    real_pthread_mutex_unlock(m); // We need to release the "real" mutex
    optimized_mutex_lock(om);     // otherwise if a thread calls
    real_pthread_mutex_lock(m);   // pthread_mutex_lock, grabs the
                                  // optimized mutex, and tries to
                                  // acquire the  "real" mutex, there
                                  // might be a deadlock, as the "real"
                                  // mutex lock is held after
                                  // real_pthread_cond_wait.
}

// Note that the pthread_cond_signal and pthread_cond_broadcast
// primitives do not need to be interposed.
```

Figure 3.1 – Pseudocode for the main wrapper functions of LiTL.

explained later in this Section). Second (and regardless of the previous reason), LiTL is aimed at being easily portable across C standard libraries. Given that the POSIX standard does not specify the memory layout and contents of the `pthread_mutex_t` structure[1], it it is non-trivial to devise an "in-place modification" approach that is at the same time safe, efficient and portable.

The above-mentioned design choice implies that LiTL must keep track of the life-cycle of all the locks through interposition of the calls to `pthread_mutex_init` and `pthread_mutex_destroy`, and that each interposed call to `pthread_mutex_lock` must trigger a lookup for the instance of the optimized lock. In addition, lock instances that are statically initialized can only be discovered and tracked upon the first invocation of `pthread_mutex_lock` on them (i.e., a failed lookup leads to the creation of a new mapping).

The `lock/unlock` API of several lock algorithms requires an additional parameter (called *struct* hereafter) in addition to the lock pointer, e.g., in the case of an MCS lock, this parameter corresponds to the record to be inserted in (or removed from) the lock's waiting queue. In the general case, a struct cannot be reused nor freed before the corresponding lock has been released. For instance, an application may rely on nested critical sections (i.e., a thread T must acquire a lock L_2 while holding another lock L_1). In this case, T must use a distinct struct for L_2 in order to preserve the integrity of L_1's struct. In order to gracefully support the most general cases, LiTL systematically allocates exactly one struct per lock instance and per thread (a static array is allocated alongside the lock instance, upon the first access to the lock instance), while taking care of avoiding false-sharing of cache lines among threads. LiTL uses the default memory allocator (glibc ptmalloc), which has per-thread arenas to avoid lock contention (since glibc 2.15) [56].

3.1.2 Supporting condition variables

Efficiently dealing with condition variables inside each optimized lock algorithm would be complex and tedious as most locks have not been designed with condition variables in mind. Indeed, most lock algorithms suffer from the so-called *thundering-herd* effect, where all waiting threads unnecessary contend on the lock after a call to `pthread_cond_broadcast`[2], which might lead to a scalability collapse. The Linux Pthread implementation does not suffer from the *thundering-herd* effect, as it only wakes up a single thread from the wait queue of the condition variable and directly transfers the remaining threads to the wait queue of the Pthread lock. However, to implement this optimization, all the waiting threads must block on a single memory

[1] In fact, different standard libraries [44, 46] and even different versions of the same library have significantly different implementations.

[2] 19 out of 40 of our studied application uses this operation, in most cases to implement barriers.

address[3], which is incompatible with lock algorithms that are not based on a competitive succession policy.

We therefore use the following generic strategy: our wrapper for `pthread_cond_wait` internally calls the actual `pthread_cond_wait` function. To issue this call, we hold a real Pthread mutex lock (of type `pthread_mutex_t`), which we systematically acquire just after the optimized lock. This strategy (depicted in the pseudocode of Figure 3.1) does not introduce high contention on the real Pthread lock. Indeed, (i) for workloads that do not use condition variables[4], the Pthread lock is only requested by the holder of the optimized lock associated with the critical section and, (ii) workloads that use condition variables are unlikely to have more than two threads competing for the Pthread lock (the holder of the optimized lock and a notified thread).

A careful reader might suggest to take the Pthread lock only before calling `pthread_cond_wait` on it. This approach has been proposed by Lozi et al. [72], but we discovered that it suffers from liveness hazards due to a race condition. Indeed, when a thread T calls `pthread_cond_wait`, it is not guaranteed that the two steps (releasing the lock and blocking the thread) are always executed atomically. Thus, a wake-up notification issued by another thread may get interleaved between the two steps and T may remain indefinitely blocked.

We acknowledge that the additional acquire and release calls to the uncontended[5] Pthread lock lengthen the critical section, which might increase the contention (i.e., multiple threads trying to acquire the lock simultaneously). However, the large number of studied applications (40) allows us to observe different critical-section lengths, and the different threads configurations considered (*one node*, *max nodes* and *opt nodes*) allow us to observe different probabilities of conflict for a given application.

3.1.3 Support for specific lock semantics

Our implementation is compliant with the specification of the `DEFAULT non-robust` POSIX mutex type [57]. More precisely, we do not support lock holder crashes (robustness), relocking the same lock can lead to deadlock or undefined behavior, and the behavior of unlocking a lock with a non-holder thread is undefined (it depends on the underlying lock algorithm).

[3]This is a restriction of the Linux `futex` syscall.

[4]LiTL comes with a switch to turn off the condition variable algorithm at compile time. However, in order to make fair comparisons, we always use LiTL with the condition variable algorithm turned on for all the studied applications.

[5]There is one case where there might be some contention on the Pthread lock: if a thread A is preempted just after `optimized_mutex_unlock` and another thread B acquires the lock at the same time, B will wait on the Pthread lock until A is scheduled again. However, even in this exceptional event, the contention remains very low.

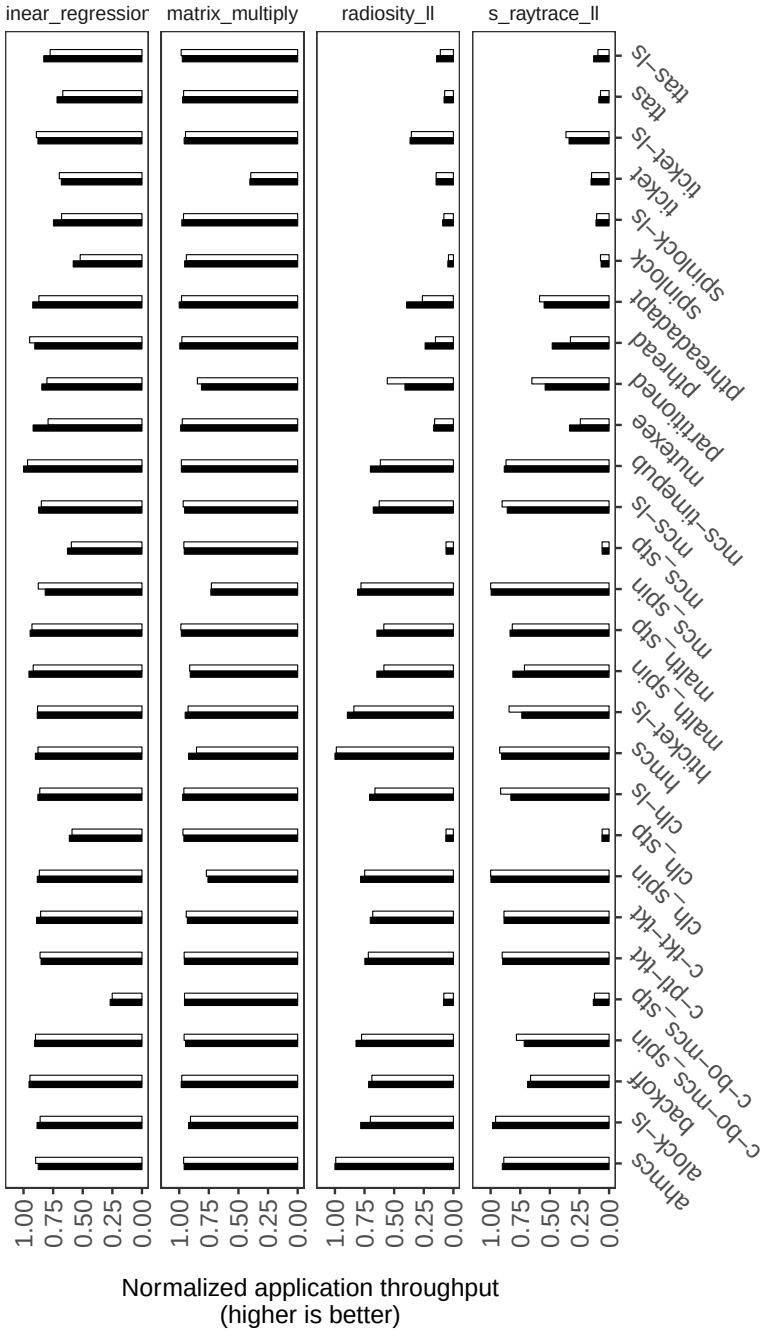


Figure 3.2 – Performance comparison (throughput) of manually implemented locks (black bars) vs. transparently interposed locks using LiTL (white bars) for 4 different applications.
The throughput is normalized with respect to the best performing configuration for a given application (**A-64 machine**).

The design of LiTL is compatible with specific lock semantics when the underlying lock algorithms offer the corresponding properties. For example, LiTL supports non-blocking lock requests (`pthread_mutex_trylock`) for all the currently implemented locks except CLH-based locks and HTicket-ls, which are not compatible with the try-lock non-blocking operation[6]. Although not yet implemented, LiTL could easily support blocking requests with timeouts for the so-called "abortable" locks (e.g., MCS-Try [96] and MCS-TimePub [54]). Moreover, support for optional Pthread mutex behavior like reentrance and error checks[7] could be easily integrated in the generic wrapper code by managing fields for the current owner and the lock acquisition counter. Note that none of the applications that we have studied requires a non-`DEFAULT` POSIX mutex type.

3.2 Implementation

The library relies on a scalable concurrent hash table (CLHT [31]) in order to store, for each Pthread mutex instance used in the application, the corresponding optimized lock instance, and the associated per-thread structs. For well-established locking algorithms like MCS, the code of LiTL borrows from other libraries [33, 4, 72, 42]. Other algorithms (i.e., CLH, C-BO-MCS, C-PTL-TKT, C-TKT-TKT, HMCS, AHMCS, Malthusian, Partitioned, Spinlock, TTAS) are implemented from scratch based on the description of the original papers. The complete list of implemented algorithms is discussed in Section 4.1.1. For algorithms that are based on a parking waiting policy, our implementation directly relies on the `futex` Linux system call.

Finally, the source code of LiTL relies on preprocessor macros rather than function pointers. We have observed that the use of function pointers in the critical path introduced a surprisingly high overhead (up to a 40% throughput decrease). Moreover, all data structures of the interposition library as well as the ones used to implement the lock algorithms are cache-aligned, in order to mitigate the effect of false sharing. The applications' data structures are not modified, as our approach aims at being transparent.

3.3 Lookup overhead

To assess the overhead of performing a lookup in the hash table each time a lock is accessed, we designed a micro-benchmark in which threads perform concurrent

[6]The design of the Partitioned (and by extension C-PTL-TKT) lock does not allow implementing a perfect trylock, i.e., a trylock that never blocks. As a consequence, if two threads try to acquire the lock simultaneously, one of them spins until the other thread releases the Partitioned lock.

[7]Using respectively the `PTHREAD_MUTEX_RECURSIVE` and `PTHREAD_MUTEX_ERRORCHECK` attributes.

lookups, varying the number of threads (from 1 to 64) and the number of elements[8] (from 1 to 32768). On the A-64 machine (4× 8-cores AMD Opteron 6272 machine, see Table 4.2 for more details), no matter the number of lock instances, at 1 thread, a look-up costs 20 cycles, and from 2 to 64 threads, 25 cycles. The 5-cycle difference is explained by the fact that on the A-64 machine, two siblings cores share some microarchitectural units of the CPU.

Regardless of the number of lock instances, the number of threads, and the lock algorithm (as only a pointer is stored), the cost is constant and low. In terms of memory footprint, CLHT stores 3 pairs *(pthread lock instance, optimized lock instance)* per 64-byte cache line. Overall, CLHT is a good choice as a hash map, and using a hash map should not influence the results significantly.

3.4 Experimental validation

In this Section, we assess the performance of LiTL using the A-64 machine. To that end, we compare the performance (throughput) of each lock on a set of applications running in two distinct configurations: manually modified applications and unmodified applications using interposition with LiTL. Clearly, one cannot expect to obtain exactly the same results in both configurations, as the setups differ in several ways, e.g., with respect to the exercised code paths, the process memory layout and the allocation of the locks (e.g., stack- vs. heap-based). However, we show that, for both configurations, (i) the achieved performance is close and (ii) the general trends for the different locks remain stable.

We selected four applications: linear_regression, matrix_multiply, radiosity_ll and s_raytrace_ll (see §4.1.3 for the complete list of studied applications). The first two applications do not use condition variables, thus allowing us to compare LiTL with manual lock implementation without the extra uncontended Pthread lock acquisition. Because the two others use condition variables, we compare LiTL with manual lock implementations and with the condition variable algorithm. These four applications are particularly lock-intensive: they represent unfavorable cases for LiTL. Moreover, we focus the discussion on the results under the highest contention level (i.e., when the application uses all the cores of the target machine), as this again represents an unfavorable case for LiTL.

Figure 3.2 shows the normalized performance (throughput) of both configurations (manual/interposed) for each *(application, lock)* pair. In addition, Table 3.1 summarizes the performance differences for each application.

[8]The key and value are both pointers – 8 bytes –, to the original pthread lock instance and to the LiTL lock instance (plus per-thread structs) respectively.

		linear_regression	matrix_multiply	radiosity_ll	s_raytrace_ll
Manual	# Cases where Manual is better	6	13	2	13
	Average gain	3 %	1 %	7 %	4 %
	Relative standard deviation	2 %	1 %	8 %	4 %
LiTL	# Cases where LiTL is better	22	15	26	15
	Average gain	3 %	2 %	3 %	3 %
	Relative standard deviation	3 %	2 %	3 %	4 %

Table 3.1 – Detailed statistics for the performance comparison of manually implemented locks vs. transparently interposed locks using LiTL. (**A-64 machine**).

We observe that, for all four applications, the results achieved by the two versions of the same lock are very close: the average performance difference is never higher than 8%. Besides, Figure 3.2 highlights that the general trends observed with the manual versions are preserved with the interposed versions.

	linear_regression	matrix_multiply	radiosity_ll	s_raytrace_ll
Best	97 %	98 %	100 %	98 %
Equals	87 %	93 %	91 %	93 %
Worse	96 %	99 %	98 %	98 %

Table 3.2 – Percentage of lock pairs *(A, B)* where, if performance with manually implemented locks of A is worse, equals or better than B, it is also respectively worse, equals or better than B with transparently interposed locks using LiTL.
We use a 5% threshold, i.e., A is better (resp. worse) than B if A's performance is at least 5% better (resp. worse) than B (**A-64 machine**).

Table 3.2 compares the relative performance of all lock pairs. The table shows that in most cases (at least 87%), comparing two manually implemented lock algorithms leads to the same conclusion as comparing their transparently interposed versions.

Application	C_n	p-value
linear_regression	−1.8 %	0.84
matrix_multiply	−0.2 %	0.60
radiosity_ll	−3.1 %	0.72
s_raytrace_ll	−0.2 %	0.85

Table 3.3 – For each application, the p-value of the paired Student t-test testing the null hypothesis $Mean_{with} - Mean_{without} = C$. C_n is C normalized w.r.t. the performance of the best lock on a given benchmark).

3.5 Statistical test

To assess that the conclusions we draw regarding the choice of a lock and the performance of locks with respect to each other (i.e., lock performance hierarchy) are the same with and without interposition, we use a *Student paired t-test*. A Student paired t-test tests if two populations for which observations can be paired have the same mean (for example, a population of patients before and after taking a medical treatment).

The null hypothesis tested is $Mean_{with} - Mean_{without} = 0$. However, because the goal is to assess that the lock performance hierarchy stays the same (not that the means are the same, i.e., strictly no overhead), $Mean_{with} - Mean_{without} = C$ is used as the null hypothesis, where C is a (per-application) constant. If C is a constant, then it means that there is a constant overhead, thus the lock performance hierarchy is left unchanged (contrary to an overhead dependent of the lock algorithm or proportional to the performance, in which case the lock performance hierarchy may change). Ideally, the constant C should be small enough, meaning that in addition to not affecting relative lock comparisons, the overhead of using LiTL on absolute performance is low. We choose C equal to the average throughput difference with and without interposition for all locks for a given application.

Table 3.3 shows the constant C_n (C normalized w.r.t. the performance of the best lock on a given benchmark) as well as the t-test's p-value. For example, for linear_regression, when removing 1.8% of the maximal throughput (0.03 seconds) to each interposed configuration, the p-value is 0.84. A p-value must be compared against a threshold α, upon which we reject/accept the null hypothesis (i.e., in our case, "means are equal, up to a constant"). The higher the p-value, the lower the risk to incorrectly reject the null hypothesis. All the tested applications have p-value > 0.05 (the most commonly used threshold [83]), thus we never reject the null hypothesis, thus the means can be considered equal (up to a constant C).

As a consequence, based on the results of the above table, we conclude that **using LiTL to study the behavior of locks algorithms only yields very modest differences with respect to the performance behavior of a manually modified version**.

4 Study

In this Chapter, we present our lock algorithms study. Section 4.1 presents the studied lock algorithms, testbed platforms and applications we study, as well as our tuning choices and our experimental methodology. Sections 4.2, 4.3 and 4.4 respectively present our results for throughput, energy efficiency and tail latency. Section 4.5 provides an analysis of the lock/application behavior.

4.1 Study's methodology

In this Section we describe the methodology of our study. Sections 4.1.1, 4.1.2 and 4.1.3 describe the different lock algorithm, testbed platforms and applications. Section 4.1.4 presents our tuning choices and our experimental methodology.

4.1.1 Studied algorithms

We now describe the 28 mutex lock algorithms that are representative of both well-established and state-of-the-art approaches. Our choice of studied locks is guided by the decision to focus on *portable* lock algorithms. We therefore exclude the following locks that require strong assumptions on the application/OS behavior, code modifications, or fragile performance tuning: HCLH, HBO, FC-MCS (see Dice et al. [38] for detailed arguments). We also do not study delegation-based algorithms, because they require critical sections to be expressed as a form of closure (i.e., functions) [38], which is incompatible with our transparent approach (i.e., without source code modification). Finally, we do not consider runtime approaches like LC and GLS, which require special kernel support and/or monitoring threads.

We use the *_Spin* and *_STP* suffixes to differentiate variants of the same algorithm that only differ in their waiting policy (pure spinning vs spin-then-park). Unless explicitly specified by the lock algorithm implementation, we use the `PAUSE` instruction to pause

between spinning loop iterations. The *-ls* tag corresponds to algorithm implementations borrowed from libslock [33]. As well, note that the GNU C library for Linux provides two versions of Pthread mutex locks [45]: the default one uses immediate parking (via the `futex` syscall) and the second one uses an adaptive spin-then-park strategy. The latter version can be enabled with the `PTHREAD_MUTEX_ADAPTIVE_NP` option [68]. Our set of algorithms is summarized in Table 4.1 and includes eight competitive succession locks (Backoff, Mutexee, Pthread, PthreadAdapt, Spinlock, Spinlock-ls, TTAS, TTAS-ls), ten direct handoff locks (ALock-ls, CLH-ls, CLH_Spin, CLH_STP, MCS-ls, MCS_Spin, MCS_STP, Ticket, Ticket-ls, Partitioned), six hierarchical locks (C-BO-MCS_Spin, C-BO-MCS_STP, C-PTL-TKT, C-TKT-TKT, HTicket-ls, HMCS), and four load-control locks (AHMCS, Malth_Spin, Malth_STP, MCS-TimePub).

	Name	Reference	Short description
competitive	**Backoff**	[80]	Test-and-set (TAS) with exponential bounded backoff if the lock is already held.
	Mutexee	[42]	A spin-then-park (STP) lock designed with energy efficiency in mind.
	Pthread	[46]	TAS with direct parking.
	PthreadAdapt	[68]	An adaptive STP algorithm, performing a number of trylocks (before blocking) that depends on the number of trylocks performed by the lock holder when it acquired the lock.
	Spinlock	[9]	Compare-and-set algorithm with busy waiting.
	Spinlock-ls	[33]	TAS algorithm with busy waiting.
	TTAS	[9]	Performs non-atomic loads on the lock memory address before trying to acquire it atomically with a TAS instruction.
	TTAS-ls	[33]	Similar to TTAS but uses an exponential bounded backoff if the TAS fails.
direct handoff	**ALock-ls**	[9]	The *waiting* threads are organized inside a fixed-sized array, i.e., there is a fixed bound N on the number of waiting threads. A thread waits on one of the private cache-aligned array slot. At unlock-time, the lock holder wakes the next thread by changing the content of the slot the next thread waits on.
	CLH_Spin	[29, 76]	Waiting threads are organized as an inverse linked-list, where a thread spins on the context (i.e., linked-list node) of its predecessor. At unlock-time, the lock holder wakes up the thread at the head of the waiting list.
	CLH_STP	[29, 76]	Similar to CLH_Spin but uses a STP waiting policy.
	CLH-ls	[33]	Similar to CLH_Spin but uses the `PREFETCHW` x86 CPU instruction while spinning.
	MCS_Spin	[80]	Waiting threads are organized as a linked-list, where a thread spins on its private context. At unlock-time, the lock holder wakes up its successor.
	MCS_STP	[80]	Similar to MCS_Spin but uses a STP waiting policy.
	MCS-ls	[33]	Similar to MCS_Spin but uses the `PREFETCHW` x86 CPU instruction while spinning.
	Ticket	[91]	A thread trying to acquire the lock atomically takes a "ticket" (implemented as an incrementing counter) and spins while its ticket is not equal to the "next-ticket" number. At unlock-time, the lock holder increments the "next-ticket" number.
	Ticket-ls	[33]	Similar to Ticket but a thread waits proportionally to the number of threads waiting before him.
	Partitioned	[36]	Similar to Ticket but the "next-ticket" number is implemented inside an array, where a thread waits on its "ticket" slot ($slot = ticket \,\%\, size(array)$).
hierarchical	**C-BO-MCS_Spin**	[38]	A thread first tries to acquire a MCS_Spin local lock shared by all threads on the same NUMA node (the local lock), then competes on the Backoff top lock with other threads holding their respective local locks.
	C-BO-MCS_STP	[38]	Similar to C-BO-MCS_Spin but uses a STP waiting policy.
	C-PTL-TKT	[38]	Similar to C-BO-MCS_Spin but the local locks are Ticket locks and the top lock is a Partitioned lock.
	C-TKT-TKT	[38]	Similar to C-BO-MCS_Spin but the top and local locks are Ticket locks.
	HTicket-ls	[33]	Similar to C-TKT-TKT but a thread waits proportionally to the number of threads waiting before him.
	HMCS	[25]	Similar to C-BO-MCS_Spin but the top and local locks are MCS_Spin locks.
load-control	**AHMCS**	[24]	Similar to HMCS, but when a thread tries to acquire the lock, it remembers if the last time it released the lock there was a thread waiting. If not, it only locks the top lock because it assumes low contention the lock.
	Malth_Spin	[34]	A variant of the MCS_Spin lock where, when there is contention on a lock, a subset of the spinning competing threads are put aside temporarily to let the others progress more easily.
	Malth_STP	[34]	Similar to Malth_Spin but threads use a STP waiting policy.
	MCS-TimePub	[54]	A variant of the MCS_Spin lock, in which a waiting thread relinquishes its core if it detects (heuristically, using timers and thresholds) that the lock holder has been preempted. At unlock-time, the lock holder might bypass some waiting threads if it detects they have been preempted.

Table 4.1 – A short description of the 28 multicore lock algorithms we consider.

4.1.2 Testbed

Name Total #cores	**A-64** 64	**A-48** 48
Server model	Dell PE R815	Dell PE R815
Processors Microarchitecture Clock frequency Last-level cache (per node) Introduction date	4× AMD Opteron 6272 Bulldozer / Interlagos 2.1 GHz 8 MB 2011	4× AMD Opteron 6344 Piledriver / Abu Dhabi 2.6 GHz 8 MB 2012
Interconnect	HT3 - 6.4 GT/s per link	HT3 - 6.4 GT/s per link
Memory #NUMA nodes (#cores/node)	256 GB DDR3 1600 MHz 8 (8)	64 GB DDR3 1600 MHz 8 (6)
Network interfaces (10 GbE)	2× 2-port Intel 82599	2× 2-port Intel 82599
OS & tools Linux kernel glibc gcc	Ubuntu 12.04 3.17.6 (CFS scheduler) 2.15 4.6.3	Ubuntu 12.04 3.17.6 (CFS scheduler) 2.15 4.6.3

Name Total #cores	**I-48** 48 (no hyperthreading)	**I-20** 20 (no hyperthreading)
Server model	SuperMicro SS 4048B-TR4FT	SuperMicro X9DRW
Processors Microarchitecture Clock frequency Last-level cache (per node) Introduction date	4× Intel Xeon E7-4830 v3 Haswell-EX 2.1 GHz 30 MB 2015	2× Intel Xeon E5-2680 v2 Ivy Bridge-EP 2.8 GHz 25 MB 2013
Interconnect	QPI - 8 GT/s per link	QPI - 8 GT/s per link
Memory #NUMA nodes (#cores/node)	256 GB DDR4 2133 MHz 4 (12)	256 GB DDRR 1600 MHz 2 (10)
Network interfaces (10 GbE)	2-port Intel X540-AT2	-
OS & tools Linux kernel glibc gcc	Ubuntu 12.04 3.17.6 (CFS scheduler) 2.15 4.6.4	Ubuntu 14.04 3.13 (CFS scheduler) 2.19 4.6.3

Table 4.2 – Hardware characteristics of the testbed platforms.

Our experimental testbed consists of four Linux-based x86 multicore servers whose main characteristics are summarized in Table 4.2. All the machines run the Ubuntu 12.04 OS with a 3.17.6 Linux kernel (CFS scheduler), except the I-20 machine running an Ubuntu 14.04 OS with a 3.13 Linux kernel. We tried to keep the software configuration as similar as possible for the different versions: they all use glibc (GNU C Library) version 2.15 (2.19 for I-20) and gcc version 4.6.3 (4.6.4 on I-48). We configured the BIOS of the A-64 and the A-48 machines in performance mode (processor throttling is turned off so that all cores run at maximum speed, e.g., no C-state, no turbo mode). The BIOS of the I-48 and I-20 machines in performance mode for the throughput experiments, and in energy-saving mode for the energy-efficiency experiments. For all configurations, hyper-threading is disabled.

4.1.3 Studied applications

Application	Benchmark Suite	Type
kyotocabinet	-	database
memcached-old	-	memory cache
memcached-new	-	memory cache
mysqld	-	database
rocksdb	-	key/value store
sqlite	-	database
ssl_proxy	-	ssl reverse proxy
upscaledb	-	key/value store
blackscholes	PARSEC 3.0	financial analysis
bodytrack	PARSEC 3.0	computer vision
canneal	PARSEC 3.0	engineering
dedup	PARSEC 3.0	enterprise storage
facesim	PARSEC 3.0	animation
ferret	PARSEC 3.0	similarity search
fluidanimate	PARSEC 3.0	animation
freqmine	PARSEC 3.0	data mining
p_raytrace	PARSEC 3.0	rendering
streamcluster	PARSEC 3.0	data mining
streamcluster_ll	PARSEC 3.0	data mining
swaptions	PARSEC 3.0	financial analysis
vips	PARSEC 3.0	media processing
x264	PARSEC 3.0	media processing
histogram	Phoenix 2	image
kmeans	Phoenix 2	statistics
linear_regression	Phoenix 2	statistics
matrix_multiply	Phoenix 2	mathematical computations
pca	Phoenix 2	statistics
pca_ll	Phoenix 2	statistics
string_match	Phoenix 2	text processing
barnes	SPLASH2x	physics simulation
fft	SPLASH2x	mathematical computations
fmm	SPLASH2x	physics simulation
lu_cb	SPLASH2x	mathematical computations
lu_ncb	SPLASH2x	mathematical computations
ocean_cp	SPLASH2x	physics simulation
ocean_ncp	SPLASH2x	physics simulation
radiosity	SPLASH2x	rendering
radiosity_ll	SPLASH2x	rendering
radix	SPLASH2x	sorting
s_raytrace	SPLASH2x	rendering
s_raytrace_ll	SPLASH2x	rendering
volrend	SPLASH2x	rendering
water_nsquared	SPLASH2x	physics simulation
water_spatial	SPLASH2x	physics simulation
word_count	SPLASH2x	text processing

Table 4.3 – Real-world applications considered.

Table 4.3 lists the applications we chose for our comparative study of lock performance

and lock energy efficiency. More precisely, we consider (i) the applications from the PARSEC benchmark suite version 3.0 (emerging workloads) [17], (ii) the applications from the Phoenix 2.0 MapReduce benchmark suite [90], (iii) the applications from the SPLASH2x high-performance computing benchmark suite [17][1], (iv) the MySQL database version 5.7.7 [84] running the Cloudstone workload [100], (v) SSL proxy, an event-driven SSL endpoint written with the Boost C++ library that processes small messages, (vi) upscaledb 2.2.0 [26], an embedded key/value running the ham_bench benchmark, (vii) the Kyoto Cabinet database version 1.2.76 [41], a standard relational database management system running the included benchmark, (viii) Memcached, versions 1.4.15 and 1.4.36[2] [81], an in-memory cache system, (ix) RocksDB 4.8 [40], a persistent key/value store running the included benchmark, and (x) SQLite 3.13 [101], an embedded SQL database using the dbt2 TPC-C workload generator[3]. We use remote network injection for the MySQL and the SSL proxy applications. For Memcached, similarly to other setups used in the literature [72, 42], the workload runs on a single machine: we dedicate one socket of the machine where we run `memaslap` to inject network traffic to the Memcached instance, the two running on two distinct sets of cores. For the Kyoto Cabinet application, like in previous work [34], we redirect calls to `rw_lock` to classic `mutex_lock` calls. This might change the synchronization pattern of the application, yet this aplication is still interesting to consider because its performance is known to vary according to lock algorithms [22]. By default, phoenix launches one thread per available core, and pins each thread to one core. However, to have the same baseline for all our benchmarks, we decided to disable pinning in phoenix, leaving to the scheduler the thread placement decisions. Note that when benchmarks are evaluated in a thread-to-node pinning configuration (see Section 4.2.3), phoenix is also evaluated on a thread-to-node pinning configuration.

In order to evaluate the impact of workload changes on locking performance and energy efficiency, we also consider "long-lived" variants of four of the above workloads (pca, s_raytrace, radiosity and streamcluster) denoted with a "_ll" suffix. The motivation behind these versions is to stress the application's steady-state phase, where the locks are mostly acquired/released. By contrast, the short-lived versions allow us to benchmark the performance of the initialization and cleanup operations of an application. For each application, we modified it to report throughput (in operations per seconds, e.g., number of rays traced for an application that renders a 3-D phase) and use larger input size. We capture the throughput of the "steady-state" phase exclusively, ignoring the impact of the start/shutdown phases. Note that six of the applications only accept, by design, a number of threads that corresponds to a power of two: facesim and fluidanimate from PARSEC, fft, ocean cp, ocean ncp, radix, all from SPLASH2x. We decide to not include experiments for these six applications on the two

[1] We excluded the Cholesky application because of extremely short completion times.

[2] Memcached 1.4.15 uses a global lock to synchronize all accesses to a shared hash table. This lock is known to be the main bottleneck. Newer versions use per-bucket locks, thus suffer less from contention.

[3] https://sourceforge.net/projects/osdldbt/

48-core machines and the 20-core machine, in order to keep the presentation of results uniform and easy to understand. Besides, we were not able to evaluate the applications using network injection on the I-20 machine due to a lack of high-throughput network connectivity.

Some *(application,lock algorithm,machine)* configurations cannot be evaluated, for the following reasons. First, due to a lack of memory (especially on the A-48, which only has 64 GB of memory), and because some applications allocate too many lock instances and the memory footprint of some lock algorithms is high: (i) AHMCS with dedup and fluidanimate on all machines, and (ii) CLH, ALock-ls, TTAS-ls with dedup on A-48 results are not reported. Second, fluidanimate, Memcached-old, Memcached-new, streamcluster, streamcluster_ll, vips rely on trylock operations. CLH algorithms and HTicket-ls do not support trylock, and Partitioned and C-PTL-TKT trylock implementations might block threads for a short time (which can cause deadlocks with Memcached-*). Those configurations are not evaluated. Finally, most of the studied applications use a number of threads equal to the number of cores, except the four following ones: dedup (3× threads), ferret (4× threads), MySQL (hundreds of threads) and SQLite (hundreds of threads). For applications with significantly more threads than cores (SQLite and MySQL), we exclude results for algorithms using a spinning waiting policy: these applications suffer from the lock holder preemption issue (see Section 4.5.1 for more details) up to a point where performance drops close to zero.

4.1.4 Tuning and experimental methodology

For the lock algorithms that rely on static thresholds, we use the recommended values from the original papers and implementations. The algorithms based on a spin-then-park waiting policy (e.g., Malth_STP [34]) rely on a fixed threshold for the spinning time that corresponds to the duration of a round-trip context switch [63]—in this case, we calibrate the duration using a microbenchmark on the testbed platform. All the applications are run with memory interleaving (via the `numactl` utility) in order to avoid NUMA memory bottlenecks[4]. Datasets are copied inside a temporary file-storage facility (tmpfs) before running experiments, to avoid disk I/O. For most of the experiments, the application threads are not pinned to specific cores. Note that for hierarchical locks, which are composed of one top lock and one per-NUMA node bottom lock, a thread always tries to acquire the bottom lock where it is *currently* running[5]. Doing so, cache coherence traffic is limited, which is one of the main reason behind the design of hierarchical locks. The effect of pinning is nonetheless discussed

[4]For the Memcached-* experiments where some nodes are dedicated to network injection, memory is interleaved only on the nodes dedicated to the server.

[5]Before acquiring a lock, outside the critical section, the thread queries its current node via the `rdtscp` instruction. Upon acquisition, a pointer to a local lock is stored inside the lock instance data structure, which allows the lock holder at unlock time to release the appropriate local lock, even if this thread has been migrated to another NUMA node while it was inside the critical section.

in Section 4.2.3.

Generally, in the experiments presented, we study both the throughput, the energy-efficiency impact and the tail latency (here defined as the 99th percentile of client reponse time) of a lock algorithm for a given level of contention, i.e., the number of threads of the application. We vary the level of contention at the granularity of a NUMA node (i.e., 8 cores for the A-64 machine, 6 cores for the A-48 machine, 12 cores for the I-48 machine and 10 cores for the I-20 machine). Note that for Memcached-old and Memcached-new, we use one socket of the machine to run the injection threads, so the maximum number of cores tested is lower than the total number of cores on the machine: the figures and tables are modified to take this into account.

We consider three metrics: application-level throughput, tail latency, and energy efficiency. More precisely, for throughput, (i) for MySQL, SSL Proxy, upscaledb, Kyoto Cabinet, RocksDB and SQLite, the application throughput is used as a performance metric, (ii) for the long-lived applications, progress points are inserted in the source code of the application, and (iii) for all the other applications, the inverse of the total execution time is used. For tail latency, we consider the application tail latency, here defined as the 99th percentile of client response time. We perform energy consumption measurements using the RAPL (Running Average Power Limit) [58] power meter interface on the two Intel machines (I-48 and I-20). RAPL is an on-chip facility that provides counters to measure the energy consumption of several components: cores, package and DRAM. We do not capture energy for our two AMD machines as they do not have APM (Application Power Management), AMD's version of RAPL.

We run each experiment at least 5 times[6] and compute the average value. For long-lived and server workloads, a 30-second warmup phase precedes a 60-second capture phase, before killing the application. In pratice, for the applications we studied, these timings are high enough to reach the best stable performance of the application. For configurations exhibiting high variability (i.e., more than 5% of relative standard deviation), we run more experiments, trying to lower the relative standard deviation of the configuration, to increase the confidence in our results. More precisely, we found that roughly 15% of the *(application, lock algorithm, machine, number of threads)* configurations have a relative standard deviation (rel.stdev.) higher than 5%. Besides, 6% of the configurations have a rel.stdev higher than 10% and 2% higher than 20%. C-BO-MCS_STP, TTAS and Spinlock-ls are the studied lock algorithms that exhibit the higher variability: the rel.stdev of these locks is higher than 5% for 20% of the configurations. Concerning the applications, ocean_cp, ocean_ncp, streamcluster and fft exhibit a high rel.stdev (roughly 50% of the configurations have a rel.stdev higher than 5%). Finally, streamcluster, dedup and streamcluster_ll are applications for which some configurations exhibit a very high rel.stdev (higher than 20% in 10% of the cases). In order to

[6]The number of experiments is choosen as a good sweet-spot between having enough runs to compute a significant average value and the overall experiment time of our study.

mitigate the effects of variability, when comparing two locks, we consider a margin of 5%: lock A is considered better than lock B if B's performance (resp. energy efficiency or tail latency) is below 95% of A's. Besides, in order to make fair comparisons among applications, the results presented for the Pthread locks are obtained using the same library interposition mechanism (see Chapter 3) as with the other locks.

Finally, for the sake of space, we do not report all the results for the four studied machines. We rather focus on the A-64 machine for the different studies and provide summaries of the results for the other machines, which are in accordance to the results on the A-64 machine. Nevertheless, the entire set of results can be found in the companion technical report [50]. We also do not systematically report, for the sake of readability, the standard deviations as they are low for most configuration. Note that the raw dataset (for all the experiments, on all machines) of throughput, tail latency and energy is available online [51], letting the readers perform their own analysis.

4.2 Study of lock throughput

In this Section, we use LiTL to compare the performance (throughput) behavior of the different lock algorithms on different workloads and at different levels of contention. Our experimental methodology is described in Section 4.1. In Sections 4.3 and 4.4 we present the results for energy efficiency and tail latency, respectively.

As a summary, Section 4.2.1 provides preliminary observations that drive the study. Section 4.2.2 answers the main questions of the study regarding the observed lock behavior. Section 4.2.3 discusses additional observations, such as how the machine, the BIOS configuration, and the thread pinning affect the results as well as the performance of Pthread locks. Section 4.2.4 discusses the implications of our study for software developers and for the lock algorithm research community.

4.2.1 Preliminary observations

Before proceeding with the detailed study, we highlight some important characteristics of the applications.

Selection of lock-sensitive applications

Table 4.4 shows two metrics for each application and for different numbers of nodes on the A-64 machine (results for the other machines are available in the companion technical report [50]): the performance gain of the best lock over the worst one, as well as the relative standard deviation for the performance of the different locks. Note that the columns of Table 4.4 cannot be compared to each other. Indeed, the numbers reported are the performance gain and relative standard deviation for the best vs. worst lock at a given number of nodes, i.e., gain at *max nodes* compares the performance of the best vs. worst lock at *max nodes*, whereas gain at *opt nodes* compares the performance of the best vs. worst lock at their *respective* optimal number of nodes (where they perform best).

Besides, the numbers reported at max nodes are generally higher than at *opt nodes* because performance gaps between locks tend to increase under high contention, which is why we chose the A-64 machine: it has the highest number of cores among our different machines. For the moment, we only focus on the relative standard deviations at the maximum number of nodes (*max nodes*—highest contention) given in the fifth column (the detailed results from this table are discussed in Section 4.2.2).

We consider that an application is *lock-sensitive* if the relative standard deviation for the performance of the different locks at *max nodes* is higher than 10% (highlighted in bold font in the Table). More precisely, we observe that about 60% of the applications are

affected by locks, for all machines except the I-20 where the percentage of application is slightly lower (49%). Table 4.5 summarizes the results for the four studied machines. Some applications are lock-sensitive on some machines and not on others. For example, fmm is only lock-sensitive on the AMD machines, not the Intel ones. For such applications, we observe a moderate relative standard deviation at *max nodes* (30%), meaning that they are considered lock-sensitive but they are not the applications that are the most affected by locks. Indeed, we do not observe applications that are highly affected by locks on one machine and not on another. In the remainder of this study, we focus on lock-sensitive applications.

Selection of the number of nodes

In multicore applications, optimal performance is not always achieved at the maximum number of available nodes (abbreviated as *max nodes*) due to various kinds of scalability bottlenecks. Therefore, for each *(application, lock)* pair, we empirically determine the *optimized configuration* (abbreviated as *opt nodes*), i.e., the number of nodes that yields the best performance. For the A-64 and A-48 machines, we consider 1, 2, 4, 6, and 8 nodes. For the I-48 machine, we consider 1, 2, 3, and 4 nodes. For the I-20 machine, we consider 1 and 2 nodes. Note that 6 nodes on A-64 and A-48 correspond to 3 nodes on I-48, i.e., 75% of the available cores.

Table 4.6 shows for each *(application, lock)* pair, for the A-64 machine the performance gain of *opt nodes* over *max nodes* and the number of nodes for *opt nodes* (results for the other machines are available in the companion technical report [50]). A line full of black boxes means that the optimal number of nodes is the maximal number of nodes, i.e., for all locks, the best performance is seen at *max nodes* (the performance of the application does not collapse). However, it is still interesting to consider these applications, because a line full of black boxes does not mean that all locks performs the same, e.g., for water_nsquared, the gain between the best vs. the worst locks at *max nodes* and *opt nodes* is of 94% (Table 4.4). In addition, Table 4.7 provides a breakdown of the *(application, lock)* pairs according to their optimized number of nodes for all machines.

We observe that, for many applications, the optimized number of nodes is lower than the max number of nodes. Moreover, we observe (Table 4.6) that the performance gain of the optimized configuration is often extremely large. We note that the performance gains for the I-20 is lower than the ones for the other machines, which have more cores. This confirms that tuning the degree of parallelism has frequently a very strong impact on performance. We also notice that, for some applications, the optimized number of nodes varies according to the chosen lock (on pca_ll ALock-ls is optimal at 4 nodes, Backoff at 8 nodes), the chosen waiting policy (on pca_ll Malth_Spin is optimal at 4 nodes, Malth_STP at 8 nodes) and the workload (Backoff is optimal at 2 nodes on pca

	Gain *one node*	R.Dev. *one node*	Gain *max nodes*	R.Dev. *max nodes*	Gain *opt nodes*	R.Dev. *opt nodes*
barnes	10 %	2 %	36 %	8 %	31 %	7 %
blackscholes	11 %	2 %	2 %	1 %	2 %	1 %
bodytrack	1 %	0 %	9 %	2 %	4 %	1 %
canneal	5 %	1 %	7 %	2 %	7 %	2 %
dedup	**819 %**	**57 %**	**989 %**	**54 %**	**819 %**	**57 %**
facesim	**9 %**	**2 %**	**771 %**	**67 %**	**13 %**	**3 %**
ferret	**1 %**	**0 %**	**349 %**	**56 %**	**101 %**	**25 %**
fft	8 %	2 %	11 %	3 %	9 %	2 %
fluidanimate	**48 %**	**11 %**	**284 %**	**28 %**	**127 %**	**20 %**
fmm	**17 %**	**5 %**	**42 %**	**10 %**	**42 %**	**10 %**
freqmine	7 %	2 %	6 %	1 %	6 %	1 %
histogram	7 %	2 %	19 %	5 %	13 %	3 %
kmeans	9 %	3 %	12 %	2 %	12 %	2 %
kyotocabinet	**414 %**	**25 %**	**2047 %**	**56 %**	**414 %**	**25 %**
linear_regression	**9 %**	**3 %**	**198 %**	**20 %**	**49 %**	**9 %**
lu_cb	8 %	2 %	5 %	1 %	5 %	1 %
lu_ncb	26 %	5 %	8 %	2 %	8 %	2 %
matrix_multiply	**6 %**	**2 %**	**608 %**	**26 %**	**169 %**	**20 %**
memcached-new	**63 %**	**15 %**	**1021 %**	**53 %**	**120 %**	**19 %**
memcached-old	**73 %**	**14 %**	**308 %**	**50 %**	**73 %**	**14 %**
mysqld	**166 %**	**42 %**	**174 %**	**36 %**	**122 %**	**33 %**
ocean_cp	**19 %**	**4 %**	**129 %**	**14 %**	**21 %**	**4 %**
ocean_ncp	**16 %**	**4 %**	**113 %**	**12 %**	**14 %**	**4 %**
p_raytrace	2 %	0 %	1 %	0 %	2 %	0 %
pca	**5 %**	**2 %**	**347 %**	**32 %**	**40 %**	**8 %**
pca_ll	**6 %**	**1 %**	**713 %**	**44 %**	**160 %**	**20 %**
radiosity	**3 %**	**1 %**	**91 %**	**15 %**	**13 %**	**4 %**
radiosity_ll	**10 %**	**2 %**	**2285 %**	**68 %**	**176 %**	**26 %**
radix	3 %	1 %	8 %	2 %	8 %	2 %
rocksdb	4 %	1 %	16 %	4 %	16 %	4 %
s_raytrace	**9 %**	**2 %**	**1898 %**	**58 %**	**232 %**	**31 %**
s_raytrace_ll	**5 %**	**1 %**	**1601 %**	**63 %**	**402 %**	**51 %**
sqlite	**66 %**	**19 %**	**2382 %**	**102 %**	**81 %**	**25 %**
ssl_proxy	**37 %**	**6 %**	**1309 %**	**59 %**	**58 %**	**11 %**
streamcluster	**14 %**	**3 %**	**1122 %**	**56 %**	**14 %**	**3 %**
streamcluster_ll	**24 %**	**5 %**	**1423 %**	**56 %**	**35 %**	**8 %**
string_match	5 %	2 %	11 %	2 %	11 %	2 %
swaptions	8 %	2 %	10 %	2 %	10 %	2 %
upscaledb	**158 %**	**22 %**	**748 %**	**43 %**	**197 %**	**24 %**
vips	**2 %**	**1 %**	**197 %**	**25 %**	**5 %**	**1 %**
volrend	**7 %**	**1 %**	**163 %**	**22 %**	**24 %**	**5 %**
water_nsquared	**10 %**	**2 %**	**94 %**	**14 %**	**94 %**	**14 %**
water_spatial	**23 %**	**5 %**	**98 %**	**15 %**	**96 %**	**15 %**
word_count	4 %	1 %	19 %	3 %	12 %	2 %
x264	4 %	1 %	6 %	2 %	5 %	2 %

Table 4.4 – For each application, performance gain of the best vs. worst lock and relative standard deviation (**A-64 machine**).

	A-64	A-48	I-48	I-20
# tested applications	45	39	37	35
# lock-sensitive applications	28	23	21	17
ratio	62 %	59 %	57 %	49 %

Table 4.5 – Number of applications and number of lock performance sensitive applications (**all machines**).

and at 8 nodes on pca_ll).

4.2.2 Main questions

How much do locks affect applications?

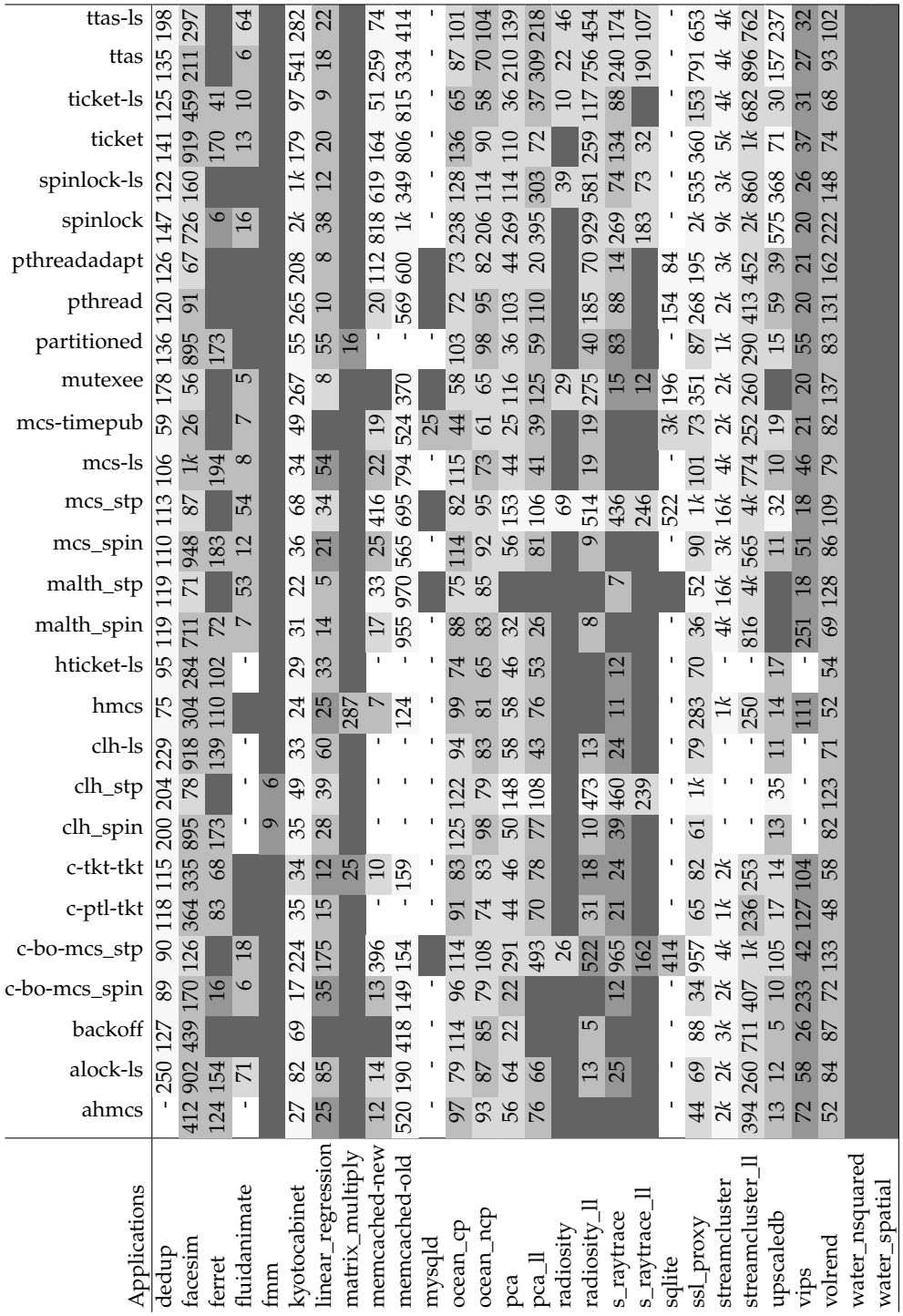
Table 4.4 shows, for each application, the performance gain of the best lock over the worst one at *one node*, *max nodes*, and *opt nodes* for the A-64 machine. The table also shows the relative standard deviation for the performance of the different locks.

We observe that the number of nodes affects the performance of applications. **At *one node*, the impact of locks on lock-sensitive applications is moderate for most applications.** Nonetheless, for the most lock-sensitive ones (upscaledb, MySQL, Kyoto Cabinet, dedup), we observe that the impact is high. More precisely, most applications exhibit a gain of the best lock over the worst one that is lower than 30%. In contrast, **at *max nodes*, the impact of locks is very high for all lock-sensitive applications**. More precisely, the gain brought by the best lock over the worst lock ranges from 42% to 2382%. Finally, **at *opt nodes*, the impact of locks is high, but noticeably lower than at *max nodes*.** We explain this difference by the fact that, at *max nodes*, some of the locks trigger a performance collapse for certain applications (as shown in Table 4.6), which considerably increases the observed performance gaps between locks. Note that the collapse is not necessarily related to a given lock, but is also a property of the application and how the machine behaves We observe the same trends on the A-48, the I-48 and the I-20 machines (see the companion technical report [50]).

Are some locks always among the best?

Table 4.8 displays, for each machine, the coverage of each lock, i.e., how often it stands as the best one (or is within 5% of the best) over all the studied applications, over the different locks. Table 4.9 shows the per-lock results for the A-64 machine. The details for the other machines are available in the companion technical report [50].

We make the following observations. On the A-64, A-48 and I-48 machines, **no lock is among the best for more than 76% of the applications at *one node* and for more than**



Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	250	127	89	90	118	115	200	204	229	75	95	119	119	110	113	106	59	178	136	120	126	147	122	141	125	135	198
facesim	412	902	439	170	126	364	335	895	78	918	304	284	711	71	948	87	1k	26	56	895	91	67	726	160	919	459	211	297
ferret	124	154		16		83	68	173		139	110	102	72		183		194			173			6		170	41		
fluidanimate	-	71		6	18			-	-	-		-	7	53	12	54	8	7	5				16		13	10	6	64
fmm								9	6																			
kyotocabinet	27	82	69	17	224	35	34	35	49	33	24	29	31	22	36	68	34	49	267	55	265	208	2k	1k	179	97	541	282
linear_regression	25	85		35	175	15	12	28	39	60	25	33	14	5	21	34	54		8	55	10	8	38	12	20	9	18	22
matrix_multiply							25				287									16								
memcached-new	12	14		13	396	-	10	-	-	-	7	-	17	33	25	416	22	19		-	20	112	818	619	164	51	259	74
memcached-old	520	190	418	149	154	-	159	-	-	-	124	-	955	970	565	695	794	524	370	-	569	600	1k	349	806	815	334	414
mysqld	-	-	-	-		-	-	-	-	-	-	-	-		-		-	25		-				-	-	-	-	-
ocean_cp	97	79	114	96	114	91	83	125	122	94	99	74	88	75	114	82	115	44	58	103	72	73	238	128	136	65	87	101
ocean_ncp	93	87	85	79	108	74	83	98	79	83	81	65	83	85	92	95	73	61	65	98	95	82	206	114	90	58	70	104
pca	56	64	22	22	291	44	46	50	148	58	58	46	32		56	153	44	25	116	36	103	44	269	114	110	36	210	139
pca_ll	76	66			493	70	78	77	108	43	76	53	26		81	106	41	39	125	59	110	20	395	303	72	37	309	218
radiosity					26											69			29					39		10	22	46
radiosity_ll		13	5		522	31	18	10	473	13			8		9	514	19	19	275	40	185	70	929	581	259	117	756	454
s_raytrace		25		12	965	21	24	39	460	24	11	12		7		436			15	83	88	14	269	74	134	88	240	174
s_raytrace_ll					162				239							246			12				183	73	32		190	107
sqlite	-	-	-	-	414	-	-	-	-	-	-	-	-			522	-	3k	196	-	154	84	-	-	-	-	-	-
ssl_proxy	44	69	88	34	957	65	82	61	1k	79	283	70	36	52	90	1k	101	73	351	87	268	195	2k	535	360	153	791	653
streamcluster	2k	2k	3k	2k	4k	1k	2k	-	-	-	1k	-	4k	16k	3k	16k	4k	2k	2k	1k	2k	3k	9k	3k	5k	4k	4k	4k
streamcluster_ll	394	260	711	407	1k	236	253	-	-	-	250	-	816	4k	565	4k	774	252	260	290	413	452	2k	860	1k	682	896	762
upscaledb	13	12	5	10	105	17	14	13	35	11	14	17			11	32	10	19		15	59	39	575	368	71	30	157	237
vips	72	58	26	233	42	127	104	-	-	-	111	-	251	18	51	18	46	21	20	55	20	21	20	26	37	31	27	32
volrend	52	84	87	72	133	48	58	82	123	71	52	54	69	128	86	109	79	82	137	83	131	162	222	148	74	68	93	102
water_nsquared																												
water_spatial																												

Table 4.6 – For each *(lock-sensitive application, lock)* pair, performance gain (in %) of *opt nodes* over *max nodes*. The background color of a cell indicates the number of nodes for *opt nodes*: 1|2|4|6|8 . Dashes correspond to untested cases (**A-64 machine**).

	A-64	A-48
1 Node	19 %	16 %
2 Nodes	23 %	21 %
4 Nodes	26 %	23 %
6 Nodes	11 %	16 %
8 Nodes	21 %	24 %

	I-48
1 Node	37 %
2 Nodes	17 %
3 Nodes	17 %
4 Nodes	29 %

	I-20
1 Node	39 %
2 Nodes	61 %

Table 4.7 – Breakdown of the *(lock-sensitive application, lock)* pairs according to their optimized number of nodes (**all machines**).

.

# nodes	Coverage	A-64	A-48	I-48	I-20
1	[min; max]	[39 %; 73 %]	[33 %; 71 %]	[21 %; 76 %]	[42 %; 75 %]
	Average	59 %	59 %	51 %	57 %
	Relative Standard Deviation	10 %	11 %	14 %	10 %
Max	[min; max]	[0 %; 29 %]	[0 %; 33 %]	[0 %; 47 %]	[8 %; 75 %]
	Average	14 %	14 %	19 %	42 %
	Relative Standard Deviation	8 %	9 %	13 %	16 %
Opt	[min; max]	[15 %; 50 %]	[4 %; 48 %]	[0 %; 53 %]	[8 %; 75 %]
	Average	30 %	24 %	20 %	43 %
	Relative Standard Deviation	9 %	11 %	14 %	16 %

Table 4.8 – Statistics on the coverage of locks on lock-sensitive applications for three configurations: *one node*, *max nodes*, and *opt nodes* (**all machines**). The coverage indicates how often a lock algorithm stands as the best one (or is within 5 % of the best).

53% of the applications both at *max nodes* and at the optimal number of nodes. The results for the I-20 show that the coverage of a given lock algorithm is larger than for the other machines (75% at *one node*, *max nodes* and *opt nodes*). This can be explained by the fact that this machine has less cores (and NUMA sockets) than the three others. Nonetheless, for all machines, no lock algorithm is optimal for all applications. We also observe that the average coverage is much higher at *one node* than at *max nodes*, and slightly higher at *opt nodes* than at *max nodes*. This is directly explained by the observations made in Section 4.2.2. First, at *one node*, locks have a much lower impact on applications than in other configurations and thus yield closer results, which increases their likelihood to be among the best ones. Second, at *max nodes*, all of the different locks cause, in turn, a performance collapse, which reduces their likelihood to be among the best locks. This latter phenomenon is not observed at *opt nodes*.

Is there a clear hierarchy between locks?

Figure 4.1 shows pairwise comparisons for all locks, at *max nodes* on the A-64 machine.

We observe that **there is no clear global performance hierarchy between locks**. More precisely, for most pairs of locks (*row A, col B*), there are some applications for which *A*

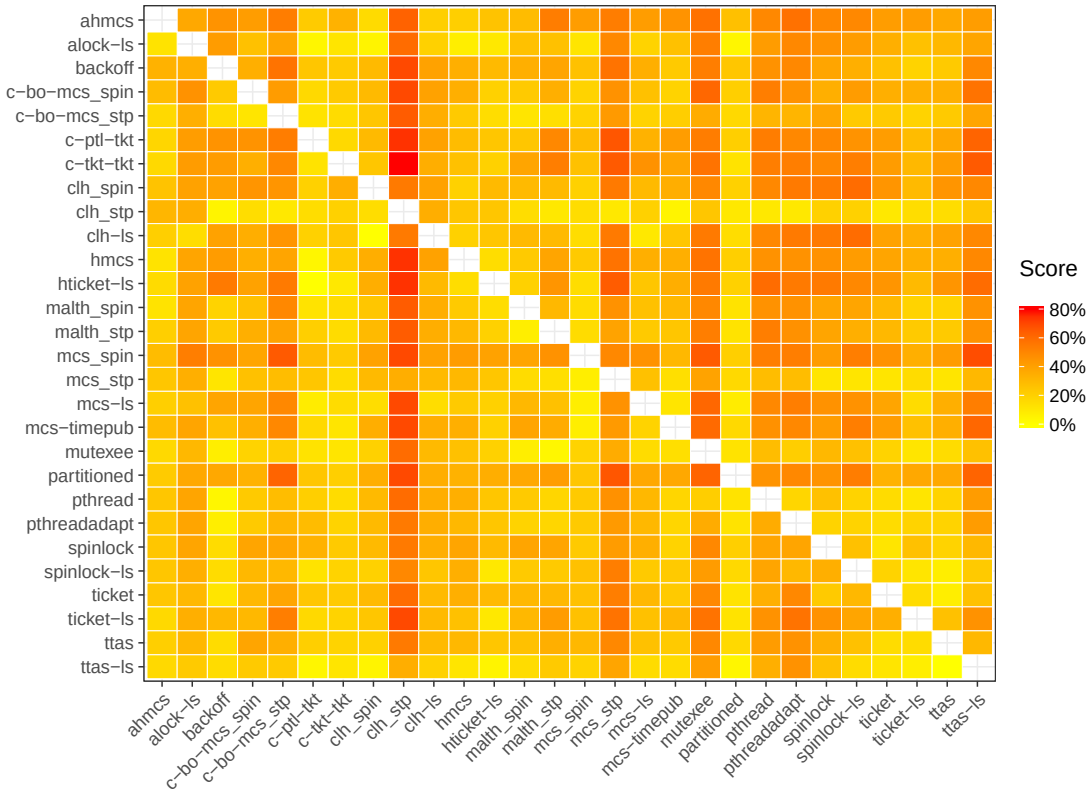


Figure 4.1 – For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B. The cell $(rowA, colB)$ color indicates the score of lock A vs. lock B, i.e., the percentage of applications for which lock A is at least 5% better than lock B. The more lock A outperforms B, the more red (dark) the cell is. For example, for roughly 40% of the applications, AHMCS performs at least 5% better than Backoff at *opt nodes*. Similarly, the figure shows that Backoff is at least 5% better than AHMCS for roughly 35% of the applications. From these two values, we can conclude that the two above-mentioned locks perform very closely for 25% of the applications. (**A-64 machine**).

Locks	Number of nodes		
	one node	*max nodes*	*opt nodes*
ahmcs	54 %	21 %	50 %
alock-ls	50 %	0 %	23 %
backoff	62 %	23 %	31 %
c-bo-mcs_spin	50 %	12 %	27 %
c-bo-mcs_stp	46 %	11 %	18 %
c-ptl-tkt	62 %	17 %	42 %
c-tkt-tkt	73 %	8 %	38 %
clh_spin	65 %	5 %	30 %
clh_stp	60 %	15 %	20 %
clh-ls	55 %	5 %	35 %
hmcs	50 %	15 %	42 %
hticket-ls	70 %	15 %	40 %
malth_spin	58 %	8 %	27 %
malth_stp	43 %	25 %	29 %
mcs_spin	65 %	19 %	38 %
mcs_stp	61 %	18 %	21 %
mcs-ls	58 %	4 %	31 %
mcs-timepub	57 %	29 %	36 %
mutexee	57 %	14 %	21 %
partitioned	71 %	12 %	42 %
pthread	43 %	21 %	21 %
pthreadadapt	39 %	25 %	21 %
spinlock	73 %	23 %	23 %
spinlock-ls	62 %	15 %	31 %
ticket	69 %	15 %	35 %
ticket-ls	65 %	12 %	31 %
ttas	73 %	12 %	31 %
ttas-ls	54 %	0 %	15 %

Table 4.9 – For each lock, fraction of the lock-sensitive applications for which the lock yields the best performance for three configurations: *one node*, *max nodes* and *opt nodes* (**A-64 machine**).

is better than *B*, or vice-versa (Figure 4.1). The only marginal exceptions are the cells having 0% for value. This corresponds to pairs of locks (*row A*, *col B*) for which *A* never yields better performance than *B*. The results at *max nodes* (available in the companion technical report [50]) exhibit similar trends as the ones at *opt nodes*. Besides, we make the same observations (both at *opt nodes* and *max nodes*) on the A-48, the I-48 machines and the I-20 (see the companion technical report [50]).

	A-64		A-48		I-48		I-20	
Lock	Max	Opt	Max	Opt	Max	Opt	Max	Opt
ahmcs	58 %	17 %	55 %	50 %	44 %	44 %	46 %	38 %
alock-ls	96 %	46 %	70 %	50 %	53 %	47 %	29 %	29 %
backoff	62 %	38 %	38 %	43 %	53 %	37 %	43 %	36 %
c-bo-mcs_spin	65 %	42 %	62 %	62 %	47 %	32 %	29 %	29 %
c-bo-mcs_stp	82 %	46 %	87 %	83 %	85 %	60 %	80 %	73 %
c-ptl-tkt	58 %	25 %	58 %	53 %	47 %	29 %	29 %	21 %
c-tkt-tkt	58 %	35 %	67 %	52 %	37 %	32 %	14 %	14 %
clh_spin	85 %	35 %	60 %	53 %	86 %	71 %	50 %	50 %
clh_stp	85 %	65 %	93 %	93 %	93 %	93 %	92 %	92 %
clh-ls	85 %	35 %	67 %	60 %	79 %	79 %	58 %	58 %
hmcs	54 %	31 %	38 %	38 %	42 %	32 %	14 %	14 %
hticket-ls	65 %	40 %	50 %	56 %	50 %	36 %	17 %	17 %
malth_spin	73 %	46 %	62 %	52 %	63 %	63 %	43 %	43 %
malth_stp	57 %	46 %	74 %	74 %	60 %	60 %	33 %	33 %
mcs_spin	77 %	31 %	67 %	43 %	53 %	47 %	29 %	29 %
mcs_stp	75 %	57 %	78 %	74 %	75 %	75 %	80 %	73 %
mcs-ls	81 %	42 %	67 %	48 %	58 %	53 %	29 %	29 %
mcs-timepub	50 %	29 %	61 %	48 %	55 %	50 %	47 %	40 %
mutexee	68 %	57 %	74 %	61 %	70 %	60 %	40 %	40 %
partitioned	79 %	33 %	68 %	63 %	71 %	53 %	36 %	36 %
pthread	68 %	61 %	78 %	74 %	70 %	70 %	53 %	47 %
pthreadadapt	68 %	54 %	70 %	70 %	75 %	60 %	53 %	40 %
spinlock	69 %	50 %	81 %	67 %	74 %	63 %	64 %	50 %
spinlock-ls	77 %	46 %	81 %	57 %	74 %	63 %	57 %	36 %
ticket	77 %	50 %	90 %	62 %	89 %	79 %	43 %	36 %
ticket-ls	69 %	42 %	76 %	57 %	68 %	53 %	36 %	29 %
ttas	69 %	38 %	81 %	52 %	74 %	58 %	43 %	36 %
ttas-ls	92 %	54 %	90 %	60 %	84 %	68 %	71 %	57 %

Table 4.10 – For each lock, at *max nodes* and at *opt nodes*, fraction of the lock-sensitive applications for which the lock is harmful, i.e., the performance gain brought by the best lock with respect to the given lock is greater than 15 % (**all machines**).

Are all locks potentially harmful?

Our goal is to determine, for each lock, if there are applications for which it yields substantially lower performance than other locks and to quantify the magnitude of such performance gaps. Table 4.10 displays, for each machine, the fraction of applications that are significantly hurt by a given lock at *max nodes* and at *opt nodes*. Table 4.11 shows the detailed results at *max nodes* for the A-64 machine (results for all machines in the companion technical report [50]).

On the four machines, we observe that, **both at *max nodes* and at the optimal number**

of nodes, all locks are potentially harmful, yielding sub-optimal performance for a significant number of applications (Table 4.10). We also notice that locks are significantly less harmful at *opt nodes* than at *max nodes*. This is explained by the fact that several of the locks create performance collapse at *max nodes*, which does not occur at *opt nodes*. Moreover, we observe that, for each lock, the performance gap to the best lock can be significant (Table 4.10).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	609	5	142	141	29	13	588	590	988	150	134	135	133	131	133	127	84	0	16	8	4	4	3	6	4	5	591
facesim	298	694	338	115	85	258	231	687	52	700	219	196	531	40	710	52	771	0	23	685	56	44	572	117	719	337	147	224
ferret	310	270	8	50	0	239	229	312	0	252	274	277	209	0	317	0	349	4	1	314	0	1	10	4	308	93	8	8
fluidanimate	-	284	0	53	71	30	12	-	-	-	65	-	31	87	35	86	44	44	76	9	1	7	21	2	13	10	4	187
fmm	41	37	22	16	27	29	15	39	33	38	35	32	21	18	2	0	32	0	25	20	25	23	2	27	19	28	22	32
kyotocabinet	9	91	29	0	171	24	28	33	484	43	12	19	22	22	28	474	38	52	397	45	409	276	2k	2k	214	89	589	334
linear_regression	17	89	10	34	198	14	17	17	64	56	13	29	8	0	22	58	54	4	16	47	16	2	85	31	33	12	34	38
matrix_multiply	9	78	3	14	5	27	54	9	3	12	608	59	5	3	28	2	168	0	6	44	3	3	5	59	3	59	4	55
memcached-new	0	20	38	70	871	-	10	-	-	-	0	-	35	81	20	582	31	17	53	-	103	193	1k	764	221	80	331	110
memcached-old	117	54	63	0	1	-	14	-	-	-	6	-	289	307	149	192	264	175	108	-	209	225	305	45	216	223	33	74
mysqld	-	-	-	-	53	-	-	-	-	-	-	-	-	0	-	7	-	173	10	-	97	102	-	-	-	-	-	-
ocean_cp	28	17	38	32	47	24	21	41	38	19	31	14	23	27	43	32	30	0	13	31	11	19	129	54	55	7	23	34
ocean_ncp	24	17	23	24	31	8	12	27	25	18	13	3	16	19	33	34	11	8	5	26	28	24	113	34	22	0	11	31
pca	51	57	26	27	346	43	47	49	221	53	51	41	30	0	57	229	39	25	124	31	121	45	266	107	104	26	200	128
pca_ll	64	52	0	8	713	56	58	66	379	31	60	35	14	10	61	369	27	26	165	41	166	51	522	331	116	23	273	193
radiosity	13	12	4	4	38	8	5	9	0	13	7	8	8	12	0	90	9	0	42	0	0	0	1	54	0	19	34	61
radiosity_ll	0	41	43	26	1k	37	47	31	1k	49	0	17	68	67	26	2k	57	60	535	76	569	262	2k	1k	585	200	1k	802
s_raytrace	0	33	65	37	2k	29	39	46	1k	37	16	31	26	64	0	1k	16	13	282	103	230	122	714	252	412	145	661	514
s_raytrace_ll	10	20	42	39	645	11	13	0	2k	5	41	19	37	21	0	2k	10	14	284	56	201	67	1k	744	554	172	1k	916
sqlite	-	-	-	-	405	-	-	-	-	-	-	-	-	-	-	591	-	2k	375	-	336	181	-	-	-	-	-	-
ssl_proxy	0	17	48	5	790	11	26	16	879	27	159	15	16	35	41	900	48	29	319	36	293	153	1k	447	271	89	594	499
streamcluster	49	21	137	43	195	0	15	-	-	-	0	-	219	1k	121	1k	188	13	32	8	95	142	527	129	302	181	215	206
streamcluster_ll	65	20	188	58	277	0	21	-	-	-	15	-	262	1k	144	1k	228	28	44	27	80	120	549	196	321	177	232	189
upscaledb	8	18	30	9	110	16	12	16	281	19	8	14	0	5	17	267	21	34	107	25	215	109	747	496	106	49	226	318
vips	48	38	6	184	21	95	73	-	-	-	84	-	196	0	28	1	28	3	2	33	0	2	3	6	16	10	8	14
volrend	2	27	36	19	72	0	7	26	58	17	0	2	18	63	27	47	22	25	80	25	78	105	162	87	25	15	40	48
water_nsquared	94	48	4	6	10	8	2	35	35	58	14	11	7	6	3	2	9	7	7	4	6	7	0	6	4	6	5	37
water_spatial	97	48	1	9	6	3	3	40	39	63	8	4	8	5	9	9	5	10	3	1	0	0	2	1	1	0	0	40

Table 4.11 – For each lock-sensitive application, at *max nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the performance gain is greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases (**A-64 machine**).

4.2.3 Additional observations

Impact of the number of nodes

Table 4.12 shows, for each application on the A-64 machine, the number of pairwise changes in the lock performance hierarchy when the number of nodes is modified. We observe that, **for all applications, the lock performance hierarchy changes significantly according to the chosen number of nodes**. Moreover, we observe the same trends on the A-48, I-48 and I-20 machines (see the companion technical report [50]).

Impact of the machine

We look at the number of pairwise lock inversions observed between the machines (both at *max nodes* and at *opt nodes*). For a given application at a given node configuration, we check whether two locks are in the same order or not on the target machines. We observe that **the lock performance hierarchy changes significantly according to the chosen machine**. Interestingly, we observe that there is approximately the same number of inversions between each pair of machines, roughly 30% for all configurations. The detailed results for each pair of machines are available inside the companion technical report [50].

A note on Pthread locks

The various results presented in this Section show that the current Linux **Pthread locks perform reasonably well (i.e., are among the best locks) for a significant share of the studied applications**, thus providing a different insight than recent results, which were mostly based on synthetic workloads [33]. Beyond the changes of workloads, these differences could also be explained by the continuous refinement of the Linux Pthread implementation. It is nevertheless important to note that on each machine, some locks stand out as the best ones for a higher fraction of the applications than Pthread locks. Finally, we note that Pthread locks and PthreadAdapt locks exhibit similar performance.

Impact of thread pinning

As explained in Section 4.1, all the previously-described experiments were run without any restriction on the placement of threads (i.e., a thread might be scheduled on any core of the machine), leaving the corresponding decisions to the Linux scheduler. However, in order to better control cores allocation and improve locality, some developers and system administrators use pinning to explicitly restrict the placement of each thread to one or several core(s). The impact of thread pinning can vary greatly accord-

Applications	% of pairwise changes between configurations			
	1/2	2/4	4/8	1/2/4/8
dedup	11 %	4 %	13 %	18 %
facesim	17 %	43 %	85 %	97 %
ferret	0 %	71 %	25 %	85 %
fluidanimate	7 %	6 %	23 %	30 %
fmm	37 %	13 %	19 %	50 %
kyotocabinet	15 %	12 %	14 %	30 %
linear_regression	48 %	46 %	47 %	88 %
matrix_multiply	41 %	26 %	45 %	72 %
memcached-new	53 %	18 %	0 %	64 %
memcached-old	77 %	73 %	0 %	95 %
mysqld	24 %	29 %	14 %	38 %
ocean_cp	46 %	45 %	69 %	94 %
ocean_ncp	54 %	51 %	56 %	90 %
pca	41 %	50 %	29 %	92 %
pca_ll	31 %	40 %	47 %	94 %
radiosity	10 %	50 %	51 %	81 %
radiosity_ll	67 %	26 %	15 %	90 %
s_raytrace	7 %	69 %	28 %	96 %
s_raytrace_ll	4 %	87 %	20 %	97 %
sqlite	29 %	19 %	45 %	81 %
ssl_proxy	62 %	13 %	21 %	77 %
streamcluster	66 %	29 %	32 %	93 %
streamcluster_ll	61 %	34 %	30 %	95 %
upscaledb	41 %	17 %	14 %	54 %
vips	1 %	3 %	83 %	83 %
volrend	19 %	28 %	39 %	85 %
water_nsquared	20 %	21 %	13 %	49 %
water_spatial	6 %	9 %	12 %	26 %

Table 4.12 – For each lock-sensitive application, percentage of pairwise changes in the lock performance hierarchy when changing the number of nodes. For example, in the case of the facesim application, there are 17 % of the pairwise performance comparisons between locks that change when moving from a 1-node configuration to a 2-node configuration. Similarly, there are 97 % of pairwise comparisons that change at least once when considering the 1-node, 2-node, 4-node and 8-node configurations. (**A-64 machine**).

ing to workloads and can yield both positive and negative effects [33, 74]. In order to assess the generality of our observations, we also performed the complete set of experiments on the A-64 machine with an alternative configuration in which each thread is pinned to a given node, leaving the scheduler free to place the thread among the cores of the node. Note that for an experiment with a N-node configuration, the complete application runs on exactly the first N nodes of the machine. We chose thread-to-node pinning rather than thread-to-core pinning because we observed that the former generally provided better performance for our studied applications, especially the ones using more threads than cores. The detailed results of our experiments with thread-to-node pinning are available in the companion technical report [50]. Overall, we observe that **all the conclusions presented in this Chapter still hold with per-node thread pinning**.

Impact of BIOS configuration

The experiments presented in this Section were all ran with the BIOS configured in performance mode, for all machines. In performance mode: (i) processor throttling is turned off, so that all cores always run at full speed (i.e., maximum available frequency, Intel Turbo Boost / AMD Turbo Core deactivated), and (ii) idle power saving processor C-states are deactivated, thus cores are always immediately available to execute threads (i.e., they never need to be resumed from a low-power mode). In addition, for the I-48 and I-20 machines, we also executed the throughput experiments with the BIOS configured in energy-saving move. In such a configuration, processor throttling and idle power saving C-states are activated, letting the hardware and the kernel manage the processors' state, aiming at reducing power consumption. We observe quantitative throughput differences between the two configurations. However, changing the BIOS configuration does not only affect lock performance but also application performance. Yet, a full study of the impact of the BIOS configuration modes on the performance of applications falls out of the scope of this thesis. Nonetheless, we observe that **all the conclusions presented in this Chapter still hold when the BIOS is configured in energy-saving move**.

4.2.4 Effects of the lock choice on application performance

The results of our study have several implications for both the software developers and the lock algorithm research community. First, we observe that **the choice of a lock algorithm should not be hardwired into the code of applications**: applications should always use standard synchronization APIs (e.g., the POSIX Pthread API), so that one can easily interpose the implementation of the lock algorithm.

Second, the Pthread library should **not provide only one lock algorithm (i.e., the Pthread lock algorithm) to software developers** as it is currently the case. It is a "good

generic solution"; still **Pthread locks certainly do not bring the best performance for every application**.

Third, the research community should perform **further research on optimized lock algorithms**. Specifically, there is a need for **dynamic approaches** to lock algorithms that automatically adapt to the running workload and its environment (e.g., the machine, the collocated workloads). Besides, previous work only focused on the lock/unlock API, while we observe that applications also stress trylocks, barriers and condition variables, thus future research needs to consider **complete locking APIs** (more details in Section 4.5). Finally, metrics other than throughput are becoming more and more important, and as a consequence, when designing a new lock algorithm, researchers should not only consider throughput, but **all performance metrics**, including latency and energy efficiency (as we will see in details in Sections 4.3 and 4.4).

4.3 Study of lock energy efficiency

In this Section, we perform experiments on the I-48 and I-20 machines in order to evaluate the energy efficiency of the different lock algorithms. In Sections 4.2 and 4.4, we present the results for throughput and tail latency, respectively. We are interested in energy efficiency as defined by Falsafi et al. [42]: energy efficiency represents the amount of work produced for a fixed amount of energy and can be defined as *throughout per power* (abbreviated TPP thereafter, in $\frac{\#operations/second}{watt} = \frac{\#operations/second}{joule/second} = \#operations/joule$). Higher TPP represents better energy efficiency. As explained in Section 4.1, we use Intel's RAPL facility to measure the energy consumption of several components: cores, chip package and DRAM. Moreover, the BIOS for the energy experiment is configured in energy saving mode.

This Section is structured as follows. First, Section 4.3.1 discusses the results of the energy-efficiency study. We also discuss the similarities and differences between performance and energy-efficiency observations drawn from the study. Next, Section 4.3.2 discusses and validates the POLY conjecture previously introduced by Falsafi et al. [42], stating that energy efficiency and throughput go hand in hand with locks.

4.3.1 Energy-efficiency lock behavior

For the sake of brevity, we do not describe all the individual results for energy efficiency, available in the companion technical report [50]. Overall, we observe that **all the conclusions presented in Section 4.2 about throughput still hold with energy efficiency**. More precisely, we observe that: (i) 50% of the applications are lock-sensitive with respect to energy efficiency, (ii) the optimized number of nodes for many applications is lower than the max number of nodes, (iii) the energy-efficiency gap is often large between different kinds of locks, (iv) the impact of locks on lock-sensitive applications is moderate at *one node*, and very high at both *opt nodes* and *max nodes*, (v) no lock is among one of the bests for more than 83% of the lock-sensitive applications at *one node* and for more than 61% both at *max nodes* and *opt nodes*, (vi) there is no clear global performance hierarchy among locks, (vii) all locks are potentially harmful, both at *max nodes* and *opt nodes*, yielding sub-optimal energy efficiency for a significant number of applications, (viii) the lock performance hierarchy changes significantly according to the chosen number of nodes. We observe, similarly to performance, that the I-20 exhibits less pronounced trends than the I-48 machine. Compared to the four twelve-core NUMA sockets of the I-48 machine, the I-20 machine only has twenty cores, divided into two NUMA sockets. As a consequence, the *max node* configuration for the I-20 uses half the threads than the I-48. Thus, some bottlenecks leading to collapse when using a high number of threads are not observable on the smaller I-20 machine.

	I-48	I-20
$\geq +5\,\%$	64 %	38 %
$\leq -5\,\%$	4 %	9 %
between $-5\,\%$ and $+5\,\%$	32 %	53 %

Table 4.13 – Percentage of lock-sensitive applications for which the energy-efficiency gain of *opt nodes* over *max nodes* is at least 5% higher than the performance gain, at least 5% lower than the performance gain or between +5% and -5% of the performance gain (**I-48 and I-20 machines**.)

	I-48	I-20
lower *opt nodes*	25 %	11 %
same *opt nodes*	74 %	87 %
higher *opt nodes*	1 %	2 %

Table 4.14 – Percentage of lock-sensitive applications for which *opt nodes* is lower, the same or higher for energy efficiency w.r.t. performance. We use a 5% tolerance margin, i.e., if the application performance at *opt nodes* is $N1$ and the energy efficiency at *opt nodes* is $N2$, and $N1 \neq N2$, we look the performance at $N2$ and the energy efficiency at $N1$, and if the performance or the energy-efficiency difference is lower than 5%, we consider that the application's *opt nodes* is the same for performance and energy efficiency. (**I-48 and I-20 machines**).

We observe similar general trends between performance and energy efficiency. However, looking at the detailed results and comparing them allows us to discover new interesting facts. The following observations are made from the results on the I-48 machine. The results for the I-20 machine are discussed at the end of the Section.

We first observe that the set of lock-sensitive applications for throughput is almost the same as the set with respect to energy efficiency. In other words, changing the lock algorithm affects the throughput if and only if it affects the energy efficiency. This insight simplifies the monitoring/profiling and optimization process of such applications.

Table 4.13 shows the gain difference of *opt nodes* over *max nodes* between energy efficiency and throughput. **The gain between *opt nodes* and *max nodes* for energy efficiency is generally higher than the one for throughput.** We observe that on the I-48, the gain for energy efficiency is higher for at least half of the lock-sensitive applications, and the same for 32% of the lock-sensitive applications. Intuitively, for energy efficiency, wasting resources while waiting behind locks costs both in terms of throughput and wasted energy.

Table 4.14 shows the percentage of lock-sensitive applications where *opt nodes* is lower, the same or higher while considering energy efficiency w.r.t. throughput. On the I-

48, 25% of the lock-sensitive applications collapse at a lower number of nodes with energy efficiency than with throughput, 74% at the same number of nodes, and 1% at a higher number of nodes. We can conclude that, **when throughput collapses, energy efficiency generally starts collapsing at a similar degree of parallelism.**

The results on the I-20 machine are similar (available in the companion technical report [50]).

4.3.2 POLY

The POLY conjecture introduced by Falsafi et al. [42] states that "energy efficiency and throughput go hand in hand in the context of lock algorithms". More precisely, POLY suggests that "locks can be optimized to improve energy efficiency without degrading throughput", and that "[the insights from] prior throughput-oriented research on lock algorithms can be applied almost as-is in the design of energy-efficiency locks". The POLY conjecture could explain why we observe similar trends between our performance and energy-efficiency results. In this Section, our goal is to test this conjecture on a large number of lock algorithms and applications (the initial paper about POLY considered 3 lock algorithms and 6 applications).

Figure 4.2 shows the correlation between performance and energy efficiency. Figures 4.3 and 4.4 show the detailed results respectively at *one node* and *max nodes* for each lock-sensitive application (results at *one node* and *max nodes* for the I-20 machines are available in the companion technical report [50]). The energy efficiency (in TPP – throughput per power, see Section 4.3) and the throughput are normalized w.r.t. the best performing (resp. energy-efficient) lock for each *(machine, application, type, node)* configuration. Both at *one node* and *max nodes*, most data points fall on, or very close to a linear regression between the two variables (the blue diagonal line). Even at *max nodes* where locks throughput *and* energy efficiency are more disparate (e.g., for applications like linear_regression or s_raytrace), i.e., the choice of a lock significantly affects application throughput/energy efficiency, there is clearly a linear correlation between the two.

Based on Figure 4.2, Malth_STP and (to a lesser extent) MCS-TimePub are outliers. These two algorithms use complex load-control algorithms: (i) Malth_STP parks a subset of the threads, while the others always spin for a few cycles before acquiring the lock ; (ii) MCS-TimePub allows spinning threads to bypass parked ones). The "exotic" behaviors of these locks most probably explain why the throughput and the energy consumption are not so well correlated with respect to other locks. Besides, on Figure 4.3, MySQL and (to a lesser extent) SQLite are outliers. These are the only two applications launching thousand of threads, stressing heavily the Linux scheduler. We conjecture that the overhead of context switches (due to both lock parking and thread

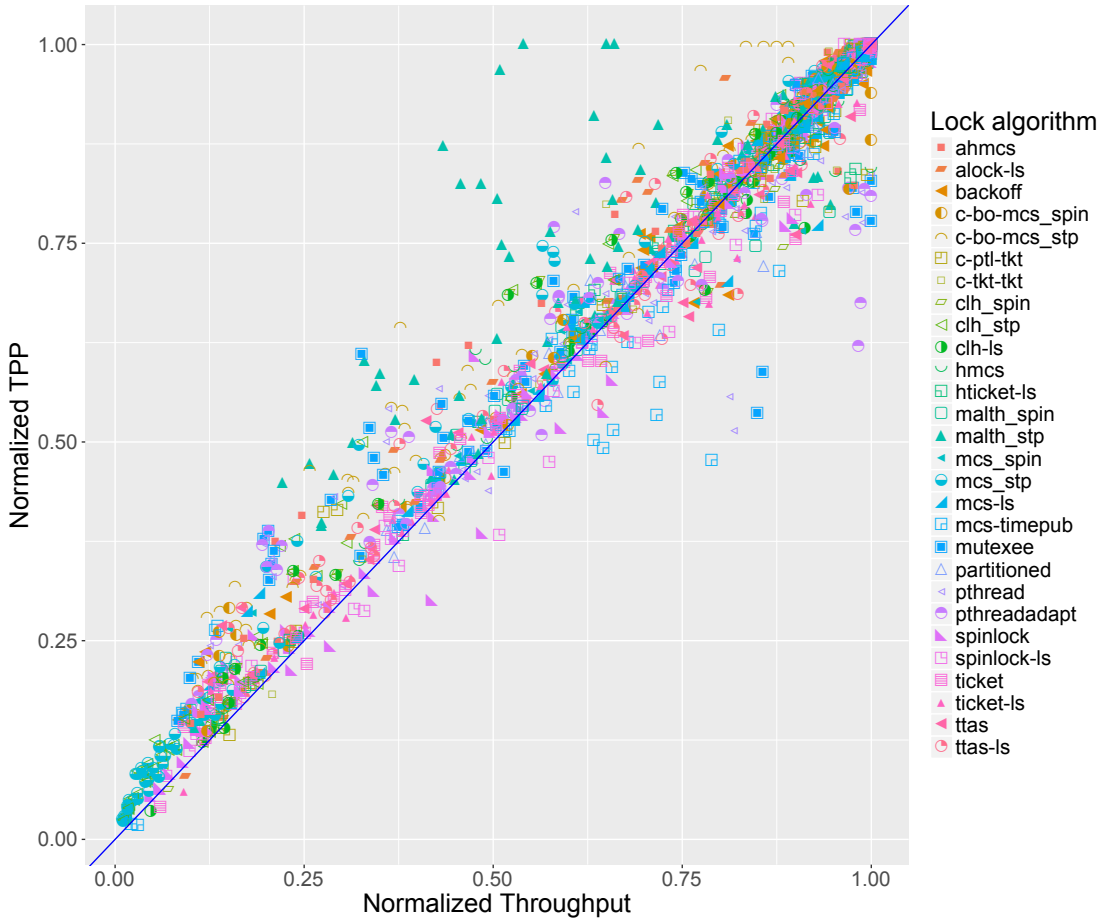


Figure 4.2 – Correlation of throughput with energy efficiency (TPP) on various lock-sensitive applications with various lock algorithms and various contention levels (**all machines**).

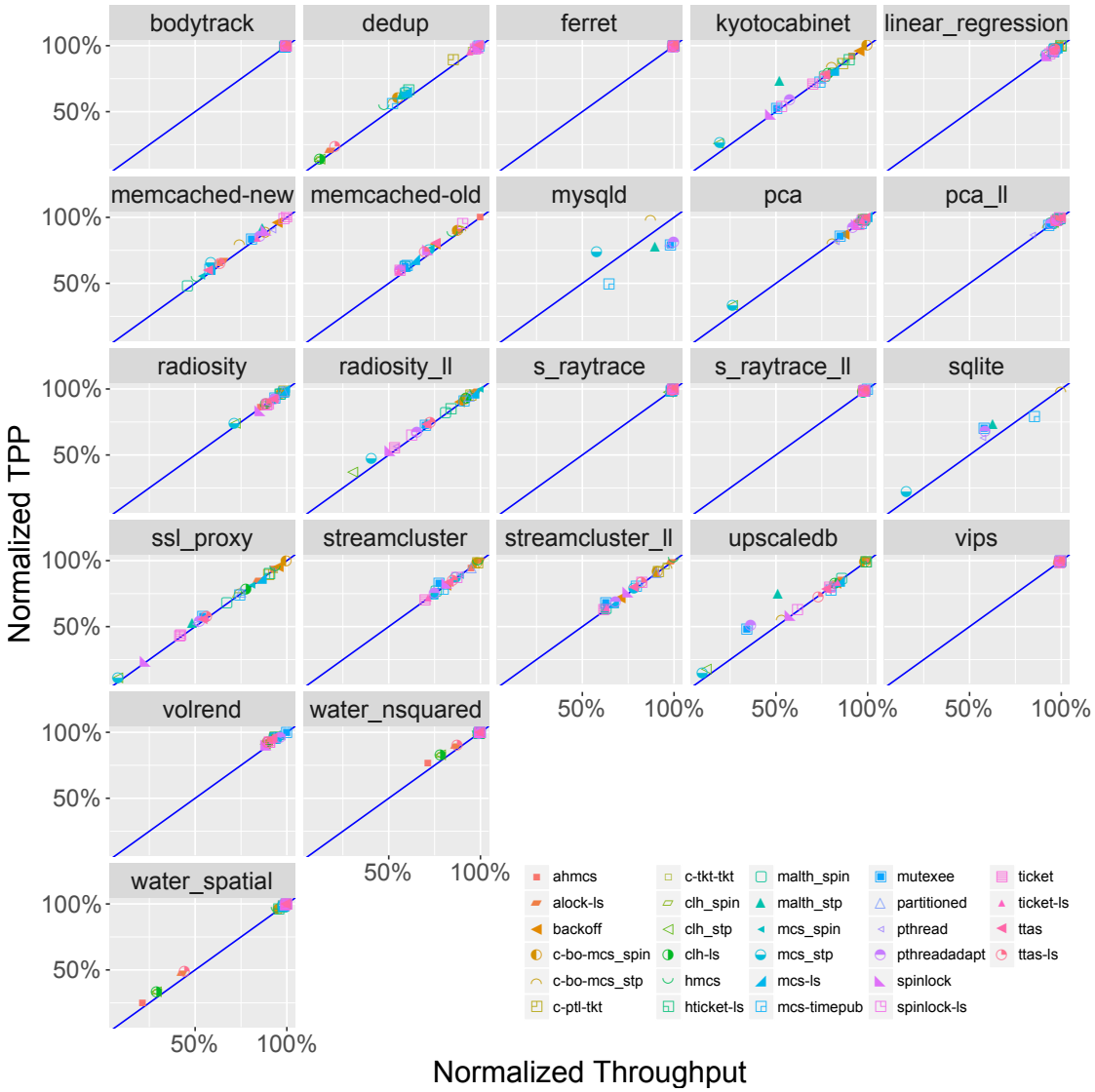


Figure 4.3 – Correlation of throughput with energy efficiency (TPP) on various lock-sensitive applications at *one node* for the different lock algorithms (**I-48 machine**).

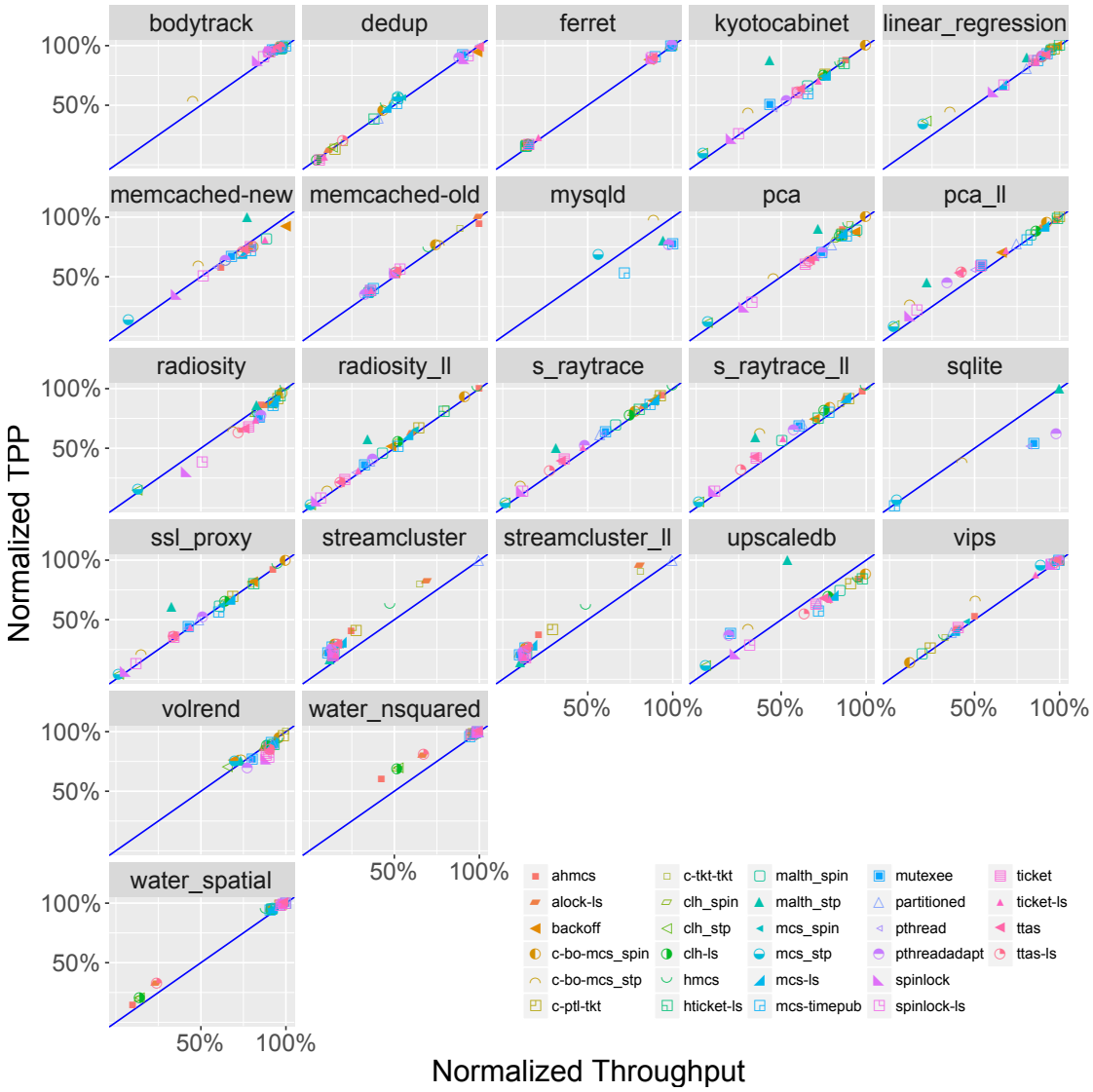


Figure 4.4 – Correlation of throughput with energy efficiency (TPP) on various lock-sensitive applications at *max nodes* for the different lock algorithms (**I-48 machine**).

	I-20	I-48
bodytrack	-	0.98
dedup	1.00	1.00
ferret	0.98	0.96
kyotocabinet	0.89	0.88
linear_regression	0.96	0.98
memcached-new	0.99	0.91
memcached-old	1.00	0.97
mysqld	-	0.55
pca	0.97	0.96
pca_ll	0.95	0.91
radiosity	0.98	0.98
radiosity_ll	0.89	0.94
s_raytrace	0.97	0.95
s_raytrace_ll	0.94	0.98
sqlite	0.98	0.94
ssl_proxy	-	0.95
streamcluster	0.97	0.99
streamcluster_ll	0.91	0.98
upscaledb	0.91	0.87
vips	0.97	0.96
volrend	-	0.96
water_nsquared	1.00	0.99
water_spatial	0.99	1.00

Table 4.15 – Pearson correlation coefficient between throughput and TPP for all lock-sensitive applications. Dashes mark applications that are not lock-sensitive (or not evaluated due to a lack of high-throughput network connectivity, see Section 4.1.3) on the I-20 machine. (**I-48 and I-20 machines**).

preemption) slightly breaks the correlation between throughput and energy.

To quantitatively assess the correlation between energy efficiency and performance, we compute the Pearson correlation coefficient (PCC). The PCC is the value of the slope of a linear regression between two variables: the closer to 1, the greater the correlation between the variables. Intuitively, it quantifies the dispersion of the different configurations around the diagonal blue line. Table 4.15 shows the PCC on I-48 and I-20 for all the studied lock-sensitive applications. We observe that except MySQL that has a low PCC (0.55), all other configurations have a PCC at least equal to 0.87, which indicates a strong correlation between the performance and energy efficiency. More generally, **the PCC across all configurations (3.1k experiments) is 0.95**, an almost perfect correlation coefficient.

MySQL, upscaledb, Kyoto Cabinet and radiosity_ll have a PCC lower than 0.9. We

observe that these four applications are highly contended. Looking at the detailed results, we observe that lock algorithms that use a parking waiting policy generally have a lower performance-to-energy-efficiency ratio (*PtE ratio* thereafter) than spinning algorithms. For example, for MySQL, algorithms using a fixed threshold for the spinning loop part of the spin-then-park waiting policy (e.g., C-BO-MCS_STP with a PtE of 0.89), have a lower PtE than algorithms that do adaptive spin-then-park (e.g., Mutexee with a PtE of 1.28), and even lower than algorithms that do spinning (e.g., MCS-TimePub[7] with a PtE of 1.34). Intuitively, these results are expected, because at high levels of contention, parking locks can save energy compared to spinning, but spinning might still result in higher throughput [42].

To conclude, we can state that **the POLY conjecture holds on our experimental testbeds**, i.e., for lock algorithms, energy efficiency and throughput go hand in hand.

[7]MySQL is highly multi-threaded (hundreds of threads), and, as a consequence, MCS-TimePub is the only spinning lock algorithm that we study because it has a preemption tolerance mechanism. With other spinning algorithms the application throughput drops close to zero.

4.4 Study of lock tail latency

In this Section, we are interested in the effect of lock algorithms on the application quality of service (QoS). More precisely, the QoS metric that we consider is the application tail latency, here defined as the 99th percentile of client response time. Note that in Sections 4.2 and 4.3 we discussed the results for throughput and energy efficiency, respectively. Understanding the relationship between throughput and tail latency allows us to understand, for example, if some lock properties (i.e., the fairness of FIFO locks) that improve the tail latency of lock acquisitions indeed improve the application tail latency. This analysis also enables us to understand which locks to choose to improve the tail latency of an application, sometimes at the (controlled) expense of throughput.

To perform this analysis, we capture the 99th percentile of the client response time on the A-64 machine for the seven server applications among the lock-sensitive applications that we have studied: Kyoto Cabinet, Memcached-new, Memcached-old, MySQL, SQLite, SSL Proxy, upscaledb. We further captured throughput and energy-efficiency metrics. Note that, as we discuss in Section 4.3.2, throughput and energy efficiency are correlated, thus we do not clutter the plots with energy-efficiency information and only show throughput. We have also performed the same experiments on the I-48 machine (our largest Intel multicore machine) and made similar observations as the ones described hereafter for the A-64 machine.

Figure 4.5 reports for each application and each lock algorithm at *opt nodes* the normalized (w.r.t. Pthread) 99th tail latency, as well as the normalized (w.r.t. Pthread) execution time (black squares). The results at *one node* and *max nodes* are available in the companion technical report [50]). Locks are sorted by increasing tail latency. Note that we plot execution time (rather than throughput) so that "lower is better" for both displayed metrics (latency and execution time). However, in the text we talk about throughput (as the inverse of the execution time) for homogeneity with the other Sections.

4.4.1 How does tail latency behave when locks suffer from high levels of contention?

At *max nodes*, the maximum tail latency is generally higher than at *opt nodes* and *one node*. For example, for Kyoto Cabinet, at *max nodes*, the tail latency of CLH_STP is 5× higher than Pthread, while it is of roughly 1.6× higher than Pthread at *one node* and *opt nodes*. The tail latency skyrockets at *max nodes*: locks suffer from extreme levels of contention and threads wait for a long time to acquire locks. On average, when increasing the number of threads (from *one node* to *max nodes*), the request execution time increases 3.3× and the tail latency increases 22.9×. Similarly, from *opt nodes* to *max*

nodes, the request execution time increases 3.4× and the tail latency increases 21.0×. The experiments with a single thread for all the studied applications except MySQL and SQLite[8] are available in the companion technical report [50]. Overall, we found that, on the studied applications with a single-threaded configuration, the choice of a lock has very little effect on the throughput or the tail latency of the application.

4.4.2 Do fair lock algorithms improve the application tail latency?

On the one hand, FIFO locks (cf. Section 2.1.2) promise fairness among threads acquiring a lock. On the other hand, unfair locks might increase tail latencies by letting some threads wait for long durations before acquiring the lock. Interestingly, we observe that fairness affects the tail latency for only two applications: Kyoto Cabinet and upscaledb. For them, we observe low tail latency with almost all FIFO locks. Moreover, all hierarchical locks, which by design do not strictly impose fairness, exhibit roughly the same tail latencies, which are higher than the tail latencies of FIFO locks. Still, for the four other studied applications, we do not observe a correlation between lock fairness and application tail latency.

The main distinction among the group of applications where fair lock algorithms improve the application tail latency and where they do not is how an operation (e.g., a request) uses locks. If an operation is mainly implemented as a single critical section, then lock properties that affect lock acquisition tail latencies and throughput also affect the application, which is the case for upscaledb and Kyoto Cabinet. For example, for upscaledb, at *opt nodes*, we measured that 90 % of the response time is consumed either while waiting for a single global lock, or inside the critical sections. On the contrary, for Memcached-new, which is one of the applications where fair lock algorithms do not necessarily improve the application tail latency, roughly 45 % of the response time is spent either waiting for locks or inside critical sections (55 % of the response time is spent in parallel code sections). Besides, Memcached-new uses more than one lock while processing a request, and two different threads might use different locks to process different requests: locks are thus less stressed. To summarize, we observe that, on the seven studied applications, lock properties affect application tail latency only for applications where an operation is mainly implemented as a single critical section.

4.4.3 Do lock tail latencies affect application throughput?

Some lock algorithms explicitly try to trade fairness for higher throughput. For example, hierarchical locks prefer to give a lock to a thread on the same NUMA node than to a thread executing on another node. Interestingly, in practice, we observe that

[8] Running MySQL or SQLite with a single thread totally changes the workload, thus numbers cannot be compared with other configurations with more threads.

this property, which directly affects tail latency and throughput of lock acquisitions, effectively affects the application tail latency and throughput for only two applications: upscaledb and Kyoto Cabinet. For these applications, we generally observe that hierarchical locks lead to higher tail latency and higher throughput. For example, for upscaledb at *opt nodes*, increasing the tail latency from 100μs to 1000μs increases the throughput by 26 % (using MCS vs. HMCS). Using Ticket and C-TKT-TKT on Kyoto Cabinet, at *opt nodes*, increasing the tail latency by 3×, leading to a 33 % throughput increase. At *max nodes*, Mutexee exhibits 80 % higher tail latency than Pthread, but improves throughput by 60 %. Applications where the tail latency is affected by the lock fairness property of some locks (§4.4.2) are the same applications that are affected by the fairness/throughput tradeoff property.

For the other applications where an operation is "large", i.e., an operation consists of many critical sections and/or whose critical sections are protected by different lock instances accessed by different threads, we observe that lower application tail latency is correlated with higher application throughput. In such cases, the tail latencies of individual locks are in the scale of hundreds of μs and do not have a significant weight in the operation latencies. Thus, the lock tail latency does not directly influence the application tail latency and throughput.

Among the 7 server applications for which we studied tail latency, we obtained unexpected results for Memcached-old. This application is known to suffer from extreme levels of contention (see Section 4.5): the main bottleneck is a single global lock serializing most requests. One might expect that lock properties should directly affect the application throughput and tail latency. However, Memcached-old uses the trylock operation to acquire a lock. Interestingly, most of the lock algorithms have been designed to optimize the lock/unlock operation, not the trylock one, and in practice, there is no such thing as a "fair trylock", even for locks that promise FIFO lock acquisitions.

4.4.4 Implications

Contrary to throughput (see Section 4.2.2), studying tail latency allows us to draw simpler conclusions, as the results are more stable across applications and machines. We observe two groups of applications that behave differently regarding tail latency.

If an operation is mostly implemented as a single critical section, then **lock properties that affect lock acquisition tail latency and throughput affect application tail latency and throughput**. In practice, **low tail latency can be achieved with FIFO locks**. If **throughput is more important** and a developer is **inclined to trade tail latency for throughput, hierarchical locks are a good choice.**

In contrast, for applications **with "larger" operations that consist of many critical sections and/or the critical sections are protected by different lock instances accessed**

by different threads, the tail latency of locks does not necessarily affect the application tail latency. For such applications, a developer should choose a lock that best improves the application throughput: the tail latency improvements will follow.

Interestingly, we observe in our set of studied applications that software developers use the trylock operation to implement busy waiting, while the original operation is designed to allow a developer to write a fallback code if the locking attempt fails. Because the trylock is only a one-shot attempt to acquire a lock, there is actually no lock algorithm that provides a fair trylock. We believe that developers use trylocks this way because the default Pthread lock operation is blocking: a developer knows when a critical section is short, and thus would like to avoid the overhead of a thread blocking if the lock is unavailable. Pragmatically, the trylock operation should not be used this way, but this demonstrates the need to extend the Pthread lock API with a **lock operation informing the lock algorithm that a thread should busy wait and not block, e.g., `pthread_mutex_busylock`**[9].

[9]There is a function named `pthread_spin_lock` that allows spining on a lock instance, but this function only accepts a `pthread_spinlock_t` lock, not a `pthread_mutex_t` lock. Thus, there is no way to either spin or block on the same lock instance.

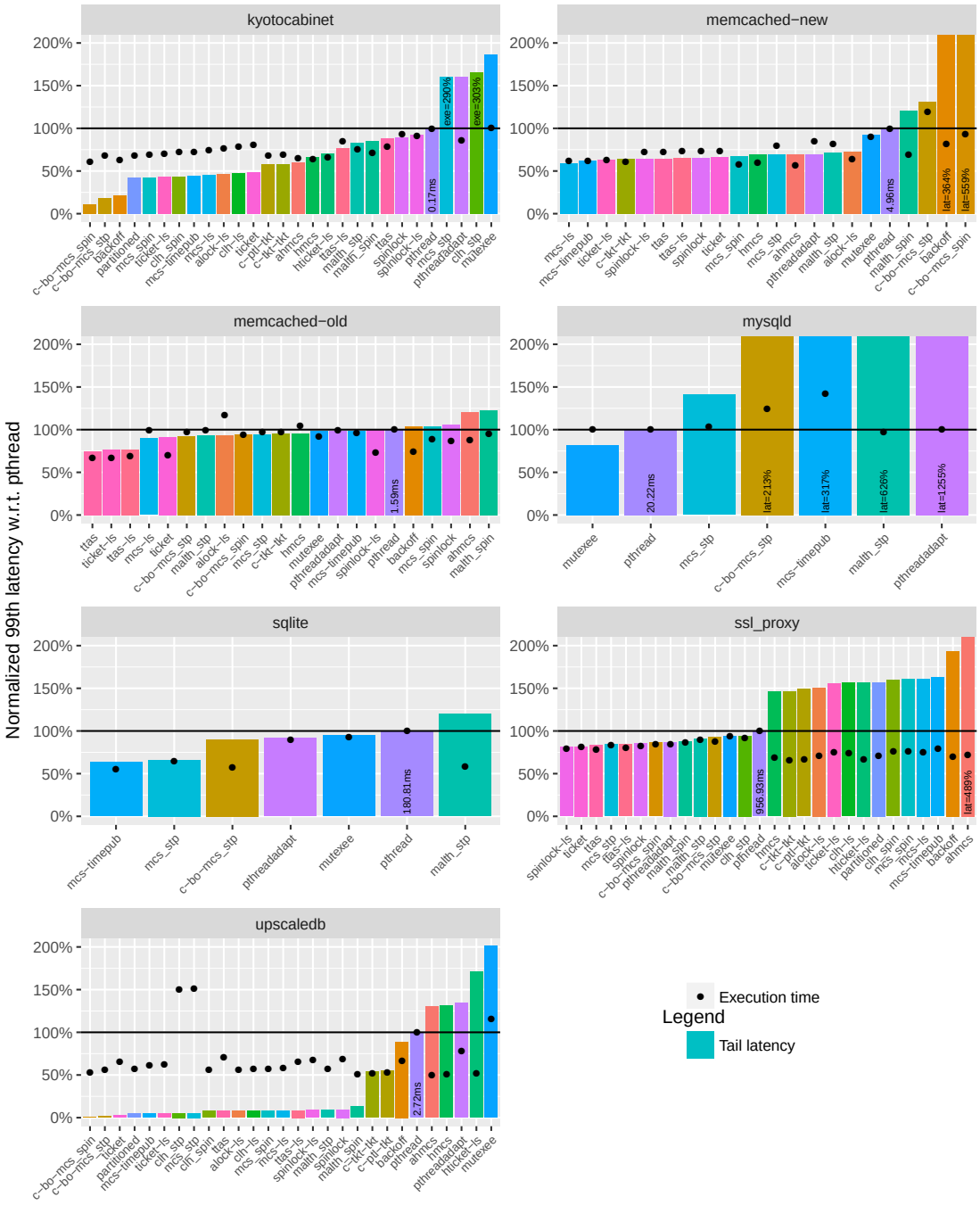


Figure 4.5 – For each server application, the bars represent the normalized 99th tail latency (w.r.t. Pthread) and the dots execution time (lower is better) normalized (w.r.t. Pthread) of each lock algorithm (**A-64 at *opt nodes***).

4.5 Analysis of lock/application behavior

In order to understand the performance of a lock algorithm on a given application, we perform a detailed analysis that explains, for each of the studied applications, which types of locks work well/poorly and why. We highlight that a lock can have many side-effects on the performance of an application.

In Section 4.5.1, we give general insights that we draw from our analysis by presenting, for every application, the performance bottleneck it suffers from, and which lock(s) to prefer or to avoid when running it. We found that, beyond the pure performance of a lock algorithm under high contention, different applications stress different aspects of a lock algorithm (e.g., memory footprint, scheduler preemption tolerance). In Section 4.5.2, we present seven *properties* shared by the studied lock algorithms, which, when cross-referenced with the performance bottlenecks of an application and a set of general guidelines that we provide, can help a developer to predict whether a lock algorithm performs well or poorly on a given application.

Note that the above-mentioned analysis was performed on the A-64 machine, and was performed with the aim to find the main (lock-related) performance bottlenecks. For each bottleneck, we explain if it is more common at *opt nodes* or *max nodes*. Nonetheless, the observations made in Sections 4.2 and 4.3 are not specific to lock performance on the A-64 machine. Thus, we think that the conclusions of this Section can be applied to different machines, and not only to throughput but also to energy efficiency.

4.5.1 Summary of the lock/application behavior analysis

In this Section, we give general insights that we draw from the detailed analysis of the different lock-sensitive applications. Table 4.16 lists, for each lock-sensitive application its main performance bottleneck with respect to locking (in column 2). We also recommend which family of lock algorithms (i.e., lock algorithms sharing a similar property) to prefer or avoid for each of the studied applications (detailed in Section 4.5.2). For example, we observed that the performance bottleneck of fluidanimate is due to a high number of uncontended lock acquisitions. As a consequence, it is better to use a light lock algorithm, i.e., a lock that can be acquired very quickly when there is no other thread trying to acquire it at the same time (e.g., with only one atomic CPU instruction). Overall, we identified 9 performance bottlenecks across 22 applications, that can be summarized into four categories: lock contention, scheduling issues, memory footprint and memory contention.

	Performance Bottleneck(s)	Advice
facesim	scheduling issue: *lock handover*	**avoid** FIFO locks
radiosity	lock contention: high	**avoid** light or parking locks
radiosity_ll	lock contention: extreme	**prefer** hierarchical locks
ferret	scheduling issue: *lock handover*	**avoid** FIFO locks
streamcluster	lock contention: extreme (mixing trylocks and locks)	**prefer** locks with a contention-hardened trylock operation
dedup	kernel lock contention inside the page fault handler	**prefer** locks with small memory footprint
vips	scheduling issue: *lock handover*	**avoid** FIFO locks
fluidanimate	page fault memory erase page lot of uncontended lock acquisitions	**prefer** light locks
pca	memory contention	**prefer** locks lowering memory traffic
linear_regression	lock contention: high	**avoid** light or parking locks
s_raytrace	lock contention: high	**avoid** light or parking locks
s_raytrace_ll	lock contention: high	**avoid** light or parking locks
ocean_cp/ncp	scheduling issue: *lock handover* lock contention: high	**avoid** light or FIFO locks
water_spatial water_nsquared	page fault memory erase page	**prefer** locks with small memory footprint
fmm	page fault memory erase page	**prefer** locks with small memory footprint
volrend	lock contention: extreme	**prefer** hierarchical locks
mysql	lock contention: extreme memory contention scheduling issue: *lock holder preemption*	**prefer** parking locks
ssl_proxy	lock contention: extreme	**prefer** hierarchical locks
kyotocabinet	lock contention: extreme	**prefer** hierarchical locks
upscaledb	lock contention: extreme	**prefer** hierarchical locks
memcached-old	lock contention: extreme (with trylocks)	**prefer** locks with a contention-hardened trylock operation
memcached-new	lock contention: high	**avoid** light or parking locks
sqlite	scheduling issue: *lock holder preemption*	**prefer** parking locks

Table 4.16 – Lock-sensitive application performance bottleneck(s) and lock algorithms choice advice.

Lock contention

One of the key performance factors of a lock algorithm is how well it behaves under contention, i.e., its performance when a set of threads tries to acquire the same lock instance at the same time. Depending on their design, lock algorithms achieve their best performance at different levels of contention. For example, lock algorithms like Spinlock and TTAS are simple enough so that acquiring the lock under a low level of contention is only a matter of a few cycles. However, this simplicity leads to a performance collapse under higher levels of contention. On the contrary, algorithms like MCS or HMCS are designed to perform best under high levels of contention, at the expense of a high cost to acquire the lock when there is no other thread competing to acquire it. We observe four different performance bottlenecks depending on how many threads concurrently try to acquire a lock instance and how they try to acquire it: high levels of contention, extreme levels of contention, trylock contention and many uncontended lock acquisitions. Note that lock contention can be observed both at *opt nodes* and *max nodes*.

High levels of contention. A high number of threads (between approx. 10 to 40 threads on A-64) are waiting to acquire the same lock instance at the same time. To measure the contention level on a lock, we take regular snapshots of the application state, looking at how many threads are currently waiting for a lock. More precisely, each time a thread requests a lock, it puts the lock address inside a private cache-aligned memory location, and all such locations are read by a background thread every second. This provides us with a low-overhead approximation of the real number of threads waiting for a lock, with respect to a more straightforward yet costly approach where a counter is atomically incremented before waiting for a lock and decreased when the lock is acquired. Radiosity, linear_regression, s_raytrace, s_raytrace_ll are the four lock-sensitive applications that suffer from this performance bottleneck.

Radiosity is parallelized using per-core distributed task queues, where each thread can steal work from another task queue. Radiosity allocates a large number of locks (4k); still only two locks are highly contended. With HMCS, one of the best locks for this application, on average, 60% of all the total threads wait on one of the two stressed locks, while there is virtually no contention on the other 4k locks. For linear_regression, we observe that there is only one lock inside the application that protects a distributed task queue. This lock suffer from high levels of contention (65% of the threads waiting on the lock). S_raytrace and s_raytrace_ll render a 3-D scene partitioned among threads and there is a global task queue protected by a single lock. Still, the contended lock is not the global task queue lock, but a lock protecting a single counter used to implement a global unique identifier generator. For the short-lived version (resp. for the long-lived version), on average, 40% (resp. 60%) of the threads are waiting for the same lock (using HMCS, one of the best lock algorithms). When using an atomic

`fetch_and_add`, we observe a 1.8× (resp. 3×) performance improvement for the short-lived version.

For high levels of contention, lock algorithms that rely on local spinning (e.g., MCS) or on a hierarchical approach (e.g., AHMCS) are well suited (see Section 2.1.2). Light lock algorithms (e.g., Spinlock) and lock algorithms using a parking waiting policy must be avoided when possible.

Extreme levels of contention. A very high number of threads (more than 40 on A-64) are waiting to acquire the same lock instance. This phenomenon can be observed on seven of the lock-sensitive applications: radiosity_ll, volrend, MySQL, SSL Proxy, Kyoto Cabinet, upscaledb.

Radiosity_ll, the long lived version of radiosity, also suffers from lock contention. Contrary to the short lived version, radiosity_ll puts more pressure on the locks[10]. Volrend suffers from lock contention on the lock instances protecting different distributed task queues, as well as on a lock instance used to implement a barrier that separates the computation steps. These task queue locks (as well as the barrier lock) suffer from extreme levels of contention, especially the barrier lock that suffers from spikes of contention when all the threads wait for the barrier at the same time. MySQL suffers from lock contention on a lock that protects the page cache, a data structure that serves as an in-memory cache for the SQL table data stored on disk. This lock is heavily stressed: we observe on average 50 threads (on a 64-core machine) competing for the same lock instance, resulting in 40% of the thread lifetimes spent waiting to acquire this lock. The SSL Proxy application implements a reverse SSL proxy using OpenSSL via the Boost ASIO library. This application is subject to a huge performance collapse: the optimized number of nodes is one. In this application, the main bottleneck is a lock protecting the error queue of OpenSSL, which suffers from extreme levels of contention (on average 85% of the threads wait on the same lock). Similarly to Zemek [111], we found that the problem comes from an inefficient usage of the OpenSSL library by the Boost ASIO library. Indeed, the original lock that the OpenSSL library requests is a reader-writer lock; still Boost ignores it and uses a classic mutex lock, lowering the potential degree of parallelism. Kyoto Cabinet is a straightforward implementation of a database. As explained by Afek et al. [1], the most contended lock instance is the lock protecting the global hash table storing the data. Indeed, all database operations (create/insert/update/delete/lookup) need to acquire the same lock, which becomes highly contended. Upscaledb is an in-memory key/value store tailored for efficiency of analytical functions. Contrary to popular database engines like InnoDB for MySQL that use fine-grained locking (generally one lock for a row/set of rows), upscaledb

[10]The short-lived version is launched with a BF [53] refinement epsilon of $(1.5e-3)$ and the long-lived version is launched with a BF refinement epsilon of $(1.5e-5)$. With a lower epsilon, computations are refined more frequently, creating more tasks.

uses only one lock instance to protect the whole database. Such a poor design choice explains why upscaledb does not scale: indeed we observe that all of the threads spend 98% of their execution time waiting for the lock.

For these applications, the well-performing lock algorithms are the ones designed to support extreme levels of contention, such as AHMCS, HMCS and the cohort locks.

Trylock contention. Some of the studied applications (e.g., Memcached-old, streamcluster) use the (non-blocking) *trylock* operation to acquire a lock instance. However, most of the existing papers on lock algorithms focus on the design and evaluation of lock operations with blocking semantics. Trylock is a non-blocking operation, and we observe that an algorithm that optimizes the (blocking) *lock* operation can have a totally different behavior for its trylock operation. In fact, most algorithms (even the more elaborate ones, e.g., AHMCS) have a trylock operation as simple as the one of the simplest algorithm (Spinlock), which consists of a simple atomic instruction on a single memory address. As an example, the MCS trylock operation is a *compare-and-set* on the tail pointer of the waiter's linked list.

Streamcluster, and its long-lived version streamcluster_ll, are examples of applications that stress trylocks. Streamcluster heavily relies on a custom barrier implementation to synchronize threads between the different phases of the application. This barrier implementation uses a mix of trylock and lock operations, as well as condition variables. During Streamcluster execution, 30% of the threads are on average either inside a trylock or a lock invocation. Because streamcluster mixes locks and trylocks, we observe that algorithms having a contention-hardened trylock operation, like HMCS, exhibit better application performance. Such algorithms include rather complex trylock implementations, with tens of instructions. On the contrary, poor-performing algorithms, like Spinlock, have extremely simple trylock implementations (i.e., Spinlock simply does one *compare-and-set* instruction). As a result, an uncontested trylock costs on average 220 cycles with HMCS and 170 cycles with C-BOMCS (two well-performing locks in Streamcluster), while it costs 60 cycles with Spinlock and 80 cycles with MCS (two poor performing locks when trylock is heavily contended). Another example where trylock is important is Memcached-old. Instead of calling the Pthread mutex lock operation, Memcached-old relies on trylock to improve reactivity for short critical sections. The most contended lock is a global lock protecting the cache hash-table (`item_global_lock`), followed by the lock protecting the in-house memory allocator (`cache_lock`). As a results, on average 80% of the threads wait behind one of these locks. These results illustrate that contention-hardened trylocks can play an important performance role under high levels of contention.

Among the studied algorithms, only a few algorithms (HMCS, cohort locks, Partitioned and MCS-TimePub) implement a trylock operation performing well under high

levels of contention. For example, the HMCS and the cohort locks implement a trylock in a hierarchical manner, leading to better performance on NUMA machines[11].

Many uncontended lock acquisitions. One of the applications (fluidanimate) creates a large number of lock instances (500k locks). These locks are used to protect each cell of the grid, and are only used by one or two threads at the same time: most of the time a thread acquires the lock without any competition. More precisely, fluidanimate calls `pthread_mutex_lock` 5 billions times and half of the acquisitions are immediate, while for the other half a thread waits only because there is another thread inside the critical section, never because there are other waiting threads.

While the main performance bottleneck of facesim is related to memory (see below), we found that, similarly to the SyncPerf study [5], as lock are rarely contended, an important performance factor is the best-case critical path, i.e., the time to acquire a lock instance when it is not contended. We observe that the "lightest" lock algorithms (i.e., the ones with a short code path for acquisition in the absence of contention) exhibit very good performance (e.g., Backoff, Spinlock, Ticket, TTAS, which require roughly 40 cycles to acquire a lock under no contention). On the contrary, lock algorithms like cohort locks or HMCS (that require roughly 190 cycles to acquire an uncontended lock) perform the worst, because a thread needs to acquire two locks (the NUMA-local lock and the global one) most of the time, hampering the execution.

For application highly sensitive to the time spent acquiring a lock instance in the absence of contention, we recommend to use the "lightest" lock algorithms, such as Backoff, Spinlock, Ticket or TTAS.

Scheduling issues

The performance of some of the studied applications mainly depends on how well a given lock algorithm behaves with respect to scheduling choices. We observe two different performance bottlenecks related to scheduling: the lock holder preemption effect and the lock handover effect.

Lock holder preemption. The *lock holder preemption* effect is a well-known issue [18] with lock algorithms using a spinning waiting policy. It happens when a thread A waiting for a lock instance preempts a thread B that is the lock holder. Doing so, A runs on a core waiting for B to release the lock instance, while the rescheduling of B is

[11] The trylock algorithms for HMCS and cohort algorithms acquire the per-socket lock instance, and if successful, try to acquire the global lock instance. The Partitioned lock first checks non-atomically if there is another thread waiting for the lock, then does a classic (blocking) mutex lock acquisition. The MCS-TimePub trylock runs an adaptive algorithm that is long, thus lowering the number of concurrent atomic instructions.

delayed because of A, thus delaying B to finish the critical section, and release the lock instance for A. This pattern is highly inefficient. In the worst scenario, this can lead to lock convoy: while the lock holder is descheduled, each thread progresses and eventually tries to acquire the lock instance, spinning, thus delaying the rescheduling of the lock holder. This issue is usually observed in highly-threaded applications, where the scheduler has to frequently decide which thread to run on which core. This effect is more likely to be seen at *max nodes*; still some applications are already highly-threaded at *opt nodes* (e.g., MySQL and SQLite). Note that all kinds of spinning algorithms are affected by this phenomenon: the simplest ones (e.g., TTAS), FIFO (e.g., MCS_Spin) and hierarchical approaches (e.g., HMCS). In fact, lock holder preemption is mainly a property of the program concurrency-design, not the lock design. The lock holder is more likely to be preempted inside critical sections with applications composed of long critical sections and that over-subscribe threads to cores (e.g., databases).

MySQL and SQLite are two highly-threaded applications suffering from the *lock holder preemption* effect. MySQL uses a large thread pool (hundreds of threads) to handle queries from clients. SQLite creates a server that listens for client requests on a Unix socket and uses a globally shared work queue protected by a single lock instance; still many other lock instances are used to synchronize internal data structures. The benchmark used (see Section 4.1.3) creates hundreds of threads.

In order to mitigate this effect, it is recommended to choose lock algorithms using a parking waiting policy. Indeed, with this policy, when a thread waits for too long, it deschedules itself, and the scheduler does not schedule it back until the lock instance has been released. In particular, we recommend Malth_STP, because, thanks to its concurrency control mechanism, it is able to put aside some threads and let others progress. The smaller set of running threads allows lowering the pressure put on the lock instances, and as a consequence the overall performance of the application is improved. Another well-performing lock is the MCS-TimePub lock algorithm, which is specifically designed to mitigate the *lock holder preemption* effect. Interistingly, some operating system provide a mechanism (e.g., `schedctl` in Solaris) to give hints to the OS scheduler when a thread is inside a critical section in order to avoid preempting it.

Lock handover. This phenomenon (also known as the *lock waiter preemption problem* [97]) happens with algorithms that use a direct handoff succession policy (see Section 2.1.2). When a thread waiting in line for a lock is preempted, all other waiting threads after this one are delayed. Worse, these threads spinlock their entire timeslice, postponning the rescheduling of the descheduled thread. In principle, this problem is unlikely to appear on platforms that do not use more threads than cores. In practice, lock waiter preemption actually occurs quite often even when there are never more threads than cores. Indeed, the Linux CFS scheduler sometimes migrates two (or more) threads on the same core, thus leading to situations where the next-acquiring

thread is preempted, and where other waiting threads spin uselessly. These migrations are mainly observed when there are many blocking calls inside the application (e.g., condition variables, I/O). This phenomenon is more likely to happen at *max nodes*.

There are six of the lock-sensitive applications that suffer from the *lock handover* effect: facesim, ferret, vips, ocean_cp and ocean_ncp, streamcluster. Facesim creates one thread per core that implements a fork-join computation model [17]. The applications uses a barrier to synchronize the successive fork-join phases, implemented with a mutex lock and a condition variable. When threads wait on the condition variable, they might be migrated by the scheduler so that when they are unblocked (i.e., when leaving `pthread_cond_wait`) they are scheduled on the same core. There are 10× more migrations for a poor performing lock algorithm (MCS, 40k) than for the MCS-TimePub lock algorithm (4k): with a poor performing lock, threads have more chances to share the same core. Note that a straightforward solution to "fix" facesim is to pin each thread to a distinct core, thus avoiding inefficient migrations. For example, with MCS pinning improves performance and yields roughly the same results as MCS-TimePub, one of the best performing locks.

Ferret is parallelized using a pipeline model with 6 stages, where the four middle stages use a thread-pool to handle requests. Ferret is subject to the *lock handover* effect: treads are migrated because they stress the condition variables propagating work through the stages. To assess the impact of this effect, we compute the lock handover latency, i.e., the time delta between when a thread releases the lock and when the next thread that was waiting for the lock acquires it. The lock handover latency is on average 15× higher with MCS than with Spinlock (30M instead of 2M cycles). As a comparison, on a micro-benchmark that does not suffer from the *lock handover* effect (1 thread pinned on each core, all trying to acquire the same lock), the average lock handover latency is of 460 cycles with MCS, and 46k with Spinlock.

Vips automatically builds a parallel image processing pipeline, each stage being supported by an independent pool of threads. Threads are migrated inside vips after page faults and calls to condition variables.

Ocean_cp and ocean_ncp are applications simulating large-scale ocean movements. We observe that the main bottleneck in the ocean applications is a barrier implemented with condition variables and used to synchronize the different phases of the simulation.

Streamcluster heavily relies on a barrier to synchronize the threads, and the barrier implementation uses a mix of trylock and lock operations, as well as condition variables.

For applications suffering from the *lock handover* effect, FIFO algorithms using a wait-

ing policy based on pure spinning (e.g., Ticket, MCS) should be avoided in such cases.

Memory footprint

A less known category of locking performance bottlenecks is related to the memory footprint of a lock instance. Indeed, not all lock algorithms occupy the same space in memory, and if many lock instances are allocated by the application, it can become a critical performance factor. We observe two different performance bottlenecks related to the memory footprint of a lock, which depend on the memory allocation pattern.

Erasing new memory pages inside the page fault handler. With applications like fmm, fluidanimate, water_spatial and water_nsquared, one thread creates and initializes all the lock instances at the beginning of a run, allowing all other threads to use them. More precisely, water_spatial creates 125k lock instances, water_nsquared 32k, fmm 2k and fluidanimate 500k. The allocating thread requests memory pages from the kernel, that are erased (i.e., filled with zeros) upon the first access. For an application with many lock instances, a lock algorithm with a big memory footprint triggers many memory page requests to the kernel, each of them needing to be erased. For example, with fmm, a poor performing lock (AHMCS) triggers 17% (400k) more page-faults than a well-performing lock (Spinlock). Water_spatial is another good example of an application where this effect has a severe impact on performance: the execution time difference between Spinlock (a well-performing lock) and AHMCS (a bad performing lock) can be explained by the difference of the time spent erasing pages (1 vs 19 seconds). This bottleneck is observed both at *opt nodes* and *max nodes*, and happens during the initialization phase of the application. One way to alleviate the bottleneck is to rewrite the application to allocate locks concurrently (though this might cause other issues, see the next bottleneck description). Another way is to reduce the ratio of the initialization time over the steady-state time by increasing the steady-state time. However, this is not always possible. For example, the number of allocated locks for water_nsquared is proportional to the input size, upon which the steady-state time depends. In such applications, we thus recommend to use lock algorithms that have a low memory footprint (e.g., Spinlock, Ticket) to decrease the number of pages that need to be erased.

Applications that need to control their memory footprint can benefit from dynamically allocating per-node data structures of hierarchical locks upon first access [64]. It benefits to applications where locks are in fact rarely acquired by threads from multiple NUMA nodes. However, it leads to more dynamic allocations to be made, which might introduces kernel lock contention inside the page fault handler (see below).

Kernel lock contention inside the page fault handler. On some applications, at both *opt nodes* and *max nodes*, all threads are constantly creating new lock instances, putting pressure on the memory allocator (i.e., `malloc`). Internally, `malloc` requests pages of memory to the kernel (via `brk` and `mmap`), which generates page faults when the pages are first accessed. The page fault handler tries to insert the new page into a process-shared data structure (the virtual address space data structure), protected by a single reader/writer lock [27]. The contention on this kernel lock becomes more performance critical than the one on the application-level locks, because all threads need exclusive write access to the data structure, and the lock is generally kept for a long time.

Dedup is an example of application where there is kernel lock contention inside the page fault handler. Through its lifetime, dedup creates a very large number of locks (266k), which puts a huge pressure on the memory allocator. To measure the impact of the lock algorithm memory footprint on the performance of dedup, we compare CLH-ls, which has a huge memory footprint, with Pthread, which has a low memory footprint. Using CLH-ls, we observe an increase of the number of calls to `mmap` by a factor of 96 and an increase of the number of calls to `brk` by a factor of 46. Moreover, we observe that using the Pthread lock algorithm, at *opt nodes*, dedup spends 3.3 seconds (30%) of the total execution time inside the kernel page fault handler, whereas with CLH-ls it spends 80 seconds (80%) of the total execution time. One can argue that the performance bottleneck has been introduced by the design of our transparent interposition design, which requires one dynamic memory allocation per lock instance, even if the original POSIX lock instances were not dynamically allocated (i.e., the instance is on the stack), or allocated in batches. However, dedup by itself, i.e. without LiTL, continuously stresses the memory allocator, because it continuously allocates chunks of data, each containing a lock instance. Indeed, when we modify the source code of dedup to increase the allocated size of a lock instance that protects a chunk from concurrent modifications, without LiTL, we still observe a performance decrease of 60%.

As a consequence, the fewer memory pages are used when allocating lock instances, the fewer insertions of new pages inside the virtual address space are made, and thus the lower contention on this lock is observed. We thus recommend lock algorithms having a low memory footprint like Spinlock or Backoff for such applications.

Memory contention

Lock algorithms can have significant side effects on applications that are primarily affected by other kinds of bottlenecks, like main memory contention.

Pca (and its long-lived version pca_ll) is a good example of such a phenomenon. Validating the observations on pca from the original paper [90], we found that pca suffers

either from lock contention (for algorithms that do not support high levels of contention, e.g., Spinlock) or memory controller saturation[12] (for the others, e.g., Pthread). For example, with Pthread (pca suffers from memory controller saturation), we observe a 44% performance increase when we interleave the memory pages of the application, i.e., when the memory pages of the application are allocated in a round-robin fashion on all the NUMA nodes of the machine. This is a clear indicator that, without interleaving, the memory controller of one NUMA node becomes overwhelmed, receiving too many requests from all the threads. Besides, even with interleaving, the memory bottleneck does not fully disappear. Indeed, we observe an increase from 0.4 stalled cycles per instruction (SPI) outside locking primitives with Malth_Spin (one of the best locks) to 2.25 SPI with MCS[13] (a bad performing lock). However, note that the stalled cycles are observed inside the parallel code sections of pca. By being somewhat "too" fast, MCS allows many threads to run in parallel, thus increasing the memory contention of the parallel code sections of pca. More precisely, the number of stalled cycles due to memory accesses, which account for 98% of all stalled cycles, is 20× higher with MCS than with Malth_Spin. Note that this phenomenon is more likely to appear at *max nodes*, because memory contention exists when a large number of threads access memory concurrently.

In such cases, we recommend lock algorithm that reduce the number of concurrently running threads in the application, thus the number of concurrent memory accesses (e.g., Malth_Spin).

4.5.2 Guidelines for lock algorithms selection

In Section 4.5.2, we describe the different properties of the studied lock algorithms, and in Section 4.5.2 we discuss guidelines to help a developer choosing a lock algorithm for a given application.

Lock properties

Knowing the performance bottleneck of an application, a developer can now decide which lock algorithms to use in an application. Table 4.17 summarizes the main properties of each lock algorithm. Overall, we identified seven properties shared by the

[12]While experimentally assessing the performance overhead of LiTL (see Section 3.4), we noticed a corner case with pca. More precisely, we observe that, most of the time, LiTL improves the performance w.r.t. the manually implemented version. This performance difference comes from the condition variable algorithm of LiTL that lengthens the critical section. Indeed, as pca and pca_ll suffer from memory contention, longer critical sections lower the number of threads running in parallel outside the critical sections, thus improving performance. However, the best locks with LiTL are also among the best manually implemented locks.

[13]A careful reader may argue that MCS should not cause heavy cache coherence traffic, because it uses local spinning: MCS should be mostly spinning on the L1 cache and triggers cache coherence traffic only when the lock holder releases the lock to the next waiting thread.

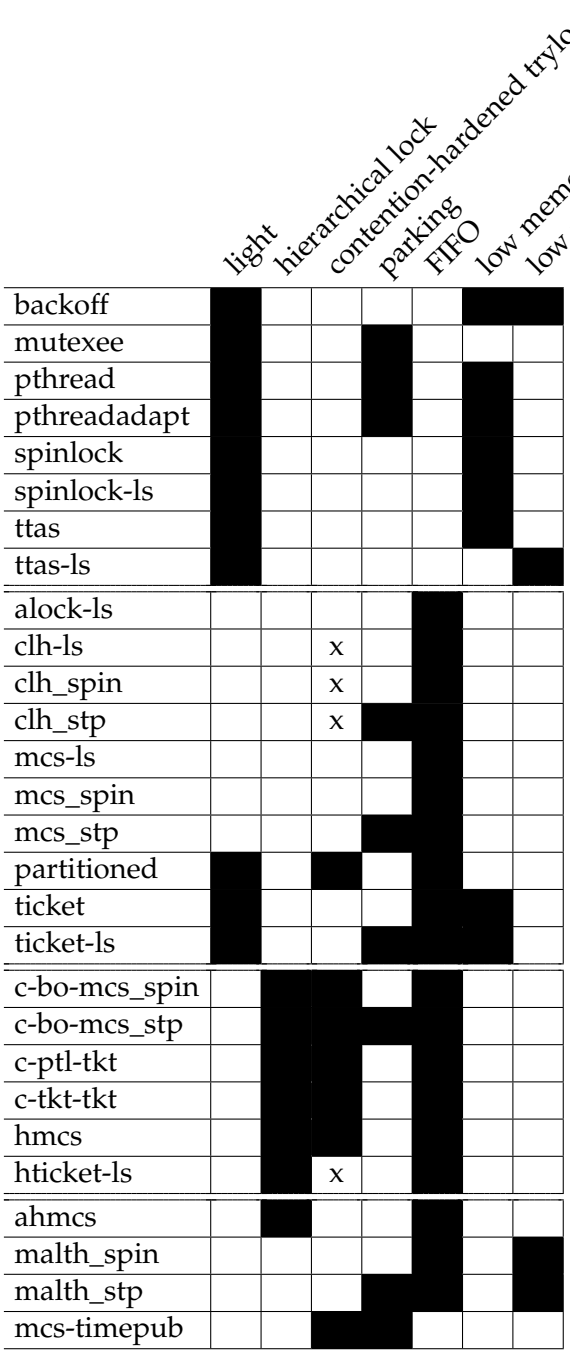


Table 4.17 – Lock algorithm properties. The algorithms are grouped by categories as defined in Section 2.1.2. For example, ahmcs does not use a parking waiting policy, nor does it have a low memory footprint. However, it is a hierarchical lock algorithm. Some lock algorithms do not support the trylock operation and thus cannot be run with applications that use this operation: we denote these cases by a cross sign.

☐ locks without the property ■ locks with the property [x] trylock not supported

studied lock algorithms that have an impact on performance. We also describe how the different design properties described in Section 2.1.2 are related to these "behavioral" properties. We first present properties related to different levels of contention, then properties that can affect scheduling, and finish with properties related to memory.

1. *Light:* lock algorithms having a short code path to acquire the lock when uncontended. Algorithms such as Spinlock, Backoff or TTAS have this property, where an uncontended lock acquisition is almost only an atomic instruction. Algorithms using a context such as MCS or CLH are generally heavier, because they need to setup the context before acquiring the lock, even if there is no contention. We also observe that there is no hierarchical lock that is light: cohort lock algorithms acquire both local and global locks, and even AHMCS, which implements a fast path; still needs to acquire one uncontended MCS lock. Finally, all existing load-control lock algorithms are heavy, because the load control decision is on the critical path.

   Note that for applications where a single thread acquires a lock, biased locking [35] can improve performance. This technique can be used to enhance any lock algorithm with an atomic-free fast path, and switches to the default lock algorithm upon the first lock acquisition by a second thread.

2. *Hierarchical lock:* lock algorithms designed to take into account NUMA architectures, where the cost of accessing a lock instance from a different socket is higher than the one when the lock instance is already inside a cache of the local socket. This category is the same category as described in Section 2.1.2.

3. *Contention-hardened trylock:* lock algorithms with a trylock operation tolerating moderate to high levels of contention. We observe that some applications use the trylock operation to do busy-wait, i.e., the trylock operation is continuously called in a loop until the lock is acquired. In practice, a large number of atomic instructions are executed concurrently, flooding the memory interconnect with cache-coherence traffic. Here, lock algorithms that lower the cache-coherence traffic are the ones that perform the best. We observe that hierarchical locks have a contention-hardened trylock, because a thread needs to trylock both the local and the global lock[14]. We also observe that algorithms like MCS-TimePub and Partitioned have a contention-hardened trylock because their trylock operation takes time (i.e., the operation consists of one atomic instruction and a significant number of non-atomic instructions), thus lowering the cache-coherence traffic.

4. *Parking:* lock algorithms using a spin-then-park or a direct parking waiting policy (see Section 2.1.2).

[14]With the exception of AHMCS, where the trylock can be directly made on the top MCS lock.

5. *FIFO:* lock algorithms imposing an order on the acquisitions of a lock instance according to the thread arrival times, i.e., if a thread A tries to acquire the same lock instance as B before B, A enters the critical section before B. Note that some lock algorithms leave some degree of freedom regarding this order, i.e., a thread might enter the critical section before another thread that had been waiting for a longer amount of time (e.g., with the cohort lock algorithms that favor threads running on the same socket as the lock holder). This category regroups a subset the lock algorithms using a direct handoff succession policy (see Section 2.1.2).

6. *Low memory footprint:* lock algorithms having a low memory footprint. All locks that need a context (e.g., MCS, CLH, Malthusian) have a high memory footprint, because each thread needs its own context per lock held (or being acquired). Besides, hierarchical lock algorithms also have a high memory footprint because one lock instance is composed of one top lock instance, and one instance per NUMA node, but the footprint can be lowered by dynamically allocating per-node data structures of hierarchical locks upon first access [64].

7. *Low (memory) interconnect traffic:* lock algorithms that only induce a moderate traffic on the memory interconnect of the machine. Algorithms using a load-control mechanism sensitive to the concurrency level (e.g., Malthusian) reduce the number of threads running concurrently, thus the pressure on the memory interconnect. Surprisingly, lock algorithms that perform both poorly under contention *and* which do not flood the interconnect with cache-coherence messages (e.g., Backoff, TTAS-ls) are good choices to lower the memory interconnect utilization.

Choice guidelines

Figure 4.6 shows a series of steps to follow in order to select which lock algorithm to use with each application. The steps are questions the developer needs to answer that help select a small subset of lock algorithms. A box with a white background represents a question and a box with a gray background suggests the developer to select or avoid some locks. For example, for upscaledb, the developer starts by asking if the application has more threads than cores. Upscaledb does not have more threads than cores. Next, the application is profiled to know if it performs many calls to the scheduler (e.g., with I/O, conditions variables), which might lead to thread migrations. Upscaledb does not call the scheduler often, so the developer can still consider FIFO algorithms. Moving forward, upscaledb does not create many lock instances, does not use the trylock operation and does not suffer from memory contention. We are now at the last step, where the developer has to chose a lock algorithm regarding the levels of contention the lock instances inside upscaledb suffer from. Remember that because upscaledb does not have more threads than cores, and does not call the scheduler often, the developer should choose an algorithm that uses a spinning waiting policy.

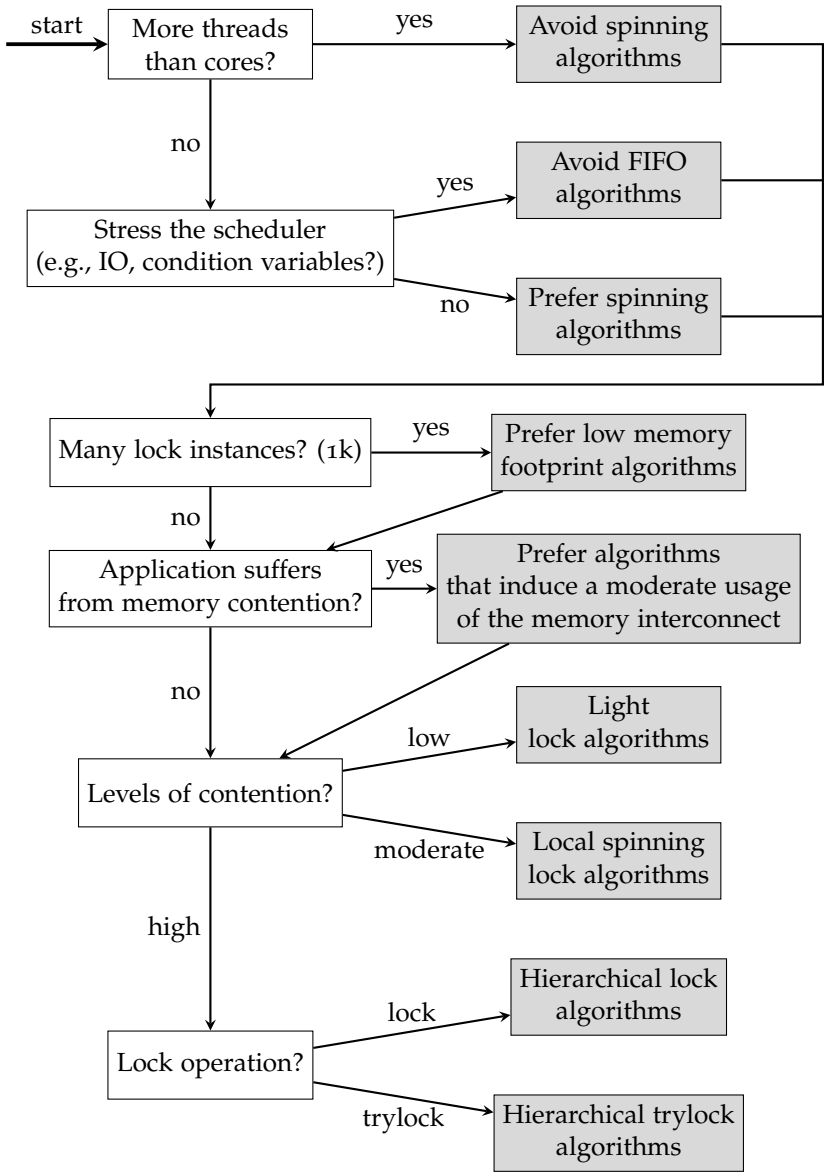


Figure 4.6 – Steps to follow for the application developer to chose a lock algorithm.

We observe that upscaledb suffers from extreme levels of contention. Therefore the developer should choose a hierarchical spinning lock algorithm, for example AHMCS.

A word of caution: these guidelines are cursory, because carefully tuning a lock algorithm is highly dependent on a given workload and machine. They give a hint to the developer for the choice of a lock, and mostly target applications in which lock access patterns are stable (e.g., the most contended lock is always the same and it always suffers from a constant level of contention). Many lock bottlenecks can be suppressed by redesigning the application with smaller critical sections, or by using more scalable synchronization primitives, such as lock-free data structures. Besides, some techniques enhancing lock algorithms (e.g., lazy lock allocation [64], biased locking [35]) can be beneficial to adapt a given lock that is not initially the best for a given workload. Finally, for applications where the access pattern of a lock varies during the workload, adaptive lock algorithm such as GLK [10] can be used.

Note also that these guidelines do not cover all the possible configurations. For example, if an application allocates many lock instances, and these instances suffer from extreme levels of contention, there is no hierarchical lock algorithm having a low memory footprint. Nonetheless, we propose these guidelines based on our analysis of the set of studied applications: they cover each application, and we believe that the set is large enough to be representative.

5 Conclusion

There is a large number of lock algorithms for multicore machines, leaving developers with the cumbersome task of choosing which algorithm to use for an application. One of the main reasons for this complexity is that there were no clear guidelines and methodologies helping developers to select an appropriate lock for their workloads.

This thesis has presented a broad study of the throughput, tail latency and energy efficiency of 28 lock algorithms with 40 applications on Linux/x86 and four different multicore machines. More precisely, we considered, (i) applications from three benchmark suites, as well as six real-world applications; (ii) lock algorithms from four categories (competitive, direct handoff, hierarchical, load-control, as defined in Section 2.1.2) with different waiting policies (spinning and spin-then-park); (iii) four multicore machines that differed in vendors (AMD, Intel), number of cores (20, 48, 64), NUMA topology (single-hop, I-48/I-20, vs. multiple-hop distance between nodes, A-64/A-48); (iv) multiple pinning (without pinning vs. thread-to-node pinning); (v) multiple BIOS configuration (performance vs. energy saving). In the quest to understand lock behavior, when choosing the best lock, for these 40 applications, application throughput and energy efficiency have been improved (on average) by respectively 90% and 110% with respect to the default POSIX mutex lock.

For this study, we designed LiTL, an interposition library allowing the transparent replacement of lock algorithms used for Pthread mutex locks. To be compatible with a wide range of applications, LiTL proposes a novel and efficient algorithm to support condition variables, a synchronization mechanism frequently used in practice. Besides, LiTL is not only compatible with locks that use the classical `lock/unlock` API, but also with lock algorithms requiring an additional per-thread structure to keep per-thread metadata (14 out of 28). In practice, LiTL has an overhead low enough to be used to compare the performance of lock algorithms.

From our study, we draw several conclusions, several of which have not been previously discovered: applications not only stress the lock/unlock interface, but also the

full locking API (e.g., trylocks, condition variables), the memory footprint of a lock can directly affect the application performance, for many applications, the interaction between locks and scheduling is an important application performance factor and lock tail latencies may or may not affect application tail latency. We also confirm previous findings [33, 42] on a larger number of applications, machines, and lock algorithms: no single lock is systematically the best, choosing the best lock is difficult (as it depends on the application, the workload, the machine, etc), and energy efficiency and throughput go hand in hand in the context of lock algorithms (i.e., improving the throughput necessarily improves the energy efficiency). Finally, from the insights of our in-depth analysis of lock-related performance bottlenecks, we gave guidelines for the choice of a lock algorithm based on given application characteristics, such as the number of lock instances, the levels of lock contention or if the application is over-threaded or not.

5.1 Lessons learned

Below, we highlight five main implications of this work on the research community.

First, lock-related research cannot simply focus on one of the many functions of locking. As we have seen, many applications not only the lock/unlock, but also other synchronization primitives such as condition variables and trylocks. For example, half of the applications use conditions variables to synchronize threads, e.g., dedup uses condition variables to implement a synchronized task queue. Memcached (among others), uses a trylock to protect very short critical sections, with the aim to avoid costly descheduling of threads waiting for a lock. **Lock designers must offer a full suite of lock, unlock, trylock, condition variables**, and maybe even barriers, and reader-writer locks.

Second, the effect of the memory footprint of locks as well as the importance of the lock-scheduler interactions has been underestimated. We observed that the lock holder preemption and the lock handover effects frequently occur in practice, even on a dedicated machine. However, many locks have been designed with the hypothesis that the scheduler decisions do not affect lock performance. **Lock designers should not ignore the OS scheduler and design their lock to deal with sub-optimal scheduling decisions.**

Third, some locks improve performance at the expense of a bigger memory footprint. For example, AHMCS stores lock acquisition statistics, MCS stores per-thread contexts, and hierarchical locks store per-node local locks. This is often necessary in order to reduce the number of cache line transfers involved in lock acquisitions/releases. However, some applications create thousands of lock instances, thus the memory footprint of the locks becomes problematic (e.g., incurring cache thrashing, page faults due to memory allocation). **Lock designers must keep in mind that the memory footprint**

of their lock affects performance, and thus should either keep it the lowest possible, or rely on techniques such as lazy allocations [64] when possible.

Fourth, despite the fact that lock algorithms affect performance in many ways (as discussed previously), most of the existing profiling tools only give a coarse-grained view of the effect of locks on performance. For example, the Free-lunch Profiler [30] measures the time spent inside critical sections, which does not give enough information on how to avoid such bottlenecks. One exception is SyncPerf [5], which focuses on lock acquisition patterns, but ignores aspects such as memory footprint and scheduling interactions. Therefore, **lock profiling tools should not only focus on the general effects of locks on performance, but also profile the interactions of all synchronization primitives with scheduling and memory, and the threads' lock access patterns.**

Fifth, different lock instances inside the same applications serve different purpose, or applications go through phases. Thus, lock access patterns differ, and choosing one lock at the beginning of the application for all lock instances will certainly be suboptimal in many cases. Previous work [70, 10] proposed to switch the lock algorithm by a given lock instance at run time. However, these solutions do not consider the full spectrum of lock-related performance bottlenecks presented in Section 4.5. As a consequence, **there is a need for dynamic and more complete approaches to better control locks' behaviors.**

5.2 Future research

In the light of the results presented in this thesis, we describe four research directions that would be interesting to follow.

Multicore performance

Application performance on multicore machines not only depends on locking, but also on other factors such as the OS scheduler (pinning strategies, scheduling policies, ...), memory policy, memory allocation, compiler, library version, kernel configuration, etc. Finding the optimal point in this multi-dimensional configuration space is hard: even choosing a lock algorithm is very difficult. Before designing a solution to this problem, we need to fully and deeply understand the determinants of each dimension, i.e., what are the main factors affecting the performance. As future work, it would be interesting to explore the scheduling and memory allocation dimensions, two of the dimensions (besides locks) that impact the most the performance of the studied applications.

Besides, modern multicore applications have evolved and now require to handle workloads at the nanosecond and microsecond scale. Yet, current scheduling and memory allocation techniques are still not able to cope with such time scales [14]. More research

is needed towards understanding and designing solutions for the needs of scheduling and memory allocation of modern applications. Moreover, the interactions between locking, scheduling and memory allocation still need to be explored.

Delegation algorithms

Recent lock implementations leverage delegation to execute critical sections [74, 112, 93]. Delegation has been shown to be highly efficient under very high levels of contention. However, delegation requires to dedicate core(s) to only execute critical sections and require to re-engineer the application (critical sections need to be expressed as functions). Further research is needed to design delegation-based solutions that not only work well for the specific case it was designed (a single application suffering from high levels of contention running on a dedicated machine), but with all other situations that we encountered during our study (e.g., over-threading, usage of different lock operations such as trylocks).

Automatic and dynamic solutions

Choosing the right lock algorithm is hard, and this choice might change at run time. In Section 4.5.2 we provided guidelines to facilitate this choice. However, these guidelines suppose that all lock instances in the application exhibit the same access pattern, which must be consistent during the execution (i.e., no phases). GLS [10] and SANL [112] are two solutions that automatically and dynamically change the lock algorithms used with respect to the contention levels and the number of running threads (per-lock instance). Yet, these solution do not consider many parameters affecting performance are not accounted considered (memory, scheduling, ...). Moreover, for solution like SANL that uses a delegation based lock, the critical sections of the application still need to be re-engineer, thus the solution is not transparent. A natural following of our work would be to automate the choice of a lock algorithm by integrating our guidelines inside a tool that select the best-suited lock algorithm. Besides, leveraging techniques traditionally used by transparent $N : M$ scheduling runtimes [102] might allow to transparently delegate the execution context of a running thread, thus transparently supporting delegation based algorithms.

Leveraging transactional memory

Transactional memory (TM) offers a clean and composable abstraction for concurrent programming on multicore machines. While research on transactional memory has been prolific, in some occasions it has been deemed to be a "research toy" due to too high overheads and too strong restrictions [23]. However, recent works [39] offer to dynamically switch between TM implementations with respect to the running work-

load, keeping the best of each implementation. Besides, recent lock algorithms such as AHMCS-HTM [24] and techniques such as lock elision [89] leverage transactional memory to improve locking performance, especially for low levels of contention. We believe that mixing transactional memory and locking is a promising research direction towards lock algorithms supporting all kinds of access patterns and execution environments.

Availability

The source code of LiTL and the data sets of our experimental results are available online [51]. The figures and tables for all the machines are available in the companion technical report [50].

Bibliography

[1] Yehuda Afek et al. "Amalgamated Lock-Elision". *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*. Ed. by Yoram Moses. Vol. 9363. Lecture Notes in Computer Science. Springer, 2015, pp. 309–324.

[2] Anant Agarwal et al. "Adaptive Backoff Synchronization Techniques". *Proceedings of the 16th Annual International Symposium on Computer Architecture. Jerusalem, Israel, June 1989*. Ed. by Jean-Claude Syre. ACM, 1989, pp. 396–406.

[3] Martin Aigner et al. "Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures". *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. Ed. by Jonathan Aldrich et al. ACM, 2015, pp. 451–469.

[4] Samy Al Bahra. *Concurrency Kit*. http://concurrencykit.org/. 2015.

[5] Mohammad Mejbah Ul Alam et al. "SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs". *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. Ed. by Gustavo Alonso et al. ACM, 2017, pp. 298–313.

[6] AMD. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models ooh-oFh Processors*. http://support.amd.com/TechDocs/42301_15h_Mod_ooh-oFh_BKDG.pdf. 2010.

[7] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*. Vol. 30. AFIPS Conference Proceedings. AFIPS / ACM / Thomson Book Company, Washington D.C., 1967, pp. 483–485.

[8] Nikos Anastopoulos et al. "Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors". *22nd IEEE International Symposium on*

Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008. IEEE, 2008, pp. 1–8.

[9] Thomas E. Anderson. "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors". *IEEE Trans. Parallel Distrib. Syst.* 1.1 (1990), pp. 6–16.

[10] Jelena Antic et al. "Locking Made Easy". *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*. ACM, 2016, p. 20.

[11] ARM. *In what situations might I need to insert memory barrier instructions?* http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka14041.html. 2011.

[12] Marc Auslander et al. *Enhancement to the MCS Lock for Increased Functionality and Improved Programmability*. U.S. Patent Application Number 20030200457 (abandoned). 2003.

[13] Woongki Baek et al. "Green: a framework for supporting energy-conscious programming using controlled approximation". *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. Ed. by Benjamin G. Zorn et al. ACM, 2010, pp. 198–209.

[14] Luiz André Barroso et al. "Attack of the killer microseconds". *Commun. ACM* 60.4 (2017), pp. 48–54.

[15] Noah Beck et al. "'Zeppelin': An SoC for multichip architectures". *2018 IEEE International Solid-State Circuits Conference, ISSCC 2018, San Francisco, CA, USA, February 11-15, 2018*. IEEE, 2018, pp. 40–42.

[16] Emery D. Berger et al. "Hoard: A Scalable Memory Allocator for Multithreaded Applications". *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000.* Ed. by Larry Rudolph et al. ACM Press, 2000, pp. 117–128.

[17] Christian Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, 2011.

[18] Mike W. Blasgen et al. "The Convoy Phenomenon". *Operating Systems Review* 13.2 (1979), pp. 20–25.

[19] Silas Boyd-Wickizer et al. "Non-scalable Locks are Dangerous". *Proceedings of the Linux Symposium*. Ottawa, Canada, 2012.

[20] Trevor Brown et al. "Investigating the Performance of Hardware Transactions on a Multi-Socket Machine". *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*. Ed. by Christian Scheideler et al. ACM, 2016, pp. 121–132.

[21] Irina Calciu et al. "Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores". *Principles of Distributed Systems - 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings*. Ed. by Roberto Baldoni et al. Vol. 8304. Lecture Notes in Computer Science. Springer, 2013, pp. 83–97.

[22] Irina Calciu et al. "NUMA-aware reader-writer locks". *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*. Ed. by Alex Nicolau et al. ACM, 2013, pp. 157–166.

[23] Calin Cascaval et al. "Software transactional memory: why is it only a research toy?" *Commun. ACM* 51.11 (2008), pp. 40–46.

[24] Milind Chabbi et al. "Contention-conscious, locality-preserving locks". *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*. Ed. by Rafael Asenjo et al. ACM, 2016, 22:1–22:14.

[25] Milind Chabbi et al. "High performance locks for multi-level NUMA systems". *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*. Ed. by Albert Cohen et al. ACM, 2015, pp. 215–226.

[26] Christoph Rupp. *Upscaledb*. https://upscaledb.com/. 2017.

[27] Austin T. Clements et al. "RadixVM: scalable address spaces for multithreaded applications". *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*. Ed. by Zdenek Hanzálek et al. ACM, 2013, pp. 211–224.

[28] HyperTransport Technology Consortium et al. "HyperTransport I/O link specification". *Revision* 1 (2008), pp. 111–118.

[29] Travis S. Craig. *Building FIFO and Priority-Queuing Spin Locks from Atomic Swap*. Tech. rep. TR 93-02-02. University of Washington, 1993.

[30] Florian David et al. "Continuously measuring critical section pressure with the free-lunch profiler". *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. Ed. by Andrew P. Black et al. ACM, 2014, pp. 291–307.

[31] Tudor David et al. "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 '15, Istanbul, Turkey, March 14-18, 2015*. Ed. by Özcan Özturk et al. ACM, 2015, pp. 631–644.

[32] Tudor David et al. "Concurrent Search Data Structures Can Be Blocking and Practically Wait-Free". *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*. Ed. by Christian Scheideler et al. ACM, 2016, pp. 337–348.

[33] Tudor David et al. "Everything you always wanted to know about synchronization but were afraid to ask". *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. Ed. by Michael Kaminsky et al. ACM, 2013, pp. 33–48.

[34] Dave Dice. "Malthusian Locks". *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. Ed. by Gustavo Alonso et al. ACM, 2017, pp. 314–327.

[35] Dave Dice et al. "Quickly Reacquirable Locks" (2006).

[36] David Dice. "Brief announcement: a partitioned ticket lock". *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*. Ed. by Rajmohan Rajaraman et al. ACM, 2011, pp. 309–310.

[37] David Dice et al. "Flat-combining NUMA locks". *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*. Ed. by Rajmohan Rajaraman et al. ACM, 2011, pp. 65–74.

[38] David Dice et al. "Lock Cohorting: A General Technique for Designing NUMA Locks". *TOPC* 1.2 (2015), 13:1–13:42.

[39] Diego Didona et al. "ProteusTM: Abstraction Meets Performance in Transactional Memory". *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*. Ed. by Tom Conte et al. ACM, 2016, pp. 757–771.

[40] Open Source Facebook. *Rocksdb*. http://rocksdb.org/. 2017.

[41] FAL Labs. *Kyoto Cabinet*. http://fallabs.com/kyotocabinet/. 2011.

[42] Babak Falsafi et al. "Unlocking Energy". *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.* Ed. by Ajay Gulati et al. USENIX Association, 2016, pp. 393–406.

[43] Panagiota Fatourou et al. "Revisiting the combining synchronization technique". *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*. Ed. by J. Ramanujam et al. ACM, 2012, pp. 257–266.

[44] Rich Felker. *musl libc*. https://www.musl-libc.org/. 2017.

[45] Free Software Foundation FSF. *pthread_mutex_lock GNU C library implementation*. https://sourceware.org/git/?p=glibc.git;a=blob;f=nptl/pthread_mutex_lock.c;hb=HEAD. 2017.

[46] Free Software Foundation FSF. *The GNU C Library*. https://www.gnu.org/software/libc/manual/. 2017.

[47] Sanjay Ghemawat et al. *TCMalloc: Thread-Caching Malloc*. https://github.com/gperftools/gperftools/. 2017.

[48] Vincent Gramoli. "More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms". *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*. Ed. by Albert Cohen et al. ACM, 2015, pp. 1–10.

[49] Rachid Guerraoui et al. "Lock – Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems". *ACM Transaction on Computer System* (2018).

[50] Hugo Guiroux. *Thesis figures and tables*. https://github.com/multicore-locks/litl/blob/master/paper/journal/appendix_journal.pdf. 2018.

[51] Hugo Guiroux et al. *LiTL source code and data sets*. https://github.com/multicore-locks/. 2016.

[52] Hugo Guiroux et al. "Multicore Locks: The Case Is Not Closed Yet". *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*. Ed. by Ajay Gulati et al. USENIX Association, 2016, pp. 649–662.

[53] Pat Hanrahan et al. "A rapid hierarchical radiosity algorithm". *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1991, Providence, RI, USA, April 27-30, 1991*. Ed. by James J. Thomas. ACM, 1991, pp. 197–206.

[54] Bijun He et al. "Preemption Adaptivity in Time-Published Queue-Based Spin Locks". *High Performance Computing - HiPC 2005, 12th International Conference, Goa, India, December 18-21, 2005, Proceedings*. Ed. by David A. Bader et al. Vol. 3769. Lecture Notes in Computer Science. Springer, 2005, pp. 7–18.

[55] Danny Hendler et al. "Flat combining and the synchronization-parallelism tradeoff". *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*. Ed. by Friedhelm Meyer auf der Heide et al. ACM, 2010, pp. 355–364.

[56] IEEE. *mallopt(3) man page*. http://man7.org/linux/man-pages/man3/mallopt.3.html. 2017.

[57] IEEE. *pthread_mutex_lock(3p) man page*. http://man7.org/linux/man-pages/man3/pthread_mutex_lock.3p.html. 2017.

[58] Intel. *Intel 64 and IA-32 Architectures, Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*. 2016.

[59] Intel. *Intel Xeon Processor E7-4800/8800 v3 Product Families*. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e7-v3-datasheet-vol-1.pdf. 2015.

[60] Intel. *Introduction to the intel quickpath interconnect*. https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html. 2009.

[61] Ryan Johnson et al. "Decoupling contention management from scheduling". *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*. Ed. by James C. Hoe et al. ACM, 2010, pp. 117–128.

[62] Alain Kägi et al. "Efficient Synchronization: Let Them Eat QOLB". *Proceedings of the 24th International Symposium on Computer Architecture, Denver, Colorado, USA, June 2-4, 1997*. Ed. by Andrew R. Pleszkun et al. ACM, 1997, pp. 170–180.

[63] Anna R. Karlin et al. "Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor". *Proceedings of the Thirteenth ACM Symposium on Operating System Principles, SOSP 1991, Asilomar Conference Center, Pacific Grove, California, USA, October 13-16, 1991*. Ed. by Henry M. Levy. ACM, 1991, pp. 41–55.

[64] Sanidhya Kashyap et al. "Scalable NUMA-aware Blocking Synchronization Primitives". *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. USENIX Association, 2017, pp. 603–615.

[65] Leonidas I. Kontothanassis et al. "Scheduler-Conscious Synchronization". *ACM Trans. Comput. Syst.* 15.1 (1997), pp. 3–40.

[66] Konstantinos Koukos et al. "Towards more efficient execution: a decoupled access-execute approach". *International Conference on Supercomputing, ICS'13, Eugene, OR, USA - June 10 - 14, 2013*. Ed. by Allen D. Malony et al. ACM, 2013, pp. 253–262.

[67] Bradley C. Kuszmaul. "SuperMalloc: a super fast multithreaded malloc for 64-bit machines". *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management, ISMM 2015, Portland, OR, USA, June 13-14, 2015*. Ed. by Antony L. Hosking et al. ACM, 2015, pp. 41–55.

[68] Kaz Kylheku. *What is PTHREAD_MUTEX_ADAPTIVE_NP?* http://stackoverflow.com/a/25168942. 2014.

[69] Ecole Polytechnique de Lausanne. *Distributed Programming Laboratory (LPD)*. 2018. (Visited on 06/08/2018).

[70] Beng-Hong Lim. "Reactive synchronization algorithms for multiprocessors". PhD thesis. Massachusetts Institute of Technology, Cambridge, USA, 1995.

[71] Jean-Pierre Lozi. "Towards more scalable mutual exclusion for multicore architectures. (Vers des mécanismes d'exclusion mutuelle plus efficaces pour les architectures multi-cœur)". PhD thesis. Pierre and Marie Curie University, Paris, France, 2014.

[72] Jean-Pierre Lozi et al. "Fast and Portable Locking for Multicore Architectures". *ACM Trans. Comput. Syst.* 33.4 (2016), 13:1–13:62.

[73] Jean-Pierre Lozi et al. "Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications". *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*. Ed. by Gernot Heiser et al. USENIX Association, 2012, pp. 65–76.

[74] Jean-Pierre Lozi et al. "The Linux scheduler: a decade of wasted cores". *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*. Ed. by Cristian Cadar et al. ACM, 2016, 1:1–1:16.

[75] Victor Luchangco et al. "A Hierarchical CLH Queue Lock". *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference, Dresden, Germany, August 28 - September 1, 2006, Proceedings*. Ed. by Wolfgang E. Nagel et al. Vol. 4128. Lecture Notes in Computer Science. Springer, 2006, pp. 801–810.

[76] Peter S. Magnusson et al. "Queue Locks on Cache Coherent Multiprocessors". *Proceedings of the 8th International Symposium on Parallel Processing, Cancún, Mexico, April 1994*. Ed. by Howard Jay Siegel. IEEE Computer Society, 1994, pp. 165–171.

[77] Paul E. McKenney. "Is Parallel Programming Hard, And, If So, What Can You Do About It? (v2017.01.02a)". *CoRR* abs/1701.00854 (2017). arXiv: 1701.00854.

[78] Paul E. McKenney. "Pattern Languages of Program Design 2". Ed. by John M. Vlissides et al. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. Chap. Selecting Locking Designs for Parallel Programs, pp. 501–531.

[79] Paul E. McKenney. "Selecting Locking Primitives for Parallel Programming". *Commun. ACM* 39.10 (1996), pp. 75–82.

[80] John M. Mellor-Crummey et al. "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors". *ACM Trans. Comput. Syst.* 9.1 (1991), pp. 21–65.

[81] *Memcached*. http://memcached.org/. 2017.

[82] Thannirmalai Somu Muthukaruppan et al. "Price theory based power management for heterogeneous multi-cores". *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*. Ed. by Rajeev Balasubramonian et al. ACM, 2014, pp. 161–176.

[83] Regina Nuzzo. "Scientific method: Statistical errors". *Nature* 506.7487 (2014), pp. 150–152.

[84] Oracle Corporation. *MySQL*. https://www.mysql.com/. 2017.

[85] Y. Oyama et al. "Executing Parallel Programs with Synchronization Bottlenecks Efficiently". *Proceedings of the International Workshop on Parallel and Distributed Computing For Symbolic And Irregular Applications (PDSIA'99)*. World Scientific, 1999.

[86] Venkatesh Pallipadi et al. "The ondemand governor". *Proceedings of the Linux Symposium*. Vol. 2. 00216. sn. 2006, pp. 215–230.

[87] Lennart Poettering. *Measuring Lock Contention*. http : / / opointer . de / blog / projects/mutrace.html. 2011.

[88] Zoran Radovic et al. "Hierarchical Backoff Locks for Nonuniform Communication Architectures". *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA'03), Anaheim, California, USA, February 8-12, 2003*. IEEE Computer Society, 2003, pp. 241–252.

[89] Ravi Rajwar et al. "Speculative lock elision: enabling highly concurrent multithreaded execution". *Proceedings of the 34th Annual International Symposium on Microarchitecture, Austin, Texas, USA, December 1-5, 2001*. Ed. by Yale N. Patt et al. ACM/IEEE Computer Society, 2001, pp. 294–305.

[90] Colby Ranger et al. "Evaluating MapReduce for Multi-core and Multiprocessor Systems". *13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*. IEEE Computer Society, 2007, pp. 13–24.

[91] David P. Reed et al. "Synchronization with Eventcounts and Sequences". *Commun. ACM* 22.2 (1979), pp. 115–123.

[92] Haris Ribic et al. "Energy-efficient work-stealing language runtimes". *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*. Ed. by Rajeev Balasubramonian et al. ACM, 2014, pp. 513–528.

[93] Sepideh Roghanchi et al. "ffwd: delegation is (much) faster than you think". *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 342–358.

[94] Adrian Sampson et al. "EnerJ: approximate data types for safe and general low-power computation". *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall et al. ACM, 2011, pp. 164–174.

[95] Michael L. Scott. *Shared-Memory Synchronization*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[96] Michael L. Scott et al. "Scalable queue-based spin locks with timeout". *Proceedings of the 2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01), Snowbird, Utah, USA, June 18-20, 2001*. Ed. by Michael T. Heath et al. ACM, 2001, pp. 44–52.

[97] Jianchen Shan et al. "APPLES: Efficiently Handling Spin-lock Synchronization on Virtualized Platforms". *IEEE Trans. Parallel Distrib. Syst.* 28.7 (2017), pp. 1811–1824.

[98] Kai Shen et al. "Power containers: an OS facility for fine-grained power and energy management on multicore servers". *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*. Ed. by Vivek Sarkar et al. ACM, 2013, pp. 65–76.

[99] Karan Singh et al. "Real time power estimation and thread scheduling via performance counters". *SIGARCH Computer Architecture News* 37.2 (2009), pp. 46–55.

[100] Will Sobel et al. *Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0*. 2008.

[101] SQLite Consortium. *SQLite*. https://www.sqlite.org/. 2017.

[102] Srinath Sridharan et al. "Adaptive, efficient, parallel execution of parallel programs". *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. Ed. by Michael F. P. O'Boyle et al. ACM, 2014, pp. 169–180.

[103] 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. 2002.

[104] Herb Sutter. "The free lunch is over: A fundamental turn toward concurrency in software". *Dr. Dobb's journal* 30.3 (2005), pp. 202–210.

[105] Nathan R. Tallent et al. "Analyzing lock contention in multithreaded applications". *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*. Ed. by R. Govindarajan et al. ACM, 2010, pp. 269–280.

[106] Jons-Tobias Wamhoff et al. "The TURBO Diaries: Application-controlled Frequency Scaling Explained". *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany*. Ed. by Uwe Aßmann et al. Vol. 239. LNI. GI, 2015, pp. 141–142.

[107] Tianzheng Wang et al. "Be my guest: MCS lock now welcomes guests". *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*. Ed. by Rafael Asenjo et al. ACM, 2016, 21:1–21:12.

[108] Qiang Wu et al. "A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance". *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-38 2005), 12-16 November 2005, Barcelona, Spain*. IEEE Computer Society, 2005, pp. 271–282.

[109] Fen Xie et al. "Compile-time dynamic voltage scaling settings: opportunities and limits". *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*. Ed. by Ron Cytron et al. ACM, 2003, pp. 49–62.

[110] Chao Xu et al. "Automated OS-level Device Runtime Power Management". *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*. Ed. by Özcan Özturk et al. ACM, 2015, pp. 239–252.

[111] Konrad Zemek. *Asio, SSL, and scalability*. https://konradzemek.com/2015/08/16/asio-ssl-and-scalability/. 2015.

[112] Mingzhe Zhang et al. "Scalable Adaptive NUMA-Aware Lock". *IEEE Trans. Parallel Distrib. Syst.* 28.6 (2017), pp. 1754–1769.