Towards more scalable mutual exclusion for multicore architectures

Jean-Pierre Lozi

UPMC
SORBONNE UNIVERSITÉS

PhD Thesis, Université Pierre et Marie Curie
Whisper team, LIP6/INRIA

Subject: Computer Science
Option: Distributed Systems

Presented by: Jean-Pierre Lozi

Towards More Scalable Mutual Exclusion for Multicore Architectures

Vers des mécanismes d'exclusion mutuelle plus efficaces pour les architectures multi-cœur

To be presented on 16/07/14 in front of the following jury:

M. Luc Bouganim	INRIA Paris-Rocquencourt	Paris, France	*Examiner*
M. Tim Harris	Oracle Labs	Cambridge, UK	*Examiner*
M. Maurice Herlihy	Brown University	Providence, RI, USA	*Examiner*
M. Gilles Muller	Univ. Pierre et Marie Curie (LIP6/INRIA)	Paris, France	*Advisor*
M. Vivien Quéma	INP / ENSIMAG (LIG)	Grenoble, France	*Reviewer*
M. Wolfgang Schröder-Preikschat	Friedrich-Alexander-Universität	Erlangen, Germany	*Reviewer*
M. Gaël Thomas	Univ. Pierre et Marie Curie (LIP6/INRIA)	Paris, France	*Advisor*

Abstract

The scalability of multithreaded applications on current multicore systems is hampered by the performance of lock algorithms, due to the costs of access contention and cache misses. The main contribution presented in this thesis is a new lock algorithm, Remote Core Locking (RCL), that aims to improve the performance of critical sections in legacy applications on multicore architectures. The idea of RCL is to replace lock acquisitions by optimized remote procedure calls to a dedicated hardware thread, which is referred to as the *server*. RCL limits the performance collapse observed with other lock algorithms when many threads try to acquire a lock concurrently and removes the need to transfer lock-protected shared data to the hardware thread acquiring the lock because such data can typically remain in the server's cache.

Other contributions presented in this thesis include a profiler that identifies the locks that are the bottlenecks in multithreaded applications and that can thus benefit from RCL, and a reengineering tool developed with Julia Lawall that transforms POSIX locks into RCL locks. Eighteen applications were used to evaluate RCL: the nine applications of the SPLASH-2 benchmark suite, the seven applications of the Phoenix 2 benchmark suite, Memcached, and Berkeley DB with a TPC-C client. Eight of these applications are unable to scale because of locks and benefit from RCL on an x86 machine with four AMD Opteron processors and 48 hardware threads. Using RCL locks, performance is improved by up to 2.5 times with respect to POSIX locks on Memcached, and up to 11.6 times with respect to Berkeley DB with the TPC-C client. On an SPARC machine with two Sun Ultrasparc T2+ processors and 128 hardware threads, three applications benefit from RCL. In particular, performance is improved by up to 1.3 times with respect to POSIX locks on Memcached, and up to 7.9 times with respect to Berkeley DB with the TPC-C client.

Keywords. Multicore, synchronization, lock, combining, RPC, locality, busy-waiting, memory contention, profiling, reengineering.

Acknowledgments

I would like to thank my advisors Gilles Muller and Gaël Thomas, as well as Julia Lawall for their help, support and reactivity during all of my PhD. Working with them has been a pleasure and a very positive experience. I would also like to thank the rest of the Regal and Whisper teams. In particular, I would like to thank other PhD students from room 25-26/231 without whom Paris would not have been such an enjoyable experience during my PhD years.

A big thanks Michael Scott for having kindly let me use the Niagara2-128 machine at the University of Rochester for more than a year, as well as Oracle, from which the machine was a gift. I would also like to thank James Roche for having been so quick to reboot it when needed.

A special thanks to Alexandra Fedorova from Simon Fraser University for her insight without which publishing a paper to USENIX ATC [71] would have been impossible, and to Vivien Quéma and Wolfgang Schröder-Preikschat for their constructive comments that helped improve this thesis.

Finally, finishing this PhD would not have been possible without the incredible support I received from my family during the past four years.

Preface

This thesis presents the main research that I conducted in the Whisper (formerly Regal) team at Laboratoire d'Informatique de Paris 6 (LIP6), to pursue a PhD in Computer Science from the doctoral school "École Doctorale Informatique, Télécommunications et Électronique" (EDITE) in Paris. My PhD advisors were Gilles Muller and Gaël Thomas (LIP6/INRIA).

Research presented in this thesis. The main focus of the research presented in this thesis was the design of better techniques to ensure mutual exclusion on multicore architectures. My main contribution, presented in Chapter 4, is the design of a new lock algorithm, named Remote Core Locking (RCL), that dedicates one or several hardware threads for the serial execution of critical sections. This work led to the following publications, at a French conference and an international one:

- ***Le Remote Core Lock (RCL): une nouvelle technique de verrouillage pour les architectures multi-cœur.*** Jean-Pierre Lozi. *8ème Conférence Française en Systemes d'Exploitation (CFSE '8).* Saint-Malo, France, 2011. Best Paper award. [69]

- ***Remote Core Locking: Migrating Critical-section Execution to Improve the Performance of Multithreaded Applications.*** Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall and Gilles Muller. *In Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC '12).* Boston, USA, 2012. [71]

An INRIA research report [72] was also produced. Finally, my work was presented at many occasions. In particular, I presented a poster at EuroSys 2011 in Salzburg, Austria, and participated to the Work in Progress (WiP) session at SOSP'11 in Cascais, Portugal.

Other research. As a member of the Whisper team, I was also involved in other projects. In particular, I worked on EHCtor/Hector, a tool that makes it possible to detect resource-release omission faults in kernel and application code: I helped analyze the reports generated by the tool, and I devised techniques to exploit some of the bugs it found. This research has led to the two following publications:

- ***EHCtor: Detecting Resource-Release Omission Faults in Error-Handling Code for Systems Software.*** Suman Saha, Jean-Pierre Lozi. *9ème Conférence Française en Systemes d'Exploitation (CFSE '9).* Grenoble, 2013. [94]

- ***Hector: Detecting Resource-Release Omission Faults in Error-Handling Code for Systems Software.*** Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia Lawall, and Gilles Muller. *In Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13).* Budapest, 2013. Best Paper award. [95]

In the context of that work, I submitted a bug report to the PHP developers [105, 70] that describes a possible exploit for a memory leak that was found with Hector. I wrote a bug fix that was accepted by the PHP team and is part of the codebase since PHP 5.4.5.

Since my work on EHCtor/Hector was only secondary to my main research, it is not described further in this thesis: the reader is invited to refer to the aforementioned papers for more information.

Contents

Chapter 1

Introduction

Until the early 2000's, the performance of Central Processing Units (CPUs) had been steadily improving for decades thanks to rising hardware clock frequencies. However, due to physical limitations, CPU manufacturers have now found it impossible to keep increasing clock frequencies and instead focus on embedding several execution units in the same CPU. These execution units are referred to as *cores*, and technologies such as Simultaneous Hyper-Threading (SMT) replicate some parts of these cores in order to make them run several *hardware threads* simultaneously. The number of hardware threads on consumer CPUs keeps increasing in all computing devices, from servers to mobile phones. It is not uncommon nowadays for multiprocessor servers to include dozens of hardware threads.

The main downside of manufacturers increasing the number of hardware threads in CPUs instead of increasing their frequency is that applications do not see their performance automatically improve as newer and faster CPUs are released. Harvesting the performance of modern multicore architectures is difficult because most programs cannot be fully parallelized, and Amdahl's Law [4] shows that as the number of hardware threads increases, the critical path, i.e., portions of the application's source code that cannot be parallelized, ends up being the limiting factor. This is why, in order to use multicore architectures efficiently, it is important to focus on shortening the critical path. In most legacy applications, a large part of the critical path consists of *critical sections*, i.e., sections of code that are to be run in mutual exclusion.[1] Critical sections are usually protected by *locks*, i.e., synchronization mechanisms that make it possible for several threads to ensure mutual exclusion for the execution of sections of code that access the same resources.

In order to fully exploit the processing power of recent multicore machines, legacy applications must be optimized to exploit parallelism efficiently, which can be extremely complex and requires a lot of work: while making legacy applications scale up to a few hardware threads is relatively simple with naïve, coarse-grained parallelization, getting them to scale on newer multicore architectures with dozens of hardware threads remains a challenge. Since a large number of very complex applications has been developed over the previous decades without taking these architectures into account, rewriting large parts of the legacy codebase in order to harvest the performance of modern multicore architectures is a very costly task that will take several years.

One way of getting legacy applications to scale on multicore architectures is to shorten the critical path by reducing the size of critical sections, i.e., to use *fine-grained locking* [6, 55]:

[1] The critical path can also contain other parts such as program initialization and finalization as well as transitions between loop-parallel program sections. These parts are identified as *data management housekeeping* and *problem irregularities* by Amdahl [4].

while this approach is generally regarded as being very efficient, it can be extremely complex for programmers. There is no general technique that makes it possible to reduce the size of critical sections, instead, developers must use different approaches for each one of them, thereby increasing the complexity of the concurrent logic of the application and increasing the probability of introducing hard-to-find concurrency bugs. It is also possible to avoid the use of critical sections in some cases and to rely on atomic instructions for synchronization instead, by using *lock-free* algorithms and data structures [56, 77, 52, 40, 62, 63]. While this approach can be very efficient in some cases and efficient lock-free algorithms have been proposed for most standard data structures such as stacks, queues, and skiplists, it is not possible to replace any set of critical sections with an efficient lock-free algorithm. Finally, using *transactional memory*, either implemented in software or hardware, has been proposed as an alternative to locking [61, 54, 98, 53, 45]. While transactional memory can be easier to use than locks, it is currently not widely used due to poor hardware support and weak performance of software implementations. Moreover, with transactional memory, critical sections cannot perform any operation that cannot be undone, including most I/O.

Other solutions have been proposed to harvest the performance of multicore architectures. Some approaches simply focus on improving the implementation of one specific mechanism on multicore architectures, like Remote Procedure Calls (RPC) [10, 11, 42], but these techniques are too limited to make it possible to harvest the processing power of multicore machines by legacy applications in the general case. Some specific improvements to parts of the operating system such as the scheduler have been proposed for multicore architectures [64], but they do not remove the need to modify applications. Whole new operating systems designs have also been proposed, such as Opal [22], Corey [13], Multikernel [9], and Helios [78], but legacy operating systems have become so complex and feature-rich that switching to completely new operating systems now would come at a major redevelopment cost and legacy applications would need to be completely rewritten. Some tools have been proposed to help with the redesign of applications on multicore architectures, such as profilers [87, 65], which can be useful to detect bottlenecks. Profilers do not aim to fix these bottlenecks however, other techniques have to be used once the causes of the lack of scalability have been identified.

Another way to improve the performance of multithreaded applications on multicore architectures is to improve the way locks are implemented. The main advantage of this technique is that it does not require a complete redesign of applications. Over the last twenty years, a number of studies [2, 8, 12, 49, 51, 59, 75, 96, 100, 102] have attempted to optimize the performance of locks on multicore architectures, either by reducing access contention or by improving cache locality. Access contention occurs when many threads simultaneously try to enter critical sections that are protected by the same lock, thereby saturating the memory bus with messages from the underlying cache-coherence protocol in order to reach an agreement as to which thread should obtain the lock first. The lack of cache locality becomes a problem when a critical section accesses shared data that has recently been written by another hardware thread, resulting in cache misses, which greatly increase the critical section's execution time. Addressing access contention and cache locality together remains a challenge. These issues imply that some applications that work well on a small number of hardware threads do not scale to the number of hardware threads found in today's multicore architectures.

Recently, several approaches have been proposed to execute a succession of critical sections on a single *server* (or *combiner*) hardware thread to improve cache locality [51, 102, 39]. Such approaches also incorporate a fast transfer of control from other *client* hardware threads to the server, to reduce access contention. Suleman et al. [102] propose a hardware-based solution,

evaluated in simulation, that introduces new instructions to perform the transfer of control, and uses a hardware thread from a special fast core to execute critical sections. Software-only algorithms in which the server is an ordinary client thread and the role of server is handed off between clients periodically have also been proposed [82, 51, 39]. These algorithms are referred to as *combining locks*. Combining locks are faster than traditional locks, but they sometimes incur an overhead for the management of the server role and the list of threads, and they are vulnerable to preemption. Furthermore, neither Suleman et al.'s algorithm nor combining locks propose a mechanism to handle condition variables, which makes them unable to support many widely used applications.

The objective of the research presented in this thesis is to focus on decreasing the time spent by applications in critical sections in such a way that avoids redesigning whole applications, by focusing on reducing the time to enter critical sections and improving their memory access locality. The main contribution presented in this thesis is a new locking technique, Remote Core Locking (RCL), that aims to improve the performance of legacy multithreaded applications on multicore hardware by executing critical sections that are protected by highly contended locks on one or several dedicated server hardware threads. In particular, RCL targets legacy server applications that run on modern multicore servers. It is entirely implemented in software and supports x86 and SPARC multicore architectures. At the basis of RCL is the observation that most applications do not scale to the number of hardware threads found in modern multicore architectures, and thus it is possible to *dedicate* the hardware threads that do not contribute to improving the performance of the application to serving critical sections. It is therefore not necessary to burden the application threads with the role of server, as done in combining locks. The design of RCL addresses both access contention and locality. Contention is solved by a fast transfer of control to a server, using a dedicated cache line for each client to achieve busy-wait synchronization with the server hardware thread. Locality is improved because shared data is likely to remain in the server hardware thread's cache, allowing the server to access such data without incurring cache misses. In this, RCL is similar to combining locks, but it has a lower overall overhead, it resists better to preemption because the dedicated server thread always makes progress, and it proposes a mechanism to handle condition variables, which makes it directly usable in real-world applications. RCL is well-suited to improve the performance of a legacy application in which contended locks are an obstacle to performance, since using RCL enables improving resistance to contention and locality without requiring a deep understanding of the source code. On the other hand, modifying locking schemes to use fine-grained locking, lock-free algorithms or transactional memory is time-consuming, requires an overhaul of the source code, and does not improve locality.

Other contributions presented in this thesis include a methodology along with a set of tools to facilitate the use of RCL in legacy applications. Because RCL serializes critical sections associated with locks managed by the same server hardware thread, transforming all locks into RCLs on a smaller number of servers induces *false serialization*: some servers serialize the execution of critical sections that are protected by different locks and therefore do not need to be executed in mutual exclusion. In some cases, false serialization can introduce a significant overhead. Therefore, the programmer must first decide which lock(s) should be transformed into RCLs and which server(s) handle which lock(s). A profiler was written for this purpose. It is designed to identify which locks are frequently used by the application, to measure how much time is spent on locking, and to measure how good the data locality of critical sections is. Based on this information, a set of simple heuristics are proposed to help the programmer decide which locks must be transformed into RCLs. An automatic reengineering tool for C programs was

designed with the help of Julia Lawall in order to transform the code of critical sections so that it can be executed as a remote procedure call on the server hardware thread: the code within a critical section must be extracted as a function. The argument passed to that function will be its *context* object, i.e., an object that contains copies of all variables referenced or updated by the critical section that are declared in the function containing the critical section code. RCL takes the form of a runtime for Linux and Solaris that is compatible with POSIX threads, and that supports a mixture of RCL and POSIX locks in a single application.

The performance of RCL is compared to other locks with a custom microbenchmark which measures the execution time of critical sections that access a varying number of shared memory locations. Furthermore, based on the results of the profiler, three benchmarks from the SPLASH-2 [107, 99, 110] suite, three benchmarks in the Phoenix 2 [101, 103, 112, 92] suite, Memcached [26, 41], and Berkeley DB [80, 79] with a TPC-C benchmark developed at Simon Fraser University were identified as applications that could benefit from RCL. In each of these applications, RCL is compared against a basic spinlock, the standard POSIX lock, MCS [75], and Flat Combining [51]. RCL is also compared with CC-Synch and DSM-Synch [39], two state-of-the-art algorithms that were designed concurrently with RCL, and therefore were not included in the evaluation of the RCL paper that was published at USENIX ATC [71]. Comparisons are made for a same number of hardware threads, which means that there are fewer application threads in the RCL case, since one or more hardware threads are dedicated to RCL servers.

RCL is evaluated on two machines: (i) Magnycours-48, an x86 machine with four AMD Opteron CPUs and 48 hardware threads running Linux 3.9.7, and (ii) Niagara2-128, a SPARC machine with two Ultrasparc T2 CPUs and 128 hardware threads running Solaris 10. Key highlights of the results are:

- On a custom microbenchmark, under high contention, RCL is faster than all other evaluated approaches: on Magnycours-48 (resp. Niagara2-128), RCL is 3.2 (resp. 1.8) times faster than the second best approach, CC-Synch, and 5.0 (resp. 7.2) times faster than the POSIX lock.

- On other benchmarks, contexts are small, and thus the need to pass a context to the server has only a marginal performance impact.

- On most benchmarks, only one lock is frequently used and therefore only one RCL server is needed. The only exception is Berkeley DB with the TPC-C client, which requires two or three RCL servers to reach optimal performance by reducing false serialization.

- On Magnycours-48 (resp. Niagara2-128), RCL improves the performance of five (resp. one) application(s) from the SPLASH-2 and Phoenix 2 benchmark suites more than all other evaluated locks.

- For Memcached with Set requests, on Magnycours-48 (resp. Niagara2-128), RCL yields a speedup of 2.5 (resp. 1.3) times over the POSIX lock, 1.9 (resp. 1.2) times over the basic spinlock and 2.0 (resp. 1.2) times over MCS. The number of cache misses in critical sections is divided by 2.9 (resp. 2.3) by RCL, which shows that it can greatly improves locality. Combining locks were not evaluated in this experiment because they do not implement condition variables, which are used by Memcached.

- For Berkeley DB with the TPC-C client, when using Stock Level transactions, on Magnycours-48 (resp. Niagara2-128) RCL yields a speedup of up to 11.6 (resp. 7.6) times over the original Berkeley DB locks for 48 (resp. 384) simultaneous clients. RCL resists better than

other locks when the number of simultaneous clients increases. In particular, RCL performs much better than other locks when the application uses more client threads than there are available hardware threads on the machine, even when other locks are modified to yield the processor in their busy-wait loops.

Organization of the document. The thesis is structured as follows:

- Chapter 2 focuses on multicore architectures. It presents the general design of these architectures and describes the most common bottlenecks they suffer from. The two machines used in the evaluations are also described in that chapter.
- Chapter 3 presents the evolution of lock algorithms. Detailed algorithms of all locks that are used in the evaluation in Chapter 5 are presented in that chapter, as a reference.
- Chapter 4 presents the main contributions of the research work presented in this thesis, namely, RCL and its implementation, the profiler that makes it possible to identify which applications and locks can benefit from RCL, and the reengineering tool that automatically transforms applications so that they can be used with RCL.
- Chapter 5 presents an evaluation of RCL's performance. First, a microbenchmark is used to obtain a first estimate of the performance of RCL as well as that of some of the locks presented in Chapter 3. Then, the profiler designed to help decide when using RCL would be beneficial for an application is presented. Finally, using the results of the microbenchmark combined with the results of the profiler presented in Chapter 4, a set of applications that are likely to benefit from RCL is identified, and RCL as well as other locks are evaluated with the applications from that set.
- Finally, Chapter 6 concludes the thesis and considers future research directions.

Chapter 2

Multicore architectures

This chapter presents multicore architectures and their bottlenecks. Section 2.1 quickly presents the various components of a multicore architecture. Section 2.2 presents hardware threads, i.e., the minimal execution units of multicore machines. Section 2.3 describes how hardware threads communicate with each other through the CPU caches and the RAM. Section 2.4 presents heterogeneous multicore architectures, i.e., architectures that use various cores with different characteristics. Section 2.5 presents the machines used in the evaluation. Finally, Section 2.6 concludes the chapter.

2.1 Overview

Historically, most CPUs contained a single processing core, with a single hardware thread, and manufacturers mainly improved the performance of CPUs by increasing their clock speed, which went from a few megahertz in the early 1980's to several gigahertz twenty years later. However, in the early 2000's, increasing the CPU clock speed became increasingly difficult due to power dissipation, which makes CPUs with high clock speeds consume too much energy and overheat. Manufacturers instead switched to bundling several processing cores into CPUs in order to keep increasing overall processing performance, even though exploiting the computing power of multicore architectures requires parallelizing applications efficiently, whereas clock speed increases automatically improved the performance of all software.

A typical, current, consumer multicore machine is shown in Figure 2.1. It can have one or several *Central Processing Units* (*CPUs*, two in the figure). Each CPU can have one or several *dies* (four in the figure), and each die contains one or several *cores* (16 in the figure). Some machines only run one hardware thread per core, while others use hardware multithreading to run several hardware threads in parallel (64 in the figure). Communication between hardware threads on the same core is typically ensured by low-level CPU caches (L1 or L2, L1 in the figure). Communication between cores is typically ensured by high-level CPU caches (L2 or L3, L2 in the figure). Communication between dies and/or CPUs is ensured by a data bus. Nowadays, buses are most frequently implemented in the form of a point-to-point interconnect. In Figure 2.1, all dies are directly connected to each other, but that is not always the case: sometimes, several hops are needed for the communication between two dies. All dies are connected to the Random Access Memory (RAM) via their memory controller. In Uniform Memory Access (UMA) architectures, accessing the memory from any of the dies comes at the same cost (latency and speed). The

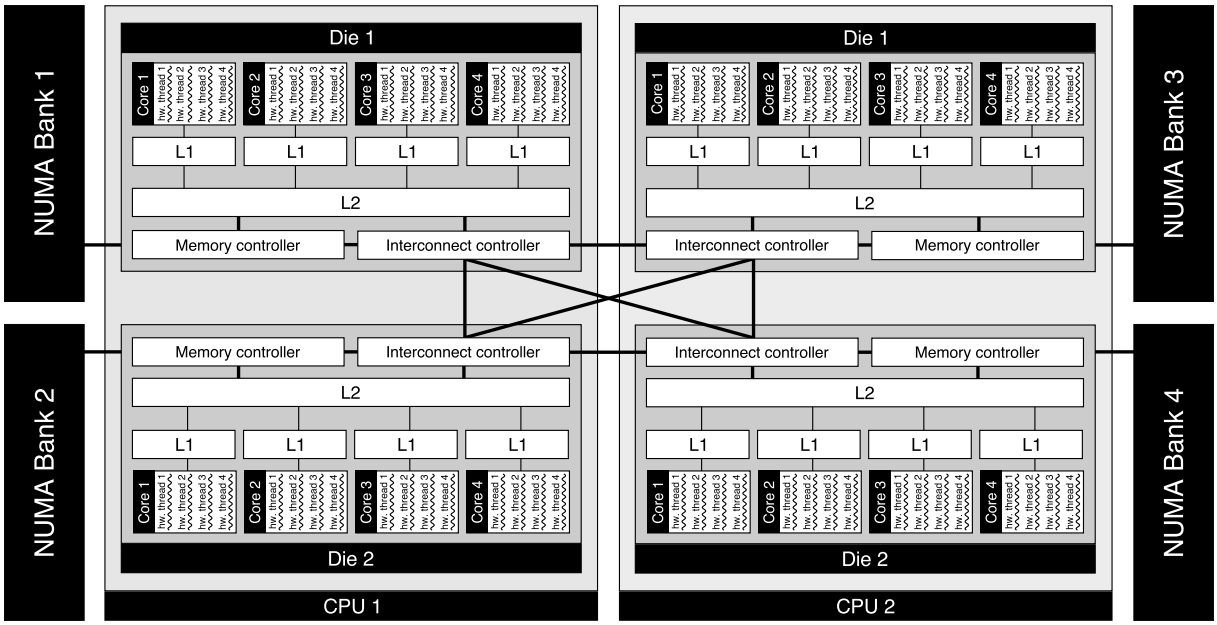


Figure 2.1: Example of a multicore architecture

architecture presented in Figure 2.1 uses Non-Uniform Memory Access, because the memory is split in different memory banks, and dies access local memory banks faster than remote memory banks.

Moreover, the machine presented in Figure 2.1 is *cache-coherent*, which means that all of its memory is directly addressable, and the underlying cache-coherence mechanism, implemented in hardware, transparently fetches data in the right memory bank or CPU cache to bring it to the hardware thread that requests it. Some new architectures are now *non-cache-coherent*, which means that each core or CPU can only access its own local memory, and explicit messages must be used by the software layer to transfer data from one core to the other. However, non-cache-coherent architectures are not very commonly used yet. Section 2.3.1 focuses on cache-coherent vs. non-cache-coherent architectures in more detail.

2.2 Hardware threads

Until the early 2000's, most consumer CPUs were only able to run a single software thread at any given time, i.e., they only provided a *single hardware thread*. Multitasking was handled by schedulers, which used time-sharing to enable several software threads to use the CPU concurrently. The first technique that was commonly used to introduce parallelism in CPUs was *instruction pipelines*: each instruction is broken up into a number of steps, and different steps from consecutive instructions are executed in parallel [93]. With the introduction of multicore CPUs, the parallelism provided by the multiple cores was combined with the parallelism provided by the instruction pipeline, in order to improve performance.

In many multicore architectures, several software threads can be running at a given time if they are placed on different cores. Moreover, in order to increase parallelism even more, some CPU manufacturers do not only bundle several CPU cores into a single CPU, they also replicate some parts of the cores in order to make each one of them execute several software threads simultaneously: from the point of view of the developer, each core provides *multiple hardware threads* that are completely independent execution units, and one software thread can be running

on each of these hardware threads at any given time [106]. In practice, however, two hardware threads running concurrently on the same core may slow down each other more than if they were running on different cores, because they share resources such as, for instance, their Arithmetic and Logic Unit (ALU). The idea of hardware multithreading is to increase parallelism at a lower cost than by adding more cores, because some components that are not usually bottlenecks can be shared by several hardware threads.

Each hardware thread must have access to the following components, some of which possibly being shared: (i) A set of registers that store the data that is currently being used by the hardware thread (generally not shared), (ii) an arithmetic and logic unit, often completed with a Floating Point Unit (FPU) for operations on floating point numbers, (iii) a Transaction Lookaside Buffer (TLB), which is a cache that is used to accelerate the translation of virtual memory addresses into physical addresses, and (iv) elements that are located outside CPU cores and sometimes shared by several cores, as shown in Figure 2.1, such as data and instruction CPU caches, or memory/interconnect controllers. Hardware multithreading can be implemented using three techniques: coarse-grained multithreading, fine-grained multithreading, and simultaneous multithreading.

Coarse-grained multithreading. Also called *block multithreading* or *cooperative multithreading*, coarse-grained multithreading lets a thread run until it is blocked by an event that causes a long enough stall, such as a cache miss or a page fault. The thread will not be scheduled again until the data or signal it was waiting for has arrived. Each hardware thread must have its own set of data and control registers, so that the CPU can switch between hardware threads in a single CPU cycle.

Fine-grained multithreading. Also called *interleaved multithreading*, with fine-grained multithreading, the CPU starts executing an instruction from a different hardware thread at each cycle, in a round-robin fashion. Since instructions from several threads are executed in parallel in the pipeline, each stage in the pipeline must track which thread's instruction it is processing. Moreover, since more threads are executed in parallel than with coarse-grained multithreading, shared resources such as the TLB and CPU caches must be larger so that they do not become bottlenecks.

Simultaneous multithreading (SMT). It is the most advanced implementation of hardware multithreading, and it is designed for superscalar CPUs. In a traditional superscalar CPU with one hardware thread, several instructions are issued from a single thread at each CPU cycle. With SMT, CPUs issue several instructions from multiple threads at each CPU cycle. This requires to track which thread's instruction is being processed for each thread at each stage of the pipeline. However, SMT has the advantage to use issue slots better than traditional superscalar processors, because single threads only have a limited amount of instruction-level parallelism, whereas multiple threads are typically independent from each other. SMT is used in some Intel (HyperThreading) and Sun/Oracle CPUs. In particular, SMT is used by the UltraSPARC T2+, the CPU used by Niagara2-128, one of the machines described in Section 2.5.2 and used in the evaluation in Chapter 5.

2.3 Communication between hardware threads

This section describes the means by which hardware threads communicate with each other. Section 2.3.1 describes CPU caches and how they are used for communicating between hardware threads. It also presents cache-coherent and non-cache-coherent architectures. Section 2.3.2 focuses on NUMA architectures.

2.3.1 CPU caches

An overview of CPU caches is given in Section 2.3.1.1. Cache-coherent and non-cache-coherent architectures are described in Sections 2.3.1.2 and 2.3.1.3, respectively.

2.3.1.1 Overview

CPU caches (simply referred to as "caches" henceforth) are fast components that store small amounts of data closer to CPU cores in order to speed up memory access. The data stored in caches can either be duplicates of data stored in the RAM and that is expected to be used soon by a hardware thread, or values that have been recently computed and that will be flushed back to the RAM later. As shown in Figure 2.1, caches are usually organized in a *hierarchy*, with typically two to three levels (L1, L2 and L3 caches). The farther caches are from CPU cores, the larger and slower they are. The unit of addressable data in a cache is a *cache line*, which means that each time data is transfered to a cache, the minimum of data that can be transfered is the size of the cache line for that cache (a typical cache line size in current architectures could be 64 or 128 bytes). When a hardware thread tries to read or write data that is not available in its lowest level cache, a *cache miss* is triggered, and the cache line is fetched from higher level caches or from the RAM. Caches can either store instructions, data, or both.

Replacement policy. In the case of a cache miss, the cache may have to evict one of the cache lines to make room for the newly fetched cache line. There are many possible policies to determine which cache line should be evicted from the cache. For instance, the *Least Recently Used (LRU)* or the *Least Frequently Used (LFU)* cache line can be evicted.

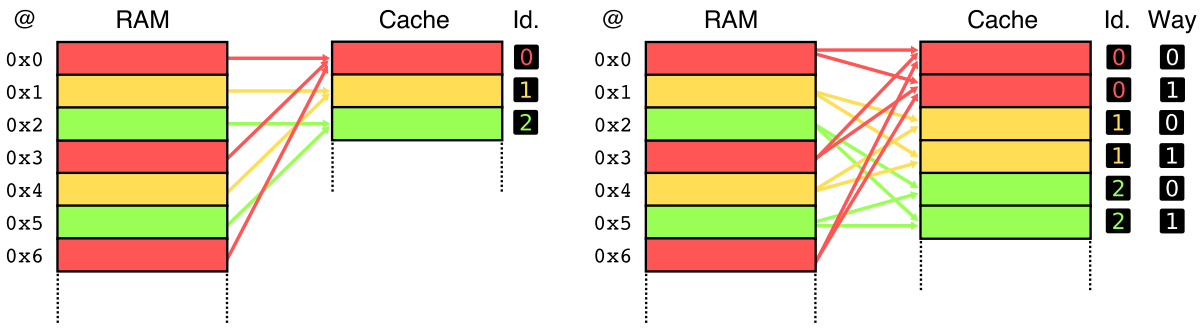
Write policy. When data is written into a cache line that is present in the cache (*write hit*), it must be flushed into the main memory. There are two main approaches:

- With a *write-through* policy, a write to the cache causes a write to the main memory.
- With a *write-back* policy, writes to the main memory are delayed: cache lines that have been written over in the cache are marked *dirty*, and dirty cache lines are written to the main memory when they are evicted from the cache (following the replacement policy).

If a hardware thread needs to write data to a cache line that is not present into the cache (*write miss*), two main approaches are possible:

- With a *no-write-allocate* policy, the cache line is not loaded into the cache, instead, the data is directly written into the main memory.
- With a *write-allocate* policy, the cache line is loaded into the cache, then overwritten into the cache with the new data (*write hit*).

Any pair of write-hit and write-miss policies is functional, but typically, most caches use either write-through combined with no-write-allocate, or write-back combined with write-allocate. However, many different types of caches exist that use variations of the policies listed above. Moreover, write policies are more complicated on cache-coherent architectures, because the same cache line may be stored in the caches of several cores, and some caches may not hold the most recent version of the data. This issue is discussed in more detail in Section 2.3.1.2.



(a) Direct mapped cache

(b) 2-way set associative cache

Figure 2.2: CPU cache associativity

Associativity. When a cache line is inserted into the cache, the *replacement policy* determines which cache line is evicted and will be replaced with the new cache line. If the replacement policy may evict any cache line, regardless of the address it maps to (by picking the *least recently used* cache line, for instance), the cache is said to be *fully associative*, as illustrated in Figure 2.2a. If, on the contrary, a memory location may only be mapped to one location in the cache, determined by its address, the cache is said to be *direct mapped*. Typically, the replacement policy can pick a location in the cache among a set of n cache lines, the cache is said to be *n-way associative*. Figure 2.2b illustrates the behavior of a n-way set associative cache for $n = 2$: each element in memory can be stored in two different locations in the cache. Increasing n decreases the number of cache misses, because if multiple memory locations that map to the same address in the cache are used in a short time period, several of them will be able to fit in the cache. On the other hand, n entries in the cache must be checked in order to determine whether a cache line is present in the cache, therefore, increasing n induces a overhead. In n-way set associative caches, a popular replacement policy is *Pseudo-LRU (PLRU)*: if a cache line must be inserted into a set s, the least recently used element of s is evicted.

Prefetching data. In order to reduce the number of cache misses, CPUs try to predict which data will be used in the near future and load it in cache before it is accessed. Modern CPUs include a data prefetching unit that performs this job. The data prefetching unit performs well when an application has regular access patterns (stride accesses such as scanning an array, or following pointers in linked-list traversals). Even though a lot of memory access patterns are too complex to be recognized by the data prefetching unit, they rarely hinder performance [60]. Modern processors also prefetch instructions, which can be a complex task due to branches in programs: the instruction prefetch is sometimes part of a complex branch prediction algorithm.

2.3.1.2 Cache-coherent architectures

On multicore architectures, each core has its own local caches. If different caches hold different versions of the same data, two hardware threads may not have the same view of the global memory, and it may be hard for hardware threads to ensure that the data they are reading is up-to-date. To prevent this, most CPUs nowadays are *cache-coherent*, which means that even though cores have their own local caches with possibly different versions of the same data, a *cache-coherence protocol* is implemented in the hardware in order to make sure that different hardware threads have a consistent view of the global memory.

An architecture is said to be cache-coherent if the following conditions are met [86]:

- If a hardware thread t writes then reads from a location l, with no writes from other hardware threads at the same location between the write and the read operation, then t must read the value it wrote at the location l.
- A read from a hardware thread t_1 to a location l that was previously written by t_2 must return the value that t_2 wrote at that location if enough time passed between the read and the write operation, and if no other hardware thread wrote data at l between the read and the write operation.
- Writes to the same location are serialized, i.e., two writes to the same location by any two hardware threads are seen in the same order by all hardware threads.

Section 2.3.1.2.a describes *cache-coherence protocols* that ensure that all hardware threads have a consistent view of the global memory. Section 2.3.1.2.b presents instructions that facilitate synchronization between hardware threads. Finally, Section 2.3.1.2.c gives a quick overview of common bottlenecks on cache-coherent architectures.

2.3.1.2.a Cache-coherence protocol

In cache-coherent architectures, hardware threads typically rely on the cache-coherence protocol for communication: when a hardware thread t_1 needs to send a value to a hardware thread t_2, it writes the value at a known memory address, and t_2 reads it later, possibly busy-waiting for the value at that address to be modified by t_1. Most cache-coherence protocols that are used in current multicore machines are based on the MESI protocol [85]. The name MESI comes from the four possible *states* it defines for cache lines: Modified, Exclusive, Shared and Invalid. These states can be described thus:

- **Modified.** The data in the cache line has been modified locally (i.e., it is *dirty*), and *only* resides in this cache. The copy in the main memory is is not up to date, therefore, if the cache line is evicted or changes its state, its data must be flushed to the main memory.
- **Exclusive.** The data in the cache line is unmodified (i.e., it is *clean*), and it does not reside in any other cache.
- **Shared.** The data in the cache line is clean but other copies may reside in other caches.
- **Invalid.** The cache line does not contain valid data. This typically happens when a shared cache line was modified in one of the caches: other copies of the data got *invalidated* by the cache-coherence protocol.

While the MESI protocol is functional, its performance is not optimal on architectures with a large number of cores because of its communication overhead. In particular, the MESI protocol may send many high-latency messages that contain redundant data: if a cache requests data that resides in many different caches (Shared state), all caches may send the same data to the requesting cache, which results in wasted bandwidth. Another drawback of the MESI protocol is that the only way for data from a Modified cache line to be accessed by remote hardware threads is to flush that data to the main memory and fetch it again, when fast cache-to-cache communications should be sufficient. Improved versions of the MESI protocol, with more states, have been implemented to solve these issues. Two widely-used variants are the MESIF protocol and the MOESI protocol.

MESIF protocol. The MESIF protocol has been used by Intel CPUs since the Nehalem architecture. It adds a Forwarded state to the MESI protocol and modifies its Shared state. The Forwarded (F) state is a variant of the Shared state that expresses the fact that the cache should act as the designated responder for that cache line. At most one cache holds a copy of data in the Forwarded state. If a cache requests data that exists in various caches, and if one of the caches holds a copy of the data that is in the Forwarded state, only that cache will send the data: no redundant messages are sent. If no version of the data is in the Forwarded state, the data will be fetched from the main memory, which may induce a overhead. This can happen if a cache line that was in the Forwarded state was evicted. To avoid this issue, the most recent requester of the data is automatically assigned the Forwarded state, which decreases the risk of Forwarded cache lines getting evicted.

MOESI protocol. The MOESI protocol is used by AMD and Sun/Oracle CPUs. It adds an Owned state and modifies the Shared state of the MESI protocol. The Owned (O) state expresses the fact that the cache holds one of the copies of a cache line (as with the Shared state) and has the exclusive right to modify it. All modifications to that cache line must be broadcast to all other caches that own it (in the Shared state): this allows for direct core-to-core communication without going through the main memory. An Owned cache line may change its state to Modified after invalidating all shared copies, and it may change its state to Shared after flushing the data to memory. The semantics of the Shared state in the MOESI protocol are modified: unlike with the MESI protocol, a Shared cache line may hold invalid data if an Owned cache line holds the correct, most recent version of the data. The Owned cache line is responsible for eventually flushing its data to the main memory. If no Owned cache line holds the data, the Shared cache line holds data that is ensured to be valid. Shared cache lines may change their state to Exclusive or Modified after invalidated all other shared copies.

As shown in Figure 2.3, with the MOESI protocol, when two hardware threads from the same die communicate together, all they need to use is their local cache and the minimum subset of caches that they share. When hardware threads from different dies or CPUs communicate, the data must go through the interconnect, but no access to the main memory is needed.

2.3.1.2.b Instructions used for synchronization

While, on cache-coherent architectures, reading and writing data to shared memory locations is sufficient for basic communication between hardware threads, specific instructions are sometimes needed for more complex synchronization schemes. First, because CPUs automatically reorder independent instructions, some specific instructions can be used to ensure system-wide ordering constraints between read and write operations of different hardware threads: these instructions are known as *memory barriers.* Second, it is sometimes useful to execute several operations in a way that appears atomic to other hardware threads. This can be done thanks to *atomic instructions.*

Memory barriers. Modern CPUs use *out-of-order* execution, i.e., they may reorder instructions to improve performance: instructions that can be instantly executed are sometimes executed before earlier instructions that would cause a stall waiting for their input data. While out-of-order execution is completely transparent in architectures that provide a single hardware thread, it can cause unpredictable behavior in architectures that provide several hardware threads: the reordering of instructions is designed to be transparent for the hardware thread that executes them, but other hardware threads see the side effects of these instructions in the real order

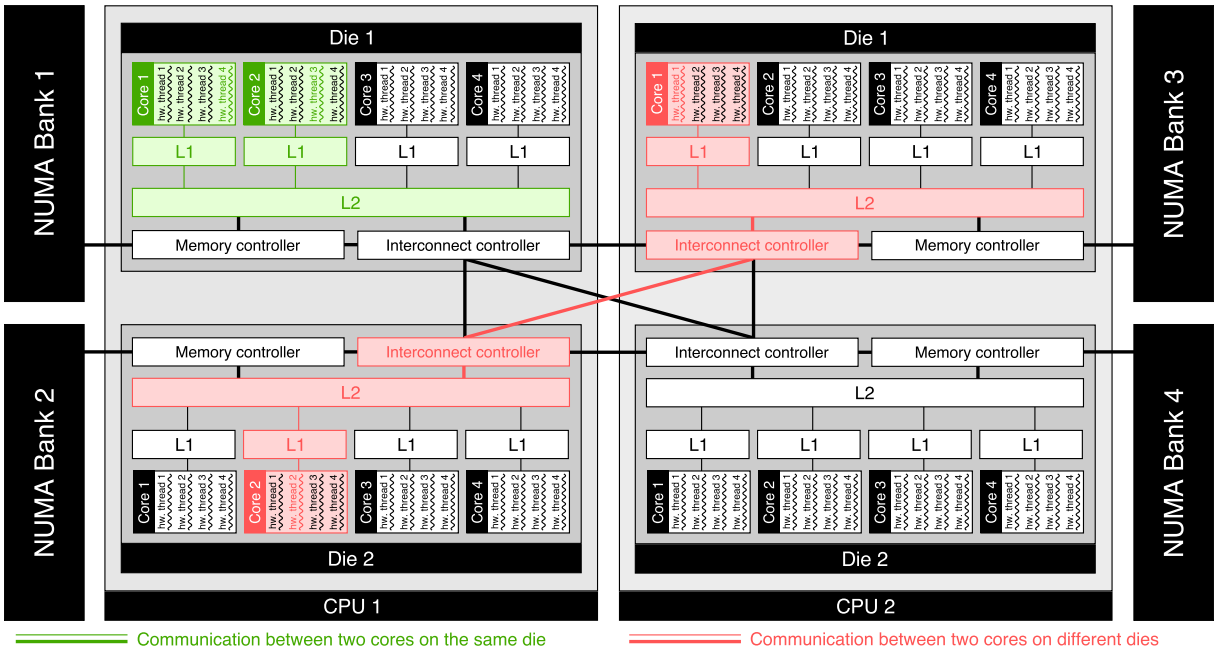


Figure 2.3: Communication between hardware threads with the MOESI protocol

in which they are executed. Memory barriers, also known as memory fences, make it possible to enforce an ordering constraint on memory operations issued before and after the barrier instruction. Most modern architectures (including x86) do not ensure that stores may not be reordered after loads to different addresses. To prevent this, a *store fence* can be issued between the store and load instructions. Atomic instructions also act as memory barriers: all pending load and store operations must be executed before the atomic instruction.[1]

Atomic instructions. The instruction set of current CPUs often includes a set of atomic instructions. These instructions combine several operations whose execution appears to be *atomic* to hardware threads, i.e., no other instruction from any hardware thread can modify the shared data they operate on during their execution. Atomic instructions make it possible for programs to modify shared data without having to acquire locks. Common atomic instructions include: (i) *test-and-set*, which reads (and returns) the value v_1 at a given address and replaces it with a new value v_2 if v_1 is non-zero[2], (ii) *fetch-and-store*, which reads (and returns) the value v_1 at a given memory address and replaces it with a new value v_2, (iii) *fetch-and-add*, which fetches a value v_1 at a given address, adds a value v_2 to v_1 and stores the result at v_1's address, (iv) *atomic swap*, or *atomic exchange*, which swaps the values at two given addresses in memory, and (v) *Compare-And-Swap (CAS)*, which compares the value v_1 at a memory address to a value v_2 and, in case of equality, writes a value v_3 at v_1's address. A whole class of algorithms, named *lock-free algorithms*, rely exclusively on atomic instructions instead of locks for synchronization [56, 77, 52, 40, 62, 63].

[1] On x86 architectures, atomic instructions only force the execution of pending memory instructions if the `LOCK` prefix is used. The `LOCK` prefix also ensures that during the execution of the atomic instruction, the hardware thread has exclusive ownership of the cache line on which a read-modify-write operation is performed.

[2] Several definitions of test-and-set exist. According to some authors, test-and-set behaves exactly like the fetch-and-store instruction described in the same paragraph.

2.3.1.2.c Bottlenecks

The cache-coherence protocol can often be a source of bottlenecks. It is important for developers to ensure good *cache locality* in multithreaded applications, i.e., to try to keep data in local caches as much as possible in order to avoid the overhead caused by cache misses. A common cache bottleneck is the presence of *false sharing*: two hardware threads may frequently access independent variables that are located in the same cache line, which results in the cache line needlessly "ping-ponging" between the caches, thereby causing unnecessary cache misses. Profilers such as DProf [87] are designed to locate cache locality bottlenecks. Sheriff [68] specifically detects false sharing and protects applications from it by adaptively isolating shared updates from different threads into separate physical addresses. Corey [13], an operating system for manycore architectures, proposes several techniques that aim to improve cache locality. In particular, it makes it possible for applications to dedicate cores for handling specific kernel functions or data. The research work presented in this thesis, RCL, tackles the specific issue of improving cache locality inside critical sections by executing them all on the same hardware thread.

2.3.1.3 Non-cache-coherent architectures

As seen in the previous section, the cache-coherence protocol incurs an overhead, and this overhead may get worse as the number of cores increases. To prevent this, some manufacturers have designed *non-cache-coherent architectures*, in which each core owns part of the global memory, and hardware threads must use *message-passing* in order to request data from other cores. Non-cache-coherent CPUs include Intel's Single Chip Cloud Computer (SCC) and to some extent, Tilera's TILE-Gx CPUs. In both the SCC and the TILE-Gx CPUs, cores are organized in a grid. Writing code for non-cache-coherent architectures can be very complex since cores cannot simply read and write from known memory addresses to communicate, and must rely on custom message-based protocols instead. Moreover, operating systems and applications have to be rewritten for these architectures. Fortunately, non-cache-coherent multicore architectures have many common points with distributed systems, and research on distributed operating systems has been ongoing for decades [66, 91, 104]. This led to the design of some research operating systems for non-cache-coherent architectures such as Corey [13] and Barrelfish [9] that solely rely on message-passing. Similarly, some software components such as garbage collectors [114] have been written for non-cache-coherent architectures. However, it will take a lot of time for operating systems and other software components to become as feature-rich as currently-used legacy software that has been developed for decades on cache-coherent architectures. Moreover, it has been shown [14] that legacy operating systems such as Linux can be made to scale on current cache-coherent multicore architectures with dozens of cores.

2.3.2 NUMA architectures

The main memory (RAM) can be organized in two ways on multicore architectures: UMA or NUMA. *Uniform Memory Access (UMA)* architectures use a very simple design: all CPUs address all of their memory requests to a single memory controller that is itself connected to the RAM. With this design, accessing any part of the RAM has the same cost (in latency and bandwidth) from any hardware thread. The main issue with UMA machines is that the unique memory controller can be a bottleneck on multicore architectures, especially as the number of hardware threads increases. *Non-Uniform Memory Access (NUMA)* architectures, on the other hand, use several memory controllers: cores are grouped into *NUMA nodes*, each of which is

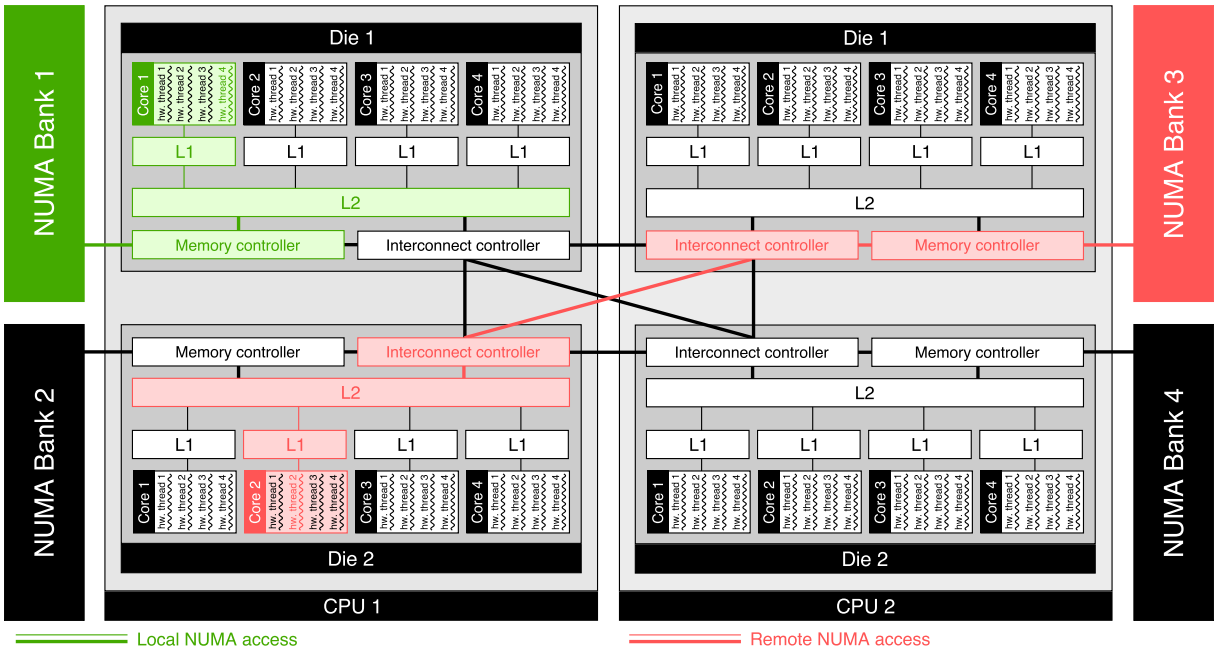


Figure 2.4: Local and remote NUMA accesses

connected to a *NUMA bank* that contains part of the main memory. The hardware handles mapping memory addresses to the right NUMA nodes: typically, the range of memory addresses is split into as many contiguous chunks as there are nodes, and addresses from chunk n are all mapped to the n^{th} NUMA bank. If a hardware thread needs to access a memory address that is located in a remote NUMA bank, then requests have to be sent on the interconnect to the node that owns that NUMA bank, possibly with several hops if there is no direct interconnect link between the two nodes. This indirection increases latency: local accesses are faster than remote accesses. Moreover, the interconnect links may decrease bandwidth if they are saturated, and if a lot of memory accesses from various nodes access the same NUMA bank, the memory controller of that NUMA bank can become a bottleneck, as is the case with UMA architectures.

In Figure 2.4, each die is a NUMA node that is connected to its local NUMA bank. An example of local NUMA access is shown in green: when a hardware thread reads data from memory, that data is copied into its L2 and L1 cache from which it can access it. When a hardware thread needs to access remote data, however, it must request it to the die whose NUMA bank holds the data. In the figure, any two dies have a direct interconnect link that connect them, therefore, one hop is sufficient. This is not always the case: requests sometimes have to be forwarded across several nodes in more complex multicore architectures, which comes at an increased latency cost.

Since remote accesses in NUMA architectures are costly, developers must be careful to write programs that avoid them: lack of NUMA locality can hinder the scalability of key software components such as garbage collectors [44]. Similarly to how developers can use profilers such as DProf [87] in order to detect remote cache or RAM accesses and improve cache locality, specific profilers such as MemProf [65] have been designed to help developers detect remote NUMA accesses and avoid them when possible. Some tools such as Carrefour [27] make it possible to automatically improve NUMA locality system-wide, by gathering statistics (such as those

provided by hardware performance counters [18]), and deciding when to migrate, interleave or replicate memory pages.

Hyper-Transport Assist. In some NUMA architectures, when a hardware thread t_1 in NUMA node n_1 needs to access data from a cache line that is stored in the remote caches of a hardware thread t_2 in NUMA node n_2, with the data being allocated in the RAM of node n_3, t_1 broadcasts a read or a write request to all caches because it does not know which cache owns the cache line. Since this scenario is fairly common, the resulting requests increase the load on caches and interconnect links, which may lead to non-negligible overhead. To prevent this, AMD Opteron CPUs use an optimization known as HyperTransport Assist [24] (a.k.a. HT Assist). With HT Assist, part of the highest level cache of die[3] d_3, whose memory controller handles node n_3, holds a *cache directory* (or *probe filter*) that contains information about the location of the cache lines that hold data from that node (n_3). Thanks to the cache directory, t_1 simply sends its request to d_3 instead of broadcasting it. The cache directory of d_3 indicates that the data is stored into t_2's caches, therefore, d_3 sends a message to t_2's die, which replies with the requested cache line to d_3, and d_3 forwards the cache line to t_1's die. Therefore, d_3 is used as a proxy and all communication is point-to-point, which is more efficient than broadcasting requests to all caches in the hope that one of them will respond with the cache line.

Interleaved memory. Current NUMA-capable systems usually make it possible to use *interleaved memory* instead of NUMA. With interleaved memory, memory addresses are allocated to each bank in turn. Consequently, contiguous reads and write access each bank in turn, which can improve memory throughput because less time is wasted waiting for memory banks to become ready for memory operations. Moreover, with interleaved memory, when hardware threads access a chunk of contiguous memory, the load is naturally balanced among memory controllers. NUMA usually offers better results than interleaved memory for applications that were designed with memory locality in mind, and should be more scalable in future architectures with many cores and memory banks.

2.4 Hetereogeneous architectures

While most current multicore architectures provide a set of identical cores, architectures that provide cores with various characteristics (different processing speeds, cores specialized for specific tasks) have been proposed: IBM's Cell processor, for instance, features one general-purpose CPU core and eight coprocessors organized in a ring. Even on common consumer architectures where all cores provided by CPUs tend to be identical, small, specialized cores provided by the Graphics Processing Unit (GPU) can be used for computations.

Exploiting the performance of heterogeneous multicore architectures is even more challenging than with homogeneous multicore architectures due to their increased complexity. Some schedulers that try to predict which threads could be executed more efficiently on faster cores have been proposed [64]. Other works propose to dedicate faster cores to specific tasks such as executing critical sections [102]. Finally, whole new operating systems such as Helios [78] have been proposed, with the claim that current operating systems are not able to scale on future heterogeneous architectures with a large number of cores.

[3]Or CPU, if the architecture uses one memory controller per CPU instead of per die.

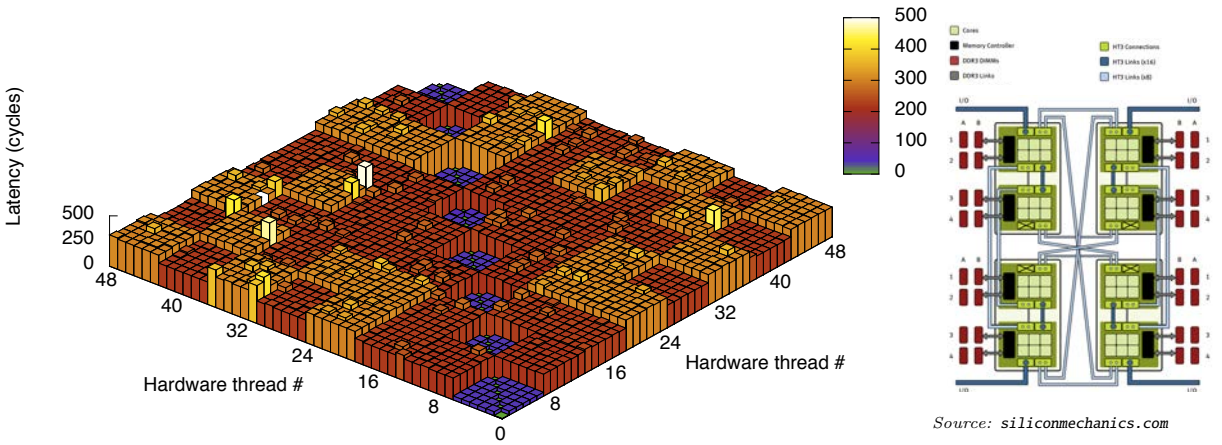
2.5 Machines used in the evaluation

This section describes the two machines used in the evaluation in Chapter 5. Section 2.5.1 describes Magnycours-48, a machine with 48 hardware threads that uses AMD Opteron processors. This machine is still available on the market, it is currently sold by Dell as a general-purpose server. Section 2.5.2 describes Niagara2-128, a machine that uses Sun UltraSPARC T2+ CPUs and offers more hardware threads (128) even though it is older: the UltraSPARC T2+, released in 2008, was replaced with the SPARC T3 in 2010. Finally, Section 2.5.3 discusses and compares the performance of the two machines using benchmarks.

2.5.1 Magnycours-48

Magnycours-48 is an x86 machine with four AMD Opteron 6172 CPUs (the Opteron 6100 series is codenamed "Magny-cours", hence the name of the machine). The CPUs' clock speed is 2.100GHz. Each of the CPUs has twelve cores split across two dies: Magnycours-48 features 48 cores in total. Since Opterons do not use hardware multithreading, Magnycours-48 also provides 48 hardware threads. Each core has a local L1 and L2 cache, while the L3 cache is shared among all six cores on the die. Each core has two dedicated 2-way set associative 128KB L1 caches, one for instructions and one for data, for a total of 256KB of L1 cache memory. Each core also has a 16-way set associative 512KB cache that contains both instructions and data. The six cores on each die share a 96-way set associative 6MB L3 cache. All caches have a 64KB cache line size. Each die is a NUMA node, therefore, Magnycours-48 has eight NUMA banks. Each bank handles 32GB of 1.333GHz U/RDDR3 memory, for a total of 256GB quad-channel main memory. The interconnect links between the six dies do not form a complete graph: each die is only connected to the other die on the same CPU and to three remote dies. Therefore, the diameter of the interconnect graph is two: inter-core communications (fetching cache lines from remote caches, or NUMA accesses, for instance) require at most two hops. The structure of the interconnect graph can be seen in Figure 2.5b, along with the rest the architecture of Magnycours-48. The interconnect uses HyperTransport 3.0 links with a theoretical peak bandwidth of 25.6GB/s at 6.4GT/s (GigaTransfers per second).

Boyd-Wickizer et al. use a set of tools in their paper about Corey [13] to measure various metrics about their hardware. In particular, they provide an application named Memal that loads cache lines in the L1, L2 or L3 cache of a specific core c_1, and accesses them remotely with a remote core c_2, in order to measure the cost of cache misses. Many cache lines are accessed and the benchmark returns the average access time. Running Memal for every pair of cores on Magnycours-48 gives the results shown in Figure 2.5a when cache lines are initially loaded in the L1 cache. Local L1 accesses cost around 3 cycles (in green), and accessing data from a core on the same die costs around 38 cycles (in purple). There are two distinct costs for remote cache accesses, which is due to the non-complete interconnect graph described in the previous paragraph: accessing to data on a remote die that is directly connected to the local die (i.e., it is one hop away) costs around 220 cycles (in red), while accessing data from a die that is two hops away costs around 300 cycles (in yellow). The spikes in the graph are caused by the fact that these results are not averaged, therefore, any random interference during a run (context switches, for instance) can lead to erroneous longer access times. The reason why the results were not averaged is that given the large number of data points, running the experiment once already takes a long time. Running Memal for L2 and L3 caches does not alter remote access times, which shows that fetching a cache line from a remote CPU cache or from the NUMA bank of



(a) Cost of local and remote accesses (b) Architecture

Figure 2.5: Cache latencies and architecture of Magnycours-48

that CPU incurs the same cost. The local access times for L2 and L3 cache misses are around 15 and 30 cycles, respectively.

Software environment. Magnycours-48 runs Ubuntu 11.10 (Oneiric Ocelot) with a 3.9.7 Linux kernel, glibc 2.13, libnuma 2.0.5, and gcc 4.6.1.

2.5.2 Niagara2-128

Niagara2-128 is a SPARC machine with two Sun UltraSPARC T2+ CPUs (codenamed Niagara 2). Each CPU has a clock frequency of 1.165GHz and comes with eight cores on a single die. Each core runs eight hardware threads thanks to simultaneous multithreading. Each core has a 16KB 8-way set associative instruction cache and a 8KB 4-way set associative data cache. L1 cache lines are 16 bytes wide. The L2 cache is shared among all cores in a CPU. It is a 4MB 16-way associative cache, with 64-byte cache lines. The 32GB dual-channel FB-DIMM main memory is interleaved (NUMA is disabled). The two CPUs are connected with an interconnect whose theoretical peak bandwidth is 63 GB/s (42 GB/s read and 21 GB/s write).

The results of running Memal on Niagara2-128[4] for all hardware thread pairs with data in the L1 cache are shown in Figure 2.6. Local L1 accesses cost around 42 cycles (in green), and local L2 accesses have similar latency (not shown in the figure). Interestingly, even though accessing data on a remote core on the same CPU through the L2 cache costs around 46 cycles (in purple) for the first CPU, it costs 60 cycles for the second CPU. We have not been able to find the source of this discrepancy. Accessing data that is located on a remote CPU costs about 90 cycles (in pink). Again, the yellow spikes on the graph are artifacts that would be removed by averaging the results over several runs.

Software environment. Niagara2-128 runs Solaris 10 (SunOS 5.10) with gcc 4.7.1.

2.5.3 Performance comparison

This section discusses and compares the performance of Magnycours-48 and Niagara2-128. Section 2.5.3.1 compares the cache access latencies of the two machines. Section 2.5.3.2 uses a

[4]Memal had to be ported to Solaris to run on Niagara2-128.

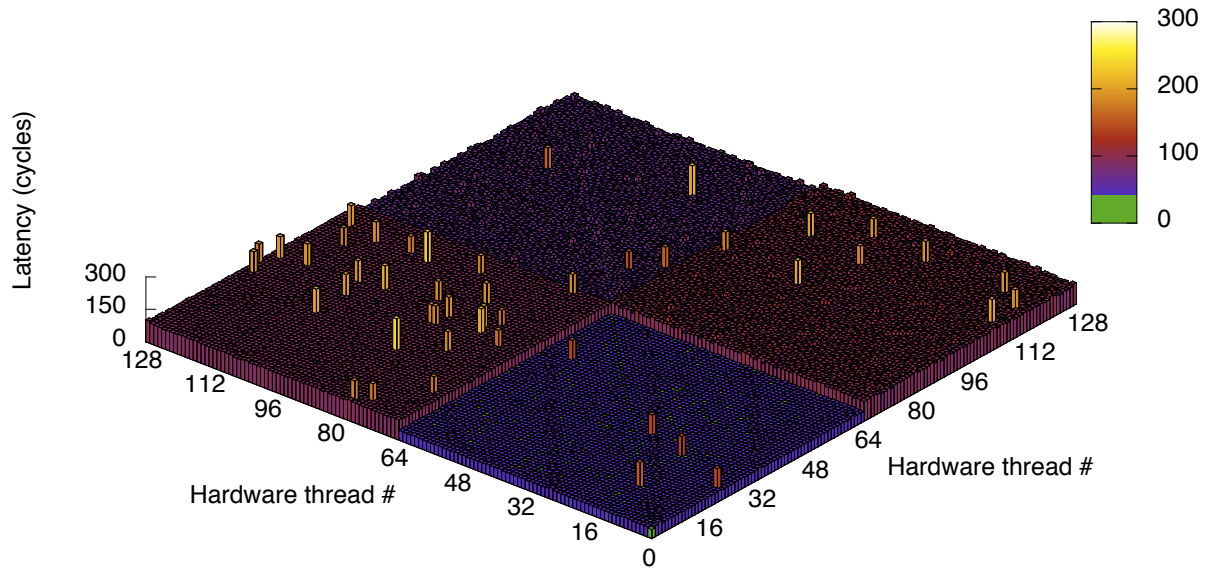


Figure 2.6: Cost of local and remote accesses on Niagara2-128

custom benchmark to measure the overhead of write-access contention on a shared cache line. Section 2.5.3.3 uses a benchmark suite to compare the sequential and the parallel performance of the two machines. Finally, Section 2.5.3.4 summarizes the results.

2.5.3.1 Cache access latencies

Memal was used in Sections 2.5.1 and 2.5.2 to measure the cache access latencies of Magnycours-48 and Niagara2-128, respectively. These results are summarized in Figure 2.7a, with latencies converted from cycles (c) to nanoseconds (ns) in order to allow for easier comparison between the two machines. On Niagara2-128, hardware threads access data from the local core up to 25.8 times slower than on Magnycours-48, and they access data from a different core on the local die up to 2.9 times slower than on Magnycours-48. However, Niagara2-128 can almost be twice as fast as Magnycours-48 when it comes to accessing data that is located on a remote die thanks to its faster interconnect and the fact that its two CPUs are directly connected (at most one hop is needed).

In summary, Magnycours-48 is slower when it comes to uncontended inter-die communication, but Niagara2 has slower uncontended communication inside its dies. Since Magnycours-48 has eight dies with six hardware threads on each instead of only two dies with sixty-four hardware threads on each for Niagara2-128, Magnycours-48 uses more inter-die communication, which is its weak point, and Niagara2-128 uses more communication that is local to its dies, which is also its weak point. To conclude, it is difficult to determine which of the two machines has the best performance when it comes to cache access latencies.

2.5.3.2 Contention overhead

A custom benchmark was written to assess the overhead of contention on regular and atomic instructions on Magnycours-48 and Niagara2-128. This benchmark runs a *monitored thread* that executes an instruction 1,000,000 times on a shared variable, and the execution time of every 1,000$^{\text{th}}$ instruction is measured: not all instructions are monitored in order to prevent the performance measurements from causing too much overhead. The measurements are then

	Local core access	Local die access	Remote die access
Magnycours-48	L1: 3c / 1.4ns L2: 15c / 7.1ns L3: 30c / 14.3ns	38c / 18.1ns	One hop: 220c / 104.1ns Two hops: 300c / 142.9ns
Niagara2-128	L1 / L2: 42c / 36.1ns	CPU 1: 46c / 39.5ns CPU 2: 60c / 52.5ns	90c / 77.3ns

(a) Cache access latencies

		1 thread	2 threads	24 threads	48 threads	64 threads	128 threads
Magnycours-48	**Store**	73c / 63.4ns	183c / 157.7ns	3,032c / 2,602.6ns	6,610c / 5,674.4ns		
Local NUMA bank	**CAS**	98c / 84.1ns	182c / 156.6ns	947c / 813.4ns	5,561c / 4,773.6ns		
Magnycours-48	**Store**	73c / 63.2ns	220c / 189.5ns	4,112c / 3,529.6ns	9,206c / 7,902.5ns		
Remote NUMA bank	**CAS**	98c / 84.1ns	341c / 293.3ns	5,656c / 4,855.6ns	11,749c / 10,084.9ns		
Niagara2-128	**Store**	56c / 48.0ns	56c / 48.0ns	1,623c / 1,393.2ns	4768c / 4,092.9ns	6,444c / 5,531.5ns	14,752c / 12,662.9ns
	CAS	75c / 64.3ns	75c / 64.3ns	1,633c / 1,401.8ns	4,749c / 4,076.3ns	6,353c / 5,454.0ns	14,860c / 12,755.8ns

(b) Overhead of contention on store and CAS instructions

Figure 2.7: Cost of cache accesses and instructions

averaged in order to produce an estimate of the execution time of the monitored instruction, which can either be an assignment (store) or an atomic Compare-And-Swap (CAS) instruction (see Section 2.3.1.2.b). Concurrently, the benchmark runs *non-monitored threads* that repeatedly write random values to the shared variable used by the monitored instruction, in order to simulate contention. The more hardware threads are added, the more contended the shared variable is. All threads (monitored or non-monitored) are bound to separate hardware threads. Threads are bound in such a way that they are first spread on the first hardware thread of each core of the first die, then CPU, and so on until they are bound to the first hardware thread of every core in the machine. After this, the same process continues with the second hardware thread of each core, until all hardware threads are used. On Magnycours-48, the shared variable can either be allocated on the NUMA bank of the monitored hardware thread, or on a remote NUMA bank. Figure 2.7b summarizes the results of the experiment.

Magnycours-48. When only one thread (the monitored thread) is used on Magnycours-48, the cost of a store (resp. CAS) instruction is 63.4 nanoseconds (resp. 84.1 nanoseconds). In this case, the shared variable is always stored in the L1 cache. When two threads are used, they are located on two cores of the same die, and the non-monitored thread often brings the shared variable to its local L1 cache, invalidating it from the monitored thread's L1 and L2 caches. However, even though the difference in latency between accessing a variable that resides in a local L1 cache and a remote L1 cache is only 16.7 nanoseconds (see Figure 2.7a), using two threads instead of one increases the costs of store and CAS instructions by at least 94.3 nanoseconds. Therefore, the overhead of adding another thread is not only caused by the additional cache misses: the synchronization mechanisms from the cache-coherence protocol induce an overhead when two threads try to write to the same cache line concurrently. Adding more threads keeps increasing the cost of store and CAS instructions to several thousand nanoseconds: using a store (resp. CAS) instruction under high contention is up to 124.6 (resp. 107.2) more costly than executing it locally under low contention.

On Magnycours-48, accessing a shared variable that belongs to a local NUMA bank has a lower latency than accessing data that belongs to a remote NUMA bank (-28.1% for store instructions, -52.7% for CAS instructions), even though the shared variable is usually directly transferred between caches during the benchmark and no RAM access is needed. This is due to the HT Assist technology described in Section 2.3.2: when the monitored thread repeatedly

accesses a variable that was allocated on a remote NUMA bank, all messages have to transit by the die whose memory controller is connected to that NUMA node, because the L3 cache of that die contains the NUMA node's cache directory. On the contrary, when the monitored thread accesses data that belongs to its own NUMA node, this indirection is not needed, because the monitored thread can directly access the corresponding cache directory on its local die.

Finally, store and CAS instructions have similar costs on Magnycours-48. CAS instructions are more expensive than store instructions under low contention (+55% with 24 threads) and under high contention when accessing data from a remote NUMA bank (+27.6% with 48 threads), however, they scale better than store instructions when they are performed on a local NUMA bank (−15.9% with 48 threads).

Niagara2-128. Executing store and CAS instructions on Niagara2-128 is faster than on Magnycours-48 under low contention (24.1% faster for store instructions and 23.5% faster for CAS instructions with one thread). Moreover, Niagara2-128 also scales better than Magnycours-48 on this benchmark: Niagara2-128 is up to 48.2% faster for store instructions and 59.6% faster for CAS instructions than Magnycours-48 when using 48 hardware threads. Increasing the number of threads beyond 48 threads keeps increasing the overhead of store and CAS instructions. In fact, the overhead increases linearly with the number of threads from 24 threads onwards, and with 128 threads, executing a single store (resp. CAS) instruction takes 263.8 (resp. 198.7) times longer than executing it locally under low contention. Finally, even though CAS instructions are 34.0% slower than store instructions under low contention, they perform similarly under moderate to high contention.

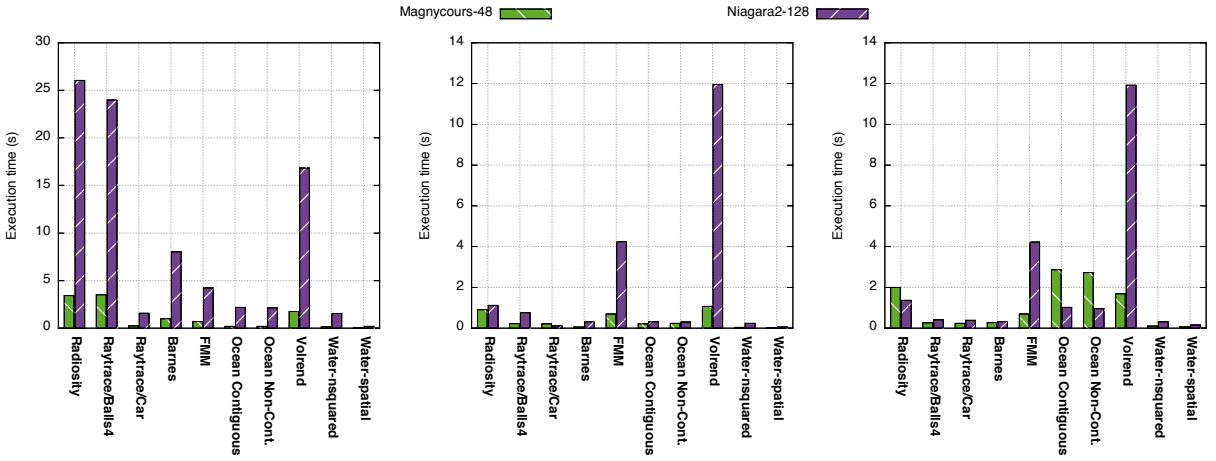
To conclude, Niagara2-128 is able to perform a more write accesses to a cache line than Magnycours-48: its architecture resists better when contention is high for concurrent accesses to a cache line.

2.5.3.3 Application performance

The second version of the Stanford ParalleL Applications for SHared memory (SPLASH-2) is a benchmark suite that consists of parallel scientific applications for cache-coherent architectures [107, 99, 110]. In this section, the applications from the SPLASH-2 suite are run on both Magnycours-48 and Niagara2-128, for one, 48, and 128 threads. The results are then analyzed: they provide some insight regarding the sequential and the parallel performance of both machines.

Sequential performance. Figure 2.8 shows the results of running the SPLASH-2 applications on Magnycours-48 and Niagara2-128 (each data point is averaged five times). Figure 2.8a shows the results of running the single-threaded version, which measures the sequential performance of a single hardware thread on both machine (no communication involved). Magnycours-48 clearly outperforms Niagara2-128, with performance improvements ranging between 6.1 times and 12.0 times. On average, Magnycours-48 is 8.9 times faster than Niagara2-128.

Parallel performance. Figure 2.8b shows the performance of the SPLASH-2 applications with 48 threads: Magnycours-48 still outperforms Niagara2-128 most of the time. However, the performance gap is reduced, and on one benchmark (Raytrace/Car), Niagara2-128 manages to outperform Magnycours-48. Niagara2-128 is able to run 64 threads on the same die and can therefore benefit from faster communications than Magnycours-48 (no need to go through the interconnect), which helps compensate its very low sequential performance. Niagara2-128 outperforms Magnycours-48 on Raytrace/Car because this benchmark spends most of its time



(a) Single-threaded (b) 48 threads (c) 128 threads

Figure 2.8: SPLASH-2 results

performing synchronization (lock acquisitions) instead of sequential computations, as will be seen in the evaluation (Chapter 4). Figure 2.8c shows that for 128 threads, Magnycours-48 is still much faster than Niagara2-128 for most benchmarks, even though it only has 48 hardware threads. Niagara2-128 manages to outperform Magnycours-48 on three benchmarks, however, (Radiosity, Ocean Contiguous and Ocean Non-Contiguous) thanks to its larger amount of hardware threads.

On a side note, it can be seen in Figure 2.8 that adding more threads when executing the SPLASH-2 applications often worsens performance instead of improving it: the SPLASH-2 suite was released in the 90's, when multi-CPU systems only featured a few CPUs with low sequential performance. Such legacy applications are often unable to scale on newer systems, and increasing the number of threads increases contention on shared resources such as locks and more generally, cache lines, which decreases performance.

To conclude, Niagara2-128 has much worse sequential performance than Magnycours-48. Its faster communication between hardware threads and larger number of hardware threads make it possible to reduce the performance gap with Magnycours-48 when a lot of threads are used, it is still clearly outperformed by Magnycours-48 on most applications of the SPLASH-2 suite.

2.5.3.4 Summary

Magnycours-48 is a machine that has much faster sequential performavnce and faster intra-die communication, while Niagara2-128 is a slower machine that sometimes exhibits better performance when it comes to communication between hardware threads, especially under high contention. However, the faster communication and larger amount of hardware threads of Niagara2-128 is not sufficient to make it perform better than Magnycours-48 on a parallel benchmark suite. Since Magnycours-48 has better sequential performance relative to its communication performance than Niagara2-128, it can be expected that synchronization will be more of a bottleneck on Magnycours-48 than on Niagara2-128. The evaluation in Chapter 5 will confirm this intuition.

2.6 Conclusion

Computer architectures are in a phase of mutation. After decades of CPU performance increases mainly coming from higher clock frequencies, manufacturers now tend to split components: the processing power of CPUs is split among cores and memory is split into NUMA banks. Communications between the cores, the caches, and the RAM become more and more complex, and the overhead of communication becomes more and more of a bottleneck: instructions can be several hundreds of time slower under high contention than they are under low contention. In this context, synchronization mechanisms have to evolve in order to perform well on newer machines: the next chapter presents the evolution of lock algorithms, i.e., synchronization mechanisms that make it possible to ensure that sections of code are executed in mutual exclusion.

Chapter 3

Lock algorithms

The lock is one of the oldest synchronization mechanisms [35, 57], and yet, it is still extensively used in modern applications. A lock makes it possible for multiple threads to execute sections of code in mutual exclusion. One of the main advantages of locks is their simplicity: in order to execute a section of code that needs to use a resource in mutual exclusion, a thread simply acquires a lock and is ensured that no other thread that uses the same resource will execute the critical code concurrently. Locks induce several overheads however. First, designing a lock algorithm that makes acquisition not costly when many threads try to acquire it concurrently is a challenge on multicore machines with a lot of hardware threads: naïvely designed locks require extensive communication between hardware threads to ensure mutual exclusion under high contention. Second, while using locks can be straightforward if developers do not try to minimize the size of critical sections, this approach can induce major overheads: long critical sections increase the duration of the critical path, and therefore reduce the potential performance improvement provided by parallelism in multicore machines, as stated by Amdahl's law [4]. To improve performance, developers can try to split critical sections into smaller ones, i.e., use *finer-grained locking*. Doing so can be very difficult however, because finding ways of reducing the size of critical sections is often not trivial, and leads to complex locking schemes that increase the probability of introducing concurrency bugs. Third, each time a thread executes a critical section, the shared data protected by the lock has to be brought to that thread, which causes costly cache misses on the critical path. While the second issue has to be solved by developers because it is application-dependent, lock algorithms focus on solving the first and third issues.

This section presents a selection of lock algorithms that includes the most common types of locks, including state-of-the-art combining locks. The lock algorithms are compared in the light of the following criteria:

- **Reactivity.** Locks that use busy-waiting react faster than locks that put threads to sleep when the lock is taken, because they do not require context switches between the execution of critical sections. The downside of this approach is that busy-waiting consumes a lot of CPU time and energy.
- **Performance under high contention.** An efficient lock algorithm should allow a thread to enter a critical section quickly when many threads on many hardware threads try to acquire the lock concurrently.
- **Number of atomic instructions.** Some lock algorithms use a lot of contended atomic instructions on shared synchronization variables, which can induce bottlenecks.

- **Ordering of critical sections.** Execution of critical sections in the First-In-First-Out (FIFO) order is preferred over other orderings, because it improves fairness.[1]
- **Potential starvation.** Well-designed lock algorithms prevent starvation, i.e., the lack of progress of one or several threads.
- **Resistance to preemption.** The performance of some lock algorithms drops when the lock holder or threads that wait for the lock get preempted by the operating system's scheduler. This is not a desirable property because even though current architectures offer dozens of hardware threads, some applications still spawn a larger amount of software threads than there are hardware threads. This is the case for Berkeley DB with the TPC-C benchmark presented in the evaluation in Chapter 5.
- **Parameters.** Some algorithms use one or several parameters that need to be fine-tuned for optimal performance. Finding satisfactory values for these parameters can sometimes be complex and time-consuming.
- **Data locality of internal structures.** Some lock algorithms try to reduce the migration of the shared variables and structures they use internally during the execution.
- **Data locality of critical sections.** Some lock algorithms consecutively execute a large number of critical sections on the same hardware thread in order to make sure that shared variables and structures used by critical sections remain in that hardware thread's caches.
- **Usability in legacy applications.** The most commonly used lock in legacy applications is the POSIX lock, which is a blocking lock that comes with additional synchronization mechanisms such as condition variables. If a lock algorithm implements the same API as the blocking lock and makes it possible to easily implement condition variables, it can be easily used in legacy applications.

Sections 3.2 to 3.8 presents a selection of lock algorithms: blocking locks, a basic spinlock, the CLH lock, the MCS lock, time-published locks, the Oyama lock, Flat Combining, CC-Synch, and DSM-Synch. Section 3.9 compares these lock algorithms. Section 3.10 briefly presents other locks that were not considered in the study. Finally, Section 3.11 concludes.

3.1 Blocking locks

The main idea of blocking locks is that a thread that is not able to acquire the lock instantly is put to sleep. An example implementation of a blocking lock can be the following. In order to acquire the lock, a thread (1) tries to set a global lock variable to `true`; in case of failure, the lock is already taken, therefore, (2) it is enqueued in a global queue and (3) it is put to sleep. The operating system makes sure that the thread cannot be preempted during operations (1) and (2) in order to avoid a lost wakeup (this can be achieved with the `futex()` system call in Linux, for instance). To release the lock, a thread either resets the lock variable if the list is empty, or wakes up the next thread in the list. Again, the unlock function must be carefully

[1] FIFO ordering may not be preferred in real-time systems, where the scheduler must not be fair to processes. However, none of the locks considered in this chapter are designed with real-time systems in mind, and real-time systems are beyond the scope of this thesis.

designed to avoid any problematic concurrent interactions between threads that try to acquire the lock and the thread that releases it.

Blocking locks use less power and CPU time than lock algorithms that busy-wait for locks. However a context switch is required between each lock acquisition, which makes blocking locks lack reactivity. Blocking locks are used in most legacy applications, since they are the only type of lock that works properly on machines that provide only one hardware thread: on such a machine, busy-waiting consumes CPU time without leaving a chance for the lock holder to release the lock. This is why the most common locks on *NIX systems, the *POSIX locks*, are blocking locks.

POSIX locks are acquired and released using the `pthread_mutex_lock/unlock()` functions. They also make it possible to wait on *condition variables*: the `pthread_cond_wait()` function atomically releases a POSIX lock while causing the calling thread to sleep on a condition variable. A subsequent call by another thread to `pthread_cond_signal()` can be used to wake up one of the threads that waits on that condition variable, upon which the lock will be reacquired. A call to `pthread_cond_broadcast()` can be used to wake up all threads that wait on a given condition variable. Condition variables are frequently used in legacy applications: roughly half of the multithreaded applications that use POSIX locks in Debian 6.0.3 (October 2011) also use condition variables.

3.2 Basic spinlock

Spinlocks are lock algorithms that use busy-waiting (a.k.a. *spinning*) on a global condition variable that represents the set of the lock. This section presents a basic spinlock that uses the atomic compare-and-swap (`CAS()`) instruction presented in Section 2.3.1.2.b in a busy-wait loop to try to set the value of a global lock variable. The thread that manages to set the lock variable owns the lock and is able to execute the critical section. The lock is simply released by resetting the lock variable through a standard assignment. This basic algorithm is shown in Algorithm 1.

Algorithm 1: Basic spinlock

```
1 types:
2   lock_t              int *;

3 function lock(lock_t lock)
4     while ¬CAS(lock, 0, 1) do          // Try to atomically set the lock variable
5         pause();                       // Busy-wait loop hint

6 function unlock(lock_t lock)
7     *lock := 0;                        // Reset the lock variable
```

The basic spinlock is very efficient under low contention on multicore architectures thanks to the very low number of instructions needed to implement it. This lock algorithm makes it possible to hand the lock over very quickly between threads. However, when contention is high, the performance of the basic spinlock is extremely poor, because of the write-access contention on the cache line that contains the `lock` variable: when many threads try to execute the atomic compare-and-swap instruction concurrently on the same cache line, the memory bus gets saturated by the messages of the cache-coherence protocol, and the cost of a compare-and-swap instruction becomes very high, as shown in Section 2.5.3.2. More evolved spinlocks algorithms exist. In some variants, at each iteration of the busy-wait loop, the lock variable is read, and a compare-and-swap instruction is only executed if the lock variable indicates that the lock is not taken: this optimization can decrease write-access contention on the lock variable. As will

be described in Section 3.10, some lock algorithms are similar to spinlocks except they make threads yield the processor for some time at each iteration of the busy-wait loop, which makes their behavior closer to that of blocking locks.

Legacy applications that use POSIX locks can easily switch to spinlocks, because spinlocks provide the same interface as POSIX locks: their `lock()` and `unlock()` functions only use one argument that represent the lock. Moreover, a thread can easily wait on a condition variable with spinlocks using POSIX primitives with the following steps: the thread (i) acquires a POSIX lock, (ii) releases the spinlock, (iii) calls the POSIX function for acquiring a condition variable (`pthread_cond_wait()`), (iv) releases the POSIX lock and (v) reacquires the spinlock. The `pthread_cond_signal()` and `pthread_cond_broadcast()` can then be used directly to wake up threads that are waiting on condition variables.

`PAUSE` instruction. In Algorithm 1, the `pause()` function call implements a busy-wait loop hint, which corresponds to the `PAUSE` instruction on an x86 machine. If a `PAUSE` instruction is not used, the branch predictor predicts that the CAS operation will always be unsuccessful, and the pipeline gets filled with speculative CAS instructions that are expected to be unsuccessful. When the lock is released, a memory order violation occurs: the processor sees that the variable has been modified, and all invalid speculative CAS instructions must be flushed from the pipeline, which induces a significant overhead. The `PAUSE` instruction waits long enough for speculative comparison instructions to not get enqueued in the pipeline, but not long enough to induce significant overhead on exit of the busy-wait loop. Moreover, preventing the pipeline from filling up with speculative CAS operations saves energy as well as CPU time for other hardware threads on the same core. Some CPUs may not provide a busy-wait loop hint, in which case high-latency instructions can be used to simulate a `PAUSE` instruction. Since UltraSPARC T2+ CPUs do not provide a `PAUSE` instruction, the performance of several high-latency instructions were evaluated on Niagara2-128 (one of the machines presented in Section 2.5.2), and the "`RD %CCR,%G0`", instruction, recommended by Dave Dice [32], was found to be the most efficient: this instruction is therefore used in all busy-wait loops in the evaluation (Chapter 5).

3.3 CLH

The CLH lock [25, 74], invented independently at the University of Washington by Travis Craig, and at the Swedish Institute of Computer Science by Anders Landin and Eric Hagersten, aims to improve on spinlocks. As was explained in Section 3.2, the main issue with spinlocks is that all threads busy-wait on the same synchronization variable, which causes high contention on the cache line that contains it. CLH removes the need for busy-waiting on a single global variable, and therefore performs better under high contention. The CLH algorithm is shown in Algorithm 2.

CLH uses a global queue for each lock. This global queue initially contains a dummy node whose `succ_must_wait` variable is set to **`false`**. Each thread owns a node. In order to acquire the lock, a thread sets its node's `succ_must_wait` variable to **`true`**, atomically inserts itself into the queue using the fetch-and-store instruction presented in Section 2.3.1.2.b, and waits for the `succ_must_wait` variable of its predecessor thread's node to be set to **`false`**. When this happens, the thread owns the lock and can execute its critical section. To release the lock, a thread simply sets its node's `succ_must_wait` variable to **`false`**, which unlocks the next thread in the queue.

Since each thread busy-waits on a different `succ_must_wait` variable, instead of all threads busy-waiting on the same variable, CLH is more efficient than the basic spinlock under high

Algorithm 2: CLH

```
1  structures:
2    node_t          { node_t *prev,                    // Pointer to successor in the queue
                       boolean succ_must_wait };        // Used for busy-waiting
3  types:
4    lock_t          node_t *;

   // The 'lock' parameter points to the tail of a list of nodes, each node representing a thread that waits for the lock.
      Initially, the list contains a dummy node with the values (nil, false). The 'node' parameter points to a thread-
      local node.
5  function lock(lock_t *lock, node_t *node)
6      var node_t *pred;
7      node → succ_must_wait := true;                   // Successor will have to wait
8      node → prev := fetch_and_store(lock, node);      // Queue for lock
9      pred := node → prev;                             // Get predecessor
10     while pred → succ_must_wait do
11         pause();                                     // Busy-wait for lock
12 function unlock(node_t **node)
13     var node_t *pred := *node → prev;                // Get predecessor
14     *node → succ_must_wait := false;
15     *node := pred;
```

contention. CLH provides another advantage over the basic spinlock: since threads enqueue their nodes in a FIFO queue, FIFO ordering of lock acquisitions is ensured. This property eliminates the risk of starvation caused by lock acquisitions.

CLH may be slightly more difficult to use in legacy applications than the basic spinlock, because its `lock()` and `unlock()` function use two arguments instead of one in POSIX locks: (i) a pointer to the tail of the list of nodes, and (ii) an pointer to the local thread's node, which can be modified. This can be solved by using a thread-local variable to store the pointer to the local thread's node. If thread-local variables are not provided by the programming language, compiler, and/or runtime that is used by the legacy application, some light code refactoring might be needed to make it use CLH. Condition variables can be implemented for CLH using the same technique as the one proposed for spinlocks (see Section 3.2).

3.4 MCS

The MCS lock [76] is similar to the CLH lock since it also uses a global queue with one node per thread, and threads busy-wait on one synchronization variable per thread instead of a global one. In MCS however, a thread always use the same node during the whole execution. This node can be allocated locally, which can improve performance on non-cache-coherent architectures. The MCS algorithm is shown in Algorithm 3.

In MCS, each thread owns a node for each lock. When a thread t needs to acquire the lock, it enqueues itself atomically into the global queue. If the node is the only element in the queue, the lock was free and t can go on with the execution of the critical section. Otherwise, the queue contains the nodes of all the other threads that are waiting to acquire the lock. In that case, t busy-waits on its node's `locked` variable until it is set to **`false`**, which means that its predecessor is done executing its critical section and t now owns the lock. In order to release the lock, thread t first checks whether its node is the last element in the queue. If this is the case, t atomically removes itself from the queue: the lock is not held by any thread anymore. If the node was not the last one in the queue or if removing the node atomically from the queue failed, at least one

Algorithm 3: MCS

```
1  structures:
2    node_t          { node_t *next,                    // Pointer to successor in the queue
                       boolean locked };                // Used for busy-waiting
3  types:
4    lock_t          node_t *;

   // The 'lock' parameter points to the tail of a list of nodes, each node representing a thread that waits for the lock
      (initially, *lock := nil). The 'node' parameter points to a thread-local node.
5  function lock(lock_t *lock, node_t node)
6      var node_t *pred;
7      node → next := nil;                              // Initially, no successor
8      pred := fetch_and_store(lock, node);             // Queue for lock
9      if pred ≠ nil then                               // If lock was not free
10         node → locked := true;                       // Prepare to busy-wait
11         pred → next := node;                         // Link behind predecessor
12         while node → locked do
13             pause();                                 // Busy-wait for lock
14 function unlock(lock_t *lock, node_t *node)
15     if node → next = nil then                        // If no known successor
16         if CAS(lock, node, nil) then
17             return;                                  // No successor, lock free
18     while node → next := nil do
19         pause();                                     // Wait for successor
20     node → next → locked := false;                   // Pass lock
```

other thread is waiting for the lock. After making sure that the next thread's node is properly enqueued, t hands over the lock to the next thread in the queue by setting that thread's `locked` variable to **`false`**.

CLH and MCS are both *queue locks*: they use a global queue of waiting threads, which ensures FIFO ordering for lock acquisitions. The global queue makes it possible to use as many synchronization variables as there are threads, thereby removing the main overhead of the basic spinlock: threads do not busy-wait on a single global synchronization variable. MCS has an advantage over CLH: threads always reuse the same node as their local node, and they always busy-wait on a variable that belongs to their local node. This can improve performance on non-cache-coherent architectures if each thread's node is allocated locally: no busy-waiting on a remotely allocated variable is needed, and busy-waiting on a remotely-allocated variable in non-cache-coherent architectures can use a lot of bandwidth and may lack reactivity. On NUMA architectures, MCS makes it possible for a client to spin on a locally-allocated synchronization variable, but the synchronization variable is not contended (one per thread) and only accessed in read mode in the busy-wait loop, therefore, it should remain cached until it is invalidated just before the client exits the busy-wait loop. Consequently, performance gains of MCS over CLH on NUMA architectures should be negligible.

Using MCS in legacy applications leads to the same issues as using CLH. In particular, the `lock()` and `unlock()` functions of MCS use two arguments instead of one for POSIX locks. A variant of MCS, named K42 [7], solves this issue: its `lock()` and `unlock()` only take the tail of the queue as their argument, without the need of thread-local variables. While using K42 can be more practical than using MCS in some legacy applications, its performance has been shown to be worse [15], and it is patented by IBM, which limits its use.

3.5 Time-published locks

While queue locks such as CLH and MCS perform better than basic spinlocks due to the fact that they use one synchronization variable per thread for busy-waiting, their performance can drop drastically in an environment where threads can get preempted. Preemption can be problematic in two cases [81]: (i) the thread that owns the lock gets preempted, in which case all the threads that are waiting for the lock waste CPU time busy-waiting needlessly, and (ii) when the lock is released, the next thread in the queue has been preempted, therefore, the following threads in the queue also waste CPU time busy-waiting even though the lock is free and they should be allowed to acquire it.

Another issue with queue locks and preemption is the *convoy effect* [64], in which the FIFO ordering of lock acquisitions and the FIFO scheduling policy of the operating system interact in such a way that critical sections take one or several of the scheduler's time quanta to be executed. As an example, let us consider a very basic scenario where a convoy occurs with a queue lock. Suppose four threads t_1, t_2, t_3 and t_4 run on a machine with three hardware threads, and all of them execute critical sections whose execution time is significantly shorter than a time quantum. The scheduler uses a FIFO policy with this order for thread scheduling:

$$\text{[Q1] Scheduler queue:} \qquad t_1 \leftarrow t_2 \leftarrow t_3 \leftarrow t_4$$

In this case, threads t_1, t_2 and t_3 are each running on one of the three hardware threads, and t_4 sleeps. The queue lock's queue contains all threads, in that order:

$$\text{[Q1] Lock queue:} \qquad t_4 \leftarrow t_3 \leftarrow t_2 \leftarrow t_1$$

All threads wait for t_4 to execute its critical section. Since thread t_4 sleeps, it cannot execute its critical section until the next time quantum. At the next quantum, the situation is the following:

$$\text{[Q2] Scheduler queue:} \qquad t_4 \leftarrow t_1 \leftarrow t_2 \leftarrow t_3$$

I.e., t_4, t_2 and t_1 are running, and t_3 is sleeping. Thread t_4 can now execute its critical section, after which the situation is the following (supposing t_4 enqueues itself again to execute a new critical section):

$$\text{[Q2] Lock queue:} \qquad t_3 \leftarrow t_2 \leftarrow t_1 \leftarrow t_4$$

Since t_3 is sleeping, it cannot execute its critical section during this time quantum. At the next time quantum, the scheduler's queue becomes:

$$\text{[Q3] Scheduler queue:} \qquad t_3 \leftarrow t_4 \leftarrow t_1 \leftarrow t_2$$

At which point t_3 can execute its critical section, and the lock queue becomes:

$$\text{[Q3] Lock queue:} \qquad t_2 \leftarrow t_1 \leftarrow t_4 \leftarrow t_3$$

This is again in a situation in which the lock holder t_2 cannot execute its critical section until the next time quantum because it is sleeping. In this scenario, it always takes at least one time quantum to execute a critical section. Since time quanta are usually orders of magnitude slower than critical sections, this phenomenon can drastically slow down the execution of the application. More complex convoys can occur, in which executing a critical section takes more than one time quantum.

Time-published locks [49] are modified versions of the MCS and CLH locks that help prevent the issues caused by preemption in MCS and CLH locks. They use a timestamp-based heuristic to solve the two problems stated at the beginning of this section (which occur when the lock holder or waiting threads in the list get preempted), as well as convoys. Each thread periodically writes the current system time to a shared location. If a thread t fails to acquire the lock for a long amount of time, it checks the timestamp of the lock holder, and if that timestamp is stale, it assumes that the lock holder has been preempted. Therefore, it yields the processor in order to increase the probability for the lock holder to be scheduled again: this technique helps solve problem (i). Moreover, if a thread t_1 reads a stale timestamp for a thread t_2 that is waiting in the queue, it assumes that t_2 has been preempted, and therefore remove t_2 from the queue: this helps solve problem (ii). Finally, the timestamp-based heuristic quickly removes convoys : since a convoy results in stale timestamps, the preempted threads get removed from the queue, and the waiting threads yield, forcing a reordering of the scheduling queue.

The algorithm of the modified MCS lock, known as MCS-TP, is shown in Algorithms 4[2] and 5. While synchronization variable of MCS can only hold two values that indicate whether the lock is held or not, the modified synchronization variable of MCS-TP can take five values: (i) `INIT`, which is used before any lock acquisition, (ii) `AVAILABLE`, which means the thread holds the lock, (iii) `WAITING`, which means the threads is busy-waiting in the queue, (iv) `TIMED_OUT`, which means the thread has been removed from the queue because another thread assumed it had been preempted, and (v) `FAILED` which is used when a lock acquisition failed because it was too long (in which case the thread tries to acquire the lock again). MCS-TP is a complex algorithm whose main drawback is that it requires several constants for which it may be difficult to find satisfactory values: (i) `MAX_CS_TIME` is an approximate upper bound on the length of critical sections, which is application-dependent and cannot be easily evaluated without profiling, (ii) `PATIENCE` is the amount of time a thread should wait in the queue, for which a satisfactory value can only be determined empirically, (iii) `UPDATE_DELAY`, which is the amount of time it takes a thread to see a timestamp published on another thread, including any potential clock skew. Again, this value is hard to evaluate and is hardware-dependent.

3.6 Oyama

The Oyama lock [82] is a type of lock that aims to improve data locality by making threads execute several critical sections consecutively by a *server* thread: since critical sections that are protected by a given lock often perform operations on the same set of shared variables, executing several of them consecutively on the same hardware thread makes it possible for these variables to remain in that hardware thread's caches, thus reducing the number of cache misses on the critical path and therefore improving performance. Similarly to MCS and CLH, Oyama uses a global queue in which threads enqueue themselves when they need to execute a critical section. With Oyama, however, each node contains a pointer to a function that encapsulates the critical section so that it can be executed remotely. Oyama also uses a global synchronization variable that can take three values: (i) `FREE`, which means that the lock is free, (ii) `LOCKED`, which means

[2]Line 38 is not present in the MCS-TP algorithm presented in the technical report that describes the algorithm [25]. However, omitting it causes the current lock to sometimes not register itself as being the last acquired lock by the current thread, even though that thread's node is left in the global queue. Therefore, on the next time the `trylock()` function is called, the algorithm may consider that the last considered lock was a different one while this is not the case, which can cause threads to incorrectly insert their nodes several times in the global queue and lead to deadlocks.

Algorithm 4: MCS-TP, `lock()` function

```
1  types:
2    enum_status_t    enum { INIT, AVAILABLE, WAITING, TIMED_OUT, FAILED };
3  structures:
4    node_t           { lock_t *last_lock, long_long time, node_t *next, boolean locked, enum_status_t status };
     lock_t           { node_t *tail, long_long cs_start_time };
   // The 'lock' parameter points to the tail of a list of nodes, each node representing a thread that waits for the lock
      (initially, *lock := nil). The 'node' parameter points to a thread-local node.
5  function lock(lock_t *lock)
6      while ¬trylock(lock) do
7          ;
8  function trylock(lock_t *lock, node_t *node) : int
9      var node_t *pred;
10     var long_long start_time := get_timestamp();
11     if node → status ≠ TIMED_OUT or node → last_lock ≠ lock or
12       ¬CAS(&node → status, TIMED_OUT, WAITING) then          // Try to reclaim previous position in queue.
13         node → status := WAITING;
14         node → next := 0;
15         pred := test_and_set(&lock → tail, node);
16         if ¬pred then                                        // The lock was free.
17             lock → cs_start_time := get_timestamp();
18             return 1;
19         else
20             pred → next := node;
21     while true do
22         if node → status = AVAILABLE then
23             lock → cs_start_time := get_timestamp();         // Lock free, update timestamp and acquire it.
24             return 1;
25         else if node → status = FAILED then
               // The lock acquisition failed. If the lock holder appears to have been preempted, yield to leave it a
                  chance to release the lock.
26             if get_timestamp() − lock → cs_start_time > MAX_CS_TIME then
27                 yield();
28             node → last_lock := lock;
29             return 0;
           // Busy-wait loop: busy-wait for a limited amount of time (PATIENCE) then try to time out. If the lock
              holder appears to have been preempted, yield to leave it a chance to release the lock.
30         while node → status = WAITING do
31             node → time := get_timestamp();
32             if get_timestamp() − start_time ≤ PATIENCE then
33                 continue;
34             if ¬CAS(&node → status, WAITING, TIMED_OUT) then
35                 break;
36             if get_timestamp() − lock → cs_start_time > MAX_CS_TIME then
37                 yield();
38             node → last_lock := lock;
39             return 0;
```

that a critical section is being executed, but no other critical section needs to be executed after that, and (iii) a pointer to the global queue. When a thread t needs to execute a lock, it tries to atomically switch the value of the global variable from `FREE` to `LOCKED`. In case of failure, another thread is executing a critical section, therefore, t enqueues itself atomically into the global queue. In case of success, t owns the lock: it executes its critical section, then tries to release the lock using a compare-and-swap operation. This operation can fail if other threads

Algorithm 5: MCS-TP, `unlock()` function

```
1  function unlock(lock_t *lock, node_t *node)
2      var int scanned_nodes = 0;
3      var node_t *succ;
4      var node_t *curr := node;
5      var node_t *last := nil;
6      while true do
7          succ := curr → next;
8          if ¬succ then
9              if CAS(&lock → tail, curr, nil) then          // Leave the queue if last in line
10                 curr → status := FAILED;
11                 return;                                   // We were last in line
12             while ¬succ do                                // Find last element
13                 succ := curr → next;
14         scanned_nodes++;
15         if scanned_nodes < MAX_THREADS then
16             curr → status := FAILED;
17         else if last = nil then
18             last := curr;                                 // Handle treadmill case
19         if succ → status = WAITING then
20             long_long succ_time = succ → time;
21             if get_timestamp() − succ_time ≤ UPDATE_DELAY
22                 and CAS(&succ → status, WAITING, AVAILABLE) then
23                 while last and last ≠ curr do
24                     last → status := FAILED;
25                     last := last → next;
26                 return;
27         curr := succ;
```

enqueued themselves during the execution of the critical section. In this case, t becomes a server: it detaches the queue, and executes all of the critical sections it contains. Thread t then tries to release the lock atomically again, and in case of failure, it may have to detach the queue again and to execute the newly-enqueued critical sections. The scheme goes on until t manages to release the lock.

The basic idea of Oyama, namely, making a thread execute a streak of critical sections in order to improve data locality, is also used by the more recent lock algorithms presented in the to next sections (Flat Combining, CC-Synch and DSM-Synch). This approach does not only improve data locality, it also accelerates the critical path because when a thread executes a streak of critical sections, no synchronization is needed between the execution of each critical section. The only communication overhead is caused by the cache misses needed for fetching the code and parameters of the next critical sections to execute (which can usually be avoided with prefetching), and signaling threads that their critical sections have been executed. In comparison, with the basic spinlock, blocking locks, or queue locks, before starting to execute a critical section, the thread that releases the lock must write to a contended variable, and the new thread that acquires the lock must see that change and get out of its busy-wait loop. This overhead can be costly, because it is located on the critical path: if n threads execute critical sections concurrently at a very high rate, on average, each thread has to wait for $n-1$ other critical sections to be executed before its critical section is executed, which means that the overhead of synchronization has to be paid $n-1$ times.

While Oyama has several advantages over the locks presented in the previous sections, it also has drawbacks: first, it uses a global lock variable which could get highly contended. Second, the implementation uses a large amount of atomic instructions. Third, critical sections are executed

in LIFO order which is bad for fairness. Fourth, the thread that executes critical sections may starve in a scenario of high contention where threads enqueue themselves in the queue faster than one thread can execute them. In that case, the thread executing the critical sections will never be able to resume its execution.

Finally, Oyama cannot be used directly in legacy applications, for two reasons. First, Oyama requires critical sections to be encapsulated into functions. This requires extensive code refactoring, and no solution to this problem is given in the Oyama paper [82]. Second, condition variables cannot easily be implemented for Oyama because by making the server thread sleep, they would prevent it from executing remaining critical sections in the queue. Moreover, since server threads are normal application threads, waiting on condition variables could randomly prevent an application thread from making progress, which could cause undesirable unexpected effects such as deadlocks. These two issues with Oyama also affect the three lock algorithms (Flat Combining, CC-Synch and DSM-Synch) that will be presented in the next two sections.

3.7 Flat Combining

Flat Combining [51] is not a lock algorithm per se, but rather a synchronization paradigm that aims to combine operations on a shared object in a way that decreases algorithmic complexity and improves data locality. It is sometimes possible for a thread to execute n operations on a shared object with a more efficient sequential algorithm than for n threads to execute one operation each. With Flat Combining, when a thread needs to execute such an operation on an object, it enqueues itself in a global queue, then waits for a *combiner thread* to execute its operation. If there is no combiner thread, it becomes a combiner thread itself and executes all operations from the queue, hopefully using an efficient sequential algorithm. When the thread is done executing all pending operations, it resumes its normal execution.

Flat Combining has two main benefits. First, it makes it possible to use a fast sequential combining algorithm for a queue of operations. Second, as is the case with Oyama, since many operations on a shared object are executed consecutively by the same thread, (i) no synchronization is needed during the execution of the critical sections that are executed by the combiner thread other than signaling threads that their critical section has been executed, and (ii) data locality is improved because the shared data used by critical section remains on the same hardware thread. It is interesting to note that if the combiner thread does not use a special combining algorithm (since it is not always possible to find such an algorithm) but simply executes requests one after the other, Flat Combining becomes an efficient lock mechanism that executes the operations in mutual exclusion while improving data locality and reducing the overhead of synchronization. Using Flat Combining as a lock is similar to using Oyama, but it is more efficient, since Flat Combining eliminates three of its drawbacks. First, Flat Combining never leads to starvation. Second, it executes its request in the FIFO order instead of LIFO. And third, it uses less atomic instructions than Oyama.

The algorithm of Flat Combining is shown in Algorithm 6. Like Oyama, Flat Combining uses a global lock and a global queue whose nodes contain pointers to functions that encapsulate critical sections. Each thread owns one node per lock, which can either be active or inactive. Flat Combining also uses a global counter for tracking old, inactive nodes and removing them from the queue. When a thread t needs to execute a critical section, it first checks if its node is active. If its node is inactive, it activates it and inserts it into the queue. Then, t either waits for a combiner to execute its request, or for the global lock to be free, in which case it can acquire

Algorithm 6: Flat Combining

```
1  structures:
2    node_t          { request_t req,  ret_val_t ret, int active, int age, node_t *next };

   // 'lock' represents the lock. It is the head of a shared queue. Initially, the queue is empty and *lock = nil. 'node'
      initially points to a thread-local node with the values {nil, nil, false, 0, nil}. 'list_lock' is a spinlock that
      protects the shared queue. 'count' is a counter whose initial value is 0.
3  function execute_cs(request_t req, node_t **lock, node_t *node, int *list_lock, int *count) : ret_val_t
4      var node_t *sup, *prev, *cur;
5      var int local_count;
6      node → req := req;
7      while true do
8          if ¬node → active then                                   // Request inactive?
9              node → active := true;                               // Make it active
10             repeat                                               // Enqueue it
11                 sup := *lock;
12                 node → next := sup;
13             until CAS(*lock, sup, node);
14         while *list_lock ≠ 0 and node → req ≠ nil
15             and node → active do                                 // Busy-wait while global lock taken,
16             pause();                                             // node active and request not executed
17         if node → req = nil then                                 // A combiner executed our request
18             return node → ret;                                   // We return the request
19         else if CAS(list_lock, 0, 1) then                        // No combiner did, become combiner
20             break;
21     if ¬node → active then                                       // Request inactive again?
22         node → active := true;                                   // Make it active
23         repeat                                                   // Enqueue it
24             sup := *lock;
25             node → next := sup;
26         until CAS(*lock, sup, node);
27     *count++;                                                    // Increment global counter
28     local_count := *count;                                       // Take snapshot of global counter
29     cur := *lock;
30     while cur ≠ nil do                                           // Now combiner: execute all requests
31         if cur → req ≠ nil then
32             Critical section:
33             <apply cur → req to object's state and store the return value to cur → ret>
34             cur → req := nil;
35             cur → age := local_count;
36         cur := cur → next;
37     if ¬(count mod CLEANUP_FREQUENCY) then                       // Once in a while, cleanup list
38         prev := *lock;                                           // (remove inactive nodes)
39         cur := prev → next;
40         while cur ≠ nil do
41             if cur → age + CLEANUP_OLD_THRESHOLD < local_count then
42                 prev → next := cur → next;
43                 cur → active := false
44             else
45                 prev := cur;
46             cur := prev → next;
47     *list_lock := 0;                                             // Release global lock
48     return lock → ret;                                           // Executed own request: return result
```

the global lock and become a combiner itself. When t becomes a combiner, it increments the global counter, then executes the critical sections of all the other threads. Sometimes, after the

combiner has executed all of the critical sections from the queue, it performs a *cleanup phase* in which it removes old nodes from the queue. In order to determine if a request is old enough to be removed from the queue, its age is calculated using the difference between the current value of the global counter and the value it had when its node last became a combiner. Finally, the combiner releases the global lock. Note that nodes are inserted at the head of the queue, and the combiner follows the queue from head to tail. Moreover, unlike with Oyama, when the combiner reaches the end of the list, it simply releases the global lock instead of going through the list again in case new nodes have enqueued themselves during the combining phase. Therefore, unlike Oyama, Flat Combining never leads to starvation.

Flat Combining, like Oyama, has the advantage of improving data locality. It has a few drawbacks however: first, it uses a global lock, which can get highly contended. Second, Flat Combining uses two parameters: (i) `CLEANUP_FREQUENCY`, which determines how often old nodes are removed from the queue, and (ii) `CLEANUP_OLD_THRESOLD`, which determines how old nodes must be before they get removed from the queue. Modifying the values of these parameters make it possible to switch between a lenient and an aggressive cleaning policy for the queue. With a lenient cleaning policy, many unused nodes pollute the queue, and the combiner has to skip them when it executes the streak of critical sections. This makes the critical path longer, especially since cache misses are required to fetch the nodes from the queue. Moreover, threads have to reenqueue themselves more often, which may cause contention on the head of the queue. With an aggressive cleaning policy, the global lock is held for longer, because it is held while the queue is being cleaned: this also makes the critical path longer. Therefore, calibrating the parameters of Flat Combining can be a hard task: since both an aggressive and a lenient cleaning policy can be detrimental to the performance of the algorithm, empirical measurements are needed to determine the policy that works best for the application and the hardware used.

Note on the chronology. The presentation of the lock algorithms until this point gave an overview of the state of the art of lock algorithms at the time the main contribution of this thesis, RCL (see Chapter 4), was designed. The next two algorithms, CC-Synch and DSM-Synch, were developed simultaneously with RCL and published only a few weeks before the submission of the RCL paper at USENIX ATC [71].

3.8 CC-Synch and DSM-Synch

This section presents CC-Synch and DSM-Synch [39], two algorithms proposed by Fatourou et al. that aim to improve on Flat Combining. Like Flat Combining, CC-Synch and DSM-Synch use a global queue, and application threads occasionally become combiners that execute streaks of requests. Moreover, CC-Synch and DSM-Synch are designed to be used with an efficient sequential combining algorithm, but without such an algorithm, CC-Synch and DSM-Synch can simply be used as fast lock mechanisms.

CC-Synch. The algorithm of CC-Synch is shown in Algorithm 7. Unlike Flat Combining, CC-Synch does not use a global lock. Instead, each thread busy-waits on its own `wait` variable inside of its node. The first thread to enqueue itself becomes the unique combiner, and it executes its own critical section as well as the critical sections from the queue until (i) it reaches the head of the queue or (ii) it has executed a total of `MAX_COMBINER_CS` critical sections. In the latter case, the combiner hands over the role of combiner to the next thread in the queue. When a combiner runs out of requests to execute, the queue is restored to its initial state, i.e., it contains

Algorithm 7: CC-Synch

```
1  structures:
2    node_t          { request_t req, ret_val_t ret, boolean wait, boolean completed, node_t *next };
   // 'lock' represents the lock. It is the tail of a shared queue that initially contains a dummy node with the values
   // {nil, nil, false, false, nil}. 'node' initially points to a thread-local node with the values {nil, nil, false, false,
   // nil}.
3  function execute_cs(request_t req, node_t **lock, node_t **node) : ret_val_t
4      var node_t *next_node, *cur_node, *tmp_node, *tmp_node_next;
5      var int counter := 0;
6      next_node := *node;                                  // The current thread uses a (possibly recycled) node
7      next_node → next := nil;
8      next_node → wait := true;
9      next_node → completed := false;
10     cur_node := atomic_swap(*lock, next_node);           // 'cur_node' is assigned to the current thread
11     cur_node → req := req;                               // The current thread announces its request
12     cur_node → next := next_node;
13     *node := cur_node;
14     while cur_node → wait do                             // The current thread busy-waits until it is unlocked
15         pause();
16     if cur_node → completed then
17         return cur_node → ret;                           // If the request is already applied, return its value
18     tmp_node := cur_node;                                // The current thread is the combiner
19     while tmp_node → next ≠ nil and counter < MAX_COMBINER_OPERATIONS do
20         counter++;
21         tmp_node_next := tmp_node → next;
22         Critical section:
23         <apply tmp_node → req to object's state and store the return value to tmp_node → ret>
24         tmp_node → completed := true;                    // tmp_node's req is applied
25         tmp_node → wait := false;                        // Signal the client thread
26         tmp_node := tmp_node_next;                       // Proceed to the next node
27     tmp_node → wait := false;
28     return cur_node → ret;
```

a single dummy node whose `wait` parameter is set to **`false`**: the next thread to enqueue itself will become the next combiner.

The `MAX_COMBINER_CS` parameter prevents starvation. The reason why CC-Synch needs this parameter when Flat Combining does not is that in Flat Combining, threads insert their new nodes at the head of the queue, therefore, the critical sections of newly inserted nodes will never be executed by the current combiner. With CC-Synch, new nodes are inserted at the tail of the queue, therefore, without the `MAX_COMBINER_CS` parameter, threads may enqueue themselves faster than the combiner executes critical sections, which would lead to starvation: the combiner would only ever execute critical sections for other threads and would never be able to hand over the role of combiner to another application thread. While picking a low value for `MAX_COMBINER_CS` is costly because it prevents long streaks of critical sections from being executed on the same node (which makes the lock behave more like a queue lock), picking a high value is only detrimental to fairness, which is only a minor concern in most applications. Therefore, picking a satisfactory value is not very complex: a large value ensures that the lock with perform well in most cases. Fatourou et al. recommend a value of n times the number of hardware threads of the architecture, with n being a small integer.

The fact that CC-Synch does not use a global queue fixes the potential problem of an contended global lock in Flat Combining, and since combiners follow the queue and hand over their role to the first thread in the queue whose request has not been executed, all inactive

Algorithm 8: DSM-Synch

```
1  structures:
2    node_t          { request_t req, ret_val_t ret, boolean wait, boolean completed, node_t *next };

   // 'lock' represents the lock. It is the tail of a shared queue that initially contains a dummy node with the values
   // {nil, nil, false, false,  nil}. 'nodes' is an array that contains two thread-local nodes, initialized with the values
   // {nil, nil, false, false, nil}. 'toggle' is a thread-local variable that determines which one of the thread-local
   // nodes is currently used.
3  function execute_cs(request_t req, node_t **lock, node_t nodes[2], int *toggle) : ret_val_t
4      var node_t *tmp_node, *my_node, *my_pred_node;
5      var int counter := 0;
6      *toggle := 1 - *toggle;                          // The current thread toggles its toggle variable
7      my_node = nodes[*toggle];                        // The current thread chooses to use one of its nodes
8      my_node → wait := true;
9      my_node → completed := false;
10     my_node → next := nil;
11     my_node → req := req;                            // The current thread announces its request
12     my_pred_node := atomic_swap(*lock, my_node);     // The current thread inserts my_node in the list
13     if my_pred_node ≠ nil then                       // If a node already exists in the list
14         my_pred_node → next := my_node;              // Fix 'next' of previous node
15         while my_node → wait do                      // The current thread busy-waits until it is unlocked
16             pause();
17         if my_node → completed then                  // If the current thread's request is already applied
18             return my_node → ret;                    // The current thread returns its return value
19     tmp_node := my_node
20     while true do                                    // The current thread is the combiner
21         counter++;
22         Critical section:
23         <apply tmp_node → req to object's state and store the return value to tmp_node → ret>
24         tmp_node → completed := true;                // The request of 'tmp_node' is applied
25         tmp_node → wait := false;                    // Signal the client thread
26         if tmp_node → next = nil or
27            tmp_node → next → next = nil or
28            counter ≥ MAX_COMBINER_OPERATIONS then
               // The current thread helped H threads or fewer than 2 nodes are in the list
29             break;
30         tmp_node := tmp_node → next;                 // Proceed to the next node
31     if tmp_node → next = nil then                    // The request is the only record in the list
32         if CAS(*lock, tmp_node, nil) then
33             return my_node → ret;
34         while tmp_node → next = nil do               // Some thread is appending a node
35             yield();                                 // Wait until it finishes its operation
36     tmp_node → next → wait := false;                 // Unlock next node's owner
37     return my_node → ret;
```

nodes are located before the combiner in the queue: the queue never requires any cleanup. This removes the need for the two `CLEANUP_FREQUENCY` and `CLEANUP_OLD_THRESHOLD` parameters, and improves performance since the cleanup of Flat Combining makes the critical path longer.

DSM-Synch. DSM-Synch is a modified version of CC-Synch that aims to perform better on non-cache-coherent architectures. With CC-Synch, each thread owns a dedicated node in the beginning of the algorithm, but when a thread needs to enqueue itself, the dummy node at the tail of the queue becomes its new node, and its old node becomes the new dummy node. Therefore, with CC-Synch, even if the nodes are initially allocated locally, threads are not ensured to always use locally allocated nodes because nodes are handed over between threads. As shown

in Algorithm 8[3], DSM-Synch uses two nodes per thread per lock, and threads always use one of them when they need to enter the queue: if these two nodes are allocated locally, all threads will only ever use locally allocated nodes. Consequently, with CC-Synch, the long busy-wait loop used by threads to wait for the execution of their critical sections (lines 16 and 17 in Algorithm 7) busy-waits on data that may not have been allocated locally, whereas with DSM-Synch, the corresponding busy-wait loop (lines 16 and 17 or Algorithm 7) only busy-waits on data that has been allocated locally as long as the nodes were initially allocated locally. This optimization was designed for non-cache-coherent architectures, for which busy-waiting on remote data can use a lot of interconnect bandwidth and may lack reactivity. The optimization also makes it possible to busy-wait on a locally allocated variable on NUMA architectures, but since the synchronization variable is not contended (one per client) and only accessed in read mode, performance gains over CC-Synch should be negligible on these architectures as the variable will typically remain cached until it is invalidated just before the corresponding client exits its busy-wait loop. DSM-Synch is slightly more complex than CC-Synch: DSM-Synch uses more atomic instructions and two busy-wait loops (lines 15-16 and 34-35 of Algorithm 8) instead of one.

3.9 Comparison of lock algorithms

This section compares the lock algorithms presented heretofore based on the metrics presented in the beginning of the chapter. Figure 3.1 illustrates that comparison.

Reactivity and performance under high contention. All locks that use busy-waiting are very reactive because they do not require context switches between the execution of critical sections as is the case with blocking locks. The basic spinlock performs very poorly under high contention due to the fact that it uses a single synchronization variable. Queue locks (CLH, MCS and MCS-TP) perform better because they use one synchronization variable per thread. Oyama and Flat Combining perform even better because they execute streaks of critical sections without needing synchronization between them other than signaling threads when their critical section has been executed. However, Oyama and Flat Combining still use a global lock. CC-Synch and DSM-Synch remove the global lock completely.

Number of atomic instructions. The basic spinlock issues a lot of contended atomic instructions when many threads busy-wait on the same synchronization variable. In all queue locks (CLH, MCS, MCS-TP), Oyama, and combining locks (Flat Combining, CC-Synch, and DSM-Synch), threads use atomic instructions to insert their nodes into the global queue. Oyama and Flat Combining use an internal global spinlock, which also makes them use a significant amount of potentially contended atomic instructions.

Ordering and starvation. Most lock algorithms execute critical sections in FIFO order with the exception of (i) the basic spinlock, in which the fastest thread to request the lock acquires it, and (ii) Oyama, since it uses a LIFO queue. For this reason, basic spinlocks can lead to starvation (one thread that is faster than the others may always obtain the lock first). Oyama can also cause starvation if critical sections are added to the queue at a very high rate, i.e., too fast for the thread that executes them to ever reach the end of the queue. Starvation is impossible for

[3]The implementation of DSM-Synch proposed in Fatourou et al.'s original paper [39] resets `tmp_node->next` to `NULL` between lines 36 and 37. This operation is not present in the official implementation [38] of DSM-Synch. It seems to cause a lost update of the `tmp_node->next` variable that can cause a deadlock of the combiner, by making it loop indefinitely at lines 34-35.

	Reactivity	Performance under high contention	Number of atomic instructions	Ordering of critical sections	Starvation possible	Resistance to preemption	Number of parameters	Choosing efficient parameters	Data locality of internal structures	Data locality in critical sections	Usability in legacy applications
Basic spinlock	Good	Poor	Very high	Random	Yes	Average	0	-	Poor	Average	Easy
Blocking lock	Poor	Good	Average	FIFO	No	Good	0	-	Average	Average	-
CLH	Good	Average	Average	FIFO	No	Very poor	0	-	Good	Average	Medium
MCS	Good	Average	Average	FIFO	No	Very poor	0	-	Better	Average	Medium
MCS-TP	Good	Average	Average	FIFO	No	Good	5	Hard	Better	Average	Medium
Oyama	Good	Good	High +	LIFO	Yes	Average	0	-	Average	Good	Hard
Flat Combining	Good	Good	High	FIFO	No	Average	2	Hard	Average	Good	Hard
CC-Synch	Good	Very good	Average	FIFO	No	Average	1	Easy	Good	Good	Hard
DSM-Synch	Good	Very good	Average	FIFO	No	Average	1	Easy	Better	Good	Hard

Figure 3.1: Comparison of lock algorithms

Flat Combining, because threads enqueue themselves at the head of the queue, i.e., behind the combiner. CC-Synch and DSM-Synch use the parameter `MAX_COMBINER_OPERATIONS` to prevent starvation.

Resistance to preemption. All locks that use busy-waiting and a global queue are prone to convoys, as will be seen in Section 5.3.5.2. Therefore, their resistance to preemption is very low. The resistance to preemption of other locks is average, except for MCS-TP which was specifically designed to be resistant to preemption and convoys.

Parameters. Locks that do not use parameters always run at optimal performance, while locks that use parameters may require fine-tuning to perform well. In particular, MCS-TP is hard to configure because it uses five parameters, and some of them, such as the upper bound on the length of critical sections, depend both on the architecture and the application used: such values can only be determined precisely through complex profiling. The parameters used by Flat Combining make it possible to choose between a lenient or aggressive cleanup policy for the queue, but both policies can be detrimental, therefore, choosing efficient parameters requires empirical evaluation. CC-Synch and DSM-Synch use a single parameter for which a large value can safely be chosen for good performance, even if too large a value may be detrimental to fairness.

Data locality of internal structures. The basic spinlock has very poor data locality on its shared synchronization variable because all threads concurrently apply compare-and-swap operations on it: the shared variable "ping-pongs" between the caches of all cores. CLH has better locality because different threads busy-wait on different synchronization variables. MCS and MCS-TP have better locality for their synchronization variables on non-cache-coherent architectures than CLH because they ensure that each thread busy-waits on its own node. Oyama and Flat Combining use both a global lock and a local synchronization variable for each node in the global queue, therefore, their data locality is average for internal structures. CC-Synch uses one synchronization variable per thread, like CLH. DSM-Synch, like MCS, also ensures that each thread always busy-waits on its own queue node, therefore, it should waste less bandwidth and be more reactive on non-cache-coherent architectures.

Data locality in critical sections. Oyama, Flat Combining, CC-Synch and DSM-Synch all improve data locality by making threads execute streaks of critical sections: since critical sections

of a given lock often protect a set of shared variables, these variables are likely to remain in a local cache during the execution of several critical sections.

Usability in legacy applications. Legacy applications typically use blocking locks, implemented as POSIX locks, because in contrast with other lock algorithms, blocking locks work properly on architectures with a single hardware thread. Legacy applications can easily switch to the basic spinlock, because it uses the same interface as POSIX locks: the `lock()` and `unlock()` functions take only one argument. The `lock()` and `unlock()` functions of queue locks (CLH, MCS, and MCS-TP) take two arguments instead of one for POSIX locks: this issue can either be solved with some light refactoring or with thread-local variables, as was explained in section 3.3. Additionally, the K42 variant of MCS described in Section 3.4 can be used. Implementing condition variables for the basic spinlock using POSIX primitives is trivial using the algorithm described in Section 3.2, and the same technique can be used for queue locks. Finally, using Oyama or combining locks (Flat Combining, CC-Synch and DSM-Synch) in legacy applications is difficult for two reasons. First, these locks need critical sections to be encapsulated into functions, which requires a lot of code refactoring, and no solution to this problem is provided by their authors. And second, implementing condition variables in these locks is challenging, because they could cause a server/combiner thread to sleep and therefore prevent remaining critical sections from being executed. Moreover, since server/combiner threads are normal application threads, any application thread could randomly be put to sleep, which could cause undesirable unexpected effects such as deadlocks.

3.10 Other lock algorithms

The previous sections have presented a set of lock algorithms that range from a very basic spinlock implementation to recently-published state-of-the-art combining locks. However, in the past decades, many other lock algorithms have been proposed. This section presents a few notable examples.

Backoff locks improve on the basic spinlock, with the objective to make it perform better when many threads perform concurrent lock acquisition attempts. The main idea of backoff locks is to make threads sleep for a *backoff delay* in the busy-wait loop. Doing so reduces contention and also has the advantage of saving powing and CPU time. The delay typically increases at each iteration, often linearly (*linear backoff lock*) or exponentially (*exponential backoff lock*). According to Anderson et al. [5], increasing the delay exponentially is the most efficient strategy. The ticket lock [75] also aims to improve the performance of spinlocks under high contention. It uses two shared counters: the first one contains the number of lock acquisition requests, while the other one contains the number of times the lock has been released. In order to acquire the lock, a thread atomically reads and increments the value of the first counter with a fetch-and-add instruction (see Section 2.3.1.2.b) and busy-waits until the second counter is equal to the value it has read from the first counter. The advantage the ticket lock has over the basic spinlock or backoff locks is that threads busy-wait on the second counter using an instruction that only reads the corresponding cache line instead of an instruction that attempts to write to it (e.g., a compare-and-swap instruction). Mellor-Crummey et al. [76] show that both backoff locks and the ticket lock are slower than MCS under high contention. However, David et al. [30] show that the ticket lock performs better than a wide range of other locks under low contention, and, given its small memory footprint, they recommend its use over more complex lock algorithms unless it is sure sure that a specific lock will be very highly contended.

Abellan et al. propose GLocks [1] in order to provide fast, contention-resistant locking for multicore and manycore architectures. The key idea of GLocks is to use a token-based message passing protocol that uses a dedicated on-chip network implemented in hardware instead of the cache hierarchy. Since the resources needed to build this network grow with the number of supported Glocks, Abellan et al. recommend to only use them for the most contended locks and to use spinlocks otherwise. The main drawback of GLocks is that they require specific hardware support not provided by current machines.

Finally, given the large amount of proposed lock algorithms, choosing one is not always a simple task. Smartlocks [37] aim to solve this issue by dynamically switching between existing lock algorithms in order to choose the most appropriate one at runtime. They use heuristics and machine learning in order to optimize towards a user-defined goal which may relate to performance or problem-specific criteria. In particular, on heterogeneous architectures, Smartlocks are able to optimize which waiter will get the lock next for the best long-term effect when multiple threads are busy-waiting for a lock.

Hierarchical locks. Hierarchical locks trade fairness for throughput by executing several critical sections consecutively on the same *cluster* of hardware threads (core, die, CPU, or NUMA bank). Doing so allows for better throughput, since synchronization local to a cluster is faster than global synchronization, for two reasons: (i) critical sections executed on the same cluster can reuse shared variables that are stored in their common caches, and (ii) the synchronization variables used for busy-waiting are allocated on a local NUMA node, which may reduce the overhead of busy-waiting, as shown in Section 2.5.3.2. The Hierarchical Backoff Lock (HBO) [90] is a backoff lock with an adaptive delay that favors hardware threads of the same cluster: they are granted shorter backoff times, whereas remote hardware threads are granted longer backoff times. The Hierarchical CLH lock (H-CLH) [73] creates a CLH-style queue for each cluster, and the thread at the head of each local queue occasionally splices the local queue into a global queue. Critical sections are executed following the global queue, as if it were a traditional CLH queue. However, given the way the global queue is built, nodes from the same cluster are neighbors in the global queue and their critical sections are executed consecutively.

Dice et al. [33] propose to combine Flat Combining and MCS to create efficient hierarchical locks: each cluster uses one instance of Flat Combining that efficiently creates local MCS queues of threads, and merges them into a global MCS queue. The lock is handed over in the MCS queue exactly like with a MCS lock, except the global queue created by the combiners is ordered by clusters, like with H-CLH. However, Dice et al.'s approach is more efficient than H-CLH because with H-CLH, all threads need to enqueue themselves by using an atomic instruction that is applied to the global tail of the queue, which can cause bottlenecks. Moreover, with H-CLH, threads must know which thread is the master of their local cluster, which complicates their busy-waiting semantics.

Finally, Dice et al. propose Lock Cohorting [34], a general technique that makes it possible to build hierarchical locks from any two non-hierarchical lock algorithms G and S, as long as these lock algorithms satisfy certain (widespread) properties. The general idea is to use one instance of S per cluster, and one global instance of G. The first thread to acquire a lock acquires both G and S, and then releases only S if other threads from the same cluster are waiting for the lock (these threads will only have to acquire S to own the lock), otherwise, it releases both G and S to let threads from other clusters acquire the lock. Dice et al. use Lock Cohorting to build new hierarchical locks by choosing either a backoff lock, the ticket lock, MCS, or CLH for G and for S. They show that some of the resulting combinations outperform both HBO and H-CLH.

3.11 Conclusion

This section has presented and compared the properties of a wide range of lock algorithms, including spinlocks, blocking locks (that are used in most legacy applications through the POSIX library), queue locks (CLH, MCS, MCS-TP), and combining locks (Flat Combining, CC-Synch, DSM-Synch). All of these lock algorithms have drawbacks. The next section presents the main contribution of this thesis, Remote Core Locking (RCL), a lock mechanism that aims to solve these drawbacks while providing additional performance improvements over state-of-the-art lock algorithms such as CC-Synch and DSM-Synch.

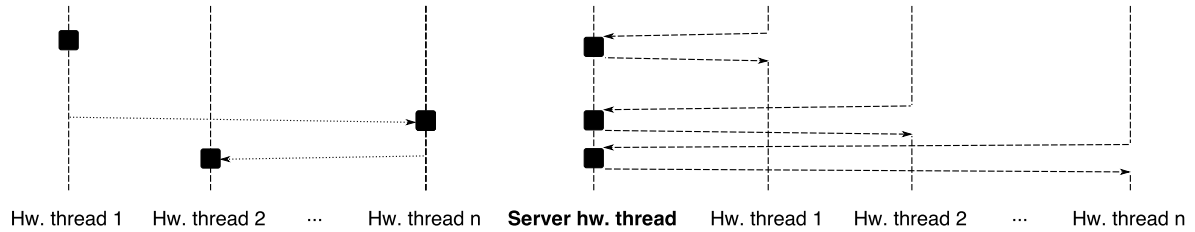
Chapter 4

Contribution

This chapter presents the main contributions of the research presented in the thesis. Section 4.1 presents Remote Core Locking (RCL), a lock mechanism designed to improve the performance of applications on multicore architectures, and compares it with the lock algorithms described in Chapter 3. Section 4.2 describes a profiler that helps predict which locks may benefit from RCL, and a reengineering that makes it possible to transform a legacy application into an application that uses RCL with a minimal amount of work. Finally, Section 4.3 concludes.

4.1 Remote Core Lock

RCL goes one step further than combining locks by fully dedicating a *server* hardware thread to the execution of critical sections. RCL has two main advantages. First, it reduces synchronization overhead, because the *client* threads and the server communicate using a fast client/server cache-aligned messaging scheme that is similar to the one used in Barrelfish [9]. And second, RCL improves data locality, because the shared variables that the critical sections protect are all accessed from the same dedicated server hardware thread, where no application thread can be scheduled. Therefore, the shared variables are more likely to always remain in that hardware thread's cache hierarchy.



(a) Traditional locks (b) RCL

Figure 4.1: Critical sections with traditional locks vs. RCL

RCL has additional advantages. In particular, the fact that the server hardware thread is dedicated to the execution of critical sections ensures that the server threads can never be preempted by application threads: the server always makes progress on the critical path.

Furthermore, the fact that server threads are not application threads make it possible for RCL to implement condition variables.

If more than one lock is used by the application, the RCL server can handle the critical sections of several locks, but doing so results in *false serialization*, i.e., the unnecessary serialization of independent critical sections. To alleviate this issue, more servers can be used, each of them executing the critical sections of one or several locks. RCL is not meant to be used for all locks, however: RCL's strongest point is its high performance under high contention, and the lock algorithms used for uncontended locks have negligible impact of performance, therefore, RCL should only be used for contended locks. As shown in Section 5.3.1, the number of contended locks in an application is generally low enough that only a few RCL servers are needed to reach optimal performance. Since modern multicore architectures provide a lot of hardware threads whose processing power cannot always be harvested efficiently because applications lack scalability, many hardware threads are often left unused by applications: these hardware threads can be used for RCL servers in order to improve performance.

In order to use RCL, it is necessary to choose the locks for which RCL is expected to be beneficial and to encapsulate the critical sections associated with these locks into functions as is the case with Oyama and combining locks. This chapter first describes the core RCL algorithm, then presents a profiling tool that helps developers choose which locks to implement using RCL as well as a reengineering that encapsulates critical sections into functions in order to make them directly usable with RCL.

4.1.1 Core algorithm

With RCL, critical sections are replaced by remote procedures call to a server that executes the code of the critical sections. In order to implement the remote procedure call, each server owns an array of request structures (Figure 4.2) that is used for communication with the clients. This array is $C \cdot L$ bytes long, where C is a constant representing the maximum number of clients (a large number, typically much higher than the number of available hardware threads), and L is the size of the hardware cache line. Each request structure req_i is L bytes and allows communication between a specific client c_i and the server. The array is aligned so that each structure req_i is mapped to a single cache line.[1] The array of requests is allocated on the NUMA bank of the server on NUMA architectures.

The first three machine words of each request req_i contain respectively: (i) the address of the lock associated with the critical section, (ii) the address of a structure encapsulating the *context*, i.e., the variables referenced or updated by the critical section that are declared by the function containing the critical section code, and (iii) the address of a function that encapsulates the critical section for which the client c_i has requested the execution, or `NULL` if no critical section execution is requested.

Client side. In order to execute a critical section, a client c_i first writes the address of the lock into the first word of the structure req_i, then writes the address of the context structure into the second word, and finally writes the address of the function that encapsulates the code of the critical section into the third word. The client then actively waits for the third word of req_i to be reset to `NULL`, which indicates that the server has executed the critical section. In order to improve energy efficiency on x86 architectures, if there are less clients than the

[1] If the architecture uses several cache line sizes, the largest cache line size is chosen in order to avoid *false sharing* (see Section 2.3.1.2.c).

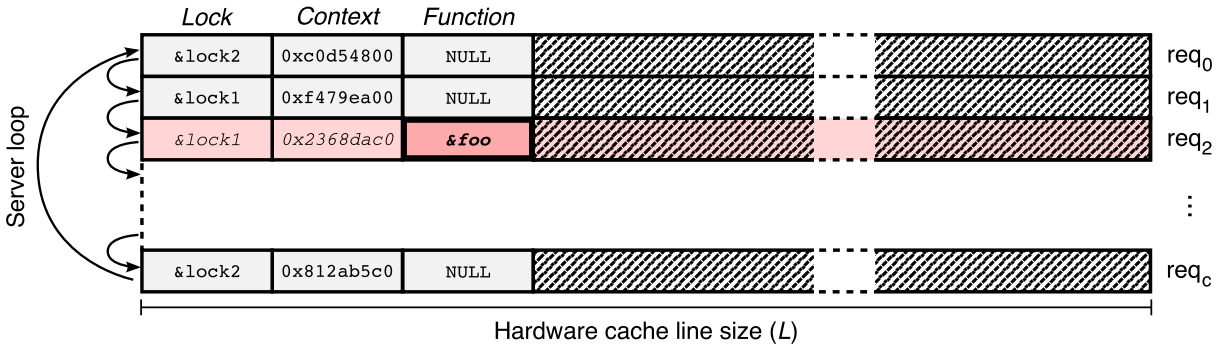


Figure 4.2: The request array

number of available hardware threads, the SSE3 `MONITOR`/`MWAIT` instructions can be used to avoid busy-waiting: the client hardware thread (assuming clients threads are bound to hardware threads) will physically sleep and be woken up automatically when the server writes into the third word of req_i. Optionally, the busy-wait loop of the client may yield the processor at each iteration, as will be discussed in Section 5.3.5.3.

Server side. A servicing thread iterates over the requests, waiting for one of the requests to contain a non-`NULL` value in its third word. When such a value is found, the servicing thread checks whether the requested lock is free and, if so, acquires the lock and executes the critical section using the function pointer and the context. When the servicing thread is done executing the critical section, it resets the third word to `NULL`, and resumes iterating over the request array.

4.1.2 Implementation of the RCL Runtime

Legacy applications may use ad hoc synchronization mechanisms and rely on libraries that themselves may block or busy-wait. The core algorithm, described in Section 4.1.1, only refers to a single servicing thread, and thus requires that this thread is never blocked at the operating system level and never busy-waits. This section describes how the RCL runtime ensures liveness and responsiveness in these cases, and presents implementation details.

4.1.2.1 Ensuring liveness and responsiveness

Three kinds of situations may induce liveness or responsiveness issues if the server uses a single servicing thread. First, the servicing thread may be blocked at the operating system level. This can happen when a critical section tries to acquire a POSIX lock that is already held, performs an I/O, or waits on a condition variable, for instance. Second, the servicing thread may enter a busy-wait loop if a critical section tries to acquire a nested RCL or a lock that uses busy-waiting, or if it uses some other form of ad hoc synchronization [111]. Finally, the servicing thread may be preempted at the operating level, either because its timeslice expires [81] or because of a page fault. Blocking and waiting within a critical section may cause a deadlock, because the servicing thread is unable to execute critical sections associated with other locks, even when doing so may be necessary to allow the blocked critical section to unblock. Additionally, blocking, of any form, including waiting and preemption, degrades the responsiveness of the server because a blocked thread is unable to serve other locks managed by the same server.

To solve these issues RCL uses a pool of servicing threads on each server to ensure liveness and responsiveness, as is described in the next paragraphs.

Ensuring liveness. Blocking and waiting within a critical section raise a problem of liveness, because a single servicing thread cannot execute critical sections associated with other locks, even when doing so may be necessary to allow the blocked critical section to unblock. A pathological case happens when the servicing thread busy-waits in a critical section while waiting for a variable to be set by another critical section that is handled by the same server but is protected by a different lock: this situation is illustrated by clients c_2 and c_3 on Figure 4.3. To ensure liveness, the pool of servicing threads on each server ensures that when a servicing thread blocks or waits, there is always at least one other *free* servicing thread that is not currently executing a critical section, and this servicing thread will eventually be elected. To ensure the existence of a free servicing thread, the RCL runtime provides a *management thread*, which is activated regularly at each expiration of a *timeout* (set to the operating system's timeslice value) and runs at highest priority. When activated, the management thread checks that at least one of the servicing threads has made progress since the last activation of the management thread, using a server-global flag `is_alive`. The management thread clears this flag just before sleeping, and any servicing thread that enters a critical section sets it. If the management thread observes that `is_alive` is cleared when it wakes up, it assumes that all servicing threads are either blocked or waiting. In this case, it checks that a free thread exists in the pool of servicing threads and, if this is not the case, it adds a new one.

Ensuring responsiveness. Blocking, of any form, including waiting and preemption, degrades the responsiveness of the server because a blocked servicing thread is unable to serve other locks managed by the same server. The RCL runtime implements a number of strategies to improve responsiveness issues introduced by the underlying operating system and by RCL design decisions.

As explained in Section 3.5, a well-known problem in the use of locks is the risk that the operating system will preempt a thread at the expiration of a timeslice while it is executing a critical section, thereby extending the duration of the critical section and thus increasing contention [81]. RCL dedicates a pool of threads on each dedicated sever hardware thread to the execution of critical sections, which makes it possible to manage these threads according to a scheduling policy that does not use preemption. The POSIX FIFO scheduling policy is used, because it both respects priorities, as is needed to ensure liveness, and makes it so that that all threads keep running until they are blocked or manually yield the processor. The use of the FIFO policy raises a liveness issue: if a servicing thread is executing a busy-wait loop, it will never be preempted by the operating system, and a free thread will never be elected. To solve this, when no progress is detected by the manager thread, it elects a servicing thread by first decrementing and then incrementing the priorities of the other threads, effectively moving them to the end of the FIFO queue. The use of the FIFO policy also implies that when a servicing thread unblocks after blocking in a critical section, it is placed at the end of the FIFO queue. If there are many servicing threads, there may be a long delay before the unblocked thread is rescheduled. To minimize this delay, the RCL runtime tries to always minimize the number of servicing threads: a servicing thread leaves the pool whenever it observes that there is at least one other free servicing thread, since that other thread will be able to handle any requests.

The RCL management thread ensures the liveness of the server but only reacts after a timeout. When all servicing threads are blocked in the operating system, the operating system's scheduler is used to elect a new free thread immediately. Concretely, the RCL runtime maintains a *backup thread* that runs at a lower priority than all servicing threads. The FIFO scheduling policy never

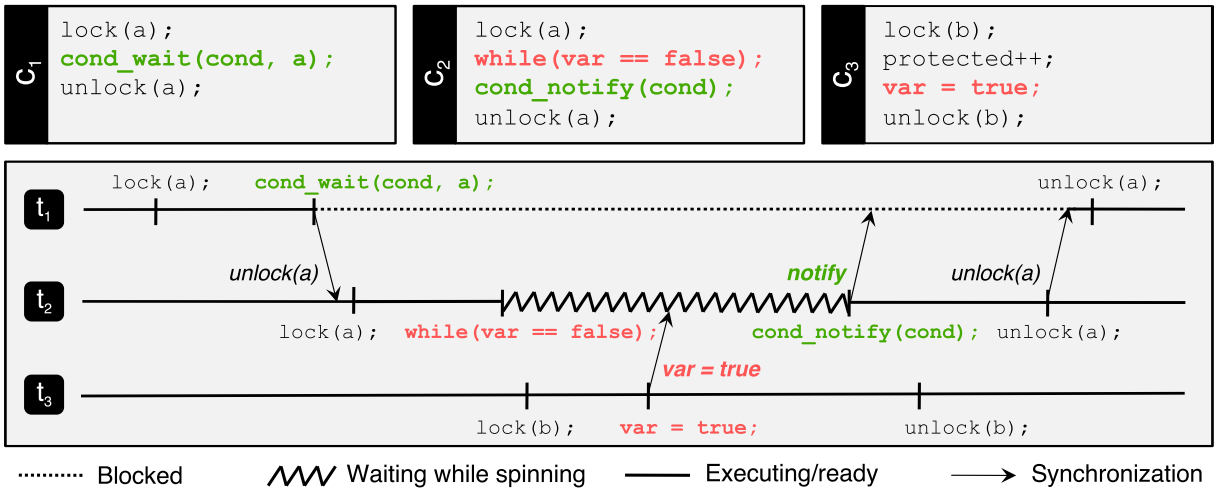


Figure 4.3: Ad hoc synchronization example

elects a lower priority thread when a higher priority thread can run, and thus the backup thread is only elected when all servicing threads are blocked. When the backup thread is elected, it adds a new servicing thread, which immediately preempts the backup thread and can service the next request.

Finally, when a critical section needs to execute a nested RCL managed by the same hardware thread and the lock is already owned by another servicing thread, the servicing thread immediately yields the processor in order to allow the owner of the lock to release it.

Example. Figure 4.3 presents a complete example of ad hoc synchronization in a critical section that illustrates a worst case scenario in terms of active waiting. Initially, thread t_1 is the only servicing thread, and it handles a critical section of client c_1. That critical section tries to acquire a condition variable (`cond_wait()` function call), causing t_1 to block at the operating system level. At that point, there is no more runnable servicing thread. Therefore, the operating system's scheduler immediately elects the backup thread, which adds a new free servicing thread t_2 that immediately preempts the backup thread. Thread t_2 executes c_2's request. Client c_2's critical section causes t_2 to busy-wait on the variable `var`. At some point, the management thread awakens, and clears the server-global `is_alive` flag. Then, at the management thread's next activation, since t_1 is blocked in the operating system and t_2 is busy-waiting, no servicing thread has been able to set `is_alive`. Since there is no free servicing thread at that point, the management thread creates a new servicing thread, t_3, and elects it, as it is now the only servicing thread that has not recently been elected. Therefore, at that point, the server has three servicing threads, t_1, t_2 and t_3, one of which, t_3, is free. When t_3 executes the request from c_3, it modifies the value of `var`.

After executing client c_3's request, thread t_3 detects that there are other servicing threads running. Therefore, it yields the processor in order to allow them to run. Thread t_2 is the only other servicing thread that is not blocked at the operating system level, and consequently, it is elected. Since the critical section executed by t_3 has set `var`, thread t_2 can exit its busy-wait loop and unblock the condition variable, causing t_1 to awaken. Client c_2's critical section ends, and t_2 detects that the server has two free servicing threads: itself and t_3. Consequently, thread t_2 leaves the servicing pool. Thread t_3 is then elected, but it notices that there is no pending critical section. Furthermore, since it notices that another servicing thread is running and that

Algorithm 9: Executing a critical section (client)

```
1  thread-local variables:
2    int client_index;
3    boolean is_server_thread;
4    server_t *my_server;
5  function execute_cs(lock_t *lock, function_t *function, void *context)
6      var request_t *request := &lock→server→requests[client_index];
7      if ¬is_server_thread or my_server ≠ lock→server then    // RCL to a remote hardware thread
8          request→lock := lock;
9          request→context := context;
10         request→function := function;
11         while r→function ≠ nil do
12             pause();
13         return;
14     else                                                    // Local nested lock
15         while local_compare_and_swap(&lock→is_locked, false, true) = true do
16             yield();                                        // Give a chance to other thread to release lock
17         context := function(context);                       // Execute the critical section
18         lock→is_locked := false;
19         return;
```

thread is not free, it yields the processor to let it run. Thread t_1 is therefore elected, and it completes client c_1's request, before noticing that two servicing threads are free, itself and t_3. Consequently, it leaves the pool of servicing threads: the server is back to its initial state, except its only servicing thread is now t_3.

4.1.2.2 Algorithm details

This section describes client and the server sides of the algorithm in more detail. Algorithm 9 shows the `execute_cs()` function that is used by the client to execute a critical section. Algorithm 10 shows the code executed by the servicing threads, and Algorithm 11 shows the code executed by the manager and backup threads.

Executing a critical section. In order to request the execution of a critical section with RCL, a client thread simply submits its request by filling the `lock`, `context`, and `function` fields of its `request` structure in the `requests` array, as seen in lines 8-13 of Algorithm 9. Servicing threads use the same method to request the execution of a critical section that is managed by a remote RCL server. However, in order to request the execution of a critical section that is managed by the local RCL server, a servicing thread must ensure that the lock is free, and wait until it is otherwise. In order to give a chance to the servicing thread that owns the lock to finish executing its critical section, the thread repetitively yields the processor (lines 11-12).

Servicing threads. As explained in Section 4.1.1, servicing threads iterate over the request array. In order to avoid the need to reallocate the request array when new client threads are created, its size is fixed and chosen to be very large (256KB). Moreover, the client identifier allocator implements an adaptive long-lived renaming algorithm [17] that keeps track of the highest client identifier and tries to reallocate smaller ones.

Algorithm 10 shows the code executed by servicing threads. Only the *fast path* (lines 9-22) is executed when the pool of servicing threads contains a single thread. A *slow path* (lines 12-21) is executed when the pool contains several servicing threads. Lines 9-15 of the fast path implement the RCL server loop as described in Section 4.1.1 and indicates that the servicing

Algorithm 10: Structures and servicing thread (server)

```
1  structures:
   lock_t:          { server_t *server, boolean is_locked };
   request_t:       { function_t *function, void *context, lock_t *lock };
   thread_t:        { server_t *server, int timestamp, boolean is_servicing };
2  server_t:        { List<thread_t *> all_threads, LockFreeStack<thread_t *> prepared_threads,
                     int number_of_free_threads, int number_of_servicing_threads,
                     int timestamp, boolean is_alive, request_t *requests }
3  global variables:
4    int                number_of_clients;
5  function servicing_thread(thread_t *thread)
6      var server_t* server := t→server;
7      var request_t* request, lock_t* lock, function_t* function;
8      while true do
9          server→is_alive := true;
10         thread→timestamp := server→timestamp;
11         local_fetch_and_add(s→number_of_free_threads, −1);          // This thread is not free anymore
12         for i := 0 to number_of_clients do
13             request := server→requests[i];
14             if request→function ≠ nil then
15                 lock := request→lock;
16                 if local_compare_and_swap(&lock→is_locked, false, true) = false then
17                     function := request→function;
18                     if function ≠ nil then
19                         request→context := function(request→context);
                           // Execute the critical section
20                         request→function := nil;                    // Signal completion to the client
21                         lock→is_locked := false;
22         local_fetch_and_add(s→number_of_free_threads, 1);           // This thread is now free
23         if server→number_of_servicing_threads > 1 then              // Some servicing threads are blocked
24             if server→number_of_free_threads ≤ 1 then
25                 yield();                                            // Allow other servicing threads to run
26             else
27                 thread→is_servicing := false;                       // Keep only one free servicing thread
                   local_fetch_and_add(server→number_of_servicing_threads, −1);
28                 local_atomic_insert(server→prepared_threads, thread);
                   // Atomic since the manager may wake up thread before call to sleep() (on Linux, use futexes)
29                 atomic(<if ¬thread→is_servicing then sleep();>)
```

thread is not free during its execution by decrementing (line 11) and incrementing (line 22) `number_of_free_threads`. Because the thread may be preempted due to a page fault, all operations on variables shared between the threads, including `number_of_free_threads`, must be atomic. However, since servicing threads of a RCL server are all bound to that server's dedicated hardware thread, the atomic instructions that only use server-local data only need to be executed atomically in the context of their hardware thread. Therefore, local versions of the atomic operations that do not clean up the write buffers are used. This is done by not using the `LOCK` prefix for atomic instructions on x86 architectures, for instance. These local atomic operations are never contended and they are much less costly than regular atomic instructions because they do not require additional synchronization between hardware threads.

The slow path is executed if the active servicing thread detects that other servicing threads exist (line 23). If the other servicing threads are all executing critical sections (line 24), the

Algorithm 11: Management and backup threads (server)

```
1  function management_thread(server_t *server)
2      var thread_t *thread;
3      server→is_alive := false;
4      server→timestamp := 1;
5      while true do
6          if server→is_alive = false then
7              server→is_alive := true;
8              if s→number_of_free_threads =0 then        // Ensure a thread can handle remote requests
                   // Activate prepared thread or create new thread
9                  local_fetch_and_add(server→number_of_servicing_threads, 1);
10                 local_fetch_and_add(server→number_of_free_threads, 1);
11                 thread := local_atomic_remove(s→prepared_threads);
12                 if thread = nil then
13                     thread := allocate_thread(s);
14                     insert(s→all_threads, thread);
15                     thread→is_servicing := true;
16                     thread.start(prio_servicing);
17                 else
18                     thread→is_servicing := true;
19                     wakeup(thread);
20             while true do                              // Elect thread that has not recently been elected
21                 for thread in server→all_threads do
22                     if thread→is_servicing = true
23                         and thread→timestamp < server→timestamp then
24                         thread→timestamp = server→timestamp;
25                         elect(thread);
26                         goto end;
27                 server→timestamp++;                    // All threads were elected once, begin a new cycle
28         else
29             server→is_alive := false;
30         sleep(timeout);
31 function backup_thread(server_t *server)
32     while true do
33         server→is_alive := false;
34         <wake up the management thread>
```

servicing thread yields the processor (line 25) to let them run. Otherwise, it removes itself from the pool and sleeps (lines 27-29).

Management and backup threads. As explained in Section 4.1.2.1, each RCL server runs one manager thread and one backup thread. If, on wake up, the management thread notices, based on the value of `is_alive`, that none of the servicing threads have progressed since the previous timeout, it ensures that at least one free thread exists (Algorithm 11, lines 8-19) and forces the election (lines 20-27) of a thread that has not recently been elected. The backup thread (lines 31-34) simply sets `is_alive` to **`false`** and wakes up the management thread.

4.1.3 Comparison with other locks

In Section 3.9, Figure 3.1 compared the lock algorithms presented in Chapter 3. Using the same criteria, Figure 4.4 compares RCL with these lock algorithms. This section compares RCL with other lock algorithms by discussing Figure 4.4 in detail.

	Reactivity	Performance under high contention	Number of atomic instructions	Ordering of critical sections	Starvation possible	Resistance to preemption	Number of parameters	Choosing efficient parameters	Data locality of internal structures	Data locality in critical sections	Usability in legacy applications
Basic spinlock	Good	Poor	Very high	Random	Yes	Average	0	-	Poor	Average	Easy
Blocking lock	Poor	Good	Average	FIFO	No	Good	0	-	Average	Average	-
CLH	Good	Average	Average	FIFO	No	Very poor	0	-	Good	Average	Medium
MCS	Good	Average	Average	FIFO	No	Very poor	0	-	Better	Average	Medium
MCS-TP	Good	Average	Average	FIFO	No	Good	5	Hard	Better	Average	Medium
Oyama	Good	Good	High +	LIFO	Yes	Average	0	-	Average	Good	Hard
Flat Combining	Good	Good	High	FIFO	No	Average	2	Hard	Average	Good	Hard
CC-Synch	Good	Very good	Average	FIFO	No	Average	1	Easy	Good	Good	Hard
DSM-Synch	Good	Very good	Average	FIFO	No	Average	1	Easy	Better	Good	Hard
RCL	Good +	Very good +	Only local	Other	No	Good	0	-	Very good	Very good	Medium

Figure 4.4: Comparison of lock algorithms with RCL

Reactivity and performance under high contention. Similarly to combining locks (Flat Combining, CC-Synch and DSM-Synch), RCL is very reactive and performs well under high contention, thanks to the fact that (i) it uses one synchronization variable per client and (ii) it executes streaks of critical sections without needing synchronization between them, other than signaling threads when their critical section has been executed. Moreover, unlike Flat Combining, RCL does not need a global lock to hand over the role of server, since server threads keep their role during the whole execution. The fact that the server role is never handed over is also an advantage compared to combining locks since the handover process lengthens the critical path.

Number of atomic instructions. Unlike the basic spinlock, Oyama, and Flat Combining, RCL does not use global synchronization variables on which threads busy-wait using atomic instructions. Moreover, unlike queue locks (CLH, MCS, MCS-TP) and combining locks (Flat Combining, CC-Synch, DSM-Synch), RCL does not use a global queue in which application threads must insert their nodes using atomic instructions. The only atomic instructions that are used by RCL are local to a single hardware thread and therefore never suffer from contention.

Ordering and starvation. With RCL, critical sections are not served in the FIFO order: at each iteration of a servicing thread, critical sections are served following the ordering of threads in the request array. However, starvation is impossible, since the fact that a servicing thread loops over the request array ensures that when a thread t_1 asks for the execution of its critical section, at most one critical section from the same lock by another thread t_2 may be executed before the execution of t_1's critical section.

Resistance to preemption. RCL is immune to the issue of preemption inside of a critical section since it dedicates a hardware thread for the execution of critical sections: a busy-waiting client thread cannot preempt the lock holder because it will never be scheduled on the dedicated hardware thread. Therefore, in contrast with other locks, *the server always makes progress* when it executes critical sections, which ensures that the critical path will be as short as possible.

Parameters. RCL does not use any parameters. This is an advantage because fine-tuning parameters for a specific machine and a specific lock usage can be time-consuming.

Data locality of internal structures. Since the threads of each RCL server are bound to a dedicated hardware thread, they benefit from very good data locality: all data and synchronization

structures used by a RCL server are allocated on its local NUMA bank, and no application thread may be scheduled on the server hardware thread, thereby replacing data from the caches with its own (on architectures with multiple hardware threads per core, application threads may pollute the server's caches by being allocated on the same core, however). Additionally, in order to avoid of cache misses, the request structures in the request array are directly mapped to cache lines by being cache-aligned and large enough to fill a whole cache line.

Data locality in critical sections. Similarly to other combining locks, RCL improves data locality by making some threads execute streaks of critical sections. Since critical sections of a given lock often protect a set of shared variables, these variables may remain in a local cache during the execution of at least part of a streak. However, RCL takes one step further by ensuring that these threads are bound to a specific server hardware thread. Therefore, the data they handle never has to be migrated between hardware threads: the shared data that is accessed by critical sections is likely to remain in the server's caches during the whole execution. Moreover, the fact that no client thread may be scheduled on server hardware threads removes the risk of application threads polluting the caches with their data.

Usability in legacy applications. Like combining locks (Oyama, Flat Combining, CC-Synch and DSM-Synch), RCL cannot be used directly in legacy applications, because they require critical sections to be encapsulated into functions. However, RCL comes with a reengineering, proposed in Section 4.2.2 to solve this issue. Moreover, RCL is able to handle condition variables thanks to a pool of threads on the servers, without risking to put the combiner to sleep as is the case with combining locks.

4.2 Tools

This section describes two tools that were written to facilitate the use of RCL for application developers. Section 4.2.1 presents a profiler that makes it possible to predict with reasonable accuracy which locks from which applications may benefit from RCL, and Section 4.2.2 presents a reengineering that automatically transforms the code of legacy applications to make them use RCL.

4.2.1 Profiler

In order to help the user decide which locks to transform into RCLs, a profiler was implemented as a dynamically loaded library that intercepts calls involving POSIX locks, condition variables, and threads. Since RCL improves the speed of lock acquisitions under high contention as well as data locality, an application that may benefit from RCL either suffers from high lock contention or its critical sections suffer from poor data locality. The profiler measures two metrics: (i) the overall percentage of time spent in critical sections including lock acquisitions and releases, which helps detect applications that suffer high lock contention, and (ii) the average number of cache misses in critical sections, which helps detect applications whose critical sections suffer from poor data locality. As shown in the evaluation (Chapter 5), these metrics make it possible to reliably predict if an application can benefit from RCL. The profiler can also measure the two metrics for every lock, and provide per-lock information. It identifies locks with the file name and line number where they were allocated, and returns a list of the backtraces taken at the allocation points of each lock. Each lock is identified by a hash of the backtrace taken at its allocation point, and the profiler can be run again with the identifier of a lock in order to measure more precisely the two metrics for a particular lock. Measuring the global time spent in critical sections and the

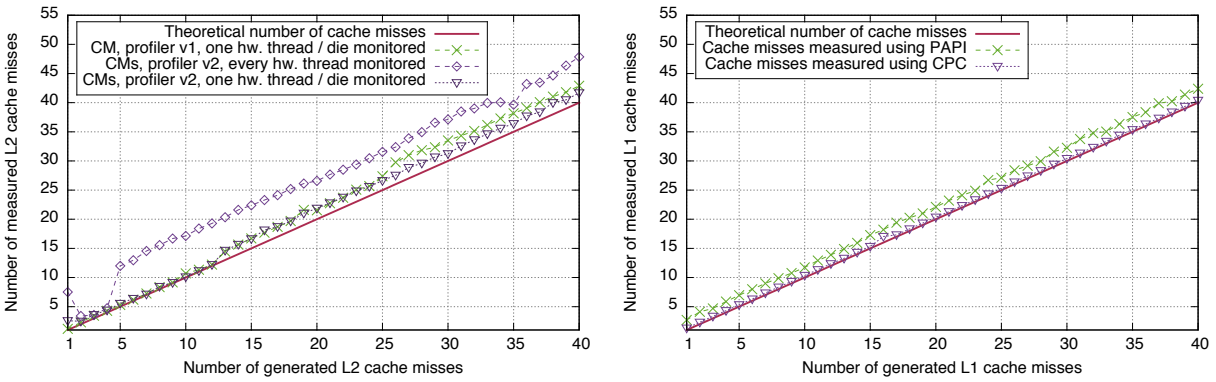
global number of cache misses in critical sections make it possible to identify which applications may benefit from RCL, and per-lock information helps deciding which locks to transform into RCLs in an application.

Implementation notes. The *first version* of the profiler, written for the USENIX ATC paper [71], only supported x86 architectures running Linux, because its evaluation of RCL only used a machine that was similar to Magnycours-48 (see Section 2.5.1). The new evaluation of RCL that is presented in this thesis uses both Magnycours-48 and Niagara-128 (see Section 2.5.2). Niagara2-128 uses a SPARC architecture and runs Solaris. Therefore, the profiler had to be ported to Solaris in order to be used with Niagara2-128. Its accuracy has also been improved, both for measuring the time spent in critical sections and the number of cache misses. The resulting *second version* of the profiler has been used for the evaluation in Chapter 5. The remainder of this section presents the improvements that were introduced in the second version of the profiler.

In order to measure the time spent in critical sections, the hardware timestamp counter must be read before and after critical sections. For this, the first version of the profiler used the Performance Application Programming Interface (PAPI) [58, 18], a cross-platform library that unifies the use of hardware performance counters on various architectures. PAPI provides the `PAPI_get_real_cyc()` function to read cycles, unfortunately, the values returned by this function are often imprecise. On Linux/x86 (Magnycours-48), PAPI uses the costly `gettimeofday()` system call to get the current time in microseconds, and multiplies it by the clock frequency to get a number of cycles. On Solaris/SPARC (Niagara2-128), PAPI reads the value from the timestamp counter, loses precision by converting it into microseconds, and then converts it again into cycles by multiplying the result by the CPU's frequency. The needless conversion operations incur a non-negligible overhead, moreover, microseconds are not precise enough for a cycle measurement: on Niagara2-128, for instance, the smallest non-zero value measured between two samples is $1,165$ cycles, which is exactly one microsecond since the hardware clock frequency is 1.165GHz. Solaris provides `gethrtime()` to read high resolution time, but this call still converts the results into nanoseconds, which incurs an overhead. The second version of the profiler directly uses assembly instructions to read the timestamp counters. While the overhead of PAPI is negligible in most cases on Magnycours-48, on Niagara2-128, using PAPI (resp. `gethrtime()`) can cause an overhead of 170% (resp. 18%) relative to reading the timestamp counter directly with assembly instructions.

The profiler measures the average number of cache misses inside critical sections, not including lock acquisition and release. Measuring cache misses can be tricky, because CPUs do not always make it possible to measure them on all hardware threads concurrently. On Magnycours-48, measuring cache misses simultaneously on all cores leads to erroneous results: it seems that the cache miss hardware counters are shared on each die, which is not clearly stated in the official documentation. Similarly, on Niagara2-128, measuring cache misses on all hardware threads simultaneously leads to erroneous results because the hardware counter registers are not replicated for each hardware thread. This was fixed by binding threads to hardware threads, and only measuring cache misses on one hardware thread per die on Magnycours-48, and on one hardware thread per core on Niagara2-128.

To ensure that the second version of the profiler is working properly, a custom benchmark that generates a known number of cache misses in its critical sections was written. This benchmark simply creates one worker thread per core, and each worker thread (i) repeatedly executes critical sections that are protected by POSIX locks, and (ii) generates cache misses between its critical sections by busy-waiting on a shared variable. Each critical section modifies the same set of n



(a) Magnycours-48

(b) Niagara2-128, with profiler v2

Figure 4.5: Tuning the profiler: number of cache misses

cache lines. To ensure that the critical sections always generate n cache misses, (i) threads never execute two critical sections consecutively: another thread has to execute its critical section in-between, which invalidates the cache lines from the local cache, and (ii) all data prefetching is avoided by using non-contiguous cache lines and interdependent cache line accesses (the address of the next cache line to read is built from the value read in the previous cache line [113]). This benchmark generates at least as many L1 and L2 cache misses as there as there are cache line accesses on Magnycours-48, since on that machine, only the L1 and the L2 caches are local to each core, and it generates as many L1 cache misses as there are cache line accesses on Niagara2-128, since on that machine, only the L1 cache is local to each core.

The benchmark is run with different versions and/or configurations of the profiler. On Magnycours-48, L2 cache misses are measured, since they are more often triggered by inter-core communication than L1 cache misses: since the L1 cache is smaller, it is subject to more cache misses that are caused by cache line eviction. Results for this machine are shown in Figure 4.5a for the first version of the profiler when one hardware thread per die is monitored, for the second version of the profiler when all hardware threads are monitored concurrently, and for the second version of the profiler when one hardware thread per die is monitored. With the second version of the profiler, measuring cache misses on all hardware threads results in measuring up to seven extra cache misses. However, measuring cache misses on only one hardware thread per die gives accurate results (never more than one extra cache miss). Also note that the number of instructions used to monitor cache misses has been reduced in the second version of the profiler compared to the first version: the number of extra cache misses measured is reduced (-1) when the number of generated cache misses is large (>25).

Measuring the number of cache misses is done with the PAPI library on Linux. The second version of the profiler, which supports Solaris, can also measure cache misses using the native Solaris CPU Performance Counter (CPC) library on that operating system. On Niagara2-128, L1 cache misses measured, since only the L1 cache is local to cores: L2 cache misses would only measure communication between the two processors (i.e., between the first 64 hardware threads and the last 64 hardware threads), while L1 cache misses are a better measurement of inter-core communication. Figure 4.5b shows the profiling results on Niagara2-128, with the PAPI library as well as the CPC library. As can be seen on the figure, on Solaris, using PAPI measures approximatively two extra cache misses, whereas with CPC, with the CPC library, the number of measured cache misses is very close to the theoretical number of cache misses.

Listing 1: Critical section from Raytrace

```
1  int GetJob(RAYJOB *job, int pid)
2  {
3    ...
4    ALOCK(gm->wplock, pid)                                    /* Lock acquisition */
5    wpentry = gm->workpool[pid][0];
6
7    if (!wpentry) {
8      gm->wpstat[pid][0] = WPS_EMPTY;
9      AULOCK(gm->wplock, pid)                                 /* Lock release */
10     return (WPS_EMPTY);
11   }
12
13   gm->workpool[pid][0] = wpentry->next;
14   AULOCK(gm->wplock, pid)                                   /* Lock release */
15   ...
16 }
```

Since not all hardware threads can be monitored concurrently in order to obtain precise results, the number of cache misses per critical section can only be measured accurately with one profiled run if all application threads perform the same tasks and generate a similar number of cache misses. This is not the case for Memcached, an application used in the evaluation (Chapter 5), for instance. In that case, the application is run n times with the profiler,where n is the number of hardware threads per die (resp. per core) on Magnycours-48 (resp. on Niagara2-128), and each time, the i^{th} $(0 < i \leq n)$ hardware thread of each die (resp. core) is monitored. The results are then averaged.

4.2.2 Reengineering legacy applications

If the profiling results show that some locks used by the application can benefit from RCL, the developer must reengineer all critical sections that may be protected by the selected locks as a separate function that can be passed to the RCL server. This reengineering amounts to an "Extract Method" refactoring [43]. It was implemented with the help of Julia Lawall using the program transformation tool Coccinelle [83], in 2115 lines of code.

The main problem in extracting a critical section into a separate function is to bind the variables used by the critical section code. The extracted function must receive the values of variables that are initialized prior to the critical section and read within the critical section, and return the values of variables that are updated in the critical section and read afterwards. Only variables local to the function are concerned, alias analysis is not required because aliases involve addresses that can be referenced from the server. Listing 1 shows a critical section from the Raytrace benchmark from the SPLASH-2 suite presented in Section 2.5.3.3, and Listing 2 shows how it is transformed by the reengineering. The critical section of lines 4-14 of Listing 1 is protected by the `ALOCK()` and `AULOCK()` macros that are transformed by the C preprocessor into calls to the POSIX library functions `pthread_mutex_lock()` (lock acquisition) and `pthread_mutex_unlock()` (lock release), respectively. After transformation, the code of this critical section is encapsulated into the `cs()` function declared at line 6 of Listing 2, and an union named `context` whose instances will be able to hold the variables used by the critical section is defined at line 1 of Listing 2. To run the critical section, an instance of the `context` union is declared and filled with the values of the variables that are read by the critical section The critical section is then run through a call to the `execute_cs()` function from the RCL runtime (line 35 of Listing 2): this function takes three parameters, the lock, the address of the function that encapsulates the critical section, and the address of the instance of the `context` union. Finally, results are read from the `context` union (line 36 of Listing 2), since this union

Listing 2: Critical section from Listing 1, after transformation

```
1  union context {
2    struct input { int pid; } input;
3    struct output { WPJOB *wpentry; } output;
4  };
5
6  void cs(void *ctx) {
7    struct output *outcontext = &(((union instance *)ctx)->output);
8    struct input *incontext = &(((union instance *)ctx)->input);
9    WPJOB *wpentry;
10   int pid = incontext->pid, int ret = 0;
11
12   /* Start of original critical section code */
13   wpentry = gm->workpool[pid][0];
14   if (!wpentry) {
15     gm->wpstat[pid][0] = WPS_EMPTY;
16     /* End of original critical section code */
17
18     ret = 1;
19     goto done;
20   }
21   gm->workpool[pid][0] = wpentry->next;
22   /* End of original critical section code */
23
24 done:
25   outcontext->wpentry = wpentry;
26   return (void *)(uintptr_t)ret;
27 }
28
29 int GetJob(RAYJOB *job, int pid)
30 {
31   int ret;
32   union instance instance = { pid, };
33
34   ...
35   ret = execute_cs(&gm->wplock[pid], &function, &instance);
36   wpentry = instance.output.wpentry;
37   if (ret) { if (ret == 1) return (WPS_EMPTY); }
38   ...
39 }
```

contains either the input (variables that are read) or the output (variables that are written) of the critical section, before and after the call to `execute_cs()`, respectively. As an optimization, if the context amounts to a single variable, this variable is located in the empty space at the end of the client's cache line (hatched space in Figure 4.2).[2] The reengineering also addresses a common pattern in critical sections, illustrated in lines 7-11 of Listing 1, where a conditional in the critical section releases the lock and returns from the function. In this case, the code is transformed such that the critical section returns a flag value indicating which lock release operation ends the critical section, and the code following the call to `execute_cs()` executes the code following the lock release operation that is indicated by the corresponding flag value (line 37 of Listing 2).

The reengineering also modifies various other POSIX functions to use the RCL runtime. In particular, the function for initializing a lock receives additional arguments indicating whether the lock should be implemented as an RCL. Finally, the reengineering also generates a header file, incorporating the profiling information, that the developer can edit to indicate which lock initializations should create POSIX locks and which ones should use RCLs. The header also makes it possible to choose which RCL servers are used for each lock.

[2]More values could be passed in the client's cache line if they fit in the request array, but simple experiments have shown that the performance gain is negligible.

4.3 Conclusion

This section has presented RCL, a synchronization mechanism that aims to perform better than other lock algorithms on modern multicore machines, by dedicating server hardware threads in order to decrease synchronization times and improve data locality. RCL is implemented in a runtime, and it is provided along with two tools: (i) a profiler that make it possible to identify which legacy applications may benefit from RCL, and (ii) a reengineering that automatically transforms these applications in order to make them use RCL. In the next chapter, a wide range of applications is run through the profiler, and using the reengineering, the applications for which the profiler found that RCL may be beneficial are transformed to use it. On these applications, the performance of RCL is compared with that of the locks presented in Chapter 3.

Chapter 5

Evaluation

This chapter describes how developers can use RCL to improve the performance of their applications, and evaluates the performance of RCL relative to other lock algorithms presented in Chapter 3 on a wide range of applications. Section 5.1 presents the Liblock, a library that makes it possible to easily switch between lock algorithms in legacy applications: the Liblock is used for all performance experiments in this chapter. Section 5.2 presents a microbenchmark that is used to compare the performance of RCL with that of other lock algorithms. Section 5.3 presents a methodology that helps developers detect which applications could benefit from RCL. The profiler is then run on a set of legacy applications, and RCL as well as other lock algorithms are evaluated on the subset of applications that were identified by the profiler as being good candidates for RCL. Section 5.4 concludes.

The evaluation presented in this chapter is more detailed than the one presented in the USENIX ATC paper [71] that proposed RCL. Moreover, while the evaluation from the USENIX ATC paper only used one machine that was similar to Magnycours-48 (see Section 2.5.1), the evaluation presented in this chapter uses two machines, Magnycours-48 and Niagara2-128 (see Section 2.5.2). These machines use two different architectures and operating systems. Finally, in addition to the lock algorithms that were used in the evaluation of the USENIX ATC paper, the evaluation presented in this chapter also evaluates the performance of CC-Synch and DSM-Synch.

Note on the figures. In this chapter, all graphs and tables present data points and values that are averaged over five runs.

5.1 Liblock

In this chapter, the performance of RCL is compared to that of other lock algorithms on a microbenchmark (Section 5.2) as well as other applications (Section 5.3). The lock algorithms RCL is compared to have all been presented in Chapter 3. They are the following: the basic spinlock, the POSIX lock, MCS, MCS-TP, Flat Combining, CC-Synch and DSM-Synch. This set of lock algorithms consists of: (i) a very basic lock algorithm (the basic spinlock), (ii) one of the most widespread lock algorithms (the POSIX lock), (iii) queue locks (MCS and MCS-TP), and (iv) combining locks (Flat Combining, CC-Synch and DSM-Synch), two of them (CC-Synch and DSM-Synch) being state-of-the-art lock algorithms that were designed at the same time as RCL. CLH is not evaluated since it should perform similarly to MCS on cache-coherent architectures, as explained in Section 3.3. The performance of Oyama is not evaluated either,

because other combining locks use the same basic idea as Oyama, with clear improvements, as explained in that same section. Since MCS-TP is a lock algorithm that only aims to improve the performance of MCS when threads often get preempted, it is only evaluated in experiments that sometimes run more application threads than there are hardware threads on the machine. Finally, hierarchical locks were not evaluated because they are usually based on non-hierarchical locks and a hierarchical version of RCL could be designed, possibly using the Lock Cohorting [34] technique described in Section 3.10.

Traditional locks can easily be replaced by non-combining locks in legacy applications by modifying the functions that acquire and release locks. However, with combining locks, critical sections have to be encapsulated into functions, because they have to be shipped to the combiner. Therefore, legacy applications have to be transformed to use these locks, using the same transformation that was presented for RCL in Section 4.2.2.

In order to ease the comparison of lock algorithms, a library named Liblock was written. It makes it possible to easily switch between lock implementations in an application. The application must use critical sections that are encapsulated into functions: the critical sections of the microbenchmark (see Section 5.2) are, and the legacy applications used in Section 5.3 are all transformed using the reengineering that was presented in Section 4.2.2. Using the transformed application instead of replacing the functions that acquire and release the locks has negligible performance for non-combining locks.

The Liblock's extensible design makes it possible for developers to make it support any lock algorithm. It comes with an implementation of each of the locks used in this chapter. The following paragraphs give more information on these implementations.

Lock implementations in the Liblock. For other lock algorithms than RCL, official implementations are used when available. Special attention is given to preserve memory barriers, prefetcher hints, and on Solaris, hints to prevent threads from being descheduled (`schedctl_start()`/`schedctl_stop()` function calls); some of these elements are occasionally added to the official implementations when they are found to improve performance. RCL aligns its mailbox structure on cache lines and is careful to allocate server data structures on the server's NUMA bank. In order to be as fair as possible, the nodes in the lists of the queue locks (MCS and MCS-TP) and combining locks (Flat Combining, CC-Synch and DSM-Synch) are cache-aligned and allocated on their thread's local NUMA bank (threads are bound in most experiments, as explained in Section 5.3.2). Recommended parameter values are used when available. Moreover, the parameter space is partially explored when needed in order to ensure that lock algorithms use satisfactory values for their parameters.

The implementation of MCS is straightforward and does not use any parameters. For MCS-TP, the implementation provided by the authors [48] is used. As seen in Section 3.5, MCS-TP uses three parameters: (i) an upper bound on the length of critical sections (`MAX_CS_TIME` in Algorithm 4), for which a value of 10 milliseconds is chosen: this value is defined by measuring the length of the critical sections of all applications used in the evaluation and by picking the lowest possible upper bound with a reasonable margin, (ii) the approximate length of time it takes a thread to see a timestamp published on another thread (`UPDATE_DELAY` in Algorithm 5): a value of 10 microseconds is chosen since it is sufficient to prevent all deadlocks, and (iii) the patience to wait in the queue (`PATIENCE` in Algorithm 4): a value of 50 microseconds is chosen, which is the value used in the original paper where MCS-TP is proposed [49]. Exploring the parameter space locally shows that these values constitute a local optimum. For Flat Combining, the original authors' code is used [50], as well as the parameter values they use. The

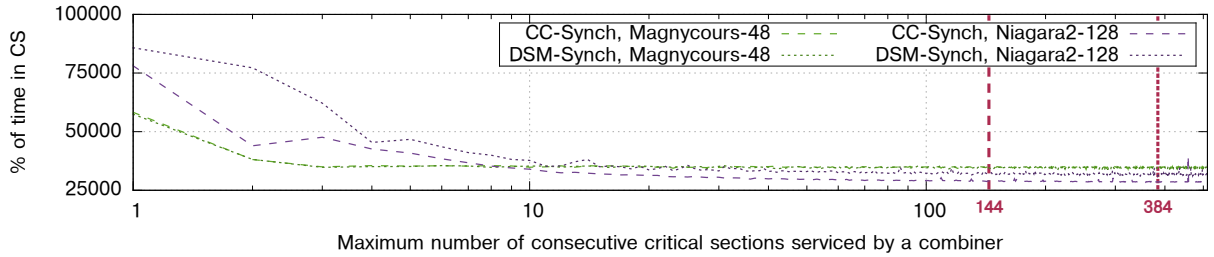


Figure 5.1: Influence of `MAX_COMBINER_CS` on CC-Synch and DSM-Synch

cleanup frequency (`CLEANUP_FREQUENCY` in Algorithm 6) is set to 100, and the cleanup threshold (`CLEANUP_OLD_THRESHOLD`) is set to 10. Again, local exploration of the parameter space shows that Flat Combining performs well with these parameter values.

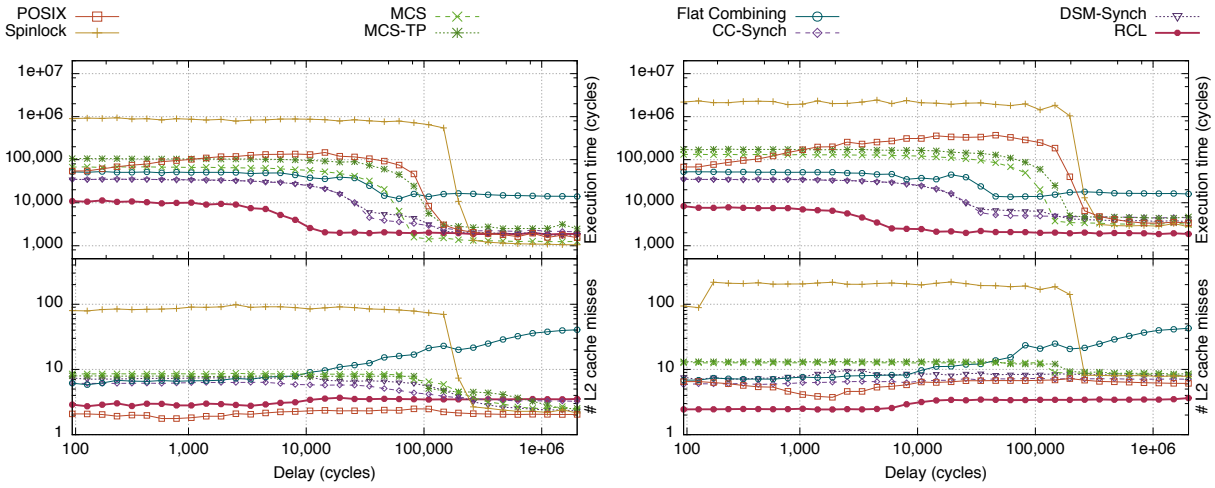
CC-Synch and DSM-Synch use a single parameter, `MAX_COMBINER_CS` in Algorithms 7 and 8, which specifies the maximum number of critical sections a combiner services before handing over the role of combiner to another thread. The original paper [39] recommends using a value of $n \times h$ with n being a small integer. The implementation proposed by the authors [38] uses a value of $n = 3$. It corresponds to a value of `MAX_COMBINER_CS` $= 3 \times h_{m48} = 144$ for Magnycours-48, and `MAX_COMBINER_CS` $= 3 \times h_{n128} = 384$ for Niagara2-128. Figure 5.1 shows the results of the microbenchmark at highest contention (delay of 100) when `MAX_COMBINER_CS` varies between 1 and 500. Both 144 and 384 are located in an area of the graph where the latency of CC-Synch and DSM-Synch is minimal, therefore, these values are used for `MAX_COMBINER_CS`.

The implementation of the RCL runtime used on Magnycours-48 is described in Section 4.1.2. On Solaris 10, using POSIX FIFO scheduling requires superuser privileges. Since a privileged user account could not be obtained on Niagara2-128, a degraded version of the RCL runtime that does not use POSIX FIFO scheduling is used for all experiments on that machine. The degraded version does not use backup or manager threads on RCL servers, and servicing threads can preempt each other when they are running on the same server hardware thread. Privileged access to a machine that runs Solaris will be needed to port the non-degraded version of the RCL runtime to that operating system: the current non-degraded Linux implementation will not run without modifications on Solaris since it uses Linux-specific mechanisms. For instance, in the non-degraded Linux implementation, the use of Futexes is needed to avoid a priority inversion that may be triggered by the FIFO scheduling, and Solaris does not provide a mechanism that is similar to Futexes.

5.2 Microbenchmark

A microbenchmark was developed in order to measure the performance of RCL relative to other lock algorithms. For other locks than RCL, the microbenchmark executes critical sections repeatedly on all h hardware threads[1] (one software thread per hardware thread, bound), except one that manages the lifecycle of the threads. In the case of RCL, critical sections are executed on $h - 2$ hardware threads only, since one hardware thread manages the lifecycle of threads and another one is dedicated to a RCL server. A single RCL server is needed, because all critical sections are protected by the same lock. The microbenchmark varies the *degree of contention*

[1] In this chapter, the total number of hardware threads of a machine is always noted h. The total number of hardware threads of Magnycours-48 and Niagara2-128 are noted h_{m48} (= 48) and h_{n128} (= 128), respectively.



(a) One shared cache line per CS

(b) Five shared cache lines per CS

	High contention (10^2 cycles)				Low contention (2.10^6 cycles)			
	Exec. time	**# cycles**	**Locality**	**# misses**	**Exec. time**	**# cycles**	**Locality**	**# misses**
Spinlock	Bad	2,186,419	Bad	94.0	Average	2,822	Average	7.7
POSIX	Average good	67,740	Average good	6.5	Average	3,321	Average	6.1
MCS	Average bad	132,448	Average bad	12.7	Average	3,086	Average	8.1
MCS-TP	Average bad	171,854	Average bad	13.1	Average	4,332	Average	8.4
Flat Combining	Average good	52,358	Average good	6.9	Bad	16,017	Bad	44.2
CC-Synch	Average good	35,194	Average good	5.9	Average	4,359	Average	7.1
DSM-Synch	Average good	35,067	Average good	7.6	Average	3,588	Average	7.8
RCL	*Good*	*8,298*	*Good*	*2.4c + 0.0s*	*Good*	*1,899*	*Good*	*2.4c + 1.2s*

(c) Comparison of the lock algorithms for five shared cache lines

Figure 5.2: Microbenchmark results on Magnycours-48

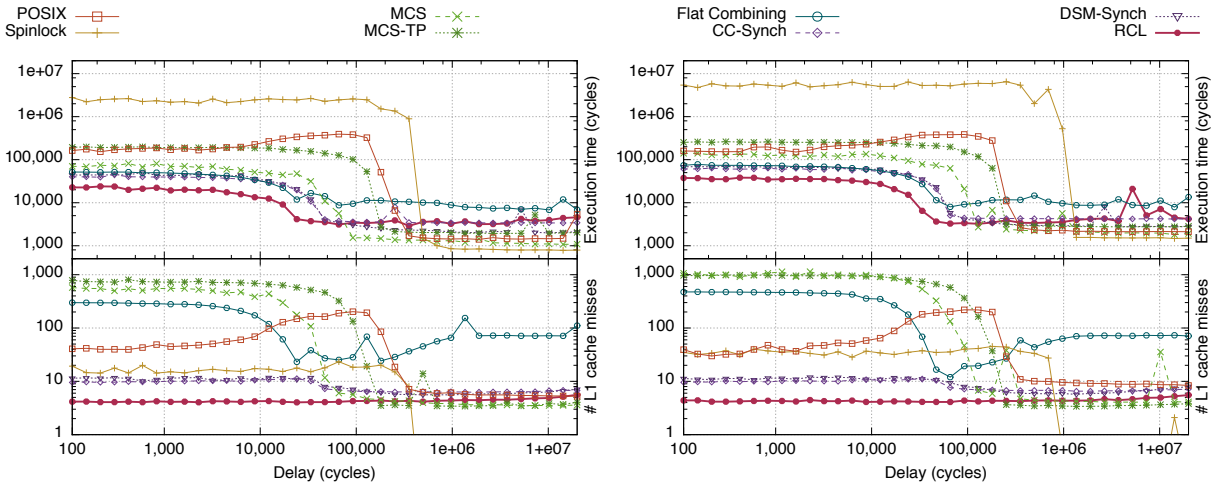
on the lock by varying the delay between the execution of the critical sections: the shorter the delay, the higher the contention. The microbenchmark also varies the *locality* of critical sections by making them read and write either one or five cache lines. To ensure that cache lines are not prefetched, they are not contiguous (Magnycours-48's prefetcher always fetches two cache lines at a time), and the address of the next memory access is built from the previously read value [113] when accessing cache lines. This technique is similar to the one described the benchmark that was used to fine-tune the second version of the profiler in Section 4.2.1.

Magnycours-48. Figure 5.2a presents the average execution time of a critical section (top) and the number of L2 cache misses (bottom) when each thread executes 10,000 critical sections that each access one shared cache line on Magnycours-48. This experiment mainly measures the effect of lock access contention. Figure 5.2b presents the increase in execution time when each critical section accesses five cache lines instead. This experiment focuses more on the effect of data locality of shared cache lines. In practice, as seen in Figure 5.5, in the evaluated applications, most critical sections trigger between one and five cache misses, therefore, performing either one or five cache line accesses is realistic. Highlights of the experiment are summarized in Figure 5.2c.

On Magnycours-48, under high contention (left part of the graph), with five cache line accesses, RCL is several times faster than all other lock algorithms.[2] CC-Synch and DSM-Synch are ~323% slower than RCL, even though they are state-of-the-art algorithms that were published shortly before RCL. Both algorithms have comparable performance, as expected on a cache-coherent architecture (see Section 3.8). Flat Combining, on which these algorithms are based, performs 49% slower than them: the removal of the global lock of Flat Combining for handing over the role of combiner is a very efficient optimization. MCS is much slower than combining locks: it is ~277% slower than CC-Synch and DSM-Synch, and 153% slower than Flat Combining. This overhead comes from the fact that with MCS and non-combining locks, synchronization between two threads is necessary between the execution of two critical sections, whereas combining locks execute dozens of critical sections consecutively without any synchronization as long as the list of requests is full. Let $h_{m48} = 48$ be the number of hardware threads provided by Magnycours-48. Since, under high contention, all $h_{m48} - 1 = 47$ threads are waiting for the execution of a critical section, on average, threads have to wait for the execution of $h_{m48} - 2 = 46$ other critical sections before they can execute theirs. Therefore, the synchronization overhead between two threads in the list is paid 46 times when waiting for the execution of each critical section. MCS-TP makes MCS more resilient to preemption, as shown later in this section, however, this optimization comes at a cost: MCS-TP has an overhead of 30% relative to MCS. The performance of the POSIX lock is reasonably good at very high contention (between that of Flat Combining and MCS), but its performance decreases when contention is average: it becomes worse than that of MCS and MCS-TP. This comes from the fact that the POSIX lock directly tries to acquire the lock using the global lock variable before sleeping: when contention is very high, i.e., the delay between two critical sections is very low, a thread that just released the lock is able to reacquire it immediately if doing so is faster than waking up the next blocked thread in the list that is waiting for the critical section. When contention is less high, a context switch is needed every time the lock is handed over from one thread to the next, which slows down every lock acquisition. Again, this overhead is paid 46 times since a thread typically has to wait for the execution of 46 other critical sections before it executes its own. Finally, the basic spinlock's performance under high contention is very poor because its repeated use of atomic compare-and-swap instructions saturates the memory bus with messages from the cache-coherence protocol.

The performance of lock algorithms at low contention does not matter as much as the performance under high contention, because when contention is low, locks are not a bottleneck. On Magnycours-48, most lock algorithms have comparable performance at low contention (right part of the graph). Flat Combining performs worse than other lock algorithms at low contention because after the execution of each critical section, the thread executes the loop at lines 37-46 of Algorithm 7 in order to clean up the global list. This loop accesses all the remote nodes in the list which increases the number of cache misses (44.2) and decreases performance. RCL is as efficient as other lock algorithms under low contention when the microbenchmark only accesses one shared cache line, but when it accesses five cache lines, RCL becomes more efficient than them, because the additional cache line accesses do not incur a overhead in the case of RCL: all accessed variables remain in the cache of the server hardware thread. This is not the case with combining locks because combiners only execute streaks of request under high contention: when contention is low, a thread that needs to execute a critical section becomes the combiner, executes its own critical section, sees that no other thread has added a request for the execution

[2]Using the SSE3 `MONITOR/MWAIT` instructions on the client side when waiting for a reply from the server, as described in Section 4.1.1, induces a overhead of less than 30% at both high and low contention. This makes the energy-efficient version of RCL quantitatively similar to the original RCL implementation presented here.



(a) One shared cache line per CS

(b) Five shared cache lines per CS

	High contention (10^2 cycles)				Low contention (2.10^6 cycles)			
	Exec. time	**# cycles**	**Locality**	**# misses**	**Exec. time**	**# cycles**	**Locality**	**# misses**
Spinlock	Bad	5,423,513	Average	34.9	Average	1,550	Very good	0.2
POSIX	Average bad	159,573	Average	39.3	Average	2,110	Average	8.3
MCS	Average bad	140,557	Very bad	1063.6	Average	1,857	Average	4.0
MCS-TP	Average bad	253,772	Very bad	967.6	Average	2,864	Average	3.8
Flat Combining	Average good	74,063	Bad	473.9	Bad	8,251	Bad	72.9
CC-Synch	Average good	60,920	Good	9.6	Average	4,201	Average	7.3
DSM-Synch	Average good	68,376	Good	11.3	Average	3,788	Average	8.6
RCL	*Good*	*37,516*	*Very good*	*4.3c + 0.0s*	*Average*	*4,339*	*Average*	*5.4c + 0.4s*

(c) Comparison of the lock algorithms for five shared cache lines

Figure 5.3: Microbenchmark results on Niagara2-128

of a critical section, and goes back to executing its client code. This effect is visible on the bottom part of Figure 5.2a and Figure 5.2b: at low contention, while the number of cache misses is higher for most lock algorithms when five cache lines are accessed instead of one, it remains almost constant with RCL. Finally, Figure 5.2c shows that most cache misses incurred by RCL are on the client side, and not on the server side: they are therefore outside the critical path of the server and do not slow down the execution of critical sections for all clients.

Niagara2-128. Figure 5.3 shows the microbenchmark results on Niagara2-128. Instead of L2 cache misses, L1 cache misses are measured on Niagara2-128, because, as explained in Section 4.2.1, they are a better metric for measuring inter-core communication on that machine. Also note that since Niagara2-128 provides $h_{n128} = 128$ hardware threads, $h_{n128} - 1 = 127$ threads execute critical sections are used instead of $h_{48} - 1 = 47$ on Niagara2-128. Therefore, results of the two experiments are not directly comparable.

The microbenchmark results are qualitatively similar on Niagara2-128 and Magnycours-48. However, the performance gap is lower due to the fact that Niagara2-128 has better communication performance relative to its sequential performance, as explained in Section 2.5.3: on Niagara2-128, synchronization is less of a bottleneck, therefore, using efficient locks does not improve performance as much as on Magnycours-48. Under high contention, CC-Synch and DSM-Synch are still 62% and 82% slower than RCL, respectively. Flat Combining is 22% slower than CC-Synch and 8% slower than DSM-Synch. Removing the global lock is not as efficient to improve performance as it

is on Magnycours-48, because, as seen in Section 2.5.3.2, the compare-and-swap instruction scales better on Niagara2-128 than on Magnycours-48. MCS is between 55% and 132% slower than combining locks, and the basic spinlock performs even worse than on Magnycours-48, because more hardware threads are used, which increases contention.

5.3 Applications

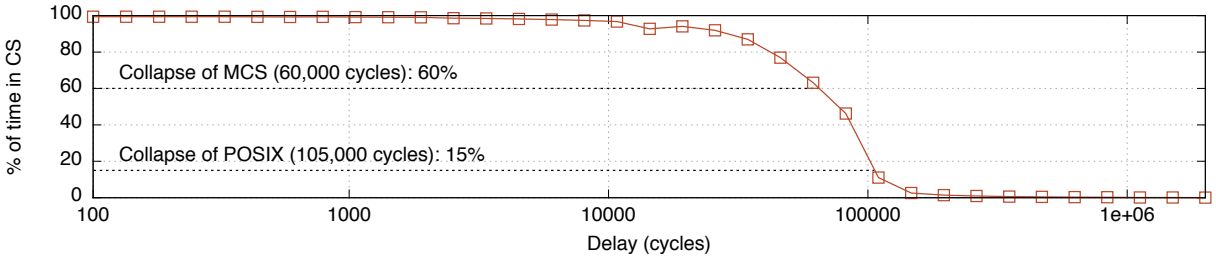
This section focuses on the performance of RCL in multithreaded applications. Section 5.3.1 presents a custom profiler as well as a methodology designed to help developers decide whether RCL may be beneficial for their application. The profiler is then run on applications from the SPLASH-2 and Phoenix 2 benchmark suites, as well as Memcached and Berkeley DB with a TPC-C client. Section 5.3.2 compares the performance of RCL and other lock algorithms on the subset of applications and datasets that were selected thanks to the profiler data. Finally, Sections 5.3.3, 5.3.4, and 5.3.5 give more detailed results for the experiments from SPLASH-2/Phoenix 2, Memcached, and Berkeley DB, respectively.

5.3.1 Profiling

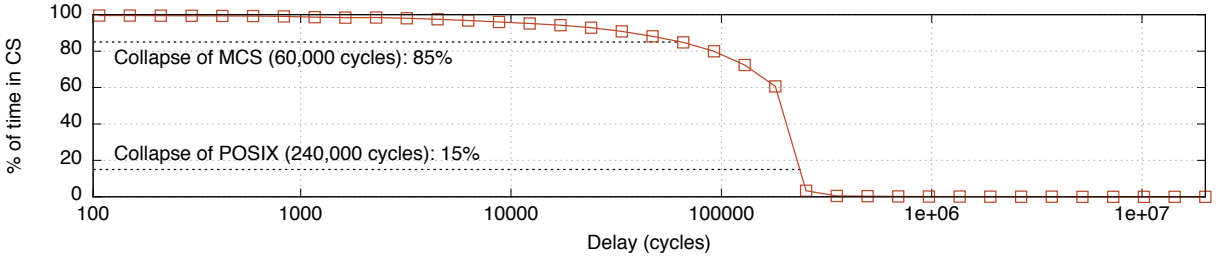
The microbenchmark makes it possible to determine at which level of contention lock algorithms collapse, but the metric it uses for measuring contention is the delay between the execution of critical sections: this metric is not a very good measure of contention in real-world applications because it is also highly dependent on other factors such as the length of critical sections. Instead, the profiler presented in Section 4.2.1 measures the percentage of execution time time spent in critical sections, including lock acquisition and release time. This metric was found to be simple to measure, and yet it is a good estimate of lock contention, because when lock contention is high enough that locks are a bottleneck, lock acquisition and release operations become very long and end up taking up most of the execution time. In order to estimate at which point locks collapse in terms of this metric instead of the delay, the microbenchmark is run through the profiler.

Figures 5.4a and 5.4b shows the result of applying the profiler to the microbenchmark with POSIX locks and one cache line access: the percentage of time spent in critical sections is plotted as a function of the delay.[3] As seen on Figure 5.2a, on Magnycours-48, the POSIX lock collapses when the delay is under $105,000$ cycles, and RCL is better than all other lock algorithms when the delay is under $60,000$ cycles. As shown on Figure 5.4a, running the microbenchmark through the profiler makes it possible to deduce that these two values correspond to 15% and 60% of the execution time spent in critical sections, respectively. Therefore, it can be deduced that the POSIX lock collapses when the microbenchmark spends 15% or more of its time in critical sections (lower threshold), and RCL performs better than all other lock algorithms when the microbenchmark spends 60% ore more of its time in critical sections (upper threshold). These results are preserved, or improved, as the number of accessed cache lines increases, because the execution time increases at least as much, and usually more, for other algorithms than for RCL, due to RCL's improved data locality. Therefore, two thresholds for benchmarked applications can be identified on Magnycours-48: assuming that they use POSIX locks, if they spend more than 15% in critical sections, using RCL may be beneficial, but not necessarily more than using other lock algorithms other than POSIX (lower threshold). If they spend more than 60% of their time in critical sections, using RCL may be more beneficial than using any of the other lock algorithms (upper threshold).

[3]The analysis assumes that the targeted applications use POSIX locks because they are the most common type of lock used in applications on *NIX systems. However, a similar analysis could be made for any type of lock.



(a) Time spent in critical sections on Magnycours-48



(b) Time spent in critical sections on Niagara2-128

			# cycles	% in CS
All hardware threads (48/128)	**Magnycours-48**	**Upper threshold**	60,000	60%
		Lower threshold	105,000	15%
	Niagara2-128	**Upper threshold**	60,000	85%
		Lower threshold	240,000	15%
Half (22 / 62), for Memcached	**Magnycours-48**	**Upper threshold**	25,000	55%
		Lower threshold	30,000	45%
	Niagara2-128	**Upper threshold**	8,000	75%
		Lower threshold	40,000	45%

(c) Thresholds

Figure 5.4: Time spent in critical sections in the microbenchmark and thresholds

A similar analysis on Niagara2-128 shows that on that machine, the benchmarked applications may benefit from RCL if they spend more than 15% in critical sections (lower threshold), and that they may benefit from RCL more than from any other lock algorithm if they spend more than 85% of their time in critical sections (upper threshold).

The thresholds mentioned heretofore have been found by running the microbenchmark on all hardware threads. However, as seen later in this section, Memcached is run with half the hardware thread dedicated to a client that sends request to the Memcached instance. Therefore, the same analysis was performed with half the hardware threads in order to find suitable thresholds for this application. All thresholds are listed in Figure 5.4c.

Applications. Figures 5.5 and 5.6 show the results of the profiler for 18 applications on Magnycours-48 and Niagara2-128, respectively. The evaluated applications are the following:

- The nine applications of the SPLASH-2 benchmark suite (parallel scientific applications, see Section 2.5.3.3). Since Raytrace is provided with two datasets, Balls4 and Car, the results for both are shown (a third data set, Teapot, is also provided but only for debugging purposes).

		% in CS = f(# hardware threads)						Lock info for max. # hardware threads			
		1	4	8	16	32	48	Description	#	#L2 cache misses / CS	% in CS
SPLASH-2	Radiosity	3.7%	6.5%	10.4%	38.4%	76.6%	89.2%	Linked list access	1	1.7	87.7%
	Raytrace/Balls4	0.5%	1.0%	1.6%	3.0%	22.6%	66.8%	Counter increment	1	1.4	65.7%
	Raytrace/Car	9.2%	19.6%	45.8%	70.1%	86.5%	91.7%	Counter increment	1	0.6	90.2%
	Barnes						10.7%				
	FMM						1.0%				
	Ocean Contiguous					0.1%†					
	Ocean Non-Cont.					0.1%†					
	Volrend						14.0%				
	Water-nsquared						6.8%				
	Water-spatial						3.3%				
Phoenix 2	Linear Regression	0.9%	26.5%	39.4%	68.7%	60.1%	81.6%	Task queue access	1	3.8	81.6%
	String Match	0.2%	5.7%	10.2%	23.0%	37.6%	63.9%	Task queue access	1	4.5	63.9%
	Matrix Multiply	0.9%	27.5%	41.8%	67.3%	79.7%	92.2%	Task queue access	1	3.1	92.2%
	Histogram						13.8%				
	PCA						12.2%				
	KMeans						1.1%				
	Word Count						4.1%				
Memcached	Get	3.2%	20.3%	40.7%	69.7%	22 hw. t.: 79.0%‡		Hashtable access	1	2.1	78.3%
	Set	3.7%	19.3%	28.7%	39.1%	22 hw. t.: 44.7%‡		Hashtable access	1	16.5	44.7%
Berkeley DB + TPC-C	Payment						10.1%				
	New Order						5.2%				
	Order Status	2.1%	2.5%	2.1%	2.3%	1.1%	41.2%	DB struct. access	11	2.4	40.1%
	Delivery						4.3%				
	Stock Level	2.1%	2.2%	2.4%	0.5%	0.4%	46.3%	DB struct. access	11	2.4	46.3%

† Number of hardware threads must be a power of 2. ‡ Other hardware threads are executing clients.

Figure 5.5: Profiling results for the evaluated applications on Magnycours-48

- The seven applications of the Phoenix 2.0.0 benchmark suite [101, 103, 112, 92] developed at Stanford. These applications implement common uses of Google's MapReduce [31] programming model in the C programming language. Phoenix 2 provides a small, a medium and a large dataset for each application. The medium datasets were used for all applications.

- Memcached 1.4.6 [26, 41], a general-purpose distributed memory caching system that is used by websites such as YouTube, Wikipedia and Reddit. Memcached is run on half of the hardware threads of each machine, because the other half of the hardware threads is dedicated to the client that generates the load. This can be done without impacting performance because Memcached's scalability is too low for it to benefit from more than half the hardware threads of Magnycours-48 or Niagara2-128. As shown in Section 5.3.4, this is true even when Memcached is modified to use other lock algorithms, including RCL.

  Memcached uses two threads that perform other tasks than executing requests: one of them dispatches incoming packets to the worker threads, and another one is responsible for performing maintenance operations on the global hashtable that stores cached data, such as resizing it when needed. Therefore, for Memcached to run n worker threads, $n + 2$ hardware threads are needed: this explains why the maximum number of hardware threads used in the experiment is $h_{m48}/2 - 2 = 22$ (resp. $h_{n128}/2 - 2 = 62$) for Magnycours-48 (resp. Niagara2-128) instead of $h_{m48}/2 = 24$ (resp. $h_{n128}/2 = 64$). The client used to generate the load is Memslap, from Libmemcached 1.0.2 [28]. Two experiments are used, in the first one, the client only executes Get requests (reads from the cache), and in the second one, it only executes Set requests (writes to the cache).

- Berkeley DB 5.2.28 [80, 79], a database engine maintained by Oracle, with TpccOverBkDB, a TPC-C [67] benchmark written by Alexandra Fedorova and Justin Fuston at Simon Fraser University. Five experiments are used, one for each of the request types offered by TPC-C. Berkeley DB takes the form of a library, and the client is an application that creates one thread that uses Berkeley DB routines for each simulated client.

The left part of tables in Figures 5.5 and 5.6 shows the time spent in critical sections for the selected applications, for different numbers of threads. The time spent in critical sections increases when the number of threads increases, because increasing the number of threads increases contention. A gray box in the tables indicates that the application is not profiled for the corresponding number of threads, because even when using one software thread per hardware thread, the time spent in critical sections is very low (under the lower threshold): therefore, locks are not a bottleneck. The right part of the tables shows the time spent in critical sections and the number of cache misses for the most used locks.

In SPLASH-2/Radiosity, the most used lock is used to concurrently access a linked list. In SPLASH-2/Raytrace, the most used lock protects a shared counter. In the Phoenix benchmarks, the most used lock protect the task queue from the MapReduce implementation. In Memcached, the most used lock protects the shared hashtable that stores all of the cached data. For the contended SPLASH-2 and Phoenix 2 applications, as well as Memcached, the time spent in critical sections for the most used lock is extremely close to the global time spent in critical sections: this shows that their lock bottleneck comes from a single lock. Most benchmarks have a low number of cache misses, except Memcached/Set (and, to some extent, Memcached/Get on Niagara2-128, but 13.5 is not a very high number for L1 cache misses). For Berkeley DB, the profiler identifies a block of eleven locks that are highly contended, because these locks are all allocated at the same code location (same file and line number) in the Berkeley DB library, and the profiler identifies lock by their allocation site. These locks protect the access to a structure that is unique for each database, and eleven databases are used in TPC-C. The time spent in critical sections by these eleven locks is equal to the global time spent in critical sections, which shows that no other lock in the application is a bottleneck. The number of cache misses is always low in the experiments with Berkeley DB.

5.3.2 Performance overview

The two metrics provided by the profiler, i.e., the time spent in critical sections and the number of cache misses, do not, of course, completely determine whether an application will benefit from RCL. Many other factors (length of critical sections, interactions between locks, etc.) affect the execution of critical sections. However, as shown in this chapter, using the time spent in critical sections as the main metric and the number of cache misses in critical sections as a secondary metric works well: the former is a good indicator of contention, and the latter of data locality.

In order to evaluate the performance of RCL, the performance of applications listed in Figures 5.5 and 5.6 is measured with the lock algorithms (including RCL) listed in Figures 5.2c and 5.3c. The following paragraphs list the applications that were selected for the evaluation based on the results of the profiler.

SPLASH-2 and Phoenix. For the SPLASH-2 and Phoenix benchmark suites, results for the experiments whose time spent in critical sections is higher than one of the thresholds found in Section 5.2 are presented. As explained in Section 5.3.1, since the time spent in critical sections grows with lock contention, when one of the thresholds is reached, using RCL may be beneficial

		% in CS = f(# hardware threads)							Lock info for max. # hardware threads			
		1	4	8	16	32	64	128	Description	#	# L1 cache misses / CS	% in CS
SPLASH-2	Radiosity	1.6%	2.8%	2.8%	3.0%	3.1%	3.1%	38.7%	Linked list access	1	5.4	35.5%
	Raytrace/Balls4	0.3%	0.5%	0.4%	0.5%	0.5%	0.6%	14.3%	Counter increment	1	4.6	13.4%
	Raytrace/Car	5.4%	5.9%	5.8%	6.0%	9.2%	44.7%	79.1%	Counter increment	1	3.8	77.0%
	Barnes							3.7%				
	FMM							0.5%				
	Ocean Cont.							0.0%				
	Ocean Non-Cont.							0.0%				
	Volrend							1.0%				
	Water-nsquared							7.3%				
	Water-spatial							0.1%				
Phoenix 2	Linear Regression							3.4%				
	String Match							2.9%				
	Matrix Multiply							10.2%				
	Histogram							3.2%				
	PCA							0.1%				
	KMeans							0.0%				
	Word Count							2.0%				
Memcached	Get	5.4%	10.4%	16.5%	42.3%	62.2%	62 hw. t.: 69.9%‡		Hashtable access	1	13.5	69.2%
	Set	5.0%	8.7%	14.0%	17.5%	19.3%	62 hw. t.: 20.4%‡		Hashtable access	1	73.4	20.2%
Berkeley DB + TPC-C	Payment							1.8%				
	New Order							0.3%				
	Order Status	0.2%	0.1%	0.1%	0.1%	43.8%	63.6%	76.4%	DB struct. access	11	4.0	76.4%
	Delivery							0.8%				
	Stock Level	0.2%	0.1%	0.1%	0.1%	55.5%	78.2%	87.1%	DB struct. access	11	3.4	87.1%

‡ Other hardware threads are executing clients.

Figure 5.6: Profiling results for the evaluated applications on Niagara2-128.

thanks to RCL's good performance under high contention. Since all SPLASH-2 and Phoenix applications always exhibit a low number of cache misses in critical sections (less than five) in Figures 5.5 and 5.6, data locality is disregarded as a criterion to select experiments. Following Figures 5.5 and 5.6, on Magnycours-48, the results of String Match (Phoenix 2), Raytrace/Balls4 (SPLASH-2), Linear Regression (Phoenix 2), Radiosity (SPLASH-2), Raytrace/Car (SPLASH-2), and Matrix Multiply (Phoenix 2) are presented, since these experiments all spend more time in critical sections than the upper threshold. On Magnycours-48, no experiment spends less time in critical sections than the upper threshold but more than the lower threshold. On Niagara2-128, none of the experiments from SPLASH-2 or Phoenix 2 spends more time in critical sections than the upper threshold. On that machine, the results of Radiosity (Phoenix 2) and Raytrace/Car (SPLASH-2) are presented, since the time they spend in critical sections is lower than the upper threshold but higher than the lower threshold. Finally, the results of Raytrace/Balls4 are shown even though the time time spent in critical sections for this experiment is below both thresholds: the objective is to illustrate a case where lock contention is low enough that using more efficient lock algorithms, including RCL, should not improve performance.

Memcached. For Memcached, on Magnycours-48, the experiment with Get requests spends more time in critical sections than the upper threshold. Even though the experiment with Set requests spends an amount of time in critical sections that is between the two thresholds, it also exhibits a large number of cache misses in critical sections (16.5 L2 cache misses). Therefore, a significant performance improvement can be expected in that experiment. Similarly, on Niagara2-128, both experiments (with Get and Set requests) spend an amount of time in critical sections that is between the lower and the upper threshold. On that machine, Memcached/Set exhibits a large number of cache misses in critical sections (72.4 L1 cache misses), which indicates that RCL could be beneficial for locality in this experiment. The performance of combining

locks was not evaluated, because Memcached periodically blocks on condition variables, for which combining locks do not provide an implementation, as explained in Section 3.6. However, condition variables are implemented for the basic spinlock and queue locks, using the algorithm described in Section 3.2.

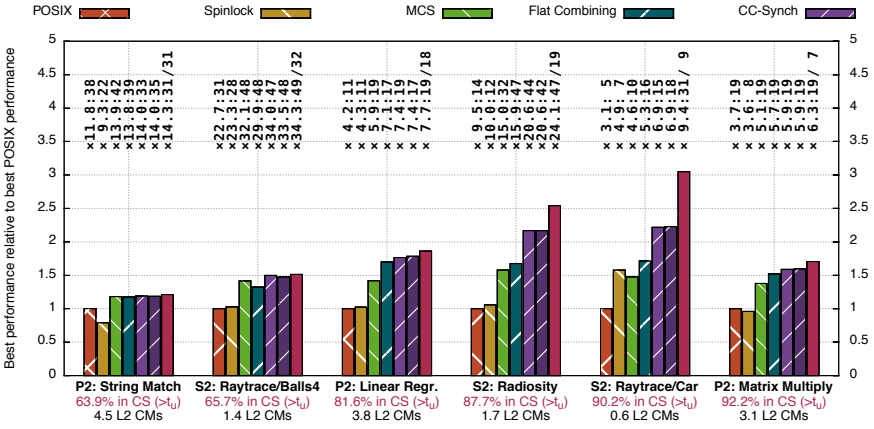
Berkeley DB with TpccOverBkDb. For Berkeley DB with TpccOverBkDb, results for Order Status and Stock Level requests on both machines are presented. On Magnycours-48, the time spent in critical sections for these two experiments, albeit high, is lower than the upper threshold. However, the profiler underestimates the actual time spent in critical sections for Berkeley DB, because Berkeley DB uses hybrid locks that busy-wait for a moment before going to sleep with a POSIX lock: the busy-waiting time is not included in the percentage of time spent in critical sections because the profiler is written for POSIX locks. The performance of Berkeley DB with TpccOverBkDb is not evaluated for the other three types of requests, namely Payment, New Order and Delivery, because the time they spend in critical sections is extremely low (lower than both thresholds). On Niagara2-128, the results of the profiler are similar, except for the fact that on that machine, the experiment with Stock Level requests spends more time in critical sections than the upper threshold.

Figure 5.7 presents an overview of the performance results of all selected experiments for all lock algorithms. As a reminder, some of the profiling data is shown again in this figure, and the amount of time spent in critical sections is compared to lower and upper thresholds (noted t_l and t_u) that are suitable for each experiment. The experiments are run with the parameters shown in Figure 5.7c. Custom parameters were used for Radiosity (SPLASH-2) in order to make the benchmark perform more work: since SPLASH-2 applications were designed in the 90's, some of them execute too fast on modern architectures to give usable results. As explained in Section 5.3.1, Phoenix 2 uses the medium datasets for its benchmarks, which corresponds to 100MB input files for Linear Regression and String Match, and a 500x500 matrix for Matrix Multiply. For Memcached, the hashtable is preloaded with 30,000 entries when Get requests (reads) are used, whereas it is preloaded with only 10,000 requests for Set requests, since in that experiment, the client fills the hashtable. For Berkeley DB with TpccOverBkDb, each client uses a separate thread, and each one of these threads executes 300 requests.

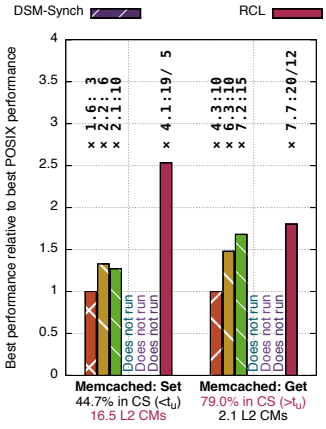
In the applications from Figures 5.7a, 5.7b, 5.7d, and 5.7e, i.e., SPLASH-2 and Phoenix 2 applications as well as Memcached, only one lock is replaced, as indicated in Figures 5.2c and 5.3c. Therefore, for RCL, these applications only use one server hardware thread. In the figures of this section, when an application is run with n hardware threads, it either means that (i) $n - s$ application threads and s servers are run when RCL is used, or (ii) n application threads are used for other applications. All software threads are bound to hardware threads in all applications, except for Berkeley DB, since in that case, when more software threads can be used than the number of hardware threads, dynamic scheduling can make better usage of the hardware threads than static scheduling.

The numbers above the histograms ($\times\alpha : \eta/\mu$) report the improvement α over the execution time of the original application on one hardware thread, the number η of hardware threads that gives the shortest execution time (i.e., the scalability peak), and the minimal number μ of hardware threads for which RCL is faster than all other lock algorithms. The histograms show the ratio of the execution time with each of the lock algorithms relative to the execution time with POSIX locks.

The following sections (5.3.3, 5.3.4 and 5.3.5) describe the performance results of all experiments, and they describe Figure 5.7 in detail. However, general trends can already be observed on this figure: RCL is generally faster than other lock algorithms, and when the percentage



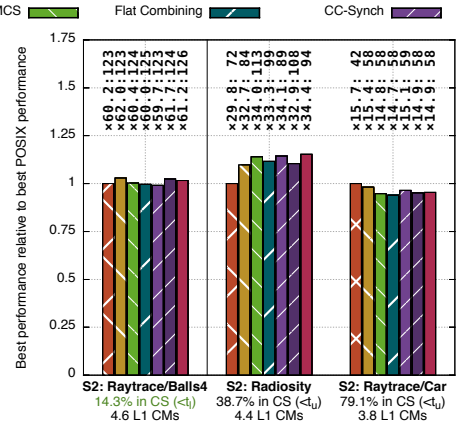
(a) Magnycours-48: SPLASH-2 and Phoenix 2



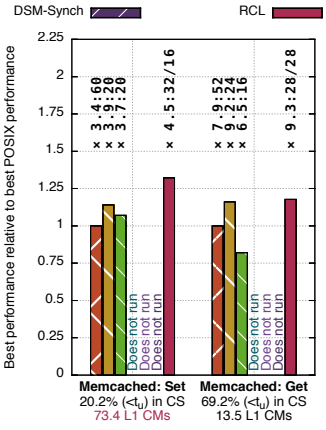
(b) Magnycours-48: Memcached

SPLASH-2	Radiosity	-batch -ae 500 -bf 0.005
	Raytrace/Car	Input files: car.env/geo
	Raytrace/Balls4	Input files: balls4.env/geo
Phoenix 2	Linear Regr.	Medium dataset: 100MB input file
	String Match	Medium dataset: 100MB input file
	Matrix Multiply	Medium dataset: 500x500 matrix
Memcached	Get	30,000 entries in base 30,000 Get requests
	Set	10,000 entries initially 10,000 Set requests
Berkeley DB + TPC-C	Order Status	300 requests per client
	Stock Level	300 requests per client

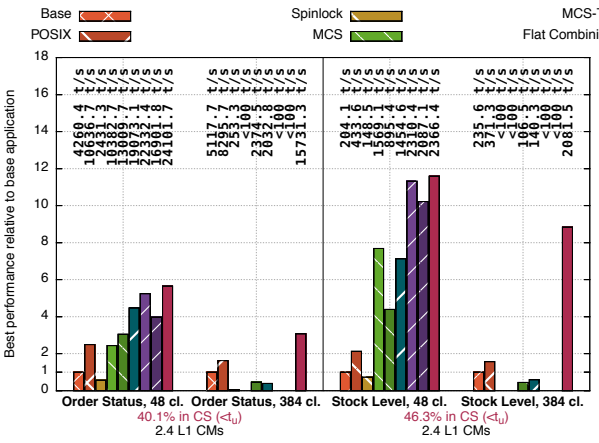
(c) Benchmark parameters



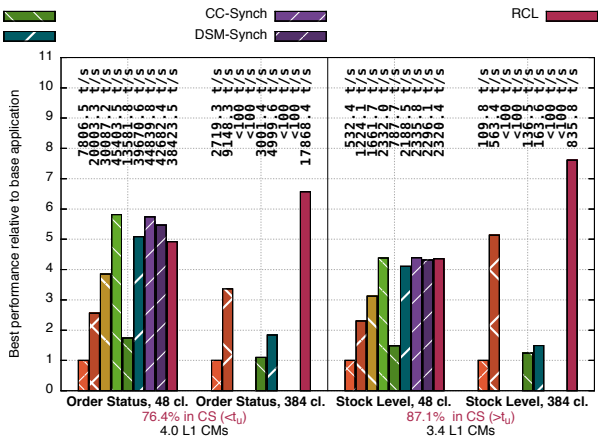
(d) Niagara2-128: SPLASH-2



(e) Niagara2-128: Memcached



(f) Magnycours-48: Berkeley DB



(g) Niagara2-128: Berkeley DB

Note: in this figure, t_l et t_u stand for the lower and the upper threshold corresponding to each experiment.

Figure 5.7: Application performance overview

of time spent in critical sections increases or when the number of cache misses increases, the performance improvement offered by RCL also increases. On Magnycours-48, the percentage of time spent in critical sections is generally higher than on Niagara2-128 for applications that use POSIX locks (i.e., all applications other than Berkeley DB). Again, this is due to the fact that on

Niagara2-128, as was explained in Section 2.5.3, the cost of communication is lower relative to its sequential performance. Therefore, synchronization is less of a bottleneck. On Berkeley DB with TpccOverBkDb, the percentage of time spent in critical sections is higher for Niagara2-128, but since Berkeley DB uses non-POSIX locks, measurements of this metric by the profiler are not reliable: the performance gains offered by RCL on Niagara2 for Berkeley DB are lower than on Magnycours-48, which seems to indicate that Berkeley DB with TpccOverBkDb actually suffers from more lock contention on Niagara2-128 than on Magnycours-48.

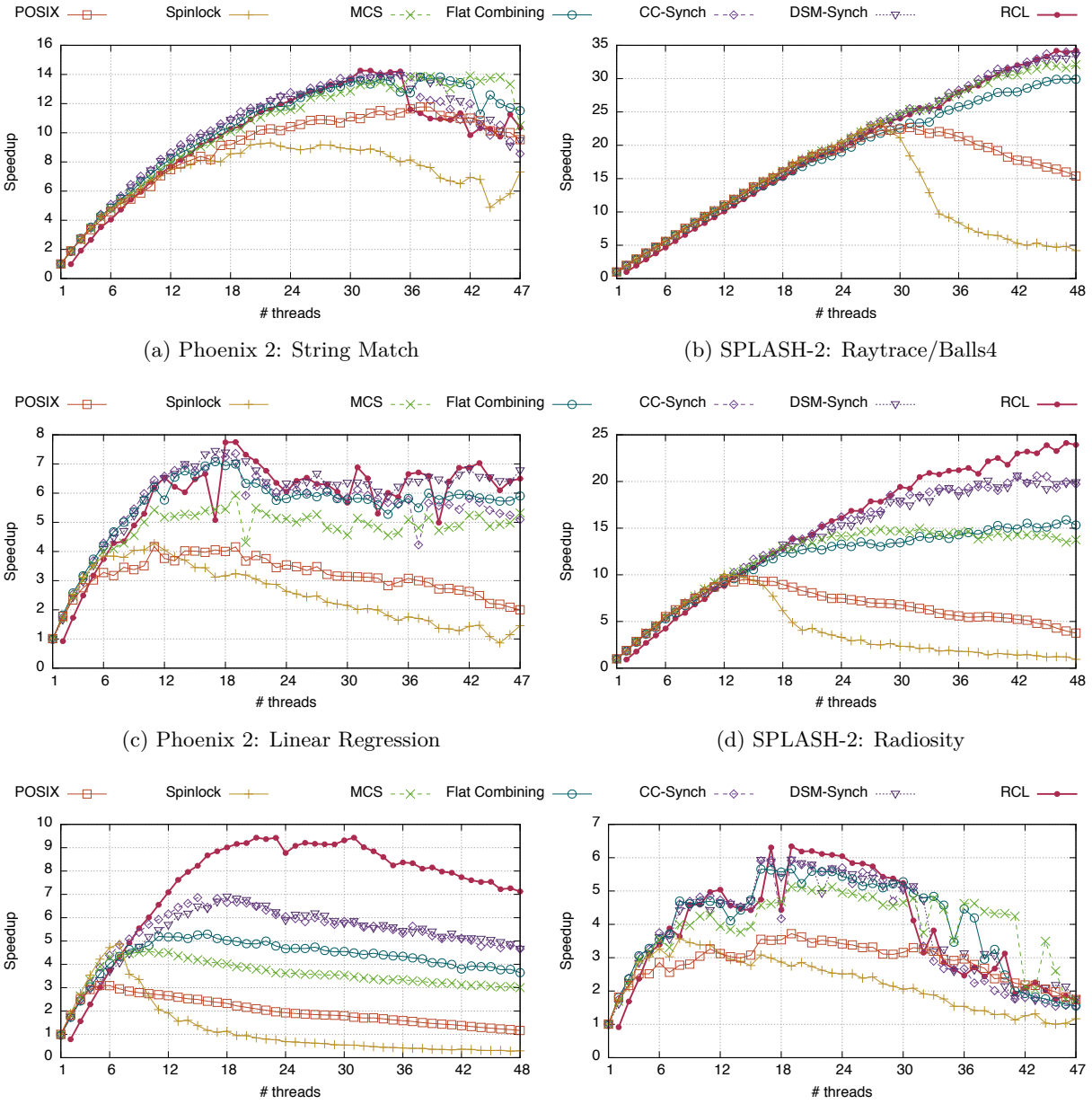
5.3.3 Performance of SPLASH-2 and Phoenix applications

This section focuses on the performance of the SPLASH-2 and Phoenix applications that were selected by the profiler. The first paragraphs focus on the six selected experiments on Magnycours-48, and the remaining paragraphs focus on the three selected experiments on Niagara2-128.

Magnycours-48. As shown on Figure 5.7a, on Magnycours-48, where all of the selected experiments from SPLASH-2 and Phoenix 2 spend more time in critical sections than the upper threshold, the performance gain for better lock algorithms, RCL in particular, increases with the time spent in critical sections: this shows that the percentage of time spent in critical sections is a good metric for selecting benchmarks. However, even though Matrix Multiply (Phoenix 2) spends 92.2% of its time in critical sections when using POSIX locks, its performance improvement with RCL is similar to Linear Regression (Phoenix 2) which only spends 81.6% of its time in critical sections. This comes from the fact that Matrix Multiply is intrinsically unable to scale for the considered data set: even though the use of RCL reduces the amount of time spent in critical sections to 1%, the best resulting speedup is only 6.3 times for 7 hardware threads.

On average, the basic spinlock performs similarly to the POSIX lock. MCS and Flat Combining improve performance significantly, with Flat Combining being slightly more efficient than MCS most of the time. CC-Synch and DSM-Synch consistently perform better than POSIX, the basic spinlock, MCS and Flat Combining, and they both provide similar results as expected on a cache-coherent machine. RCL performs significantly better than all other lock algorithms, and the performance gain increases as the time in critical section increases except for Matrix Multiply. The performance of lock algorithms in these applications is consistent with the results from the microbenchmark, except for the basic spinlock which sometimes performs better than the POSIX lock even though it always performed much worse in the microbenchmark.

Figure 5.8 shows detailed results for the selected SPLASH-2 and Phoenix 2 experiments on Magnycours-48: for each lock algorithm on each benchmark, the speedup relative to the single-threaded version is plotted as a function of the number of threads used. Since RCL loses one hardware thread for the server, its performance is worse when only a few application threads are used, but it catches up with other lock algorithms quickly (after 3 to 10 threads). The more time experiments spend in critical sections, the earlier the unmodified version of the application with POSIX locks collapes: String Match (Phoenix 2) spends 63.9% of its execution time in critical sections and starts collapsing at 39 hardware threads, Raytrace/Balls4 (SPLASH-2) spends 65.7% of its execution time in critical sections and starts collapsing at 32 hardware threads, Linear Regression (Phoenix 2) spends 81.6% of its time in critical sections and starts collapsing at 20 hardware threads, Radiosity (SPLASH-2) spends 87.7% of its time in critical sections and starts collapsing at 15 hardware threads, and Raytrace/Car spends 90.2% of its time in critical sections and starts collapsing at 8 hardware threads. An early collapse indicates high contention, which shows that the profiler efficiently identifies highly-contended locks. The higher the contention, the more using more efficient lock algorithms, RCL in particular, improves



(a) Phoenix 2: String Match

(b) SPLASH-2: Raytrace/Balls4

(c) Phoenix 2: Linear Regression

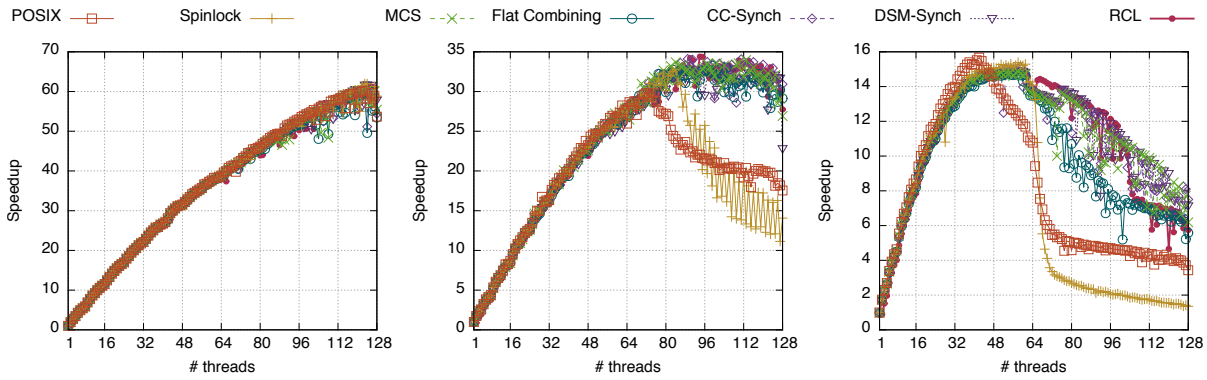
(d) SPLASH-2: Radiosity

(e) SPLASH-2: Raytrace/Car

(f) Phoenix 2: Matrix Multiply

Figure 5.8: SPLASH-2 and Phoenix 2 speedup on Magnycours-48

performance (better speedup) and scalability (later collapse). Again, the only outlier is Matrix Multiply (Phoenix 2) which spends 92.2% of its time in critical sections and yet starts collapsing for all lock algorithms at around 20 hardware threads, which seems to indicate that it suffers from another bottleneck which prevents all lock algorithms from improving performance beyond that point. Even though the basic spinlock usually collapses before the POSIX lock, its performance peak is sometimes higher than the POSIX lock's. This shows that the basic spinlock can exhibit good performance when the number of hardware threads is low because it has not saturated the bus yet, which explains the performance gap of the basic spinlock with the microbenchmark: in



(a) Raytrace/Balls4 (b) Radiosity (c) Raytrace/Car

Figure 5.9: SPLASH-2 speedup on Niagara2-128

Figure 5.2, the basic spinlock always performs poorly because the maximum number of hardware threads is always used.

Niagara2-128. As seen in Figure 5.7d, on Niagara2-128, the performance of Raytrace/Balls4 (SPLASH-2) is not improved by using more efficient lock algorithms, and Figure 5.9a shows that the performance of Raytrace/Balls4 never collapses. This is the expected behavior, because Raytrace/Balls4 spends less time in critical sections than both thresholds: the profiler correctly estimates that the experiment does not suffer from a level of lock contention that is high enough for performance to be improved by using better lock algorithms. The fact that replacing POSIX locks by other lock algorithms does not worsen performance indicates that even if a developer mistakenly replaces a POSIX lock by a more efficient one, such as RCL, due to a false positive from the profiler, no negative consequences are to be expected when it comes to performance. Moreover, the profiler does not return any false negatives on the eighteen experiments that were profiled: although these results are not shown in this thesis, the performance of other experiments whose time spent in critical sections was below both thresholds was evaluated with all lock algorithms, and the result was always the same: changing lock algorithms does not alter performance in that case.

Note that even though the performance of Raytrace/Balls4 on Figure 5.9a never collapses, the slope of the curve decreases as the number of threads increases. This may indicate that the application simply does not reach the point where adding more threads increases lock contention enough that its performance collapses and using other lock algorithms improves performance: on similar yet newer architectures with more hardware threads, one might expect that the application would reach that point and that using more efficient lock algorithms would improve performance.

Radiosity (SPLASH-2) and Raytrace/Car (SPLASH-2) spend an amount of time in critical sections that is between the two thresholds, therefore, using other lock algorithms than POSIX should improve performance, but RCL may not improve performance more than the other lock algorithms. That can be seen for Radiosity in Figure 5.9b: any other lock algorithm than POSIX improves performance, but MCS, Flat Combining, CC-Synch, DSM-Synch and RCL all give similar results. With Raytrace/Car, however, the POSIX lock is more efficient than all other lock algorithms even though according to the profiler, performance should be improved by other lock algorithms. As seen in Figure 5.9c, in that experiment, even though the POSIX lock collapses first, other lock algorithms do not perform better because the application always collapses when it reaches a speedup of 16x, which seems to indicate that another bottleneck than locks prevents the application from scaling beyond that speedup.

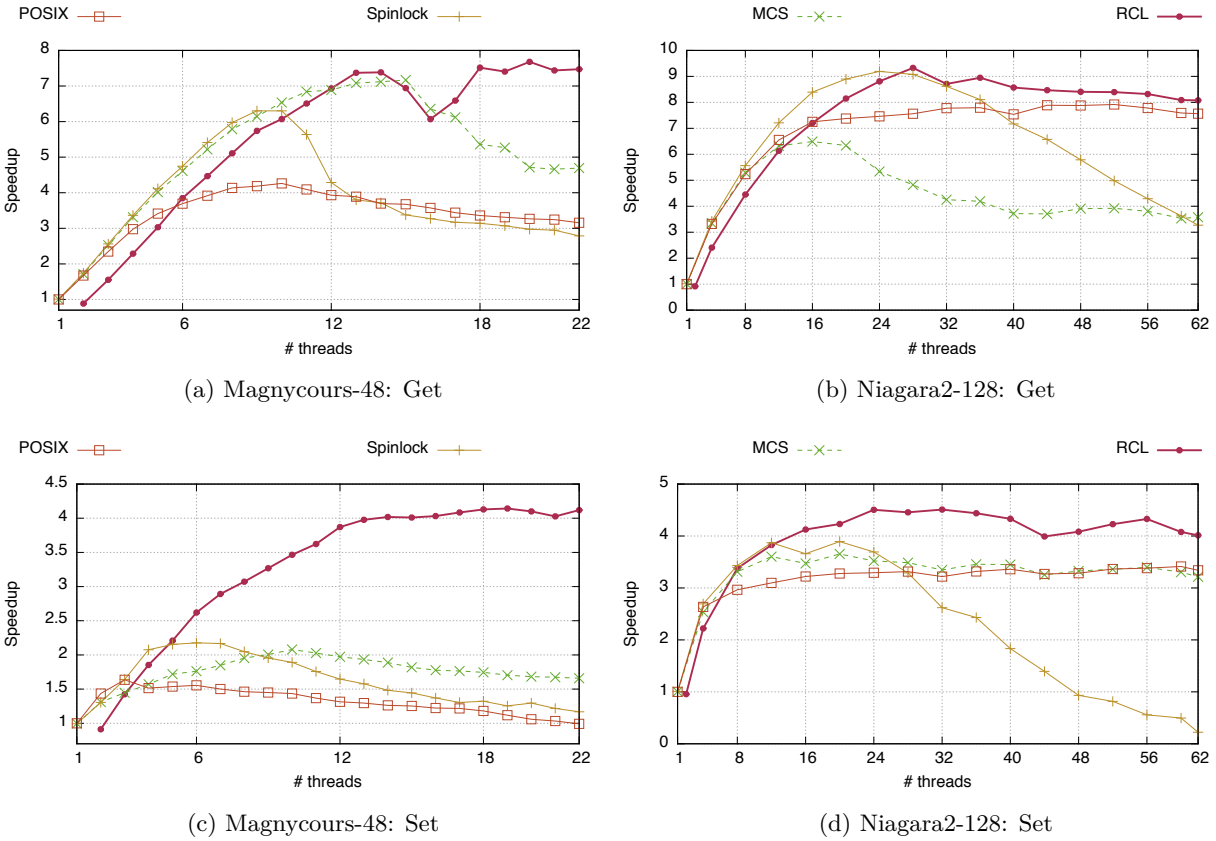
5.3.4 Performance of Memcached

This section focuses on the performance of Memcached, with Get and Set requests. As mentioned earlier in section 5.3, with Memcached, the performance of RCL is only compared to that of the POSIX lock, the basic spinlock, and MCS, because Memcached uses condition variables and combining locks are not designed to use them. The POSIX library provides primitives to handle condition variables with the POSIX lock, and using these primitives to provide support for condition variables for the basic spinlock and MCS is straightforward. Figure 5.10 shows, for all locks, the speedup of multithreaded Memcached relative to the single threaded-version, as a function of the number of worker threads used. Figures 5.10a and 5.10c show these results on Magnycours-48 for Get and Set requests, respectively. Similarly, Figures 5.10d and 5.10d show these results on Niagara2-128 for Get and Set requests.

Magnycours-48. As seen in Figure 5.5, on Magnycours-48, Memcached/Get spends 79% of its time in critical sections and improves performance by 1.80 times, which is consistent with the performance improvements observed on SPLASH-2 and Phoenix in Figure 5.7b: Memcached/Get spends more time in critical sections than Raytrace/Balls4 but less than Linear Regression, and in these two experiments, RCL improves performance by 1.51 times and 1.86 times, respectively. For Memcached/Set, RCL drastically improves performance, by 2.53 times, even though it only spends 44.7% of its time in critical sections: this discrepancy is caused by the fact that in that case, RCL does not only perform better because lock contention is high, it also increases locality.

Figure 5.11a shows the average number of cache misses inside critical sections (not counting synchronization cache misses) in the original application and when RCL is used. All applications other than Berkeley DB with TpccOverBkDB are run with their maximum number of threads (h threads for SPLASH-2 and Phoenix 2, $h/2 - 2$ threads for Memcached), and for Berkeley DB with TpccOverBkDB, one application thread per hardware thread is used. With RCL, the number of cache misses is given per server for applications that use several RCL servers. The base Memcached/Set application triggers a lot of cache misses (16.5) which slow down the execution of the critical path, and RCL is able to improve locality, roughly dividing the number of L2 cache misses by 3: after transformation, critical sections only trigger 5.7 cache misses on the RCL server. The contended critical sections in Memcached/Set protect writes to the hashtable. With RCL, large parts of the hashtable remain stored in the cache hierarchy of the server hardware thread, hence why the number of cache misses drops and performance is vastly improved. However, as seen in Figure 5.11, RCL does not always decrease the number of L2 cache misses significantly in applications whose critical sections only trigger a few (< 5) cache misses because even though it improves locality, RCL also adds cache misses for accessing context variables: a significant amount of cache misses in critical sections seems to be needed for RCL to ensure better locality.

As seen in Figures 5.10a and and 5.10c, in both cases, RCL does not only improve performance, it also improves scalability. In Memcached/Get, both the POSIX lock and the basic spinlock start collapsing at around 11 hardware threads, MCS starts collapsing at around 16 hardware threads, while RCL reaches a plateau from 18 hardware threads onwards. RCL is initially slower than other lock algorithms due to the fact that it loses one hardware thread for the server, but it reaches the performance of the POSIX lock, the basic spinlock and MCS at 6, 11 and 12 hardware threads respectively. In Memcached/Set, the POSIX lock, the basic spinlock and MCS start collapsing at 4, 8 and 11 hardware threads respectively while RCL reaches a plateau at around 14 hardware threads. RCL beats the performance of all other lock algorithms with only 5 hardware threads.



(a) Magnycours-48: Get

(b) Niagara2-128: Get

(c) Magnycours-48: Set

(d) Niagara2-128: Set

Figure 5.10: Memcached speedup

Niagara2-128. As seen in Figure 5.6, on Niagara2-128, Memcached/Get spends an amount of time in critical sections that is between the lower and the upper threshold, which means that using RCL should improve performance, but not necessarily more than other lock algorithms. This is indeed what can be observed in Figure 5.7e: both the basic spinlock and RCL improve performance, with a small advantage for RCL which might not be significant. However, as seen on Figure 5.10b, even though RCL and the basic spinlock have similar peak performance, RCL performs better when a large number of hardware threads is used: at 62 hardware threads, the speedup of the application collapses to 3.3 times when the basic spinlock is used, whereas using RCL makes it possible to maintain a speedup of 8.1 times.

Similarly to what was observed on Magnycours-48, for Memcached/Set, RCL improves performance more than it should (1.3 times, better than other lock algorithms) given the relatively low amount of time it spends in critical sections (20.2%). Again this is due to the large number of cache misses per critical section for that experiment: 73.4 L1 cache misses.[4] This number is more than halved when RCL is used: as seen on Figure 5.11b, with RCL, critical sections only trigger 32.3 L1 cache misses with RCL because large parts of the hashtable remain in the server's cache hierarchy. Again, detailed results (see Figure 5.10d) show that using RCL improves scalability as well as performance. As a side note, Figure 5.11b also shows that on

[4]This number might seem much higher than the number of cache misses for Memcached/Set on Magnycours-48, but keep in mind that on Niagara2-128, L2 cache misses are measured instead of L1 cache misses. L1 cache misses are triggered much more often than L2 cache misses because L1 caches are very small, and on Niagara2-128, 8 hardware threads share each L1 cache, which leads to frequent cache line evictions.

		L2 CMs in base app.	# RCL server L2 CMs / CS	
			S1	S2
SPLASH-2	Radiosity	1.7		1.5
	Raytrace / Car	0.6		1.0
	Raytrace / Balls4	1.4		1.0
Phoenix 2	Linear Regression	3.8		2.1
	String Match	4.5		2.2
	Matrix Multiply	3.1		2.2
Memcached	Get	2.1		3.0
	Set	*16.5*		*5.7*
Berkeley DB + TPC-C	Order Status	2.4	2.7	2.8
	Stock Level	2.4	2.6	2.7

(a) L2 cache misses on Magnycours-48

		L1 CMs in base app.	# RCL server L1 CMs / CS		
			S1	S2	S3
SPLASH-2	Radiosity	5.4			2.0
	Raytrace / Car	4.6			4.1
	Raytrace / Balls4	3.8			3.0
Memcached	Get	13.5			10.7
	Set	*73.4*			*32.3*
Berkeley DB + TPC-C	Order Status	4.0	2.1	2.8	3.1
	Stock Level	3.4		4.9	0.5

(b) L1 cache misses on Niagara2-128

Figure 5.11: Number of cache misses per critical section on the RCL server

Niagara2-128, locality is improved for all applications, even when the number of cache misses per critical section is low.

5.3.5 Performance of Berkeley DB with TpccOverBkDb

As can be seen in Figures 5.5 and 5.6, using Berkeley DB with TpccOverBkDb is more complex than using it with other applications because multiple locks are contended. Therefore, a preliminary analysis is needed to decide how many servers to use and which server should handle which locks. In Section 5.3.5.1, this analysis is performed in order to find an optimal configuration for RCL. In Section 5.3.5.2, the benchmarks are run with this configuration and the results are discussed. Since Berkeley DB with TpccOverBkDb is a benchmark that can use more client threads there are than hardware threads on the machine, repeatedly yielding the processor in busy-wait loops can be beneficial: in Section 5.3.5.3, this optimization is implemented for most of the lock algorithms, including RCL. The influence of this change on the performance of Berkeley DB with TpccOverBkDb is then discussed.

5.3.5.1 Experimental setup

A difficulty in transforming Berkeley DB for use with RCL is that the function call in the source code that allocates the most used locks allocates eleven locks in total, and not all of them suffer from high contention. The RCL runtime requires that for a given lock allocation site, all allocated locks be implemented in the same way, and thus all eleven locks must be implemented as RCLs. Using a single server to handle all locks would cause critical sections to be needlessly serialized, and using eleven servers would waste computing power that could be used for client threads. To prevent this, as explained in Chapter 4, the RCL runtime makes it possible to choose the server where each lock will be dispatched. However, in order to choose the number of servers used and to decide which locks will be handled by which server, the experiments must be run once in order to gather statistics about the execution, which reveals which locks are the most used. This analysis can either be performed with the profiler or with the RCL runtime itself because the RCL runtime can provide statistics about each RCL server. The second option was chosen in order to illustrate a use of the statistics provided by the RCL runtime.

The RCL runtime is able to provide statistics for each of its lock servers. The metrics it measures are the following:

- **Cache misses.** The RCL server is able to measure the average number of cache misses per critical section, either including synchronization costs (used for Figures 5.2 and 5.3) or not (used for Figure 5.11, since the number of cache misses measured in the non-modified applications are measured inside critical sections and do not include synchronization costs either).
- **Use rate.** The use rate measures the server workload. It is defined as the total number of executed critical sections in iterations where at least one critical section is executed, divided by the number of client threads. Therefore, a use rate of 1.0 means that all elements of the array contain pending critical section requests, whereas a low use rate means that the server spends most of its time waiting for critical sections to execute. The reason why iterations during which no critical sections are executed are ignored is that at the beginning of an experiment, there can be a relatively long sequential initialization time during which no critical sections are executed. This phase should not alter the results.
- **False serialization rate.** The false serialization rate is defined as the average number of different locks that critical sections use during one active iteration of the server loop, divided by the number of client threads. It measures the amount of *false serialization*, i.e., the needless serialization of independent critical sections that happens when one server handles several locks.
- Other statistics are provided by the RCL runtime, such as the *slow path rate* which measures how often the server uses the slow path as defined in Section 4.1.2.2, the number of times the manager thread gets woken up, the number of times the `is_alive` flag was not set (see Algorithm 11) and the total number of critical sections. These statistics can be useful for debugging, but are not used in this section because they are not related to false sharing.

In order to find which locks are the most used in Berkeley DB with TpccOverBkDb, after transformation, the experiment is run once with eleven RCL servers, each server handling one of the locks that were found by the profiler. The experiment is run for both Order Status and Stock Level requests on Magnycours-48 and Niagara2-128. The results are shown in Figure 5.12a. For Order Status, on Magnycours-48, the four most used locks are L10, L3, L4 and L6, in that order. On Niagara2-128, the four most used locks are L4, L3, L10, and L6. For Stock Level, the two most used locks are L10 and L8 on Magnycours-48 and L8 and L10 on Niagara2-128, with all other locks being much less used.[5] Using this information, the five server configurations listed in Figure 5.12b are constructed. The first and the last configurations put all locks on one server or use one lock for each server, respectively. The three other configurations place the two, three and four most used locks of the Order Status experiment on different servers in such a way that the same configurations exhibit this characteristic on both machines. Other locks are distributed so that all servers handle a similar number of locks. Additionally, configuration number 2 places the two most used locks of the Stock Level experiment on two different servers for both machines, with other locks being evenly distributed on both machines. Since, with Stock Level, two locks

[5]While the use rates differ slightly between Magnycours-48 and Niagara2-128, the lists of most used locks are globally the same on both machines: only their order differ. The fact that the profiling results are quantitatively similar on two machines with very different architectures could indicate that this profiling phase may not needed for every new machine the application is run on.

		Use rate										
		L11	L10	L9	L8	L7	L6	L5	L4	L3	L2	L1
Order Status	Magnycours-48	1.4%	3.5%	1.4%	1.4%	1.5%	1.8%	1.6%	2.4%	2.6%	1.6%	1.6%
	Niagara2-128	0.7%	1.7%	0.7%	0.7%	0.7%	1.5%	0.9%	3.2%	2.7%	0.7%	0.7%
Stock Level	Magnycours-48	1.4%	5.5%	1.5%	6.7%	1.5%	1.5%	1.5%	1.5%	1.6%	1.6%	1.6%
	Niagara2-128	0.7%	4.1%	0.7%	5.6%	0.7%	0.7%	0.7%	0.7%	0.7%	0.7%	0.7%

(a) Use rate with one lock per hardware thread

# srv.	Server configuration	Lock locations										
		L11	L10	L9	L8	L7	L6	L5	L4	L3	L2	L1
1	All locks on same server	S1	S1	S1	S1	S1	S1	S1	S1	S1	S1	S1
2	2 most cont. locks on $\neq$ serv. (OS & SL)	S2	S2	S2	S1	S2	S2	S1	S1	S1	S1	S1
3	3 most cont. locks on $\neq$ serv. (OS)	S3	S3	S3	S2	S2	S2	S1	S2	S1	S1	S1
4	4 most cont. locks on $\neq$ serv. (OS)	S4	S4	S3	S3	S2	S3	S2	S2	S1	S1	S1
11	One lock per server	S11	S10	S9	S8	S7	S6	S5	S4	S3	S2	S1

(b) RCL server configurations

Figure 5.12: Server configurations for Berkeley DB with TpccOverBkDB

are used much more than all the other ones and the other ones are all equally used, there is no need to use a configuration for three or four servers for that type of request.

The experiments are run again for each configuration on Magnycours-48 and Niagara2-128, with one client thread per hardware thread. The results are shown in Figure 5.13. The configuration that gives the maximum number of transactions per second is considered optimal and is used for the performance evaluation in Sections 5.3.5.2 and 5.3.5.3.

Magnycours-48. Results for Magnycours-48 are shown in Figure 5.13a. For Order Status, when using only one server, the false serialization rate is high (66.3%), which indicates that many independent critical sections are needlessly executed in mutual exclusion. Adding more servers decreases the false serialization rate: with two, three, four and eleven servers, the false serialization rate drops to 34.1%, 10.6%, 2.5% and 0.0% respectively. Similarly, the use rate drops from 22.7% to 7.9%, 4.0%, 3.2% and 1.9%. This decrease is not only due to the load being shared between servers: going from one server to two servers divides the use rate by more than 2 (2.87), because the execution is faster with two servers (+21%) than with one server due to the decreased amount of false serialization: with one server, false serialization makes the server more busy which leads to more clients waiting for their requests to be executed. Adding more servers reduces false serialization but it also wastes hardware threads that could be used for clients, which is why the peak performance is reached for two servers only. Using three or four servers provides similar performance, which indicates that even if a developer does not perform a deep contention analysis and mistakenly uses a few more servers than needed, the resulting overhead should be low.

For Stock Level, with only one server, both the false serialization rate and the use rate are high (46.4% and 58.5%, respectively). Using two servers is enough to make the false serialization rate drop to almost zero (0.4%). The use rate is divided by 6 which shows that much fewer clients are waiting for their critical section to be executed with two servers. Unsurprisingly, the peak performance is obtained for two servers, but interestingly, using eleven servers is only slightly slower than using two servers ($< 3\%$ performance drop), which shows again that using more servers than needed does not decrease performance significantly: using trial and error to try to decrease the false serialization rate should be sufficient for developers to obtain good results.

	# srv.	Tr. / s	Use rate					False serialization rate				
			S4	S3	S2	S1	Avg.	S4	S3	S2	S1	Avg.
Order Status	1	19,357				22.7%	22.7%				66.3%	66.3%
	2	*23,556*			*7.3%*	*8.4%*	*7.9%*			*23.9%*	*44.2%*	*34.1%*
	3	23,057		4.8%	3.8%	3.4%	4.0%		0.0%	21.2%	10.7%	10.6%
	4	23,432	4.7%	2.1%	3.1%	3.0%	3.2%	0.0%	0.0%	10.1%	0.0%	2.5%
	11	21,706	<4.0% for all 11 servers				1.9%	0.0% for all 11 servers				0.0%
Stock Level	1	1,905				58.5%	58.5%				46.4%	46.4%
	2	*2,351*			*9.4%*	*11.5%*	*10.5%*			*0.1%*	*0.0%*	*0.0%*
	11	2,286	<7.0% for all 11 servers				2.4%	0.0% for all 11 servers				0.0%

(a) RCL server statistics on Magnycours-48

	# srv.	Tr. / s	Use rate					False serialization rate				
			S4	S3	S2	S1	Avg.	S4	S3	S2	S1	Avg.
Order Status	1	11,100				20.3%	20.3%				45.1%	45.1%
	2	18,551			1.8%	2.0%	1.9%			18.4%	35.3%	26.9%
	3	*23,925*		*1.8%*	*2.2%*	*1.7%*	*1.9%*		*0.0%*	*26.8%*	*12.7%*	*13.1%*
	4	23,319	1.7%	1.1%	2.4%	1.8%	1.8%	0.0%	0.0%	15.9%	0.0%	4.0%
	11	22,479	<4.0 for all 11 servers				1.3%	0.0% for all 11 servers				0.0%
Stock Level	1	844				10.9%	10.9%				44.2%	44.2%
	2	*946*			*3.8%*	*4.1%*	*3.9%*			*0.9%*	*0.0%*	*0.4%*
	11	939	<6.0 for all 11 servers				2.4%	0.0% for all 11 servers				0.0%

(b) RCL server statistics on Niagara2-128

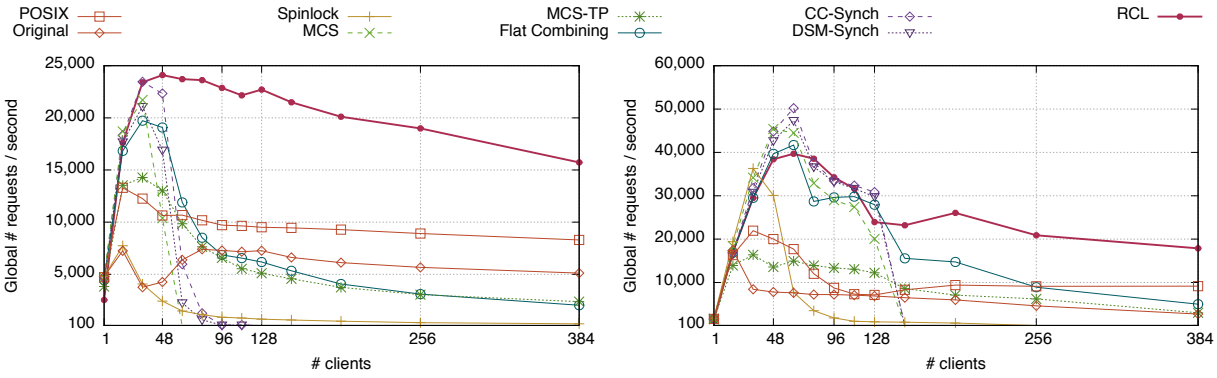
Figure 5.13: Impact of false serialization with RCL (Berkeley DB with TpccOverBkDb)

Niagara2-128. Results for Niagara2-128 are shown in Figure 5.13b. Again, increasing the number of servers decreases the false serialization rate and the use rate. However, since Niagara2-128 has more hardware threads than Magnycours-48, using more servers wastes relatively less CPU resources for the client threads, and the fact that a degraded version of the RCL runtime is used for Niagara2-128 makes false serialization more costly. Because of these two factors, more servers may be needed to reach peak performance: for Order Status, it is reached for three servers instead of two, with a much lower use rate (13.1%) than for the peak configuration of Magnycours-48 (34.1%). Similarly to what was observed on Magnycours-48, for Stock Level, using two servers is enough to remove almost all false serialization. Moreover, adding more servers than needed only slightly decreases performance: using eleven servers instead of the best configuration only leads to overhead of 6.0% for Order Status, and 0.7% for Stock Level.

5.3.5.2 Performance analysis

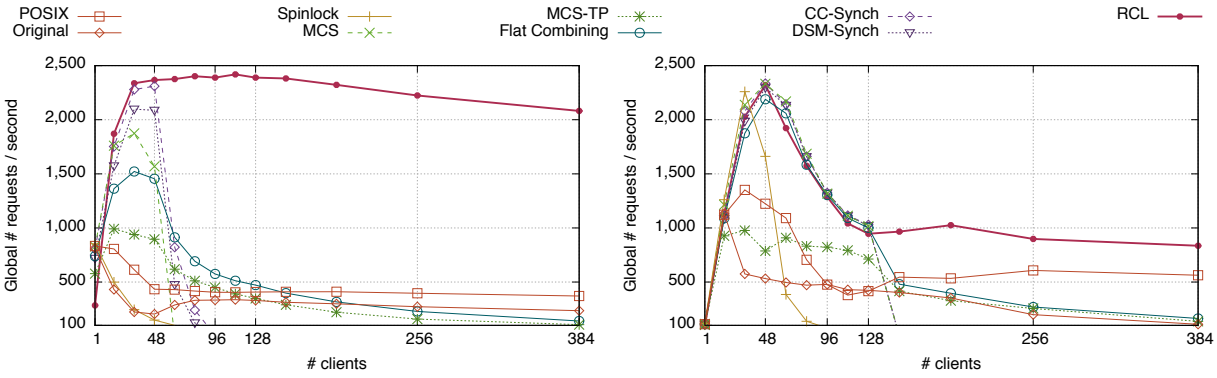
Figure 5.14 shows the performance of Berkeley DB with TpccOverBkDb, using the server configurations chosen in the previous section. In these experiments, each client uses its own thread, and up to 384 clients are run concurrently, which is more than the number of hardware threads of both machines.

Magnycours-48. The results for Magnycours-48 are shown in Figures 5.14a and 5.14c. For both Order Status and Stock Level requests, MCS collapses when more client threads are used than hardware threads because of the convoy effect described in Section 3.5. MCS-TP, which as engineered to prevent convoys, does not suffer from this issue but its peak performance is much lower than that of MCS (14.3K transactions per second instead of 21.7K for Order Status, 0.9K transactions per second instead of 1.9K for Stock Level). CC-Synch and DSM-Synch exhibit good peak performance, but these lock algorithms also collapse rapidly after $h_{m48} = 48$ client



(a) Magnycours-48: Order Status

(b) Niagara2-128: Order Status



(c) Magnycours-48: Stock Level

(d) Niagara2-128: Stock Level

Figure 5.14: Berkeley DB with TpccOverBkDb speedup

threads, i.e., when there are more client threads than hardware threads, which seems to indicate that they suffer from the convoy effect. The basic spinlock performs very poorly except when the number of client threads is low. At 384 client threads, only RCL, the POSIX lock, Berkeley DB's custom lock algorithm (noted "Original" in the figures), Flat Combining and MCS-TP have not collapsed. Interestingly, the POSIX lock is faster than Berkeley DB's custom lock, which may be due to the fact that the lock implementation of Berkeley DB is old and seems to have been designed for SPARC architectures. RCL always performs better than all other lock algorithms: it has the best peak performance (24.1K transactions per seconds for Order Status and 2.4K for Stock Level), and at 384 client threads, for Order Status (resp. Stock Level), using RCL makes the experiment's throughput 89.6% (resp. 460.7%) higher than with the next best lock algorithm (POSIX), and 207.3% (resp. 783.4%) higher than the base application.

Niagara2-128. The results for Niagara2-128 are shown in Figures 5.14c and 5.14d. For Order Status requests, RCL has a lower peak performance (39.7K transactions per second) than CC-Synch (50.2K transactions per second), DSM-Synch (47.4K transactions per second), MCS (45.4K transactions per second), and Flat Combining (41.7K transactions per second). For Stock Level requests, RCL's peak throughput is similar to CC-Synch's, DSM-Synch's and MCS-TP's. However, MCS, CC-Synch and DSM-Synch collapse rapidly after $h_{n128} = 128$ client threads, i.e., when there are more client threads than hardware threads, which may be due to the convoy effect. The basic spinlock performs poorly except when the number of client threads is low. Moreover, when there are more client threads than hardware threads, RCL performs better than

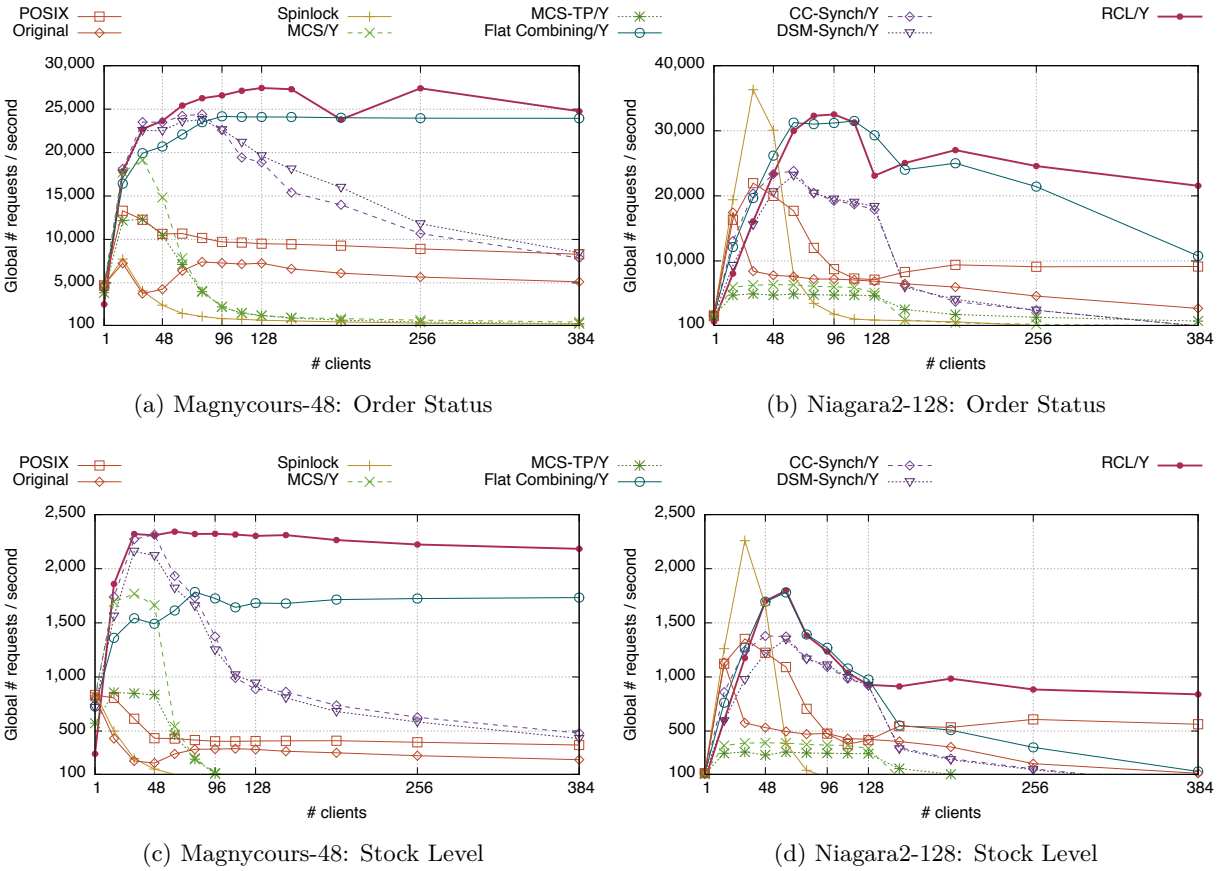
all other lock algorithms: at 384 client threads, for Order Status (resp. Stock Level), using RCL makes the throughput of the experiment 95.3% (resp. 48.3%) higher than with the next best lock algorithm (POSIX), and 557.3% higher (resp. 761.4%) than the base application.

5.3.5.3 Yielding the processor in busy-wait loops

As was explained in Section 3.5, when more application threads are running than there are hardware threads on the machine, lock algorithms that use busy-waiting other than RCL can be preempted while they execute critical sections, with application threads waking up and wasting their time quantum busy-waiting: this can lead to slowdowns and convoys. RCL, on the other hand, always makes progress because its server threads run undisturbed on dedicated hardware threads. However, one could suspect that the experiments from Section 5.3.5.2 are unfair for other lock algorithms than RCL, because a lot of these lock algorithms could simply repeatedly yield the processor in their busy-wait loops (i.e., replace the `PAUSE` instruction in busy-wait loops with a call to the `yield()` function). This technique prevents waiting threads from needlessly wasting CPU resources that could be used by a thread that makes actual progress. In order to investigate the consequences of this optimization, the experiment is run again with modified versions of the lock algorithms that repeatedly yield the processor in busy-wait loops.

Here is how the lock algorithms are modified. The POSIX lock and Berkeley DB's custom lock are not altered because they use sleeping, which already yields the processor. The basic spinlock is not altered either since making it yield the processor in busy-wait loops would make it behave like an inefficient blocking lock. For non-combining locks, application threads yield when they are busy-waiting on their synchronization variable: for MCS, this is done by replacing the `pause()` function call at line 13 of Algorithm 3 with a call to `yield()`, and for MCS-TP a call to `yield()` is inserted between lines 32 and 33 of Algorithm 4 to make sure that threads yield the processor at each iteration of the busy-wait loop. For combining locks and RCL, the clients repeatedly yield the processor when they are waiting for the combiner or server to execute their critical section: this is done by replacing the call to `pause()` by a call to `yield()` (i) at line 16 of Algorithm 6 for Flat Combining, (ii) at line 15 of Algorithm 7 for CC-Synch, (iii) at line 35 of Algorithm 8 for DSM-Synch, and (iv) at line 8 of Algorithm 9 for RCL.

Magnycours-48. Results for Magnycours-48 are shown in Figures 5.15a and 5.15c. Yielding the processor in busy-wait loops reduces the collapse of MCS, CC-Synch and DSM-Synch. It greatly improves the performance of Flat Combining. With yielding, Flat Combining is much more efficient than CC-Synch and DSM-Synch when there are more client threads than hardware threads because Flat Combining always elects a server thread that is running, whereas CC-Synch and DSM-Synch can hand over the role of the server to a descheduled thread, which significantly slows down the critical path. With Stock Level however, CC-Synch and DSM-Synch have a better peak performance than Flat Combining. RCL is still more efficient than all other lock algorithms for any number of clients. Yielding the processor improves the throughput with RCL when a lot of clients are used (+57.3% at 384 threads with Order Status, +5.0% at 384 threads for Stock Level), because clients are able to supply more concurrent requests to the server. This effect is more visible with Order Status than for Stock Level because Order Status has a much higher throughput. At 384 client threads, for Order Status (resp. Stock Level), using RCL makes the experiment's throughput 3.4% (resp. 26.0%) higher than with the next best lock algorithm (Flat Combining), and 383.6% (resp. 826.9%) higher than the base application.



(a) Magnycours-48: Order Status

(b) Niagara2-128: Order Status

(c) Magnycours-48: Stock Level

(d) Niagara2-128: Stock Level

Figure 5.15: Berkeley DB with TpccOverBkDb speedup, using sleeping

Niagara2-128. On Niagara2-128, yielding the processor in busy-wait loops alleviates the collapse of MCS, CC-Synch and DSM-Synch, but it also reduces their peak performance significantly. Yielding improves throughput in Order Status when a lot of clients are used (+20.6% at 384 threads). RCL has the best peak performance (equal to that of Flat Combining) and the best performance at 384 threads of all of the lock algorithms, even when they yield the processor in busy-wait loops. At 384 client threads, for Order Status (resp. Stock Level), using RCL makes the experiment's throughput 100.0% (resp. 562.6%) higher than with the next altered lock algorithm (Flat Combining), and 692.7% higher (resp. 564.2%) than the base application.

To conclude this section, even though yielding the processor in busy-wait loops can improve the performance of some lock algorithms when applications use more software threads than there are hardware threads, it also improves the performance of RCL at high throughput, and RCL still performs better than other lock algorithms with this optimization.

Note on the evaluation from the USENIX ATC paper. The reader may have noticed that, while being quantitatively similar, some of the results shown in this thesis differ slightly from those published in the USENIX ATC paper [71]. This is due to several factors. First, Magnycours-48 is a different machine from the one that was used for the evaluation in the USENIX ATC paper, even though it has similar characteristics. Moreover, the software it uses (kernel options, library versions, environment, etc.) may not be exactly identical. Second, while

porting the codebase to Solaris, the profiler and some of the benchmarks have been optimized. Here is a non-exhaustive list of modifications that were made to the codebase:

- The critical sections of the microbenchmark have been simplified since the original RCL paper: instead of using a complex function that is able to access any number of cache lines while using tricks to avoid prefetching, two highly optimized functions that directly access either one or five cache lines are now used. Moreover, cycles are now measured by reading the timestamp counter directly via assembly code, instead of using PAPI, similarly to what has been done with the profiler and explained in Section 5.3.1.
- The profiler has been improved, as was explained in Section 5.3.1.
- When using Memcached, the hashtable is preloaded with more entries (30,000 instead of 10,000) when using Get requests. Preloading the hashtable with more entries puts more pressure on locks because the critical sections that protect the hashtable lookup last longer. While RCL significantly improves performance with 10,000 entries, the performance gain is higher when more entries are used and becomes stable for 30,000 entries onwards. Since it is not uncommon for a Memcached hashtable to contain 30,000 entries or more, this value was chosen for the new evaluation.
- Finally, a bug was found in the Berkeley DB experiment. The size of a pool of locks had to be increased: when the throughput was high (many clients with a fast lock), some requests could fail because the large number of concurrent requests could deplete the lock pool. Fortunately, this bug only affected a few data points for the most efficient lock algorithms and was mostly detrimental to RCL. Additionally, the way throughput is measured was slightly modified: instead of averaging the throughputs per thread (which was the only information given by the benchmark), the benchmark was modified to measure the global throughput, which is more precise and may alter the results slightly. Finally the results shown in Figure 5.14c may appear quantitatively different from the Berkeley DB graph from Figure 11 of the USENIX ATC paper, but this is simply due to the fact that the new graphs show the global throughput, instead of the average throughput per thread.

Both versions of the codebase are publicly available online. They include the the RCL runtime, the Liblock, the profiler, the reengineering, the as well as the evaluated applications with the scripts used for the evaluation. The download URL can be found in Chapter 6.

5.4 Conclusion

Section 5.2 has shown that on both Magnycours-48 and Niagara2-128, RCL performs better than other lock algorithms on a custom microbenchmark that generates lock contention: it collapses at a higher contention than other lock algorithms and it performs better at maximum contention.

As seen in Section 5.3, using the time spent in critical sections as a metric to identify when a legacy application can benefit from RCL is an efficient criterion: out of the thirty-eight profiled experiments that use POSIX locks (on each machine: ten experiments from SPLASH-2, seven from Phoenix 2, and two for Memcached), only two false negatives were returned (Memcached/Set on each machine), but using the number of cache misses as a secondary metric helped detecting these false negatives. Only one false positive was found: Raytrace/Car on Niagara2, for which POSIX locks were more efficient than all other lock algorithms even though the experiment spends more time in critical sections than both thresholds.

In most cases, when the time spent in critical sections is high or when critical sections have poor locality, RCL is more efficient than other lock algorithms: it improves both peak performance and scalability, even compared to state-of-the-art lock algorithms like CC-Synch and DSM-Synch. In particular, when a large number of application threads run or a lower number of hardware threads, RCL performs very well because its threads that execute critical sections are never preempted by application threads. A typical technique used to improve the performance of lock algorithms in that case is to make locks yield the processor instead of busy-waiting, but RCL still largely outperforms other lock algorithms when this optimization is used.

One drawback of RCL is that when several locks are contended, profiling the application is needed in order to find optimal repartition of locks on servers. However, as shown in Section 5.3.5.1, using trial and error in order to minimize the false serialization rate is sufficient to obtain near-optimal performance.

Chapter 6

Conclusion

This thesis has presented RCL, a novel locking technique that focuses on both reducing lock acquisition time and improving the execution speed of critical sections through increased data locality. The key idea is to go one step further than combining locks, and to dedicate hardware threads for the execution of critical sections: since current multicore architectures have dozens of hardware threads at their disposal that cannot be fully exploited because applications lack scalability, dedicating some of these hardware threads for a specific task such as serving critical sections can only improve their performance. RCL takes the form of a runtime library for Linux and Solaris that supports x86 and SPARC architectures. In order to ease the reengineering of legacy applications, RCL proposes a profiler as well as a methodology for detecting highly contended locks and locks whose critical sections suffer from poor data locality, since these two kinds of locks can generally benefit from RCL. Once these locks have been identified, RCL can be used with a minimal amount of work, thanks to a reengineering that encapsulates critical sections into functions: this tool makes RCL as well as combining locks easily usable in legacy applications.

RCL is evaluated on the applications from the SPLASH-2 and Phoenix 2 benchmark suites, Memcached, and Berkeley DB with a TPC-C client. Two machines are used in the evaluation: Magnycours-48, an x86 machine that runs Linux with four AMD Opteron processors and 48 hardware threads, and Niagara2-128, a SPARC machine that runs Solaris with two Sun UltraSPARC T2+ processors and 128 hardware threads. The profiler efficiently finds locks that can benefit from RCL: out of thirty-eight experiments in total, the profiler only returns one false positive. Performance evaluations show that on both machines, when applications suffer from high contention or poor data locality in critical sections, RCL improves both peak performance and scalability: for instance, Memcached with Set requests on Magnycours-48 (resp. Niagara2-128) is 2.5 times (resp. 1.3 times) faster with RCL than it is with POSIX locks, and it scales up to 19 hardware threads with RCL threads instead of only 3 hardware threads for POSIX locks. RCL even offers significant performance and scalability gains over the most recent state-of-the-art combining locks: for instance, on Magnycours-48[1], SPLASH-2's Raytrace is up to 37.3% (resp. 36.7%) faster with RCL than with CC-Synch (resp. DSM-Synch), and it scales up to 23 hardware threads instead of 15 (resp. 18) hardware threads for CC-Synch (resp. DSM-Synch). Furthermore, in contrast with combining locks, RCL performs very well when more application threads are used than there are hardware threads on the machine: for instance, on Berkeley DB with the TPC-C client, Stock Level requests and 384 client threads,

[1]Raytrace does not suffer from high contention or poor data locality in critical sections on Niagara2-128.

RCL makes it possible to reach a throughput of more than 2,000 (resp. 800) requests per second on Magnycours-48 (resp. Niagara2-128), while the throughput collapses to less than 10 requests per second with CC-Synch and DSM-Synch. Yielding the processor in busy-wait loops alleviates the collapse of CC-Synch and DSM-Synch on Magnycours-48, but even with this optimization, the throughput of Berkeley DB is 4.5 times (resp. 5.1 times) higher with RCL than with CC-Synch (resp. DSM-Synch) in the same experiment. The reason why RCL performs well when there are more application threads than hardware threads is that its server threads never get preempted: the server always makes progress along the critical path.

Future research directions. In order to further improve the performance of RCL, four research directions could be explored. First, an adaptive RCL runtime could be designed. Such a runtime would (i) dynamically switch between locking strategies, so as to dedicate a server hardware thread only when a lock is contented and (ii) migrate locks between multiple servers, in order to dynamically balance the load and avoid false serialization. One of the challenges would be to implement low-overhead runtime profiling and migration strategies. Second, a hierarchical version of RCL could be designed, possibly using one transient server per cluster of hardware threads, i.e., a group of hardware threads that is located either on a single die, CPU, or NUMA node. Lock Cohorting [34] could serve as a basis to this approach, since it makes it possible to build hierarchical locks out of non-hierarchical locks. The performance of a hierarchical version of RCL could be compared to existing hierarchical locks, such as HBO [90], H-CLH [73], and hierarchical locks created by Dice et al. thanks to Lock Cohorting. Third, a modified RCL algorithm for real-time systems that supports thread priorities could be written. Brandenburg [16] has started this work: in a paper published at RTAS '13 that cites the RCL USENIX ATC paper, he compares the performance of real-time locking protocols for partitioned fixed-priority (P-FP) scheduling, and his Distributed FIFO Locking Protocol (DFLP) resembles RCL since it uses a designated synchronization processor for executing critical sections. However, like combining locks, it uses a FIFO queue for pending requests instead of an array. Finally, the last obstacle to optimal performance with RCL is that each time a server executes a critical section, it has to signal the corresponding client thread that its critical section has been executed. This is done by resetting a variable in the client's request mailbox, and writing to that mailbox may block the execution of the server because it has to fetch the cache line that holds the variable from the client in write mode. Implementing hardware support for allowing to always run this store operation in the background, while the server keeps executing critical sections, would completely eliminate all synchronization on the critical path and therefore allow RCL to reach optimal speed.

Perspectives. The evolution of CPUs goes through different phases. After an initial phase during which CPU manufacturers were able to steadily increase clock frequency for decades in order to improve sequential performance, a second phase started about fifteen years ago, with the emergence of multicore computing that forced application developers to make their applications exploit parallelism. Nowadays, as CPUs offer always more hardware threads, as memory hierarchies become always more complex, as inter-core connections become network-like, and as parallel algorithms that become more and more refined to try to exploit the newly offered processing power, CPU manufacturing is entering a third phase. Two major evolutions are on the brink of radically changing once more the way applications are designed: hardware support for transactional memory and the rise of non-cache-coherent architectures.

Transactional memory has been proposed as an alternative to locking several decades ago [61, 54, 98], with the promise to greatly simplify the conception of parallel algorithms, by removing issues that are inherent to locks such as deadlocks, livelocks and priority inversions. As applications

use more and more complex fine-grained locking schemes that lead developers to introduce various concurrency bugs in their applications, using transactional memory is becoming more and more appealing on modern multicore architectures. However, due to the lack of performance of software implementations and a lack of hardware support until very recently, using transactional memory is still rare in real-world applications, and locks are still the most used synchronization primitive. Things may change with the latest generation of x86 Intel processors, codenamed Haswell, since these processors come with Transactional Synchronization Extensions (TSX), i.e., hardware support for transactional memory. The fact that hardware transactional memory is now supported on widely-available consumer CPUs may lead it to supplant locking in the near future. In this context, applying the key concepts that make RCL so efficient at synchronization and locality will have to be investigated in the context of transactional memory. Hassan et al. [47], in a paper published at IPDPS '14, have started this work, by proposing a new software transactional memory algorithm that executes commit and invalidation routines on dedicated remote server threads. Like RCL, they use cache-aligned communication between the client and server threads. The efficient implementation they propose for their algorithm, RInval, uses RCL for locks. The next step will be to produce a similar algorithm for hardware transactional memory.

Finally, as explained in Section 2.3.1.3, the overhead of the cache-coherence protocol becomes worse when the number of cores increases. In the future, architectures will resemble more and more distributed systems, in which hundreds, or thousands of cores, will communicate over a complex, non-uniform network. Each core will own its own caches and memory banks, with no global view of the whole memory: message-passing will have to be used for all communication between threads. In this context, RCL is an interesting approach to locking, because it is reminiscent of the way mutual exclusion is generally handled in distributed systems: a single server is dedicated to the execution of work that must be executed sequentially. It would be interesting to analyze how well RCL performs on such architectures. Petrovič et al. [88] have started this analysis on partially non-cache-coherent architectures: in a paper published at PPoPP '14, they propose a universal construction inspired from RCL and combining locks that dedicates servers to the execution of critical sections on partially non-cache-coherent architectures. They show that this approach outperforms combining locks on Tilera's TILE-Gx partially non-cache-coherent CPUs. Analyzing the performance of RCL on fully non-cache-coherent CPUs is the next step.

Later works that cite RCL. Following the publications that presented RCL (in USENIX ATC '12 [71] and CFSE '8 [69], as well as the INRIA research report [72], the EuroSys 2011 poster and the SOSP '11 WiP session abstract), RCL has been cited by several other papers. As was mentioned in the previous paragraphs, (i) Brandenburg [16], in a paper published at RTAS '13, compares the performance of real-time locking protocols for partitioned fixed-priority (P-FP) scheduling and cites RCL as an example of a distributed locking protocol: the DFLP algorithm they use is reminiscent of RCL, (ii) Hassan et al. [47], in a paper published at IPDPS '14, propose a new algorithm for software transactional memories that executes commit and invalidation routines on dedicated remote server threads, with an implementation that uses RCL internally, and (iii) Petrovič et al. [88], in a paper published at PPoPP '14, propose a universal construction inspired from RCL and combining locks that dedicates servers to the execution of critical sections on partially non-cache-coherent architectures. Pusukuri et al. [89], in a paper also published at PPoPP '14, propose to migrate threads across multicore architectures so that threads seeking locks are more likely to find them on the same core, which is similar to RCL in that it improves data locality of critical sections. David et al. [30], in a paper published at SOSP '13, present an exhaustive study of synchronization that discusses RCL among other

lock mechanisms. Gidra et al. [44], in a paper published at ASPLOS '13, study the scalability of stop-the-world garbage collectors on multicore architectures and cite the USENIX ATC RCL paper as an example of a study that shows that synchronization is a bottleneck on architectures with a large number of cores. RCL has also been cited in papers that were published (i) in conferences such as OPODIS '13 [21], Euro-Par '13 [97], CLUSTER '13 [3], ECRTS '13 [20], and SAMOS '14 [84]; (ii) in journals [23, 19]; (iii) in workshops [36, 46]; and (iv) as technical reports [109, 29]. Additionally, some articles published online cite RCL, such as Dmitry Vyukov's article about combiner/aggregator synchronization primitives from Intel's Developer Zone [108].

Availability. The implementation of the RCL runtime, the Liblock, the profiler, the reengineering, as well as the test scripts and results are available at the following URL:
`http://rclrepository.gforge.inria.fr`

Appendix A

French summary of the thesis

Synthèse du rapport de thèse en français

Following the rules of the Université Pierre et Marie Curie, this appendix is a short summary of the thesis, written in French. *Afin de suivre les règles de l'université Pierre et Marie Curie, cette annexe contient une synthèse du rapport de thèse en français.*

Résumé

Le passage à l'échelle des applications multi-fil sur les systèmes multi-cœur actuels est limité par la performance des algorithmes de verrou, à cause des coûts d'accès à la mémoire sous forte congestion et des défauts de cache. La contribution principale présentée dans cette thèse est un nouvel algorithme, Remote Core Locking (RCL), qui a pour objectif d'améliorer la vitesse d'exécution des sections critiques des applications patrimoniales sur les architectures multi-cœur. L'idée de RCL est de remplacer les acquisitions de verrou par des appels de fonction distants (RPC) optimisés vers un fil d'exécution matériel dédié appelé *serveur*. RCL réduit l'effondrement des performances observé avec d'autres algorithmes de verrou lorsque de nombreux fils d'exécution essaient d'obtenir un verrou de façon concurrente, et supprime le besoin de transférer les données partagées protégées par le verrou vers le fil d'exécution matériel qui l'acquiert car ces données peuvent souvent demeurer dans les caches du serveur.

D'autres contributions sont présentées dans cette thèse, notamment un profiler permettant d'identifier les verrous qui sont des goulots d'étranglement dans les applications multi-fil et qui peuvent par conséquent être remplacés par RCL afin d'améliorer les performances, ainsi qu'un outil de réécriture de code développé avec l'aide de Julia Lawall. Cet outil transforme les acquisitions de verrou POSIX en acquisitions RCL. L'évaluation de RCL a porté sur dix-huit applications: les neuf applications des benchmarks SPLASH-2, les sept applications des benchmarks Phoenix 2, Memcached, ainsi que Berkeley DB avec un client TPC-C. Huit de ces applications sont incapables de passer à l'échelle à cause de leurs verrous et leur performance est améliorée par RCL sur une machine x86 avec quatre processeurs AMD Opteron et 48 fils d'exécution matériels. Utiliser RCL permet de multiplier les performances par 2.5 par rapport aux verrous POSIX sur Memcached, et par 11.6 fois sur Berkeley DB avec le client TPC-C. Sur

une machine SPARC avec deux processeurs Sun Ultrasparc T2+ et 128 fils d'exécution matériels, les performances de trois applications sont améliorées par RCL: les performances sont multipliées par 1.3 par rapport aux verrous POSIX sur Memcached et par 7.9 fois sur Berkeley DB avec le client TPC-C.

Mots-clé. Multicœur, synchronisation, verrou, combining, RPC, attente active, congestion mémoire, profiling, transformation de code.

Contexte

Ce document présente les principaux travaux de recherche de Jean-Pierre Lozi, conduits au sein des équipes Regal, puis Whisper, au Laboratoire d'Informatique de Paris 6 (LIP6), en vue d'obtenir le titre de Docteur en Informatique de l'école doctorale "École Doctorale Informatique, Télécommunications et Électronique" (EDITE) de Paris. Les travaux de recherche présentés dans ce document ont été encadrés par Gilles Muller et Gaël Thomas (LIP6/INRIA). Ces travaux ont permis les publications suivantes, à une conférence française ainsi qu'à une conférence internationale :

- ***Le Remote Core Lock (RCL): une nouvelle technique de verrouillage pour les architectures multi-cœur.*** Jean-Pierre Lozi. *8ème Conférence Française en Systemes d'Exploitation (CFSE '8).* Saint-Malo, France, 2011. Best Paper award. [69]
- ***Remote Core Locking: Migrating Critical-section Execution to Improve the Performance of Multithreaded Applications.*** Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall and Gilles Muller. *Dans Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC '12).* Boston, USA, 2012. [71]

Un rapport de recherche INRIA [72] a également été produit, et ces travaux ont été présentés à de nombreuses occasions, comme par exemple à la session Posters d'EuroSys 2011 à Salzbourg, en Autriche, et à la session Works in Progress (WiP) de SOSP'11 à Cascais, au Portugal.

Enfin, en parallèle avec ces travaux de recheche, j'ai participé à d'autres projets qui ont mené aux publication suivantes:

- ***EHCtor: Detecting Resource-Release Omission Faults in Error-Handling Code for Systems Software.*** Suman Saha, Jean-Pierre Lozi. *9ème Conférence Française en Systemes d'Exploitation (CFSE '9).* Grenoble, France, 2013. [94]
- ***Hector: Detecting Resource-Release Omission Faults in Error-Handling Code for Systems Software.*** Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia Lawall, and Gilles Muller. *Dans Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13).* Budapest, 2013. Best Paper award. [95]

Bien qu'ayant été produits au cours du doctorat, ces travaux n'en étaient pas le travail principal. Par conséquent, ils ne sont ni décrits dans cette synthèse, ni dans le rapport de thèse.

A.1 Introduction

Jusqu'aux années 2000, les microprocesseurs voyaient leur performance s'améliorer de façon continue grâce à l'augmentation de leur fréquence d'horloge. Cependant, dans les années 2000, des limites physiques ont été atteintes qui ont rendu de plus en plus difficile de continuer à augmenter la fréquence d'horloge. Afin de continuer à augmenter la performance des microprocesseurs, les fondeurs ont alors commencé à inclure plusieurs unités d'exécution dans le même microprocesseur. Ces unités d'exécution sont appelées *cœurs*, et des technologies telles que le Simultaneous MultiThreading (SMT) répliquent certaines parties de ces cœurs afin de leur permettre d'exécuter plusieurs *fils d'exécution matériels* simultanément. Le nombre de fils d'exécution matériels qu'offrent les microprocesseurs sont en constante augmentation dans tous les appareils électroniques actuels, des téléphones portables aux serveurs d'entreprise. Il n'est pas rare de nos jours pour les serveurs multi-processeur actuels de disposer de dizaines de fils d'exécutions matériels.

Malheureusement, alors qu'augmenter la fréquence d'horloge des processeurs permettait d'améliorer les performances des applications qu'ils exécutaient de manière quasi-automatique, faire usage de la puissance de traitement offerte par des fils d'exécution matériels supplémentaires est une tâche complexe car la plupart des programmes ne peuvent pas être complètement parallélisés. La Loi d'Amdahl [4] montre que lorsque le nombre de fils d'exécution matériels augmente, le chemin critique, c'est-à-dire les parties du code applicatif qui ne peuvent pas être parallélisées, finit par être le facteur limitant l'amélioration des performances. C'est pourquoi, afin d'utiliser les architectures multi-cœur efficacement, il est important de réduire le plus possible la taille du chemin critique. Dans la plupart des applications patrimoniales, le chemin critique consiste principalement en des *sections critiques*, c'est-à-dire des sections de code qui doivent être exécutées en exclusion mutuelle. Les sections critiques sont généralement protégées par des *verrous*, c'est-à-dire des mécanismes de synchronisation qui permettent à plusieurs fils d'exécution d'assurer l'exécution en exclusion mutuelle de sections de code qui accèdent aux mêmes ressources.

Afin de mieux exploiter la puissance de traitement des architectures multi-cœur récentes, les applications doivent être optimisées afin d'exploiter le parallélisme efficacement, ce qui peut se révéler être une tâche complexe: alors que faire en sorte que les applications passent à l'échelle jusqu'à quelques fils d'exécution matériels est relativement aisé en utilisant des techniques naïves de parallélisation à gros grain, faire en sorte que les applications passent à l'échelle sur les architectures multi-cœur actuelles, qui offrent des dizaines de fils d'exécution matériels, reste un défi. Étant donné qu'un grand nombre d'applications complexes a été développé au cours des dernières décennies pour des architectures n'offrant que quelques fils d'exécution matériels tout au plus, réécrire une grande partie de leur code source afin d'exploiter les architectures multi-cœur actuelles est une tâche complexe qui requiert un travail considérable.

Une manière de faire en sorte que les applications patrimoniales passent à l'échelle sur les architectures multi-cœur est de réduire la taille des sections critiques, c'est-à-dire d'utiliser du *verrouillage à grain fin* [6, 55]: bien que cette approche soit généralement considérée très efficace, elle peut être extrêmement complexe à mettre en œuvre pour les développeurs. Il n'existe pas de technique générale permettant de réduire la taille des sections critiques: les développeurs doivent utiliser des approches différentes dans différents cas, et ce faisant, ils complexifient la logique concurrente des applications et y insèrent souvent des bogues liés à la programmation concurrente. Ces bogues sont connus pour être très difficiles à identifier. Il est également possible d'éviter l'utilisation de sections critiques dans certains cas et d'utiliser uniquement des instructions atomiques pour la synchronisation, grâce aux algorithmes et structures de données dits *sans*

verrou [56, 77, 52, 40, 62, 63]. Alors que ces algorithmes, très efficaces dans certains cas, ont été proposés pour la plupart des structures de données classiques telles que les piles, les files d'attente, ou les listes à enjambements, il n'est pas possible de remplacer n'importe quel ensemble de sections critiques par un algorithme sans verrou efficace. Finalement, les *mémoires transactionnelles*, implémentées au niveau logiciel ou matériel, ont été proposées comme remplacement pour les verrous [61, 54, 98, 53, 45]. Bien que les mémoires transactionnelles soient moins complexes à utiliser que les verrous, elles ne sont actuellement que peu utilisées à cause d'un manque de support matériel efficace et des faibles performances des implémentations logicielles. De plus, avec les mémoires transactionnelles, les sections critiques ne peuvent exécuter d'opérations pouvant être annulées, comme par exemple la plupart des opérations d'entrée/sortie.

D'autres solutions ont été proposées pour exploiter au mieux les performances des architectures multi-cœur. Certaines approches se concentrent sur l'optimisation d'un mécanisme bien particulier pour ces machines, comme les appels de fonction à distance (Remote Procedure Calls, ou RPC) [10, 11, 42], mais ces techniques sont trop limitées pour rendre possible une utilisation efficace de la puissance de traitement des machines multi-cœur dans le cas général. Certaines optimisations spécifiques de certaines parties des systèmes d'exploitation comme par exemple l'ordonnanceur ont été proposées pour ces machines [64], mais ces améliorations sont complémentaires à une modification des applications pour les faire passer à l'échelle. De nouveaux systèmes d'exploitation spécifiquement optimisés pour les machines multi-cœur ont été proposés, tels qu'Opal [22], Corey [13], Multikernel [9], et Helios [78], mais les systèmes d'exploitation patrimoniaux ainsi que leurs applications sont devenus si évolués au cours des dernières décennies que migrer vers de nouveaux systèmes à l'heure actuelle pourrait engendrer une perte de productivité considérable. En effet, il faudrait des années aux nouveaux systèmes d'exploitation pour qu'ils atteignent le même niveau de fonctionnalité que les systèmes partimoniaux d'une part, et d'autre part, les applications doivent souvent être complètement réécrites afin de pouvoir être utilisées sur ces nouveaux systèmes. Certains outils ont été proposés pour aider à modifier les applications afin de les rendre performantes sur les architectures multi-cœur actuelles, tels que les profilers [87, 65], qui peuvent être utiles pour détecter leurs goulots d'étranglement. Les profilers ne proposent cependant pas de solution pour supprimer ces goulots d'étranglement: ils permettent uniquement de les identifier.

Une autre manière d'améliorer les performances des applications sur les architectures multi-cœur est la mise au point de verrous plus efficaces. L'avantage principal de cette approche est qu'elle ne nécessite pas de modification conséquente des applications. Ces vingt dernières années, un nombre important de travaux [2, 8, 12, 49, 51, 59, 75, 96, 100, 102] ont eu pour objectif d'améliorer la performance des verrous sur les architectures multi-cœur, soit en réduisant soit la congestion d'accès, soit en améliorant localité mémoire. La congestion d'accès se produit lorsque de nombreux fils d'exécution essaient d'entrer simultanément en section critique: les fils d'exécution saturent le bus de données de messages de synchronisation avant qu'un fil d'exécution ne soit enfin élu pour entrer seul en section critique. Le manque de localité mémoire devient problématique lorsqu'une section critique accède à des données partagées qui ont récemment été modifiées par un autre fil d'exécution, ce qui peut causer des défauts de cache et par conséquent augmenter fortement le temps d'exécution des sections critiques. Diminuer la congestion d'accès tout en améliorant la localité mémoire reste un défi, d'autant plus que lorsque le nombre de fils d'exécution matériels augmente, l'augmentation du nombre de fils d'exécution s'exécutant en parallèle a tendance à augmenter la congestion d'accès et à réduire la localité mémoire.

Récemment, plusieurs approches ont été proposées pour exécuter un grand nombre de sections critiques de manière consécutive sur un seul fil d'exécution matériel *serveur* (ou *combinateur*)

afin d'améliorer la localité des données [51, 102, 39]. De telles approches utilisent un transfert de contrôle rapide entre le serveur et les autres fils d'exécution (*clients*) afin de réduire la congestion d'accès. Suleman et al. [102] proposent une solution matérielle, évaluée en simulation, qui introduit de nouvelles instructions afin de permettre le transfert de contrôle, et qui utilise un fil d'exécution matériel sur un cœur rapide pour exécuter les sections critiques. Des algorithmes ne nécessitant pas de modification de la couche matérielle ont également été proposés. Dans ces algorithmes, les fils d'exécution clients deviennent tour à tour serveurs [82, 51, 39]. Ces algorithmes sont appelés *verrous à combinateur*. Ces algorithmes sont plus rapides que les verrous traditionnels, mais la gestion du rôle de serveur engendre parfois un surcoût, et ils sont vulnérables à la préemption. De plus, ni l'algorithme de Suleman et al. ni les verrous à combinateur ne proposent un mécanisme pour gérer les variables de condition, ce qui les rend inutilisables pour de très nombreuses applications.

L'objectif des travaux présentés dans ce rapport de thèse est de diminuer le temps passé par les applications à exécuter des sections critiques en évitant toute modification manuelle des applications, et ce, en se concentrant sur la réduction du temps nécessaire pour entrer en section critique et l'amélioration de la localité mémoire. La contribution principale présentée dans ce rapport de thèse est une nouvelle technique de verrouillage, Remote Core Locking (RCL), qui a pour but d'améliorer l'exécution des sections critiques en exécutant sur un ou plusieurs fils d'exécution matériels dédiés les sections critiques qui sont protégés par des verrous souffrant de haute congestion d'accès. Cette approche est complètement implémentée dans la couche logicielle et vise les architectures x86 et SPARC. L'idée de RCL vient de l'observation suivante : sur les architectures multi-cœur actuelles, les applications ne passent pas à l'échelle, et par conséquent, de nombreux fils d'exécution matériels ne peuvent pas être exploités par ces applications. Par conséquent, il est possible de les dédier pour une autre tâche, en l'occurrence l'exécution de sections critiques, afin d'améliorer la performance des applications. Il n'est par conséquent pas nécessaire de faire porter la charge de serveur aux fils d'exécution applicatifs, comme le font les verrous à combinateur. RCL améliore à la fois la gestion de la congestion d'accès et la localité des données. La congestion est réduite grâce aux transferts de contrôle rapides entre clients et serveurs, en utilisant une ligne de cache dédiée pour chaque client, sur laquelle le client et le serveur se synchronisent par attente active. La localité est améliorée car les données partagées restent dans les caches du serveur, ce qui permet à ce dernier d'y accéder sans causer de défauts de cache. En ce qui concerne ce dernier point, RCL est similaire aux verrous à combinateur, mais il s'agit d'une approche moins coûteuse qui résiste mieux à la contention, et qui résiste également mieux à la préemption car le fil d'exécution serveur progresse toujours. D'autre part, RCL propose un mécanisme pour permettre l'utilisation de variables de condition, ce qui le rend directement utilisable dans les applications réelles. RCL est un bon outil pour améliorer les performances des applications patrimoniales dans lesquelles les verrous congestionnés sont un goulot d'étranglement, puisque utiliser RCL permet d'améliorer la résistance à la congestion et la localité sans nécessiter une compréhension poussée du code source. À l'inverse, modifier les applications pour utiliser du verrouillage à grain fin, des algorithmes sans verrous ou des mémoires transactionnelles nécessite des modifications majeures du code source, ce qui implique des coûts de redéveloppement importants. De plus, ces approches n'améliorent pas la localité.

Cette thèse présente également une méthodologie ainsi qu'un ensemble d'outils qui facilitent l'utilisation de RCL dans les applications patrimoniales. Comme RCL sérialise l'exécution des section critiques associées aux verrous gérées par le même fil d'exécution matériel serveur, transformer tous les verrous en RCL sur un nombre limité de serveurs induit de la *fausse sérialisation* : certains serveurs sérialisent l'exécution de sections critiques qui sont protégées par

différents verrous et qui par conséquent ne nécessitent pas d'être exécutées en exclusion mutuelle. Dans certains cas, la fausse sérialisation peut engendrer un surcoût conséquent. Par conséquent, le programmeur doit d'abord décider quels verrous doivent être transformés en RCLs et quels serveurs gèrent quels verrous. Un profiler a été écrit dans ce but. Ce profiler permet d'identifier quels verrous sont utilisés fréquemment par l'application, de mesurer le temps passé en section critique, et d'estimer si la localité mémoire des sections critiques est bonne. À partir de ces informations, un ensemble d'heuristiques simples sont proposées pour aider le programmer à décider quels verrous doivent être transformés en RCLs. Un outil de transformation automatique de code a été développé avec l'aide de Julia Lawall pour transformer le code des sections critiques afin qu'il puisse être exécuté comme appel de fonction à distance sur le fil d'exécution matériel serveur : le code de chaque section critique doit être extrait dans une fonction. L'argument passé à cette fonction est son objet de *contexte*, c'est-à-dire un objet qui contient des copies de toutes les variables référencées ou mises à jour par la section critique qui sont déclarées dans la fonction contenant le code de la section critique. RCL prend la forme d'un environnement d'exécution pour Linux et pour Solaris qui est compatible avec les fils d'exécution POSIX, et qui permet l'utilisation commune de verrous POSIX et RCL au sein de la même application.

Les performances de RCL sont comparées avec celles d'autres algorithmes de verrou à l'aide d'un microbenchmark mesurant le temps d'exécution de sections critiques qui accèdent à un nombre variable d'emplacements mémoire. De plus, grâce aux résultats du profiler, trois applications de la suite SPLASH-2 [107, 99, 110], trois applications de la suite Phoenix 2 [101, 103, 112, 92], Memcached [26, 41], et Berkeley DB [80, 79] avec un benchmark TPC-C ont été identifiés comme pouvant être rendues plus performantes grâce à RCL. Dans chacune de ces applications, RCL est comparé à un verrou à attente active, aux verrous POSIX, à MCS [75], et à Flat Combining [51]. RCL est aussi comparé avec CC-Synch et DSM-Synch [39], deux algorithmes qui ont été publiés après RCL. Les comparaisons ont été faites pour le même nombre de fils d'exécution matériels, ce qui veut dire qu'il y a moins de fils d'exécution applicatifs dans le cas de RCL, puisqu'entre un et trois fils d'application matériels sont dédiés aux serveurs RCL.

RCL est évalué sur deux machines: (i) Magnycours-48, une machine x86 sous Linux qui dispose de quatre microprocesseurs AMD Opteron pour un total de 48 fils d'exécution matériels, et (ii) Niagara2-128, une machine SPARC sous Solaris qui dispose de deux microprocesseurs UltraSPARC T2+ pour un total de 128 fils d'exécution matériels. Voici une sélection de résultats provenant de l'évaluation[1] :

- Sur le microbenchmark, à haute contention, RCL est plus rapide que toutes les autres approches évaluées: sur Magnycours-48 (resp. Niagara2-128), RCL est 3.2 (resp. 1.8) fois plus rapide que le deuxième meilleur verrou, CC-Synch, et 5.0 (resp. 7.2) fois plus rapide que le verrou POSIX.

- Sur les applications des suites SPLASH-2 et Phoenix 2, Memcached, et Berkeley DB avec un client TPC-C, les objets de contexte sont petits, et par conséquent les passer aux serveur engendre un surcoût négligeable.

- Dans la plupart des expériences, seul un verrou est fréquemment utilisé, et par conséquent, un seul serveur RCL est requis. La seule exception est Berkeley DB avec le client TPC-C, puisque dans ce cas, deux à trois clients RCL sont requis pour atteindre la performance optimale en réduisant la fausse sérialisation.

[1] Certains de ces résultats ne sont pas détaillés dans la section Évaluation de cette synthèse (Chapitre A.3). Pour plus de détails, le lecteur pourra se réferer à la version non-synthétique du rapport de thèse.

- Sur Magnycours-48 (resp. Niagara2-128), RCL améliore la performance de cinq (resp. d'une) applications des suites SPLASH-2 et Phoenix-2. Pour ces applications, RCL permet un gain en performance supérieur à celui des autres verrous évalués.
- Pour Memcached avec l'utilisation de requêtes Set, sur Magnycours-48 (resp. Niagara2-128), RCL permet de multiplier la bande passante par 2.5 (resp. 1.3) par rapport au verrou POSIX, par 1.9 (resp 1.2) par rapport au verrou à attente active, et par 2.0 (resp. 1.2) par rapport à MCS. Le nombre de défauts de cache dans les sections critiques est divisé par 2.9 (resp. 2.3) par RCL, ce qui montre que RCL peut beaucoup améliorer la localité. Les verrous à combinateur n'ont pas été évalués dans cette expérience car ils ne permettent pas l'utilisation de variables de condition, qui sont utilisées par Memcached.
- Pour Berkeley DB avec le benchmark TPC-C, lors de l'utilisation de requêtes de type Stock Level, sur Magnycours-48 (resp. Niagara2-128), utiliser RCL multiplie la bande passante par 11.6 (resp. 7.6) par rapport aux verrous de base de Berkeley DB à 48 (resp. 384) clients. RCL résiste mieux à la charge que les autres verrous quand le nombre de clients simultanés augmente. En particulier, RCL est beaucoup plus efficace que les autres verrous quand l'application utilise plus de fils d'exécution client que la machine sous-jacente ne supporte de fils d'exécution matériels, et ce, même lorsque les autres verrous sont modifiés pour relâcher le processeur dans les boucles d'attente active.

La suite de cette annexe est organisée de la manière suivante. La Section A.2 présente RCL, le profiler qui permet d'identifier quelles applications et quels verrous peuvent être rendus plus performants en utilisant RCL, ainsi que l'outil permettant de transformer automatiquement les applications afin qu'elles puissent être utilisées avec RCL. La Section A.3 évalue la performance de RCL sur divers benchmarks. Finalement, la Section A.4 conclut.

A.2 Contribution

Ce chapitre présente les contributions principales des travaux de recherche présentés dans ce document. La Section A.2.1 présente RCL, et la Section A.2.2 présente le profiler et l'outil de transformation qui ont été brièvement décrits dans l'introduction.

A.2.1 Algorithme de RCL

Avec RCL, chaque section critique est remplacée par des appels de procédure distants à une fonction qui exécute son code. Afin d'implémenter l'appel de fonction distant, chaque serveur dispose d'un tableau de boîtes aux lettres (Figure A.1) qui est utilisé pour la communication avec les fils d'exécution clients. Ce tableau est long de $C \cdot L$ octets, où C est une constante qui représente le nombre maximum de clients (un grand nombre, typiquement bien supérieur au nombre de fils d'exécution matériels), et L est la taille des lignes de cache de l'architecture. Chaque boîte aux lettres req_i fait L octets et permet la communication entre un client c_i et le serveur. Le tableau est aligné de manière à ce que chaque boîte aux lettres req_i soit positionnée sur une seule ligne de cache. Les trois premiers mots machine de chaque requête req_i contiennent respectivement : (i) l'adresse du verrou associé à la section critique, (ii) l'adresse d'une structure qui encapsule le *contexte*, c'est-à-dire les variables référencées ou mises à jour par la section critique qui sont déclarées dans la fonction la contenant, et enfin (iii) l'adresse d'une fonction qui

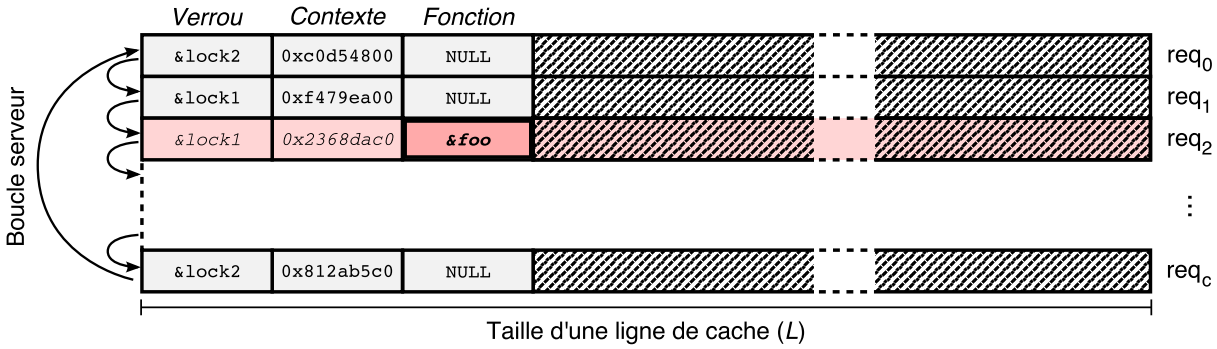


Figure A.1: Le tableau de boîtes aux lettres

encapsule la section critique pour laquelle le client c_i a demandé l'exécution, ou `NULL` si aucune exécution de section critique n'est demandée.

Côté client. Afin d'exécuter une section critique, un client c_i écrit d'abord l'adresse du verrou dans le premier mot de la structure req_i, puis il écrit l'adresse de la structure de contexte dans le deuxième mot, et pour finir, il écrit l'adresse de la fonction qui encapsule le code de la section critique dans le troisième mot. Le client attend ensuite que le troisième mot de req_i soit remis à `NULL` (attente active), ce qui indique que le serveur a exécuté la section critique.

Côté serveur. Un fil d'exécution de service itère sur les boîtes aux lettres, en attendant que l'une des requêtes ait une valeur qui ne soit pas égale à `NULL` dans son troisième mot. Lorsqu'une telle valeur est trouvée, le fil d'exécution de service vérifie que le verrou requis est libre, et, si c'est le cas, il l'acquiert et exécute la section critique en utilisant le pointeur de fonction et le contexte. Lorsque le fil d'exécution de service a terminé d'exécuter la section critique, en utilisant le pointeur de fonction, il réinitialise le troisième mot à `NULL`, et recommence à itérer sur le tableau de boîtes aux lettres.

Algorithme complet. L'algorithme présenté ci-dessus est une version simplifiée de RCL qui n'est efficace si les sections critiques ne bloquent jamais (pour des I/O par exemple), ou n'utilisent pas de synchronisation ad hoc (attente active). En pratique, l'environnement d'exécution RCL gère ces cas grâce à un pool de fils d'exécution de service. Il permet également l'utilisation de variables de condition.

A.2.2 Outils

Profiler. Afin de décider quels verrous transformer en RCL, un profiler a été implémenté sous la forme d'une bibliothèque qui intercepte les appels liés aux verrous, variables de condition, et fils d'exécution POSIX. Comme RCL améliore la performance des acquisitions de verrou sous haute contention ainsi que la localité des données, une application dont les performances peuvent être améliorées par RCL est soit fortement congestionnée au niveau de ses verrous, soit ses sections critiques ont une mauvaise localité mémoire. Par conséquent, le profiler mesure deux métriques: (i) le temps total passé en sections critiques, en incluant les acquisitions et les relâchements de verrous, ce qui permet de détecter les applications qui souffrent de forte contention au niveau de leurs verrous, et (ii) le nombre moyen de défauts de cache dans les sections critiques, ce qui permet de détecter les applications dont les sections critiques ont une mauvaise localité. Ces métriques permettent de détecter efficacement les sperformances de quels verrous peuvent être

améliorées par RCL. Le profiler peut également mesurer les deux métriques pour chaque verrou séparément.

Transformation. Si le profiler montre que les verrous d'une application peuvent être remplacés par RCL afin d'améliorer les performances, le développeur doit encapsuler toutes les sections critiques qui sont protégées par les verrous correspondants dans des fonctions qui sont passées au serveur RCL. Cette transformation de code correspond à une transformation appelée "Extract Method" [43]. Elle a été implémentée au cœur d'un outil de transformation automatique avec l'aide de Julia Lawall, en utilisant 2115 lignes de code Coccinelle [83].

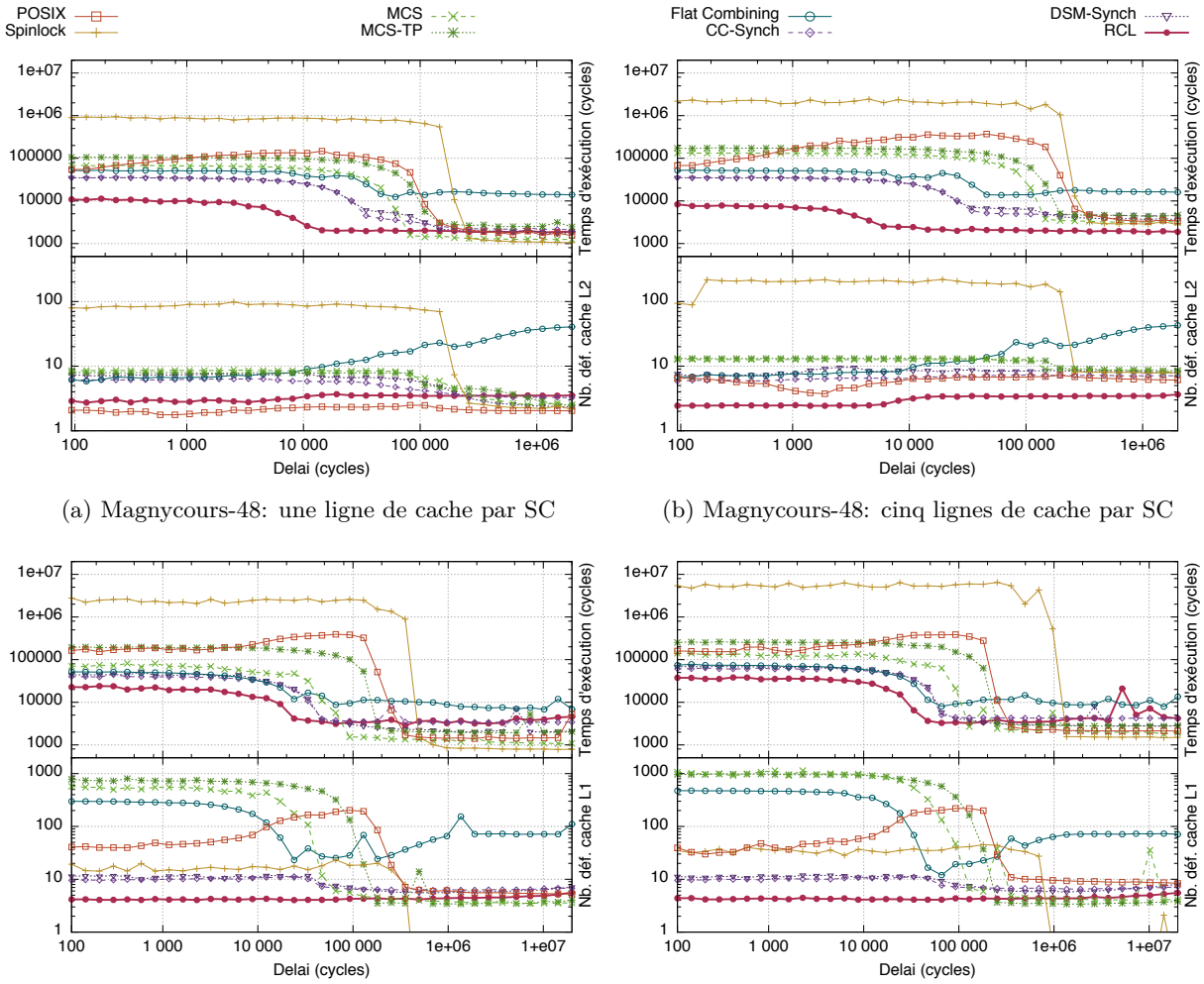
A.3 Évaluation

Ce chapitre décrit comment RCL peut être utilisé pour améliorer les performances des applications, et évalue les performances de RCL par rapport aux autres algorithmes de verrou présentés dans le Chapitre 3 sur une sélection d'applications. La Section A.3.2 présente un microbenchmark qui est utilisé pour comparer les performances de RCL avec celles d'autres algorithmes de verrou. La section A.3.2 présente une méthodologie qui permet aux développeurs de détecter quelles applications peuvent voir leur performance améliorée par RCL. Le profiler est ensuite lancé sur un ensemble d'applications dont le profiler a détecté qu'elles étaient potentiellement améliorables avec RCL.

A.3.1 Microbenchmark

Un microbenchmark a été développé pour mesurer la performance de RCL par rapport à d'autres algorithmes de verrou. Sept algorithmes de verrous ont été sélectionnés: un verrou à attente active, le verrou POSIX, MCS, MCS-TP, Flat Combining, CC-Synch et DSM-Synch. Les cinq derniers verrous sont connus pour offrir une bonne résistance en cas de forte contention. Le microbenchmark exécute des sections critiques de façon répétée sur tous les fils d'exécution matériel (un fil d'exécution logiciel fixé sur chaque fil d'exécution matériel). Afin de faire varier *degré de contention* sur le verrou, le microbenchmark fait varier le temps d'attente entre le moment où une section critique a fini d'être exécutée par un fil d'exécution et le début de la prochaine demande de prise de verrou par ce fil d'exécution: plus le délai est court, plus la contention est élevée. Le microbenchmark fait varier la *localité* des sections critiques en les faisant accéder à une ou cinq lignes de cache en lecture et en écriture. La Figure A.2 présente les résultats du microbenchmark.

Magnycours-48. La Figure A.2a présente le temps d'exécution moyen d'une section critique (en haut) et le nombre de cache misses (en bas) lorsque chaque fil d'exécution exécute 10,000 sections critiques qui accèdent chacune à une ligne de cache sur Magnycours-48. Le délai entre les tentatives d'exécution des sections critiques varie en abscisse. Cette expérience mesure principalement l'effet de la contention lors de l'accès aux verrous. La Figure 5.2b présente l'augmentation du temps d'exécution lorsque chaque section critique accède à cinq lignes de cache au lieu d'une. Cette expérience se concentre davantage sur l'effet de la localité des données des lignes de cache partagées. A haute contention (bas délai, à gauche sur les graphes), avec cinq accès, RCL est plusieurs fois plus rapide que tous les autres verrous. CC-Synch et DSM-Synch sont ~323% plus lents que RCL. Les deux algorithmes ont une performance comparable, même si Magnycours-48 est une machine NUMA. Flat Combining, sur lesquels ces algorithmes sont basés, est 49% plus lent qu'eux. MCS est beaucoup plus lent que les verrous à combinateur: il est ~277%



(a) Magnycours-48: une ligne de cache par SC

(b) Magnycours-48: cinq lignes de cache par SC

(c) Niagara2-128: une ligne de cache par SC

(d) Niagara2-128: cinq lignes de cache par SC

Figure A.2: Résultats du microbenchmark

plus lent que CC-Synch et DSM-Synch, et 153% plus lent que Flat Combining. MCS-TP est une variante de MCS qui résiste mieux à la préemption, cependant, ce verrou est 30% plus lent que MCS dans ce microbenchmark où la préemption est impossible (autant de fils d'exécution logiciels que de fils d'exécution matériels, threads fixés sur des cœurs). La performance du verrou POSIX est assez bonne à très haute contention (entre celle de Flat Combining et celle de MCS), mais sa performance diminue à contention moyenne: il devient pire que MCS et MCS-TP. Finalement, la performance du verrou à attente active (noté Spinlock) à haute contention est très mauvaise car l'attente active par de nombreux fils d'exécution sur un seul emplacement mémoire sature le bus de messages du protocole de cohérence de cache. À basse contention, pour un accès, tous les verrous se comportent de manière équivalente, sauf Flat Combining qui est plus lent d'un ordre de grandeur. Avec 5 accès, la performance de tous les verrous est dégradée à cause des défauts de cache supplémentaires, sauf pour RCL: dans ce cas, les données restent dans les caches du cœur serveur, et les défauts de cache sont évités.

Niagara2-128. Les Figures A.2c et A.2d présentent les résultats du microbenchmark sur Niagara2-128. Bien que l'architecture de Niagara2-128 soit différente (SPARC a lieu de x86 pour

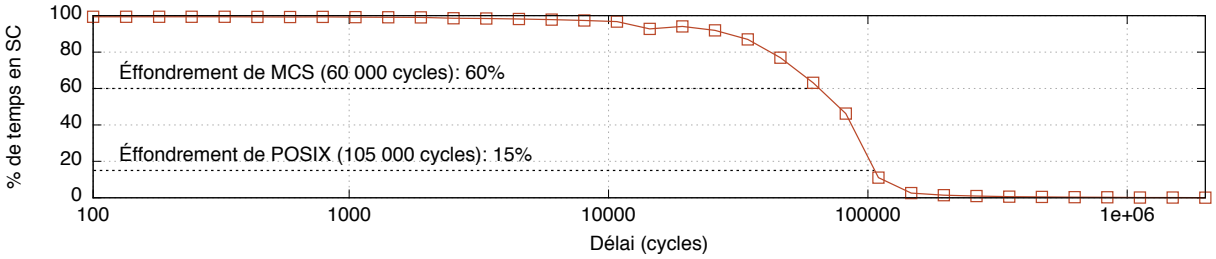
Magnycours-48, 8 fils d'exécution matériel par cœur au lieu d'un seul pour Magnycours-48), les résultats sont quantitativement similaires.

A.3.2 Applications

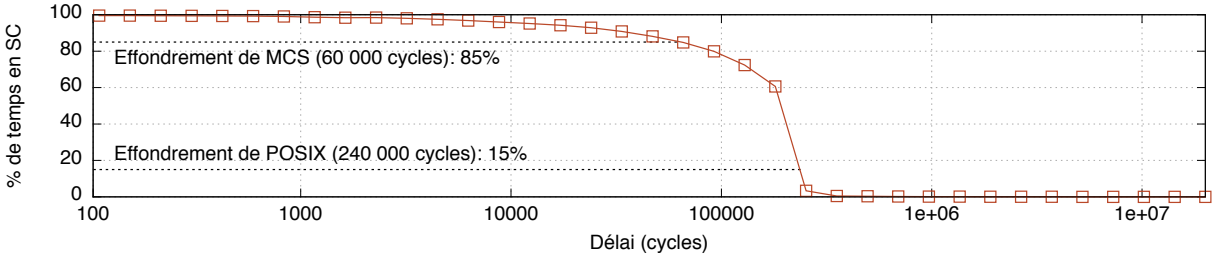
Profiling. Le profiler présenté dans la Section A.3.1 permet de prédire efficacement quels verrous peuvent bénéficier de meilleures performances grâce à RCL. Pour ce faire, le temps passé en section critique et le temps d'exécution des sections critiques, deux métriques mesurées respectivement par le profiler et par le microbenchmark, sont corrélées. La Figure A.3 montre les résultats obtenus lorsque microbenchmark avec le verrou POSIX et un accès mémoire est lancé avec le profiler: le pourcentage de temps passé en section critique est tracé en fonction du délai. D'après la Figure A.2a, sur Magnycours-48, le verrou POSIX s'effondre lorsque le délai est inférieur à $60,000$ cycles. D'après la Figure A.3a, il peut être déduit de cette information que le verrou POSIX s'effondre lorsque le microbenchmark passe 15% de son temps ou plus en section critique (seuil inférieur), et que RCL est plus performant que tous les autres verrous lorsque le microbenchmark passe 60% de son temps en section critique (seuil supérieur). Ces résultats sont préservés, ou améliorés, lorsque le nombre d'accès mémoire augmentent, car le temps d'exécution augmente au moins autant, et souvent davantage, pour pour les autres algorithmes que RCL. Par conséquent, deux seuils pour les applications évaluées dans cette section peuvent être identifiés sur Magnycours-48: en supposant qu'elles utilisent des verrous POSIX, si les applications passent plus de 15% en section critique, utiliser RCL est susceptible d'améliorer leur performance, mais pas nécessairement davantage que d'autres verrous que le verrou POSIX (seuil inférieur). Si elles passent plus de 60% de leur temps en section critique, utiliser RCL améliorera les performances davantage que d'utiliser n'importe quel autre verrou (seuil supérieur). Une évaluation similaire sur Niagara2-128 (voir Figure A.3b), indique que sur cette machine, le seuil inférieur est de 10% et le seuil supérieur est de 85%.

Performance applicative. Les deux métriques que mesurent le profiler, c'est-à-dire le temps passé en section critique et le nombre de défauts de cache, ne permettent bien sûr pas de déterminer avec certitude si une application verra ses performances améliorées grâce à l'utilisation de RCL. De nombreux autres facteurs (longueur des sections critiques, interactions entre verrous, etc.) entrent en compte lors de l'exécution des sections critiques. Cependant, comme expliqué ci-dessous, utiliser le temps passé en section critique en tant que métrique principale et le nombre de défauts de cache dans les sections critiques en tant que métrique secondaire est une technique efficace. Le temps passé en section critique est un bon indicateur de la congestion des verrous, et le nombre de défauts de cache permet d'estimer la localité des données dans les sections critiques. Les performances de RCL sont estimées sur les applications suivantes:

- Les 9 application de deuxième version de la suite de benchmarks Stanford ParalleL Applications for SHared memory, connue sous le nom de SPLASH-2 [107, 99, 110]. Il s'agit d'applications scientifiques parallèles. L'application Raytrace est proposée avec deux fichiers d'entrée possible, par conséquent, elle donne lieu à deux expériences.
- Les 7 applications de la suite de benchmarks Phoenix 2.0.0 [101, 103, 112, 92], également développée à Stanford. Ces applications implémentent des usages types du modèle de programmation MapReduce [31] proposé par Google.
- Memcached 1.4.6 [26, 41], un système de cache distribué utilisé par des sites internet tels que YouTube, Wikipedia ou Reddit. Memcached est lancé sur la moitié des fils d'exécution



(a) Temps passé en section critique sur Magnycours-48



(b) Temps passé en section critique sur Niagara2-128

		# cycles	% en CS
Magnycours-48	**Seuil supérieur**	60,000	60%
	Seuil inférieur	105,000	15%
Niagara2-128	**Seuil supérieur**	60,000	85%
	Seuil inférieur	240,000	15%

(c) Seuils

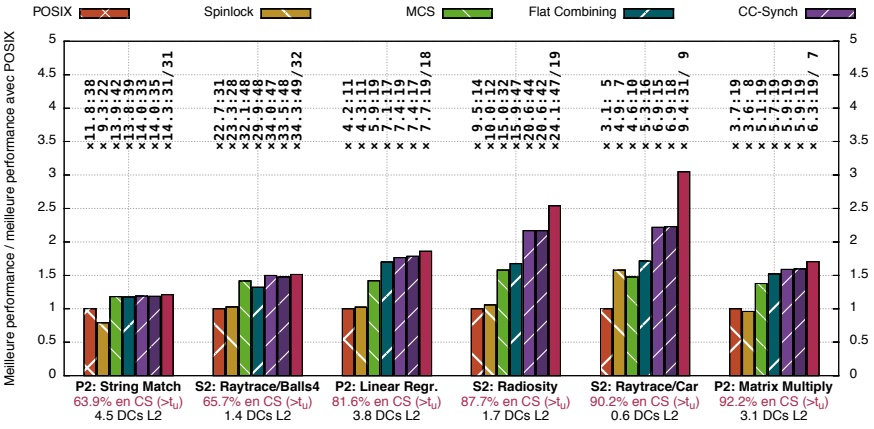
Figure A.3: Temps passé en section critique dans le microbenchmark et seuils

matériels de chaque machine, l'autre moitié étant dédié au client Memslap de Libmemcached 1.0.2 [28] qui simule des clients demandant à Memcached l'exécution de requêtes. Deux expériences sont utilisées: dans l'une, seules des requêtes Get (lectures du cache) sont exécutées, dans l'autre, seules des requêtes Set (écriture dans le cache) sont exécutées.

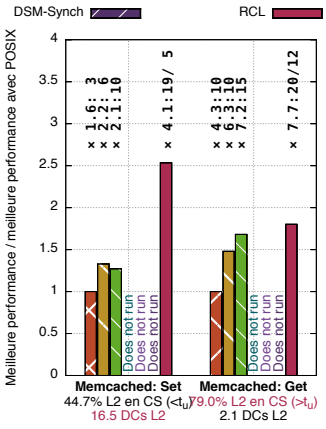
- Berkeley DB 5.2.28 [80, 79], un Système de Gestion de Bases de Données (SGBD) maintenu par Oracle, avec un benchmark TPC-C [67] nommé TpccOverBkDb, qui a été écrit par Alexandra Fedorova et Justin Fuston à l'université Simon Fraser. Cinq expériences sont utilisés, une pour chaque type de requêtes offert par TPC-C. Le benchmark est une application qui crée un fil d'exécution pour chaque client. Berkeley DB est implémenté sous la forme d'une bibliothèque, et tous les fils d'exécution appellent des fonctions offertes par cette bibliothèque.

Chacune des expériences ci-dessus sont lancées une première fois avec le profiler, afin de savoir si le temps qu'elles passent en section critique est supérieur à l'un des seuils. Si c'est le cas, ses performances sont présentées dans la Figure A.4. Cette figure rappelle également certaines données du profiler, et le pourcentage de temps en section critique est situé par rapport aux seuils inférieur et supérieur (notés t_l et t_u) correspondant à chaque expérience. Les paramètres utilisés pour les expériences sont détaillés dans la Figure A.4c.

SPLASH-2 et Phoenix. Pour SPLASH-2 et Phoenix, toutes les expériences ont un nombre de défauts de cache faible (inférieur à 5), par conséquent, cette métrique est ignorée: les sections



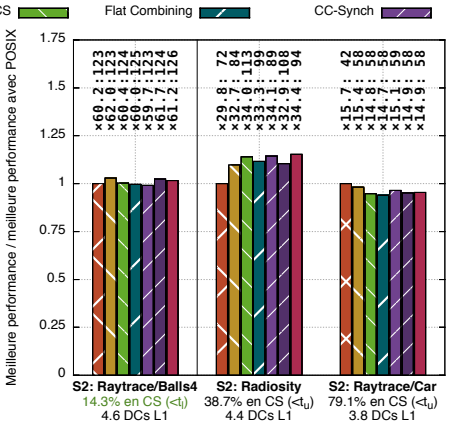
(a) Magnycours-48: SPLASH-2 et Phoenix 2



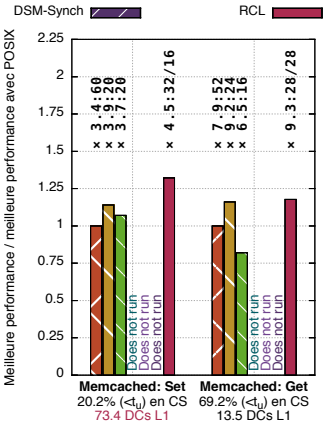
(b) Magnycours-48: Memcached

SPLASH-2	**Radiosity**	`-batch -ae 500 -bf 0.005`
	Raytrace/Car	Fichiers d'entrée: `car.env/geo`
	Raytrace/Balls4	Fichiers d'entrée: `balls4.env/geo`
Phoenix 2	**Linear Regression**	Jeu de données moyen: Fichier de 100Mo
	String Match	Jeu de données moyen: Fichiers de 100Mo
	Matrix Multiply	Jeu de données moyen: Matrice 500x500
Memcached	**Get**	30,000 entrées (init.) 30,000 requêtes Get
	Set	10,000 entrées (init.) 10,000 requêtes Set
Berkeley DB + TPC-C	**Order Status**	300 requests per client
	Stock Level	300 requests per client

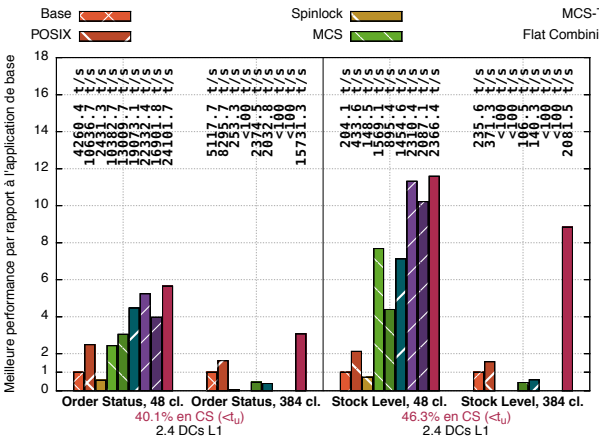
(c) Paramètres des benchmarks



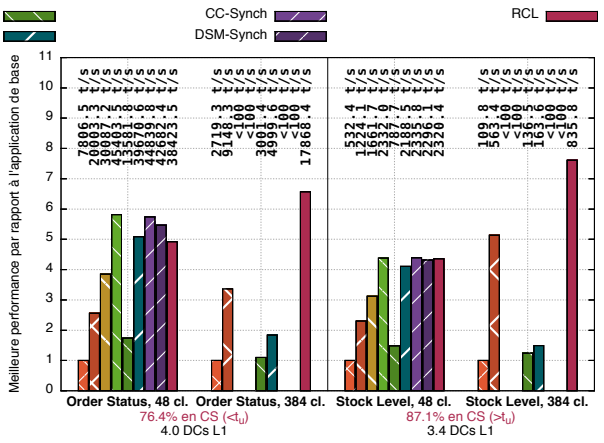
(d) Niagara2-128: SPLASH-2



(e) Niagara2-128: Memcached



(f) Magnycours-48: Berkeley DB



(g) Niagara2-128: Berkeley DB

Note: dans cette figure, t_l et t_u représentent respectivement le seuil inférieur et supérieur correspondant à chaque l'expérience.

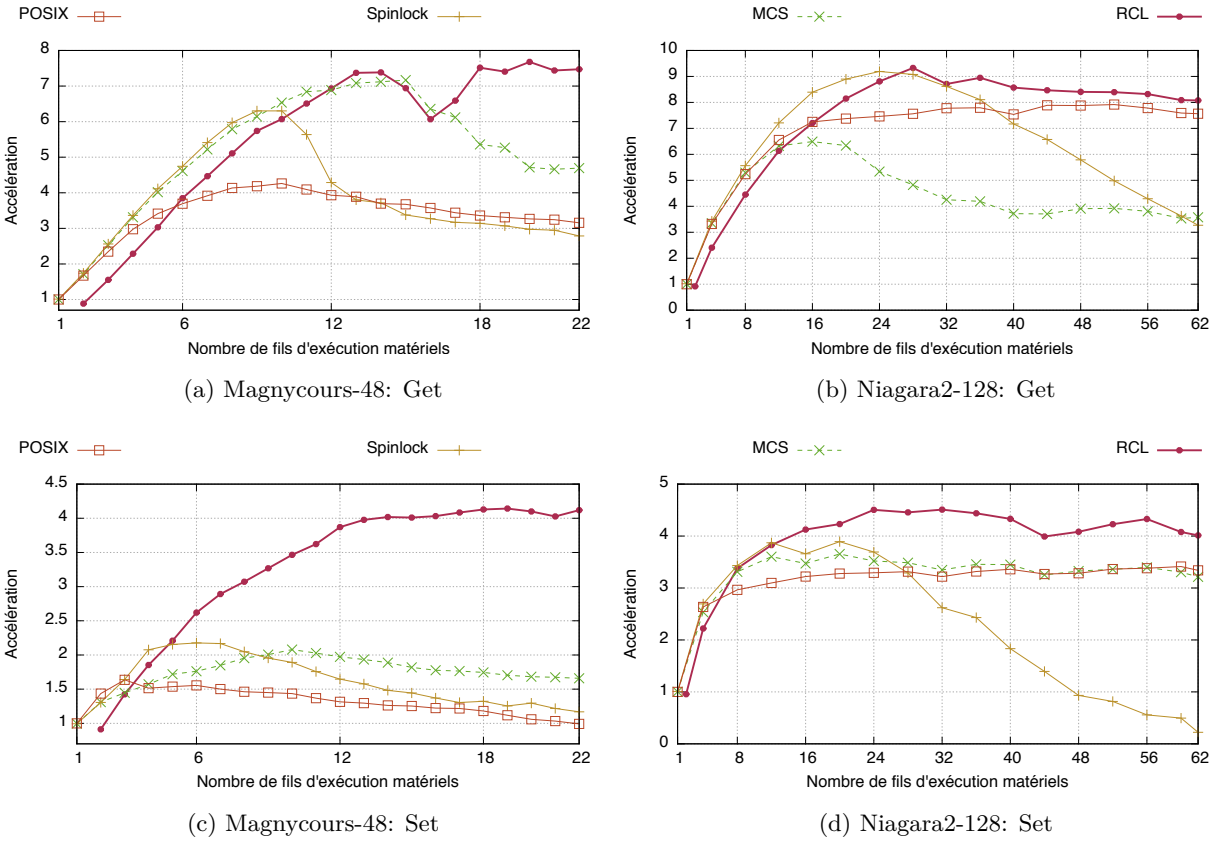
Figure A.4: Performance des différents verrous dans les applications

critiques ont toutes une bonne localité. Les Figures A.4a et A.4d donnent les résultats pour toutes les expériences des suites de benchmarks SPLASH-2 et Phoenix dont le temps passé en section critique est supérieur à l'un des seuils, c'est-à-dire les expériences dans lesquelles RCL peut améliorer les performances par rapport au verrou POSIX car leur contention est élevée.

Toutes les expériences satisfaisant ces critères passent un temps en section critique supérieur au seuil haut sur Magnycours-48 (Figure A.4a), alors que toutes les expériences satisfaisant ces critères passent un temps en section critique situé entre les deux seuils pour Niagara2-128 (Figure A.4d). Les résultats pour Raytrace/Balls4 (SPLASH-2) sont également présentés pour Niagara2-128 à titre d'exemple, même si le temps passé en section critique par cette expérience est en dessous des deux seuils.

Dans toutes les expériences de SPLASH-2 et Phoenix sur Magnycours-48 (voir Figure A.4a), RCL est plus performant que les autres verrous. Les verrous peuvent être classés en trois catégories : (i) POSIX et le verrou à attente active offrent des performances faibles, (ii) MCS et Flat Combining offrent des performances moyennes, (iii) CC-Synch et DSM-Synch offrent de bonnes performances, et (iv) RCL offre de très bonnes performances. Plus l'expérience passe de temps en section critique, meilleure est l'amélioration des performances offerte par les autres verrous que POSIX, et ce d'autant plus que le verrou est rapide. Par conséquent, plus le temps passé en section critique est élevé, plus RCL permet de gagner en performance. La seule expérience pour laquelle ce constat n'est pas vrai est Matrix Multiply (Phoenix 2), pour laquelle les autres verrous que POSIX n'améliorent que peu les performances malgré le temps important passé en section critique. Cela est dû au fait que Matrix Multiply a d'autres goulots d'étranglement que les verrous: RCL permet de faire chuter le temps passé en section critique à moins de 1%, ce qui montre bien que les verrous ne sont plus un goulot d'étranglement après transformation. Sur Niagara2-128 (voir Figure A.4d), pour Raytrace/Balls4 (SPLASH-2), changer de verrou n'améliore pas les performances, comme prévu, car le temps passé en section critique est inférieur aux deux seuils, par conséquent, la congestion sur les verrous n'est pas un goulot d'étranglement. Sur Radiosity (SPLASH-2), comme prévu, RCL améliore les performance mais pas significativement plus que les autres verrous, puisque le temps passé en section critique se trouve entre les deux seuils. Pour Raytrace/Car (SPLASH-2) changer de verrou n'améliore pas les performances. Encore une fois, une analyse plus poussée montre que d'autres goulots d'étranglement sont à l'œuvre. Afin de vérifier que le profiler ne donne pas de faux négatifs, les performances des expériences pour lesquelles le temps passé en section critique est inférieur aux deux seuils a été mesuré, et dans ce cas, changer de verrou n'altère jamais les performances significativement. Par conséquent, sur 16 expériences des suites SPLASH-2 et Phoenix sur chacune des machines (32 au total), le profiler n'a pas su prévoir le gain potentiel pour seulement deux d'entre elles: Matrix Multiply (Phoenix) sur Magnycours-48, et Raytrace/Car (SPLASH-2) sur Niagara2-128. Cependant, pour Matrix Multiply, même si le gain est plus faible que ce qui aurait pu être espéré au vu des autres expériences, le profiler a prévu avec succès que RCL serait plus performant que tous les autres verrous (temps en section critique supérieur au seuil haut).

Memcached. Pour Memcached, les verrous à combinateur (Flat Combining, CC-Synch et DSM-Synch) ne sont pas évalués car ils ne proposent pas d'implémentation pour les variables de condition alors que celles-ci sont utilisées par l'application. Sur Magnycours-48 (voir Figure A.4b), l'expérience Get passe plus de temps en section critique que le seuil supérieur. Comme prévu, dans cette expérience, RCL permet de gagner plus que les autres verrous. Même si l'expérience Set passe moins de temps en section critique que le seuil, elle génère un nombre important de défauts de cache en section critique (16.5). C'est pourquoi, pour cette expérience, RCL améliore très fortement les performances (plus que dans l'expérience Get), même si le temps passé en section critique se situe entre les deux seuils. RCL améliore fortement la localité en divisant le nombre de défauts de cache dans les sections critiques par 2.9. Les résultats sur Niagara2-128



(a) Magnycours-48: Get

(b) Niagara2-128: Get

(c) Magnycours-48: Set

(d) Niagara2-128: Set

Figure A.5: Memcached speedup

(voir Figure A.4e), sont quantitativement similaires à ceux de Magnycours-48. Le nombre de défauts de cache en section critique (73.4) est divisé par 2.3 par RCL.

La Figure A.5 présente les résultats détaillés pour Memcached, c'est-à-dire son accélération par rapport à la version de base de l'application lorsque le nombre de fils d'exécution matériels varie. Sur Magnycours-48, pour Memcached/Get et pour Memcached/Set, RCL n'améliore pas seulement les performances, il permet aussi à l'application de mieux passer à l'échelle. Pour Memcached/Get sur Magnycours-48 (voir Figure A.5a), les performances du verrou POSIX ainsi que celles du verrou à attente active s'effondrent à partir de 11 fils d'exécution et les performances de MCS s'effondrent à partir de 16 fils d'exécution. Les performances de RCL, en revanche, atteignent un plateau à partir de 18 fils d'exécution. Utiliser RCL est initialement plus lent qu'utiliser les autres verrous à nombre de fils d'exécution matériels égaux, à cause du fait que RCL perd un fil d'exécution pour l'exécution de son serveur. Cependant, malgré cet handicap initial, RCL permet d'atteindre la performance du verrou POSIX, du verrou à attente active et de MCS à partir de seulement, 6, 11 ou 12 fils d'exécution matériels, respectivement. En ce qui concerne Memcached/Set sur Magnycours-48 (voir Figure A.5b), les performances du verrou POSIX, du verrou à attente active, ainsi que de MCS commencent à s'effondrer à partir de 4, 8 ou 11 fils d'exécution matériels, respectivement. RCL, quant à lui, atteint un plateau à partir de 14 fils d'exécution matériels. Dans cette expérience, RCL est plus performant que tous les autres verrous à partir de seulement cinq fils d'exécution matériels. Sur Niagara2-128 (voir Figures A.5b, et A.5d), les gains de performance sont moindres, mais RCL offre néanmoins

toujours la meilleure performance de pointe, et a tendance à mieux passer à l'échelle que les autres verrous.

Berkeley DB avec TpccOverBkDb. Pour Berkeley DB, seules les expériences avec les requêtes de type Order Status et Stock Level passent un temps en section critique qui n'est pas inférieur aux deux seuils. Les expériences Order Status et Stock Level passent un temps en section critique qui se trouve soit entre les deux seuils, soit au dessus des deux seuils (voir Figures A.4f et A.4g). Cependant, les mesures de temps en section critique sont sous-estimées pour Berkeley DB, car cette application utilise des verrous (notés « Original » sur les figures) qui utilisent de l'attente active avant de prendre un verrou POSIX, et seul le temps d'acquisition du verrou POSIX est compté par le profiler, car celui-ci est spécialisé pour les verrous POSIX.

Comme le montrent les Figures A.4f et A.4g, RCL permet d'améliorer les performances au moins autant que tous les autres verrous. Dans ce benchmark, lorsque le nombre de clients de la base de données est important, il peut y avoir plus de fils d'exécution clients qu'il n'y a de fils d'exécution matériels dans la machine. Lorsque cela arrive (384 clients sur les histogrammes), le verrou à attente active, MCS, CC-Synch et DSM-Synch s'effondrent à cause d'un phénomène appelé convoi [64]: le fil d'exécution qui possède le verrou se fait préempter, et l'ordonnanceur réveille les fils d'exécution dans un ordre tel qu'il faut au minimum un quantum de temps entier de l'ordonnanceur pour passer d'une prise de verrou à l'autre. Lorsque ce phénomène se produit, les temps d'exécution sont si longs que les barres des histogrammes ne sont pas visibles sur les figures. MCS-TP est un verrou basé sur MCS qui est spécialement conçu pour résister à ce phénomène, mais ses performances sont faibles. RCL est très efficace lorsque le nombre de clients est élevé, car les sections critiques ne se font jamais préempter, grâce au fait qu'elles sont exécutées sur un cœur serveur dédié.

A.4 Conclusion

RCL est une nouvelle technique de verrouillage qui permet de réduire le temps de prise de verrou et d'accélérer l'exécution des sections critique en améliorant leur localité. L'idée principale de RCL est de migrer l'exécution de sections critiques vers un ou plusieurs fils d'exécution matériels serveurs. RCL a été implémenté pour Linux et pour Solaris, et supporte les architectures x86 et SPARC. Un profiler permet de détecter les applications patrimoniales qui peuvent bénéficier de RCL. Avec l'aide de Julia Lawall, un outil a été écrit pour transformer automatiquement les applications patrimoniales de manière à ce qu'elles utilisent RCL. L'évaluation de RCL, qui porte sur de nombreux benchmarks, montre que RCL permet un meilleur gain en performance que de nombreux autres verrous qui ont pourtant pour objectif de bien résister à la contention.

Perspectives. Une amélioration possible pour RCL serait la mise au point d'un environnement d'exécution RCL adaptatif qui (i) passe automatiquement d'un verrou POSIX à un RCL lorsque la contention sur le verrou augmente ou lorsque la localité de ses sections critiques diminue, et (ii) migre les verrous entre serveurs, afin de balancer la charge des serveurs dynamiquement. L'un des défis de l'écriture d'un tel environnement d'exécution est la mise au point de stratégies de profiling et de migration à faible coût.

Bibliography

[1] J. L. Abellán, J. Fernández, and M. E. Acacio. Glocks: efficient support for highly-contended locks in many-core CMPs. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '11, pages 893–905, Washington, DC, USA, 2011. IEEE Computer Society.

[2] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ISCA '89, pages 396–406, New York, NY, USA, 1989. ACM.

[3] H. Akkan, M. Lang, and L. Ionkov. HPC runtime support for fast and power efficient locking and synchronization. In *Proceedings of the 2013 IEEE International Conference on Cluster Computing*, CLUSTER '13, pages 1–7. IEEE, 2013.

[4] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[5] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan. 1990.

[6] G. R. Andrews. *Concurrent Programming: principles and Practice.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[7] M. Auslander, D. Edelsohn, O. Krieger, B. Rosenburg, and R. Wisniewski. Enhancement to the MCS lock for increased functionality and improved programmability. *U.S. patent application 10/128,745.* Oct. 2003.

[8] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for java. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 258–268, New York, NY, USA, 1998. ACM.

[9] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[10] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transansactions on Computer Systems*, 8(1):37–55, Feb. 1990.

[11] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. URPC: a toolkit for prototyping remote procedure calls. *The Computer Journal*, 39, no. 6:525–540, June 1996.

[12] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein. Optimal strategies for spinning and blocking. *Journal of Parallel and Distributed Computing*, 21(2):246–254, May 1994.

[13] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[14] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '10, Vancouver, Canada, 2010. USENIX Association.

[15] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the 13th Ottawa Linux Symposium*, OLS '13, Ottawa, Canada, July 2012.

[16] B. B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *Proceedings of the 2013 IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '13, pages 141–152, Washington, DC, USA, 2013. IEEE Computer Society.

[17] A. Brodsky, F. Ellen, and P. Woelfel. Fully-adaptive algorithms for long-lived renaming. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC '06, pages 413–427, Berlin, Heidelberg, 2006. Springer-Verlag.

[18] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, Aug. 2000.

[19] A. Burns and A. J. Wellings. Locking policies for multiprocessor ada. *Ada Letters*, 33(2):59–65, Nov. 2013.

[20] A. Burns and A. J. Wellings. A schedulability compatible multiprocessor resource sharing protocol – MrsP. In *Proceedings of the 2013 IEEE Euromicro Conference on Real-Time Systems*, ECRTS '13, pages 282–291, Washington, DC, USA, 2013. IEEE Computer Society.

[21] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. Marathe, and M. Moir. Message passing or shared memory: evaluating the delegation abstraction for multicores. In *Proceedings of the 17th International Conference on Principles of Distributed Systems*, OPODIS '13, pages 83–97, Nice, France, 2013. Springer International Publishing.

[22] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, Nov. 1994.

[23] J. Cleary, O. Callanan, M. Purcell, and D. Gregg. Fast asymmetric thread synchronization. *ACM Transactions on Architecture and Code Optimization*, 9(4):27:1–27:22, Jan. 2013.

[24] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30(2):16–29, Mar. 2010.

[25] T. S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Department of Computer Science, University of Washington, Feb. 2003.

[26] Danga Interactive. Memcached: distributed memory object caching system. `http://memcached.org`.

[27] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: a holistic approach to memory placement on NUMA systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394, New York, NY, USA, 2013. ACM.

[28] Data Differential. Libmemcached. `https://launchpad.net/libmemcached`.

[29] F. David, G. Thomas, L. Lawall, Julia, and G. Muller. Continuously measuring critical section pressure with the free lunch profiler. Research Report RR-8486, INRIA, Mar. 2014.

[30] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, New York, NY, USA, 2013. ACM.

[31] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communication of the ACM*, 51(1):107–113, Jan. 2008.

[32] D. Dice. Polite busy-waiting with wrpause on sparc. `https://blogs.oracle.com/dave/entry/polite_busy_waiting_with_wrpause`, Oct. 2012.

[33] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining NUMA locks. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 65–74, New York, NY, USA, 2011. ACM.

[34] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: a general technique for designing NUMA locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 247–256, New York, NY, USA, 2012. ACM.

[35] E. W. Dijkstra. Cooperating sequential processes. Published as EWD:EWD123pub, Sept. 1965.

[36] G. Drescher, T. Hönig, S. Maier, B. Oechslein, and W. Schröder-Preikschat. A Scalability-Aware Kernel Executive for Many-Core Operating Systems. In *Proceedings of the 1st Workshop on Runtime and Operating Systems for the Many-core Era*, WROSME '13, pages 1–10, Aachen, Germany, 2013.

[37] J. Eastep, D. Wingate, M. D. Santambrogio, and A. Agarwal. Smartlocks: lock acquisition scheduling for self-aware synchronization. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 215–224, New York, NY, USA, 2010. ACM.

[38] P. Fatourou and N. D. Kallimanis. Sim: a highly-efficient wait-free universal construction. `https://code.google.com/p/sim-universal-construction/`.

[39] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, New York, NY, USA, 2012. ACM.

[40] F. Fich, D. Hendler, and N. Shavit. On the inherent weakness of conditional synchronization primitives. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 80–87, New York, NY, USA, 2004. ACM.

[41] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5–, Aug. 2004.

[42] B. Ford and J. Lepreau. Evolving mach 3.0 to a migrating thread model. In *Proceedings of the USENIX Winter 1994 Technical Conference*, WTEC'94, pages 9–9, Berkeley, CA, USA, 1994. USENIX Association.

[43] M. Fowler. *Refactoring: improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[44] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 229–240, New York, NY, USA, 2013. ACM.

[45] T. Harris and K. Fraser. Language support for lightweight transactions. *SIGPLAN Notices*, 38(11):388–402, Oct. 2003.

[46] T. Harris, M. Herlihy, Y. Lev, Y. Liu, V. Luchangco, V. Marathe, and M. Moir. Towards whatever-scale abstractions for data-driven parallelism. In *Proceedings of the 1st International Workshop on Rack Scale Computing*, WRSC '14, 2014.

[47] A. Hassan, R. Palmieri, and B. Ravindran. Remote invalidation: optimizing the critical path of memory transactions. In *Proceedings of the 2014 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '14, Phoenix, AZ, USA, 2014. IEEE Computer Society.

[48] B. He, W. N. Scherer III, and M. L. Scott. Time-published queue-based spin locks. `http://www.cs.rochester.edu/research/synchronization/pseudocode/tp_locks.html`.

[49] B. He, W. N. Scherer III, and M. L. Scott. Preemption adaptivity in time-published queue-based spin locks. In *Proceedings of the 11th International Conference on High Performance Computing*, HiPC'05, pages 7–18, 2005.

[50] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. `http://mcg.cs.tau.ac.il/projects/flat-combining`.

[51] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[52] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–529, Washington, DC, USA, 2003. IEEE Computer Society.

[53] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[54] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[55] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[56] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th Symposium on Principles of Distributed Computing*, PODC '88, pages 276–290, New York, NY, USA, 1988. ACM.

[57] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, Oct. 1974.

[58] Innovative Computing Laboratory. Performance Application Programming Interface (PAPI). `http://icl.cs.utk.edu/papi/`.

[59] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling contention management from scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 117–128, New York, NY, USA, 2010. ACM.

[60] H. Kang and J. L. Wong. To hardware prefetch or not to prefetch?: a virtualized environment study and core binding approach. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 357–368, New York, NY, USA, 2013. ACM.

[61] T. Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 105–112, New York, NY, USA, 1986. ACM.

[62] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 223–234, New York, NY, USA, 2011. ACM.

[63] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 141–150, New York, NY, USA, 2012. ACM.

[64] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 125–138, New York, NY, USA, 2010. ACM.

[65] R. Lachaize, B. Lepers, and V. Quéma. Memprof: a memory profiler for NUMA multicore systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC '12, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association.

[66] A. L. Leiner. System specifications for the DYSEAC. *Journal of the ACM*, 1(2):57–81, Apr. 1954.

[67] S. T. Leutenegger and D. Dias. A modeling study of the TPC-C benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 22–31, New York, NY, USA, 1993. ACM.

[68] T. Liu and E. D. Berger. Sheriff: precise detection and automatic mitigation of false sharing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 3–18, New York, NY, USA, 2011. ACM.

[69] J.-P. Lozi. Le Remote Core Lock (RCL) : une nouvelle technique de verrouillage pour les architectures multi-cœur. CFSE '8, 2011.

[70] J.-P. Lozi. PHP bug report #62064. `https://bugs.php.net/bug.php?id=62064`, May 2012.

[71] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote Core Locking: migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC '12, pages 65–76, Berkeley, CA, USA, 2012. USENIX Association.

[72] J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller. Efficient locking for multicore architectures. Research Report RR-7779, INRIA, Nov. 2011.

[73] V. Luchangco, D. Nussbaum, and N. Shavit. A hierarchical clh queue lock. In *Proceedings of the 12th International Conference on Parallel Processing*, Euro-Par'06, pages 801–810, Berlin, Heidelberg, 2006. Springer-Verlag.

[74] P. Magnussen, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, IPPS '94, pages 165–171, Cancun, Mexico, Apr. 1994. IEEE Computer Society Press.

[75] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[76] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 269–278, New York, NY, USA, 1991. ACM.

[77] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[78] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 221–234, New York, NY, USA, 2009. ACM.

[79] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, USENIX ATC '99, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[80] Oracle Corporation. Berkeley DB. `http://www.oracle.com/technetwork/database/b▷erkeleydb`.

[81] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, ICDCS'82, pages 22–30, 1982.

[82] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, PDSIA'99.

[83] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 3rd European Conference on Computer Systems 2008*, Eurosys '08, pages 247–260, New York, NY, USA, 2008. ACM.

[84] L. Papadopoulos, I. Walulya, P. Tsigas, D. Soudris, and B. Barry. Evaluation of message passing synchronization algorithms in embedded systems. In *Proceedings Of The 14th International Conference On Embedded Computer Systems: architectures, Modeling, And Simulation*, SAMOS '14, Samos, Greece, 2014.

[85] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, pages 348–354, New York, NY, USA, 1984. ACM.

[86] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: the Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, third edition, 2007.

[87] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 335–348, New York, NY, USA, 2010. ACM.

[88] D. Petrović, T. Ropars, and A. Schiper. Leveraging hardware message passing for efficient thread synchronization. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 143–154, New York, NY, USA, 2014. ACM.

[89] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Lock contention aware thread migrations. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 369–370, New York, NY, USA, 2014. ACM.

[90] Z. Radovic and E. Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 241–253, Washington, DC, USA, 2003. IEEE Computer Society.

[91] K. S. Ramesh. Design and development of MINIX distributed operating system. In *Proceedings of the 1988 ACM Sixteenth Annual Conference on Computer Science*, CSC '88, pages 685–685, New York, NY, USA, 1988. ACM.

[92] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[93] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: history, overview, and perspective. *Journal of Supercomputing*, 7(1-2):9–50, May 1993.

[94] S. Saha and J.-P. Lozi. EHCtor: detecting resource-release omission faults in error-handling code for systems software. CFSE '9, 2013.

[95] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller. Hector: detecting resource-release omission faults in error-handling code for systems software. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '13, pages 1–12, Washington, DC, USA, 2013. IEEE Computer Society.

[96] M. L. Scott and W. N. Scherer. Scalable queue-based spin locks with timeout. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 44–52, New York, NY, USA, 2001. ACM.

[97] C. Sharp and G. Morgan. Hugh: a semantically aware universal construction for transactional memory systems. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par '13, pages 470–481, Aachen, Germany, 2013. Springer-Verlag.

[98] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[99] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: stanford parallel applications for shared-memory. *SIGARCH Computer Architecture News*, 20(1):5–44, Mar. 1992.

[100] S. Sridharan, B. Keck, R. Murphy, S. Chandra, and P. Kogge. Thread migration to improve synchronization performance. In *In Proceedings of the 2nd Workshop on Operating System Interference in High Performance Applications*, OSIHPA '06, 2006.

[101] Stanford University. The Phoenix system for MapReduce programming. http://mapreduce.stanford.edu.

[102] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 253–264, New York, NY, USA, 2009. ACM.

[103] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular MapReduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.

[104] A. S. Tanenbaum. Distributed operating systems anno 1992. what have we learned so far? *Distributed Systems Engineering*, 1(1):3–10, 1993.

[105] The PHP Group. PHP: hypertext preprocessor. http://www.php.net.

[106] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 392–403, New York, NY, USA, 1995. ACM.

[107] University of Delaware. The modified SPLASH-2 home page. http://www.capsl.udel.edu/splash.

[108] D. Vyukov. Combiner/aggregator synchronization primitive. https://software.intel.com/en-us/blogs/2013/02/22/combineraggregator-synchronization-primitive, Feb. 2013.

[109] J.-T. Wamhoff, S. Diestelhorst, C. Fetzer, P. Marlier, P. Felber, and D. Dice. Selective core boosting: the return of the turbo button. Technical Report TUD-Fl13-02, Technische Universität Dresden, Nov. 2013.

[110] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[111] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[112] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: scalable MapReduce on a large-scale shared-memory system. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, IISWC '09, pages 198–207, Washington, DC, USA, 2009. IEEE Computer Society.

[113] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 181–192, New York, NY, USA, 2005. ACM.

[114] J. Zhou and B. Demsky. Memory management for many-core processors with software configurable locality policies. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 3–14, New York, NY, USA, 2012. ACM.

List of Illustrations

Figures

Algorithms

Listings