Lock – Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems

RACHID GUERRAOUI, EPFL, Switzerland
HUGO GUIROUX, Université Grenoble Alpes, CNRS, Grenoble INP, France
RENAUD LACHAIZE, Université Grenoble Alpes, CNRS, Grenoble INP, France
VIVIEN QUÉMA, Université Grenoble Alpes, CNRS, Grenoble INP, France
VASILEIOS TRIGONAKIS[*†], Oracle Labs, Switzerland

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

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

CCS Concepts: • **Software and its engineering → Mutual exclusion**;

[*]Project started while the author was at EPFL.
[†]Authors appear in alphabetical order.

Authors' addresses: Rachid Guerraoui, EPFL, DCL (Station 14), I&C, Lausanne, 1015, Switzerland, rachid.guerraoui@epfl.ch; Hugo Guiroux, Université Grenoble Alpes, CNRS, Grenoble INP, Bâtiment IMAG, 700, avenue centrale, Saint-Martin-d'Hères, 38401, France, hugo.guiroux@univ-grenoble-alpes.fr; Renaud Lachaize, Université Grenoble Alpes, CNRS, Grenoble INP, Bâtiment IMAG, 700, avenue centrale, Saint-Martin-d'Hères, 38401, France, renaud.lachaize@univ-grenoble-alpes.fr; Vivien Quéma, Université Grenoble Alpes, CNRS, Grenoble INP, Bâtiment IMAG, 700, avenue centrale, Saint-Martin-d'Hères, 38401, France, vivien.quema@univ-grenoble-alpes.fr; Vasileios Trigonakis, Oracle Labs, Prime Tower, Floor 17, Hardstrasse 201, Zurich, 8005, Switzerland, vasileios.trigonakis@oracle.com.

0734-2071/2018/11-ART $15.00
https://doi.org/0000001.0000001

Additional Key Words and Phrases: Multicore, synchronization, locks, performance bottleneck, lock interface

ACM Reference Format:
Rachid Guerraoui, Hugo Guiroux, Renaud Lachaize, Vivien Quéma, and Vasileios Trigonakis. 2018. Lock – Unlock: Is That All?, A Pragmatic Analysis of Locking in Software Systems. *ACM Trans. Comput. Syst.* 1, 1 (November 2018), 158 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Multicore machines are pervasive today but it is not always easy to leverage them. Many multi-threaded applications suffer from bottlenecks related to critical sections and their corresponding locks [5, 8, 9, 15, 25, 27, 32, 42, 52, 56, 60, 63, 67–70, 76, 83, 91]. Over the past 25 years, a plethora of optimized mutual exclusion (mutex) lock algorithms have been designed to mitigate these issues [8, 20, 21, 24, 27, 29–32, 36, 37, 39, 46, 47, 55, 59, 65, 66, 70, 74, 77, 79, 81, 98]. Application and library developers can choose from this large set of algorithms for implementing efficient synchronization in their software. However, there is currently no complete study to guide this puzzling choice for realistic applications.

In particular, the most recent and comprehensive empirical performance evaluation on multicore synchronization [27], due to its breadth (from hardware protocols to high-level data structures), only provides a partial coverage of locking algorithms. Indeed, the aforementioned study only considers nine algorithms, does not consider hybrid spinning/blocking waiting policies, omits emerging approaches (e.g., load-control mechanisms described in Section 2.1) and provides a modest coverage of hierarchical locks [20, 21, 32], a recent and efficient approach for NUMA architectures. Generally, most of the observations highlighted in the existing literature are based on microbenchmarks and only consider the lock/unlock interface, ignoring other lock-related operations such as condition variables and trylocks. Besides, in the case of papers that present a new lock algorithm, the empirical observations are often focused on the specific workload characteristics for which the actual lock was designed [52, 63], or mostly based on microbenchmarks [30, 32]. Finally, existing analyses focus on traditional performance metrics (mainly throughput) and do not cover other metrics, such as energy efficiency and tail latency, which are becoming increasingly important. In this paper, we perform a thorough and practical analysis of synchronization, with the goal of providing software developers with enough information to design fast, scalable and energy-efficient synchronization in their systems.

The first contribution of this paper is a broad performance study (Sections 5, 6 and 7) on Linux/x86 (i.e., the Linux operating system running on AMD/Intel x86 64-bit processors) of 28 state-of-the-art mutual exclusion lock algorithms on a set of 40 realistic and diverse applications: PARSEC, Phoenix, SPLASH2 benchmark suites, MySQL, Kyoto Cabinet, Memcached, RocksDB, SQLite, upscaledb and an SSL proxy. Among these 40 applications, we determine that performance varies according to the choice of a lock for roughly 60% of them, and perform our in-depth study on this subset of applications. We believe this set of applications to be representative of real-world applications: we consider applications that not only stress the classic lock/unlock interface to different extents, but also exhibit different usage patterns of condition variables, trylocks, barriers and that use different number of locks (i.e., from one global lock to thousands of locks). We consider four different multicore machines and three different metrics: throughput, tail latency and energy efficiency. In our quest to understand the behavior of locking, when choosing the per-configuration best lock, we improve on average application throughput by 90%, energy efficiency by 110% and tail latency 12× with respect to the default POSIX mutex lock (note that, in many cases, different locks optimize different metrics). As we show in this paper, choosing a well performing lock is difficult, as this choice depends on many different parameters: the workload, the underlying hardware, the degree of parallelism, the number of locks, how they are used, the lock-related interfaces that the application stresses (e.g., lock/unlock, trylock, condition variables), the interaction between the application and the scheduler, and the performance metric(s) considered.

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

Our third contribution is LiTL[1], an open-source, POSIX compliant [48], low-overhead library that allows transparent interposition of Pthread mutex lock operations and support for mainstream features like condition variables. Indeed, to conduct our study, manually modifying all the applications in order to retrofit the studied lock algorithms would have been a daunting task. Moreover, using a meta-library that allows plugging different lock algorithms under a common API (such as liblock [63] or libslock [27]) would not have solved the problem, as this still requires a substantial re-engineering effort for each application. In addition, such meta-libraries provide no or limited support for important features like Pthread condition variables, used within many applications. Our approach is a pragmatic one: similarly to what is done by previous works on memory allocators [3, 12, 41, 58], we argue that transparently switching (i.e., without modifying the application) lock algorithms (resp. memory allocators) is an efficient and pragmatic solution.

From our exhaustive study and our in-depth analysis, we make a number of observations regarding locking algorithms and applications behaviors, several of which have never been previously highlighted.

Applications not only stress the lock/unlock interface, but also the full locking API (e.g., trylocks, condition variables). Most of previous works focused on the lock/unlock interface performance of locks. We observe that many performance bottlenecks are related to other, less-considered lock operations. For example, applications use trylocks to implement busy-waiting as the traditional Pthread mutex implementation forces a thread to be descheduled while waiting for a lock. However, many lock algorithms that optimize for the lock/unlock interface perform poorly for trylock operations. Applications also heavily use condition variables, which directly interact with the lock instance in a way that was mostly ignored by lock algorithm designers. Pragmatically, locks should optimize not only the lock/unlock interface, but also all the other locking interfaces proposed by the Pthread mutex API.

The memory footprint of a lock may directly affect the application performance. Many lock algorithms improve performance by using more complex data structures. As an example, some algorithms use a per-thread context to store thread lock acquisition status. Other algorithms store statistics inside the lock instance, using these statistics to adapt the lock acquisition policy at runtime. However, all this complexity has a cost, as it increases the memory footprint of each lock instance. Indeed, we observe that some applications allocate thousands of lock instances, sometimes concurrently, which stresses the memory allocator, as well as hurts the processor cache locality, and as a consequence affects the application performance. Thus, lock designers should keep in

[1]LiTL: Library for Transparent Lock interposition.

mind that the memory footprint of their algorithm is an important factor, and they should try to design algorithms with a low memory footprint.

For many applications, the interaction between locks and scheduling is an important application performance factor. It is well known [14] that some lock algorithms exhibit poor performance in the context of over-threading (i.e., when there are more threads than available cores). Interestingly, we further observe that the interaction between locks and the scheduler affects the performance of many applications. Indeed, because applications use lock interfaces other than lock/unlock (e.g., condition variables) as well as other blocking functions (e.g., synchronization barriers, I/O syscalls), the Linux scheduler can take scheduling decisions that lead to poor application performance with some lock algorithms. In particular, we see that the *lock holder preemption* [14] and the *lock waiter preemption* [85] problems, both well known in the literature, frequently manifest in practice. A direct consequence of our observation is that lock designers should be aware that the scheduler decisions can impede application performance, and thus design locks that adapt themselves to a suboptimal scheduling.

Lock tail latencies may or may not affect application tail latency. Some locks are specifically designed to ensure perfect fairness for thread acquisitions, while others trade fairness for higher lock acquisition throughput. These properties directly affect the lock tail latency. Still, we observe that the effect of lock tail latency on the application tail latency is not straightforward. More precisely, if a high-level application operation is mostly implemented as a single critical section, then the performance (throughput and tail latency) of this operation highly depends on the properties of the lock. Hence, if low tail latency is desired, it is possible to choose a lock algorithm designed for fairness. Alternatively, if the developer is willing to trade application tail latency for throughput, lock algorithms trading fairness for throughput are a good choice. In contrast, we observe that for applications with limited concurrency (i.e. where an operation/request consists of many critical sections and/or sequential parts), the tail latency of locks does not strongly affect the tail latency of the application. In this case, we observe that lower application tail latency generally means higher application throughput, and as a consequence, a developer should choose a lock that brings the best throughput.

We also confirm previous findings [27, 36, 43] on a larger number of applications, machines, and lock algorithms.

No single lock is systematically the best. We observe that for the three metrics that we consider, approximately 60% of the studied applications are significantly affected by lock performance, hereafter called *lock-sensitive applications*. For lock-sensitive applications, at their optimized contention level (individually tuned), the best locks never dominate in more than 53% of the cases. A direct implication is that providing only a single lock algorithm (i.e., the Pthread lock) to software developers certainly results in suboptimal performance for most applications.

Choosing the best lock is difficult. For a given application, the best lock varies depending on the number of contending cores, the machine and the workload. Even worse, making the wrong choice affects the application, as all locks are harmful (i.e., significantly inefficient compared to the best one) for at least several workloads. Accordingly, developers should not hardwire the choice of a lock algorithm into the code of applications.

Energy efficiency and throughput go hand in hand in the context of lock algorithms. Previous work [36] introduced the POLY[2] conjecture. The POLY conjecture states that "energy efficiency and throughput go hand in hand in the context of lock algorithms". More precisely, POLY suggests that "locks can be optimized to improve energy efficiency without degrading throughput",

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

and that "[the insights from] prior throughput-oriented research on lock algorithms can be applied almost as-is in the design of energy-efficient locks". We verify POLY on a large number of lock algorithms and applications (the initial paper about POLY considered three lock algorithms and six applications).

A high-level ramification of many of these observations is that the research community must focus its efforts on designing **low-memory footprint adaptive locks that fully and efficiently support the full lock interface, and consider all performance metrics**.

The remainder of the paper is organized as follows: Section 2 presents a taxonomy of existing lock designs and the list of algorithms covered by our study. Section 3 describes our experimental setup and the studied applications. Section 4 describes the LiTL library. Sections 5, 6 and 7 respectively describe the main throughput, energy efficiency and tail latency experimental results. Section 8 presents the detailed analysis of lock-related performance bottlenecks and gives guidelines regarding the choice of a lock algorithm. Section 9 discusses related works and Section 10 concludes the paper.

2 LOCK ALGORITHMS

In this section, we present the 28 multicore lock algorithms that we consider in this study and organize them into five different categories based on their design properties. We then discuss an important lock-algorithm design-dimension, which is the choice of a *waiting policy*, i.e., what a thread does when it cannot immediately obtain a requested lock. Finally, we describe the list of the chosen lock algorithms for our empirical study.

2.1 Background

All modern lock algorithms rely on hardware atomic instructions to ensure that a critical section is executed in mutual exclusion. To provide atomicity, the processor relies on the cache-coherence protocol of the machine to implement an atomic read-modify-write operation on a memory address. Previous work [27] demonstrated that lock algorithm performance is mainly a property of the hardware, i.e., a lock algorithm must take into account the characteristics of the underlying machine. The design of a lock algorithm is thus a careful choice of data structures, lock acquisition/release policies and (potential) load-control mechanisms.

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

2.1.1 Synchronization primitives.

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

Mutual exclusion.

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

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

Conditions variables.

Threads often rely on condition variables to receive notifications when an event happens (e.g., when data is put inside a queue). A thread that wants to wait on a condition variable calls `wait` while holding a lock. As a consequence, the thread releases the lock and blocks[3]. When the condition is fulfilled, another thread calls `signal` or `broadcast` to wake any or all blocked threads, respectively. Upon wake-up (and before exiting from `wait`), threads compete to acquire the lock in order to re-enter the critical section. Efficiently implementing condition variables on top of locks is non-trivial (see Section 4.1).

2.1.2 Categorizing lock algorithms.

The body of existing work on optimized lock algorithms for multicore architectures is rich and diverse and can be split into the following five categories. The first two categories (competitive and direct handoff succession) are based on the succession policy [30] of the lock algorithm, i.e., how lock ownership is transferred at unlock-time. These two categories are mutually exclusive. The three other categories regroup algorithms that either compose algorithms from the first two categories (hierarchical approaches), change how critical sections are executed (delegation-based approaches), or improve existing locks with load-control mechanisms. Note that overall these categories overlap: a given algorithm can fall into several categories.

1) Competitive succession. Some algorithms rely on a competitive succession policy, where the lock holder sets the lock to an available state, and all competing threads might try to acquire it concurrently, all executing an atomic instruction on the same memory address. Such algorithms generally stress the cache-coherence protocol as they trigger cache-line invalidations at unlock-time to all cores waiting for the lock, while ultimately only one core succeeds in acquiring it. Competitive succession algorithms might allow *barging*, i.e., “arriving threads can barge in front of other waiting threads” [30], leading to unfairness and starvation. Examples of algorithms using a competitive succession policy are simple spinlock [83], Backoff spinlock [8, 70], test and test-and-set (ttas) lock [8], Mutexee lock [36] and standard Pthread mutex locks [48, 59].

2) Direct handoff succession. Direct handoff locks (also known as queue-based locks) are lock algorithms in which the unlock operation identifies a waiting successor and then passes ownership to that thread [30]. As the successor of the current lock holder is known, it allows each waiting thread to wait on a non-globally shared memory address (one per waiting thread). Then, the lock holder passes ownership with the help of this private memory address, thus avoiding cache-line

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

invalidations to all the other competing cores (contrary to the competitive succession policy). This approach is known to yield better fairness. Also, this approach generally gives better throughput under contention compared to simpler locks like spinlock. With direct handoff locks, each thread spins on its own local variable, avoiding to send cache lines invalidations to all other spinning cores when the lock is acquired/released (contrary to locks based on a global variable). Examples of direct handoff lock algorithms are: MCS [70, 83], CLH [24, 66, 83].

Some algorithms do use a globally shared memory address but still use a direct handoff succession policy. For example, Ticket lock [79] repeatedly reads a single memory address in a non-atomic fashion, waiting for its turn to come. The Partitioned Ticket lock [29] uses an hybrid solution, where the same memory address can be observed by a subset of the competing threads.

3) Hierarchical approaches. These approaches aim at providing scalable performance on NUMA machines, by attempting to reduce the rate of lock migrations (i.e., cache-line transfers), which are known to be costly between NUMA nodes. This category includes HBO [77], HCLH [65], FC-MCS [31], HMCS [20] and the algorithms that stem from the *lock cohorting* framework [32]. A cohort lock is based on a combination of two lock algorithms (similar or different): one used for the global lock and one used for the local locks (there is one local lock per NUMA node); in the usual $C\text{-}L_A\text{-}L_B$ notation, L_A and L_B respectively correspond to the global and the node-level lock algorithms. The list notably includes C-BO-MCS, C-PTL-TKT and C-TKT-TKT (also known as HTicket [27]). The *BO*, *PTL* and *TKT* acronyms respectively correspond to Backoff lock, Partitioned Ticket lock, and standard Ticket lock.

4) Delegation-based approaches. Delegation-based lock algorithms are locks in which it is (sometimes or always) necessary for a thread to delegate the execution of a critical section to another thread. The typical benefits expected from such approaches are improved cache locality and better resilience under very high lock contention. This category includes Oyama [74], Flat Combining [47], RCL [63], FFWD [81], CC-Synch [37] and DSM-Synch [37].

5) Load-control mechanisms. This category includes lock algorithms implementing mechanisms that detect situations in which a lock needs to adapt itself, for example to cope with changing levels of contention (i.e., how many threads concurrently attempt to acquire a lock), or to avoid lock-related pathological behaviors (e.g., preemption of the lock holder to execute a thread waiting for the lock). This category includes MCS-TimePub[4] [46], GLS [9], SANL [98], LC [52], AHMCS[5] [21] and so-called *Malthusian algorithms* like Malth_Spin and Malth_STP[6] [30].

2.1.3 Waiting policy.

An important design dimension of lock algorithms is the *waiting policy* used when a thread cannot immediately obtain a requested lock [30]. There are three main approaches.

Spinning. The most straightforward solution for waiting is to continuously check the status of the lock until it becomes available. However, such a policy might waste energy, and the time spent waiting on a core might prevent other descheduled threads from progressing. Processors provide special instructions to inform the CPU microarchitecture when a thread is spinning. For example,

[4]MCS-TimePub is mostly known as MCS-TP. Still, we use MC-TimePub to avoid confusion with MCS_STP.
[5]The original AHMCS paper [21] presents multiple versions of AHMCS. In this article, the version *without* hardware transactional memory of AHMCS is considered.
[6]Malth_Spin and Malth_STP correspond to MCSCR-S and MCSCR-STP respectively in the terminology of Dave Dice [30]; still we do not use the latter names to avoid confusion with other MCS locks.

x86 CPUs offer the PAUSE instruction[7] that is specifically designed to avoid branch-misprediction, and which informs the core that it can release shared pipeline resources to sibling hyperthreads [30].

In case of a failed lock acquisition attempt, different lock algorithms can use different (and possibly combine several) techniques to lower the number of simultaneous acquisitions attempts and the energy consumption while waiting. Using a fixed or randomized backoff (i.e., a thread avoids attempting to acquire the lock for some time) lowers the number of concurrent atomic instructions, thus the cache-coherence traffic. Hardware facilities can also be used to lower the frequency of the waiting thread's core (DVFS [92]), or to notify the core that it can enter in an idle state to save power (via the privilegied MONITOR/MWAIT instructions [36], accessible for locks running in privilegied mode, or via a kernel module [7]). Finally, a thread can voluntarily surrender its core in a polite fashion by calling sched_yield or sleep.

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

Hybrid approaches. The motivation behind hybrid approaches is that different waiting policies have different costs. For example, the *spin-then-park* policy is a hybrid approach using a fixed or adaptive spinning threshold [54]. It tries to mitigate the cost of parking as the block and unblock operations are expensive (both in terms of energy and performance). The spinning threshold is generally equal to the time of a round-trip context switch. Other techniques mix different spinning policies, such as backoff and sched_yield [27]. Finally, more complex combinations can be implemented: some algorithms [36, 90] trade fairness for throughput by avoiding to unpark a thread at unlock-time if there is another one currently spinning (also known as *adaptive unlock*).

The choice of the waiting policy is mostly orthogonal to the lock design but, in practice, policies other than pure spinning are only considered for certain types of locks: the direct handoff locks (from categories 2, 3 and 5 above), Mutexee and the standard Pthread mutex locks. However, this choice directly affects both the energy efficiency and the performance of a lock: Falsafi et al. [36] found that pure spinning inherently hurts power consumption, and that there is no practical way to reduce the power consumption of pure spinning. They found that blocking can indeed save power, because when a thread blocks, the kernel can then put the core(s) in one of the low-power idle states [6, 50]. However, the process of blocking is costly, because the cost of the blocking and unblocking operations is high on Linux. Switching continuously between blocking and unblocking can hurt energy efficiency, sometimes even more than using pure spinning policies. Thus, there is an energy-efficiency tradeoff between spinning and parking. Note that we use hereafter the expression *parking policy* to encompass both *immediate parking* and hybrid *spin-then-park* waiting policies.

2.2 Studied algorithms

We now describe the 28 mutex lock algorithms that are representative of both well-established and state-of-the-art approaches. Our choice of studied locks is guided by the decision to focus on *portable* lock algorithms. We therefore exclude the following locks that require strong assumptions on the

[7]The MFENCE instruction can also be used and is known to yield lower energy consumption than the PAUSE instruction on certain Intel processors [36].

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

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

application/OS behavior, code modifications, or fragile performance tuning: HCLH, HBO, FC-MCS (see Dice et al. [32] for detailed arguments). We also do not study delegation-based algorithms, because they require critical sections to be expressed as a form of closure (i.e., functions) [32], which is incompatible with our transparent approach (i.e., without source code modification). Finally, we do not consider runtime approaches like LC and GLS, which require special kernel support and/or monitoring threads.

We use the *_Spin* and *_STP* suffixes to differentiate variants of the same algorithm that only differ in their waiting policy (pure spinning vs spin-then-park). Unless explicitly specified by the lock algorithm implementation, we use the `PAUSE` instruction to pause between spinning loop iterations. The *-ls* tag corresponds to algorithms borrowed from libslock [27]. As well, note that the GNU C library for Linux provides two versions of Pthread mutex locks [40]: the default one uses immediate parking (via the `futex` syscall) and the second one uses an adaptive spin-then-park strategy. The latter version can be enabled with the `PTHREAD_MUTEX_ADAPTIVE_NP` option [59]. Our set of algorithms is summarized in Table 1 and includes eight competitive succession locks (Backoff, Mutexee, Pthread, PthreadAdapt, Spinlock, Spinlock-ls, TTAS, TTAS-ls), ten direct handoff locks (ALock-ls, CLH-ls, CLH_Spin, CLH_STP, MCS-ls, MCS_Spin, MCS_STP, Ticket, Ticket-ls, Partitioned), six hierarchical locks (C-BO-MCS_Spin, C-BO-MCS_STP, C-PTL-TKT, C-TKT-TKT, HTicket-ls, HMCS), and four load-control locks (AHMCS, Malth_Spin, Malth_STP, MCS-TimePub).

Table 1. A short description of the 28 multicore lock algorithms that we consider.

Name	Reference	Short description
Competitive		
Backoff	[70]	Test-and-set (TAS) with exponential bounded backoff if the lock is already held.
Mutexee	[36]	A spin-then-park (STP) lock designed with energy efficiency in mind.
Pthread	[39]	TAS with direct parking.
PthreadAdapt	[59]	An adaptive STP algorithm, performing a number of trylocks (before blocking) that depends on the number of trylocks performed by the lock holder when it acquired the lock.
Spinlock	[8]	Compare-and-set algorithm with busy waiting.
Spinlock-ls	[27]	TAS algorithm with busy waiting.
TTAS	[8]	Performs non-atomic loads on the lock memory address before trying to acquire it atomically with a TAS instruction.
TTAS-ls	[27]	Similar to TTAS but uses an exponential bounded backoff if the TAS fails.
Direct handoff		
ALock-ls	[8]	The *waiting* threads are organized inside a fixed-sized array, i.e., there is a fixed bound N on the number of waiting threads. A thread waits on one of the private cache-aligned array slot. At unlock-time, the lock holder wakes the next thread by changing the content of the slot on which the next thread is waiting.
CLH_Spin	[24, 66]	Waiting threads are organized as an inverse linked-list, where a thread spins on the context (i.e., linked-list node) of its predecessor. At unlock-time, the lock holder wakes up the thread at the head of the waiting list.
CLH_STP	[24, 66]	Similar to CLH_Spin but uses a STP waiting policy.
CLH-ls	[27]	Similar to CLH_Spin but uses the `PREFETCHW` x86 CPU instruction while spinning.
MCS_Spin	[70]	Waiting threads are organized as a linked-list, where a thread spins on its private context. At unlock-time, the lock holder wakes up its successor.
MCS_STP	[70]	Similar to MCS_Spin but uses a STP waiting policy.
MCS-ls	[27]	Similar to MCS_Spin but uses the `PREFETCHW` x86 CPU instruction while spinning.
Ticket	[79]	A thread trying to acquire the lock atomically takes a “ticket” (implemented as an incrementing counter) and spins while its ticket is not equal to the “next-ticket” number. At unlock-time, the lock holder increments the “next-ticket” number.
Ticket-ls	[27]	Similar to Ticket but a thread waits proportionally to the number of threads waiting before him.
Partitioned	[29]	Similar to Ticket but the “next-ticket” number is implemented inside an array, where a thread waits on its “ticket” slot ($slot = ticket \% size(array)$).
Hierarchical		
C-BO-MCS_Spin	[32]	A thread first tries to acquire a MCS_Spin local lock shared by all threads on the same NUMA node (the local lock), then competes on the Backoff top lock with other threads holding their respective local locks.
C-BO-MCS_STP	[32]	Similar to C-BO-MCS_Spin but uses a STP waiting policy for te MCS locks.
C-PTL-TKT	[32]	Similar to C-BO-MCS_Spin but the local locks are Ticket locks and the top lock is a Partitioned lock.
C-TKT-TKT	[32]	Similar to C-BO-MCS_Spin but the top and local locks are Ticket locks.
HTicket-ls	[27]	Similar to C-TKT-TKT but a thread waits proportionally to the number of threads waiting before him.
HMCS	[20]	Similar to C-BO-MCS_Spin but the top and local locks are MCS_Spin locks.
Load-control		
AHMCS	[21]	Similar to HMCS, but when a thread tries to acquire the lock, it remembers if the last time it released the lock there was a thread waiting. If not, it only locks the top lock because it assumes low contention the lock. The AHMCS version *without* hardware transactional memory is considered.
Malth_Spin	[30]	A variant of the MCS_Spin lock where, when there is contention on a lock, a subset of the spinning competing threads are put aside temporarily to let the others progress more easily.
Malth_STP	[30]	Similar to Malth_Spin but threads use a STP waiting policy.
MCS-TimePub	[46]	A variant of the MCS_Spin lock, in which a waiting thread relinquishes its core if it detects (heuristically, using timers and thresholds) that the lock holder has been preempted. At unlock-time, the lock holder might bypass some waiting threads if it detects they have been preempted.

3 METHODOLOGY

In this section we describe our study's methodology. We first describe the different testbed platforms we use and the applications we study (Section 3.1). Then, in Section 3.2, we present our tuning choices and our experimental methodology.

Table 2. Hardware characteristics of the testbed platforms.

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

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

3.1 Testbed and studied applications

Our experimental testbed consists of four Linux-based x86 multicore servers whose main characteristics are summarized in Table 2. All the machines run the Ubuntu 12.04 OS with a 3.17.6 Linux kernel (CFS scheduler), except the I-20 machine running an Ubuntu 14.04 OS with a 3.13 Linux kernel. We tried to keep the software configuration as similar as possible for the different versions: they all use glibc (GNU C Library) version 2.15 (2.19 for I-20) and gcc version 4.6.3 (4.6.4 on

Table 3. Applications considered.

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

I-48). We configured the BIOS of the A-64 and the A-48 machines in performance mode (processor throttling is turned off so that all cores run at maximum speed, e.g., no C-state, no turbo mode). The BIOS of the I-48 and I-20 machines in performance mode for the throughput experiments, and in energy-saving mode for the energy-efficiency experiments. For all configurations, hyper-threading is disabled.

Table 3 lists the applications we chose for our comparative study of lock performance and lock energy efficiency. More precisely, we consider (i) the applications from the PARSEC benchmark suite version 3.0 (emerging workloads) [13], (ii) the applications from the Phoenix 2.0 MapReduce benchmark suite [78], (iii) the applications from the SPLASH2x high-performance computing benchmark suite [13][10], (iv) the MySQL database version 5.7.7 [73] running the Cloudstone workload [88], (v) SSL proxy, an event-driven SSL endpoint that processes small messages, (vi) upscaledb 2.2.0 [22], an embedded key/value running the ham_bench benchmark, (vii) the Kyoto Cabinet database version 1.2.76 [35], a standard relational database management system running the included benchmark, (viii) Memcached, versions 1.4.15 and 1.4.36[11] [16], an in-memory cache system, (ix) RocksDB 4.8 [34], a persistent key/value store running the included benchmark, and (x) SQLite 3.13 [89], an embedded SQL database using the dbt2 TPC-C workload generator[12]. We use remote network injection for the MySQL and the SSL proxy applications. For Memcached, similarly to other setups used in the literature [36, 63], the workload runs on a single machine: we dedicate one socket of the machine where we run `memaslap` to inject network traffic to the Memcached instance, the two running on two distinct sets of cores. For the Kyoto Cabinet application, like in previous work [30], we redirect calls to `rw_lock` to classic `mutex_lock` calls. This might change the synchronization pattern of the application, yet this aplication is still interesting to consider because its performance is known to vary according to lock algorithms [19]. By default, phoenix launches one thread per available core, and pins each thread to one core. However, to have the same baseline for all our benchmarks, we decided to disable pinning in phoenix, leaving to the scheduler the thread placement decisions. Note that when benchmarks are evaluated in a thread-to-node pinning configuration (see Section 5.3), phoenix is also evaluated on a thread-to-node pinning configuration.

In order to evaluate the impact of workload changes on locking performance and energy efficiency, we also consider "long-lived" variants of four of the above workloads (pca, s_raytrace, radiosity and streamcluster) denoted with a "_ll" suffix. The motivation behind these versions is to stress the application's steady-state phase, where the locks are mostly acquired/released. By contrast, the short-lived versions allow us to benchmark the performance of the initialization and cleanup operations of a lock algorithm. For each application, we modified it to report throughput (in operations per seconds, e.g., number of rays traced for an application that renders a 3-D phase) and use larger input size. We capture the throughput of the "steady-state" phase exclusively, ignoring the impact of the start/shutdown phases. Note that six of the applications only accept, by design, a number of threads that corresponds to a power of two: facesim, fluidanimate (from PARSEC), fft, ocean cp, ocean ncp, radix (from SPLASH2). We decide to not include experiments for these six applications on the two 48-core machines and the 20-core machine, in order to keep the presentation of results uniform and easy. Besides, we were not able to evaluate the applications using network injection on the I-20 machine due to a lack of high-throughput network connectivity.

Some *(application,lock algorithm,machine)* configurations cannot be evaluated, for the following reasons. First, due to a lack of memory (especially on the A-48, which only has 64 GB of memory),

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

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

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

and because some applications allocate too many lock instances and the memory footprint of some lock algorithms is high: (i) AHMCS with dedup and fluidanimate on all machines, and (ii) CLH, ALock-ls, TTAS-ls with dedup on A-48 results are not reported. Second, fluidanimate, Memcached-old, Memcached-new, streamcluster, streamcluster_ll, vips rely on trylock operations. CLH algorithms and HTicket-ls do not support trylock, and Partitioned and C-PTL-TKT trylock implementations might block threads for a short time (which can cause deadlocks with Memcached-*). Those configurations are not evaluated. Finally, most of the studied applications use a number of threads equal to the number of cores, except the four following ones: dedup (3× threads), ferret (4× threads), MySQL (hundreds of threads) and SQLite (hundreds of threads). For applications with significantly more threads than cores (SQLite and MySQL), we exclude results for algorithms using a spinning waiting policy: these applications suffer from the lock holder preemption issue (see Section 8.1.2 for more details) up to a point where performance drops close to zero.

3.2 Tuning and experimental methodology

For the lock algorithms that rely on static thresholds, we use the recommended values from the original papers and implementations. The algorithms based on a spin-then-park waiting policy (e.g., Malth_STP [30]) rely on a fixed threshold for the spinning time that corresponds to the duration of a round-trip context switch [54]—in this case, we calibrate the duration using a microbenchmark on the testbed platform. All the applications are run with memory interleaving (via the `numactl` utility) in order to avoid NUMA memory bottlenecks[13]. Datasets are copied inside a temporary file-storage facility (tmpfs) before running experiments, to avoid disk I/O. For most of the experiments detailed in the paper, the application threads are not pinned to specific cores. Note that for hierarchical locks, which are composed of one top lock and one per-NUMA node bottom lock, a thread always tries to acquire the bottom lock where it is *currently* running. Doing so, cache coherence traffic is limited, which is one of the main reason behind the design of hierarchical locks. The effect of pinning is nonetheless discussed in Section 5.3.

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

We consider three metrics: application-level throughput, tail latency, and energy efficiency. More precisely, for throughput, (i) for MySQL, SSL Proxy, upscaledb, Kyoto Cabinet, RocksDB and SQLite, the application throughput is used as a performance metric, (ii) for the long-lived applications, progress points are inserted in the source code of the application, and (iii) for all the other applications, the inverse of the total execution time is used. For tail latency, we consider the application tail latency, here defined as the 99th percentile of client response time. We perform energy consumption measurements using the RAPL (Running Average Power Limit) [51] power meter interface on the two Intel machines (I-48 and I-20). RAPL is an on-chip facility that provides counters to measure the energy consumption of several components: cores, package and DRAM. We do not capture energy for our two AMD machines as they do not have APM (Application Power Management), AMD's version of RAPL.

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

We run each experiment at least 5 times and compute the average value. For long-lived and server workloads, a 30-second warmup phase precedes a 60-second capture phase, before killing the application. For configurations exhibiting high variability (i.e., more than 5% of relative standard deviation), we run more experiments, trying to lower the relative standard deviation of the configuration, to increase the confidence in our results. More precisely, we found that roughly 15% of the *(application, lock algorithm, machine, number of threads)* configurations have a relative standard deviation (rel.stdev.) higher than 5%. Besides, 6% of the configurations have a rel.stdev higher than 10% and 2% higher than 20%. C-BO-MCS_STP, TTAS and Spinlock-ls are the studied lock algorithms that exhibit the higher variability: the rel.stdev of these locks is higher than 5% for 20% of the configurations. Concerning the applications, ocean_cp, ocean_ncp, streamcluster and fft exhibit a high rel.stdev (roughly 50% of the configurations have a rel.stdev higher than 5%). Finally, streamcluster, dedup and streamcluster_ll are applications for which some configurations exhibit a very high rel.stdev (higher than 20% in 10% of the cases). In order to mitigate the effects of variability, when comparing two locks, we consider a margin of 5%: lock A is considered better than lock B if B's performance (resp. energy efficiency or tail latency) is below 95% of A's. Besides, in order to make fair comparisons among applications, the results presented for the Pthread locks are obtained using the same library interposition mechanism (see Section 4) as with the other locks.

Finally, for the sake of space, we do not report all the results for the four studied machines. We rather focus on the A-64 machine for the different studies and provide summaries of the results for the other machines, which are in accordance to the results on the A-64 machine. Nevertheless, the entire set of results can be found in the electronic Appendix. We also do not systematically report, for the sake of readability, the standard deviations as they are low for most configuration. Note that the raw dataset (for all the experiments, on all machines) of throughput, tail latency and energy is available online [44], letting the readers perform their own analysis.

4 LITL: A LIBRARY FOR TRANSPARENT LOCK INTERPOSITION

In this section we present the LiTL library, an open-source, POSIX compliant, low-overhead library that allows transparent interposition of Pthread mutex lock operations and support for mainstream features like condition variables. We first describe the design of LiTL in Section 4.1, discuss its implementation in Section 4.2, evaluate some elementary costs introduced by LiTL in Section 4.3, and experimentally assess its performance in Section 4.4.

4.1 Design

We describe the general design principles of LiTL, how it supports condition variables, and how it can easily be extended to support specific lock semantics. The pseudo-code of the main wrapper functions of the LiTL library is depicted in Figure 1.

General principles. The primary role of LiTL is to maintain a mapping between an instance of the standard Pthread lock (`pthread_mutex_t`) and an instance of the chosen optimized lock type (e.g., MCS_Spin). This mapping is maintained in an external data structure (see details in §4.2), rather than using an "in-place" modification of the `pthread_mutex_t` structure. This choice is motivated by two main reasons. First, for applications that rely on condition variables, we need to maintain a standard `pthread_mutex_t` lock instance (as explained later in this section). Second (and regardless of the previous reason), LiTL is aimed at being easily portable across C standard libraries. Given that the POSIX standard does not specify the memory layout and contents of the

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

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

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

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

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

Fig. 1. Pseudocode for the main wrapper functions of LiTL.

`pthread_mutex_t` structure[14], it it is non-trivial to devise an "in-place modification" approach that is at the same time safe, efficient and portable.

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

The `lock`/`unlock` API of several lock algorithms requires an additional parameter (called *struct* hereafter) in addition to the lock pointer, e.g., in the case of an MCS lock, this parameter corresponds

[14]In fact, different standard libraries [38, 39] and even different versions of the same library have significantly different implementations.

to the record to be inserted in (or removed from) the lock's waiting queue. In the general case, a struct cannot be reused nor freed before the corresponding lock has been released. For instance, an application may rely on nested critical sections (i.e., a thread T must acquire a lock L_2 while holding another lock L_1). In this case, T must use a distinct struct for L_2 in order to preserve the integrity of L_1's struct. In order to gracefully support the most general cases, LiTL systematically allocates exactly one struct per lock instance and per thread (a static array is allocated alongside the lock instance, upon the first access to the lock instance), while taking care of avoiding false-sharing of cache lines among threads. LiTL uses the default memory allocator (glibc ptmalloc), which has per-thread arenas to avoid lock contention (since glibc 2.15) [49].

Supporting condition variables. Efficiently dealing with condition variables inside each optimized lock algorithm would be complex and tedious as most locks have not been designed with condition variables in mind. Indeed, most lock algorithms suffer from the so-called *thundering-herd* effect, where all waiting threads unnecessary contend on the lock after a call to `pthread_cond_broadcast`[15], which might lead to a scalability collapse. The Linux Pthread implementation does not suffer from the *thundering-herd* effect, as it only wakes up a single thread from the wait queue of the condition variable and directly transfers the remaining threads to the wait queue of the Pthread lock. However, to implement this optimization, all the waiting threads must block on a single memory address[16], which is incompatible with lock algorithms that are not based on a competitive succession policy.

We therefore use the following generic strategy: our wrapper for `pthread_cond_wait` internally calls the actual `pthread_cond_wait` function. To issue this call, we hold a real Pthread mutex lock (of type `pthread_mutex_t`), which we systematically acquire just after the optimized lock. This strategy (depicted in the pseudocode of Figure 1) does not introduce high contention on the real Pthread lock. Indeed, (i) for workloads that do not use condition variables[17], the Pthread lock is only requested by the holder of the optimized lock associated with the critical section and, (ii) workloads that use condition variables are unlikely to have more than two threads competing for the Pthread lock (the holder of the optimized lock and a notified thread).

A careful reader might suggest to take the Pthread lock only before calling `pthread_cond_wait` on it. This approach has been proposed by Lozi et al. [63], but we discovered that it suffers from liveness hazards due to a race condition. Indeed, when a thread T calls `pthread_cond_wait`, it is not guaranteed that the two steps (releasing the lock and blocking the thread) are always executed atomically. Thus, a wake-up notification issued by another thread may get interleaved between the two steps and T may remain indefinitely blocked.

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

Support for specific lock semantics. Our implementation is compliant with the specification of the `DEFAULT non-robust` POSIX mutex type [48]. More precisely, we do not support lock holder crashes (robustness), relocking the same lock can lead to deadlock or undefined behavior, and the behavior of unlocking a lock with a non-holder thread is undefined (it depends on the underlying lock algorithm).

---

[15] 19 out of 40 of our studied application uses this operation, in most cases to implement barriers.

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

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

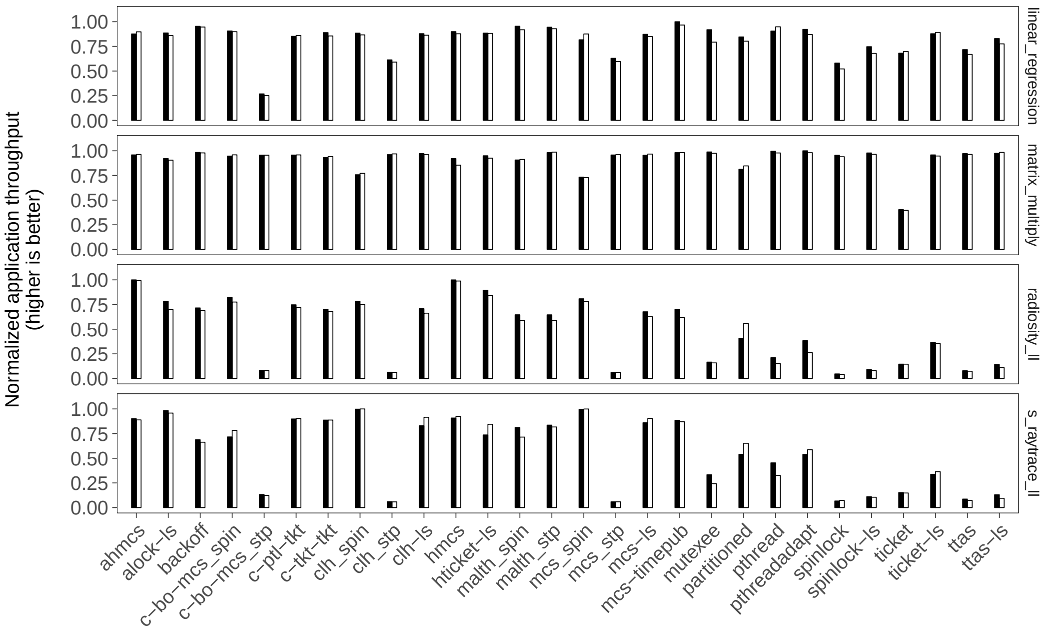

Fig. 2. Performance comparison (throughput) of manually implemented locks (black bars) vs. transparently interposed locks using LiTL (white bars) for 4 different applications. The throughput is normalized with respect to the best performing configuration for a given application (**A-64 machine**).

The design of LiTL is compatible with specific lock semantics when the underlying lock algorithms offer the corresponding properties. For example, LiTL supports non-blocking lock requests (`pthread_mutex_trylock`) for all the currently implemented locks except CLH-based locks and HTicket-ls, which are not compatible with the trylock non-blocking operation[18]. Although not yet implemented, LiTL could easily support blocking requests with timeouts for the so-called "abortable" locks (e.g., MCS-Try [84] and MCS-TimePub [46]). Moreover, support for optional Pthread mutex behavior like reentrance and error checks[19] could be easily integrated in the generic wrapper code by managing fields for the current owner and the lock acquisition counter. Note that none of the applications that we have studied requires a non-`DEFAULT` POSIX mutex type.

4.2 Implementation

The library relies on a scalable concurrent hash table (CLHT [28]) in order to store, for each Pthread mutex instance used in the application, the corresponding optimized lock instance, and the associated per-thread structs. For well-established locking algorithms like MCS, the code of LiTL borrows from other libraries [4, 27, 36, 63]. Other algorithms (i.e., CLH, C-BO-MCS, C-PTL-TKT, C-TKT-TKT, HMCS, AHMCS, Malthusian, Partitioned, Spinlock, TTAS) are implemented from scratch based on the description of the original papers. For algorithms that are based on a parking waiting policy, our implementation directly relies on the `futex` Linux system call.

[18]The design of the Partitioned (and by extension C-PTL-TKT) lock does not allow implementing a perfect trylock, i.e., a trylock that never blocks. As a consequence, if two threads try to acquire the lock simultaneously, one of them might block for a short time.
[19]Using respectively the `PTHREAD_MUTEX_RECURSIVE` and `PTHREAD_MUTEX_ERRORCHECK` attributes.

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

4.3 Lookup overhead

To assess the overhead of performing a lookup in the hash table each time a lock is accessed, we designed a micro-benchmark in which threads perform concurrent lookups, varying the number of threads (from 1 to 64) and the number of elements[20] (from 1 to 32768). On the A-64 machine, no matter the number of lock instances, at 1 thread, a look-up costs 20 cycles, and from 2 to 64 threads, 25 cycles. The 5-cycle difference is explained by the fact that on the A-64 machine, two siblings cores share some microarchitectural units of the CPU.

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

4.4 Experimental validation

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

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

Figure 2 shows the normalized performance (throughput) of both configurations (manual/interposed) for each *(application, lock)* pair. In addition, Table 4 summarizes the performance differences for each application.

We observe that, for all four applications, the results achieved by the two versions of the same lock are very close: the average performance difference is never higher than 8%. Besides, Figure 2 highlights that the general trends observed with the manual versions are preserved with the interposed versions.

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

Table 4. Detailed statistics for the performance comparison of manually implemented locks vs. transparently interposed locks using LiTL (**A-64 machine**).

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

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

	linear_regression	matrix_multiply	radiosity_ll	s_raytrace_ll
Better	97%	98%	100%	98%
Equal	87%	93%	91%	93%
Worse	96%	99%	98%	98%

Table 5 compares the relative performance of all lock pairs. The table shows that in most cases (at least 87%), comparing two manually implemented lock algorithms leads to the same conclusion as comparing their transparently interposed versions.

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

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

Table 6. For each application, the p-value of the paired Student t-test testing the null hypothesis $Mean_{with} - Mean_{without} = C$. C_n is C normalized w.r.t. the performance of the best lock on a given benchmark).

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

LiTL on absolute performance is low. We choose C equal to the average throughput difference with and without interposition for all locks for a given application.

Table 6 shows the constant C_n (C normalized w.r.t. the performance of the best lock on a given benchmark) as well as the t-test's p-value. For example, for linear_regression, when removing 1.8% of the maximal throughput (0.03 seconds) to each interposed configuration, the p-value is 0.84. A p-value must be compared against a threshold α, upon which we reject/accept the null hypothesis (i.e., in our case„ "means are equal, up to a constant"). The higher the p-value, the lower the risk to incorrectly reject the null hypothesis. All the tested applications have p-value > 0.05 (the most commonly used threshold [72]), thus we never reject the null hypothesis, thus the means can be considered equal (up to a constant C).

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

5 STUDY OF LOCK THROUGHPUT

In this section, we use LiTL to compare the performance (throughput) behavior of the different lock algorithms on different workloads and at different levels of contention. Our experimental methodology is described in Section 3. In Sections 6 and 7 we present the results for energy efficiency and tail latency, respectively.

As a summary, Section 5.1 provides preliminary observations that drive the study. Section 5.2 answers the main questions of the study regarding the observed lock behavior. Section 5.3 discusses additional observations, such as how the machine, the BIOS configuration, and the thread pinning affect the results as well as the performance of Pthread locks. Section 5.4 discusses the implications of our study for software developers and for the lock algorithm research community.

5.1 Preliminary observations

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

5.1.1 Selection of lock-sensitive applications. Table 7 shows two metrics for each application and for different numbers of nodes on the A-64 machine (results for the other machines are available in the electronic Appendix): the performance gain of the best lock over the worst one, as well as the relative standard deviation for the performance of the different locks. Note that columns of Table 7 cannot be compared to each other. Indeed, the numbers reported are the performance gain and relative standard deviation for the best vs. worst lock at a given number of nodes, i.e., gain at *max nodes* compares the performance of the best vs. worst lock at *max nodes*, whereas gain at *opt nodes* compares the performance of the best vs. worst lock at their *respective* optimal number of nodes (where they perform best).

Besides, the numbers reported at max nodes are generally higher than at *opt nodes* because performance gaps between locks tend to increase under high contention, which is why we chose the A-64 machine: it has the highest number of cores among our different machines. For the moment, we only focus on the relative standard deviations at the maximum number of nodes (*max nodes*—highest contention) given in the fifth column (the detailed results from this table are discussed in Section 5.2.1).

We consider that an application is *lock-sensitive* if the relative standard deviation for the performance of the different locks at *max nodes* is higher than 10% (highlighted in bold font in the Table). We observe similar trends on the four studied machines (see Table 8). More precisely, we observe that about 60% of the applications are affected by locks, for all machines except the I-20 where the percentage of application is slightly lower (49%). Some applications are lock-sensitive on some machines and not on others. For example, fmm is only lock-sensitive on the AMD machines, not the Intel ones. For such applications, we observe a moderate relative standard deviation at *max nodes* (< 30%), meaning that they are considered lock-sensitive but they are not the applications that are the most affected by locks. Indeed, we do not observe applications that are highly affected by locks on one machine and not on another. In the remainder of this study, we focus on lock-sensitive applications.

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

Table 9 shows for each *(application, lock)* pair, for the A-64 machine the performance gain of *opt nodes* over *max nodes* and the number of nodes for *opt nodes* (results for the other machines are available in the electronic Appendix). A line full of black boxes means that the optimal number of nodes is the maximal number of nodes, i.e., for all locks, the best performance is seen at *max nodes* (the performance of the application does not collapse). However, it is still interesting to consider these applications, because a line full of black boxes does not mean that all locks performs the same, e.g., for water_nsquared, the gain between the best vs. the worst locks at *max nodes* and *opt nodes* is of 94% (Table 7). In addition, Table 10 provides a breakdown of the *(application, lock)* pairs according to their optimized number of nodes for all machines.

We observe that, for many applications, the optimized number of nodes is lower than the max number of nodes. Moreover, we observe (Table 9) that the performance gain of the optimized configuration is often extremely large. We note that the performance gains for the I-20 is lower than the ones for the other machines, which have more cores. This confirms that tuning the degree of parallelism has frequently a very strong impact on performance. We also notice that, for some applications, the optimized number of nodes varies according to the chosen lock (on pca_ll ALock-ls is optimal at 4 nodes, Backoff at 8 nodes), the chosen waiting policy (on pca_ll Malth_Spin is optimal at 4 nodes, Malth_STP at 8 nodes) and the workload (Backoff is optimal at 2 nodes on pca and at 8 nodes on pca_ll).

5.2 Main questions

Table 7. For each application, performance gain of the best vs. worst lock and relative standard deviation (**A-64 machine**).

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

Table 8. Number of applications and number of lock performance sensitive applications (**all machines**).

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

5.2.1 How much do locks affect applications? Table 7 shows, for each application, the performance gain of the best lock over the worst one at *one node*, *max nodes*, and *opt nodes* for the A-64 machine. The table also shows the relative standard deviation for the performance of the different locks.

We observe that the number of nodes affects the performance of applications. **At *one node,* the impact of locks on lock-sensitive applications is moderate for most applications.** Nonetheless, for the most lock-sensitive ones (upscaledb, MySQL, Kyoto Cabinet, dedup), we observe that the impact is high. More precisely, most applications exhibit a gain of the best lock over the worst one that is lower than 30%. In contrast, **at *max nodes,* the impact of locks is very high for all lock-sensitive applications**. More precisely, the gain brought by the best lock over the worst lock ranges from 42% to 2382%. Finally, **at *opt nodes,* the impact of locks is high, but noticeably lower than at *max nodes.*** We explain this difference by the fact that, at *max nodes*, some of the locks trigger a performance collapse for certain applications (as shown in Table 9), which considerably increases the observed performance gaps between locks. Note that the collapse is not necessarily related to a given lock, but is also a property of the application and how the machine behaves We observe the same trends on the A-48, the I-48 and the I-20 machines (see the electronic Appendix).

5.2.2 Are some locks always among the best? Table 11 displays, for each machine, the coverage of each lock, i.e., how often it stands as the best one (or is within 5% of the best) over all the studied applications, over the different locks. The details for all machines are available in the electronic Appendix.

We make the following observations. On the A-64, A-48 and I-48 machines, **no lock is among the best for more than 76% of the applications at *one node* and for more than 53% of the applications both at *max nodes* and at the optimal number of nodes**. The results for the I-20 show that the coverage of a given lock algorithm is larger than for the other machines (75% at *one node*, *max nodes* and *opt nodes*). This can be explained by the fact that the machine has less cores (and NUMA sockets) than the three others. Nonetheless, for all machines, no lock algorithm is optimal for all applications. We also observe that the average coverage is much higher at *one node* than at *max nodes*, and slightly higher at *opt nodes* than at *max nodes.* This is directly explained by the observations made in Section 5.2.1. First, at *one node*, locks have a much lower impact on applications than in other configurations and thus yield closer results, which increases their likelihood to be among the best ones. Second, at *max nodes*, all of the different locks cause, in turn, a performance collapse, which reduces their likelihood to be among the best locks. This latter phenomenon is not observed at *opt nodes.*

5.2.3 Is there a clear hierarchy between locks? Figure 3 shows pairwise comparisons for all locks, at *max nodes* on the A-64 machine.

We observe that **there is no clear global performance hierarchy between locks**. More precisely, for most pairs of locks (*row A, col B*), there are some applications for which *A* is better than *B*, or vice-versa (Figure 3). The only marginal exceptions are the cells having 0% for value.

Table 9. For each *(lock-sensitive application, lock)* pair, performance gain (in %) of *opt nodes* over *max nodes*. The background color of a cell indicates the number of nodes for *opt nodes*: 1|2|4|6|8. Dashes correspond to untested cases (**A-64 machine**).

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

Table 10. Breakdown of the *(lock-sensitive application, lock)* pairs according to their optimized number of nodes (**all machines**).

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

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

	I-20
1 Node	39%
2 Nodes	61%

Table 11. Statistics on the coverage of locks on lock-sensitive applications for three configurations: *one node*, *max nodes*, and *opt nodes* (**all machines**). The coverage indicates how often a lock algorithm stands as the best one (or is within 5% of the best).

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

This corresponds to pairs of locks (*row A, col B*) for which *A* never yields better performance than *B*. The results at *max nodes* (available in the electronic Appendix) exhibit similar trends as the ones at *opt nodes*. Besides, we make the same observations (both at *opt nodes* and *max nodes*) on the A-48, the I-48 machines and the I-20 (see the electronic Appendix).

5.2.4 Are all locks potentially harmful? Our goal is to determine, for each lock, if there are applications for which it yields substantially lower performance than other locks and to quantify the magnitude of such performance gaps. Table 12 displays, for each machine, the fraction of applications that are significantly hurt by a given lock at *max nodes* and at *opt nodes* (results for all machines in the electronic Appendix).

On the four machines, we observe that, **both at *max nodes* and at the optimal number of nodes, all locks are potentially harmful, yielding sub-optimal performance for a significant number of applications** (Table 12). We also notice that locks are significantly less harmful at *opt nodes* than at *max nodes*. This is explained by the fact that several of the locks create performance collapse at *max nodes*, which does not occur at *opt nodes*. Moreover, we observe that, for each lock, the performance gap to the best lock can be significant (Table 12).

5.3 Additional observations

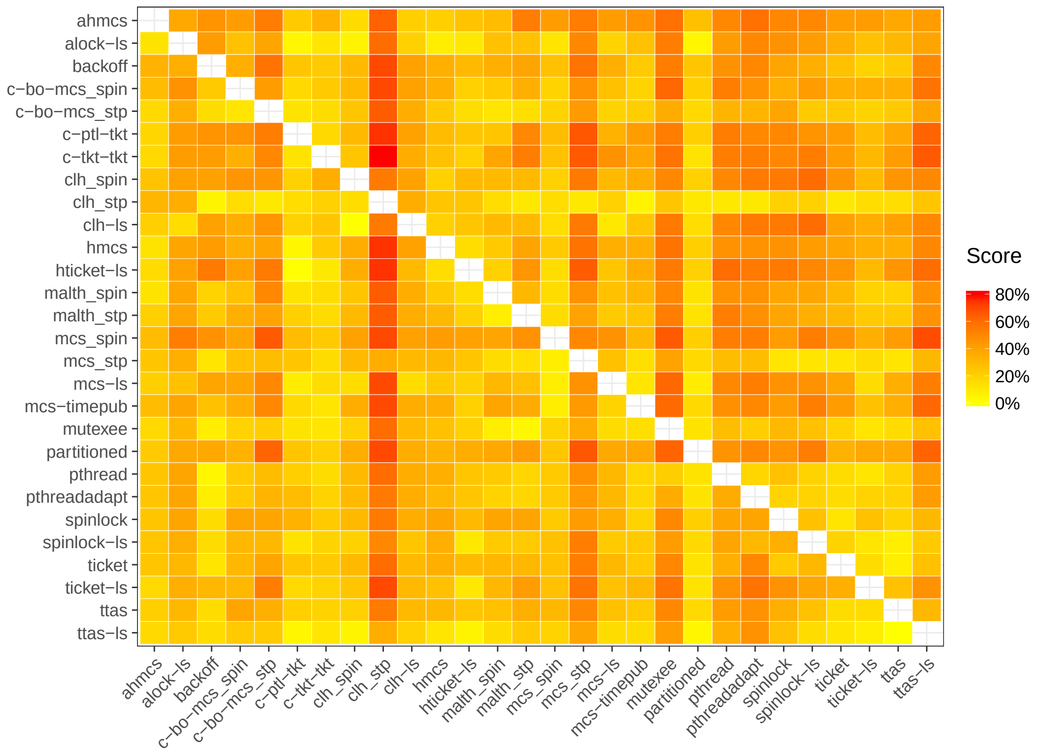

Fig. 3. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock *A* vs lock *B*: percentage of lock-sensitive applications for which lock *A* performs at least 5% better than *B*. The cell (*rowA*, *colB*) color indicates the score of lock *A* vs. lock *B*, i.e., the percentage of applications for which lock *A* is at least 5% better than lock *B*. The more lock *A* outperforms *B*, the more red (dark) the cell is. For example, for roughly 40% of the applications, AHMCS performs at least 5% better than Backoff at *opt nodes*. Similarly, the figure shows that Backoff is at least 5% better than AHMCS for roughly 35% of the applications. From these two values, we can conclude that the two above-mentioned locks perform very closely for 25% of the applications. (**A-64 machine**).

Impact of the number of nodes. Table 13 shows, for each application on the A-64 machine, the number of pairwise changes in the lock performance hierarchy when the number of nodes is modified. We observe that, **for all applications, the lock performance hierarchy changes significantly according to the chosen number of nodes**. Moreover, we observe the same trends on the A-48, I-48 and I-20 machines (see the electronic Appendix).

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

A note on Pthread locks. The various results presented in this paper show that the current Linux **Pthread locks perform reasonably well (i.e., are among the best locks) for a significant**

Table 12. For each lock, at *max nodes* and at *opt nodes*, fraction of the lock-sensitive applications for which the lock is harmful, i.e., the performance gain brought by the best lock with respect to the given lock is greater than 15% (**all machines**).

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

share of the studied applications, thus providing a different insight than recent results, which were mostly based on synthetic workloads [27]. Beyond the changes of workloads, these differences could also be explained by the continuous refinement of the Linux Pthread implementation. It is nevertheless important to note that on each machine, some locks stand out as the best ones for a higher fraction of the applications than Pthread locks. Finally, we note that Pthread locks and PthreadAdapt locks exhibit similar performance.

Impact of thread pinning. As explained in Section 3.2, all the previously-described experiments were run without any restriction on the placement of threads (i.e., a thread might be scheduled on any core of the machine), leaving the corresponding decisions to the Linux scheduler. However, in order to better control cores allocation and improve locality, some developers and system administrators use pinning to explicitly restrict the placement of each thread to one or several

Table 13. For each lock-sensitive application, percentage of pairwise changes in the lock performance hierarchy when changing the number of nodes. For example, in the case of the facesim application, there are 17% of the pairwise performance comparisons between locks that change when moving from a 1-node configuration to a 2-node configuration. Similarly, there are 97% of pairwise comparisons that change at least once when considering the 1-node, 2-node, 4-node and 8-node configurations. (**A-64 machine**).

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

core(s). The impact of thread pinning can vary greatly according to workloads and can yield both positive and negative effects [27, 64]. In order to assess the generality of our observations, we also performed the complete set of experiments on the A-64 machine with an alternative configuration in which each thread is pinned to a given node, leaving the scheduler free to place the thread among the cores of the node. Note that for an experiment with a N-node configuration, the complete application runs on exactly the first N nodes of the machine. We chose thread-to-node pinning rather than thread-to-core pinning because we observed that the former generally provided better performance for our studied applications, especially the ones using more threads than cores. The detailed results of our experiments with thread-to-node pinning are available in the electronic

Appendix. Overall, we observe that **all the conclusions presented in the paper still hold with per-node thread pinning**.

Impact of BIOS configuration. The experiments presented in this section were all ran with the BIOS configured in performance mode, for all machines. In performance mode: (i) processor throttling is turned off, so that all cores always run at full speed (i.e., maximum available frequency without Intel Turbo Boost / AMD Turbo Core), and (ii) idle power saving processor C-states are deactivated, thus cores are always immediately available to execute threads (i.e., they never need to be resumed from low-power mode). In addition, for the I-48 and I-20 machines, we also executed the throughput experiments with the BIOS configured in energy-saving move. In such a configuration, processor throttling and idle power saving C-states are activated, letting the hardware and the kernel manage the processors' state, aiming at reducing power consumption. We observe quantitative throughput differences between the two configurations. However, changing the BIOS configuration does not only affect lock performance but also application performance. As a consequence, a full study of the impact of the BIOS configuration modes on the performance of applications falls out of the scope of this article. Nonetheless, we observe that **all the conclusions presented in the paper still hold when the BIOS is configured in energy-saving move**.

5.4 Effects of the lock choice on application performance

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

Second, the Pthread library should **not provide only one lock algorithm (i.e., the Pthread lock algorithm) to software developers** as it is currently the case. It is a "good generic solution"; still **Pthread locks certainly do not bring the best performance for every application**.

Third, the research community should perform **further research on optimized lock algorithms**. Specifically, there is a need for **dynamic approaches** to lock algorithms that automatically adapt to the running workload and its environment (e.g., the machine, the possibly collocated workloads). Besides, previous work only focused on the lock/unlock API, while we observe that applications also stress trylocks, barriers and condition variables, thus future research needs to consider **complete locking APIs** (more details in Section 8). Finally, metrics other than throughput are becoming more and more important, and as a consequence, when designing a new lock algorithm, researchers should not only consider throughput, but **all performance metrics**, including latency and energy efficiency (as we will see in details in Sections 6 and 7).

6 STUDY OF LOCK ENERGY EFFICIENCY

In this section, we perform experiments on the I-48 and I-20 machines in order to evaluate the energy efficiency of the different lock algorithms. In Sections 5 and 7, we present the results for throughput and tail latency, respectively. We are interested in energy efficiency as defined by Falsafi et al. [36]: energy efficiency represents the amount of work produced for a fixed amount of energy and can be defined as *throughout per power* (abbreviated TPP thereafter, in $\frac{\#operations/second}{watt} = \frac{\#operations/second}{joule/second} = \#operations/joule$). Higher TPP represents better energy efficiency. As explained in Section 3.2, we use Intel's RAPL facility to measure the energy consumption of several components: cores, chip package and DRAM.

This section is structured as follows. First, Section 6.1 discusses the results of the energy-efficiency study. We also discuss the similarities and differences between performance and energy-efficiency

Table 14. Percentage of lock-sensitive applications for which the energy-efficiency gain of *opt nodes* over *max nodes* is at least 5% higher than the performance gain, at least 5% lower than the performance gain or between +5% and -5% of the performance gain (**I-48 and I-20 machines**).

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

observations drawn from the study. Next, Section 6.2 discusses and validates the POLY conjecture previously introduced by Falsafi et al. [36], stating that energy efficiency and throughput go hand in hand with locks.

6.1 Energy-efficiency lock behavior

For the sake of brevity, we do not describe all the individual results for energy efficiency, available in the electronic Appendix. Overall, we observe that **all the conclusions presented in the paper about throughput in Section 5 still hold with energy efficiency**. More precisely, we observe that: (i) 50% of the applications are lock-sensitive with respect to energy efficiency, (ii) the optimized number of nodes for many applications is lower than the max number of nodes, (iii) the energy-efficiency gap is often large between different kinds of locks, (iv) the impact of locks on lock-sensitive applications is moderate at *one node*, and very high at both *opt nodes* and *max nodes*, (v) no lock is among one of the bests for more than 83% of the lock-sensitive applications at *one node* and for more than 61% both at *max nodes* and *opt nodes*, (vi) there is no clear global performance hierarchy among locks, (vii) all locks are potentially harmful, both at *max nodes* and *opt nodes*, yielding sub-optimal energy efficiency for a significant number of applications, (viii) the lock performance hierarchy changes significantly according to the chosen number of nodes. We observe, similarly to performance, that the I-20 exhibits less pronounced trends than the I-48 machine. Compared to the four twelve-core NUMA sockets of the I-48 machine, the I-20 machine only has twenty cores, divided into two NUMA sockets. As a consequence, the *max node* configuration for the I-20 uses half the threads than the I-48. Thus, some bottlenecks leading to collapse when using a high number of threads are not observable on the smaller I-20 machine.

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

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

Table 14 shows the gain difference of *opt nodes* over *max nodes* between energy efficiency and throughput. **The gain between *opt nodes* and *max nodes* for energy efficiency is generally higher than the one for throughput.** We observe that on the I-48, the gain for energy efficiency is higher for at least half of the lock-sensitive applications, and the same for 32% of the lock-sensitive applications. Intuitively, for energy efficiency, wasting resources while waiting behind locks costs both in terms of throughput and wasted energy.

Table 15 shows the percentage of lock-sensitive applications where *opt nodes* is lower, the same or higher while considering energy efficiency w.r.t. throughput. On the I-48, 25% of the lock-sensitive

Table 15. Percentage of lock-sensitive applications for which *opt nodes* is lower, the same or higher for energy efficiency w.r.t. performance. We use a 5% tolerance margin, i.e., if the application performance at *opt nodes* is $N1$ and the energy efficiency at *opt nodes* is $N2$, and $N1 \neq N2$, we look the performance at $N2$ and the energy efficiency at $N1$, and if the performance or the energy-efficiency difference is lower than 5%, we consider that the application's *opt nodes* is the same for performance and energy efficiency. (**I-48 and I-20 machines**).

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

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

The results on the I-20 machine are similar (available in the electronic Appendix).

6.2 POLY

The POLY[21] conjecture introduced by Falsafi et al. [36] states that "energy efficiency and throughput go hand in hand in the context of lock algorithms". More precisely, POLY suggests that "locks can be optimized to improve energy efficiency without degrading throughput", and that "[the insights from] prior throughput-oriented research on lock algorithms can be applied almost as-is in the design of energy-efficiency locks". The POLY conjecture could explain why we observe similar trends between our performance and energy-efficiency results. In this section, our goal is to test this conjecture on a large number of lock algorithms and applications (the initial paper about POLY considered 3 lock algorithms and 6 applications).

Figure 4 shows the correlation between performance and energy efficiency. Figure 5 shows the detailed results at *one node* for each lock-sensitive application (results at *max nodes* for the I-48 and at *one node* and *max nodes* for the I-20 machines are available in the electronic Appendix). The energy efficiency (in TPP – throughput per power, see Section 6) and the throughput are normalized w.r.t. the best performing (resp. energy-efficient) lock for each *(machine, application, type, node)* configuration. Most data points fall on, or very close to a linear regression between the two variables (the blue diagonal line).

Based on Figure 4, Malth_STP and (to a lesser extent) MCS-TimePub are outliers. These two algorithms use complex load-control algorithms: (i) Malth_STP parks a subset of the threads, while the others always spin for a few cycles before acquiring the lock ; (ii) MCS-TimePub allows spinning threads to bypass parked ones). The "exotic" behaviors of these locks most probably explain why the throughput and the energy consumption are not so well correlated with respect to other locks. Besides, on Figure 5, MySQL and (to a lesser extent) SQLite are outliers. These are the only two applications launching thousand of threads, stressing heavily the Linux scheduler. We conjecture that the overhead of context switches (due to both lock parking and thread preemption) slightly breaks the correlation between throughput and energy.

To quantitatively assess the correlation between energy efficiency and performance, we compute the Pearson correlation coefficient (PCC). The PCC is the value of the slope of a linear regression between two variables: the closer to 1, the greater the correlation between the variables. Intuitively,

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

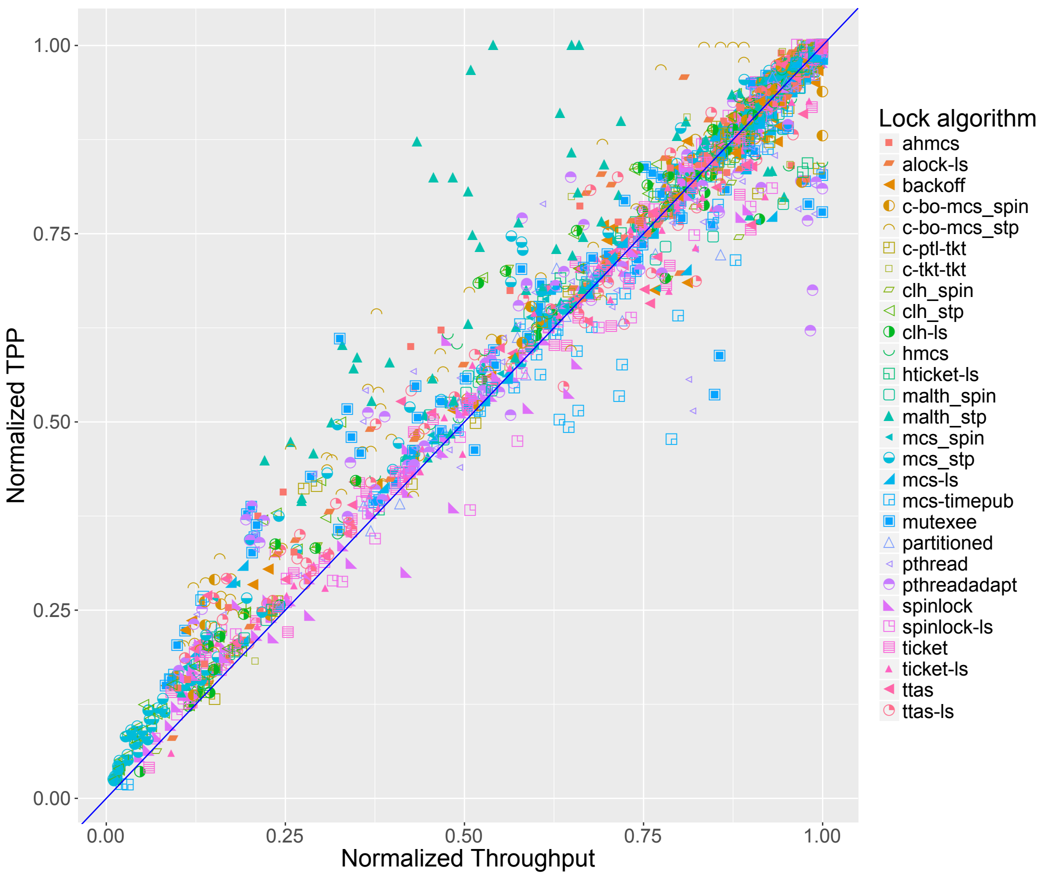

Fig. 4. Correlation of throughput with energy efficiency (TPP) on various lock-sensitive applications with various lock algorithms and various contention levels (**all machines**).

it quantifies the dispersion of the different configurations around the diagonal blue line. Table 16 shows the PCC on I-48 and I-20 for all the studied lock-sensitive applications. We observe that except MySQL that has a low PCC (0.55), all other configurations have a PCC at least equal to 0.87, which indicates a strong correlation between the performance and energy efficiency. More generally, **the PCC across all configurations (3.1k experiments) is 0.95**, an almost perfect correlation coefficient.

MySQL, upscaledb, Kyoto Cabinet and radiosity_ll have a PCC lower than 0.9. We observe that these four applications are highly contended. Looking at the detailed results, we observe that lock algorithms that use a parking waiting policy generally have a lower performance-to-energy-efficiency ratio (*PtE ratio* thereafter) than spinning algorithms. For example, for MySQL, algorithms using a fixed threshold for the spinning loop part of the spin-then-park waiting policy (e.g., C-BO-MCS_STP with a PtE of 0.89), have a lower PtE than algorithms that do adaptive spin-then-park (e.g., Mutexee with a PtE of 1.28), and even lower than algorithms that do spinning (e.g., MCS-TimePub[22]

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

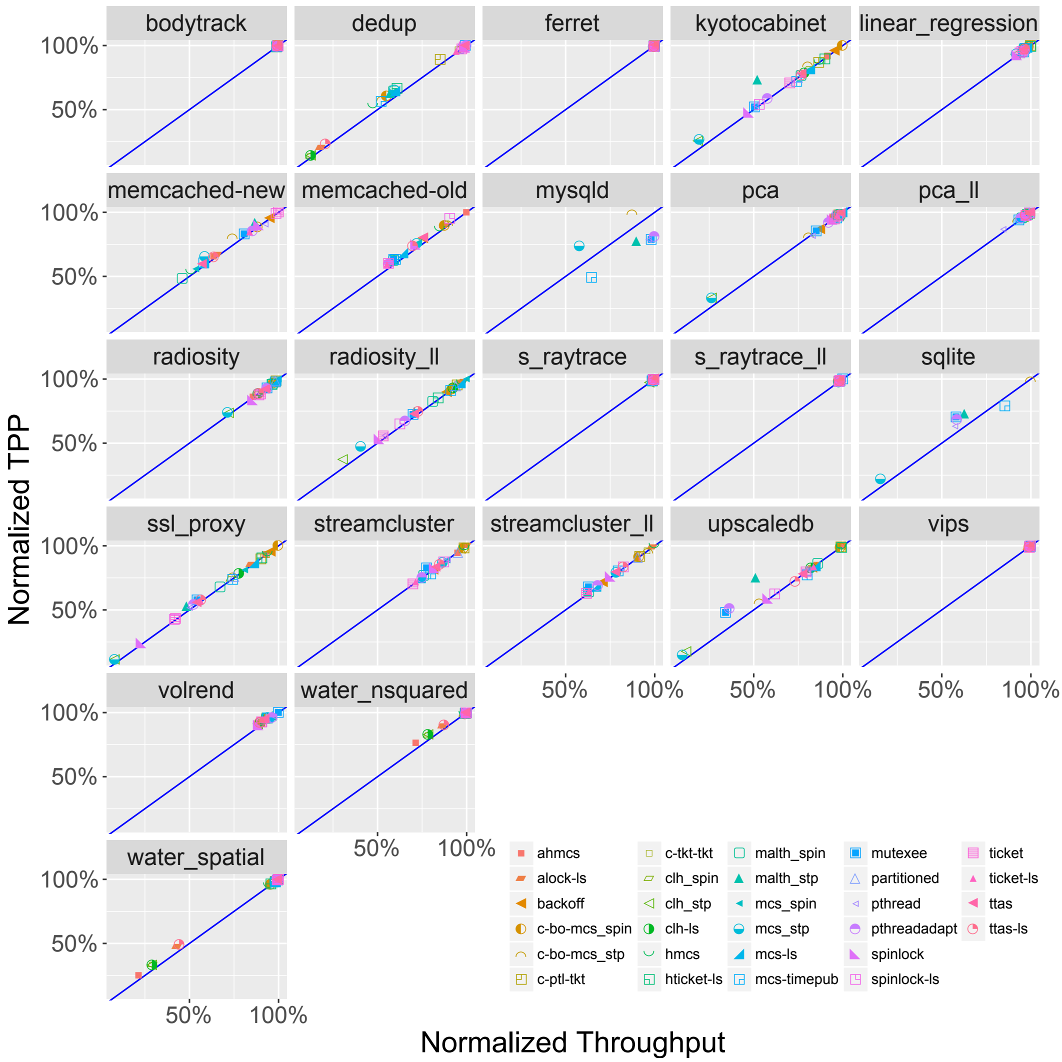

Fig. 5. Correlation of throughput with energy efficiency (TPP) on various lock-sensitive applications at *one node* for the different lock algorithms (**I-48 machine**).

with a PtE of 1.34). Intuitively, these results are expected, because at high levels of contention, parking locks can save energy compared to spinning, but spinning might still result in higher throughput [36].

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

Table 16. Pearson correlation coefficient between throughput and TPP for all lock-sensitive applications. Dashes mark applications that are not lock-sensitive (or not evaluated due to a lack of high-throughput network connectivity, see Section 3.1) on the I-20 machine. (**I-48 and I-20 machines**).

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

7 STUDY OF LOCK TAIL LATENCY

In this section, we are interested in the effect of lock algorithms on the application quality of service (QoS). More precisely, the QoS metric that we consider is the application tail latency, here defined as the 99th percentile of client response time. Note that in Sections 5 and 6 we discussed the results for throughput and energy efficiency, respectively. Understanding the relationship between throughput and tail latency allows us to understand, for example, if some lock properties (i.e., the fairness of FIFO locks) that improve the tail latency of lock acquisitions indeed improve the application tail latency. This analysis also enables us to understand which locks to choose to improve the tail latency of an application, sometimes at the (controlled) expense of throughput.

To perform this analysis, we capture the 99th percentile of the client response time on the A-64 machine for the seven server applications among the lock-sensitive applications that we have studied: Kyoto Cabinet, Memcached-new, Memcached-old, MySQL, SQLite, SSL Proxy, upscaledb. We further captured throughput and energy-efficiency metrics. Note that, as we discuss in Section 6.2, throughput and energy efficiency are correlated, thus we do not clutter the plots with energy-efficiency information and only show throughput. We have also performed the same experiments on the I-48 machine (our largest Intel multicore machine) and made similar observations as the ones described hereafter for the A-64 machine.

Figure 6 reports for each application and each lock algorithm at *opt nodes* the normalized (w.r.t. Pthread) 99th tail latency, as well as the normalized (w.r.t. Pthread) execution time (black squares). The results at *one node* and *max nodes* are available in the electronic Appendix. Locks are sorted by increasing tail latency. Note that we plot execution time (rather than throughput) so that "lower is better" for both displayed metrics (latency and execution time). However, in the text we talk about throughput (as the inverse of the execution time) for homogeneity with the other sections.

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

At *max nodes*, the maximum tail latency is generally higher than at *opt nodes* and *one node*. For example, for Kyoto Cabinet, at *max nodes*, the tail latency of CLH_STP is 5× higher than Pthread, while it is of roughly 1.6× higher than Pthread at *one node* and *opt nodes*. The tail latency skyrockets at *max nodes*: locks suffer from extreme levels of contention and threads wait for a long time to acquire locks. On average, when increasing the number of threads (from *one node* to *max nodes*), the request execution time increases 3.3× and the tail latency increases 22.9×. Similarly, from *opt nodes* to *max nodes*, the request execution time increases 3.4× and the tail latency increases 21.0×. The experiments with a single thread for all the studied applications except MySQL and SQLite[23] are available in the electronic Appendix. Overall, we found that, on the studied applications with a single-threaded configuration, the choice of a lock has very little effect on the throughput or the tail latency of the application.

7.2 Do fair lock algorithms improve the application tail latency?

On the one hand, FIFO locks (cf. Section 2.1) promise fairness among threads acquiring a lock. On the other hand, unfair locks might increase tail latencies by letting some threads wait for long durations before acquiring the lock. Interestingly, we observe that fairness affects the tail latency for only two applications: Kyoto Cabinet and upscaledb. For them, we observe low tail latency with almost all FIFO locks. Moreover, all hierarchical locks, which by design do not strictly impose fairness, exhibit roughly the same tail latencies, which are higher than the tail latencies of FIFO locks. Still, for the four other studied applications, we do not observe a correlation between lock fairness and application tail latency.

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

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

7.3 Do lock tail latencies affect application throughput?

Some lock algorithms explicitly try to trade fairness for higher throughput. For example, hierarchical locks prefer to give a lock to a thread on the same NUMA node than to a thread executing on another node. Interestingly, in practice, we observe that this property, which directly affects tail latency and throughput of lock acquisitions, effectively affects the application tail latency and throughput for only two applications: upscaledb and Kyoto Cabinet. For these applications, we generally observe that hierarchical locks lead to higher tail latency and higher throughput. For example, for upscaledb at *opt nodes*, increasing the tail latency from 100 μs to 1000 μs increases the throughput by 26% (using MCS vs. HMCS). Using Ticket and C-TKT-TKT on Kyoto Cabinet, at *opt nodes*, increasing the tail latency by 3×, leading to a 33% throughput increase. At *max nodes*, Mutexee exhibits 80% higher tail latency than Pthread, but improves throughput by 60%. Applications where the tail latency is affected by the lock fairness property of some locks (§7.2) are the same applications that are affected by the fairness/throughput tradeoff property.

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

Among the 7 server applications for which we studied tail latency, we obtained unexpected results for Memcached-old. This application is known to suffer from extreme levels of contention (see Section 8): the main bottleneck is a single global lock serializing most requests. One might expect that lock properties should directly affect the application throughput and tail latency. However, Memcached-old uses the trylock operation to acquire a lock. Interestingly, most of the lock algorithms have been designed to optimize the lock/unlock operation, not the trylock one, and in practice, there is no such thing as a “fair trylock”, even for locks that promise FIFO lock acquisitions.

7.4 Implications

Contrary to throughput (see Section 5.2), studying tail latency allows us to draw simpler conclusions, as the results are more stable across applications and machines. We observe two groups of applications that behave differently regarding tail latency.

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

In contrast, for applications **with “larger” operations that consist of many critical sections and/or the critical sections are protected by different lock instances accessed by different threads, the tail latency of locks does not necessarily affect the application tail latency**. For such applications, a developer should choose a lock that best improves the application throughput: the tail latency improvements will follow.

Interestingly, we observe in our set of studied applications that software developers use the trylock operation to implement busy waiting, while the original operation is designed to allow a developer to write a fallback code if the locking attempt fails. Because the trylock is only a one-shot attempt to acquire a lock, there is actually no lock algorithm that provides a fair trylock. We believe that developers use trylocks this way because the default Pthread lock operation is blocking: a developer knows when a critical section is short, and thus would like to avoid the overhead of a thread blocking if the lock is unavailable. Pragmatically, the trylock operation should not be used this way, but this demonstrates the need to extend the Pthread lock API with a **lock operation informing the lock algorithm that a thread should busy wait and not block, e.g., `pthread_mutex_busylock`**[24].

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

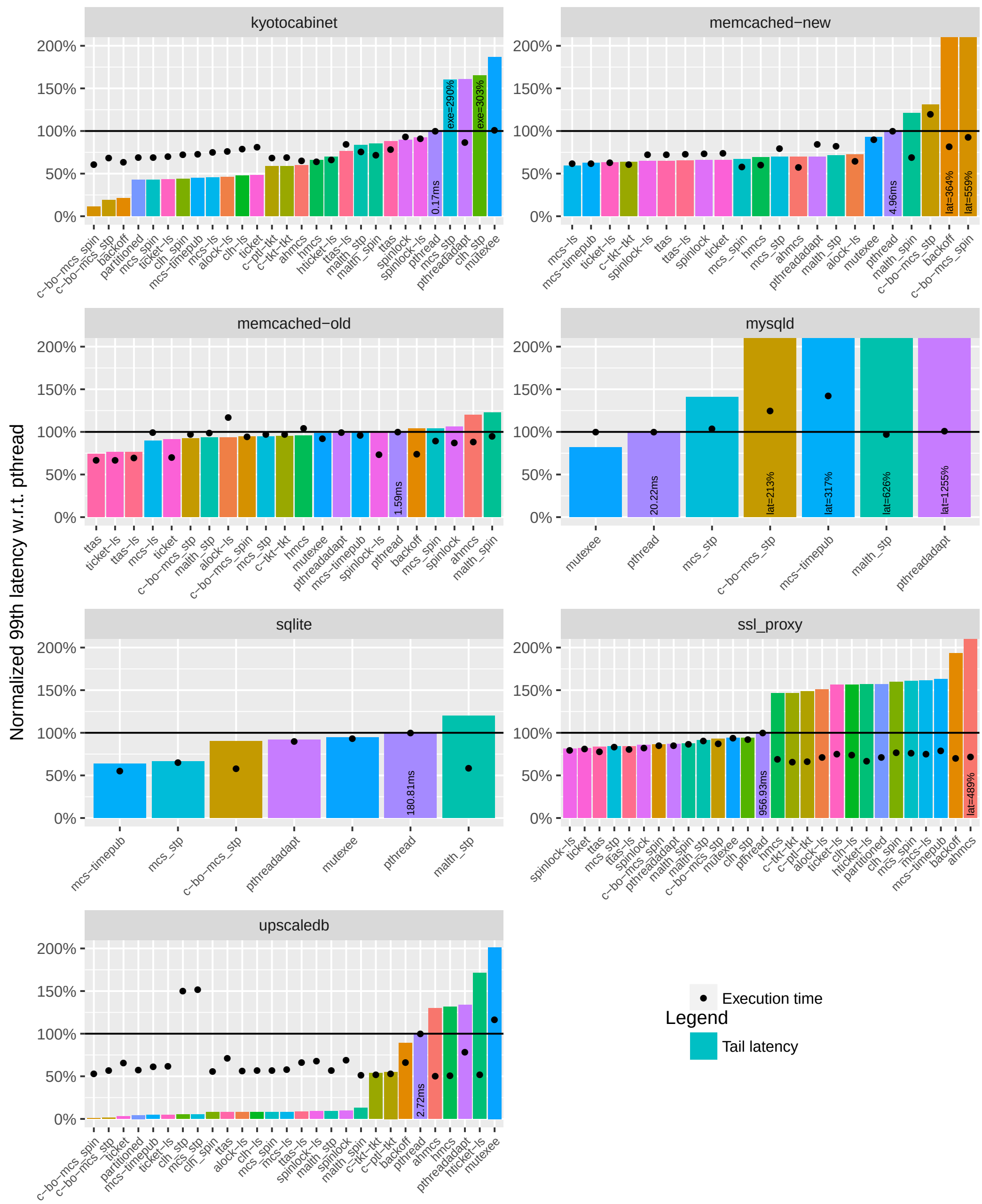

Fig. 6. For each server application, the bars represent the normalized 99th tail latency (w.r.t. Pthread) and the dots execution time (lower is better) normalized (w.r.t. Pthread) of each lock algorithm (**A-64 at *opt nodes***).

8 ANALYSIS OF LOCK/APPLICATION BEHAVIOR

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

In Section 8.1, we give general insights that we draw from our analysis by presenting, for every application, the performance bottleneck it suffers from, and which lock(s) to prefer or to avoid when running it. We found that, beyond the pure performance of a lock algorithm under high contention, different applications stress different aspects of a lock algorithm (e.g., memory footprint, scheduler preemption tolerance). In Section 8.2, we present seven *properties* shared by the studied lock algorithms, which, when cross-referenced with the performance bottlenecks of an application and a set of general guidelines that we provide, can help a developer to predict whether a lock algorithm performs well or poorly on a given application.

Note that the above-mentioned analysis was performed on the A-64 machine, and was performed with the aim to find the main (lock-related) performance bottlenecks. For each bottleneck, we explain if it is more common at *opt nodes* or *max nodes*. Nonetheless, the observations made in Sections 5 and 6 are not specific to lock performance on the A-64 machine. Thus, we think that the conclusions of this Section can be applied to different machines, and not only to throughput but also to energy efficiency.

8.1 Summary of the lock/application behavior analysis

In this section, we give general insights that we draw from the detailed analysis of the different lock-sensitive applications. Table 17 lists, for each lock-sensitive application its main performance bottleneck with respect to locking (in column 2). We also recommend which family of lock algorithms (i.e., lock algorithms sharing a similar property) to prefer or avoid for each of the studied applications (detailed in Section 8.2.1). For example, we observed that the performance bottleneck of fluidanimate is due to a high number of uncontended lock acquisitions. As a consequence, it is better to use a light lock algorithm, i.e., a lock that can be acquired very quickly when there is no other thread trying to acquire it at the same time (e.g., with only one atomic CPU instruction). Overall, we identified 9 performance bottlenecks across 22 applications, that can be summarized into four categories: lock contention, scheduling issues, memory footprint and memory contention.

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

High levels of contention. A high number of threads (between approx. 10 to 40 threads on A-64) are waiting to acquire the same lock instance at the same time. To measure the contention level on a lock, we take regular snapshots of the application state, looking at how many threads are currently

Table 17. Lock-sensitive application performance bottleneck(s) and lock algorithms choice advice.

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

waiting for a lock. More precisely, each time a thread requests a lock, it puts the lock address inside a private cache-aligned memory location, and all such locations are read by a background thread every second. This provides us with a low-overhead approximation of the real number of threads waiting for a lock, with respect to a more straightforward approach where a counter is atomically incremented before waiting for a lock and decreased when the lock is acquired. Radiosity, linear_regression, s_raytrace, s_raytrace_ll are the four lock-sensitive applications that suffer from this performance bottleneck.

Radiosity is parallelized using per-core distributed task queues, where each thread can steal work from another task queue. Radiosity allocates a large number of locks (4k); still only two locks are highly contended. With HMCS, one of the best locks, on average, 60% of all the total threads wait on one of the two stressed locks, while there is virtually no contention on the other 4k locks. For linear_regression, we observe that there is only one lock inside the application that protects a distributed task queue. This lock suffer from high levels of contention (65% of the threads waiting on the lock). S_raytrace and s_raytrace_ll render a 3-D scene partitioned among threads and there is a global task queue protected by a single lock. Still, the contended lock is not the global task queue lock, but a lock protecting a single counter used to implement a global unique identifier generator. For the short-lived version (resp. for the long-lived version), on average, 40% (resp. 60%) of the threads are waiting for the same lock (using HMCS, one of the best lock algorithms). When using an atomic `fetch_and_add`, we observe a 1.8× (resp. 3×) performance improvement for the short-lived version.

For high levels of contention, lock algorithms that rely on local spinning (e.g., MCS) or on a hierarchical approach (e.g., AHMCS) are well suited (see Section 2.1). Light lock algorithms (e.g., Spinlock) and lock algorithms using a parking waiting policy must be avoided when possible.

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

Radiosity_ll, the long lived version of radiosity, also suffers from lock contention. Contrary to the short lived version, radiosity_ll puts more pressure on the locks[25]. Volrend suffers from lock contention on the lock instances protecting different distributed task queues, as well as on a lock instance used to implement a barrier that separates the computation steps. These task queue locks (as well as the barrier lock) suffer from extreme levels of contention, especially the barrier lock that suffers from spikes of contention when all the threads wait for the barrier at the same time. MySQL suffers from lock contention on a lock that protects the page cache, a data structure that serves as an in-memory cache for the SQL table data stored on disk. This lock is heavily stressed: we observe on average 50 threads (on a 64-core machine) competing for the same lock instance, resulting in 40% of the thread lifetimes spent waiting to acquire this lock. The SSL Proxy application implements a reverse SSL proxy using OpenSSL via the Boost ASIO library. This application is subject to a huge performance collapse: the optimized number of nodes is one. In this application, the main bottleneck is a lock protecting the error queue of OpenSSL, which suffers from extreme levels of contention (on average 85% of the threads wait on the same lock). Similarly to Zemek [97], we found that the problem comes from an inefficient usage of the OpenSSL library by the Boost ASIO library. Indeed, the original lock that the OpenSSL library requests is a reader-writer lock; still Boost ignores it and uses a classic mutex lock, lowering the potential degree of parallelism. Kyoto Cabinet is a straightforward implementation of a database. As explained by Afek et al. [1], the most contended

[25]The short-lived version is launched with a BF [45] refinement epsilon of $(1.5e-3)$ and the long-lived version is launched with a BF refinement epsilon of $(1.5e-5)$. With a lower epsilon, computations are refined more frequently, creating more tasks.

lock instance is the lock protecting the global hash table storing the data. Indeed, all database operations (create/insert/update/delete/lookup) need to acquire the same lock, which becomes highly contended. Upscaledb is an in-memory key/value store tailored for efficiency of analytical functions. Contrary to popular database engines like InnoDB for MySQL that use fine-grained locking (generally one lock for a row/set of rows), upscaledb uses only one lock instance to protect the whole database. Such a poor design choice explains why upscaledb does not scale: indeed we observe that all of the threads spend 98% of their execution time waiting for the lock.

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

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

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

Among the studied algorithms, only a few algorithms (HMCS, cohort locks, Partitioned and MCS-TimePub) implement a trylock operation performing well under high levels of contention. For example, the HMCS and the cohort locks implement a trylock in a hierarchical manner, leading to better performance on NUMA machines[26].

Many uncontended lock acquisitions. One of the applications (fluidanimate) creates a large number of lock instances (500k locks). These locks are used to protect each cell of the grid, and are only

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

used by one or two threads at the same time: most of the time a thread acquires the lock without any competition. More precisely, fluidanimate calls `pthread_mutex_lock` 5 billions times and half of the acquisitions are immediate, while for the other half a thread waits only because there is another thread inside the critical section, never because there are other waiting threads.

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

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

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

Lock holder preemption. The *lock holder preemption* effect is a well-known issue [14] with lock algorithms using a spinning waiting policy. It happens when a thread A waiting for a lock instance preempts a thread B that is the lock holder. Doing so, A runs on a core waiting for B to release the lock instance, while the rescheduling of B is delayed because of A, thus delaying B to finish the critical section, and release the lock instance for A. This pattern is highly inefficient. In the worst scenario, this can lead to lock convoy: while the lock holder is descheduled, each thread progresses and eventually tries to acquire the lock instance, spinning, thus delaying the rescheduling of the lock holder. This issue is usually observed in highly-threaded applications, where the scheduler has to frequently decide which thread to run on which core. This effect is more likely to be seen at *max nodes*; still some applications are already highly-threaded at *opt nodes* (e.g., MySQL and SQLite). Note that all kinds of spinning algorithms are affected by this phenomenon: the simplest ones (e.g., TTAS), FIFO (e.g., MCS_Spin) and hierarchical approaches (e.g., HMCS). In fact, lock holder preemption is mainly a property of the program concurrency-design, not the lock design. The lock holder is more likely to be preempted inside critical sections with applications composed of long critical sections and that over-subscribe threads to cores (e.g., databases).

MySQL and SQLite are two highly-threaded applications suffering from the *lock holder preemption* effect. MySQL uses a large thread pool (hundreds of threads) to handle queries from clients. SQLite creates a server that listens for client requests on a Unix socket and uses a globally shared work queue protected by a single lock instance; still many other lock instances are used to synchronize internal data structures. The benchmark used (see Section 3.1) creates hundreds of threads.

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

Lock handover. This phenomenon (also known as the *lock waiter preemption problem* [85]) happens with algorithms that use a direct handoff succession policy (see Section 2.1.2). When a thread waiting in line for a lock is preempted, all other waiting threads after this one are delayed. Worse, these threads spinlock their entire timeslice, postponning the rescheduling of the descheduled thread. In principle, this problem is unlikely to appear on platforms that do not use more threads than cores. In practice, lock waiter preemption actually occurs quite often even when there are never more threads than cores. Indeed, the Linux CFS scheduler sometimes migrates two (or more) threads on the same core, thus leading to situations where the next-acquiring thread is preempted, and where other waiting threads spin uselessly. These migrations are mainly observed when there are many blocking calls inside the application (e.g., condition variables, I/O). This phenomenon is more likely to happen at *max nodes*.

There are six of the lock-sensitive applications that suffer from the *lock handover* effect: facesim, ferret, vips, ocean_cp and ocean_ncp, streamcluster. Facesim creates one thread per core that implements a fork-join computation model [13]. The applications uses a barrier to synchronize the successive fork-join phases, implemented with a mutex lock and a condition variable. When threads wait on the condition variable, they might be migrated by the scheduler so that when they are unblocked (i.e., when leaving `pthread_cond_wait`) they are scheduled on the same core. There are 10× more migrations for a poor performing lock algorithm (MCS, 40k) than for the MCS-TimePub lock algorithm (4k): with a poor performing lock, threads have more chances to share the same core. Note that a straightforward solution to “fix” facesim is to pin each thread to a distinct core, thus avoiding inefficient migrations. For example, with MCS pinning improves performance and yields roughly the same results as MCS-TimePub, one of the best performing locks.

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

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

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

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

For applications suffering from the *lock handover* effect, FIFO algorithms using a waiting policy based on pure spinning (e.g., Ticket, MCS) should be avoided in such cases.

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

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

Applications that need to control their memory footprint can benefit from dynamically allocating per-node data structures of hierarchical locks upon first access [55]. It benefits to applications where locks are in fact rarely acquired by threads from multiple NUMA nodes. However, it leads to more dynamic allocations to be made, which might introduces kernel lock contention inside the page fault handler (see below).

Kernel lock contention inside the page fault handler. On some applications, at both *opt nodes* and *max nodes*, all threads are constantly creating new lock instances, putting pressure on the memory allocator (i.e., `malloc`). Internally, `malloc` requests pages of memory to the kernel (via `brk` and `mmap`), which generates page faults when the pages are first accessed. The page fault handler tries to insert the new page into a process-shared data structure (the virtual address space data structure), protected by a single reader/writer lock [23]. The contention on this kernel lock becomes more performance critical than the one on the application-level locks, because all threads need exclusive write access to the data structure, and the lock is generally kept for a long time.

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

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

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

Pca (and its long-lived version pca_ll) is a good example of such a phenomenon. Validating the observations on pca from the original paper [78], we found that pca suffers either from lock contention (for algorithms that do not support high levels of contention, e.g., Spinlock) or memory controller saturation[27] (for the others, e.g., Pthread). For example, with Pthread (pca suffers from memory controller saturation), we observe a 44% performance increase when we interleave the memory pages of the application, i.e., when the memory pages of the application are allocated in a round-robin fashion on all the NUMA nodes of the machine. This is a clear indicator that, without interleaving, the memory controller of one NUMA node becomes overwhelmed, receiving too many requests from all the threads. Besides, even with interleaving, the memory bottleneck does not fully disappear. Indeed, we observe an increase from 0.4 stalled cycles per instruction (SPI) outside locking primitives with Malth_Spin (one of the best locks) to 2.25 SPI with MCS[28] (a bad performing lock). However, note that the stalled cycles are observed inside the parallel code sections of pca. By being somewhat "too" fast, MCS allows many threads to run in parallel, thus increasing the memory contention of the parallel code sections of pca. More precisely, the number of stalled cycles due to memory accesses, which account for 98% of all stalled cycles, is 20× higher with MCS than with Malth_Spin. Note that this phenomenon is more likely to appear at *max nodes*, because memory contention exists when a large number of threads access memory concurrently.

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

8.2 Guidelines for lock algorithms selection

In Section 8.2.1, we describe the different properties of the studied lock algorithms, and in Section 8.2.2 we discuss guidelines to help a developer choosing a lock algorithm for a given application.

8.2.1 Lock properties. Knowing the performance bottleneck of an application, a developer can now decide which lock algorithms to use in an application. Table 18 summarizes the main properties of each lock algorithm. Overall, we identified seven properties shared by the studied lock algorithms that have an impact on performance. We also describe how the different design properties described in Section 2.1 are related to these "behavioral" properties. We first present properties related to

[27] While experimentally assessing the performance overhead of LiTL (see Section 4.4), we noticed a corner case with pca. More precisely, we observe that, most of the time, LiTL improves the performance w.r.t. the manually implemented version. This performance difference comes from the condition variable algorithm of LiTL that lengthens the critical section. Indeed, as pca and pca_ll suffer from memory contention, longer critical sections lower the number of threads running in parallel outside the critical sections, thus improving performance. However, the best locks with LiTL are also among the best manually implemented locks.

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

Table 18. Lock algorithm properties. The algorithms are grouped by categories as defined in Section 2.1.2. For example, ahmcs does not use a parking waiting policy, nor does it have a low memory footprint. However, it is a hierarchical lock algorithm. Some lock algorithms do not support the trylock operation and thus cannot be run with applications that use this operation: we denote these cases by a cross sign.
☐ locks without the property ■ locks with the property x trylock not supported

	light	hierarchical lock	contention-hardened trylock	parking	FIFO	low memory footprint	low memory (interconnect) traffic
Competitive							
backoff	■					■	■
mutexee	■			■			
pthread	■			■		■	
pthreadadapt	■			■		■	
spinlock	■					■	
spinlock-ls	■					■	
ttas	■					■	
ttas-ls	■						■
Direct handoff							
alock-ls					■		
clh-ls			x		■		
clh_spin			x		■		
clh_stp			x	■	■		
mcs-ls					■		
mcs_spin					■		
mcs_stp				■	■		
partitioned	■		■		■		
ticket	■				■	■	
ticket-ls	■			■	■	■	
Hierarchical							
c-bo-mcs_spin		■	■		■		
c-bo-mcs_stp		■	■	■	■		
c-ptl-tkt		■	■		■		
c-tkt-tkt		■	■		■		
hmcs		■	■		■		
hticket-ls		■	x		■		
Load-control							
ahmcs		■			■		
malth_spin					■		■
malth_stp				■	■		■
mcs-timepub			■	■			

different levels of contention, then properties that can affect scheduling, and finish with properties related to memory.

(1) *Light:* lock algorithms having a short code path to acquire the lock when uncontended. Algorithms such as Spinlock, Backoff or TTAS have this property, where an uncontended lock acquisition is almost only an atomic instruction. Algorithms using a context such as MCS or CLH are generally heavier, because they need to setup the context before acquiring the lock, even if there is no contention. We also observe that there is no hierarchical lock that is light: cohort lock algorithms acquire both local and global locks, and even AHMCS, which implements a fast path; still needs to acquire one uncontended MCS lock. Finally, all existing load-control lock algorithms are heavy, because the load control decision is on the critical path.
Note that for applications where a single thread acquires a lock, biased locking [33] can improve performance. This technique can be used to enhance any lock algorithm with an atomic-free fast path, and switches to the default lock algorithm upon the first lock acquisition by a second thread.
(2) *Hierarchical lock:* lock algorithms designed to take into account NUMA architectures, where the cost of accessing a lock instance from a different socket is higher than the one when the lock instance is already inside a cache of the local socket. This category is the same category as described in section 2.1.2.
(3) *Contention-hardened trylock:* lock algorithms with a trylock operation tolerating moderate to high levels of contention. We observe that some applications use the trylock operation to do busy-wait, i.e., the trylock operation is continuously called in a loop until the lock is acquired. In practice, a large number of atomic instructions are executed concurrently, flooding the memory interconnect with cache-coherence traffic. Here, lock algorithms that lower the cache-coherence traffic are the ones that perform the best. We observe that hierarchical locks have a contention-hardened trylock, because a thread needs to trylock both the local and the global lock[29]. We also observe that algorithms like MCS-TimePub and Partitioned have a contention-hardened trylock because their trylock operation takes time (i.e., the operation consists of one atomic instruction and a significant number of non-atomic instructions), thus lowering the cache-coherence traffic.
(4) *Parking:* lock algorithms using a spin-then-park or a direct parking waiting policy (see Section 2.1.3).
(5) *FIFO:* lock algorithms imposing an order on the acquisitions of a lock instance according to the thread arrival times, i.e., if a thread *A* tries to acquire the same lock instance as *B* before *B*, *A* enters the critical section before *B*. Note that some lock algorithms leave some degree of freedom regarding this order, i.e., a thread might enter the critical section before another thread that had been waiting for a longer amount of time (e.g., with the cohort lock algorithms that favor threads running on the same socket as the lock holder). This category regroups a subset the lock algorithms using a direct handoff succession policy (see Section 2.1.2).
(6) *Low memory footprint:* lock algorithms having a low memory footprint. All locks that need a context (e.g., MCS, CLH, Malthusian have a high memory footprint, because each thread needs its own context. Besides, hierarchical lock algorithms also have a high memory footprint because one lock instance is composed of one top lock instance, and one instance per NUMA node, but the footprint can be lowered by dynamically allocating per-node data structures of hierarchical locks upon first access [55].
(7) *Low (memory) interconnect traffic:* lock algorithms that only induce a moderate traffic on the memory interconnect of the machine. Algorithms using a load-control mechanism sensitive to the concurrency level (e.g., Malthusian) reduce the number of threads running

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

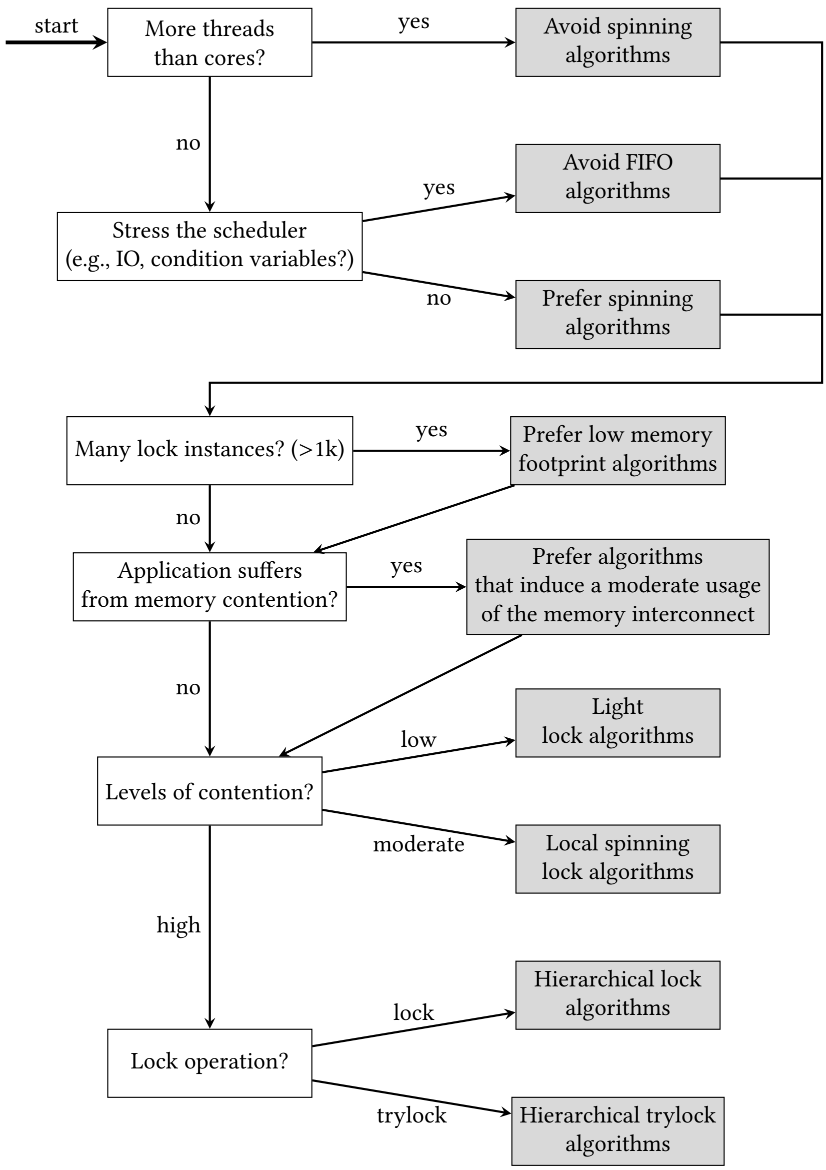

Fig. 7. Steps to follow for the application developer to chose a lock algorithm.

concurrently, thus the pressure on the memory interconnect. Surprisingly, lock algorithms that perform both poorly under contention *and* which do not flood the interconnect with cache-coherence messages (e.g., Backoff, TTAS-ls) are good choices to lower the memory interconnect utilization.

8.2.2 Choice guidelines. Figure 7 shows a series of steps to follow in order to select which lock algorithm to use with each application. The steps are questions the developer needs to answer that help select a small subset of lock algorithms. A box with a white background represents a question and a box with a gray background suggests the developer to select or avoid some locks.

For example, for upscaledb, the developer starts by asking if the application has more threads than cores. Upscaledb does not have more threads than cores. Next, the application is profiled to know if it performs many calls to the scheduler (e.g., with I/O, conditions variables), which might lead to thread migrations. Upscaledb does not call the scheduler often, so the developer can still consider FIFO algorithms. Moving forward, upscaledb does not create many lock instances, does not use the trylock operation and does not suffer from memory contention. We are now at the last step, where the developer has to chose a lock algorithm regarding the levels of contention the lock instances inside upscaledb suffer from. Remember that because upscaledb does not have more threads than cores, and does not call the scheduler often, the developer should choose an algorithm that uses a spinning waiting policy. We observe that upscaledb suffers from extreme levels of contention. Therefore the developer should choose a hierarchical spinning lock algorithm, for example AHMCS.

A word of caution: these guidelines are cursory, because carefully tuning a lock algorithm is highly dependent on a given workload and machine. They give a hint to the developer for the choice of a lock, and mostly target applications in which lock access patterns are stable (e.g., the most contended lock is always the same and it always suffers from a constant level of contention). Many lock bottlenecks can be suppressed by redesigning the application with smaller critical sections, or by using more scalable synchronization primitives, such as lock-free data structures. Besides, some techniques enhancing lock algorithms (e.g., lazy lock allocation [55], biased locking [33]) can be beneficial to adapt a given lock that is not initially the best for a given workload. Finally, for applications where the access pattern of a lock varies during the workload, adaptive lock algorithm such as GLK [9] can be used.

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

9 RELATED WORK

There is a large body of work studying different aspects of lock algorithms. This section is organized as follows. Section 9.1 presents work studying the implementation of lock algorithms, and previous approaches to transparently replace lock algorithms inside applications. Section 9.2 discusses the possibility to dynamically adapt lock synchronization at run-time. Section 9.3 considers previous studies of multicore lock algorithms. Section 9.4 covers existing works that highlight the importance of energy efficiency for both applications and lock algorithms. Finally, Section 9.5 discusses lock-related performance bottlenecks.

9.1 Lock algorithm implementations

The design and implementation of the LiTL lock library borrows code and ideas from previous open-source toolkits that provide application developers with a set of optimized implementations for some of the most-established lock algorithms: Concurrency Kit [4], liblock [61–63], libslock [27] and lockin [9, 36]. All of these toolkits require potentially tedious source code modifications in the target applications, even in the case of algorithms that have been specifically designed to lower this burden [10, 83, 93]. Moreover, among the above works, none of them provides a simple and generic solution for supporting Pthread condition variables. One noticeable exception is lockin [9, 36], which only requires including a header inside the source code of the application and recompile it linked against a specific shared library. lockin also proposes a condition variable algorithm; still the proposed algorithm does not circumvent the “thundering-herd” effect for all lock algorithms (see

Section 4.1). The authors of liblock [63] proposed an approach to support condition variables; still we discovered that it suffers from liveness hazards due to a race condition (see Section 4.1). Indeed, when a thread T calls `pthread_cond_wait`, it is not guaranteed that the two steps (releasing the lock and blocking the thread) are always executed atomically. Thus, a wake-up notification issued by another thread might get interleaved between the two steps and T might remain indefinitely blocked.

Several research works have leveraged library interposition to compare different locking algorithms on legacy applications (e.g., Johnson et al. [52] and Dice et al. [32]). However, to the best of our knowledge, they have not publicly documented the design challenges to support arbitrary application patterns (e.g., condition variables), nor disclosed the corresponding source code and the overhead of their interposition library has not been discussed.

9.2 Adaptive algorithms

Previous works discuss the possibility to dynamically adapt lock synchronization at run-time. One way is to dynamically switch between lock algorithms depending on the contention level. The work by Lim et al. [60] considers switching among three lock algorithms (TTAS, MCS and a delegation-based one), depending on the level of contention on the lock instance. SANL [98] switches between local and remote (i.e., delegation-based) locking schemes. As explained in Section 2, delegation-based algorithms require critical sections to be expressed as a form of closure, which is incompatible with our transparent approach (i.e., without source code modification). More recently, Antic et al. [9] proposed GLS, a solution that dynamically switches among three lock algorithms (Ticket, MCS, Pthread mutex), using Ticket at low contention levels, MCS at high contention levels, and Pthread when it detects overthreading (i.e., more threads than cores). While these approaches confirm our observations that there is no one-size-fit-all locking algorithm, their goal is to make locking easy for a developer, not to choose the best lock algorithm in all cases. Indeed, they only switch among a few different lock algorithms, whereas, in light of our study, there are more lock algorithms to consider, making the choice more complex. None of the solutions considers some of the bottlenecks that we observed, like trylock contention, the lock handover effect and bottlenecks related to the memory footprint of a lock instance. For example, all solutions embed all the different lock data structures into a unique one, inflating the memory layout of a lock instance: an application like dedup (using thousands of lock instances) that is good with a classical low memory footprint Ticket algorithm might not be good with the Ticket version of GLS, even if GLS never uses lock algorithms other than Ticket.

A second solution is to monitor the load pattern of the application to detect situations that are subject to pathological behavior. Load control (LC) [52] is a runtime solution, which dynamically reduces the number of threads trying to acquire the lock at the same time, to avoid pathological issues (e.g., lock convoy). LC requires kernel modifications on Linux to measure load accurately and with high resolution ($\sim 100\mu s$). This approach is thus incompatible with our work, where we focus on lock algorithms that do not require code modifications. Overall, our work highlights the need for low-memory, complete interface (i.e., lock, trylock, and condition variables), fully adaptive (i.e., from spinlocks all the way to complex HMCS locks) lock algorithms.

9.3 Studies of synchronization algorithms

Several studies have compared the performance of different multicore lock algorithms, from a theoretical angle and/or based on experimental results [8, 15, 27, 32, 56, 61, 70, 83]. Our study encompasses significantly more lock algorithms and waiting policies. Moreover, the bulk of these studies is mainly focused on characterization microbenchmarks, while we focus instead on workloads designed to mimic real applications. Two noticeable exceptions are the work from Boyd-Wickizer et al. [15] and Lozi et al. [63]; still they do not consider the same context as our study. The former is focused on kernel-level locking bottlenecks, and the latter is focused on applications in which only one or a few heavily contended critical sections have been rewritten/optimized (after a profiling phase). For all these reasons, we make observations that are significantly different from the ones based on all the above-mentioned studies.

Some related work discusses the choice of synchronization paradigms and lock algorithms [67–69]. The proposed guidelines are often a subset of our proposed guidelines in Section 8.2.2: because these works only study a smaller set of applications and lock algorithms, they generally do not cover all the cases we observed.

Other synchronization-related studies have a different scope and focus on concurrent data structures, possibly based on other facilities than locks. Gramoli [42] studies different concurrent data structures on micro-benchmarks with multiple synchronization techniques. David el al. [26, 28] evaluate theoretical and practical progress properties of concurrent search data structures. Brown et al. [17] study the performance of hardware transactional memory with microbenchmarks on modern NUMA multicore machines. Finally, Calciu et al. [18] study the tradeoff between message passing and shared memory synchronization on multicore machines. Similarly to us, they advocate that software should be designed to be largely independent of the choice of low-level communication mechanism.

9.4 Energy efficiency

Improving energy efficiency in systems and applications has been thoroughly studied in the past. For example, previous works describe user-level [71, 80, 86, 87, 95, 96] and kernel [75] facilities that both manage and predict power consumption. Prior works propose trading performance and/or precision for energy. For example, programming models [11, 82] allow developers to approximate loops to decrease power consumption. Compiler techniques [94, 95] and hardware mechanisms [57] trade off performance for energy. To the best of our knowledge, the work by Falsafi et al. [36] is the only one studying the energy efficiency of lock algorithms. We confirm their findings and validate their POLY conjecture on significantly more lock algorithms and applications.

9.5 Lock-related performance bottlenecks

Some tools have been proposed to facilitate the identification of locking bottlenecks in applications [9, 25, 63, 76, 91]. These tools are useful to identify which lock instances suffer from contention; still they do not help a software developer to choose a lock algorithm for an application. The proposed tools are orthogonal to our work. We note that, among them, the profilers based on library interposition could be stacked on top of LiTL.

Finally, lock-related performance bottlenecks have been previously analyzed. For example, many studies [2, 27, 30, 53] point out scalability problems due to excessive cache-coherence traffic with traditional spinlocks. Scheduling issues like the lock holder preemption problem have been well studied [30, 56] and some solutions try to mitigate it [46, 56]. Nonetheless, we discovered lock-related issues that, to the best of our knowledge, have not been described before. Moreover, we are the first to analyze the impact of lock algorithms on such a large panel of applications, and to discuss in depth and summarize the many different bottlenecks they exhibit.

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

10 CONCLUSION

There are a large number of lock algorithms for multicore machines, leaving developers with the cumbersome task of choosing which algorithm to use for an application. One of the main reasons for this complexity is that there were no clear guidelines and methodologies helping developers to select the right lock for their workloads. In this paper, we presented a broad study of the performance and energy efficiency of 28 locks algorithms with 40 applications on Linux/x86 and four different multicore machines. In our quest to understand lock behavior, when choosing the best lock, for these 40 applications, we improve application throughput by on average 90% and energy efficiency by 110% with respect to the default POSIX mutex lock. To perform this study, we have implemented LiTL, an interposition library allowing the transparent replacement of lock algorithms used for Pthread mutex locks. The source code of LiTL and the data sets of our experimental results are available online [44].

From our study, we draw several conclusions, several of which have not been previously discovered: applications not only stress the lock/unlock interface, but also the full locking API (e.g., trylocks, condition variables), the memory footprint of a lock can directly affect the application performance, for many applications, the interaction between locks and scheduling is an important application performance factor and lock tail latencies may or may not affect application tail latency. We also confirm previous findings [27, 36, 43] on a larger number of applications, machines, and lock algorithms: no single lock is systematically the best, choosing the best lock is difficult, and energy efficiency and throughput go hand in hand in the context of lock algorithms. Finally, from the insights of our in-depth analysis of lock-related performance bottlenecks, we give guidelines for the choice of a lock algorithm based on given application characteristics. An immediate implication of this result is that lock-related research cannot simply focus on one of the many functions of locking. Lock designers must offer a full suite of lock, unlock, trylock, condition variables, and maybe even barriers, and reader-writer locks. These observations call for further research on optimized lock algorithms, as well as tools and dynamic approaches to better understand and control their behavior.

A STUDY OF LOCK PERFORMANCE

A.1 Selection of lock sensitive application

Table 19. For each application, performance gain of the best vs. worst lock and relative standard deviation (**A-48 machine**).

	Gain *one node*	R.Dev. *one node*	Gain *max nodes*	R.Dev. *max nodes*	Gain *opt nodes*	R.Dev. *opt nodes*
barnes	7%	2%	18%	4%	18%	4%
blackscholes	3%	1%	2%	0%	2%	0%
bodytrack	2%	1%	26%	6%	19%	4%
canneal	7%	1%	8%	1%	5%	1%
dedup	**190%**	**35%**	**544%**	**51%**	**200%**	**36%**
ferret	**1%**	**0%**	**481%**	**70%**	**132%**	**30%**
fmm	**21%**	**5%**	**53%**	**13%**	**50%**	**12%**
freqmine	12%	2%	5%	1%	5%	1%
histogram	21%	4%	54%	10%	46%	8%
kmeans	4%	1%	14%	3%	14%	3%
kyotocabinet	**427%**	**26%**	**1491%**	**55%**	**427%**	**26%**
linear_regression	**40%**	**7%**	**243%**	**20%**	**243%**	**23%**
lu_cb	4%	1%	3%	1%	3%	1%
lu_ncb	12%	3%	37%	7%	37%	7%
matrix_multiply	8%	2%	17%	5%	17%	4%
memcached-new	**37%**	**7%**	**621%**	**52%**	**78%**	**19%**
memcached-old	**255%**	**22%**	**1112%**	**47%**	**255%**	**22%**
mysqld	**100%**	**25%**	**54%**	**15%**	**53%**	**15%**
p_raytrace	3%	0%	3%	0%	3%	0%
pca	**13%**	**3%**	**257%**	**32%**	**74%**	**14%**
pca_ll	**3%**	**1%**	**569%**	**39%**	**177%**	**20%**
radiosity	**33%**	**7%**	**685%**	**32%**	**45%**	**8%**
radiosity_ll	**16%**	**3%**	**1524%**	**69%**	**234%**	**29%**
rocksdb	5%	1%	9%	2%	9%	2%
s_raytrace	**6%**	**1%**	**1479%**	**55%**	**340%**	**30%**
s_raytrace_ll	**2%**	**1%**	**1015%**	**58%**	**686%**	**53%**
sqlite	**455%**	**43%**	**939%**	**51%**	**511%**	**45%**
ssl_proxy	**1130%**	**31%**	**2595%**	**67%**	**2116%**	**41%**
streamcluster	**1342%**	**29%**	**2011%**	**48%**	**955%**	**28%**
streamcluster_ll	**18%**	**3%**	**1286%**	**54%**	**44%**	**10%**
string_match	6%	1%	18%	4%	18%	4%
swaptions	1%	0%	6%	1%	6%	1%
upscaledb	**152%**	**24%**	**501%**	**40%**	**214%**	**26%**
vips	**2%**	**0%**	**781%**	**42%**	**18%**	**6%**
volrend	**9%**	**2%**	**127%**	**22%**	**29%**	**7%**
water_nsquared	**11%**	**2%**	**79%**	**11%**	**79%**	**11%**
water_spatial	**18%**	**4%**	**70%**	**12%**	**70%**	**12%**
word_count	7%	2%	35%	8%	24%	6%
x264	3%	1%	4%	1%	4%	1%

Table 20. For each application, performance gain of the best vs. worst lock and relative standard deviation (**I-48 machine in performance mode**).

	Gain *one node*	R.Dev. *one node*	Gain *max nodes*	R.Dev. *max nodes*	Gain *opt nodes*	R.Dev. *opt nodes*
barnes	8%	2%	26%	6%	26%	6%
blackscholes	0%	0%	1%	0%	1%	0%
bodytrack	2%	1%	39%	6%	5%	2%
canneal	1%	0%	1%	0%	1%	0%
dedup	**729%**	**46%**	**2316%**	**83%**	**729%**	**46%**
ferret	**1%**	**0%**	**662%**	**78%**	**81%**	**20%**
fmm	7%	2%	26%	6%	22%	5%
freqmine	2%	0%	1%	0%	1%	0%
histogram	53%	7%	31%	7%	48%	7%
kmeans	2%	0%	11%	2%	11%	2%
kyotocabinet	**462%**	**29%**	**579%**	**37%**	**413%**	**28%**
linear_regression	**18%**	**3%**	**84%**	**16%**	**80%**	**14%**
lu_cb	0%	0%	3%	1%	3%	1%
lu_ncb	9%	2%	12%	3%	12%	3%
matrix_multiply	3%	1%	7%	2%	7%	2%
memcached-new	**139%**	**20%**	**297%**	**25%**	**69%**	**14%**
memcached-old	**85%**	**19%**	**195%**	**38%**	**85%**	**19%**
mysqld	**62%**	**14%**	**57%**	**13%**	**57%**	**14%**
p_raytrace	3%	1%	3%	1%	1%	0%
pca	**278%**	**20%**	**315%**	**30%**	**308%**	**21%**
pca_ll	**90%**	**9%**	**981%**	**47%**	**403%**	**31%**
radiosity	**63%**	**8%**	**174%**	**23%**	**72%**	**9%**
radiosity_ll	**766%**	**31%**	**1979%**	**65%**	**1531%**	**48%**
rocksdb	2%	1%	11%	3%	11%	3%
s_raytrace	**15%**	**2%**	**1256%**	**50%**	**212%**	**31%**
s_raytrace_ll	**3%**	**1%**	**1260%**	**49%**	**345%**	**42%**
sqlite	**618%**	**41%**	**3581%**	**68%**	**618%**	**41%**
ssl_proxy	**1057%**	**40%**	**1594%**	**51%**	**1308%**	**45%**
streamcluster	**43%**	**11%**	**489%**	**70%**	**43%**	**11%**
streamcluster_ll	**66%**	**15%**	**569%**	**77%**	**162%**	**33%**
string_match	1%	0%	6%	2%	6%	2%
swaptions	1%	0%	3%	1%	3%	1%
upscaledb	**277%**	**27%**	**303%**	**33%**	**275%**	**28%**
vips	**1%**	**0%**	**707%**	**52%**	**24%**	**10%**
volrend	**8%**	**3%**	**151%**	**15%**	**42%**	**8%**
water_nsquared	**40%**	**9%**	**129%**	**20%**	**129%**	**20%**
water_spatial	**361%**	**33%**	**917%**	**42%**	**917%**	**42%**
word_count	9%	2%	14%	4%	9%	2%
x264	1%	0%	2%	0%	2%	0%

Table 21. For each application, performance gain of the best vs. worst lock and relative standard deviation (**I-20 machine in performance mode**).

	Gain *one node*	R.Dev. *one node*	Gain *max nodes*	R.Dev. *max nodes*	Gain *opt nodes*	R.Dev. *opt nodes*
barnes	6%	2%	12%	3%	12%	3%
blackscholes	0%	0%	1%	0%	1%	0%
bodytrack	1%	0%	1%	0%	1%	0%
canneal	2%	0%	4%	1%	4%	1%
dedup	**723%**	**46%**	**1063%**	**61%**	**723%**	**46%**
ferret	**60%**	**15%**	**408%**	**66%**	**137%**	**31%**
fmm	5%	1%	10%	2%	10%	2%
freqmine	3%	1%	4%	1%	4%	1%
histogram	7%	2%	21%	4%	7%	2%
kmeans	3%	1%	2%	1%	2%	1%
kyotocabinet	**256%**	**26%**	**254%**	**28%**	**256%**	**26%**
linear_regression	6%	1%	28%	6%	28%	6%
lu_cb	0%	0%	3%	1%	3%	1%
lu_ncb	10%	2%	6%	2%	6%	2%
matrix_multiply	1%	0%	2%	0%	2%	0%
memcached-new	38%	8%	38%	8%	38%	8%
memcached-old	**316%**	**28%**	**316%**	**28%**	**316%**	**28%**
p_raytrace	3%	1%	4%	1%	3%	1%
pca	**8%**	**2%**	**185%**	**21%**	**24%**	**6%**
pca_ll	**4%**	**1%**	**473%**	**28%**	**89%**	**15%**
radiosity	**25%**	**5%**	**77%**	**13%**	**23%**	**5%**
radiosity_ll	**12%**	**3%**	**802%**	**42%**	**70%**	**19%**
rocksdb	6%	2%	11%	2%	11%	2%
s_raytrace	**2%**	**0%**	**338%**	**25%**	**92%**	**15%**
s_raytrace_ll	**1%**	**0%**	**643%**	**30%**	**77%**	**14%**
sqlite	**394%**	**36%**	**8608%**	**71%**	**394%**	**36%**
streamcluster	**36%**	**8%**	**387%**	**27%**	**36%**	**8%**
streamcluster_ll	**47%**	**9%**	**466%**	**30%**	**113%**	**22%**
string_match	0%	0%	2%	1%	2%	1%
swaptions	0%	0%	1%	0%	1%	0%
upscaledb	**127%**	**24%**	**153%**	**26%**	**148%**	**26%**
vips	**1%**	**0%**	**115%**	**22%**	**94%**	**22%**
volrend	9%	2%	56%	8%	39%	7%
water_nsquared	**24%**	**6%**	**48%**	**10%**	**48%**	**10%**
water_spatial	**170%**	**24%**	**326%**	**31%**	**326%**	**31%**
word_count	2%	0%	4%	1%	2%	0%
x264	2%	0%	3%	1%	3%	1%

Table 22. For each application, performance gain of the best vs. worst lock and relative standard deviation (**A-64 machine with thread-to-node pinning**).

	Gain *one node*	R.Dev. *one node*	Gain *max nodes*	R.Dev. *max nodes*	Gain *opt nodes*	R.Dev. *opt nodes*
barnes	3%	1%	22%	5%	22%	5%
blackscholes	1%	0%	2%	0%	2%	0%
bodytrack	0%	0%	44%	6%	15%	3%
canneal	2%	0%	4%	1%	3%	1%
dedup	**623%**	**51%**	**1090%**	**51%**	**727%**	**56%**
facesim	**1%**	**0%**	**297%**	**25%**	**21%**	**5%**
ferret	**8%**	**3%**	**386%**	**64%**	**356%**	**63%**
fft	7%	1%	9%	2%	9%	2%
fluidanimate	**60%**	**11%**	**301%**	**39%**	**198%**	**36%**
fmm	5%	1%	12%	3%	12%	3%
freqmine	4%	1%	3%	1%	3%	1%
histogram	5%	1%	20%	5%	16%	4%
kmeans	6%	2%	5%	1%	5%	1%
kyotocabinet	**116%**	**17%**	**2034%**	**54%**	**116%**	**17%**
linear_regression	**3%**	**1%**	**101%**	**17%**	**70%**	**13%**
lu_cb	0%	0%	4%	1%	4%	1%
lu_ncb	6%	1%	5%	1%	5%	1%
matrix_multiply	4%	1%	5%	1%	5%	1%
memcached-new	**35%**	**7%**	**910%**	**47%**	**81%**	**20%**
memcached-old	**128%**	**25%**	**309%**	**49%**	**115%**	**24%**
mysqld	**85%**	**28%**	**66%**	**21%**	**59%**	**16%**
ocean_cp	**4%**	**1%**	**130%**	**20%**	**12%**	**3%**
ocean_ncp	**3%**	**1%**	**110%**	**16%**	**10%**	**3%**
p_raytrace	1%	0%	1%	0%	1%	0%
pca	**2%**	**1%**	**347%**	**32%**	**58%**	**9%**
pca_ll	**7%**	**2%**	**551%**	**41%**	**125%**	**18%**
radiosity	**5%**	**1%**	**114%**	**18%**	**7%**	**2%**
radiosity_ll	**9%**	**2%**	**2260%**	**64%**	**146%**	**22%**
radix	1%	0%	15%	3%	15%	3%
rocksdb	7%	2%	19%	5%	19%	5%
s_raytrace	**8%**	**2%**	**1192%**	**58%**	**222%**	**29%**
s_raytrace_ll	**1%**	**0%**	**1477%**	**59%**	**467%**	**52%**
sqlite	**2830%**	**43%**	**809%**	**86%**	**828%**	**44%**
ssl_proxy	**29%**	**5%**	**1250%**	**56%**	**68%**	**14%**
streamcluster	**21%**	**4%**	**706%**	**50%**	**41%**	**9%**
streamcluster_ll	**32%**	**6%**	**826%**	**52%**	**78%**	**20%**
string_match	7%	2%	8%	2%	8%	2%
swaptions	1%	0%	2%	0%	2%	0%
upscaledb	**143%**	**23%**	**1555%**	**56%**	**191%**	**25%**
vips	**81%**	**21%**	**238%**	**28%**	**294%**	**33%**
volrend	**5%**	**1%**	**106%**	**16%**	**28%**	**6%**
water_nsquared	**7%**	**2%**	**89%**	**15%**	**89%**	**15%**
water_spatial	**95%**	**14%**	**298%**	**26%**	**298%**	**26%**
word_count	2%	0%	5%	1%	4%	1%
x264	0%	0%	1%	0%	1%	0%

Table 23. For each application, performance gain of the best vs. worst lock and relative standard deviation (**I-48 machine in energy-saving mode**).

	Gain *one node*	R.Dev. *one node*	Gain *max nodes*	R.Dev. *max nodes*	Gain *opt nodes*	R.Dev. *opt nodes*
barnes	8%	2%	26%	6%	26%	6%
blackscholes	0%	0%	1%	0%	1%	0%
bodytrack	2%	1%	39%	6%	5%	2%
canneal	1%	0%	1%	0%	1%	0%
dedup	**729%**	**46%**	**2316%**	**83%**	**729%**	**46%**
ferret	**1%**	**0%**	**662%**	**78%**	**81%**	**20%**
fmm	7%	2%	26%	6%	22%	5%
freqmine	2%	0%	1%	0%	1%	0%
histogram	53%	7%	31%	7%	48%	7%
kmeans	2%	0%	11%	2%	11%	2%
kyotocabinet	**462%**	**29%**	**579%**	**37%**	**413%**	**28%**
linear_regression	**18%**	**3%**	**84%**	**16%**	**80%**	**14%**
lu_cb	0%	0%	3%	1%	3%	1%
lu_ncb	9%	2%	12%	3%	12%	3%
matrix_multiply	3%	1%	7%	2%	7%	2%
memcached-new	**139%**	**20%**	**297%**	**25%**	**69%**	**14%**
memcached-old	**85%**	**19%**	**195%**	**38%**	**85%**	**19%**
mysqld	**62%**	**14%**	**57%**	**13%**	**57%**	**14%**
p_raytrace	3%	1%	3%	1%	1%	0%
pca	**278%**	**20%**	**315%**	**30%**	**308%**	**21%**
pca_ll	**90%**	**9%**	**981%**	**47%**	**403%**	**31%**
radiosity	**63%**	**8%**	**174%**	**23%**	**72%**	**9%**
radiosity_ll	**766%**	**31%**	**1979%**	**65%**	**1531%**	**48%**
rocksdb	2%	1%	11%	3%	11%	3%
s_raytrace	**15%**	**2%**	**1256%**	**50%**	**212%**	**31%**
s_raytrace_ll	**3%**	**1%**	**1260%**	**49%**	**345%**	**42%**
sqlite	**618%**	**41%**	**3581%**	**68%**	**618%**	**41%**
ssl_proxy	**1057%**	**40%**	**1594%**	**51%**	**1308%**	**45%**
streamcluster	**43%**	**11%**	**489%**	**70%**	**43%**	**11%**
streamcluster_ll	**66%**	**15%**	**569%**	**77%**	**162%**	**33%**
string_match	1%	0%	6%	2%	6%	2%
swaptions	1%	0%	3%	1%	3%	1%
upscaledb	**277%**	**27%**	**303%**	**33%**	**275%**	**28%**
vips	**1%**	**0%**	**707%**	**52%**	**24%**	**10%**
volrend	**8%**	**3%**	**151%**	**15%**	**42%**	**8%**
water_nsquared	**40%**	**9%**	**129%**	**20%**	**129%**	**20%**
water_spatial	**361%**	**33%**	**917%**	**42%**	**917%**	**42%**
word_count	9%	2%	14%	4%	9%	2%
x264	1%	0%	2%	0%	2%	0%

Table 24. For each application, performance gain of the best vs. worst lock and relative standard deviation (**I-20 machine in energy-saving mode**).

	Gain *one node*	R.Dev. *one node*	Gain *max nodes*	R.Dev. *max nodes*	Gain *opt nodes*	R.Dev. *opt nodes*
barnes	6%	2%	12%	3%	12%	3%
blackscholes	0%	0%	1%	0%	1%	0%
bodytrack	1%	0%	1%	0%	1%	0%
canneal	2%	0%	4%	1%	4%	1%
dedup	**723%**	**46%**	**1063%**	**61%**	**723%**	**46%**
ferret	**60%**	**15%**	**408%**	**66%**	**137%**	**31%**
fmm	5%	1%	10%	2%	10%	2%
freqmine	3%	1%	4%	1%	4%	1%
histogram	7%	2%	21%	4%	7%	2%
kmeans	3%	1%	2%	1%	2%	1%
kyotocabinet	**256%**	**26%**	**254%**	**28%**	**256%**	**26%**
linear_regression	6%	1%	28%	6%	28%	6%
lu_cb	0%	0%	3%	1%	3%	1%
lu_ncb	10%	2%	6%	2%	6%	2%
matrix_multiply	1%	0%	2%	0%	2%	0%
memcached-new	38%	8%	38%	8%	38%	8%
memcached-old	**316%**	**28%**	**316%**	**28%**	**316%**	**28%**
p_raytrace	3%	1%	4%	1%	3%	1%
pca	**8%**	**2%**	**185%**	**21%**	**24%**	**6%**
pca_ll	**4%**	**1%**	**473%**	**28%**	**89%**	**15%**
radiosity	**25%**	**5%**	**77%**	**13%**	**23%**	**5%**
radiosity_ll	**12%**	**3%**	**802%**	**42%**	**70%**	**19%**
rocksdb	6%	2%	11%	2%	11%	2%
s_raytrace	**2%**	**0%**	**338%**	**25%**	**92%**	**15%**
s_raytrace_ll	**1%**	**0%**	**643%**	**30%**	**77%**	**14%**
sqlite	**394%**	**36%**	**8608%**	**71%**	**394%**	**36%**
streamcluster	**36%**	**8%**	**387%**	**27%**	**36%**	**8%**
streamcluster_ll	**47%**	**9%**	**466%**	**30%**	**113%**	**22%**
string_match	0%	0%	2%	1%	2%	1%
swaptions	0%	0%	1%	0%	1%	0%
upscaledb	**127%**	**24%**	**153%**	**26%**	**148%**	**26%**
vips	**1%**	**0%**	**115%**	**22%**	**94%**	**22%**
volrend	9%	2%	56%	8%	39%	7%
water_nsquared	**24%**	**6%**	**48%**	**10%**	**48%**	**10%**
water_spatial	**170%**	**24%**	**326%**	**31%**	**326%**	**31%**
word_count	2%	0%	4%	1%	2%	0%
x264	2%	0%	3%	1%	3%	1%

A.2 Selection of the number of nodes

Table 25. For each *(lock-sensitive application, lock)* pair, performance gain (in %) of *opt nodes* over *max nodes*. The background color of a cell indicates the number of nodes for *opt nodes*: 1|2|4|6|8. Dashes correspond to untested cases (**A-48 machine**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	-	33	80	78	61	45	-	-	-	72	82	67	62	83	84	86	86	45	703	48	45	69	49	721	59	50	-
ferret	150	190		250		154	141	197		186	142	147	201		188		207			203				7	191	146		
fmm																												
kyotocabinet	8	24	51	13	321	18	16	24		24	13	9	27	51	26	10	20	30	222	36	223	213	1k	849	140	80	504	192
linear_regression	13	10		9	181	13	11	14	18	10	10	11	18	44	6	15	24		133	14	13		35	13	15	7	10	13
memcached-new				6	267	-		-	-	-		-	93	132	27	166	5			-	15	87	524	317	101	48	171	151
memcached-old	65	81		15	230	-	79	-	-	-	67	-	28	70	88	27	79	46	25	-	27	50	1k	638	202	142	165	327
mysqld	-	-	-	-		-	-	-	-	-	-	-	-		-		-			-			-	-	-	-	-	-
pca	26	32	10	12	176	19	30	31	80	34	12	14	12	42	21	133	24		62	26	54	27	169	89	75	29	101	92
pca_ll	15	39			337	23	33	22	25	32	38	27		144	32	26	23	11	92	20	100	27	291	143	80	17	110	142
radiosity				9	85			9	538			6		36	18	557		13	52	8	56	17	143	52	39	14	49	51
radiosity_ll			6		471			8	269				6	114	10	281	19	7	300	33	261	87	841	373	227	68	342	316
s_raytrace	22	26		27	590	21	23		121	16	26	10	34	126	16	202	30		94	28	70	23	226	120	193	53	161	204
s_raytrace_ll					106				41							44			7				74		12	7	19	40
sqlite	-	-	-	-	261	-	-	-	-	-	-	-	-		-	6	-	2k	153	-	134	82	-	-	-	-	-	-
ssl_proxy	39	49	15	25	973	52	40	85	104	47	42	38	48		78	46	45	52	213	85	247	114	813	328	40	191	412	423
streamcluster	715	674	1k	937	1k	490	546	-	-	-	488	-	2k	7k	1k	12k	2k	1k	850	792	48	59	2k	1k	2k	1k	1k	928
streamcluster_ll	117	116	278	88	392	68	120	-	-	-	98	-	335	2k	389	2k	354	212	145	144	269	384	510	282	360	293	201	172
upscaledb	6	6	7	6				7	11	7	7	8			7	9	7	5		7	58	30	364	172	52	25	67	173
vips	108	80		278	13	193	148	-	-	-	169	-	667		47		44			58					45	5		
volrend	45	52	37	45	91	38	35	38	102	49	32	31	28	137	50	109	49	49	85	40	102	107	118	72	44	41	47	50
water_nsquared																												
water_spatial																												

Table 26. For each *(lock-sensitive application, lock)* pair, performance gain (in %) of *opt nodes* over *max nodes*. The background color of a cell indicates the number of nodes for *opt nodes*: 1|2|3|4. Dashes correspond to untested cases (**I-48 machine in performance mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	146	62	128	66	3k	3k	240	54	385	96	255	491	78	243	69	471	54	71	3k	84	99	82	64	3k	2k	64	77
ferret	407	355	8	382		390	359	339		350	402	319	365		349		288			322			6		333	247	7	5
kyotocabinet	8	11	37		136	12	12	16	26	12	7	7	24	17	10	6	10	15	18	19	13	16	208	130	21	13	31	35
linear_regression					20	11				6		14					34							5				
memcached-new	24			25	36	-	43	-	-	-		-		29	21	135		12	37	-	46	38	115	73	18		18	19
memcached-old	55	20	131	57	58	-	20	-	-	-	63	-	126	118	97	100	132	126	110	-	135	121	97	90	109	122	134	86
mysqld	-	-	-	-		-	-	-	-	-	-	-	-		-		-			-			-	-	-	-	-	-
pca	10	20		10	74	11	16	15	21	14	7	12	15		10		13	13	13	29	14	18	236	178	41	36	46	45
pca_ll					29				104					19		278			6	8	7	47	180	284	28	7	77	67
radiosity					38				59							97				9	5	5	97	79	10	7	15	15
radiosity_ll					475	31			46							27	7	6	18	38	5		472	364	38	29	104	105
s_raytrace		6			191			9	380	5		6				335				27	5	12	226	190	35	26	71	86
s_raytrace_ll									207							207							113	106				
sqlite	-	-	-	-	368	-	-	-	-	-	-	-	-	11	-	33	-	4k	34	-	22	17	-	-	-	-	-	-
ssl_proxy	51	45	39	31	421	60	62	35	15	39	49	49	25	16	49	19	48	44	43	76	34	18	335	276	46	40	72	78
streamcluster	306	187	1k	641	1k	422	233	-	-	-	307	-	865	1k	896	1k	750	877	1k	228	1k	1k	1k	847	1k	1k	725	729
streamcluster_ll	39	14	185	120	327	116		-	-	-	49	-	108	149	165	172	134	118	152	6	253	192	168	165	170	162	127	158
upscaledb													8			8			6		6		75	72	7		7	8
vips	152	83		556	21	316	246	-	-	-	261	-	313		94		92			118					62			
volrend		10	89	5	23	8	6	9	9			8		12	14		6		5		11	6	18	15		7	9	9
water_nsquared																												
water_spatial																												

Table 27. For each *(lock-sensitive application, lock)* pair, performance gain (in %) of *opt nodes* over *max nodes*. The background color of a cell indicates the number of nodes for *opt nodes*: 1|2. Dashes correspond to untested cases (**I-20 machine in performance mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	207	69	182	23	244	244	38	5	52	266	158	229	28	166	28	226	48	39	205	38	45	66	67	177	331	68	21
ferret	130	131		131		131	130	127		132	125	123	115		130		131			132			8	9	134	108	9	10
kyotocabinet	6		8			7	5	5			7		9								7	9	44	34	8		8	7
memcached-old						-		-	-	-		-								-								
pca					130				123							119						14	18					
pca_ll					205				178							189												
radiosity					45				47					7		52				6	10	15	41	13	6		8	
radiosity_ll					320				445							485						57	234	73	21		16	
s_raytrace					131				115							72												
s_raytrace_ll					286				319							320												
sqlite	-	-	-	-	333	-	-	-	-	-	-	-	-	29	-	27	-	10k	6	-	10	43	-	-	-	-	-	-
streamcluster	42	25	57	22	30	38	18	-	-	-	20	-	56	66	58	54	60	71	372	15	223	132	84	46	67	68	37	42
streamcluster_ll								-	-	-		-							166		100	28						
upscaledb																							17					
vips	11							-	-	-		-																
water_nsquared																												
water_spatial																												

Table 28. For each *(lock-sensitive application, lock)* pair, performance gain (in %) of *opt nodes* over *max nodes*. The background color of a cell indicates the number of nodes for *opt nodes*: 1|2|4|6|8. Dashes correspond to untested cases (**A-64 machine with thread-to-node pinning**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	173	53	40	48	47	48	165	90	163	15	48	8	49		48		52	83	59	92	90	76	54	65	50	54	138
facesim	18	19	53	48	134	19	18	20	59	20	18	18	14	56	20	60	19	22	42	19	42	58	276	127	45	17	55	61
ferret							6				9	11	9															
fluidanimate	-	41						-	-	-		-		6		6												35
kyotocabinet	23	43	50	5	47	27	23	42	407	40	23	25	37	36	43	501	42	53	222	61	260	200	1k	1k	240	97	669	365
linear_regression									21							24							18		7			9
memcached-new	17	14			78	-	10	-	-	-	14	-	7	21	17	134	29	17	12	-	31	97	588	465	167	90	317	42
memcached-old	344	278	412	77	78	-	110	-	-	-	52	-	620	635	435	480	611	429	372	-	512	561	687	360	656	709	394	467
mysqld	-	-	-	-	26	-	-	-	-	-	-	-	-		-		-	6	7	-			-	-	-	-	-	-
ocean_cp	20	18	52	68	95	18	19	19	61	18	18	25	21	58	17	58	22	32	44	19	38	56	147	97	30	19	51	48
ocean_ncp	15	9	33	22	74	11	14	14	44	9	14	15	13	37	11	42	11	17	24	12	27	36	111	67	23	13	36	35
pca	28	22	20		21	27	27	23	157	23	26	26	24		26	144	20	21	89	27	88	25	282	102	66	6	189	126
pca_ll	23	17			15	24	12	33	118	9	30	21	8		23	114			74	17	64		355	269	57		195	120
radiosity			7		5				65							68			29		25	16	103	47	28	10	48	49
radiosity_ll		6				19	24	6	807	6						750	7	13	243	34	153	83	1k	654	291	80	664	440
s_raytrace					43				302							307			107		81	31	371	162	194	26	412	284
s_raytrace_ll									179							176							150	67	24		127	81
sqlite	-	-	-	-		-	-	-	-	-	-	-	-		-	375	-		185	-	160	86	-	-	-	-	-	-
ssl_proxy	65	66	112	48	165	47	42	78	1k	70	55	55	52	55	73	1k	81	71	349	96	287	199	1k	482	340	167	765	684
streamcluster	1k	968	2k	1k	1k	915	973	-	-	-	863	-	3k	6k	2k	3k	2k	1k	1k	1k	2k	3k	5k	3k	3k	2k	2k	2k
streamcluster_ll	233	178	448	267	370	211	268	-	-	-	198	-	479	2k	285	588	373	246	265	231	423	437	979	891	888	583	641	565
upscaledb						716	360		38							37				401	16	32	353	279	1k	15	125	75
vips	27		20		15	9		-	-	-		-		19	16	20	6	20	23		21	21	18	26		18	24	27
volrend	14	11	29	20	38	7	7	11	21	14	8	9	14	20	13	19	12	15	30	13	28	35	72	58	30	17	35	35
water_nsquared																												
water_spatial																												

Table 29. For each *(lock-sensitive application, lock)* pair, performance gain (in %) of *opt nodes* over *max nodes*. The background color of a cell indicates the number of nodes for *opt nodes*: 1|2|3|4. Dashes correspond to untested cases. (**I-48 machine in energy-saving mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	146	62	128	66	3k	3k	240	54	385	96	255	491	78	243	69	471	54	71	3k	84	99	82	64	3k	2k	64	77
ferret	407	355	8	382		390	359	339		350	402	319	365		349		288			322			6		333	247	7	5
kyotocabinet	8	11	37		136	12	12	16	26	12	7	7	24	17	10	6	10	15	18	19	13	16	208	130	21	13	31	35
linear_regression					20	11				6		14					34							5				
memcached-new	24			25	36	-	43	-	-	-		-		29	21	135		12	37	-	46	38	115	73	18		18	19
memcached-old	55	20	131	57	58	-	20	-	-	-	63	-	126	118	97	100	132	126	110	-	135	121	97	90	109	122	134	86
mysqld	-	-	-	-		-	-	-	-	-	-	-	-		-		-			-			-	-	-	-	-	-
pca	10	20		10	74	11	16	15	21	14	7	12	15		10		13	13	13	29	14	18	236	178	41	36	46	45
pca_ll					29				104					19		278			6	8	7	47	180	284	28	7	77	67
radiosity					38				59							97				9	5	5	97	79	10	7	15	15
radiosity_ll					475	31			46							27	7	6	18	38	5		472	364	38	29	104	105
s_raytrace		6			191			9	380	5		6				335				27	5	12	226	190	35	26	71	86
s_raytrace_ll									207							207							113	106				
sqlite	-	-	-	-	368	-	-	-	-	-	-	-	-	11	-	33	-	4k	34	-	22	17	-	-	-	-	-	-
ssl_proxy	51	45	39	31	421	60	62	35	15	39	49	49	25	16	49	19	48	44	43	76	34	18	335	276	46	40	72	78
streamcluster	306	187	1k	641	1k	422	233	-	-	-	307	-	865	1k	896	1k	750	877	1k	228	1k	1k	1k	847	1k	1k	725	729
streamcluster_ll	39	14	185	120	327	116		-	-	-	49	-	108	149	165	172	134	118	152	6	253	192	168	165	170	162	127	158
upscaledb													8			8			6		6		75	72	7		7	8
vips	152	83		556	21	316	246	-	-	-	261	-	313		94		92			118					62			
volrend		10	89	5	23	8	6	9	9			8		12	14		6		5		11	6	18	15		7	9	9
water_nsquared																												
water_spatial																												

Table 30. For each *(lock-sensitive application, lock)* pair, performance gain (in %) of *opt nodes* over *max nodes*. The background color of a cell indicates the number of nodes for *opt nodes*: 1|2. Dashes correspond to untested cases (**I-20 machine in energy-saving mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	207	69	182	23	244	244	38	5	52	266	158	229	28	166	28	226	48	39	205	38	45	66	67	177	331	68	21
ferret	130	131		131		131	130	127		132	125	123	115		130		131			132			8	9	134	108	9	10
kyotocabinet	6		8			7	5	5			7		9								7	9	44	34	8		8	7
memcached-old						-		-	-	-		-								-								
pca					130				123							119						14	18					
pca_ll					205				178							189												
radiosity					45				47					7		52				6	10	15	41	13	6		8	
radiosity_ll					320				445							485						57	234	73	21		16	
s_raytrace					131				115							72												
s_raytrace_ll					286				319							320												
sqlite	-	-	-	-	333	-	-	-	-	-	-	-	-	29	-	27	-	10k	6	-	10	43	-	-	-	-	-	-
streamcluster	42	25	57	22	30	38	18	-	-	-	20	-	56	66	58	54	60	71	372	15	223	132	84	46	67	68	37	42
streamcluster_ll								-	-	-		-							166		100	28						
upscaledb																							17					
vips	11							-	-	-		-																
water_nsquared																												
water_spatial																												

A.3 Are some locks always among the best?

Table 31. For each lock, fraction of the lock-sensitive applications for which the lock yields the best performance for three configurations: *one node*, *max nodes* and *opt nodes* (**A-64 machine**).

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

Table 32. For each lock, fraction of the lock-sensitive applications for which the lock yields the best performance for three configurations: *one node*, *max nodes* and *opt nodes* (**A-48 machine**).

	Number of nodes		
Locks	*one node*	*max nodes*	*opt nodes*
ahmcs	60%	15%	40%
alock-ls	55%	10%	35%
backoff	71%	33%	33%
c-bo-mcs_spin	57%	24%	19%
c-bo-mcs_stp	52%	9%	9%
c-ptl-tkt	58%	16%	21%
c-tkt-tkt	62%	14%	29%
clh_spin	47%	13%	20%
clh_stp	33%	7%	7%
clh-ls	53%	0%	27%
hmcs	71%	29%	48%
hticket-ls	69%	31%	31%
malth_spin	67%	19%	10%
malth_stp	35%	4%	4%
mcs_spin	67%	14%	43%
mcs_stp	39%	9%	9%
mcs-ls	67%	5%	29%
mcs-timepub	52%	22%	35%
mutexee	61%	22%	30%
partitioned	58%	5%	21%
pthread	43%	17%	17%
pthreadadapt	57%	26%	17%
spinlock	67%	14%	24%
spinlock-ls	67%	10%	29%
ticket	71%	5%	14%
ticket-ls	71%	10%	29%
ttas	67%	10%	24%
ttas-ls	65%	0%	20%

Table 33. For each lock, fraction of the lock-sensitive applications for which the lock yields the best performance for three configurations: *one node*, *max nodes* and *opt nodes* (**I-48 machine in performance mode**).

	Number of nodes		
Locks	*one node*	*max nodes*	*opt nodes*
ahmcs	47%	26%	37%
alock-ls	55%	15%	10%
backoff	60%	30%	20%
c-bo-mcs_spin	65%	35%	35%
c-bo-mcs_stp	55%	14%	18%
c-ptl-tkt	72%	44%	50%
c-tkt-tkt	70%	40%	50%
clh_spin	47%	7%	7%
clh_stp	20%	7%	7%
clh-ls	27%	0%	0%
hmcs	75%	45%	50%
hticket-ls	73%	33%	33%
malth_spin	55%	10%	15%
malth_stp	41%	18%	18%
mcs_spin	60%	10%	20%
mcs_stp	27%	5%	5%
mcs-ls	55%	15%	15%
mcs-timepub	45%	9%	5%
mutexee	41%	27%	27%
partitioned	56%	17%	11%
pthread	41%	23%	27%
pthreadadapt	41%	14%	23%
spinlock	40%	15%	20%
spinlock-ls	40%	15%	15%
ticket	45%	10%	15%
ticket-ls	55%	10%	15%
ttas	55%	20%	20%
ttas-ls	30%	5%	5%

Table 34. For each lock, fraction of the lock-sensitive applications for which the lock yields the best performance for three configurations: *one node*, *max nodes* and *opt nodes* (**I-20 machine in performance mode**).

	Number of nodes		
Locks	*one node*	*max nodes*	*opt nodes*
ahmcs	60%	53%	53%
alock-ls	50%	38%	38%
backoff	56%	38%	44%
c-bo-mcs_spin	75%	62%	62%
c-bo-mcs_stp	47%	24%	24%
c-ptl-tkt	67%	60%	60%
c-tkt-tkt	75%	62%	62%
clh_spin	42%	25%	25%
clh_stp	42%	8%	8%
clh-ls	42%	25%	25%
hmcs	69%	62%	62%
hticket-ls	75%	75%	75%
malth_spin	56%	44%	44%
malth_stp	59%	47%	47%
mcs_spin	62%	50%	50%
mcs_stp	59%	24%	24%
mcs-ls	62%	50%	50%
mcs-timepub	53%	53%	53%
mutexee	59%	41%	47%
partitioned	60%	47%	47%
pthread	71%	35%	47%
pthreadadapt	59%	47%	47%
spinlock	75%	44%	50%
spinlock-ls	62%	44%	44%
ticket	56%	38%	38%
ticket-ls	62%	44%	44%
ttas	69%	50%	50%
ttas-ls	50%	31%	31%

Table 35. For each lock, fraction of the lock-sensitive applications for which the lock yields the best performance for three configurations: *one node*, *max nodes* and *opt nodes* (**A-64 machine with thread-to-node pinning**).

	Number of nodes		
Locks	*one node*	*max nodes*	*opt nodes*
ahmcs	50%	32%	41%
alock-ls	62%	21%	25%
backoff	75%	21%	42%
c-bo-mcs_spin	54%	17%	29%
c-bo-mcs_stp	54%	19%	19%
c-ptl-tkt	59%	32%	36%
c-tkt-tkt	54%	29%	38%
clh_spin	67%	28%	44%
clh_stp	56%	6%	11%
clh-ls	67%	11%	28%
hmcs	54%	50%	46%
hticket-ls	78%	39%	44%
malth_spin	54%	33%	38%
malth_stp	58%	38%	38%
mcs_spin	62%	38%	46%
mcs_stp	62%	19%	19%
mcs-ls	54%	29%	33%
mcs-timepub	54%	8%	27%
mutexee	65%	19%	31%
partitioned	73%	23%	36%
pthread	62%	19%	27%
pthreadadapt	65%	19%	27%
spinlock	62%	12%	12%
spinlock-ls	75%	17%	33%
ticket	75%	8%	25%
ticket-ls	79%	25%	38%
ttas	92%	17%	50%
ttas-ls	79%	4%	21%

Table 36. For each lock, fraction of the lock-sensitive applications for which the lock yields the best performance for three configurations: *one node*, *max nodes* and *opt nodes* (**I-48 machine in energy-saving mode**).

	Number of nodes		
Locks	*one node*	*max nodes*	*opt nodes*
ahmcs	47%	26%	37%
alock-ls	55%	15%	10%
backoff	60%	30%	20%
c-bo-mcs_spin	65%	35%	35%
c-bo-mcs_stp	55%	14%	18%
c-ptl-tkt	72%	44%	50%
c-tkt-tkt	70%	40%	50%
clh_spin	47%	7%	7%
clh_stp	20%	7%	7%
clh-ls	27%	0%	0%
hmcs	75%	45%	50%
hticket-ls	73%	33%	33%
malth_spin	55%	10%	15%
malth_stp	41%	18%	18%
mcs_spin	60%	10%	20%
mcs_stp	27%	5%	5%
mcs-ls	55%	15%	15%
mcs-timepub	45%	9%	5%
mutexee	41%	27%	27%
partitioned	56%	17%	11%
pthread	41%	23%	27%
pthreadadapt	41%	14%	23%
spinlock	40%	15%	20%
spinlock-ls	40%	15%	15%
ticket	45%	10%	15%
ticket-ls	55%	10%	15%
ttas	55%	20%	20%
ttas-ls	30%	5%	5%

Table 37. For each lock, fraction of the lock-sensitive applications for which the lock yields the best performance for three configurations: *one node*, *max nodes* and *opt nodes* (**I-20 machine in energy-saving mode**).

	Number of nodes		
Locks	*one node*	*max nodes*	*opt nodes*
ahmcs	60%	53%	53%
alock-ls	50%	38%	38%
backoff	56%	38%	44%
c-bo-mcs_spin	75%	62%	62%
c-bo-mcs_stp	47%	24%	24%
c-ptl-tkt	67%	60%	60%
c-tkt-tkt	75%	62%	62%
clh_spin	42%	25%	25%
clh_stp	42%	8%	8%
clh-ls	42%	25%	25%
hmcs	69%	62%	62%
hticket-ls	75%	75%	75%
malth_spin	56%	44%	44%
malth_stp	59%	47%	47%
mcs_spin	62%	50%	50%
mcs_stp	59%	24%	24%
mcs-ls	62%	50%	50%
mcs-timepub	53%	53%	53%
mutexee	59%	41%	47%
partitioned	60%	47%	47%
pthread	71%	35%	47%
pthreadadapt	59%	47%	47%
spinlock	75%	44%	50%
spinlock-ls	62%	44%	44%
ticket	56%	38%	38%
ticket-ls	62%	44%	44%
ttas	69%	50%	50%
ttas-ls	50%	31%	31%

A.4 Is there a clear hierarchy between locks?

A.4.1 At opt nodes.

Table 38. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**A-64 machine**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		38	46	42	54	23	33	16	63	21	21	26	29	54	42	54	42	46	58	27	50	58	50	50	42	42	38	42	41
alock-ls	12		42	27	38	4	12	5	60	20	8	10	27	27	12	50	19	27	54	4	42	50	46	42	35	27	31	38	29
backoff	33	35		35	58	25	23	30	70	40	35	30	35	38	27	58	35	23	54	25	46	50	38	35	27	19	23	50	37
c-bo-mcs_spin	29	46	23		42	17	23	30	70	40	35	20	23	35	19	46	27	19	62	21	54	46	35	42	35	35	35	58	36
c-bo-mcs_stp	17	35	15	12		12	15	25	65	35	23	15	12	14	19	43	19	21	36	17	32	32	38	23	23	19	23	38	25
c-ptl-tkt	18	42	46	46	54		17	30	75	40	29	25	25	50	29	67	33	42	54	21	54	50	50	46	42	29	38	62	41
c-tkt-tkt	17	42	42	35	50	12		25	80	35	27	20	38	54	27	65	46	38	58	12	54	54	50	54	42	31	42	65	41
clh_spin	26	40	40	45	45	20	35		55	40	20	30	30	30	20	55	30	35	50	20	50	55	55	60	45	30	45	50	39
clh_stp	32	35	5	15	10	15	20	15		35	25	25	15	10	15	10	20	5	25	10	10	10	20	20	10	15	15	25	17
clh-ls	21	15	40	35	45	20	25	0	55		20	25	30	30	15	55	10	25	55	15	50	55	55	60	40	35	40	50	34
hmcs	12	38	42	35	38	4	23	35	75	40		15	23	38	23	58	35	35	58	21	46	46	46	42	38	35	35	50	37
hticket-ls	16	40	55	40	55	0	10	35	75	30	15		20	45	15	65	25	35	55	20	60	55	55	50	45	30	45	60	39
malth_spin	12	38	19	27	50	12	15	25	65	35	23	15		31	15	46	27	31	50	12	46	46	38	38	31	19	19	46	31
malth_stp	21	38	23	35	39	21	15	30	65	35	31	20	8		15	39	23	25	54	12	54	46	38	35	31	23	23	46	31
mcs_spin	29	54	46	38	65	29	23	40	70	40	42	40	38	46		50	46	31	65	21	54	54	42	54	46	35	42	69	45
mcs_stp	25	35	12	27	29	25	15	30	35	30	31	25	15	14	8		27	14	39	17	29	29	12	12	12	15	12	31	22
mcs-ls	21	27	38	38	50	8	15	15	70	15	23	20	31	27	8	46		12	62	8	50	54	46	46	38	15	35	54	32
mcs-timepub	29	38	27	35	50	17	12	35	70	35	35	20	38	36	8	43	19		61	17	46	50	42	54	42	27	35	62	36
mutexee	17	31	8	19	21	12	12	20	60	30	27	20	8	4	19	36	15	14		12	29	21	31	27	19	12	15	27	21
partitioned	23	38	38	33	62	25	21	35	70	35	33	35	38	42	25	67	38	38	62		46	50	46	54	33	38	38	62	42
pthread	25	38	4	23	29	21	15	30	60	35	35	25	23	18	23	46	31	18	21	12		18	27	19	15	12	19	42	25
pthreadadapt	25	38	8	23	32	29	19	30	55	35	31	25	19	18	23	43	31	18	36	12	36		19	19	15	19	19	42	27
spinlock	25	38	15	38	38	33	23	30	55	35	38	30	38	38	23	42	35	19	50	21	38	38		27	12	27	19	31	32
spinlock-ls	25	35	15	31	31	12	19	20	50	25	35	10	23	23	27	54	23	23	42	17	38	31	35		19	12	8	23	26
ticket	25	31	12	31	38	25	23	30	60	30	35	30	31	31	27	54	31	23	50	12	35	50	23	31		15	8	27	30
ticket-ls	17	35	31	31	54	17	19	25	70	30	27	10	31	42	27	58	27	31	58	12	46	58	46	38	35		27	46	35
ttas	21	31	15	38	35	21	19	20	55	30	31	25	27	35	31	50	27	23	50	17	42	46	35	27	15	15		31	30
ttas-ls	17	23	15	23	23	4	12	5	35	20	12	5	15	23	19	38	15	15	42	4	35	46	27	15	12	8	0		19
average	22	36	27	32	42	17	19	25	63	32	28	22	26	32	21	50	28	25	50	16	43	44	39	38	30	24	27	46	22

Table 39. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**A-48 machine**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		20	25	35	60	28	25	33	73	20	5	20	30	70	30	60	20	45	60	28	65	65	50	50	55	20	40	45	40
alock-ls	20		25	30	65	28	30	20	73	20	10	33	30	65	15	60	10	35	65	22	70	70	50	50	50	20	35	35	38
backoff	35	35		52	71	32	33	40	87	40	29	31	48	71	33	67	29	33	52	26	62	62	48	38	52	33	38	50	45
c-bo-mcs_spin	30	30	14		57	11	14	33	87	40	19	19	33	67	38	71	19	48	57	26	71	62	52	43	52	33	38	60	42
c-bo-mcs_stp	25	25	10	10		5	10	33	80	27	10	6	14	43	24	61	10	22	13	11	30	13	29	5	14	10	14	25	21
c-ptl-tkt	28	17	26	32	63		16	33	87	33	11	19	37	68	37	74	26	53	53	32	68	63	58	58	53	32	47	72	44
c-tkt-tkt	20	25	14	29	57	5		27	87	27	14	12	38	76	24	71	24	52	57	16	71	67	57	52	52	24	33	50	40
clh_spin	27	13	33	13	67	13	33		73	27	13	27	33	73	13	67	7	40	53	33	73	73	60	53	53	40	40	60	41
clh_stp	27	13	7	7	0	7	7	13		27	7	7	7	7	7	27	7	0	0	7	0	0	7	0	7	7	7	7	8
clh-ls	20	0	27	20	67	20	27	13	73		13	33	40	73	27	67	13	47	53	27	73	60	60	60	53	27	53	53	41
hmcs	25	35	33	43	67	32	33	47	87	40		19	38	71	33	71	24	57	62	32	71	62	62	48	52	29	38	55	47
hticket-ls	20	27	19	25	62	6	12	33	87	27	12		38	69	38	75	12	50	56	19	69	69	62	56	56	25	50	73	43
malth_spin	20	35	10	19	71	11	14	33	87	40	14	12		62	24	67	19	29	57	11	67	67	43	38	38	24	24	50	36
malth_stp	25	25	5	29	22	16	14	27	60	27	14	19	10		19	48	19	13	13	11	13	9	14	0	14	10	5	20	18
mcs_spin	30	35	33	43	67	42	38	40	93	40	19	44	43	76		62	29	33	62	21	67	67	52	52	52	33	43	50	47
mcs_stp	25	30	5	24	22	26	19	27	33	27	19	19	14	13	14		19	4	13	16	9	13	5	5	19	19	10	15	17
mcs-ls	25	25	24	38	67	21	29	33	87	33	14	25	48	71	24	67		48	62	21	67	67	57	43	52	24	29	50	43
mcs-timepub	30	30	19	43	70	32	33	33	93	33	19	31	43	61	10	61	29		57	21	61	57	52	43	48	33	29	50	41
mutexee	35	35	14	24	48	26	24	40	87	40	24	25	29	65	29	70	29	22		26	39	17	33	19	29	24	24	35	34
partitioned	33	22	11	26	68	16	21	20	87	27	21	25	37	68	21	74	16	32	58		68	63	58	47	47	32	42	61	41
pthread	25	25	10	24	48	16	14	27	93	27	24	25	19	74	19	70	19	17	9	16		4	19	10	19	10	24	30	26
pthreadadapt	25	25	5	24	57	16	14	27	93	27	24	25	19	74	24	74	19	22	30	11	43		29	10	24	14	19	30	30
spinlock	30	40	14	43	57	37	33	33	87	33	29	31	29	71	19	71	29	14	48	26	57	43		14	33	24	14	25	37
spinlock-ls	35	40	10	29	57	21	19	27	87	27	19	12	29	76	29	76	19	24	43	16	48	29	33		29	19	5	20	32
ticket	20	15	5	24	52	11	14	13	87	13	10	12	19	67	14	67	10	19	38	5	57	33	29	5		5	10	10	25
ticket-ls	20	25	10	24	62	16	19	33	87	33	19	12	29	71	29	67	24	38	57	11	67	57	52	43	48		38	45	38
ttas	30	30	5	24	67	26	24	27	87	33	19	19	33	71	29	67	19	24	48	21	52	52	38	14	38	14		25	35
ttas-ls	20	30	5	20	55	11	10	13	73	27	5	7	10	65	10	60	5	15	45	6	55	40	30	15	35	5	10		25
average	26	26	15	28	56	20	22	29	82	30	16	21	29	65	23	66	19	31	45	19	55	48	42	32	40	22	28	41	26

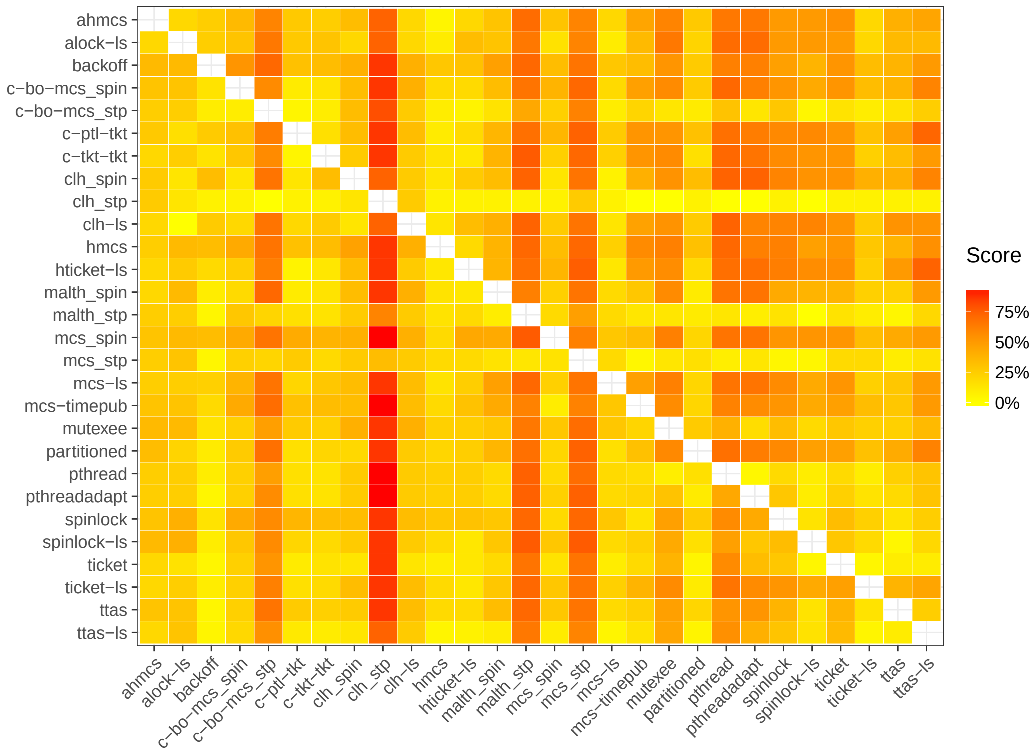

Fig. 8. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**A-48 machine**).

Table 40. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-48 machine in performance mode**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		42	53	37	58	18	26	50	79	57	11	36	58	53	53	74	58	53	58	47	58	63	68	63	63	53	58	53	52
alock-ls	32		30	15	50	6	10	60	93	60	10	20	50	50	15	75	25	50	60	50	70	60	65	70	70	50	50	65	47
backoff	32	40		30	60	22	15	47	87	60	35	27	65	55	40	75	45	45	65	39	60	60	50	55	75	65	45	65	50
c-bo-mcs_spin	37	50	55		60	17	15	73	93	73	25	33	60	70	45	80	50	65	70	56	70	70	70	75	75	70	70	85	60
c-bo-mcs_stp	26	30	25	10		6	10	47	87	47	15	7	35	36	25	68	25	41	36	17	41	36	55	50	45	35	40	60	35
c-ptl-tkt	29	72	56	33	72		11	87	93	93	11	40	78	78	61	83	67	78	67	61	67	67	72	72	72	67	72	83	65
c-tkt-tkt	42	55	50	35	75	17		87	93	93	15	40	80	85	60	85	65	80	75	67	75	75	75	75	75	70	65	90	67
clh_spin	21	0	27	13	47	7	0		73	7	0	7	47	40	0	67	13	20	60	33	67	67	60	60	67	53	53	60	36
clh_stp	21	7	13	7	0	7	7	7		7	7	7	7	0	7	27	7	7	0	7	0	0	7	7	7	7	7	7	7
clh-ls	14	0	27	7	47	0	0	7	73		0	7	40	40	0	67	7	13	53	47	60	60	60	60	60	40	40	53	33
hmcs	37	65	55	40	60	22	25	80	93	87		40	65	65	50	70	60	65	70	72	70	70	70	70	70	65	60	85	62
hticket-ls	29	67	40	20	67	7	0	67	93	73	13		47	60	33	73	33	47	67	60	73	67	67	73	73	67	67	87	54
malth_spin	32	25	20	5	45	0	0	47	93	53	15	13		40	10	60	10	20	45	11	50	45	55	55	60	35	50	60	35
malth_stp	32	30	20	15	41	11	10	33	93	40	25	7	10		10	59	15	23	27	17	32	27	55	50	50	25	25	40	30
mcs_spin	32	25	30	20	45	0	0	67	93	60	10	20	50	55		60	30	60	70	44	65	70	55	50	70	65	50	70	47
mcs_stp	21	25	20	15	14	11	10	27	40	27	15	7	25	14	10		20	23	14	11	9	18	15	10	25	20	5	25	18
mcs-ls	26	15	25	15	45	0	0	67	93	67	10	20	40	45	5	60		45	55	39	55	60	55	55	65	55	50	60	42
mcs-timepub	37	30	30	15	41	11	10	33	93	47	15	13	45	45	10	55	20		59	28	55	59	55	50	75	50	35	55	40
mutexee	26	30	15	20	41	17	15	27	93	33	25	20	35	27	20	68	25	23		17	32	27	50	35	55	30	15	35	32
partitioned	24	17	22	17	56	6	0	27	93	33	17	20	50	61	22	89	28	28	56		67	67	67	72	72	50	61	72	44
pthread	26	25	20	20	36	17	10	27	87	33	25	20	35	27	25	73	35	23	9	17		18	45	30	40	20	10	30	29
pthreadadapt	26	30	10	20	41	17	10	33	87	40	25	20	25	18	20	68	25	18	9	17	27		45	40	35	20	15	30	29
spinlock	26	30	10	20	25	17	10	27	73	27	25	20	35	30	25	50	35	20	25	17	20	30		5	30	20	0	20	25
spinlock-ls	26	30	15	20	30	17	10	27	73	27	25	20	35	25	25	65	40	25	25	11	35	35	35		30	20	5	20	28
ticket	16	20	10	10	35	6	0	20	87	20	15	20	15	15	10	65	20	10	20	6	20	20	45	35		0	5	20	21
ticket-ls	26	25	10	15	55	11	5	20	93	27	20	20	25	45	20	65	25	15	30	11	40	35	55	55	55		20	45	32
ttas	26	30	20	20	50	17	15	27	93	27	30	20	35	40	25	85	40	30	35	17	35	50	60	55	50	25		35	37
ttas-ls	26	25	20	10	30	11	10	27	87	33	15	7	25	25	15	60	30	10	35	11	40	40	50	40	45	20	5		28
average	28	31	27	19	45	11	9	42	86	46	17	20	41	42	24	68	32	35	44	31	48	48	54	51	56	41	36	52	28

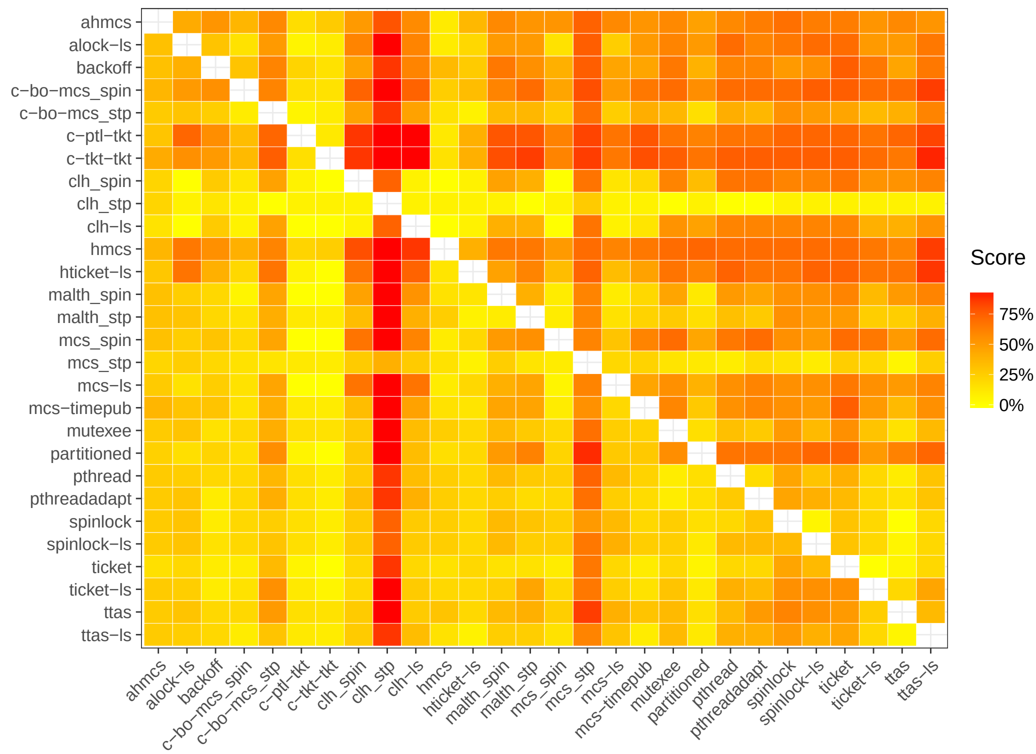

Fig. 9. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-48 machine in performance mode**).

Table 41. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-20 machine in performance mode**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		20	33	7	67	0	7	27	64	27	7	0	33	47	33	67	33	27	53	29	47	47	40	27	40	33	33	33	33
alock-ls	40		25	19	62	20	19	42	92	42	25	8	31	38	12	62	12	12	50	33	44	44	38	31	19	19	19	25	33
backoff	33	44		19	56	20	12	50	92	50	19	17	25	25	25	56	25	12	31	27	31	31	31	25	38	31	19	38	33
c-bo-mcs_spin	27	50	38		62	20	6	58	92	58	19	8	25	38	25	56	25	25	50	27	44	38	31	25	31	31	25	56	37
c-bo-mcs_stp	33	38	12	12		13	12	42	33	42	19	8	19	6	19	12	19	12	12	13	18	12	12	12	25	25	6	38	19
c-ptl-tkt	29	53	40	13	60		13	67	92	67	13	8	53	47	47	67	33	33	47	27	40	40	47	40	47	33	33	60	43
c-tkt-tkt	33	44	38	19	69	13		58	92	58	12	8	38	44	25	62	38	31	50	27	44	38	38	31	44	31	19	56	39
clh_spin	18	0	25	8	58	8	8		58	8	8	8	25	17	0	58	0	0	42	25	42	33	33	25	8	8	8	8	20
clh_stp	27	8	8	8	0	8	8	8		8	8	8	8	0	8	0	8	8	0	8	0	0	8	8	8	8	8	8	7
clh-ls	27	0	25	8	58	8	8	0	58		8	8	17	17	0	58	0	0	42	17	42	33	33	25	8	8	8	17	20
hmcs	27	44	38	12	62	7	6	58	92	58		0	31	38	19	56	25	25	50	20	44	44	38	31	38	25	31	56	36
hticket-ls	27	58	33	8	67	0	0	58	92	58	8		33	33	25	58	25	33	42	25	42	33	33	33	33	25	25	67	36
malth_spin	27	31	12	6	50	7	0	42	92	42	12	0		25	0	50	0	6	44	20	38	38	31	19	31	31	12	44	26
malth_stp	33	31	19	12	59	13	12	42	92	42	19	8	19		12	53	19	12	47	27	35	41	38	25	31	25	12	38	30
mcs_spin	33	31	38	12	56	13	6	42	92	42	19	8	31	31		50	12	12	50	27	44	38	38	19	31	25	6	44	31
mcs_stp	33	38	19	19	18	13	12	42	42	42	19	8	25	6	12		25	6	12	13	12	12	19	6	25	25	6	31	20
mcs-ls	33	31	31	19	56	13	6	42	92	50	19	8	31	31	0	56		6	44	33	38	44	38	25	38	31	19	50	33
mcs-timepub	33	38	44	19	59	13	19	42	92	50	19	8	25	24	19	59	25		53	27	47	47	44	25	31	31	19	50	36
mutexee	33	38	6	19	47	20	19	42	83	42	19	17	19	6	19	47	19	12		13	18	0	6	6	19	12	6	31	23
partitioned	43	27	27	20	60	20	13	33	92	33	20	17	27	20	13	53	20	13	40		40	33	33	20	13	13	7	40	29
pthread	33	38	12	25	53	27	25	42	75	42	25	17	25	12	25	53	25	18	18	27		12	19	19	38	25	12	38	29
pthreadadapt	33	38	12	19	47	20	19	42	75	42	19	17	19	6	19	47	19	12	24	20	18		12	12	25	25	12	31	25
spinlock	33	31	6	19	50	20	12	42	75	42	19	17	19	12	19	44	19	6	31	13	31	12		0	19	19	0	25	24
spinlock-ls	33	31	25	25	56	20	19	42	92	42	19	17	31	31	31	62	31	12	44	20	44	38	25		25	25	6	25	32
ticket	33	25	12	12	50	20	6	33	92	33	19	17	12	12	6	50	6	6	38	13	38	31	31	12		6	0	31	24
ticket-ls	33	31	19	12	56	20	6	33	92	33	19	17	19	25	6	56	6	12	44	27	38	38	25	19	12		6	50	28
ttas	33	38	31	19	56	20	12	42	92	42	19	17	25	25	25	56	25	12	44	20	44	38	25	12	25	25		38	32
ttas-ls	33	19	19	19	56	13	12	33	92	33	12	8	25	12	19	50	25	6	38	13	38	38	25	12	25	25	6		26
average	32	32	24	15	54	15	11	41	82	42	16	10	26	23	17	52	19	14	38	22	35	31	29	20	27	23	14	38	32

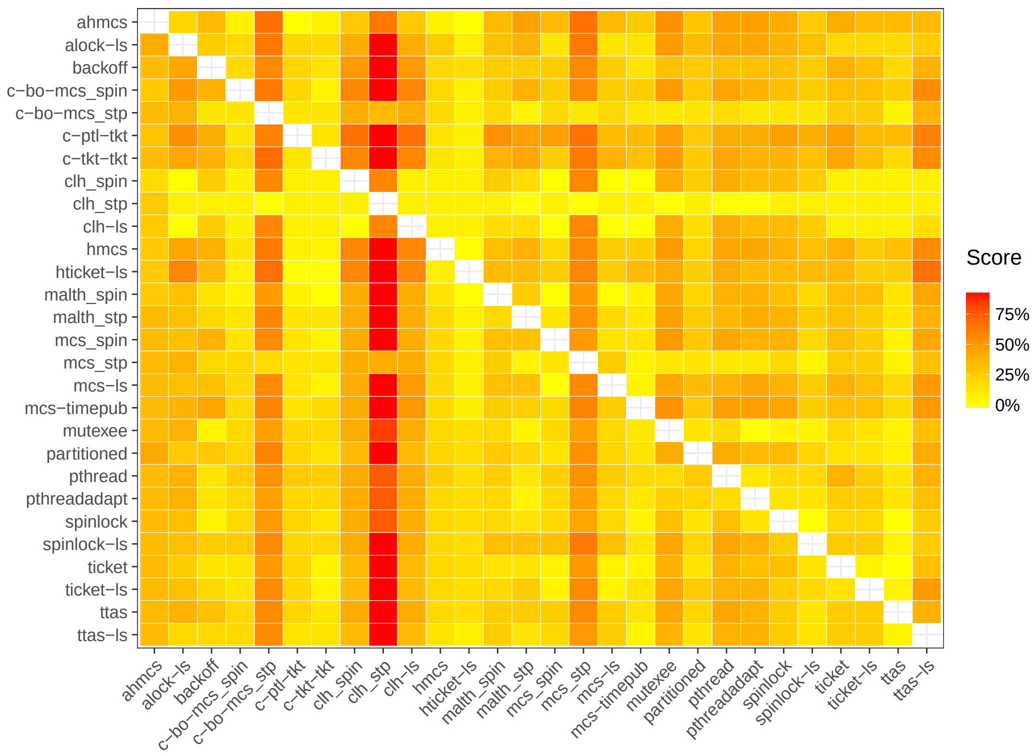

Fig. 10. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-20 machine in performance mode**).

Table 42. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**A-64 machine with thread-to-node pinning**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		27	41	41	77	25	45	18	76	29	14	35	41	59	23	68	41	41	59	25	64	59	68	45	36	27	36	45	43
alock-ls	23		38	33	67	23	29	17	89	22	21	11	38	50	12	67	17	33	54	9	62	58	67	46	29	17	33	42	37
backoff	32	38		38	58	27	29	33	78	33	38	22	38	38	33	58	33	33	58	32	62	54	71	42	46	29	21	42	41
c-bo-mcs_spin	18	38	25		62	14	17	33	83	50	17	11	29	29	21	54	25	21	54	32	67	54	67	42	33	25	29	46	37
c-bo-mcs_stp	18	25	0	12		9	8	22	39	22	17	6	17	8	12	23	17	12	23	23	23	27	33	12	17	12	8	25	17
c-ptl-tkt	25	32	41	36	73		14	17	89	28	14	22	36	55	27	73	27	36	59	32	68	59	73	50	45	27	36	59	43
c-tkt-tkt	18	29	38	42	79	18		17	89	28	25	17	38	54	17	71	25	29	54	32	71	58	67	54	46	25	33	62	42
clh_spin	29	22	39	17	72	28	28		72	33	6	17	22	39	0	67	22	33	56	22	61	56	67	56	39	33	33	44	38
clh_stp	18	6	0	6	6	6	6	6		6	6	6	6	0	6	11	6	0	17	6	17	17	33	6	6	6	6	0	8
clh-ls	18	0	28	17	56	11	17	6	72		0	11	22	33	0	61	6	17	56	11	56	56	67	50	33	22	33	44	30
hmcs	27	42	42	42	67	32	29	33	94	44		33	33	46	17	62	33	38	54	32	58	54	67	46	42	29	38	58	44
hticket-ls	18	28	44	17	78	11	22	17	89	33	6		17	39	0	67	11	17	61	22	67	61	72	61	39	28	39	61	38
malth_spin	23	29	29	21	67	14	21	28	83	39	17	17		25	8	54	12	12	50	18	50	46	54	46	33	17	25	54	33
malth_stp	27	42	25	25	58	23	25	39	61	44	29	28	25		25	35	25	31	50	32	46	42	50	38	38	25	25	38	35
mcs_spin	27	42	33	33	71	32	38	22	89	44	25	22	33	50		58	42	38	50	23	58	58	62	42	42	25	33	54	42
mcs_stp	18	29	4	17	31	9	17	22	28	28	21	11	21	8	12		21	19	15	14	23	19	42	17	12	8	4	12	18
mcs-ls	23	29	29	25	71	14	21	22	89	28	21	17	29	33	4	50		12	50	27	58	46	54	46	42	21	29	50	35
mcs-timepub	27	25	21	29	65	9	21	22	83	28	25	17	38	38	21	46	21		50	27	54	50	67	46	42	25	25	46	36
mutexee	23	25	8	21	38	14	17	22	61	22	25	11	25	15	21	42	25	23		14	23	31	50	25	17	12	12	25	24
partitioned	25	32	41	23	64	18	23	17	89	28	18	11	27	41	14	68	27	36	64		64	59	68	45	36	32	27	68	39
pthread	18	25	8	17	38	14	17	22	56	22	29	11	25	15	21	35	25	19	12	18		8	38	21	17	12	12	25	21
pthreadadapt	18	25	8	17	38	14	17	22	61	22	29	11	21	12	17	38	21	23	31	18	35		46	17	21	17	12	25	24
spinlock	18	25	4	17	38	14	12	22	39	22	29	11	17	8	17	25	17	12	17	14	25	4		4	12	12	4	17	17
spinlock-ls	32	38	8	29	50	23	25	22	50	22	29	11	29	21	29	50	29	25	38	18	38	33	50		33	21	4	29	29
ticket	18	25	8	29	54	14	21	22	78	28	25	11	21	33	21	58	21	25	50	14	54	46	67	29		0	8	38	30
ticket-ls	27	38	17	38	71	32	38	33	89	33	42	22	38	46	29	71	38	33	62	23	67	58	71	42	42		25	54	44
ttas	32	42	8	42	67	27	29	28	72	28	29	17	29	42	29	67	33	33	50	18	58	54	67	25	29	21		33	37
ttas-ls	23	25	12	25	50	9	21	22	56	28	17	6	25	17	21	46	25	17	42	9	50	42	58	21	21	17	4		26
average	23	29	22	26	58	18	22	22	72	29	21	16	27	32	17	53	24	25	46	21	51	45	59	36	31	20	22	41	23

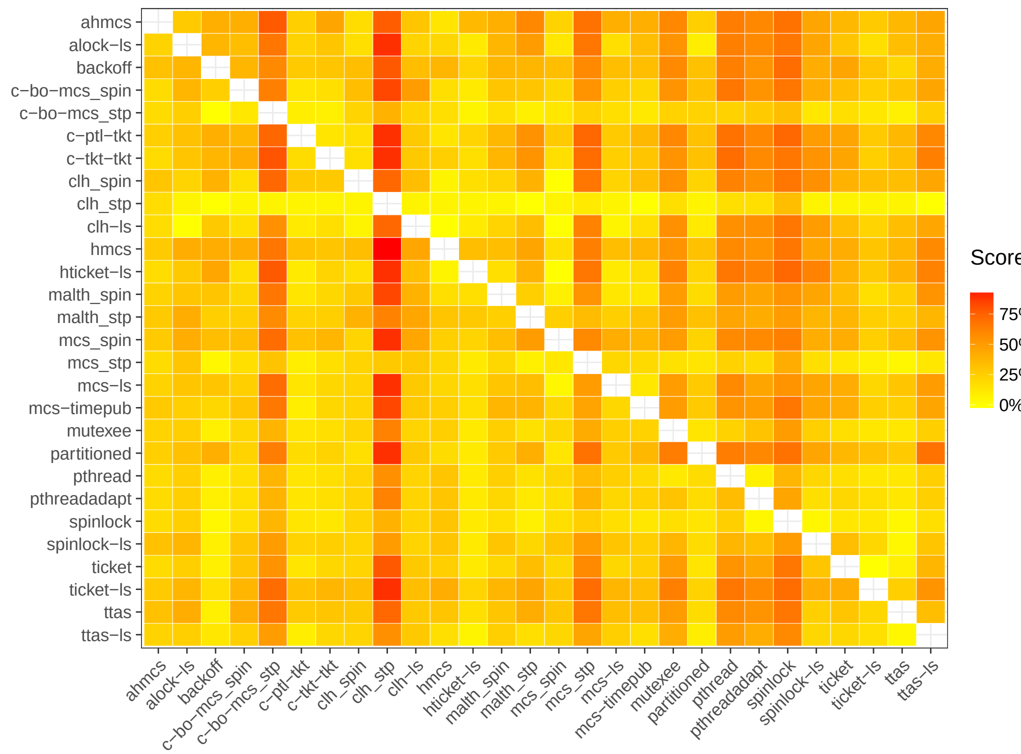

Fig. 11. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**A-64-node machine**).

Table 43. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-48 machine in energy-saving mode**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		42	53	37	58	18	26	50	79	57	11	36	58	53	53	74	58	53	58	47	58	63	68	63	63	53	58	53	52
alock-ls	32		30	15	50	6	10	60	93	60	10	20	50	50	15	75	25	50	60	50	70	60	65	70	70	50	50	65	47
backoff	32	40		30	60	22	15	47	87	60	35	27	65	55	40	75	45	45	65	39	60	60	50	55	75	65	45	65	50
c-bo-mcs_spin	37	50	55		60	17	15	73	93	73	25	33	60	70	45	80	50	65	70	56	70	70	70	75	75	70	70	85	60
c-bo-mcs_stp	26	30	25	10		6	10	47	87	47	15	7	35	36	25	68	25	41	36	17	41	36	55	50	45	35	40	60	35
c-ptl-tkt	29	72	56	33	72		11	87	93	93	11	40	78	78	61	83	67	78	67	61	67	67	72	72	72	67	72	83	65
c-tkt-tkt	42	55	50	35	75	17		87	93	93	15	40	80	85	60	85	65	80	75	67	75	75	75	75	75	70	65	90	67
clh_spin	21	0	27	13	47	7	0		73	7	0	7	47	40	0	67	13	20	60	33	67	67	60	60	67	53	53	60	36
clh_stp	21	7	13	7	0	7	7	7		7	7	7	7	0	7	27	7	7	0	7	0	0	7	7	7	7	7	7	7
clh-ls	14	0	27	7	47	0	0	7	73		0	7	40	40	0	67	7	13	53	47	60	60	60	60	60	40	40	53	33
hmcs	37	65	55	40	60	22	25	80	93	87		40	65	65	50	70	60	65	70	72	70	70	70	70	70	65	60	85	62
hticket-ls	29	67	40	20	67	7	0	67	93	73	13		47	60	33	73	33	47	67	60	73	67	67	73	73	67	67	87	54
malth_spin	32	25	20	5	45	0	0	47	93	53	15	13		40	10	60	10	20	45	11	50	45	55	55	60	35	50	60	35
malth_stp	32	30	20	15	41	11	10	33	93	40	25	7	10		10	59	15	23	27	17	32	27	55	50	50	25	25	40	30
mcs_spin	32	25	30	20	45	0	0	67	93	60	10	20	50	55		60	30	60	70	44	65	70	55	50	70	65	50	70	47
mcs_stp	21	25	20	15	14	11	10	27	40	27	15	7	25	14	10		20	23	14	11	9	18	15	10	25	20	5	25	18
mcs-ls	26	15	25	15	45	0	0	67	93	67	10	20	40	45	5	60		45	55	39	55	60	55	55	65	55	50	60	42
mcs-timepub	37	30	30	15	41	11	10	33	93	47	15	13	45	45	10	55	20		59	28	55	59	55	50	75	50	35	55	40
mutexee	26	30	15	20	41	17	15	27	93	33	25	20	35	27	20	68	25	23		17	32	27	50	35	55	30	15	35	32
partitioned	24	17	22	17	56	6	0	27	93	33	17	20	50	61	22	89	28	28	56		67	67	67	72	72	50	61	72	44
pthread	26	25	20	20	36	17	10	27	87	33	25	20	35	27	25	73	35	23	9	17		18	45	30	40	20	10	30	29
pthreadadapt	26	30	10	20	41	17	10	33	87	40	25	20	25	18	20	68	25	18	9	17	27		45	40	35	20	15	30	29
spinlock	26	30	10	20	25	17	10	27	73	27	25	20	35	30	25	50	35	20	25	17	20	30		5	30	20	0	20	25
spinlock-ls	26	30	15	20	30	17	10	27	73	27	25	20	35	25	25	65	40	25	25	11	35	35	35		30	20	5	20	28
ticket	16	20	10	10	35	6	0	20	87	20	15	20	15	15	10	65	20	10	20	6	20	20	45	35		0	5	20	21
ticket-ls	26	25	10	15	55	11	5	20	93	27	20	20	25	45	20	65	25	15	30	11	40	35	55	55	55		20	45	32
ttas	26	30	20	20	50	17	15	27	93	27	30	20	35	40	25	85	40	30	35	17	35	50	60	55	50	25		35	37
ttas-ls	26	25	20	10	30	11	10	27	87	33	15	7	25	25	15	60	30	10	35	11	40	40	50	40	45	20	5		28
average	28	31	27	19	45	11	9	42	86	46	17	20	41	42	24	68	32	35	44	31	48	48	54	51	56	41	36	52	28

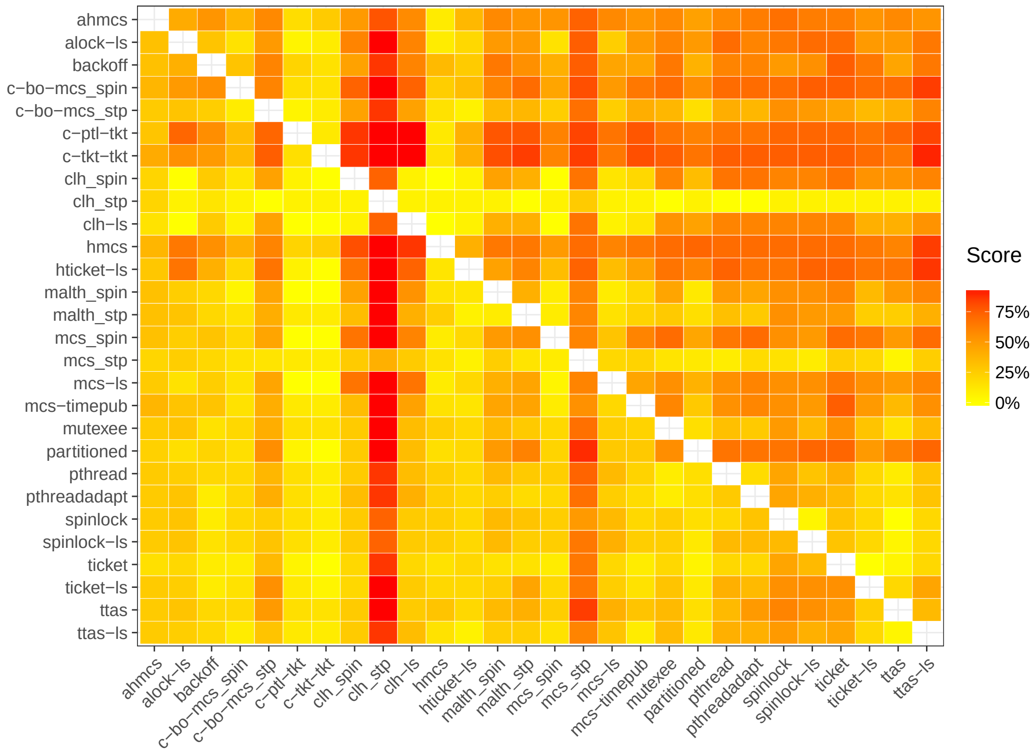

Fig. 12. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-48 machine in energy-saving move**).

Table 44. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-20 machine in energy-saving mode**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		20	33	7	67	0	7	27	64	27	7	0	33	47	33	67	33	27	53	29	47	47	40	27	40	33	33	33	33
alock-ls	40		25	19	62	20	19	42	92	42	25	8	31	38	12	62	12	12	50	33	44	44	38	31	19	19	19	25	33
backoff	33	44		19	56	20	12	50	92	50	19	17	25	25	25	56	25	12	31	27	31	31	31	25	38	31	19	38	33
c-bo-mcs_spin	27	50	38		62	20	6	58	92	58	19	8	25	38	25	56	25	25	50	27	44	38	31	25	31	31	25	56	37
c-bo-mcs_stp	33	38	12	12		13	12	42	33	42	19	8	19	6	19	12	19	12	12	13	18	12	12	12	25	25	6	38	19
c-ptl-tkt	29	53	40	13	60		13	67	92	67	13	8	53	47	47	67	33	33	47	27	40	40	47	40	47	33	33	60	43
c-tkt-tkt	33	44	38	19	69	13		58	92	58	12	8	38	44	25	62	38	31	50	27	44	38	38	31	44	31	19	56	39
clh_spin	18	0	25	8	58	8	8		58	8	8	8	25	17	0	58	0	0	42	25	42	33	33	25	8	8	8	8	20
clh_stp	27	8	8	8	0	8	8	8		8	8	8	8	0	8	0	8	8	0	8	0	0	8	8	8	8	8	8	7
clh-ls	27	0	25	8	58	8	8	0	58		8	8	17	17	0	58	0	0	42	17	42	33	33	25	8	8	8	17	20
hmcs	27	44	38	12	62	7	6	58	92	58		0	31	38	19	56	25	25	50	20	44	44	38	31	38	25	31	56	36
hticket-ls	27	58	33	8	67	0	0	58	92	58	8		33	33	25	58	25	33	42	25	42	33	33	33	33	25	25	67	36
malth_spin	27	31	12	6	50	7	0	42	92	42	12	0		25	0	50	0	6	44	20	38	38	31	19	31	31	12	44	26
malth_stp	33	31	19	12	59	13	12	42	92	42	19	8	19		12	53	19	12	47	27	35	41	38	25	31	25	12	38	30
mcs_spin	33	31	38	12	56	13	6	42	92	42	19	8	31	31		50	12	12	50	27	44	38	38	19	31	25	6	44	31
mcs_stp	33	38	19	19	18	13	12	42	42	42	19	8	25	6	12		25	6	12	13	12	12	19	6	25	25	6	31	20
mcs-ls	33	31	31	19	56	13	6	42	92	50	19	8	31	31	0	56		6	44	33	38	44	38	25	38	31	19	50	33
mcs-timepub	33	38	44	19	59	13	19	42	92	50	19	8	25	24	19	59	25		53	27	47	47	44	25	31	31	19	50	36
mutexee	33	38	6	19	47	20	19	42	83	42	19	17	19	6	19	47	19	12		13	18	0	6	6	19	12	6	31	23
partitioned	43	27	27	20	60	20	13	33	92	33	20	17	27	20	13	53	20	13	40		40	33	33	20	13	13	7	40	29
pthread	33	38	12	25	53	27	25	42	75	42	25	17	25	12	25	53	25	18	18	27		12	19	19	38	25	12	38	29
pthreadadapt	33	38	12	19	47	20	19	42	75	42	19	17	19	6	19	47	19	12	24	20	18		12	12	25	25	12	31	25
spinlock	33	31	6	19	50	20	12	42	75	42	19	17	19	12	19	44	19	6	31	13	31	12		0	19	19	0	25	24
spinlock-ls	33	31	25	25	56	20	19	42	92	42	19	17	31	31	31	62	31	12	44	20	44	38	25		25	25	6	25	32
ticket	33	25	12	12	50	20	6	33	92	33	19	17	12	12	6	50	6	6	38	13	38	31	31	12		6	0	31	24
ticket-ls	33	31	19	12	56	20	6	33	92	33	19	17	19	25	6	56	6	12	44	27	38	38	25	19	12		6	50	28
ttas	33	38	31	19	56	20	12	42	92	42	19	17	25	25	25	56	25	12	44	20	44	38	25	12	25	25		38	32
ttas-ls	33	19	19	19	56	13	12	33	92	33	12	8	25	12	19	50	25	6	38	13	38	38	25	12	25	25	6		26
average	32	32	24	15	54	15	11	41	82	42	16	10	26	23	17	52	19	14	38	22	35	31	29	20	27	23	14	38	32

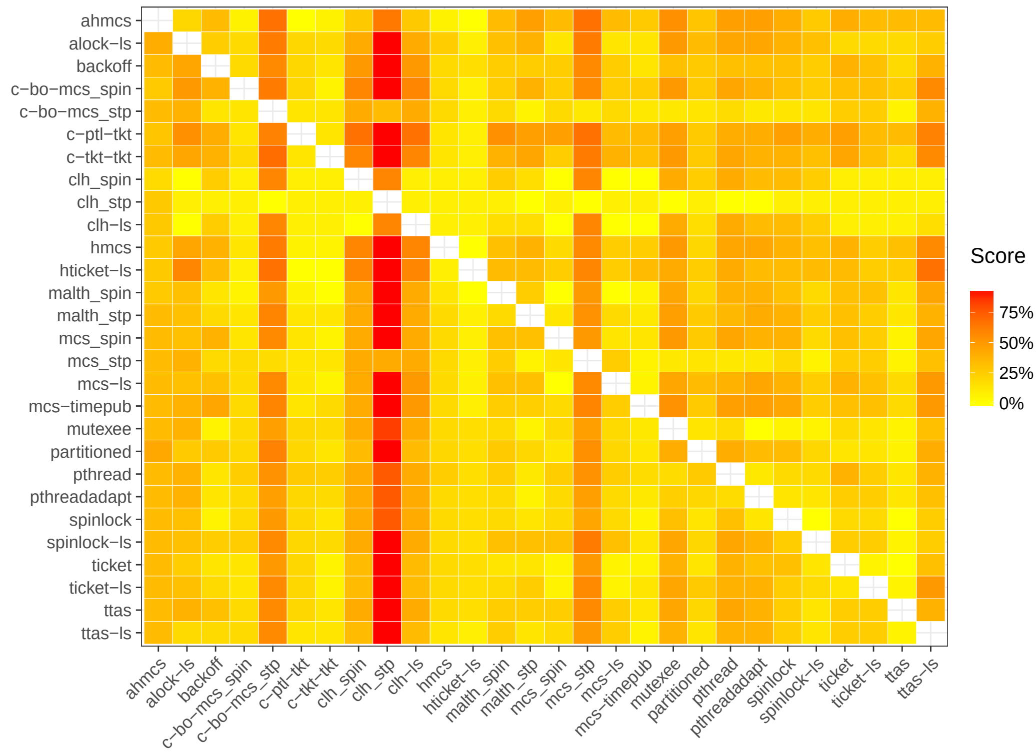

Fig. 13. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-20 machine in energy-saving move**).

A.5 At *max nodes*

Table 45. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**A-64 machine**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		46	50	33	67	32	29	42	58	42	21	42	50	42	58	67	58	33	46	45	54	46	71	71	67	54	58	67	50
alock-ls	42		38	27	62	8	8	25	55	25	23	0	31	35	31	62	38	19	54	38	54	46	58	58	54	38	50	62	38
backoff	46	54		35	81	33	31	55	70	60	42	40	42	38	46	73	65	27	54	50	46	46	69	58	69	54	50	85	53
c-bo-mcs_spin	42	69	42		69	42	35	70	70	70	42	40	35	42	54	65	58	23	50	46	54	54	65	69	58	62	62	85	54
c-bo-mcs_stp	29	38	12	15		17	19	30	40	35	23	15	19	14	19	39	19	11	11	17	14	14	54	35	27	15	31	46	24
c-ptl-tkt	45	71	42	50	71		29	65	75	65	46	25	38	42	50	75	50	42	54	62	50	42	62	67	67	50	54	83	54
c-tkt-tkt	46	69	50	46	77	21		50	80	55	38	25	46	54	54	77	69	38	62	50	58	50	69	69	69	54	62	85	56
clh_spin	16	40	30	10	55	15	20		50	40	15	15	10	25	15	60	35	25	50	40	45	35	60	70	50	40	55	60	36
clh_stp	37	35	10	15	40	20	20	20		35	20	20	15	5	20	10	20	0	10	15	5	5	60	40	15	20	20	25	21
clh-ls	32	35	30	15	65	15	15	30	60		30	15	10	30	35	60	35	25	50	50	50	45	60	65	55	40	50	65	40
hmcs	38	54	46	35	62	25	31	60	75	55		25	38	35	50	65	42	31	58	50	54	42	65	65	69	46	58	77	50
hticket-ls	47	60	50	25	65	25	40	70	75	55	45		30	30	50	65	55	35	50	60	50	45	65	60	65	45	60	85	52
malth_spin	38	58	27	31	58	29	31	75	80	70	42	35		31	42	65	46	23	50	54	46	42	62	58	65	42	46	77	49
malth_stp	42	54	38	27	68	38	31	70	80	60	46	40	23		38	54	42	21	50	54	57	43	58	69	54	58	54	85	50
mcs_spin	25	50	35	27	65	25	23	30	70	45	38	35	38	42		62	65	27	50	33	46	42	62	62	58	50	58	77	46
mcs_stp	29	38	15	19	43	25	19	40	30	35	27	25	19	14	19		27	7	11	21	14	14	54	31	23	23	19	38	25
mcs-ls	29	46	27	27	65	12	15	45	75	30	31	15	27	38	15	58		8	46	21	42	38	62	50	50	31	46	65	38
mcs-timepub	62	69	46	58	79	46	38	70	85	65	58	55	62	50	50	71	58		68	54	64	61	73	73	73	69	65	92	64
mutexee	42	42	27	46	61	25	27	40	80	50	38	30	35	32	46	75	46	18		33	36	32	69	58	54	31	69	85	45
partitioned	41	50	17	25	79	29	21	40	75	35	46	30	29	25	33	71	46	17	54		46	46	62	67	46	46	50	75	44
pthread	38	46	23	35	64	33	23	45	70	50	42	35	42	25	42	64	50	18	21	33		25	73	62	42	31	62	77	43
pthreadadapt	46	46	19	35	68	38	31	55	75	50	50	35	42	29	50	71	54	18	39	29	39		73	62	54	38	62	88	48
spinlock	25	38	4	31	38	33	19	35	30	40	35	30	31	27	27	31	35	15	15	25	8	8		15	15	23	12	31	25
spinlock-ls	29	42	8	19	46	21	19	30	55	35	35	15	35	19	27	54	38	15	15	25	8	15	73		31	19	31	46	30
ticket	25	35	4	19	62	25	15	30	70	35	31	25	23	27	23	69	42	12	27	17	23	8	65	46		12	42	65	32
ticket-ls	38	58	12	27	77	33	27	55	75	55	50	30	42	31	46	73	50	19	50	38	42	38	69	58	65		62	73	48
ttas	33	46	15	31	54	29	23	40	75	45	38	30	31	27	38	73	38	15	15	33	12	12	65	38	35	15		54	36
ttas-ls	29	31	4	8	42	12	8	10	45	30	12	5	19	15	19	50	27	4	12	17	12	12	65	27	27	12	35		22
average	37	49	27	28	62	26	24	45	66	47	36	27	32	31	37	61	45	20	40	37	38	34	65	56	50	38	49	69	37

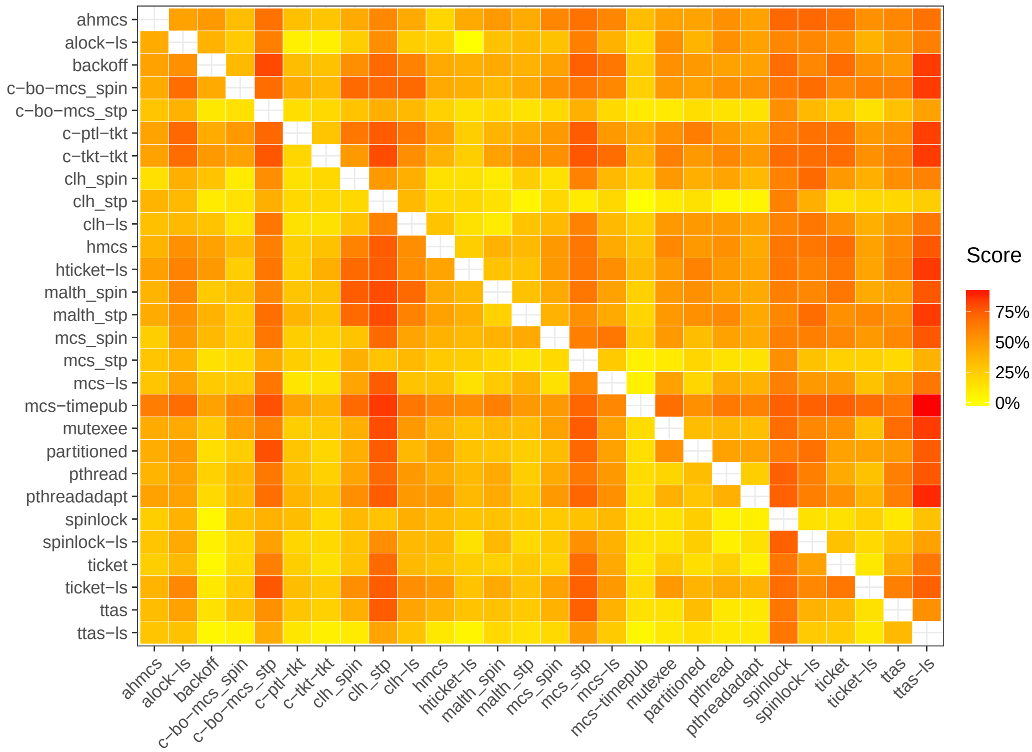

Fig. 14. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**A-64 machine**).

Table 46. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**A-48 machine**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		40	35	30	75	28	40	40	73	33	10	33	45	75	60	75	50	45	75	33	75	70	75	75	75	55	75	75	54
alock-ls	25		30	30	75	28	30	27	73	20	10	20	45	75	35	75	35	50	75	33	75	60	75	75	70	55	70	70	50
backoff	55	60		48	81	42	48	53	87	67	43	38	67	81	67	81	62	43	62	58	71	71	81	71	86	71	67	80	64
c-bo-mcs_spin	45	45	19		71	32	38	60	87	47	29	25	57	71	52	81	43	52	67	47	71	67	76	71	76	62	67	80	57
c-bo-mcs_stp	25	25	5	10		11	10	27	53	27	14	6	14	17	24	57	14	22	0	16	4	4	57	19	24	5	5	25	19
c-ptl-tkt	39	44	32	42	74		37	67	87	60	16	25	53	74	47	79	47	53	68	58	68	63	74	68	74	58	68	83	58
c-tkt-tkt	25	40	24	38	76	11		47	87	40	14	12	52	81	48	81	33	48	71	42	71	62	76	71	76	52	71	85	53
clh_spin	27	27	27	13	73	20	27		73	27	20	20	27	73	27	73	20	33	67	47	73	60	73	73	67	47	73	73	47
clh_stp	27	13	7	7	33	7	7	13		27	7	7	7	27	7	33	7	7	0	7	7	0	53	20	7	7	13	13	14
clh-ls	27	13	20	20	73	27	27	13	73		13	20	33	73	20	73	13	33	73	33	73	60	73	73	67	47	67	73	45
hmcs	30	50	38	38	71	37	33	60	87	53		19	48	71	62	81	43	57	71	53	71	67	76	71	76	57	71	85	58
hticket-ls	40	53	19	19	69	31	25	53	87	40	25		31	69	44	81	31	50	69	56	69	62	75	69	75	56	69	87	54
malth_spin	35	35	10	14	71	11	19	53	87	53	19	12		71	43	81	33	38	52	37	57	62	76	62	67	52	62	75	48
malth_stp	25	25	5	24	65	11	10	27	60	27	14	19	10		19	65	19	22	17	16	22	9	48	29	29	14	19	35	25
mcs_spin	35	45	24	33	76	37	38	33	93	40	29	38	43	76		71	29	19	62	47	67	62	71	76	67	52	67	75	52
mcs_stp	25	25	10	19	39	21	14	27	27	27	19	19	14	17	14		19	9	4	21	9	4	33	29	19	19	19	25	20
mcs-ls	25	25	14	24	67	21	19	27	87	33	19	12	43	71	38	81		38	62	32	62	57	76	62	67	43	57	75	46
mcs-timepub	40	50	24	38	78	32	43	53	93	53	38	38	48	70	48	65	52		52	53	65	65	76	76	81	76	62	75	57
mutexee	25	25	19	19	83	16	14	33	80	27	14	12	38	61	33	78	29	30		21	43	30	86	71	52	33	57	60	40
partitioned	28	39	11	26	74	11	16	27	87	33	21	25	32	68	32	74	16	21	63		68	63	74	68	63	53	74	78	46
pthread	25	25	10	24	78	16	14	27	87	27	24	25	33	65	24	83	29	22	9	21		13	86	57	52	14	48	60	37
pthreadadapt	30	35	5	29	74	21	19	27	93	33	29	25	29	78	24	87	24	26	43	21	52		86	62	62	29	71	75	44
spinlock	25	25	10	24	38	26	19	27	40	27	24	25	24	38	19	48	24	10	10	26	10	10		10	24	19	10	20	23
spinlock-ls	25	25	5	14	67	16	14	27	73	27	19	12	29	52	24	62	24	10	5	16	14	14	81		33	14	10	30	27
ticket	25	15	0	10	62	11	10	27	87	13	10	6	14	57	10	76	5	10	24	16	33	24	76	48		5	38	55	28
ticket-ls	25	30	0	14	81	16	19	27	87	40	24	12	33	76	38	81	24	19	48	21	57	48	81	62	81		62	75	44
ttas	25	20	5	14	76	16	14	27	80	27	14	12	29	48	29	76	24	24	10	16	33	14	81	43	33	19		50	32
ttas-ls	25	20	10	15	70	11	10	13	67	27	10	7	20	45	20	65	20	10	15	11	30	20	75	40	25	20	30		27
average	30	32	15	24	69	21	23	35	78	35	20	19	34	62	34	73	28	30	43	32	50	42	73	58	57	38	52	63	30

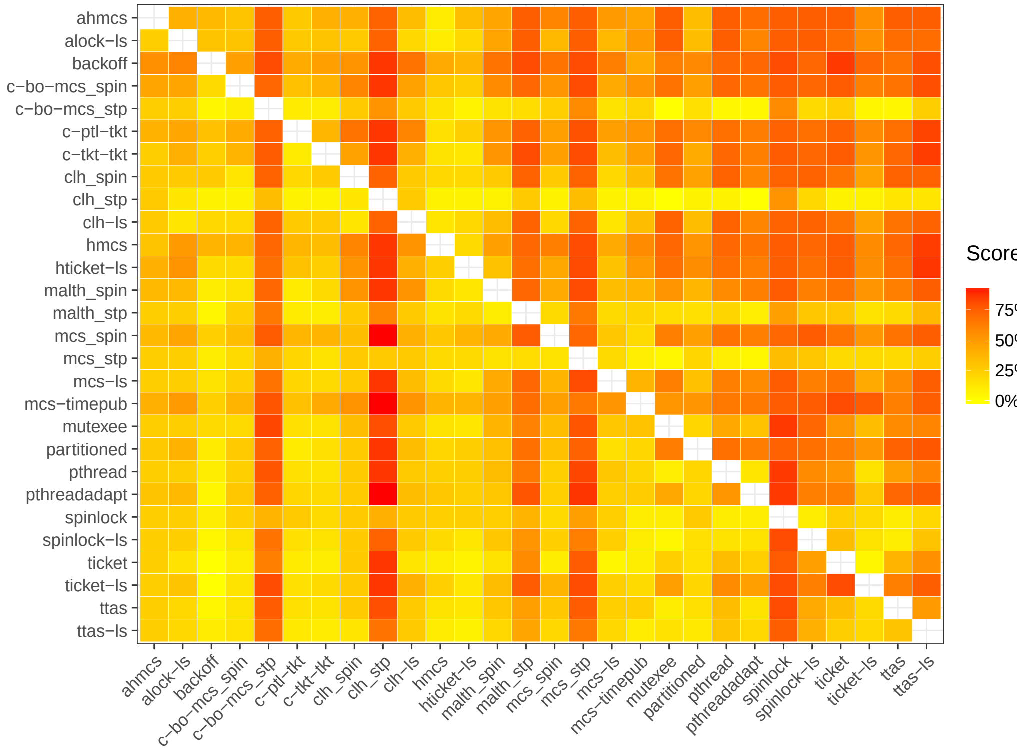

Fig. 15. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**A-48 machine**).

Table 47. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-48 machine in performance mode**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		42	58	42	74	47	26	57	79	64	11	36	58	58	53	74	58	58	63	47	63	63	79	74	68	58	74	68	57
alock-ls	47		35	40	75	33	25	60	87	53	30	27	55	55	30	70	25	55	60	67	75	65	75	75	75	70	70	75	56
backoff	37	45		35	70	28	25	53	87	60	30	40	55	50	35	70	45	50	55	61	65	65	60	55	80	65	55	70	54
c-bo-mcs_spin	42	50	50		70	28	30	73	93	73	25	47	70	70	55	80	55	50	70	56	70	70	75	75	75	75	70	85	62
c-bo-mcs_stp	21	25	25	15		17	15	27	87	27	20	13	20	14	20	59	20	23	9	17	9	23	60	55	35	25	15	30	27
c-ptl-tkt	24	56	50	39	78		11	80	87	87	6	33	61	67	56	78	56	61	67	50	67	67	72	72	67	67	67	78	59
c-tkt-tkt	53	55	45	45	80	33		73	87	87	35	33	75	70	50	80	50	60	75	61	75	75	75	75	70	70	70	80	64
clh_spin	29	0	27	13	73	20	13		73	13	7	13	40	47	0	67	7	27	60	60	67	67	73	73	73	53	67	73	42
clh_stp	21	13	13	7	7	13	13	13		13	7	7	7	0	7	20	7	7	0	13	0	0	33	7	13	13	7	7	10
clh-ls	21	0	27	7	73	7	0	0	73		7	7	40	47	0	67	7	27	60	53	60	60	73	73	60	47	67	67	38
hmcs	37	70	65	45	70	44	30	87	93	93		47	70	65	70	75	70	65	70	61	75	70	75	75	75	75	70	85	68
hticket-ls	29	73	40	13	73	20	13	67	93	73	7		60	60	40	73	47	33	67	67	67	67	73	73	73	73	67	87	57
malth_spin	26	30	20	10	70	11	10	40	93	47	5	7		50	10	60	15	15	60	44	60	60	70	60	75	55	55	65	42
malth_stp	32	30	20	15	64	22	20	40	87	47	25	27	20		20	59	25	32	36	33	45	45	65	65	65	55	50	60	41
mcs_spin	37	25	35	25	70	22	15	67	93	53	15	33	50	60		65	20	35	70	61	75	75	70	60	70	70	55	70	52
mcs_stp	21	25	20	15	18	17	15	27	47	27	15	13	20	18	15		20	23	14	17	18	18	30	10	30	25	10	25	20
mcs-ls	37	25	35	20	70	17	15	73	93	53	15	20	55	50	15	70		30	65	61	70	70	70	55	65	65	50	60	49
mcs-timepub	37	35	30	20	68	22	20	47	93	53	20	33	50	45	30	64	25		59	56	64	64	70	60	85	65	55	65	49
mutexee	26	25	15	20	68	22	15	27	93	33	25	27	30	23	20	73	30	27		22	32	41	70	65	65	45	45	55	38
partitioned	41	17	22	28	78	33	11	20	87	33	33	20	33	50	22	83	22	22	44		61	50	72	72	78	61	72	78	46
pthread	26	25	15	20	73	22	15	27	87	33	20	27	25	23	20	68	25	23	5	17		23	65	60	55	30	40	55	34
pthreadadapt	26	25	10	20	68	22	15	27	87	33	25	27	25	18	20	68	25	23	23	22	32		65	60	50	35	40	50	35
spinlock	21	25	15	20	20	17	15	27	67	27	20	20	25	20	20	45	25	15	10	17	10	20		0	30	25	0	15	21
spinlock-ls	21	25	20	20	35	17	15	27	87	27	20	20	35	25	35	80	25	15	15	17	20	20	65		30	25	5	20	28
ticket	26	20	5	15	55	11	15	13	87	27	20	13	15	15	10	60	20	5	15	6	25	30	60	60		0	20	35	25
ticket-ls	26	25	10	20	70	22	20	27	87	33	20	20	25	30	20	60	25	15	25	22	30	35	60	60	70		50	65	36
ttas	26	25	15	20	75	17	20	27	93	33	30	20	35	35	30	85	45	30	30	17	40	50	75	75	35	30		40	39
ttas-ls	26	20	15	10	65	22	20	27	87	33	15	13	30	20	20	70	40	10	30	17	30	35	70	65	30	25	5		31
average	30	31	27	22	63	22	17	42	85	46	19	24	40	40	27	67	31	31	43	39	48	49	67	60	59	48	46	58	30

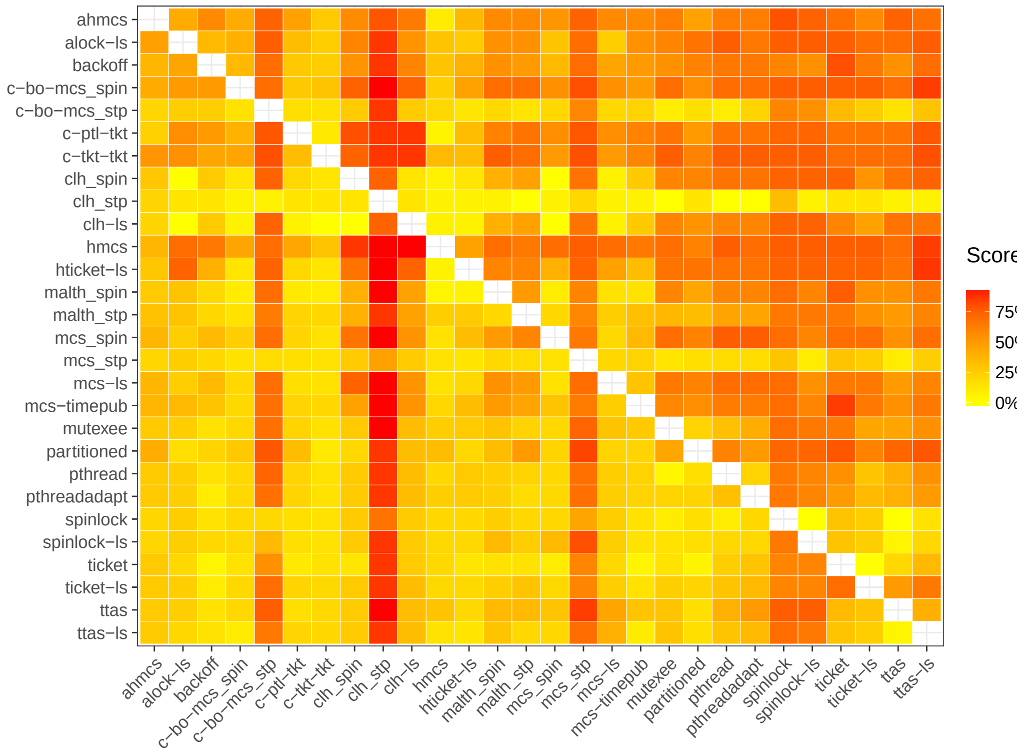

Fig. 16. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-48 machine in performance mode**).

Table 48. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-20 machine in performance mode**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		20	33	0	73	0	0	27	73	27	7	0	33	47	33	73	33	33	53	21	53	47	47	27	40	33	33	33	33
alock-ls	40		25	19	69	20	19	33	83	33	19	8	31	38	12	69	12	12	50	33	50	50	44	38	25	19	25	31	34
backoff	33	38		19	62	20	19	50	92	50	19	17	25	31	19	62	19	12	38	40	50	50	50	31	50	38	25	44	37
c-bo-mcs_spin	33	50	44		69	27	6	58	92	58	19	8	38	50	31	69	38	31	50	33	56	50	44	38	38	31	38	62	43
c-bo-mcs_stp	27	31	19	19		20	19	33	50	33	19	17	31	12	31	35	31	29	12	20	12	12	19	12	31	31	6	31	24
c-ptl-tkt	29	47	40	7	60		7	67	92	67	13	8	53	47	40	67	33	33	53	33	53	47	47	40	47	40	40	60	43
c-tkt-tkt	40	44	38	19	69	20		58	92	58	19	8	44	50	31	69	38	31	50	33	56	50	44	38	44	38	31	62	43
clh_spin	27	8	25	8	67	8	8		67	17	0	8	25	8	0	67	0	0	42	25	42	33	42	25	17	8	17	17	23
clh_stp	27	17	8	8	25	8	8	17		17	8	8	8	0	8	0	8	8	0	8	0	0	8	8	8	8	8	8	9
clh-ls	18	8	25	8	67	8	8	0	67		0	8	25	17	0	67	0	0	42	17	42	33	42	25	8	8	8	17	21
hmcs	40	38	38	12	69	13	6	50	92	58		0	31	50	25	69	25	31	50	27	56	50	44	38	38	31	38	56	40
hticket-ls	27	58	33	8	67	0	0	58	92	58	8		33	42	17	67	25	25	50	33	50	42	42	33	33	42	33	67	39
malth_spin	27	31	12	6	56	7	0	42	92	42	12	0		31	0	56	0	6	50	27	50	50	44	25	38	31	19	38	29
malth_stp	33	31	12	19	59	20	19	33	92	33	19	17	25		19	59	19	18	47	33	47	53	50	31	38	31	12	38	34
mcs_spin	33	31	38	19	56	13	6	42	92	50	12	8	38	38		56	19	12	50	33	56	50	44	25	38	25	19	44	35
mcs_stp	27	31	19	19	24	20	19	33	42	33	19	17	31	12	19		31	18	12	20	12	12	19	6	31	31	6	25	22
mcs-ls	33	31	31	19	56	13	6	42	92	50	12	8	31	31	0	56		6	56	33	56	50	44	25	38	31	19	44	34
mcs-timepub	33	38	44	19	53	20	19	42	92	50	19	17	31	24	19	53	31		47	40	53	47	50	31	44	38	19	50	38
mutexee	33	31	12	19	65	20	19	42	92	42	19	17	19	12	19	65	19	18		20	18	24	38	25	31	19	25	31	29
partitioned	50	20	27	27	67	27	27	33	92	25	13	17	27	20	20	67	20	13	40		47	47	47	40	20	20	20	40	34
pthread	33	31	12	19	65	20	19	42	92	42	19	17	19	6	19	65	19	18	18	20		24	38	31	25	19	12	25	28
pthreadadapt	27	31	12	19	65	20	19	33	92	33	19	17	19	6	19	65	19	18	29	20	29		31	19	19	19	12	25	27
spinlock	27	31	6	19	56	20	19	33	92	33	19	17	19	12	19	62	19	6	31	20	38	25		0	19	19	0	19	25
spinlock-ls	27	31	25	19	62	20	19	33	92	33	19	17	31	31	31	75	31	19	38	20	38	31	44		31	31	0	19	32
ticket	33	25	12	12	56	20	12	25	92	25	12	17	12	6	6	56	6	0	44	20	44	50	44	25		12	0	25	26
ticket-ls	33	31	19	12	56	13	6	42	92	42	19	8	19	25	6	56	12	0	50	33	50	50	31	25	25		19	50	31
ttas	33	31	31	19	62	20	19	33	92	33	19	17	38	31	31	75	31	19	44	27	44	50	44	25	38	31		31	36
ttas-ls	33	19	19	12	56	13	12	33	92	33	19	8	31	19	25	69	25	12	38	13	44	44	44	19	31	25	6		29
average	32	31	24	15	60	16	13	38	85	40	15	11	28	26	19	61	21	16	40	26	42	40	40	26	31	26	18	37	32

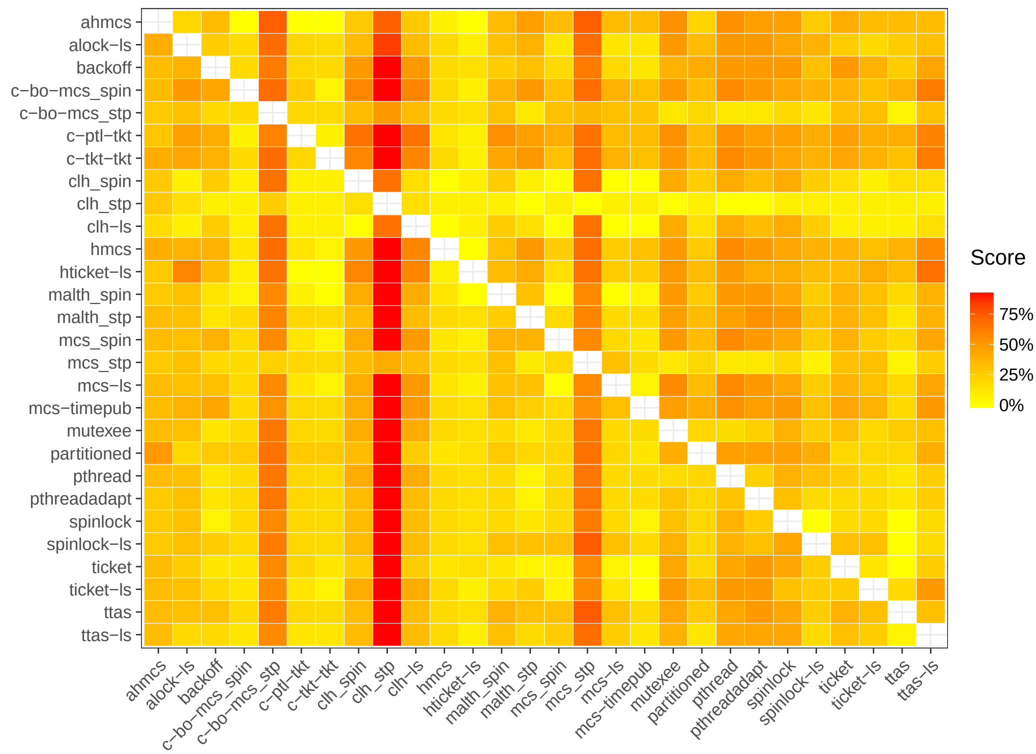

Fig. 17. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-20 machine in performance mode**).

Table 49. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**A-64 machine with thread-to-node pinning**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		27	64	50	73	30	32	29	82	35	9	35	45	59	32	82	41	50	77	30	82	73	82	77	82	50	77	77	55
alock-ls	41		46	42	62	27	33	22	94	22	17	17	42	50	17	75	25	50	75	32	75	71	75	71	75	50	71	75	50
backoff	32	33		29	58	32	29	28	89	33	33	22	38	42	33	79	42	33	58	45	58	67	79	71	75	46	54	75	49
c-bo-mcs_spin	32	50	58		67	32	33	50	94	61	21	22	46	42	42	71	46	33	62	45	71	75	79	75	71	54	67	88	55
c-bo-mcs_stp	23	29	12	8		14	17	22	72	22	12	6	25	15	21	62	25	15	46	27	46	38	83	67	62	33	58	67	34
c-ptl-tkt	25	32	55	50	77		9	33	89	33	18	17	41	55	23	77	36	55	73	45	73	64	73	68	82	41	68	86	52
c-tkt-tkt	27	29	54	50	75	18		22	89	28	25	17	46	58	25	79	42	50	75	45	75	67	79	75	83	46	71	88	53
clh_spin	24	17	56	33	72	28	28		78	28	11	17	17	44	0	78	22	44	78	33	78	72	78	78	78	50	78	78	48
clh_stp	18	6	6	6	22	11	11	11		11	6	6	6	0	6	0	6	0	6	11	6	6	72	33	11	6	11	6	11
clh-ls	29	6	44	33	67	17	22	11	83		11	22	17	44	6	78	11	39	78	33	78	72	78	78	78	33	78	78	45
hmcs	36	58	67	62	75	36	38	50	94	61		39	42	58	33	75	42	50	75	50	75	67	75	75	79	54	75	88	60
hticket-ls	18	33	67	33	83	11	22	39	94	44	6		22	44	11	78	22	33	78	33	78	67	78	78	83	50	78	94	51
malth_spin	23	38	42	29	58	14	12	39	94	39	21	22		42	12	71	21	33	62	36	62	58	71	67	75	38	62	79	45
malth_stp	27	42	33	21	65	23	21	39	83	44	25	28	21		21	50	29	27	50	41	50	54	67	67	62	50	54	71	43
mcs_spin	27	38	54	38	71	32	29	28	94	44	25	28	33	58		79	46	50	62	32	71	67	75	71	79	54	67	83	53
mcs_stp	18	25	4	12	23	14	12	22	17	22	17	11	12	12	8		17	12	4	18	8	8	71	33	29	12	12	25	18
mcs-ls	27	33	46	38	62	23	21	28	94	33	21	33	33	46	12	71		25	62	45	62	71	75	71	75	42	62	79	48
mcs-timepub	27	29	50	42	65	18	21	28	94	33	17	22	38	38	21	69	25		65	41	69	69	79	71	79	58	71	92	49
mutexee	23	25	21	17	35	18	17	22	89	22	21	11	29	31	29	77	29	23		18	27	31	83	71	46	29	62	75	36
partitioned	35	18	41	32	59	14	14	22	89	22	23	11	18	36	14	73	14	27	73		68	59	73	68	73	41	68	86	43
pthread	18	25	12	17	27	18	17	22	89	22	25	11	29	31	21	73	25	19	15	23		31	83	71	46	21	54	67	34
pthreadadapt	23	25	0	12	35	18	21	28	83	22	29	11	25	23	17	65	21	23	42	23	38		83	71	54	12	46	62	34
spinlock	18	25	0	12	4	18	12	22	28	22	25	11	17	17	17	17	17	12	4	14	0	0		4	17	8	0	21	13
spinlock-ls	23	29	4	17	12	23	17	22	56	22	25	11	21	21	21	42	21	17	8	23	8	8	79		38	8	21	38	23
ticket	18	21	12	8	21	9	8	22	83	22	21	6	17	29	12	58	8	8	33	14	29	21	75	54		0	54	71	27
ticket-ls	27	33	21	29	50	32	29	33	94	33	33	22	38	38	25	79	25	25	62	32	62	67	79	71	88		62	75	47
ttas	23	29	0	25	25	23	21	22	83	22	25	11	29	33	21	71	25	21	8	23	8	29	75	42	33	17		25	29
ttas-ls	23	21	4	12	21	14	12	22	83	22	8	6	21	21	17	67	17	4	4	14	17	25	75	46	29	17	42		25
average	25	29	32	28	51	21	21	27	82	31	20	18	28	37	19	66	26	29	50	31	51	49	77	64	62	34	56	68	25

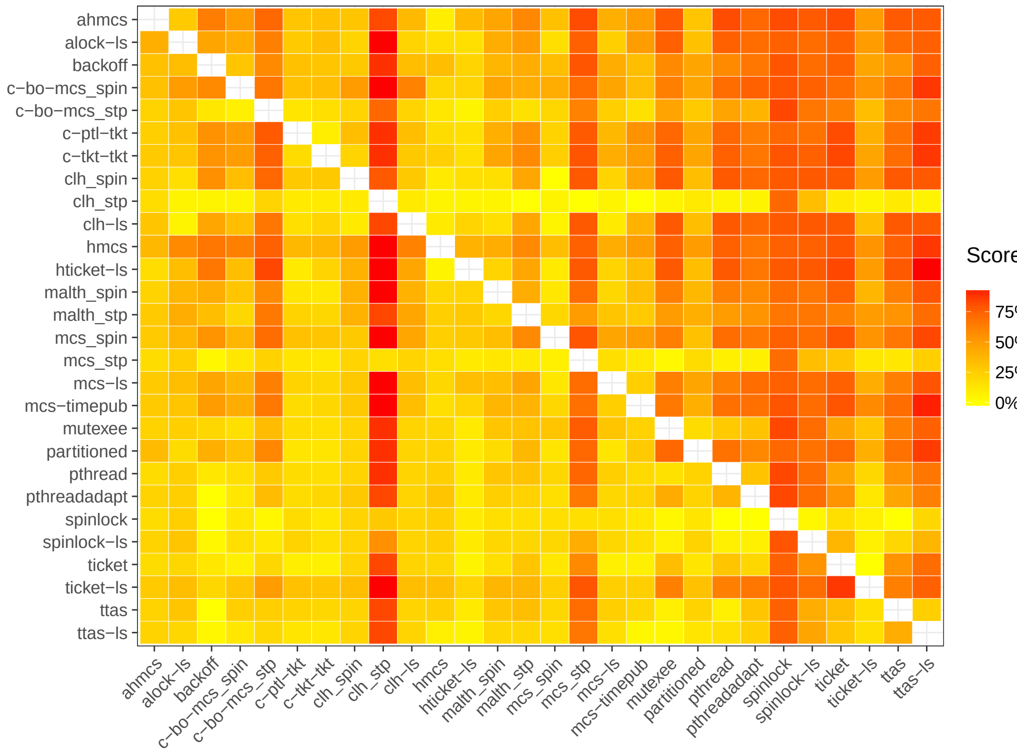

Fig. 18. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**A-64-node machine**).

Table 50. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-48 machine in energy-saving mode**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		42	58	42	74	47	26	57	79	64	11	36	58	58	53	74	58	58	63	47	63	63	79	74	68	58	74	68	57
alock-ls	47		35	40	75	33	25	60	87	53	30	27	55	55	30	70	25	55	60	67	75	65	75	75	75	70	70	75	56
backoff	37	45		35	70	28	25	53	87	60	30	40	55	50	35	70	45	50	55	61	65	65	60	55	80	65	55	70	54
c-bo-mcs_spin	42	50	50		70	28	30	73	93	73	25	47	70	70	55	80	55	50	70	56	70	70	75	75	75	75	70	85	62
c-bo-mcs_stp	21	25	25	15		17	15	27	87	27	20	13	20	14	20	59	20	23	9	17	9	23	60	55	35	25	15	30	27
c-ptl-tkt	24	56	50	39	78		11	80	87	87	6	33	61	67	56	78	56	61	67	50	67	67	72	72	67	67	67	78	59
c-tkt-tkt	53	55	45	45	80	33		73	87	87	35	33	75	70	50	80	50	60	75	61	75	75	75	75	70	70	70	80	64
clh_spin	29	0	27	13	73	20	13		73	13	7	13	40	47	0	67	7	27	60	60	67	67	73	73	73	53	67	73	42
clh_stp	21	13	13	7	7	13	13	13		13	7	7	7	0	7	20	7	7	0	13	0	0	33	7	13	13	7	7	10
clh-ls	21	0	27	7	73	7	0	0	73		7	7	40	47	0	67	7	27	60	53	60	60	73	73	60	47	67	67	38
hmcs	37	70	65	45	70	44	30	87	93	93		47	70	65	70	75	70	65	70	61	75	70	75	75	75	75	70	85	68
hticket-ls	29	73	40	13	73	20	13	67	93	73	7		60	60	40	73	47	33	67	67	67	67	73	73	73	73	67	87	57
malth_spin	26	30	20	10	70	11	10	40	93	47	5	7		50	10	60	15	15	60	44	60	60	70	60	75	55	55	65	42
malth_stp	32	30	20	15	64	22	20	40	87	47	25	27	20		20	59	25	32	36	33	45	45	65	65	65	55	50	60	41
mcs_spin	37	25	35	25	70	22	15	67	93	53	15	33	50	60		65	20	35	70	61	75	75	70	60	70	70	55	70	52
mcs_stp	21	25	20	15	18	17	15	27	47	27	15	13	20	18	15		20	23	14	17	18	18	30	10	30	25	10	25	20
mcs-ls	37	25	35	20	70	17	15	73	93	53	15	20	55	50	15	70		30	65	61	70	70	70	55	65	65	50	60	49
mcs-timepub	37	35	30	20	68	22	20	47	93	53	20	33	50	45	30	64	25		59	56	64	64	70	60	85	65	55	65	49
mutexee	26	25	15	20	68	22	15	27	93	33	25	27	30	23	20	73	30	27		22	32	41	70	65	65	45	45	55	38
partitioned	41	17	22	28	78	33	11	20	87	33	33	20	33	50	22	83	22	22	44		61	50	72	72	78	61	72	78	46
pthread	26	25	15	20	73	22	15	27	87	33	20	27	25	23	20	68	25	23	5	17		23	65	60	55	30	40	55	34
pthreadadapt	26	25	10	20	68	22	15	27	87	33	25	27	25	18	20	68	25	23	23	22	32		65	60	50	35	40	50	35
spinlock	21	25	15	20	20	17	15	27	67	27	20	20	25	20	20	45	25	15	10	17	10	20		0	30	25	0	15	21
spinlock-ls	21	25	20	20	35	17	15	27	87	27	20	20	35	25	35	80	25	15	15	17	20	20	65		30	25	5	20	28
ticket	26	20	5	15	55	11	15	13	87	27	20	13	15	15	10	60	20	5	15	6	25	30	60	60		0	20	35	25
ticket-ls	26	25	10	20	70	22	20	27	87	33	20	20	25	30	20	60	25	15	25	22	30	35	60	60	70		50	65	36
ttas	26	25	15	20	75	17	20	27	93	33	30	20	35	35	30	85	45	30	30	17	40	50	75	75	35	30		40	39
ttas-ls	26	20	15	10	65	22	20	27	87	33	15	13	30	20	20	70	40	10	30	17	30	35	70	65	30	25	5		31
average	30	31	27	22	63	22	17	42	85	46	19	24	40	40	27	67	31	31	43	39	48	49	67	60	59	48	46	58	30

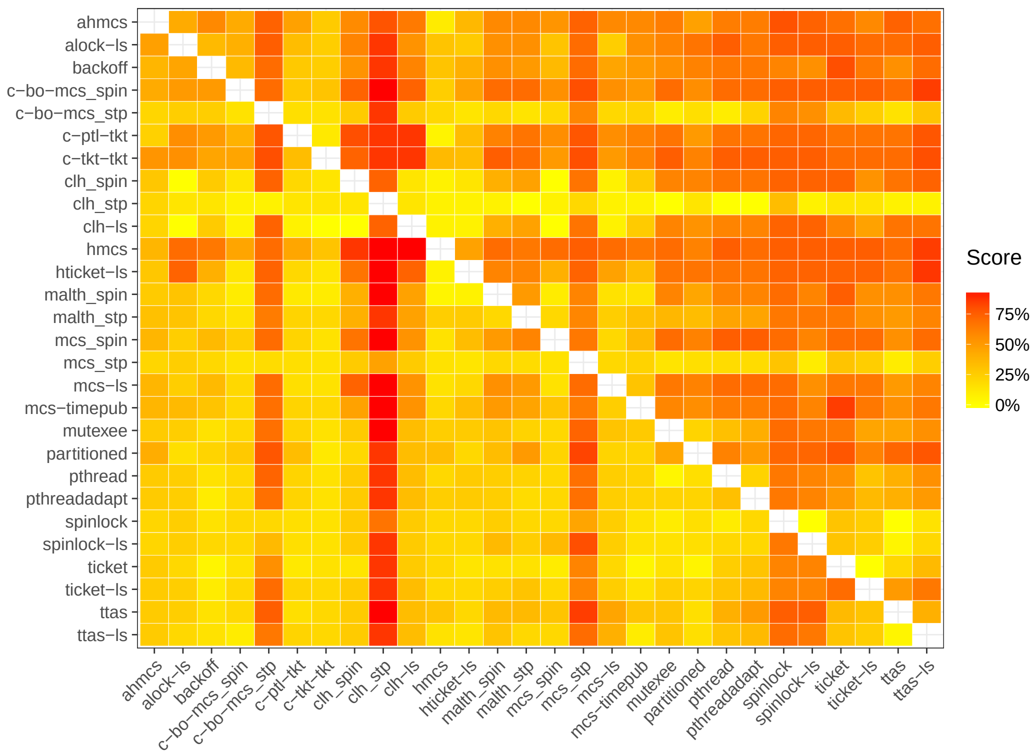

Fig. 19. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-48 machine in energy-saving move**).

Table 51. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-20 machine in energy-saving mode**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		20	33	0	73	0	0	27	73	27	7	0	33	47	33	73	33	33	53	21	53	47	47	27	40	33	33	33	33
alock-ls	40		25	19	69	20	19	33	83	33	19	8	31	38	12	69	12	12	50	33	50	50	44	38	25	19	25	31	34
backoff	33	38		19	62	20	19	50	92	50	19	17	25	31	19	62	19	12	38	40	50	50	50	31	50	38	25	44	37
c-bo-mcs_spin	33	50	44		69	27	6	58	92	58	19	8	38	50	31	69	38	31	50	33	56	50	44	38	38	31	38	62	43
c-bo-mcs_stp	27	31	19	19		20	19	33	50	33	19	17	31	12	31	35	31	29	12	20	12	12	19	12	31	31	6	31	24
c-ptl-tkt	29	47	40	7	60		7	67	92	67	13	8	53	47	40	67	33	33	53	33	53	47	47	40	47	40	40	60	43
c-tkt-tkt	40	44	38	19	69	20		58	92	58	19	8	44	50	31	69	38	31	50	33	56	50	44	38	44	38	31	62	43
clh_spin	27	8	25	8	67	8	8		67	17	0	8	25	8	0	67	0	0	42	25	42	33	42	25	17	8	17	17	23
clh_stp	27	17	8	8	25	8	8	17		17	8	8	8	0	8	0	8	8	0	8	0	0	8	8	8	8	8	8	9
clh-ls	18	8	25	8	67	8	8	0	67		0	8	25	17	0	67	0	0	42	17	42	33	42	25	8	8	8	17	21
hmcs	40	38	38	12	69	13	6	50	92	58		0	31	50	25	69	25	31	50	27	56	50	44	38	38	31	38	56	40
hticket-ls	27	58	33	8	67	0	0	58	92	58	8		33	42	17	67	25	25	50	33	50	42	42	33	33	42	33	67	39
malth_spin	27	31	12	6	56	7	0	42	92	42	12	0		31	0	56	0	6	50	27	50	50	44	25	38	31	19	38	29
malth_stp	33	31	12	19	59	20	19	33	92	33	19	17	25		19	59	19	18	47	33	47	53	50	31	38	31	12	38	34
mcs_spin	33	31	38	19	56	13	6	42	92	50	12	8	38	38		56	19	12	50	33	56	50	44	25	38	25	19	44	35
mcs_stp	27	31	19	19	24	20	19	33	42	33	19	17	31	12	19		31	18	12	20	12	12	19	6	31	31	6	25	22
mcs-ls	33	31	31	19	56	13	6	42	92	50	12	8	31	31	0	56		6	56	33	56	50	44	25	38	31	19	44	34
mcs-timepub	33	38	44	19	53	20	19	42	92	50	19	17	31	24	19	53	31		47	40	53	47	50	31	44	38	19	50	38
mutexee	33	31	12	19	65	20	19	42	92	42	19	17	19	12	19	65	19	18		20	18	24	38	25	31	19	25	31	29
partitioned	50	20	27	27	67	27	27	33	92	25	13	17	27	20	20	67	20	13	40		47	47	47	40	20	20	20	40	34
pthread	33	31	12	19	65	20	19	42	92	42	19	17	19	6	19	65	19	18	18	20		24	38	31	25	19	12	25	28
pthreadadapt	27	31	12	19	65	20	19	33	92	33	19	17	19	6	19	65	19	18	29	20	29		31	19	19	19	12	25	27
spinlock	27	31	6	19	56	20	19	33	92	33	19	17	19	12	19	62	19	6	31	20	38	25		0	19	19	0	19	25
spinlock-ls	27	31	25	19	62	20	19	33	92	33	19	17	31	31	31	75	31	19	38	20	38	31	44		31	31	0	19	32
ticket	33	25	12	12	56	20	12	25	92	25	12	17	12	6	6	56	6	0	44	20	44	50	44	25		12	0	25	26
ticket-ls	33	31	19	12	56	13	6	42	92	42	19	8	19	25	6	56	12	0	50	33	50	50	31	25	25		19	50	31
ttas	33	31	31	19	62	20	19	33	92	33	19	17	38	31	31	75	31	19	44	27	44	50	44	25	38	31		31	36
ttas-ls	33	19	19	12	56	13	12	33	92	33	19	8	31	19	25	69	25	12	38	13	44	44	44	19	31	25	6		29
average	32	31	24	15	60	16	13	38	85	40	15	11	28	26	19	61	21	16	40	26	42	40	40	26	31	26	18	37	32

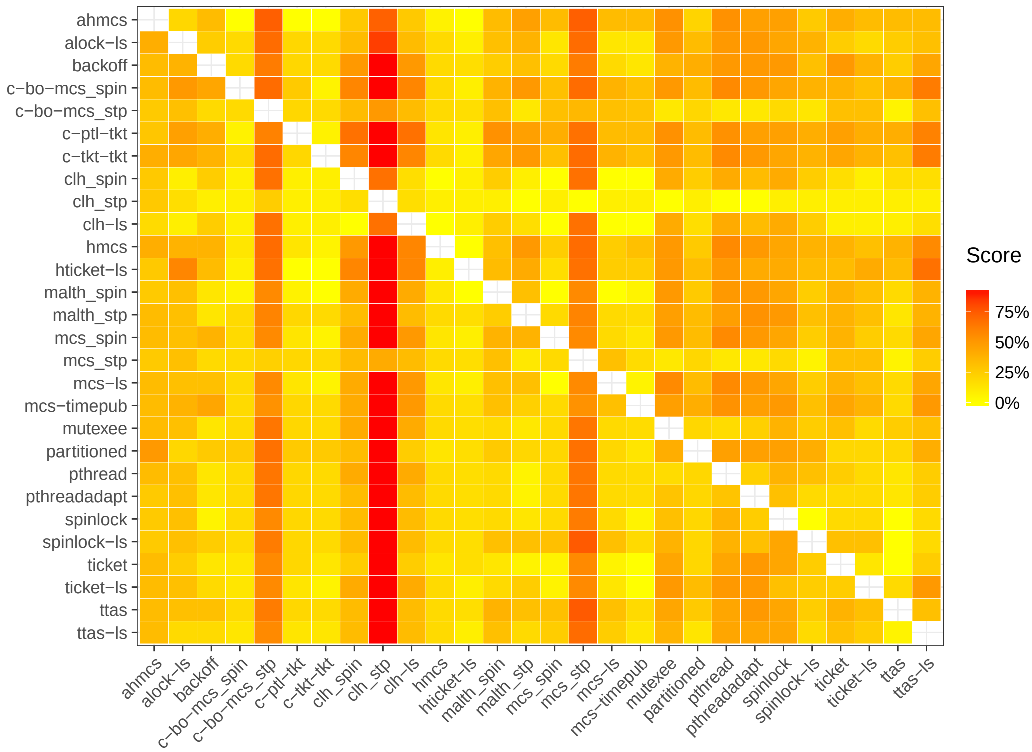

Fig. 20. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A performs at least 5% better than B (**I-20 machine in energy-saving move**).

A.6 Are all locks potentially harmful?

Table 52. For each lock-sensitive application, at *max nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the performance gain is greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. For example, the table shows that for the dedup application, the best lock (0%, here mutexee) is 609% better than the alock-ls lock. The gray cells highlight configurations where a given lock hurts the application, i.e., the performance gain brought by the best lock with respect to the given lock is greater than 15%. Thus, for each lock in a column, the number of gray cells corresponds to the number of applications for which the lock is defeated by a gap of 15% or more by the best lock(s) for this application. (**A-64 machine**).

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

Table 53. For each lock-sensitive application, at *opt nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the performance gain is greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**A-64 machine**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	462	29	255	252	64	45	538	531	819	296	233	197	197	206	203	207	222	0	36	36	28	17	29	22	28	24	544
facesim	2	3	6	4	7	1	0	3	12	3	3	1	2	7	1	6	2	3	4	3	7	13	6	9	5	2	3	7
ferret	88	49	6	32	0	89	100	54	0	51	82	91	84	0	51	0	56	3	0	55	0	0	7	3	54	40	5	6
fluidanimate	-	126	0	46	46	31	13	-	-	-	59	-	24	23	21	22	34	36	69	10	2	8	5	0	0	0	0	76
fmm	41	36	22	16	27	29	15	27	25	38	33	31	21	18	2	0	32	0	25	20	25	23	1	27	19	28	22	32
kyotocabinet	13	37	0	12	9	20	25	29	414	40	19	21	22	31	23	347	35	34	77	22	82	59	65	88	47	26	40	48
linear_regression	2	12	16	9	19	9	15	1	30	7	0	6	4	4	11	29	10	11	18	4	16	4	48	29	22	13	26	24
matrix_multiply	9	78	3	14	5	24	23	9	3	12	83	59	5	3	23	2	168	0	6	24	3	3	5	59	3	59	4	55
memcached-new	0	18	53	69	119	-	12	-	-	-	5	-	29	53	7	48	20	10	66	-	90	55	36	34	36	34	34	35
memcached-old	14	72	2	30	29	-	43	-	-	-	54	-	19	23	21	19	32	43	44	-	50	50	15	5	13	14	0	9
mysqld	-	-	-	-	55	-	-	-	-	-	-	-	-	0	-	8	-	121	8	-	96	96	-	-	-	-	-	-
ocean_cp	7	8	6	11	13	7	9	3	2	1	9	8	8	20	10	19	0	14	18	7	6	14	11	11	8	7	8	10
ocean_ncp	4	2	8	12	2	1	0	4	13	4	1	1	3	4	12	11	4	9	3	3	6	11	13	2	4	2	6	4
pca	4	3	10	11	22	6	8	6	38	3	3	3	6	4	7	39	3	7	11	3	16	7	6	3	4	0	3	2
pca_ll	4	3	12	19	54	3	0	5	160	3	2	0	2	18	0	157	2	2	32	0	43	42	41	20	42	1	2	3
radiosity	9	9	4	4	9	6	5	8	0	10	6	8	6	9	0	13	8	0	10	0	0	0	1	11	0	8	10	10
radiosity_ll	0	25	36	26	79	5	25	19	175	32	0	17	55	67	15	161	32	34	69	26	134	113	131	85	90	38	56	62
s_raytrace	0	6	65	23	87	6	11	5	123	10	5	17	26	53	0	117	14	12	232	10	75	94	120	103	119	30	123	124
s_raytrace_ll	10	20	42	39	185	11	13	0	401	5	41	19	37	21	0	390	10	14	244	56	200	67	394	388	396	172	388	390
sqlite	-	-	-	-	3	-	-	-	-	-	-	-	-	5	-	16	-	0	68	-	80	60	-	-	-	-	-	-
ssl_proxy	3	3	16	16	25	0	2	7	31	5	0	0	27	32	10	23	9	11	38	7	58	27	20	28	19	11	15	18
streamcluster	9	7	6	2	5	6	1	-	-	-	5	-	8	8	0	0	7	3	14	5	10	5	0	4	6	6	5	6
streamcluster_ll	12	12	19	5	12	0	16	-	-	-	11	-	33	27	23	20	26	22	35	10	18	34	26	3	15	19	12	12
upscaledb	0	11	30	4	8	4	3	8	196	12	0	2	1	10	11	194	15	19	117	15	108	58	32	33	27	21	33	30
vips	3	5	2	2	2	3	2	-	-	-	5	-	1	2	2	2	5	2	2	3	0	1	3	1	1	1	2	4
volrend	2	4	10	5	11	2	2	4	8	3	0	1	5	8	3	7	4	4	15	3	17	18	23	14	8	4	10	11
water_nsquared	94	48	4	6	10	8	2	35	35	58	14	11	7	6	3	2	9	7	7	4	6	7	0	6	4	6	5	37
water_spatial	95	48	1	9	6	3	3	40	39	63	8	4	8	5	9	9	5	10	3	1	0	0	2	1	1	0	0	40

Table 54. For each lock-sensitive application, at *max nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the performance gain is greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**A-48 machine**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	-	10	226	223	67	32	-	-	-	243	208	75	71	275	269	204	285	0	544	23	18	35	21	511	25	25	-
ferret	453	389	10	391	0	480	455	402	0	383	438	466	472	0	387	2	395	7	0	397	0	0	14	10	395	271	10	10
fmm	50	45	40	41	36	39	39	53	40	48	40	41	39	34	5	0	39	3	36	38	33	33	3	36	41	39	41	40
kyotocabinet	8	33	21	0	470	21	16	31	327	37	12	14	27	191	26	328	30	43	327	42	364	259	1k	1k	176	83	474	234
linear_regression	8	9	0	5	242	9	9	4	59	7	6	4	9	73	8	102	8	17	20	12	20	0	74	26	26	4	25	16
memcached-new	2	0	19	40	550	-	5	-	-	-	1	-	116	302	26	350	5	6	51	-	88	143	620	390	152	49	170	154
memcached-old	65	103	0	24	298	-	79	-	-	-	69	-	34	175	84	343	84	59	121	-	126	87	1k	646	202	137	174	332
mysqld	-	-	-	-	33	-	-	-	-	-	-	-	-	13	-	9	-	54	0	-	1	0	-	-	-	-	-	-
pca	13	20	0	1	199	7	16	19	160	20	9	0	3	104	27	256	16	2	71	14	75	23	213	87	71	14	96	86
pca_ll	15	40	0	8	569	20	30	23	240	31	36	25	4	282	32	242	24	12	117	19	136	48	298	148	83	19	113	146
radiosity	10	10	13	16	108	6	10	14	684	8	11	14	10	64	0	622	10	0	72	12	78	33	138	67	51	25	60	68
radiosity_ll	0	7	33	11	871	18	24	24	1k	25	0	29	62	560	14	1k	30	39	611	66	662	223	2k	689	484	171	604	584
s_raytrace	27	28	8	46	1k	27	34	1	767	19	31	19	62	544	15	1k	36	0	200	32	169	89	388	235	340	101	301	354
s_raytrace_ll	5	0	22	48	771	24	35	7	1k	7	4	44	64	414	12	1k	14	24	213	30	143	76	660	344	392	124	407	435
sqlite	-	-	-	-	92	-	-	-	-	-	-	-	-	11	-	161	-	939	58	-	60	0	-	-	-	-	-	-
ssl_proxy	94	104	0	100	1k	127	117	191	2k	101	101	104	175	2k	193	3k	110	165	160	216	660	330	2k	752	173	317	725	938
streamcluster	47	43	104	106	190	15	29	-	-	-	0	-	214	1k	154	2k	195	129	94	61	153	185	285	167	211	163	111	81
streamcluster_ll	56	64	180	21	214	0	38	-	-	-	10	-	226	1k	256	1k	235	132	97	63	169	287	339	155	251	192	118	96
upscaledb	0	7	21	1	150	5	5	10	222	10	1	1	0	34	12	221	14	16	80	17	172	105	500	255	89	42	120	249
vips	140	112	3	338	33	240	191	-	-	-	214	-	781	6	70	7	66	4	3	82	0	0	3	3	67	23	5	12
volrend	9	16	9	10	76	3	4	10	87	11	0	0	2	126	19	101	13	24	55	10	70	74	91	42	17	9	17	20
water_nsquared	78	42	8	15	14	11	11	29	29	48	13	14	8	9	5	6	16	8	10	6	8	8	0	10	9	12	12	32
water_spatial	69	35	2	4	4	1	0	34	33	45	6	4	9	8	19	21	4	19	2	5	0	1	15	1	3	1	2	28

Table 55. For each lock-sensitive application, at *opt nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the performance gain is greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**A-48 machine**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	-	19	163	162	49	33	-	-	-	189	144	51	53	196	191	136	199	0	16	20	19	15	17	7	14	20	-
ferret	122	69	6	41	0	130	131	69	0	69	123	129	90	0	69	1	62	4	0	65	0	0	9	3	70	51	5	6
fmm	50	45	40	41	36	39	39	47	40	48	40	41	38	34	5	0	39	3	36	38	33	33	3	36	41	39	41	40
kyotocabinet	25	34	0	10	69	28	24	33	426	38	24	30	25	141	25	385	36	37	65	30	79	43	53	46	43	27	18	42
linear_regression	85	92	93	87	136	87	91	78	161	89	86	81	78	133	100	242	69	127	0	91	105	93	151	117	114	90	120	99
memcached-new	3	0	16	33	78	-	1	-	-	-	2	-	12	74	0	70	0	4	50	-	65	30	16	18	26	1	0	1
memcached-old	2	14	1	10	22	-	2	-	-	-	3	-	6	64	0	254	5	10	80	-	82	26	2	2	1	0	5	2
mysqld	-	-	-	-	33	-	-	-	-	-	-	-	-	13	-	9	-	52	0	-	0	0	-	-	-	-	-	-
pca	2	3	3	2	23	2	1	4	64	2	11	0	5	63	19	74	7	17	20	3	29	9	32	12	11	1	11	10
pca_ll	2	3	1	5	56	0	0	2	177	1	0	0	4	59	2	176	3	2	15	1	20	20	3	4	4	4	3	3
radiosity	30	26	29	25	33	22	27	23	44	27	26	27	25	43	0	29	24	4	33	22	34	34	15	29	28	30	26	31
radiosity_ll	0	4	25	11	70	15	24	14	234	21	0	29	53	208	4	218	9	30	78	25	111	72	72	66	78	61	59	64
s_raytrace	5	2	9	16	130	5	10	1	294	3	4	9	21	186	0	340	5	0	55	4	59	55	50	53	51	32	54	50
s_raytrace_ll	5	0	22	48	323	24	35	7	686	7	4	44	64	414	12	672	14	24	192	30	143	76	336	333	339	110	327	282
sqlite	-	-	-	-	31	-	-	-	-	-	-	-	-	175	-	511	-	0	55	-	68	35	-	-	-	-	-	-
ssl_proxy	69	64	4	92	74	79	86	89	873	64	70	77	124	2k	98	2k	74	109	0	104	163	141	130	139	134	72	93	138
streamcluster	6	8	6	16	20	14	17	-	-	-	0	-	8	8	3	6	1	8	20	6	907	954	7	0	10	6	4	3
streamcluster_ll	29	35	32	15	14	6	12	-	-	-	0	-	34	39	30	28	32	33	44	19	30	43	28	19	36	33	30	29
upscaledb	1	8	21	2	164	7	7	10	208	10	0	0	2	43	11	213	13	17	92	16	84	69	37	38	33	21	40	36
vips	15	18	3	15	17	16	17	-	-	-	17	-	14	6	16	7	15	4	3	15	0	0	3	3	15	17	5	12
volrend	0	2	6	1	23	0	3	6	23	0	1	1	6	27	6	28	1	10	11	5	12	12	17	9	8	3	6	7
water_nsquared	78	42	8	15	14	11	11	29	29	48	13	14	8	9	5	6	16	8	10	6	8	8	0	10	9	12	12	32
water_spatial	69	35	2	4	4	1	0	34	33	45	6	4	9	8	19	21	4	19	2	5	0	1	15	1	3	1	2	28

Table 56. For each lock-sensitive application, at *max nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the performance gain is greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**I-48 machine in performance mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	716	2	155	67	2k	2k	2k	655	2k	128	252	469	70	227	62	448	80	2	2k	11	20	10	5	2k	1k	0	411
ferret	640	560	16	614	1	609	589	556	1	558	629	660	662	0	552	0	537	11	0	557	1	0	14	14	527	401	14	14
kyotocabinet	14	33	39	0	161	21	18	37	487	34	17	19	58	78	30	423	28	50	121	53	125	91	578	322	66	41	64	72
linear_regression	11	6	6	12	74	21	1	3	83	28	0	21	8	7	4	79	47	11	10	8	12	10	68	53	17	13	18	13
memcached-new	90	24	2	25	87	-	42	-	-	-	64	-	13	45	36	296	41	31	50	-	72	56	126	89	34	0	30	33
memcached-old	14	0	93	26	29	-	8	-	-	-	38	-	186	171	91	93	170	166	150	-	195	189	96	80	183	183	99	94
mysqld	-	-	-	-	0	-	-	-	-	-	-	-	-	6	-	9	-	56	3	-	4	8	-	-	-	-	-	-
pca	8	17	3	0	108	3	10	15	315	14	3	11	6	5	9	286	12	14	29	27	36	29	243	190	51	36	42	48
pca_ll	3	10	52	9	280	0	1	16	925	14	1	1	19	72	9	980	9	23	85	30	103	190	691	550	80	46	146	132
radiosity	18	13	7	2	48	4	3	19	173	22	0	2	8	11	5	147	5	7	17	16	22	17	142	106	28	20	30	38
radiosity_ll	1	65	106	8	779	54	22	92	2k	91	0	25	129	149	58	2k	71	90	211	174	267	159	2k	1k	379	247	436	440
s_raytrace	0	13	9	28	453	0	7	23	1k	24	0	20	42	96	8	1k	11	12	58	66	62	101	787	689	161	101	177	243
s_raytrace_ll	2	14	45	29	151	13	15	31	1k	30	0	35	104	135	11	1k	13	26	63	58	68	74	845	817	176	98	213	245
sqlite	-	-	-	-	196	-	-	-	-	-	-	-	-	0	-	505	-	4k	37	-	35	15	-	-	-	-	-	-
ssl_proxy	7	47	22	0	473	45	26	55	1k	51	6	19	61	109	57	1k	45	62	129	98	134	94	2k	730	189	125	189	181
streamcluster	47	0	486	162	424	81	13	-	-	-	36	-	356	418	298	342	288	274	432	18	386	485	393	274	488	457	225	217
streamcluster_ll	49	0	466	177	433	121	2	-	-	-	52	-	359	467	346	379	319	303	444	13	456	569	424	308	522	502	268	298
upscaledb	3	23	27	0	63	4	4	26	281	26	4	0	26	21	27	303	22	36	138	36	138	106	195	187	58	39	57	58
vips	213	127	2	706	48	410	324	-	-	-	347	-	408	1	139	6	135	2	0	167	2	0	3	3	100	5	0	1
volrend	9	10	150	2	42	1	0	14	26	10	0	1	2	21	10	11	2	3	15	1	25	25	40	30	15	10	17	21
water_nsquared	128	52	0	0	4	2	4	97	89	90	5	2	2	0	3	6	2	6	2	0	1	5	2	3	2	5	0	44
water_spatial	917	320	2	11	9	2	1	614	618	620	13	8	9	9	9	9	9	8	0	1	2	0	0	0	0	1	0	321

Table 57. For each lock-sensitive application, at *opt nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the performance gain is greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**I-48 machine in performance mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	451	5	86	68	18	5	726	715	728	93	65	60	59	59	59	59	94	0	7	0	0	1	7	5	2	1	381
ferret	46	45	8	48	1	44	50	49	0	46	45	81	63	0	45	0	64	7	0	55	1	0	8	9	45	44	7	8
kyotocabinet	10	25	6	0	14	12	9	22	384	24	13	15	31	58	23	413	20	35	94	33	106	71	128	90	42	30	29	32
linear_regression	10	6	6	11	45	9	1	3	79	21	0	6	8	7	4	79	10	11	10	8	12	10	68	45	17	13	16	13
memcached-new	53	24	2	0	37	-	0	-	-	-	62	-	9	13	12	68	37	17	9	-	18	13	5	9	14	0	10	11
memcached-old	0	13	13	9	11	-	23	-	-	-	15	-	72	70	32	31	59	60	62	-	71	78	35	29	85	74	15	42
mysqld	-	-	-	-	0	-	-	-	-	-	-	-	-	5	-	7	-	56	3	-	4	4	-	-	-	-	-	-
pca	8	7	10	0	31	2	4	10	277	10	6	8	1	11	10	308	9	11	26	8	32	21	12	14	17	10	7	12
pca_ll	3	10	52	9	195	0	1	16	403	14	1	1	18	45	9	186	9	19	75	20	89	97	182	69	40	37	39	39
radiosity	16	9	6	0	7	0	0	17	71	17	0	1	5	9	4	25	4	6	12	7	16	11	22	15	16	12	13	20
radiosity_ll	1	61	102	8	52	18	21	87	1k	88	0	24	129	149	57	2k	59	80	163	99	249	159	263	218	248	169	162	163
s_raytrace	0	7	9	24	90	0	7	13	179	18	0	13	38	88	5	211	6	12	57	31	54	80	172	172	94	60	62	84
s_raytrace_ll	2	14	44	29	151	8	15	31	342	30	0	35	104	135	11	343	13	26	63	58	68	74	343	345	176	98	213	245
sqlite	-	-	-	-	0	-	-	-	-	-	-	-	-	41	-	618	-	35	61	-	74	55	-	-	-	-	-	-
ssl_proxy	0	42	23	6	54	27	9	61	1k	53	0	12	80	153	48	1k	37	58	124	57	144	130	445	209	178	125	135	121
streamcluster	8	3	25	5	6	3	1	-	-	-	0	-	41	38	19	15	36	14	32	7	15	35	19	17	41	42	17	14
streamcluster_ll	21	0	125	43	41	16	15	-	-	-	15	-	151	158	91	99	103	109	145	21	78	160	122	75	161	161	84	75
upscaledb	1	23	27	0	63	2	1	26	264	26	1	0	17	21	26	274	22	36	124	35	124	106	68	67	48	37	46	47
vips	24	24	2	22	22	22	22	-	-	-	24	-	23	1	23	6	22	2	0	22	2	0	3	3	23	5	0	1
volrend	17	7	41	3	23	0	0	12	24	11	1	0	5	15	3	13	2	9	17	7	20	27	26	20	18	9	15	18
water_nsquared	128	52	0	0	4	2	4	97	89	90	5	2	2	0	3	6	2	6	2	0	1	5	2	3	2	5	0	44
water_spatial	917	320	2	11	9	2	1	614	618	620	13	8	9	9	9	9	9	8	0	1	2	0	0	0	0	1	0	321

Table 58. For each lock-sensitive application, at *max nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the performance gain is greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**I-20 machine in performance mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	1k	25	261	56	178	165	695	501	758	428	202	316	56	233	54	305	83	1	132	0	1	22	24	110	222	25	307
ferret	363	343	46	369	2	367	366	337	0	342	354	367	407	0	338	0	338	29	0	339	0	0	66	67	340	296	68	69
kyotocabinet	4	12	6	0	251	8	3	14	253	17	10	6	27	19	10	249	10	16	70	21	61	38	112	71	22	12	21	20
memcached-old	2	3	3	2	315	-	2	-	-	-	3	-	1	20	1	254	2	3	81	-	97	36	5	2	0	1	2	4
pca	1	5	10	4	184	0	4	6	167	5	3	3	4	11	5	163	3	2	14	3	15	31	36	5	7	2	3	10
pca_ll	2	3	7	1	473	2	0	0	423	2	1	0	3	3	1	437	1	1	9	0	16	3	1	0	0	0	0	1
radiosity	23	11	5	5	52	1	3	19	77	17	2	1	6	13	3	60	0	4	7	12	11	21	50	22	14	7	14	16
radiosity_ll	2	9	33	0	577	2	0	16	799	22	1	4	29	46	8	802	10	15	56	59	76	156	427	167	89	40	78	59
s_raytrace	0	0	1	2	337	0	0	0	312	1	1	1	1	3	0	224	0	0	2	0	3	3	2	0	1	0	0	0
s_raytrace_ll	2	2	4	3	582	1	1	2	641	2	2	2	2	1	1	643	1	2	3	1	5	1	0	1	1	1	1	1
sqlite	-	-	-	-	321	-	-	-	-	-	-	-	-	0	-	388	-	9k	22	-	27	58	-	-	-	-	-	-
streamcluster	23	0	61	17	33	24	12	-	-	-	9	-	55	55	49	45	61	51	387	6	168	129	85	27	82	80	31	29
streamcluster_ll	20	0	67	17	29	26	9	-	-	-	9	-	59	60	50	50	67	57	466	0	233	158	91	39	80	90	34	36
upscaledb	3	11	21	1	143	0	7	14	152	13	1	3	5	13	10	150	10	14	129	21	125	123	56	35	13	11	12	20
vips	115	62	3	50	0	62	21	-	-	-	85	-	39	0	25	0	29	1	1	80	2	1	3	2	49	52	2	1
water_nsquared	47	17	0	0	1	0	0	32	31	33	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16
water_spatial	325	110	0	3	3	0	0	219	212	211	4	3	2	3	3	2	3	2	0	0	0	0	0	0	0	1	0	107

Table 59. For each lock-sensitive application, at *opt nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the performance gain is greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**I-20 machine in performance mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	437	5	82	81	15	9	722	711	703	105	66	80	74	78	71	76	77	3	8	2	0	4	6	8	6	6	378
ferret	101	92	46	103	2	102	103	92	0	91	102	109	136	0	90	0	89	29	0	89	0	0	54	54	87	90	54	53
kyotocabinet	3	13	3	0	249	6	3	14	255	16	8	6	22	21	12	250	10	18	74	21	58	32	53	33	19	12	17	18
memcached-old	2	3	3	2	315	-	2	-	-	-	3	-	1	20	1	254	2	3	81	-	97	36	5	2	0	1	2	4
pca	1	5	10	4	23	0	4	6	19	5	3	3	4	11	5	20	3	2	14	3	15	14	15	5	7	2	3	10
pca_ll	2	3	7	1	87	2	0	0	88	2	1	0	3	3	1	86	1	1	9	0	16	3	1	0	0	0	0	1
radiosity	23	11	5	5	5	1	2	19	20	17	2	1	6	6	3	5	0	4	4	6	2	5	6	7	7	5	5	13
radiosity_ll	2	9	33	0	61	2	0	16	65	22	1	4	29	46	8	54	10	15	56	56	69	63	57	54	56	40	53	53
s_raytrace	0	0	1	2	89	0	0	0	91	1	1	1	1	3	0	88	0	0	2	0	3	3	2	0	1	0	0	0
s_raytrace_ll	2	2	4	3	76	1	1	2	77	2	2	2	2	1	1	77	1	2	3	1	5	1	0	1	1	1	1	1
sqlite	-	-	-	-	25	-	-	-	-	-	-	-	-	0	-	394	-	16	47	-	49	42	-	-	-	-	-	-
streamcluster	8	0	28	19	27	12	18	-	-	-	13	-	24	16	18	17	25	10	28	15	3	23	25	8	36	34	19	13
streamcluster_ll	20	0	67	17	29	26	9	-	-	-	9	-	59	60	50	50	67	57	112	0	67	100	91	39	80	90	34	36
upscaledb	3	11	21	1	143	0	7	14	145	13	1	3	5	13	10	147	10	14	129	21	125	123	34	29	13	11	12	20
vips	93	62	3	50	0	62	21	-	-	-	85	-	39	0	25	0	29	1	1	80	2	1	3	2	49	52	2	1
water_nsquared	47	17	0	0	1	0	0	32	31	33	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16
water_spatial	325	110	0	3	3	0	0	219	212	211	4	3	2	3	3	2	3	2	0	0	0	0	0	0	0	1	0	107

Table 60. For each lock-sensitive application, at *max nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the performance gain is greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**A-64 machine with thread-to-node pinning**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	652	14	159	160	37	24	1k	720	1k	78	141	59	59	61	59	71	80	0	215	11	12	13	13	187	13	14	635
facesim	3	5	39	34	131	5	4	6	56	6	4	3	0	47	5	55	4	6	29	5	30	56	297	131	29	2	38	51
ferret	366	310	3	215	0	356	370	326	0	328	385	368	338	0	322	0	277	1	0	320	0	0	7	2	287	173	3	4
fluidanimate	-	301	0	49	53	27	12	-	-	-	198	-	138	158	133	156	146	156	101	69	57	61	71	3	57	3	0	202
kyotocabinet	9	42	26	0	60	24	27	31	493	42	12	18	21	27	27	468	38	52	338	45	400	277	2k	2k	207	72	577	344
linear_regression	6	10	14	5	15	6	6	3	71	11	0	2	3	7	1	68	4	5	38	5	33	16	100	50	33	11	56	53
memcached-new	1	9	33	46	175	-	6	-	-	-	15	-	16	38	0	217	23	10	45	-	94	171	909	701	182	73	435	79
memcached-old	123	61	56	6	2	-	14	-	-	-	0	-	290	300	155	181	284	148	108	-	206	235	308	40	224	234	51	79
mysqld	-	-	-	-	66	-	-	-	-	-	-	-	-	0	-	4	-	60	1	-	1	2	-	-	-	-	-	-
ocean_cp	2	0	34	44	78	0	0	3	52	3	3	5	2	44	0	48	3	13	27	0	25	38	130	76	13	1	29	30
ocean_ncp	4	0	23	12	67	2	2	3	39	2	4	4	3	35	1	37	1	9	23	2	20	28	109	59	11	2	24	28
pca	48	40	38	17	82	47	47	43	285	41	46	46	43	0	46	285	39	40	123	47	132	50	347	138	91	19	234	160
pca_ll	62	48	0	8	155	56	61	67	380	38	64	53	31	4	57	380	24	23	146	52	157	52	551	372	108	19	273	184
radiosity	6	5	8	3	7	1	1	4	71	6	0	1	2	3	1	70	2	3	32	3	32	19	114	49	29	11	49	52
radiosity_ll	0	41	44	25	93	37	47	27	1k	47	0	21	67	65	26	1k	57	64	531	77	521	280	2k	1k	585	188	1k	792
s_raytrace	2	9	53	35	225	16	18	9	1k	17	0	30	30	68	0	1k	14	17	332	31	264	144	778	340	398	104	764	564
s_raytrace_ll	6	3	44	28	38	10	12	0	1k	6	7	19	45	23	0	1k	8	16	304	54	187	66	1k	746	561	187	1k	914
sqlite	-	-	-	-	31	-	-	-	-	-	-	-	-	0	-	676	-	809	367	-	342	179	-	-	-	-	-	-
ssl_proxy	1	11	72	5	143	11	14	13	933	13	0	7	14	32	14	945	26	27	317	28	299	155	1k	407	259	80	600	545
streamcluster	44	19	145	87	119	11	11	-	-	-	0	-	241	706	159	256	195	65	81	30	132	255	523	253	272	181	158	111
streamcluster_ll	32	11	140	69	146	4	18	-	-	-	0	-	230	826	101	247	171	63	85	20	160	204	491	250	309	180	161	159
upscaledb	6	17	29	8	190	808	415	14	285	18	5	12	0	4	17	285	21	34	98	508	112	107	687	463	2k	49	193	128
vips	71	33	8	238	0	136	83	-	-	-	95	-	223	2	39	3	25	5	5	35	4	3	4	10	30	13	8	13
volrend	7	4	31	17	41	0	0	6	24	8	1	2	10	19	6	19	6	11	41	7	37	49	105	66	28	12	34	37
water_nsquared	89	42	0	5	4	1	1	55	55	53	6	3	3	3	3	3	3	8	1	0	0	0	1	0	1	0	0	31
water_spatial	297	42	0	4	3	2	1	217	218	57	6	2	4	4	4	4	4	5	2	1	1	0	1	0	1	0	0	35

Table 61. For each lock-sensitive application, at *opt nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the performance gain is greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**A-64 machine with thread-to-node pinning**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	402	36	237	221	70	53	716	689	727	183	198	170	96	181	96	201	118	0	263	6	7	18	34	218	37	36	464
facesim	0	1	3	4	13	1	0	1	12	1	1	0	0	8	1	10	0	0	4	1	4	12	20	16	2	0	2	7
ferret	355	310	3	215	0	349	346	326	0	327	346	322	302	0	322	0	277	1	0	320	0	0	7	2	287	173	3	4
fluidanimate	-	183	0	49	53	27	12	-	-	-	198	-	138	144	133	141	146	156	101	69	57	61	71	3	57	3	0	124
kyotocabinet	6	19	0	13	30	16	23	10	39	21	9	13	5	11	6	12	16	18	62	7	65	49	115	85	7	4	4	13
linear_regression	6	10	14	5	14	6	6	3	42	11	0	2	3	7	1	36	4	5	33	5	33	16	69	45	25	11	49	41
memcached-new	1	12	55	64	81	-	13	-	-	-	18	-	27	33	0	58	11	10	51	-	73	60	71	65	23	6	49	47
memcached-old	64	39	0	96	88	-	77	-	-	-	115	-	77	78	56	58	77	53	44	-	63	65	69	0	40	35	0	3
mysqld	-	-	-	-	37	-	-	-	-	-	-	-	-	4	-	8	-	58	0	-	2	5	-	-	-	-	-	-
ocean_cp	1	1	5	2	9	1	0	2	12	4	4	0	0	8	1	12	1	2	5	0	8	5	11	6	4	1	2	4
ocean_ncp	0	1	3	2	6	2	0	0	7	3	1	0	1	9	1	7	1	3	9	1	5	4	9	5	0	0	0	5
pca	15	15	15	15	50	15	16	15	50	15	16	16	15	0	15	58	15	16	18	15	23	20	17	17	15	13	15	15
pca_ll	32	27	0	8	122	26	43	25	120	26	26	27	21	4	28	124	24	21	42	30	57	52	43	28	32	19	26	29
radiosity	4	2	2	0	3	0	0	3	5	3	0	1	2	2	0	2	1	2	3	1	6	3	6	2	2	1	2	3
radiosity_ll	0	33	44	25	93	14	19	19	70	39	0	21	62	65	22	82	46	45	83	32	145	107	110	69	75	59	68	65
s_raytrace	2	9	53	35	126	16	18	9	221	17	0	25	30	68	0	217	14	17	109	29	101	86	86	68	69	62	69	72
s_raytrace_ll	6	3	44	28	38	10	12	0	464	6	7	19	45	23	0	466	8	16	304	54	187	66	459	407	433	186	437	461
sqlite	-	-	-	-	32	-	-	-	-	-	-	-	-	0	-	66	-	827	67	-	73	53	-	-	-	-	-	-
ssl_proxy	0	8	32	15	49	22	30	3	48	8	4	12	22	39	7	37	13	21	51	6	67	38	41	41	32	9	31	33
streamcluster	16	11	20	21	40	9	3	-	-	-	3	-	26	25	22	21	24	24	33	4	30	29	28	0	15	17	1	13
streamcluster_ll	23	23	36	43	62	3	0	-	-	-	3	-	77	78	61	56	78	46	57	13	53	75	69	9	28	27	9	21
upscaledb	6	17	29	8	190	11	12	14	179	18	5	12	0	4	16	181	21	32	98	21	83	57	73	48	28	29	30	30
vips	57	49	6	294	1	152	108	-	-	-	127	-	276	0	40	0	38	2	0	55	0	0	2	2	51	12	2	4
volrend	0	1	8	4	10	0	0	2	9	2	0	1	3	6	0	7	2	3	15	1	14	18	28	12	5	2	6	8
water_nsquared	89	42	0	5	4	1	1	55	55	53	6	3	3	3	3	3	3	8	1	0	0	0	1	0	1	0	0	31
water_spatial	297	42	0	4	3	2	1	217	218	57	6	2	4	4	4	4	4	5	2	1	1	0	1	0	1	0	0	35

Table 62. For each lock-sensitive application, at *max nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the performance gain is greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**I-48 machine in energy-saving mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	716	2	155	67	2k	2k	2k	655	2k	128	252	469	70	227	62	448	80	2	2k	11	20	10	5	2k	1k	0	411
ferret	640	560	16	614	1	609	589	556	1	558	629	660	662	0	552	0	537	11	0	557	1	0	14	14	527	401	14	14
kyotocabinet	14	33	39	0	161	21	18	37	487	34	17	19	58	78	30	423	28	50	121	53	125	91	578	322	66	41	64	72
linear_regression	11	6	6	12	74	21	1	3	83	28	0	21	8	7	4	79	47	11	10	8	12	10	68	53	17	13	18	13
memcached-new	90	24	2	25	87	-	42	-	-	-	64	-	13	45	36	296	41	31	50	-	72	56	126	89	34	0	30	33
memcached-old	14	0	93	26	29	-	8	-	-	-	38	-	186	171	91	93	170	166	150	-	195	189	96	80	183	183	99	94
mysqld	-	-	-	-	0	-	-	-	-	-	-	-	-	6	-	9	-	56	3	-	4	8	-	-	-	-	-	-
pca	8	17	3	0	108	3	10	15	315	14	3	11	6	5	9	286	12	14	29	27	36	29	243	190	51	36	42	48
pca_ll	3	10	52	9	280	0	1	16	925	14	1	1	19	72	9	980	9	23	85	30	103	190	691	550	80	46	146	132
radiosity	18	13	7	2	48	4	3	19	173	22	0	2	8	11	5	147	5	7	17	16	22	17	142	106	28	20	30	38
radiosity_ll	1	65	106	8	779	54	22	92	2k	91	0	25	129	149	58	2k	71	90	211	174	267	159	2k	1k	379	247	436	440
s_raytrace	0	13	9	28	453	0	7	23	1k	24	0	20	42	96	8	1k	11	12	58	66	62	101	787	689	161	101	177	243
s_raytrace_ll	2	14	45	29	151	13	15	31	1k	30	0	35	104	135	11	1k	13	26	63	58	68	74	845	817	176	98	213	245
sqlite	-	-	-	-	196	-	-	-	-	-	-	-	-	0	-	505	-	4k	37	-	35	15	-	-	-	-	-	-
ssl_proxy	7	47	22	0	473	45	26	55	1k	51	6	19	61	109	57	1k	45	62	129	98	134	94	2k	730	189	125	189	181
streamcluster	47	0	486	162	424	81	13	-	-	-	36	-	356	418	298	342	288	274	432	18	386	485	393	274	488	457	225	217
streamcluster_ll	49	0	466	177	433	121	2	-	-	-	52	-	359	467	346	379	319	303	444	13	456	569	424	308	522	502	268	298
upscaledb	3	23	27	0	63	4	4	26	281	26	4	0	26	21	27	303	22	36	138	36	138	106	195	187	58	39	57	58
vips	213	127	2	706	48	410	324	-	-	-	347	-	408	1	139	6	135	2	0	167	2	0	3	3	100	5	0	1
volrend	9	10	150	2	42	1	0	14	26	10	0	1	2	21	10	11	2	3	15	1	25	25	40	30	15	10	17	21
water_nsquared	128	52	0	0	4	2	4	97	89	90	5	2	2	0	3	6	2	6	2	0	1	5	2	3	2	5	0	44
water_spatial	917	320	2	11	9	2	1	614	618	620	13	8	9	9	9	9	9	8	0	1	2	0	0	0	0	1	0	321

Table 63. For each lock-sensitive application, at *opt nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the performance gain is greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**I-48 machine in energy-efficiency mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	451	5	86	68	18	5	726	715	728	93	65	60	59	59	59	59	94	0	7	0	0	1	7	5	2	1	381
ferret	46	45	8	48	1	44	50	49	0	46	45	81	63	0	45	0	64	7	0	55	1	0	8	9	45	44	7	8
kyotocabinet	10	25	6	0	14	12	9	22	384	24	13	15	31	58	23	413	20	35	94	33	106	71	128	90	42	30	29	32
linear_regression	10	6	6	11	45	9	1	3	79	21	0	6	8	7	4	79	10	11	10	8	12	10	68	45	17	13	16	13
memcached-new	53	24	2	0	37	-	0	-	-	-	62	-	9	13	12	68	37	17	9	-	18	13	5	9	14	0	10	11
memcached-old	0	13	13	9	11	-	23	-	-	-	15	-	72	70	32	31	59	60	62	-	71	78	35	29	85	74	15	42
mysqld	-	-	-	-	0	-	-	-	-	-	-	-	-	5	-	7	-	56	3	-	4	4	-	-	-	-	-	-
pca	8	7	10	0	31	2	4	10	277	10	6	8	1	11	10	308	9	11	26	8	32	21	12	14	17	10	7	12
pca_ll	3	10	52	9	195	0	1	16	403	14	1	1	18	45	9	186	9	19	75	20	89	97	182	69	40	37	39	39
radiosity	16	9	6	0	7	0	0	17	71	17	0	1	5	9	4	25	4	6	12	7	16	11	22	15	16	12	13	20
radiosity_ll	1	61	102	8	52	18	21	87	1k	88	0	24	129	149	57	2k	59	80	163	99	249	159	263	218	248	169	162	163
s_raytrace	0	7	9	24	90	0	7	13	179	18	0	13	38	88	5	211	6	12	57	31	54	80	172	172	94	60	62	84
s_raytrace_ll	2	14	44	29	151	8	15	31	342	30	0	35	104	135	11	343	13	26	63	58	68	74	343	345	176	98	213	245
sqlite	-	-	-	-	0	-	-	-	-	-	-	-	-	41	-	618	-	35	61	-	74	55	-	-	-	-	-	-
ssl_proxy	0	42	23	6	54	27	9	61	1k	53	0	12	80	153	48	1k	37	58	124	57	144	130	445	209	178	125	135	121
streamcluster	8	3	25	5	6	3	1	-	-	-	0	-	41	38	19	15	36	14	32	7	15	35	19	17	41	42	17	14
streamcluster_ll	21	0	125	43	41	16	15	-	-	-	15	-	151	158	91	99	103	109	145	21	78	160	122	75	161	161	84	75
upscaledb	1	23	27	0	63	2	1	26	264	26	1	0	17	21	26	274	22	36	124	35	124	106	68	67	48	37	46	47
vips	24	24	2	22	22	22	22	-	-	-	24	-	23	1	23	6	22	2	0	22	2	0	3	3	23	5	0	1
volrend	17	7	41	3	23	0	0	12	24	11	1	0	5	15	3	13	2	9	17	7	20	27	26	20	18	9	15	18
water_nsquared	128	52	0	0	4	2	4	97	89	90	5	2	2	0	3	6	2	6	2	0	1	5	2	3	2	5	0	44
water_spatial	917	320	2	11	9	2	1	614	618	620	13	8	9	9	9	9	9	8	0	1	2	0	0	0	0	1	0	321

Table 64. For each lock-sensitive application, at *max nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the performance gain is greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**I-20 machine in energy-saving mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	1k	25	261	56	178	165	695	501	758	428	202	316	56	233	54	305	83	1	132	0	1	22	24	110	222	25	307
ferret	363	343	46	369	2	367	366	337	0	342	354	367	407	0	338	0	338	29	0	339	0	0	66	67	340	296	68	69
kyotocabinet	4	12	6	0	251	8	3	14	253	17	10	6	27	19	10	249	10	16	70	21	61	38	112	71	22	12	21	20
memcached-old	2	3	3	2	315	-	2	-	-	-	3	-	1	20	1	254	2	3	81	-	97	36	5	2	0	1	2	4
pca	1	5	10	4	184	0	4	6	167	5	3	3	4	11	5	163	3	2	14	3	15	31	36	5	7	2	3	10
pca_ll	2	3	7	1	473	2	0	0	423	2	1	0	3	3	1	437	1	1	9	0	16	3	1	0	0	0	0	1
radiosity	23	11	5	5	52	1	3	19	77	17	2	1	6	13	3	60	0	4	7	12	11	21	50	22	14	7	14	16
radiosity_ll	2	9	33	0	577	2	0	16	799	22	1	4	29	46	8	802	10	15	56	59	76	156	427	167	89	40	78	59
s_raytrace	0	0	1	2	337	0	0	0	312	1	1	1	1	3	0	224	0	0	2	0	3	3	2	0	1	0	0	0
s_raytrace_ll	2	2	4	3	582	1	1	2	641	2	2	2	2	1	1	643	1	2	3	1	5	1	0	1	1	1	1	1
sqlite	-	-	-	-	321	-	-	-	-	-	-	-	-	0	-	388	-	9k	22	-	27	58	-	-	-	-	-	-
streamcluster	23	0	61	17	33	24	12	-	-	-	9	-	55	55	49	45	61	51	387	6	168	129	85	27	82	80	31	29
streamcluster_ll	20	0	67	17	29	26	9	-	-	-	9	-	59	60	50	50	67	57	466	0	233	158	91	39	80	90	34	36
upscaledb	3	11	21	1	143	0	7	14	152	13	1	3	5	13	10	150	10	14	129	21	125	123	56	35	13	11	12	20
vips	115	62	3	50	0	62	21	-	-	-	85	-	39	0	25	0	29	1	1	80	2	1	3	2	49	52	2	1
water_nsquared	47	17	0	0	1	0	0	32	31	33	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16
water_spatial	325	110	0	3	3	0	0	219	212	211	4	3	2	3	3	2	3	2	0	0	0	0	0	0	0	1	0	107

Table 65. For each lock-sensitive application, at *opt nodes*, performance gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the performance gain is greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**I-20 machine in energy-efficiency mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	437	5	82	81	15	9	722	711	703	105	66	80	74	78	71	76	77	3	8	2	0	4	6	8	6	6	378
ferret	101	92	46	103	2	102	103	92	0	91	102	109	136	0	90	0	89	29	0	89	0	0	54	54	87	90	54	53
kyotocabinet	3	13	3	0	249	6	3	14	255	16	8	6	22	21	12	250	10	18	74	21	58	32	53	33	19	12	17	18
memcached-old	2	3	3	2	315	-	2	-	-	-	3	-	1	20	1	254	2	3	81	-	97	36	5	2	0	1	2	4
pca	1	5	10	4	23	0	4	6	19	5	3	3	4	11	5	20	3	2	14	3	15	14	15	5	7	2	3	10
pca_ll	2	3	7	1	87	2	0	0	88	2	1	0	3	3	1	86	1	1	9	0	16	3	1	0	0	0	0	1
radiosity	23	11	5	5	5	1	2	19	20	17	2	1	6	6	3	5	0	4	4	6	2	5	6	7	7	5	5	13
radiosity_ll	2	9	33	0	61	2	0	16	65	22	1	4	29	46	8	54	10	15	56	56	69	63	57	54	56	40	53	53
s_raytrace	0	0	1	2	89	0	0	0	91	1	1	1	1	3	0	88	0	0	2	0	3	3	2	0	1	0	0	0
s_raytrace_ll	2	2	4	3	76	1	1	2	77	2	2	2	2	1	1	77	1	2	3	1	5	1	0	1	1	1	1	1
sqlite	-	-	-	-	25	-	-	-	-	-	-	-	-	0	-	394	-	16	47	-	49	42	-	-	-	-	-	-
streamcluster	8	0	28	19	27	12	18	-	-	-	13	-	24	16	18	17	25	10	28	15	3	23	25	8	36	34	19	13
streamcluster_ll	20	0	67	17	29	26	9	-	-	-	9	-	59	60	50	50	67	57	112	0	67	100	91	39	80	90	34	36
upscaledb	3	11	21	1	143	0	7	14	145	13	1	3	5	13	10	147	10	14	129	21	125	123	34	29	13	11	12	20
vips	93	62	3	50	0	62	21	-	-	-	85	-	39	0	25	0	29	1	1	80	2	1	3	2	49	52	2	1
water_nsquared	47	17	0	0	1	0	0	32	31	33	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16
water_spatial	325	110	0	3	3	0	0	219	212	211	4	3	2	3	3	2	3	2	0	0	0	0	0	0	0	1	0	107

A.7 Impact of the number of nodes.

Table 66. For each lock-sensitive application, percentage of pairwise changes in the lock performance hierarchy when changing the number of nodes (**A-48 machine**).

	% of pairwise changes between configurations			
Applications	1/2	2/4	4/8	1/2/4/8
dedup	14%	10%	22%	32%
ferret	0%	72%	15%	83%
fmm	23%	23%	18%	36%
kyotocabinet	25%	8%	14%	38%
linear_regression	18%	36%	32%	61%
memcached-new	58%	39%	0%	76%
memcached-old	37%	29%	0%	55%
mysqld	29%	0%	5%	33%
pca	31%	33%	29%	76%
pca_ll	20%	25%	53%	91%
radiosity	31%	45%	15%	76%
radiosity_ll	30%	53%	18%	84%
s_raytrace	21%	43%	33%	94%
s_raytrace_ll	24%	51%	27%	96%
sqlite	5%	14%	52%	67%
ssl_proxy	35%	26%	14%	56%
streamcluster	15%	59%	35%	85%
streamcluster_ll	32%	49%	38%	95%
upscaledb	23%	16%	11%	44%
vips	0%	5%	84%	84%
volrend	19%	21%	39%	77%
water_nsquared	29%	28%	22%	60%
water_spatial	15%	15%	6%	31%

Table 67. For each lock-sensitive application, percentage of pairwise changes in the lock performance hierarchy when changing the number of nodes (**I-48 machine in performance mode**).

	% of pairwise changes between configurations			
Applications	1/2	2/3	3/4	1/2/3/4
dedup	13%	28%	22%	48%
ferret	26%	65%	15%	87%
kyotocabinet	12%	7%	4%	19%
linear_regression	34%	38%	39%	78%
memcached-new	47%	29%	0%	56%
memcached-old	14%	15%	0%	25%
mysqld	7%	29%	24%	38%
pca	47%	12%	15%	59%
pca_ll	41%	30%	14%	76%
radiosity	25%	15%	10%	42%
radiosity_ll	23%	10%	7%	31%
s_raytrace	65%	19%	9%	89%
s_raytrace_ll	86%	15%	10%	98%
sqlite	29%	33%	19%	57%
ssl_proxy	14%	4%	6%	20%
streamcluster	24%	22%	23%	44%
streamcluster_ll	20%	19%	25%	43%
upscaledb	7%	8%	6%	15%
vips	0%	0%	76%	76%
volrend	31%	34%	21%	71%
water_nsquared	0%	0%	4%	4%
water_spatial	13%	13%	5%	29%

Table 68. For each lock-sensitive application, percentage of pairwise changes in the lock performance hierarchy when changing the number of nodes (**I-20 machine in performance mode**).

	% of pairwise changes between configurations
Applications	1/2
dedup	27%
ferret	18%
kyotocabinet	9%
memcached-old	0%
pca	52%
pca_ll	37%
radiosity	56%
radiosity_ll	75%
s_raytrace	21%
s_raytrace_ll	21%
sqlite	48%
streamcluster	46%
streamcluster_ll	46%
upscaledb	13%
vips	74%
water_nsquared	0%
water_spatial	0%

Table 69. For each lock-sensitive application, percentage of pairwise changes in the lock performance hierarchy when changing the number of nodes (**A-64 machine with thread-to-node pinning**).

	% of pairwise changes between configurations			
Applications	1/2	2/4	4/8	1/2/4/8
dedup	10%	11%	13%	19%
facesim	0%	43%	30%	73%
ferret	23%	13%	15%	41%
fluidanimate	28%	9%	10%	36%
kyotocabinet	30%	16%	10%	47%
linear_regression	27%	50%	25%	80%
memcached-new	52%	23%	0%	68%
memcached-old	36%	20%	0%	51%
mysqld	26%	14%	38%	57%
ocean_cp	0%	30%	46%	76%
ocean_ncp	0%	25%	48%	74%
pca	25%	48%	16%	81%
pca_ll	8%	53%	58%	95%
radiosity	0%	54%	12%	66%
radiosity_ll	53%	52%	14%	96%
s_raytrace	5%	46%	44%	88%
s_raytrace_ll	0%	87%	23%	96%
sqlite	45%	10%	5%	45%
ssl_proxy	62%	15%	13%	74%
streamcluster	62%	24%	23%	84%
streamcluster_ll	56%	23%	26%	81%
upscaledb	47%	20%	20%	58%
vips	13%	6%	15%	26%
volrend	23%	22%	36%	80%
water_nsquared	20%	10%	7%	38%
water_spatial	3%	0%	3%	6%

Table 70. For each lock-sensitive application, percentage of pairwise changes in the lock performance hierarchy when changing the number of nodes (**I-48 machine in energy-saving mode**).

	% of pairwise changes between configurations			
Applications	1/2	2/3	3/4	1/2/3/4
dedup	13%	28%	22%	48%
ferret	26%	65%	15%	87%
kyotocabinet	12%	7%	4%	19%
linear_regression	34%	38%	39%	78%
memcached-new	47%	29%	0%	56%
memcached-old	14%	15%	0%	25%
mysqld	7%	29%	24%	38%
pca	47%	12%	15%	59%
pca_ll	41%	30%	14%	76%
radiosity	25%	15%	10%	42%
radiosity_ll	23%	10%	7%	31%
s_raytrace	65%	19%	9%	89%
s_raytrace_ll	86%	15%	10%	98%
sqlite	29%	33%	19%	57%
ssl_proxy	14%	4%	6%	20%
streamcluster	24%	22%	23%	44%
streamcluster_ll	20%	19%	25%	43%
upscaledb	7%	8%	6%	15%
vips	0%	0%	76%	76%
volrend	31%	34%	21%	71%
water_nsquared	0%	0%	4%	4%
water_spatial	13%	13%	5%	29%

Table 71. For each lock-sensitive application, percentage of pairwise changes in the lock performance hierarchy when changing the number of nodes (**I-20 machine in energy-saving mode**).

	% of pairwise changes between configurations
Applications	1/2
dedup	27%
ferret	18%
kyotocabinet	9%
memcached-old	0%
pca	52%
pca_ll	37%
radiosity	56%
radiosity_ll	75%
s_raytrace	21%
s_raytrace_ll	21%
sqlite	48%
streamcluster	46%
streamcluster_ll	46%
upscaledb	13%
vips	74%
water_nsquared	0%
water_spatial	0%

A.8 Impact of the machine.

Table 72. For each pair of machines, at *max nodes* and *opt nodes*, percentage of pairwise changes in the lock performance hierarchy (**all machines**).

# nodes	A-64 vs. A-48	A-48 vs. I-48	A-64 vs. I-48	I-48 vs. I-20
Max	25%	26%	28%	33%
Opt	31%	36%	34%	36%

B STUDY OF LOCK ENERGY EFFICIENCY

B.1 Selection of lock sensitive application

Table 73. For each application, energy-efficiency gain of the best vs. worst lock and relative standard deviation (**I-48 machine in energy-saving mode**).

	Gain *one node*	R.Dev. *one node*	Gain *max nodes*	R.Dev. *max nodes*	Gain *opt nodes*	R.Dev. *opt nodes*
barnes	7%	2%	17%	4%	17%	4%
blackscholes	1%	0%	1%	0%	1%	0%
bodytrack	1%	0%	80%	9%	11%	3%
canneal	1%	0%	2%	0%	2%	0%
dedup	**619%**	**44%**	**2789%**	**68%**	**619%**	**44%**
ferret	**1%**	**0%**	**569%**	**75%**	**28%**	**8%**
fmm	6%	2%	22%	6%	18%	4%
freqmine	2%	0%	1%	0%	1%	0%
histogram	17%	3%	30%	6%	17%	3%
kmeans	2%	0%	7%	2%	4%	1%
kyotocabinet	**293%**	**26%**	**967%**	**37%**	**293%**	**26%**
linear_regression	**8%**	**2%**	**192%**	**22%**	**86%**	**14%**
lu_cb	3%	1%	2%	1%	2%	1%
lu_ncb	7%	2%	4%	1%	4%	1%
matrix_multiply	2%	1%	7%	2%	7%	2%
memcached-new	**107%**	**21%**	**629%**	**27%**	**88%**	**17%**
memcached-old	**69%**	**18%**	**191%**	**37%**	**69%**	**18%**
mysqld	**103%**	**19%**	**87%**	**18%**	**87%**	**18%**
p_raytrace	2%	1%	3%	1%	1%	0%
pca	**204%**	**19%**	**778%**	**35%**	**204%**	**19%**
pca_ll	**16%**	**3%**	**1139%**	**44%**	**52%**	**14%**
radiosity	**36%**	**7%**	**577%**	**31%**	**39%**	**8%**
radiosity_ll	**169%**	**22%**	**4028%**	**62%**	**223%**	**28%**
rocksdb	3%	1%	7%	2%	7%	2%
s_raytrace	**3%**	**1%**	**2308%**	**49%**	**81%**	**20%**
s_raytrace_ll	**2%**	**1%**	**1941%**	**45%**	**189%**	**33%**
sqlite	**359%**	**35%**	**5657%**	**75%**	**395%**	**37%**
ssl_proxy	**793%**	**37%**	**2306%**	**51%**	**804%**	**38%**
streamcluster	**43%**	**11%**	**520%**	**65%**	**43%**	**11%**
streamcluster_ll	**60%**	**15%**	**613%**	**74%**	**98%**	**22%**
string_match	1%	0%	8%	2%	8%	2%
swaptions	1%	0%	2%	0%	2%	0%
upscaledb	**586%**	**30%**	**768%**	**39%**	**586%**	**30%**
vips	**2%**	**0%**	**636%**	**46%**	**9%**	**3%**
volrend	11%	2%	44%	9%	19%	4%
water_nsquared	**31%**	**7%**	**67%**	**13%**	**67%**	**13%**
water_spatial	**303%**	**31%**	**589%**	**38%**	**589%**	**38%**
word_count	4%	1%	5%	1%	4%	1%
x264	1%	0%	1%	0%	1%	0%

Table 74. For each application, energy-efficiency gain of the best vs. worst lock and relative standard deviation (**I-20 machine in energy-saving mode**).

	Gain *one node*	R.Dev. *one node*	Gain *max nodes*	R.Dev. *max nodes*	Gain *opt nodes*	R.Dev. *opt nodes*
barnes	5%	1%	7%	2%	7%	2%
blackscholes	1%	0%	1%	0%	1%	0%
bodytrack	7%	2%	2%	1%	2%	1%
canneal	1%	0%	2%	1%	2%	1%
dedup	**489%**	**41%**	**1171%**	**46%**	**489%**	**41%**
ferret	**40%**	**9%**	**325%**	**61%**	**75%**	**18%**
fmm	5%	1%	8%	2%	8%	2%
freqmine	8%	1%	1%	0%	1%	0%
histogram	8%	2%	30%	6%	8%	2%
kmeans	2%	0%	3%	1%	2%	0%
kyotocabinet	**747%**	**32%**	**1684%**	**34%**	**747%**	**32%**
linear_regression	**10%**	**2%**	**102%**	**13%**	**24%**	**5%**
lu_cb	1%	0%	1%	0%	1%	0%
lu_ncb	7%	2%	8%	1%	8%	1%
matrix_multiply	2%	0%	5%	1%	5%	1%
memcached-new	47%	9%	47%	9%	47%	9%
memcached-old	**204%**	**25%**	**204%**	**25%**	**204%**	**25%**
p_raytrace	4%	1%	3%	1%	2%	1%
pca	**8%**	**2%**	**1314%**	**28%**	**18%**	**5%**
pca_ll	**6%**	**1%**	**1020%**	**29%**	**37%**	**8%**
radiosity	**19%**	**4%**	**406%**	**24%**	**20%**	**5%**
radiosity_ll	**16%**	**3%**	**4327%**	**42%**	**32%**	**8%**
rocksdb	7%	1%	7%	2%	7%	2%
s_raytrace	**4%**	**1%**	**2043%**	**28%**	**47%**	**10%**
s_raytrace_ll	**2%**	**0%**	**2581%**	**29%**	**32%**	**7%**
sqlite	**364%**	**34%**	**5444%**	**78%**	**364%**	**34%**
streamcluster	**25%**	**6%**	**118%**	**20%**	**25%**	**6%**
streamcluster_ll	**23%**	**7%**	**153%**	**24%**	**79%**	**19%**
string_match	1%	0%	3%	1%	3%	1%
swaptions	1%	0%	1%	0%	1%	0%
upscaledb	**661%**	**36%**	**1027%**	**37%**	**661%**	**36%**
vips	**1%**	**0%**	**66%**	**18%**	**49%**	**17%**
volrend	15%	4%	39%	6%	15%	4%
water_nsquared	20%	5%	27%	7%	27%	7%
water_spatial	**207%**	**26%**	**296%**	**30%**	**296%**	**30%**
word_count	3%	1%	9%	2%	3%	1%
x264	3%	1%	2%	1%	2%	1%

Table 75. Number of tested applications and number of lock energy efficiency sensitive applications (**all machines**).

	I-48	I-20
# tested applications	38	36
# lock-sensitive applications	20	17
ratio	53%	47%

B.2 Selection of the number of nodes

Table 76. For each *(lock-sensitive application, lock)* pair, energy-efficiency gain (in %) of *opt nodes* over *max nodes*. The background color of a cell indicates the number of nodes for *opt nodes*: 1|2|3|4. Dashes correspond to untested cases. (**I-48 machine in energy-saving mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	221	95	151	105	1k	900	323	86	665	101	226	119	104	113	115	154	107	103	368	83	110	108	102	4k	3k	90	113
ferret	441	397	6	400		396	406	378		395	428	461	415		393		399			395					362	271	6	5
kyotocabinet	68	70	105	60	200	84	73	73	335	67	65	69	87	34	73	345	74	94	65	88	83	75	255	232	88	77	96	102
linear_regression	16				70				46	6					9	58	44			15		5	11	11		6		
memcached-new	40	14	26	44	62	-	58	-	-	-	7	-	8	12		484	6	17	52	-	91	64	211	140	64	30	23	26
memcached-old	78	50	152	97	96	-	69	-	-	-	102	-	183	185	147	139	213	167	176	-	195	187	138	184	166	158	148	129
mysqld	-	-	-	-		-	-	-	-	-	-	-	-		-	8	-			-		5	-	-	-	-	-	-
pca	31	40	19	18	94	37	27	46	247	40	28	37	35	30	40	232	39	36	47	54	49	57	367	306	89	76	88	86
pca_ll		9	12		135			9	701	10			10	98	8	744	8	19	30	19	30	59	304	262	55	30	76	71
radiosity		9	9		43	10	6	11	389	9			7	5	7	366	9	11	20	26	19	17	174	125	27	25	37	38
radiosity_ll		32	47		473	45	13	41	1k	40		14	50	9	29	2k	37	49	70	104	58	38	728	595	98	99	196	199
s_raytrace		15		9	188	7	6	17	1k	20		15	15	23	16	1k	9	9	19	37	21	35	317	301	65	59	100	130
s_raytrace_ll	8		14	7		13	13		665		9	11	5			671							178	177				20
sqlite	-	-	-	-	256	-	-	-	-	-	-	-	-		-	381	-	6k	75	-	63	47	-	-	-	-	-	-
ssl_proxy	86	121	101	71	474	122	99	96	351	107	82	93	89	50	123	366	121	125	124	162	107	76	552	478	110	125	169	177
streamcluster	1k	573	2k	2k	2k	1k	595	-	-	-	793	-	2k	2k	2k	2k	1k	2k	2k	427	2k	2k	2k	2k	2k	2k	1k	2k
streamcluster_ll	315	72	330	472	555	234	60	-	-	-	126	-	302	504	302	495	226	307	353	48	350	299	397	474	367	359	324	319
upscaledb	45	49	50	36	58	50	44	46	84	46	44	42	41		48	54	46	63	51	54	61	68	236	163	55	50	40	61
vips	75	118		581	36	255	163	-	-	-	165	-	338		91		124			132					114	6		
water_nsquared																												
water_spatial																												

Table 77. For each *(lock-sensitive application, lock)* pair, energy-efficiency gain (in %) of *opt nodes* over *max nodes*. The background color of a cell indicates the number of nodes for *opt nodes*: 1 2. Dashes correspond to untested cases. (**I-20 machine in energy-saving mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	321	49	83	52	170	109	85	29	78	72	54	82	50	48	53	87	49	48	148	46	43	44	44	141	208	48	35
ferret	169	167		182		179	174	168		170	177	177	142		170		169			173					174	143		
kyotocabinet	36	33	39	34		39	46	36	164	37	37	37	42	8	35	162	36	34	6	41	20	42	79	67	45	35	43	37
linear_regression									61							64												
memcached-old						-		-	-	-		-								-								
pca					49				1k							1k					14	27	24					
pca_ll					154				490							730												
radiosity					18				323					6		284			12	10	16	23	74	27	12		14	11
radiosity_ll			8		157			7	3k				8	10		4k			27	32	44	68	345	116	58	21	44	32
s_raytrace					28				1k							1k												
s_raytrace_ll					63				2k							2k												
sqlite	-	-	-	-	409	-	-	-	-	-	-	-	-	21	-	452	-	7k	7	-	43	50	-	-	-	-	-	-
streamcluster	82	46	130	46	60	54	64	-	-	-	53	-	103	113	98	104	95	106	130	58	127	178	124	86	113	132	93	87
streamcluster_ll								-	-	-		-							25		19	45	13		13	9		
upscaledb	20	21	19	25	26	19	24	23	72	20	17	19	17		19	51	18	25		23			45	35	19	15	17	25
vips				11		7		-	-	-	5	-																
water_spatial																												

B.3 Are some locks always among the best?

Table 78. For each lock, fraction of the lock-sensitive applications for which the lock yields the best energy-efficiency for three configurations: *one node*, *max nodes* and *opt nodes* (**I-48 machine in energy-saving mode**).

	Number of nodes		
Locks	*one node*	*max nodes*	*opt nodes*
ahmcs	56%	17%	50%
alock-ls	53%	16%	32%
backoff	68%	21%	37%
c-bo-mcs_spin	68%	37%	53%
c-bo-mcs_stp	57%	14%	24%
c-ptl-tkt	76%	24%	59%
c-tkt-tkt	79%	21%	53%
clh_spin	43%	7%	14%
clh_stp	29%	7%	7%
clh-ls	43%	0%	21%
hmcs	74%	37%	58%
hticket-ls	71%	21%	43%
malth_spin	53%	11%	16%
malth_stp	43%	33%	24%
mcs_spin	58%	11%	37%
mcs_stp	33%	19%	19%
mcs-ls	58%	11%	37%
mcs-timepub	43%	10%	24%
mutexee	38%	19%	24%
partitioned	65%	24%	29%
pthread	38%	24%	24%
pthreadadapt	43%	19%	29%
spinlock	42%	16%	21%
spinlock-ls	53%	16%	32%
ticket	53%	11%	21%
ticket-ls	53%	11%	21%
ttas	53%	21%	26%
ttas-ls	37%	5%	11%

Table 79. For each lock, fraction of the lock-sensitive applications for which the lock yields the best energy-efficiency for three configurations: *one node*, *max nodes* and *opt nodes* (**I-20 machine in energy-saving mode**).

	Number of nodes		
Locks	*one node*	*max nodes*	*opt nodes*
ahmcs	60%	40%	53%
alock-ls	50%	44%	38%
backoff	69%	44%	50%
c-bo-mcs_spin	75%	50%	62%
c-bo-mcs_stp	53%	18%	24%
c-ptl-tkt	73%	53%	67%
c-tkt-tkt	81%	56%	69%
clh_spin	50%	33%	33%
clh_stp	33%	8%	8%
clh-ls	50%	33%	33%
hmcs	69%	50%	56%
hticket-ls	83%	58%	75%
malth_spin	56%	38%	38%
malth_stp	53%	53%	47%
mcs_spin	62%	44%	44%
mcs_stp	53%	18%	18%
mcs-ls	56%	44%	44%
mcs-timepub	59%	47%	53%
mutexee	59%	47%	47%
partitioned	80%	47%	60%
pthread	59%	24%	24%
pthreadadapt	59%	47%	53%
spinlock	62%	38%	38%
spinlock-ls	69%	44%	50%
ticket	69%	31%	38%
ticket-ls	69%	44%	56%
ttas	81%	44%	56%
ttas-ls	56%	31%	31%

B.4 Is there a clear hierarchy between locks?

Table 80. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A is more energy-efficient by at least 5% than B (**I-48 machine in energy-saving mode**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		33	44	17	61	25	28	46	77	38	22	23	61	61	50	72	50	50	67	50	67	61	67	61	67	61	61	61	51
alock-ls	28		37	11	42	12	5	57	93	57	11	7	47	47	21	68	21	47	63	41	63	58	63	58	68	47	47	58	44
backoff	33	42		26	58	29	16	50	93	50	32	29	63	58	32	68	42	47	47	41	63	58	58	42	63	63	47	68	49
c-bo-mcs_spin	28	47	42		53	24	11	64	93	64	21	21	63	63	42	74	42	58	74	41	68	63	63	58	58	58	63	79	53
c-bo-mcs_stp	28	37	26	5		6	5	43	86	50	21	7	47	43	37	67	32	43	43	18	48	43	53	47	47	42	53	68	39
c-ptl-tkt	19	53	53	18	59		6	79	93	71	12	29	65	71	47	76	41	59	71	47	71	65	65	65	65	65	65	82	56
c-tkt-tkt	28	53	47	16	63	18		71	93	71	26	14	74	68	47	79	53	58	74	53	68	63	63	58	68	63	68	84	57
clh_spin	15	0	14	7	43	7	0		71	0	0	0	43	43	0	71	0	21	57	21	57	57	64	57	50	36	43	50	31
clh_stp	23	7	7	7	0	7	7	7		7	7	7	7	0	7	7	7	7	0	7	0	0	7	7	7	7	7	7	6
clh-ls	15	0	14	7	43	7	0	7	71		0	0	36	43	0	71	0	29	57	14	57	57	64	57	57	29	36	36	30
hmcs	17	47	53	21	47	18	16	79	93	71		29	63	58	42	68	42	53	68	41	63	63	68	58	68	63	58	79	54
hticket-ls	23	57	29	14	57	7	0	57	93	57	7		57	71	29	71	21	43	71	57	71	64	64	64	71	64	64	86	51
malth_spin	22	21	11	5	32	0	0	29	93	29	11	0		21	5	58	5	26	37	6	42	37	53	47	53	21	21	53	27
malth_stp	28	32	16	11	33	12	11	29	93	29	21	7	16		16	62	16	29	29	12	29	38	53	53	37	21	21	37	29
mcs_spin	17	21	37	16	42	6	0	57	93	50	11	14	47	53		53	16	53	68	41	58	63	53	47	63	53	47	58	42
mcs_stp	17	21	16	11	5	6	5	29	29	29	16	7	21	14	11		26	24	14	6	5	14	11	5	21	21	5	21	15
mcs-ls	17	16	21	11	42	6	0	57	93	57	11	7	47	47	5	53		42	53	35	53	53	53	47	74	47	42	63	39
mcs-timepub	22	21	16	11	33	6	5	36	93	36	16	7	32	33	11	52	16		48	24	48	52	53	47	53	42	26	47	33
mutexee	33	32	5	21	33	18	11	29	93	29	26	21	42	33	21	67	26	33		12	29	29	47	37	53	32	16	37	32
partitioned	19	18	29	6	47	18	0	29	93	29	18	7	41	47	18	76	18	24	59		59	65	65	65	59	35	41	65	39
pthread	28	32	16	16	29	18	11	29	93	29	26	14	37	24	21	67	32	33	14	12		19	53	26	42	26	21	32	30
pthreadadapt	28	32	11	16	29	18	11	29	86	29	26	14	32	19	21	62	21	29	14	12	19		42	37	47	21	16	32	28
spinlock	22	26	11	16	26	12	5	29	71	29	21	14	37	21	16	47	32	26	26	6	16	26		0	21	21	5	21	22
spinlock-ls	22	32	21	26	37	12	11	29	79	29	32	21	37	32	21	63	32	26	32	12	32	37	53		26	26	16	26	30
ticket	17	21	11	11	37	6	5	21	93	21	16	7	11	16	11	63	11	11	26	0	26	21	47	32		5	5	32	22
ticket-ls	22	21	11	5	37	12	0	21	93	21	16	7	16	21	11	63	11	32	26	0	42	37	47	47	42		16	37	26
ttas	28	32	16	21	42	18	11	29	93	29	26	21	37	32	26	74	37	42	32	12	37	47	58	42	53	26		37	35
ttas-ls	28	21	16	11	32	12	11	29	86	29	16	7	26	26	16	58	32	21	32	12	37	47	47	37	53	26	0		28
average	23	29	23	13	39	12	7	40	86	38	17	13	41	39	22	63	25	36	45	23	45	46	53	45	51	38	34	50	23

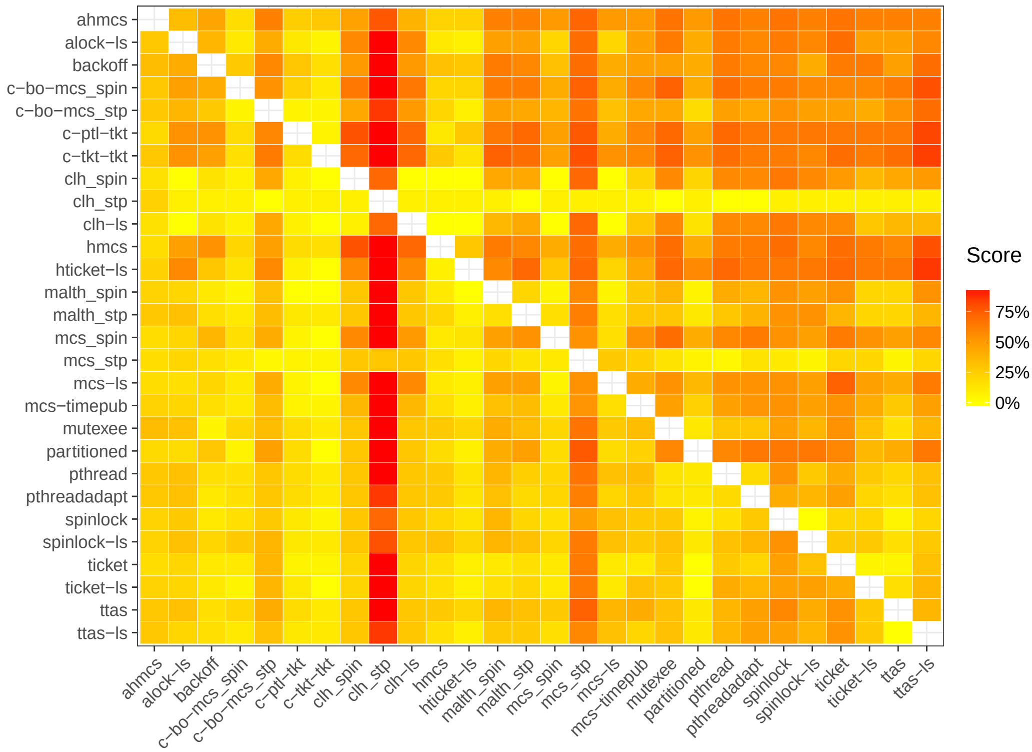

Fig. 21. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A is more energy-efficient at least 5% better than B (**I-48 machine in energy-saving move**).

Table 81. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A is more energy-efficient by at least 5% than B (**I-20 machine in energy-saving mode**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		27	33	13	67	14	7	27	73	27	13	0	40	27	20	67	33	33	53	29	47	47	53	27	33	33	20	33	33
alock-ls	20		31	12	69	13	6	25	92	33	12	0	31	19	6	56	6	12	50	13	44	44	50	19	25	12	12	19	27
backoff	33	38		31	56	27	19	50	83	42	31	17	31	19	31	50	31	19	31	20	38	25	25	19	31	25	12	25	32
c-bo-mcs_spin	20	38	31		62	13	6	58	92	50	19	8	31	31	31	62	25	25	44	20	50	44	50	38	44	25	25	50	37
c-bo-mcs_stp	27	25	12	12		20	12	33	58	33	19	8	19	12	19	29	19	12	24	13	18	18	19	12	25	19	12	25	21
c-ptl-tkt	14	40	40	7	67		0	50	92	58	7	8	47	40	27	73	33	33	40	20	53	33	47	40	40	27	33	53	38
c-tkt-tkt	20	38	31	12	75	13		58	92	50	12	8	44	38	25	69	25	31	50	20	56	44	50	44	44	31	25	50	39
clh_spin	9	0	25	0	58	0	0		67	0	0	0	17	0	0	58	0	8	42	8	42	33	33	17	8	0	8	17	17
clh_stp	18	8	8	8	0	8	8	8		8	8	8	8	0	8	8	8	8	0	8	0	0	8	8	8	8	8	8	7
clh-ls	9	0	25	0	58	0	0	0	67		0	0	8	0	0	58	0	0	42	8	42	33	33	17	8	0	8	17	16
hmcs	20	31	31	6	62	13	0	58	92	50		0	31	31	19	62	19	25	44	20	50	44	50	38	44	31	25	50	35
hticket-ls	18	50	33	8	75	0	0	50	92	50	8		33	25	17	67	17	33	42	25	42	33	42	25	33	25	25	58	34
malth_spin	20	25	19	6	50	13	6	25	92	25	19	0		6	6	50	6	6	50	7	38	44	38	12	12	12	0	25	23
malth_stp	27	31	19	12	47	20	12	42	92	42	19	8	19		19	53	19	18	35	27	29	29	31	19	31	31	19	25	29
mcs_spin	13	31	31	6	56	13	6	33	92	33	19	0	25	25		56	0	25	50	20	38	44	50	25	25	25	12	44	30
mcs_stp	27	25	12	12	6	20	12	33	33	33	19	8	19	6	12		12	12	24	13	12	18	19	6	25	25	6	19	17
mcs-ls	13	25	31	6	56	13	6	33	92	42	19	0	19	25	0	56		19	50	20	38	44	50	25	31	25	12	44	29
mcs-timepub	27	31	31	12	47	20	12	33	92	33	25	8	12	12	12	53	12		53	13	41	47	50	12	25	25	6	31	29
mutexee	27	31	6	19	47	20	12	33	75	33	19	17	19	12	19	47	19	12		13	12	0	12	6	12	12	6	25	21
partitioned	29	27	33	20	67	13	13	25	92	25	20	8	27	27	20	67	20	20	47		47	40	40	27	20	13	7	40	31
pthread	27	25	12	25	59	27	12	33	75	33	19	17	31	12	25	53	31	18	18	13		12	25	6	25	25	6	19	25
pthreadadapt	27	31	6	19	53	20	12	33	75	33	19	17	19	12	19	53	19	12	12	13	18		19	6	12	12	6	25	22
spinlock	27	25	6	19	50	13	12	33	83	33	19	17	19	12	19	50	19	6	38	13	38	19		0	12	12	0	12	22
spinlock-ls	27	31	25	25	62	27	12	33	92	33	25	17	31	19	25	62	25	12	44	13	38	31	31		25	25	0	12	30
ticket	13	19	19	12	56	13	6	25	92	25	19	8	12	12	6	56	6	12	31	7	38	19	25	12		0	0	12	21
ticket-ls	13	25	19	12	56	7	6	25	92	25	19	8	12	19	6	56	6	12	44	13	44	31	38	25	12		6	31	25
ttas	27	31	25	25	62	27	12	33	92	33	25	17	31	19	25	69	31	19	50	13	50	44	44	6	25	25		25	33
ttas-ls	27	19	19	12	50	20	12	33	83	33	19	8	19	12	19	50	19	6	44	13	31	38	31	6	31	25	0		25
average	21	27	23	13	55	15	8	34	83	34	17	8	24	17	16	55	17	17	39	16	37	32	36	18	25	20	11	29	21

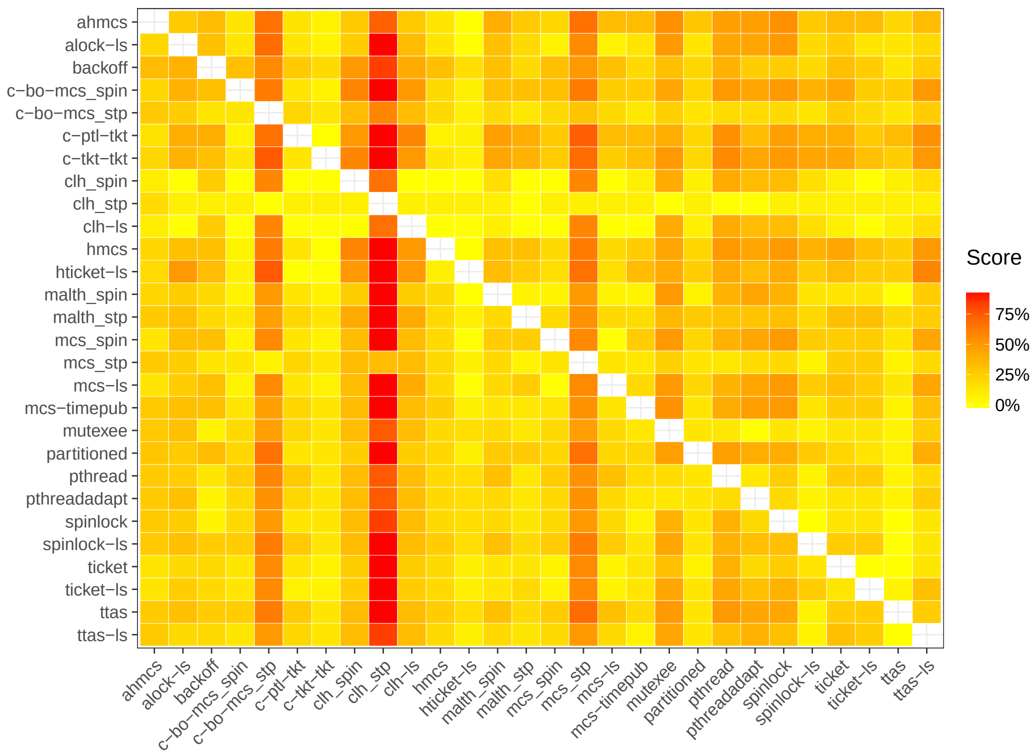

Fig. 22. For each pair of locks *(rowA, colB)* at *opt nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A is more energy-efficient at least 5% better than B (**I-20 machine in energy-saving move**).

Table 82. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A is more energy-efficient by at least 5% than B (**I-48 machine in energy-saving mode**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		50	56	39	72	38	22	69	77	69	11	38	61	44	67	78	67	61	67	69	67	67	78	78	72	67	67	67	60
alock-ls	44		32	32	74	24	26	57	86	57	32	14	58	47	26	74	21	53	74	65	74	74	74	74	74	68	63	68	54
backoff	33	42		26	79	24	21	57	93	57	26	21	58	47	37	74	37	42	74	71	63	68	74	68	89	79	53	74	55
c-bo-mcs_spin	50	47	58		68	47	32	71	93	79	32	50	68	63	42	74	47	63	68	65	74	74	74	74	74	74	53	79	63
c-bo-mcs_stp	28	26	11	16		18	16	29	93	29	21	14	21	24	16	76	21	19	14	18	19	14	63	74	37	32	16	37	30
c-ptl-tkt	31	47	35	29	71		6	79	86	79	6	21	59	53	41	71	35	65	71	59	71	71	71	71	76	76	71	82	57
c-tkt-tkt	44	58	53	26	74	53		86	93	93	26	29	74	53	58	74	58	68	68	59	74	74	74	74	74	74	68	79	64
clh_spin	31	0	21	7	71	7	7		71	14	7	14	43	36	7	71	7	43	57	57	71	57	71	71	79	64	57	71	41
clh_stp	23	14	7	7	0	14	7	14		14	7	7	7	0	7	7	7	7	0	7	0	0	7	7	14	14	7	7	8
clh-ls	23	0	21	0	71	7	0	0	71		7	14	43	29	0	71	7	36	64	71	64	57	71	71	64	57	57	64	39
hmcs	44	63	63	37	63	65	32	86	93	93		57	68	53	68	74	68	68	63	65	74	68	74	74	74	74	68	84	67
hticket-ls	31	64	36	0	71	36	7	71	93	79	0		57	50	43	71	50	50	71	71	71	71	71	71	79	79	71	93	58
malth_spin	22	26	11	16	53	6	11	43	93	50	11	7		26	16	68	21	42	47	53	58	58	58	68	74	58	47	74	41
malth_stp	33	47	26	26	62	35	26	57	93	57	26	29	32		32	62	37	48	38	59	43	43	63	63	63	53	53	68	47
mcs_spin	28	26	26	21	74	18	21	64	93	50	21	21	53	42		68	32	53	68	71	68	68	68	68	74	63	42	68	51
mcs_stp	22	26	5	16	10	18	16	29	29	29	16	14	16	14	11		21	24	5	18	5	5	5	5	21	21	5	21	16
mcs-ls	22	21	26	26	68	18	16	57	93	50	16	29	53	37	21	68		42	58	59	68	63	68	63	63	58	53	74	48
mcs-timepub	28	26	21	21	67	18	16	43	93	43	16	21	42	29	11	62	26		57	59	57	62	68	68	63	63	37	58	43
mutexee	33	26	5	21	67	18	16	29	93	29	26	21	26	33	16	76	21	29		24	29	14	63	68	63	53	42	58	37
partitioned	31	24	18	18	71	29	24	21	93	21	29	7	29	29	12	71	18	24	47		47	47	71	71	65	53	53	76	41
pthread	28	26	21	16	57	18	16	29	93	29	21	14	32	29	16	76	21	24	14	24		19	79	79	58	37	37	47	35
pthreadadapt	33	26	11	16	62	18	16	29	93	29	21	14	32	19	16	76	21	24	29	35	33		74	74	58	58	37	53	37
spinlock	22	26	0	21	21	18	16	29	93	29	21	14	26	21	16	74	21	16	5	18	5	5		11	32	32	0	16	22
spinlock-ls	22	26	5	21	21	18	16	29	93	29	21	21	26	26	21	74	21	16	5	18	5	5	53		26	21	0	16	24
ticket	22	11	0	5	47	12	11	14	86	21	21	7	11	16	0	58	16	5	16	12	26	26	58	58		0	11	47	23
ticket-ls	28	26	0	16	58	12	16	21	86	29	21	7	16	16	16	63	26	21	21	29	26	26	58	58	74		42	58	32
ttas	33	26	11	21	74	18	21	29	93	29	26	21	37	26	16	74	26	32	37	29	32	42	74	74	53	37		32	38
ttas-ls	28	16	11	11	63	18	16	29	93	29	16	7	26	21	11	68	21	11	32	18	32	37	68	68	37	32	0		30
average	30	30	22	19	59	23	17	43	87	45	19	20	40	33	24	69	29	36	43	44	47	45	64	63	60	52	41	58	30

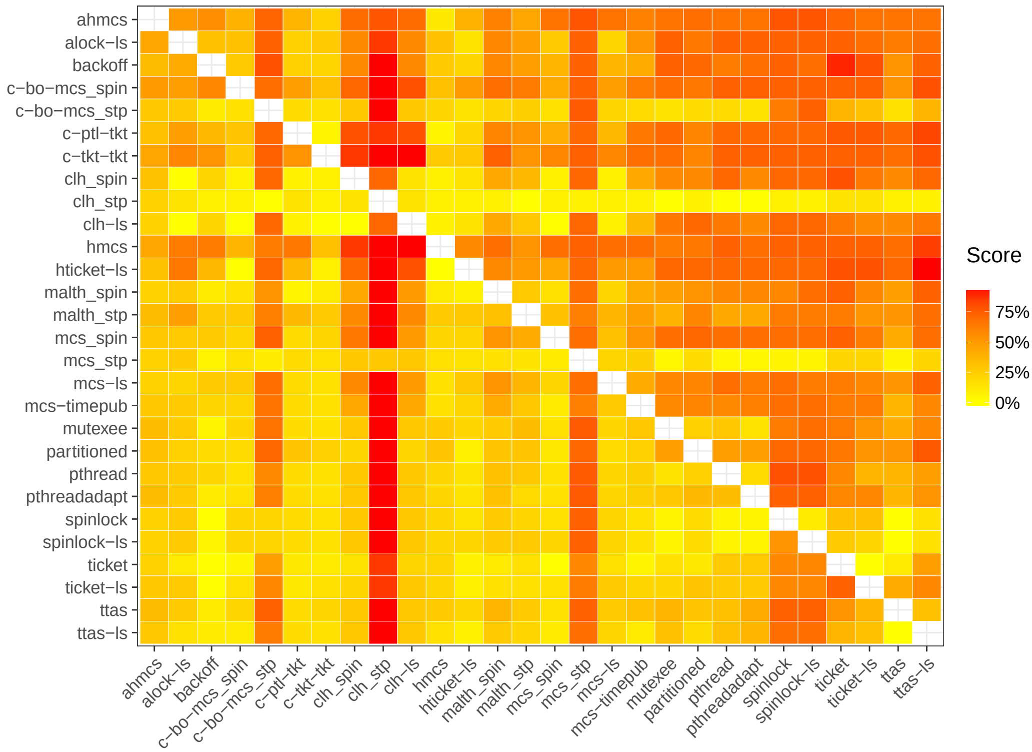

Fig. 23. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A is more energy-efficient at least 5% better than B (**I-48 machine in energy-saving move**).

Table 83. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A is more energy-efficient by at least 5% than B (**I-20 machine in energy-saving mode**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		27	40	7	67	7	13	27	82	27	20	0	27	27	27	80	33	33	53	29	53	53	60	33	33	27	20	40	35
alock-ls	27		38	25	75	20	12	25	83	33	25	0	31	25	12	75	12	25	50	27	56	50	56	38	31	19	31	44	35
backoff	33	38		25	69	20	25	50	92	42	25	17	31	12	31	69	31	19	44	40	50	50	56	38	56	44	25	38	40
c-bo-mcs_spin	27	38	50		75	20	12	58	92	50	12	17	31	38	38	75	31	31	50	27	62	50	56	50	50	38	38	56	43
c-bo-mcs_stp	27	25	19	19		20	12	25	92	25	19	8	31	12	25	71	31	24	12	20	12	12	31	25	25	31	19	31	26
c-ptl-tkt	14	33	47	0	60		13	50	92	58	0	0	47	33	20	73	27	33	53	27	60	47	53	47	47	40	47	60	40
c-tkt-tkt	27	25	44	12	75	27		50	92	33	19	0	38	38	19	75	25	31	56	33	69	50	56	56	56	31	31	50	41
clh_spin	9	8	25	0	75	0	0		75	0	0	0	17	8	0	75	0	8	42	17	42	42	50	33	17	8	17	25	22
clh_stp	18	17	8	8	0	8	8	17		17	8	8	8	0	8	17	8	8	0	8	0	0	8	8	8	8	8	8	8
clh-ls	9	8	33	8	67	0	0	0	75		0	0	17	8	0	75	0	8	42	25	42	42	50	33	17	8	17	25	23
hmcs	27	31	44	6	75	13	19	50	92	50		0	44	38	25	75	25	31	50	27	62	50	50	50	50	44	38	56	42
hticket-ls	18	50	42	8	75	8	8	50	92	50	8		50	25	8	75	25	33	50	42	50	42	50	33	50	33	33	58	40
malth_spin	20	19	25	6	62	13	6	33	92	25	19	0		6	6	62	6	6	56	27	44	50	56	25	38	31	12	38	29
malth_stp	40	38	31	31	71	33	25	50	92	50	31	25	31		25	65	31	29	41	40	47	53	56	31	56	50	31	44	43
mcs_spin	13	31	44	12	62	13	6	33	92	42	19	0	44	31		62	6	38	56	40	56	50	56	31	38	38	19	44	36
mcs_stp	20	25	19	19	6	20	12	25	25	25	19	8	25	6	12		19	12	18	20	18	12	19	6	31	31	6	19	18
mcs-ls	13	25	38	6	62	13	6	33	92	42	19	0	31	38	0	62		19	56	33	56	50	56	31	38	25	19	44	34
mcs-timepub	27	31	38	19	59	20	12	42	92	33	25	8	38	24	12	59	19		53	40	53	47	62	31	44	38	25	38	37
mutexee	20	25	6	19	71	20	19	25	92	25	19	17	19	12	19	65	19	18		20	29	41	38	25	25	19	6	25	27
partitioned	21	13	33	33	73	27	13	17	92	17	20	0	20	20	13	73	20	13	40		53	47	53	47	33	20	20	47	33
pthread	20	25	6	19	65	20	19	25	92	25	19	17	19	6	19	65	19	18	12	20		29	50	19	31	31	6	19	26
pthreadadapt	20	25	6	19	65	20	19	25	92	25	19	17	19	6	19	71	19	18	12	20	18		31	19	19	19	6	19	25
spinlock	20	25	6	19	50	20	19	25	92	25	19	17	19	12	19	69	19	6	31	20	25	25		0	19	19	0	12	23
spinlock-ls	20	25	25	19	62	20	19	25	92	25	25	17	31	25	31	81	31	19	38	20	44	31	56		31	31	0	12	32
ticket	7	19	19	12	56	13	6	17	92	17	19	0	12	6	0	62	6	12	44	13	38	44	50	31		6	0	19	23
ticket-ls	20	25	19	12	62	13	12	42	92	33	19	8	12	12	6	62	6	12	50	40	50	50	44	38	38		25	38	31
ttas	20	25	31	19	62	20	19	25	92	25	25	17	31	25	31	81	31	25	44	27	56	50	56	25	44	31		25	36
ttas-ls	20	19	25	12	56	13	12	25	92	25	19	8	31	19	25	75	25	12	50	20	50	44	56	25	38	25	12		31
average	21	26	28	15	61	16	13	32	87	31	17	8	28	19	17	69	19	20	41	27	44	41	49	31	36	28	19	35	21

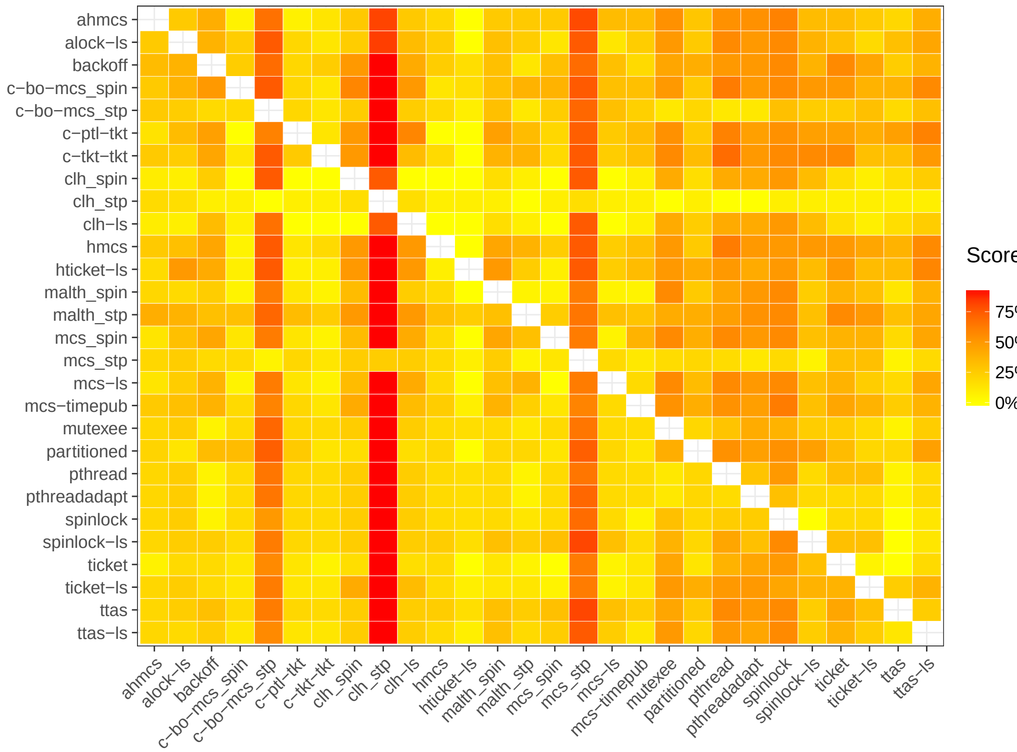

Fig. 24. For each pair of locks *(rowA, colB)* at *max nodes*, scores of lock A vs lock B: percentage of lock-sensitive applications for which lock A is more energy-efficient at least 5% better than B (**I-20 machine in energy-saving move**).

B.5 Are all locks potentially harmful?

Table 84. For each lock-sensitive application, at *max nodes*, energy efficiency gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the energy efficiency gain is greater than 15%. A line with many gray cells corresponds to an application whose energy efficiency is hurt by many locks. A column with many gray cells corresponds to a lock that has lower energy-efficiency than many other locks. Dashes correspond to untested cases. (**I-48 machine in energy-saving mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	733	5	120	87	666	452	2k	603	3k	103	161	81	74	77	75	109	95	8	156	0	11	12	9	2k	2k	1	389
ferret	539	486	13	501	0	531	499	465	0	485	521	569	557	0	480	0	490	10	0	482	0	0	12	11	479	339	11	12
kyotocabinet	14	36	33	0	119	32	17	33	967	33	18	17	52	14	35	952	33	68	98	56	114	86	363	285	66	43	58	64
linear_regression	15	2	0	3	116	2	1	4	171	10	3	0	3	11	10	192	49	14	7	23	10	9	64	50	15	15	8	10
memcached-new	73	39	8	32	63	-	46	-	-	-	64	-	22	0	33	629	43	39	49	-	73	57	183	98	33	23	39	40
memcached-old	6	0	87	30	27	-	10	-	-	-	38	-	170	169	92	88	173	151	161	-	190	186	88	78	163	159	85	85
mysqld	-	-	-	-	0	-	-	-	-	-	-	-	-	25	-	46	-	87	28	-	28	30	-	-	-	-	-	-
pca	11	18	14	0	97	16	7	24	777	19	7	17	12	11	17	741	16	18	43	30	53	42	312	248	66	51	58	58
pca_ll	2	9	42	4	254	0	1	14	1k	13	1	1	17	123	8	1k	8	23	68	28	79	124	487	349	70	39	90	84
radiosity	15	17	14	3	50	9	6	27	576	24	0	6	13	16	9	542	12	16	31	31	37	28	233	162	46	36	52	59
radiosity_ll	0	58	94	7	527	48	20	80	4k	80	0	24	117	75	53	4k	64	94	181	158	237	144	2k	1k	327	223	387	379
s_raytrace	5	17	11	23	387	6	8	25	2k	28	0	21	43	100	16	2k	10	15	57	63	60	91	638	607	146	97	157	221
s_raytrace_ll	1	8	35	18	54	9	15	23	2k	22	0	33	78	70	7	2k	8	24	46	42	48	53	634	627	139	73	138	215
sqlite	-	-	-	-	147	-	-	-	-	-	-	-	-	0	-	2k	-	6k	87	-	94	61	-	-	-	-	-	-
ssl_proxy	8	51	22	0	336	44	24	54	2k	54	5	24	62	66	60	2k	49	78	127	101	134	90	1k	692	183	130	185	180
streamcluster	146	20	325	245	348	143	25	-	-	-	63	-	340	520	251	403	224	274	351	0	288	300	357	398	417	385	245	244
streamcluster_ll	167	4	349	284	388	141	10	-	-	-	66	-	373	613	277	465	249	281	394	0	309	334	390	417	461	437	273	277
upscaledb	19	43	48	13	127	24	18	45	767	44	19	18	34	0	47	766	42	73	158	57	163	170	371	246	59	48	48	83
vips	89	136	0	636	48	283	184	-	-	-	188	-	373	1	107	4	142	3	0	153	0	0	3	3	131	14	0	1
water_nsquared	66	24	2	1	1	1	1	45	44	46	1	2	1	1	1	0	1	4	0	0	1	1	0	0	2	0	0	23
water_spatial	588	209	0	5	5	1	2	400	399	396	7	5	5	5	4	4	4	6	0	1	0	0	0	0	1	0	0	204

Table 85. For each lock-sensitive application, at *opt nodes*, energy efficiency gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the energy efficiency gain is greater than 15%. A line with many gray cells corresponds to an application whose energy efficiency is hurt by many locks. A column with many gray cells corresponds to a lock that has lower energy-efficiency than many other locks. Dashes correspond to untested cases. (**I-48 machine in energy-efficiency mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	386	1	64	71	12	3	618	609	607	89	50	55	59	56	53	54	77	0	2	2	0	1	1	2	4	0	331
ferret	18	17	6	20	0	27	18	18	0	18	17	19	27	0	17	0	18	6	0	17	0	0	7	7	25	18	5	6
kyotocabinet	8	27	4	0	17	15	9	22	292	27	15	11	30	36	24	277	22	39	92	33	87	70	109	85	41	29	29	30
linear_regression	0	3	0	1	27	1	1	3	86	4	2	0	4	7	2	85	4	14	8	7	7	4	48	35	15	8	7	9
memcached-new	52	49	4	12	22	-	13	-	-	-	87	-	38	9	62	52	65	46	20	-	10	17	11	1	0	15	38	36
memcached-old	0	11	25	11	9	-	10	-	-	-	15	-	60	58	30	32	46	58	58	-	65	67	33	5	66	68	25	36
mysqld	-	-	-	-	0	-	-	-	-	-	-	-	-	23	-	35	-	87	24	-	23	23	-	-	-	-	-	-
pca	1	1	15	1	21	2	1	1	203	2	0	2	0	2	1	204	0	4	16	1	22	8	5	2	5	2	1	2
pca_ll	1	1	27	5	51	0	0	5	48	3	0	0	7	13	1	47	1	4	30	8	38	42	46	24	10	8	8	8
radiosity	15	8	5	0	5	0	0	15	39	15	0	1	6	10	2	38	3	4	10	3	15	10	22	16	15	10	11	15
radiosity_ll	0	21	33	5	10	3	7	29	223	29	0	10	46	61	20	155	21	31	66	27	114	78	125	85	117	63	66	61
s_raytrace	2	2	6	14	69	0	2	8	81	7	0	6	25	63	0	77	1	6	32	19	33	42	77	76	50	25	29	39
s_raytrace_ll	2	18	29	21	69	5	11	34	186	34	0	30	85	86	14	189	18	36	60	55	62	67	188	187	161	90	159	186
sqlite	-	-	-	-	0	-	-	-	-	-	-	-	-	43	-	395	-	37	53	-	71	58	-	-	-	-	-	-
ssl_proxy	0	18	5	1	31	12	9	36	803	29	0	11	48	91	24	794	17	37	76	33	96	87	322	137	134	77	83	75
streamcluster	5	0	22	1	4	1	0	-	-	-	1	-	30	34	12	15	33	28	21	5	12	32	21	14	42	39	20	17
streamcluster_ll	6	0	72	10	22	19	13	-	-	-	21	-	93	94	54	56	76	54	79	11	49	79	62	48	97	92	45	48
upscaledb	0	17	20	1	75	1	0	20	473	21	0	1	15	21	21	586	18	29	108	24	99	95	70	60	25	20	28	38
vips	8	8	0	8	9	7	7	-	-	-	8	-	7	1	8	4	8	3	0	9	0	0	3	3	8	8	0	1
water_nsquared	66	24	2	1	1	1	1	45	44	46	1	2	1	1	1	0	1	4	0	0	1	1	0	0	2	0	0	23
water_spatial	588	209	0	5	5	1	2	400	399	396	7	5	5	5	4	4	4	6	0	1	0	0	0	0	1	0	0	204

Table 86. For each lock-sensitive application, at *max nodes*, energy efficiency gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the energy efficiency gain is greater than 15%. A line with many gray cells corresponds to an application whose energy efficiency is hurt by many locks. A column with many gray cells corresponds to a lock that has lower energy-efficiency than many other locks. Dashes correspond to untested cases (**I-20 machine in energy-saving mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	1k	3	92	59	101	50	636	415	617	99	53	85	51	52	55	89	62	1	78	0	0	2	1	66	118	2	252
ferret	305	292	22	313	0	303	307	296	0	291	302	300	324	0	294	0	293	11	0	292	0	0	18	18	295	254	18	17
kyotocabinet	11	19	12	6	274	14	24	22	2k	27	17	13	36	0	19	2k	19	29	64	38	76	47	118	77	46	25	39	27
linear_regression	3	2	0	0	13	2	0	6	100	4	0	2	4	2	5	101	3	3	3	3	6	3	12	5	5	4	2	3
memcached-old	5	7	0	7	204	-	4	-	-	-	16	-	6	30	4	92	7	9	134	-	131	47	22	10	4	3	5	7
pca	3	3	16	2	75	0	0	3	1k	5	0	2	4	8	3	1k	2	4	17	2	25	41	37	7	8	2	5	11
pca_ll	1	2	8	2	246	0	1	1	709	1	2	1	2	3	2	1k	2	2	7	1	14	3	0	0	0	0	1	1
radiosity	18	11	8	1	24	0	1	18	406	21	1	1	10	12	2	306	4	4	16	15	22	28	86	33	19	8	19	22
radiosity_ll	4	13	31	0	212	3	3	26	4k	23	2	6	29	34	7	4k	11	18	63	57	89	114	438	154	88	41	69	54
s_raytrace	1	1	1	3	84	0	0	2	2k	1	1	1	2	4	1	2k	1	1	3	1	6	4	1	0	0	0	0	1
s_raytrace_ll	2	2	4	4	114	3	2	2	3k	2	2	2	3	2	2	3k	2	3	5	2	5	1	0	1	2	2	1	2
sqlite	-	-	-	-	312	-	-	-	-	-	-	-	-	24	-	2k	-	5k	0	-	41	43	-	-	-	-	-	-
streamcluster	22	0	58	6	16	18	12	-	-	-	5	-	46	47	39	43	39	46	83	1	53	118	78	27	67	75	28	29
streamcluster_ll	33	0	71	11	27	28	11	-	-	-	12	-	64	70	53	54	58	58	118	5	72	152	95	43	93	95	36	41
upscaledb	21	30	46	22	399	19	26	34	1k	30	18	20	26	0	30	1k	29	40	176	38	183	192	73	56	32	25	31	45
vips	34	47	2	65	0	58	48	-	-	-	56	-	22	0	36	0	35	1	0	43	0	0	2	1	31	32	1	1
water_spatial	296	104	0	3	2	0	0	198	197	197	3	2	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	101

Table 87. For each lock-sensitive application, at *opt nodes*, energy efficiency gain, (in %) obtained by the best lock(s) with respect to each of the other locks. A gray cell highlights a configuration where a given lock hurts the application, i.e., the energy efficiency gain is greater than 15%. A line with many gray cells corresponds to an application whose energy efficiency is hurt by many locks. A column with many gray cells corresponds to a lock that has lower energy-efficiency than many other locks. Dashes correspond to untested cases. (**I-20 machine in energy-efficiency mode**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	c-ptl-tkt	c-tkt-tkt	clh_spin	clh_stp	clh-ls	hmcs	hticket-ls	malth_spin	malth_stp	mcs_spin	mcs_stp	mcs-ls	mcs-timepub	mutexee	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	340	1	53	53	8	4	480	481	488	69	45	48	47	50	48	48	59	0	4	0	2	3	2	0	3	1	282
ferret	50	46	22	46	0	44	48	48	0	44	45	44	75	0	46	0	46	11	0	44	0	0	18	18	44	45	18	17
kyotocabinet	3	13	1	0	370	4	6	13	747	16	7	3	21	16	10	745	10	21	93	23	84	29	53	33	26	16	22	17
linear_regression	3	2	0	0	13	2	0	6	24	4	0	2	4	2	5	23	3	3	3	3	6	3	12	5	5	4	2	3
memcached-old	5	7	0	7	204	-	4	-	-	-	16	-	6	30	4	92	7	9	134	-	131	47	22	10	4	3	5	7
pca	3	3	16	2	17	0	0	3	16	5	0	2	4	8	3	14	2	4	12	2	10	11	11	7	8	2	5	11
pca_ll	1	2	8	2	36	0	1	1	37	1	2	1	2	3	2	34	2	2	7	1	14	3	0	0	0	0	1	1
radiosity	18	10	6	1	5	0	1	16	19	16	1	1	5	6	2	5	4	4	3	4	5	4	6	5	6	4	4	10
radiosity_ll	4	13	21	0	21	3	3	18	29	19	2	6	19	22	7	18	11	18	28	19	31	27	20	17	19	17	17	17
s_raytrace	1	1	1	3	44	0	0	2	47	1	1	1	2	4	1	43	1	1	3	1	6	4	1	0	0	0	0	1
s_raytrace_ll	2	2	4	4	31	3	2	2	32	2	2	2	3	2	2	31	2	3	5	2	5	1	0	1	2	2	1	2
sqlite	-	-	-	-	3	-	-	-	-	-	-	-	-	30	-	363	-	0	19	-	26	21	-	-	-	-	-	-
streamcluster	5	7	8	14	13	20	7	-	-	-	8	-	13	8	9	10	11	11	24	0	6	23	24	8	23	18	4	8
streamcluster_ll	33	0	70	11	27	28	11	-	-	-	12	-	64	64	53	54	58	58	74	5	44	74	72	43	71	78	36	41
upscaledb	3	10	25	0	305	3	3	11	567	10	3	3	10	2	12	660	11	14	182	14	189	195	22	18	13	10	14	19
vips	34	47	2	48	0	48	48	-	-	-	48	-	22	0	36	0	35	1	0	43	0	0	2	1	31	32	1	1
water_spatial	296	104	0	3	2	0	0	198	197	197	3	2	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	101

B.6 Impact of the number of nodes.

Table 88. For each lock-sensitive application, percentage of pairwise changes in the lock energy-efficiency hierarchy when changing the number of nodes (**I-48 machine in energy-saving mode**).

	% of pairwise changes between configurations			
Applications	1/2	2/3	3/4	1/2/3/4
dedup	7%	21%	19%	41%
ferret	19%	66%	8%	84%
kyotocabinet	16%	5%	5%	22%
linear_regression	26%	24%	38%	72%
memcached-new	59%	29%	0%	70%
memcached-old	14%	14%	0%	23%
mysqld	5%	0%	0%	5%
pca	49%	13%	13%	62%
pca_ll	47%	31%	15%	85%
radiosity	24%	14%	10%	43%
radiosity_ll	25%	7%	10%	33%
s_raytrace	69%	19%	12%	95%
s_raytrace_ll	84%	17%	10%	97%
sqlite	19%	33%	19%	57%
ssl_proxy	15%	6%	6%	21%
streamcluster	22%	22%	28%	48%
streamcluster_ll	20%	21%	25%	42%
upscaledb	12%	7%	3%	17%
vips	0%	0%	78%	78%
water_nsquared	0%	0%	0%	0%
water_spatial	3%	4%	8%	12%

Table 89. For each lock-sensitive application, percentage of pairwise changes in the lock energy-efficiency hierarchy when changing the number of nodes (**I-20 machine in energy-saving mode**).

	% of pairwise changes between configurations
Applications	1/2
dedup	29%
ferret	17%
kyotocabinet	15%
linear_regression	17%
memcached-old	0%
pca	55%
pca_ll	32%
radiosity	63%
radiosity_ll	69%
s_raytrace	22%
s_raytrace_ll	21%
sqlite	62%
streamcluster	50%
streamcluster_ll	39%
upscaledb	17%
vips	70%
water_spatial	0%

B.7 Impact of the machine.

Table 90. Considering energy efficiency and performance, at *max nodes* and *opt nodes*, percentage of pairwise changes in the lock performance hierarchy.

# nodes	I-48 energy efficiency vs. performance	I-20 energy efficiency vs. performance
Max	12%	10%
Opt	14%	12%

C POLY

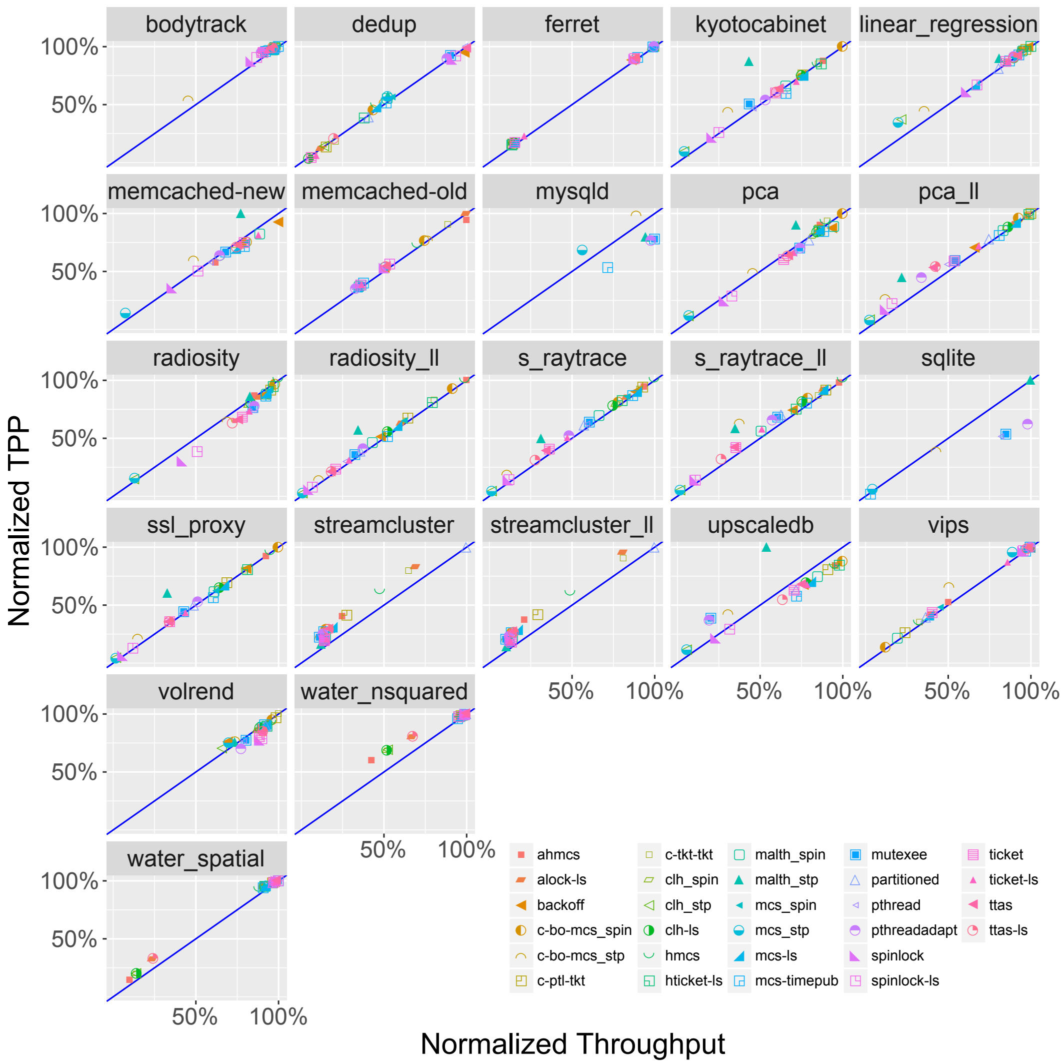

Fig. 25. Correlation of throughput with energy efficiency (TPP) on various lock-sensitive applications at *max nodes* for the different lock algorithms (**I-48 machine**).

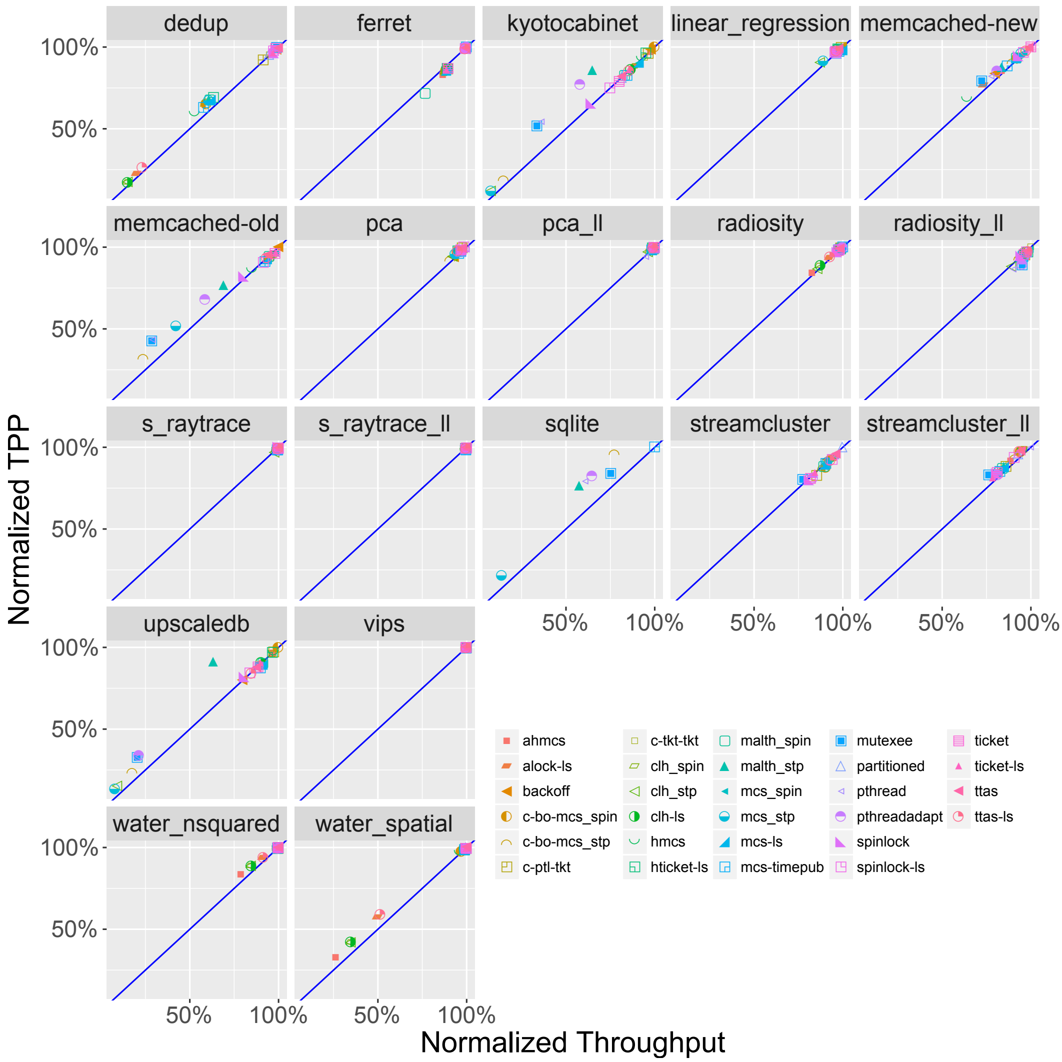

Fig. 26. Correlation of performance (throughput) with energy efficiency (TPP) on various lock-sensitive applications at *one node* for the different lock algorithms (**I-20 machine**).

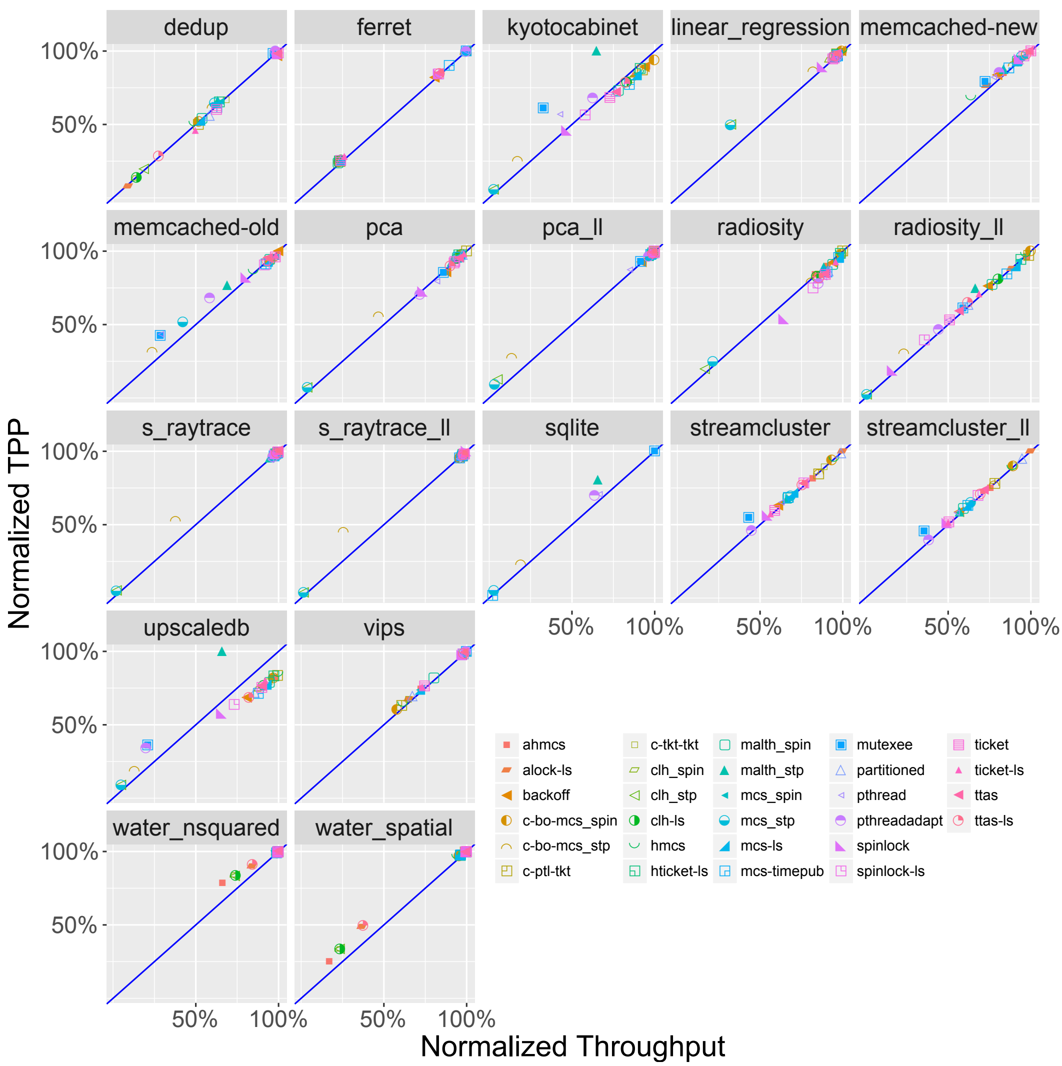

Fig. 27. Correlation of performance (throughput) with energy efficiency (TPP) on various lock-sensitive applications at *max nodes* for the different lock algorithms (**I-20 machine**).

D STUDY OF LOCK TAIL LATENCY

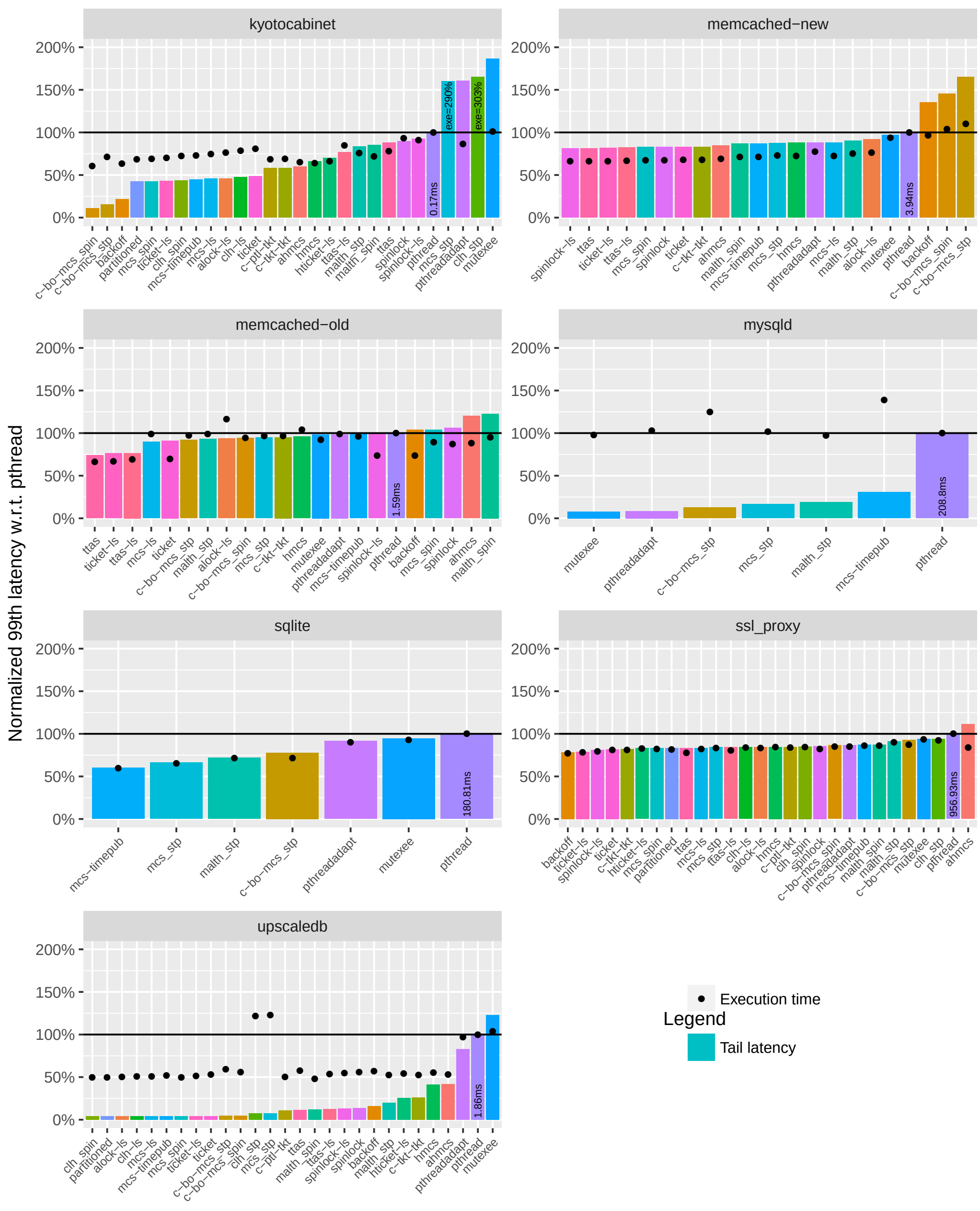

Fig. 28. For each server application, the bars represent the normalized 99th tail latency (w.r.t. Pthread) and the dots the execution time (lower is better) normalized (w.r.t. Pthread) of each lock algorithm (**A-64 at *one node***).

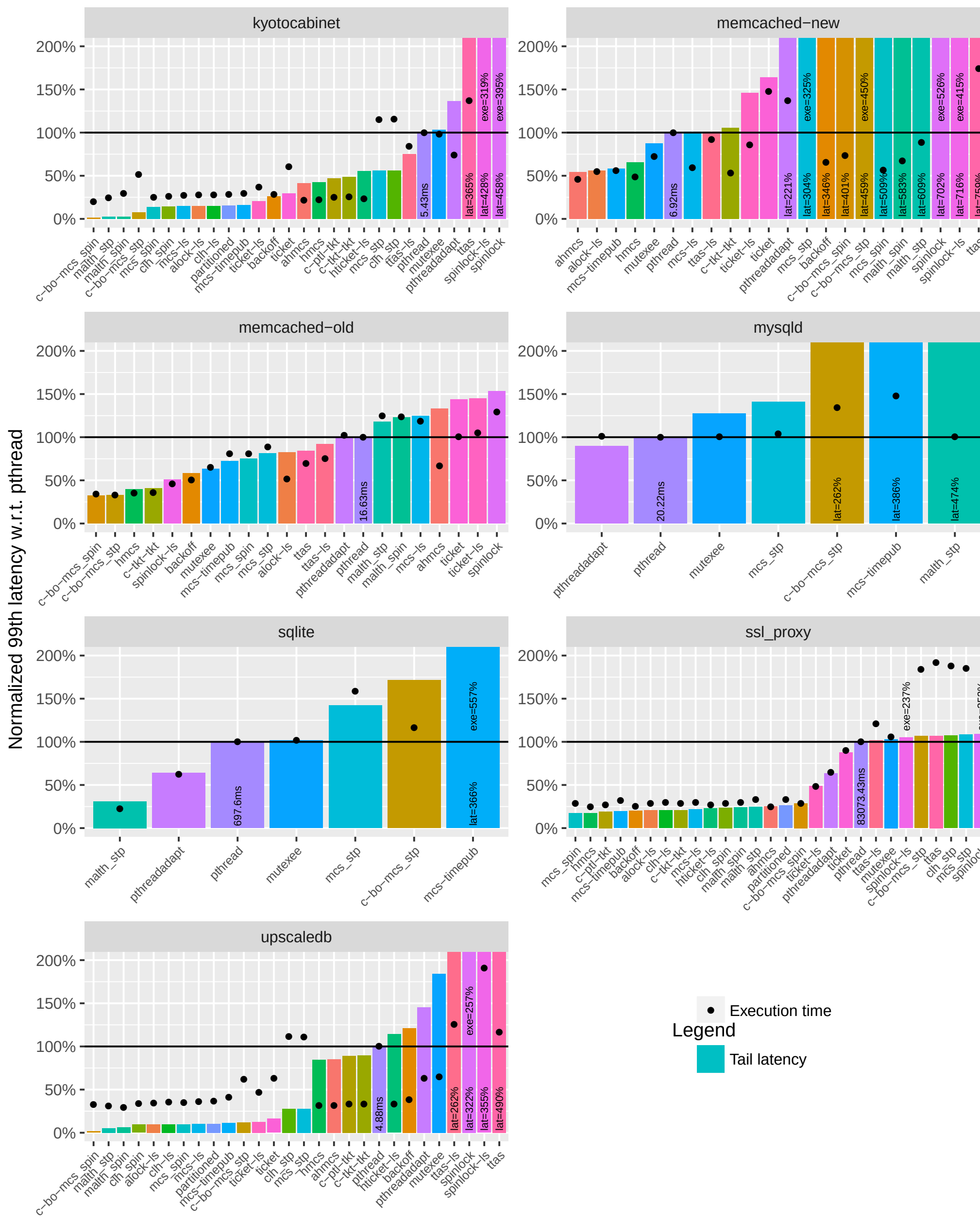

Fig. 29. For each server application, the bars represent the normalized 99th tail latency (w.r.t. Pthread) and the dots the execution time (lower is better) normalized (w.r.t. Pthread) of each lock algorithm (**A-64 at *max nodes***).

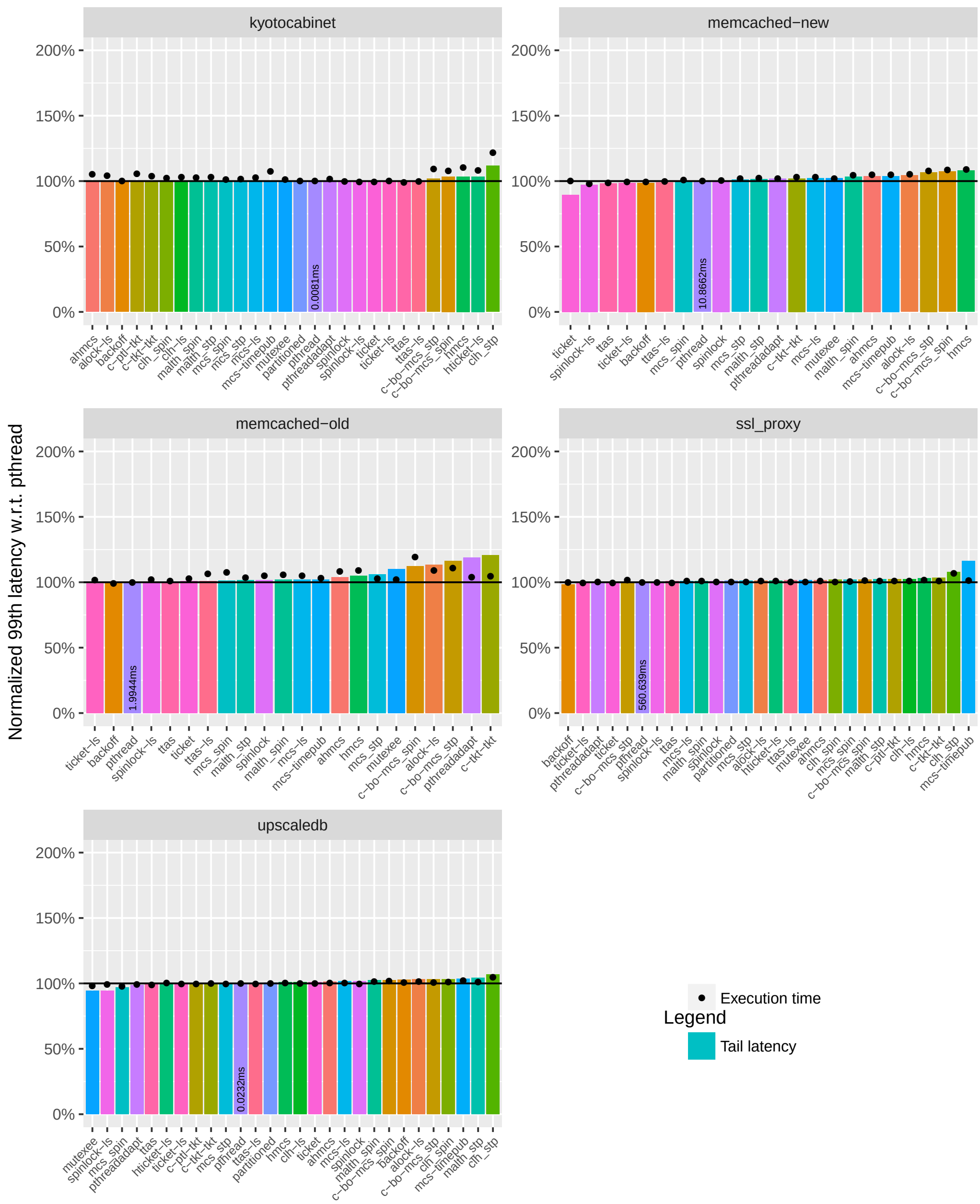

Fig. 30. For each server application, the bars represent the normalized 99th tail latency (w.r.t. Pthread) and the dots the execution time (lower is better) normalized (w.r.t. Pthread) of each lock algorithm (**A-64 *single threaded***).

ACKNOWLEDGMENTS

We thank the anonymous referees, the associate editor and the editor-in-chief, Todd C. Mowry, for their insightful comments on earlier drafts of this paper. We are also grateful to Tim Harris for his feedback on an earlier version of this work. Dave Dice provided detailed answers for our questions on Malthusian locks. Baptiste Lepers provided valuable insights for some of the case studies. Pierre Neyron provided his help to set up experiments on the I-48 machine and Fabien Salvi on the I-20 machine. Finally, this work has been partially supported by: LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01), the EmSoc "Replicanos" and AGIR "CAEC" projects of Université Grenoble-Alpes and GrenobleINP, the FSN OCCIiware (https://www.occiware.org/bin/view/Main/) project, the FUI/FEDER "Studio virtuel" project, the ANR "RainbowFS" (http://rainbowfs.lip6.fr/) project, and the INRIA/LIG "Digitalis" (http://digitalis.inria.fr/index.php/Digitalis) project.

REFERENCES

[1] Yehuda Afek, Alexander Matveev, Oscar R. Moll, and Nir Shavit. 2015. Amalgamated Lock-Elision. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings (Lecture Notes in Computer Science)*, Yoram Moses (Ed.), Vol. 9363. Springer, Berlin, Heidelberg, 309–324. https://doi.org/10.1007/978-3-662-48653-5_21

[2] Anant Agarwal and Mathews Cherian. 1989. Adaptive Backoff Synchronization Techniques. In *Proceedings of the 16th Annual International Symposium on Computer Architecture. Jerusalem, Israel, June 1989*, Jean-Claude Syre (Ed.). ACM, New York, NY, USA, 396–406. https://doi.org/10.1145/74925.74970

[3] Martin Aigner, Christoph M. Kirsch, Michael Lippautz, and Ana Sokolova. 2015. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, New York, NY, USA, 451–469. https://doi.org/10.1145/2814270.2814294

[4] Samy Al Bahra. 2015. Concurrency Kit. Retrieved November 8, 2018 from http://concurrencykit.org/

[5] Mohammad Mejbah Ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). ACM, New York, NY, USA, 298–313. https://doi.org/10.1145/3064176.3064186

[6] AMD. 2010. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors. Retrieved November 8, 2018 from http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf

[7] Nikos Anastopoulos and Nectarios Koziris. 2008. Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*. IEEE, Washington, DC, USA, 1–8. https://doi.org/10.1109/IPDPS.2008.4536358

[8] Thomas E. Anderson. 1990. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Trans. Parallel Distrib. Syst.* 1, 1 (1990), 6–16. https://doi.org/10.1109/71.80120

[9] Jelena Antic, Georgios Chatzopoulos, Rachid Guerraoui, and Vasileios Trigonakis. 2016. Locking Made Easy. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*. ACM, New York, NY, USA, 20. http://dl.acm.org/citation.cfm?id=2988357

[10] Marc Auslander, David Edelsohn, Orran Krieger, Bryan Rosenburg, and Robert Wisniewski. 2003. Enhancement to the MCS Lock for Increased Functionality and Improved Programmability. U.S. Patent Application Number 20030200457 (abandoned).

[11] Woongki Baek and Trishul M. Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, New York, NY, USA, 198–209. https://doi.org/10.1145/1806596.1806620

[12] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000.*, Larry Rudolph and Anoop Gupta (Eds.). ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/356989.357000

[13] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton University.

[14] Mike W. Blasgen, Jim Gray, Michael F. Mitoma, and Thomas G. Price. 1979. The Convoy Phenomenon. *Operating Systems Review* 13, 2 (1979), 20–25. https://doi.org/10.1145/850657.850659

[15] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2012. Non-scalable Locks are Dangerous. In *Proceedings of the Linux Symposium*. Linux, Ottawa, Canada, 119–130.

[16] Brad Fitzpatrick. 2018. Memcached. Retrieved November 8, 2018 from http://memcached.org

[17] Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. 2016. Investigating the Performance of Hardware Transactions on a Multi-Socket Machine. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, Christian Scheideler and Seth Gilbert (Eds.). ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/2935764.2935796

[18] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra J. Marathe, and Mark Moir. 2013. Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores. In *Principles of Distributed Systems - 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings (Lecture Notes in Computer Science)*, Roberto Baldoni, Nicolas Nisse, and Maarten van Steen (Eds.), Vol. 8304. Springer, Berlin, Heidelberg, 83–97. https://doi.org/10.1007/978-3-319-03850-6_7

[19] Irina Calciu, David Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. 2013. NUMA-aware reader-writer locks. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc (Eds.). ACM, New York, NY, USA, 157–166. https://doi.org/10.1145/2442516.2442532

[20] Milind Chabbi, Michael W. Fagan, and John M. Mellor-Crummey. 2015. High performance locks for multi-level NUMA systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, Albert Cohen and David Grove (Eds.). ACM, New York, NY, USA, 215–226. https://doi.org/10.1145/2688500.2688503

[21] Milind Chabbi and John M. Mellor-Crummey. 2016. Contention-conscious, locality-preserving locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, Rafael Asenjo and Tim Harris (Eds.). ACM, New York, NY, USA, 22:1–22:14. https://doi.org/10.1145/2851141.2851166

[22] Christoph Rupp. 2018. Upscaledb. Retrieved November 8, 2018 from https://upscaledb.com

[23] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2013. RadixVM: scalable address spaces for multithreaded applications. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek (Eds.). ACM, New York, NY, USA, 211–224. https://doi.org/10.1145/2465351.2465373

[24] Travis S. Craig. 1993. *Building FIFO and Priority-Queuing Spin Locks from Atomic Swap*. Technical Report TR 93-02-02. University of Washington.

[25] Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2014. Continuously measuring critical section pressure with the free-lunch profiler. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, New York, NY, USA, 291–307. https://doi.org/10.1145/2660193.2660210

[26] Tudor David and Rachid Guerraoui. 2016. Concurrent Search Data Structures Can Be Blocking and Practically Wait-Free. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, Christian Scheideler and Seth Gilbert (Eds.). ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/2935764.2935774

[27] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, New York, NY, USA, 33–48. https://doi.org/10.1145/2517349.2522714

[28] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, New York, NY, USA, 631–644. https://doi.org/10.1145/2694344.2694359

[29] David Dice. 2011. Brief announcement: a partitioned ticket lock. In *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, Rajmohan Rajaraman and Friedhelm Meyer auf der Heide (Eds.). ACM, New York, NY, USA, 309–310. https://doi.org/10.1145/1989493.1989543

[30] Dave Dice. 2017. Malthusian Locks. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). ACM, New York, NY, USA, 314–327. https://doi.org/10.1145/3064176.3064203

[31] David Dice, Virendra J. Marathe, and Nir Shavit. 2011. Flat-combining NUMA locks. In *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, Rajmohan Rajaraman and Friedhelm Meyer auf der Heide (Eds.). ACM, New York, NY,

USA, 65–74. https://doi.org/10.1145/1989493.1989502
[32] David Dice, Virendra J. Marathe, and Nir Shavit. 2015. Lock Cohorting: A General Technique for Designing NUMA Locks. *TOPC* 1, 2 (2015), 13:1–13:42. https://doi.org/10.1145/2686884
[33] Dave Dice, Mark S. Moir, and III William N. Scherer. 2003. Quickly Reacquirable Locks. Patent No. US7814488B1, Filed September 9th., 2002, Issued September 3rd., 2003.
[34] Open Source Facebook. 2017. Rocksdb. Retrieved November 8, 2018 from http://rocksdb.org
[35] FAL Labs. 2012. Kyoto Cabinet. Retrieved November 8, 2018 from http://fallabs.com/kyotocabinet
[36] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. 2016. Unlocking Energy. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, Ajay Gulati and Hakim Weatherspoon (Eds.). USENIX Association, Denver, CO, 393–406. https://www.usenix.org/conference/atc16/technical-sessions/presentation/falsafi
[37] Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, J. Ramanujam and P. Sadayappan (Eds.). ACM, New York, NY, USA, 257–266. https://doi.org/10.1145/2145816.2145849
[38] Rich Felker. 2018. musl libc. Retrieved November 8, 2018 from https://www.musl-libc.org
[39] Free Software Foundation FSF. 2018. The GNU C Library. Retrieved November 8, 2018 from https://www.gnu.org/software/libc/manual
[40] Free Software Foundation FSF. 2018. pthread_mutex_lock GNU C library implementation. Retrieved November 8, 2018 from https://sourceware.org/git/?p=glibc.git;a=blob;f=nptl/pthread_mutex_lock.c;hb=HEAD
[41] Sanjay Ghemawat and Paul Menage. 2018. TCMalloc: Thread-Caching Malloc. Retrieved November 8, 2018 from https://github.com/gperftools/gperftools
[42] Vincent Gramoli. 2015. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, Albert Cohen and David Grove (Eds.). ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2688500.2688501
[43] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. 2016. Multicore Locks: The Case Is Not Closed Yet. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, Ajay Gulati and Hakim Weatherspoon (Eds.). USENIX Association, Denver, CO, 649–662. https://www.usenix.org/conference/atc16/technical-sessions/presentation/guiroux
[44] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. 2018. LiTL source code and data sets. Retrieved November 8, 2018 from https://github.com/multicore-locks
[45] Pat Hanrahan, David Salzman, and Larry Aupperle. 1991. A rapid hierarchical radiosity algorithm. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1991, Providence, RI, USA, April 27-30, 1991*, James J. Thomas (Ed.). ACM, New York, NY, USA, 197–206. https://doi.org/10.1145/122718.122740
[46] Bijun He, William N. Scherer III, and Michael L. Scott. 2005. Preemption Adaptivity in Time-Published Queue-Based Spin Locks. In *High Performance Computing - HiPC 2005, 12th International Conference, Goa, India, December 18-21, 2005, Proceedings (Lecture Notes in Computer Science)*, David A. Bader, Manish Parashar, Sridhar Varadarajan, and Viktor K. Prasanna (Eds.), Vol. 3769. Springer, Berlin, Heidelberg, 7–18. https://doi.org/10.1007/11602569_6
[47] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, Friedhelm Meyer auf der Heide and Cynthia A. Phillips (Eds.). ACM, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540
[48] IEEE. 2013. pthread_mutex_lock(3p) man page. Retrieved November 8, 2018 from http://man7.org/linux/man-pages/man3/pthread_mutex_lock.3p.html
[49] IEEE. 2017. mallopt(3) man page. Retrieved November 8, 2018 from http://man7.org/linux/man-pages/man3/mallopt.3.html
[50] Intel. 2015. Intel Xeon Processor E7-4800/8800 v3 Product Families. Retrieved November 8, 2018 from http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e7-v3-datasheet-vol-1.pdf
[51] Intel. 2016. Intel 64 and IA-32 Architectures, Software Developer's Manual, Volume 3B: System Programming Guide, Part 2. Retrieved November 8, 2018 from https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf
[52] Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mowry. 2010. Decoupling contention management from scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, James C. Hoe and Vikram S. Adve (Eds.). ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/1736020.1736035

[53] Alain Kägi, Doug Burger, and James R. Goodman. 1997. Efficient Synchronization: Let Them Eat QOLB. In *Proceedings of the 24th International Symposium on Computer Architecture, Denver, Colorado, USA, June 2-4, 1997*, Andrew R. Pleszkun and Trevor N. Mudge (Eds.). ACM, New York, NY, USA, 170–180. https://doi.org/10.1145/264107.264166

[54] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan S. Owicki. 1991. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles, SOSP 1991, Asilomar Conference Center, Pacific Grove, California, USA, October 13-16, 1991*, Henry M. Levy (Ed.). ACM, New York, NY, USA, 41–55. https://doi.org/10.1145/121132.286599

[55] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Scalable NUMA-aware Blocking Synchronization Primitives. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. USENIX Association, Santa Clara, CA, 603–615. https://www.usenix.org/conference/atc17/technical-sessions/presentation/kashyap

[56] Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. 1997. Scheduler-Conscious Synchronization. *ACM Trans. Comput. Syst.* 15, 1 (1997), 3–40. https://doi.org/10.1145/244764.244765

[57] Konstantinos Koukos, David Black-Schaffer, Vasileios Spiliopoulos, and Stefanos Kaxiras. 2013. Towards more efficient execution: a decoupled access-execute approach. In *International Conference on Supercomputing, ICS'13, Eugene, OR, USA - June 10 - 14, 2013*, Allen D. Malony, Mario Nemirovsky, and Samuel P. Midkiff (Eds.). ACM, New York, NY, USA, 253–262. https://doi.org/10.1145/2464996.2465012

[58] Bradley C. Kuszmaul. 2015. SuperMalloc: a super fast multithreaded malloc for 64-bit machines. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management, ISMM 2015, Portland, OR, USA, June 13-14, 2015*, Antony L. Hosking and Michael D. Bond (Eds.). ACM, New York, NY, USA, 41–55. https://doi.org/10.1145/2754169.2754178

[59] Kaz Kylheku. 2014. What is PTHREAD_MUTEX_ADAPTIVE_NP? Retrieved November 8, 2018 from http://stackoverflow.com/a/25168942

[60] Beng-Hong Lim. 1995. *Reactive synchronization algorithms for multiprocessors*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, USA. http://hdl.handle.net/1721.1/36018

[61] Jean-Pierre Lozi. 2014. *Towards more scalable mutual exclusion for multicore architectures. (Vers des mécanismes d'exclusion mutuelle plus efficaces pour les architectures multi-cœur)*. Ph.D. Dissertation. Pierre and Marie Curie University, Paris, France. https://tel.archives-ouvertes.fr/tel-01067244

[62] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia L. Lawall, and Gilles Muller. 2012. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, Boston, MA, 65–76. https://www.usenix.org/conference/atc12/technical-sessions/presentation/lozi

[63] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia L. Lawall, and Gilles Muller. 2016. Fast and Portable Locking for Multicore Architectures. *ACM Trans. Comput. Syst.* 33, 4 (2016), 13:1–13:62. https://doi.org/10.1145/2845079

[64] Jean-Pierre Lozi, Baptiste Lepers, Justin R. Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues (Eds.). ACM, New York, NY, USA, 1:1–1:16. https://doi.org/10.1145/2901318.2901326

[65] Victor Luchangco, Daniel Nussbaum, and Nir Shavit. 2006. A Hierarchical CLH Queue Lock. In *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference, Dresden, Germany, August 28 - September 1, 2006, Proceedings (Lecture Notes in Computer Science)*, Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner (Eds.), Vol. 4128. Springer, Berlin, Heidelberg, 801–810. https://doi.org/10.1007/11823285_84

[66] Peter S. Magnusson, Anders Landin, and Erik Hagersten. 1994. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing, Cancún, Mexico, April 1994*, Howard Jay Siegel (Ed.). IEEE Computer Society, Washington, DC, USA, 165–171. https://doi.org/10.1109/IPPS.1994.288305

[67] Paul E. McKenney. 1996. Pattern Languages of Program Design 2. In *Pattern Languages of Program Design 2*, John M. Vlissides, James O. Coplien, and Norman L. Kerth (Eds.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, Chapter Selecting Locking Designs for Parallel Programs, 501–531. http://dl.acm.org/citation.cfm?id=231958.232968

[68] Paul E. McKenney. 1996. Selecting Locking Primitives for Parallel Programming. *Commun. ACM* 39, 10 (1996), 75–82. https://doi.org/10.1145/236156.236174

[69] Paul E. McKenney. 2017. Is Parallel Programming Hard, And, If So, What Can You Do About It? (v2017.01.02a). *CoRR* abs/1701.00854 (2017), 477. arXiv:1701.00854 http://arxiv.org/abs/1701.00854

[70] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (1991), 21–65. https://doi.org/10.1145/103727.103729

[71] Thannirmalai Somu Muthukaruppan, Anuj Pathania, and Tulika Mitra. 2014. Price theory based power management for heterogeneous multi-cores. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS*

'14, Salt Lake City, UT, USA, March 1-5, 2014, Rajeev Balasubramonian, Al Davis, and Sarita V. Adve (Eds.). ACM, New York, NY, USA, 161–176. https://doi.org/10.1145/2541940.2541974

[72] Regina Nuzzo. 2014. Scientific method: Statistical errors. *Nature* 506, 7487 (2014), 150–152. https://doi.org/10.1038/506150a

[73] Oracle Corporation. 2017. MySQL. Retrieved November 8, 2018 from https://www.mysql.com

[74] Y. Oyama, K. Taura, and A. Yonezawa. 1999. 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)*. World Scientific, Sendai, Japan, 23.

[75] Venkatesh Pallipadi and Alexey Starikovskiy. 2006. The ondemand governor. In *Proceedings of the Linux Symposium*, Vol. 2. sn, Linux, Ottawa, Canada, 215–230.

[76] Lennart Poettering. 2009. Measuring Lock Contention. Retrieved November 8, 2018 from http://0pointer.de/blog/projects/mutrace.html

[77] Zoran Radovic and Erik Hagersten. 2003. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA'03), Anaheim, California, USA, February 8-12, 2003*. IEEE Computer Society, Washington, DC, USA, 241–252. https://doi.org/10.1109/HPCA.2003.1183542

[78] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*. IEEE Computer Society, Washington, DC, USA, 13–24. https://doi.org/10.1109/HPCA.2007.346181

[79] David P. Reed and Rajendra K. Kanodia. 1979. Synchronization with Eventcounts and Sequences. *Commun. ACM* 22, 2 (1979), 115–123. https://doi.org/10.1145/359060.359076

[80] Haris Ribic and Yu David Liu. 2014. Energy-efficient work-stealing language runtimes. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, Rajeev Balasubramonian, Al Davis, and Sarita V. Adve (Eds.). ACM, New York, NY, USA, 513–528. https://doi.org/10.1145/2541940.2541971

[81] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. ffwd: delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, New York, NY, USA, 342–358. https://doi.org/10.1145/3132747.3132771

[82] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, New York, NY, USA, 164–174. https://doi.org/10.1145/1993498.1993518

[83] Michael L. Scott. 2013. *Shared-Memory Synchronization*. Morgan & Claypool Publishers, San Rafael, CA. https://doi.org/10.2200/S00499ED1V01Y201304CAC023

[84] Michael L. Scott and William N. Scherer III. 2001. Scalable queue-based spin locks with timeout. In *Proceedings of the 2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01), Snowbird, Utah, USA, June 18-20, 2001*, Michael T. Heath and Andrew Lumsdaine (Eds.). ACM, New York, NY, USA, 44–52. https://doi.org/10.1145/379539.379566

[85] Jianchen Shan, Xiaoning Ding, and Narain H. Gehani. 2017. APPLES: Efficiently Handling Spin-lock Synchronization on Virtualized Platforms. *IEEE Trans. Parallel Distrib. Syst.* 28, 7 (2017), 1811–1824. https://doi.org/10.1109/TPDS.2016.2625249

[86] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. 2013. Power containers: an OS facility for fine-grained power and energy management on multicore servers. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, Vivek Sarkar and Rastislav Bodík (Eds.). ACM, New York, NY, USA, 65–76. https://doi.org/10.1145/2451116.2451124

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

[88] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, O Fox, and David Patterson. 2008. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. Retrieved November 8, 2018 from https://pdfs.semanticscholar.org/34dd/c3da70f5b17ae0a73266ad1e4f9ae155811f.pdf

[89] SQLite Consortium. 2018. SQLite. Retrieved November 8, 2018 from https://www.sqlite.org

[90] Sun Microsystems. 2002. Multithreading in the Solaris Operating Environment. Retrieved November 8, 2018 from http://home.mit.bme.hu/~meszaros/edu/oprendszerek/segedlet/unix/2_folyamatok_es_utemezes/solaris_multithread.pdf

[91] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. 2010. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, R. Govindarajan, David A. Padua, and Mary W. Hall (Eds.). ACM, New York, NY, USA, 269–280. https://doi.org/10.1145/1693453.1693489

[92] Jons-Tobias Wamhoff, Stephan Diestelhorst, Christof Fetzer, Patrick Marlier, Pascal Felber, and Dave Dice. 2015. The TURBO Diaries: Application-controlled Frequency Scaling Explained. In *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany (LNI)*, Uwe Aßmann, Birgit Demuth, Thorsten Spitta, Georg Püschel, and Ronny Kaiser (Eds.), Vol. 239. GI, Dresden, Germany, 141–142. https://dl.gi.de/20.500.12116/2537

[93] Tianzheng Wang, Milind Chabbi, and Hideaki Kimura. 2016. Be my guest: MCS lock now welcomes guests. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, Rafael Asenjo and Tim Harris (Eds.). ACM, New York, NY, USA, 21:1–21:12. https://doi.org/10.1145/2851141.2851160

[94] Qiang Wu, Margaret Martonosi, Douglas W. Clark, Vijay Janapa Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David M. Brooks. 2005. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-38 2005), 12-16 November 2005, Barcelona, Spain*. IEEE Computer Society, Washington, DC, USA, 271–282. https://doi.org/10.1109/MICRO.2005.7

[95] Fen Xie, Margaret Martonosi, and Sharad Malik. 2003. Compile-time dynamic voltage scaling settings: opportunities and limits. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, Ron Cytron and Rajiv Gupta (Eds.). ACM, New York, NY, USA, 49–62. https://doi.org/10.1145/781131.781138

[96] Chao Xu, Felix Xiaozhu Lin, Yuyang Wang, and Lin Zhong. 2015. Automated OS-level Device Runtime Power Management. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, New York, NY, USA, 239–252. https://doi.org/10.1145/2694344.2694360

[97] Konrad Zemek. 2015. Asio, SSL, and scalability. Retrieved November 8, 2018 from https://konradzemek.com/2015/08/16/asio-ssl-and-scalability

[98] Mingzhe Zhang, Haibo Chen, Luwei Cheng, Francis C. M. Lau, and Cho-Li Wang. 2017. Scalable Adaptive NUMA-Aware Lock. *IEEE Trans. Parallel Distrib. Syst.* 28, 6 (2017), 1754–1769. https://doi.org/10.1109/TPDS.2016.2630695

Received July 2017; revised March 2018; accepted October 2018