Multicore Locks: The Case Is Not Closed Yet

Hugo Guiroux, Renaud Lachaize, Vivien Quéma

Multicore Locks: The Case Is Not Closed Yet

Hugo Guiroux and Renaud Lachaize, ***Université Grenoble Alpes and Laboratoire d'Informatique de Grenoble;*** **Vivien Quéma,** ***Université Grenoble Alpes, Grenoble Institute of Technology, and Laboratoire d'Informatique de Grenoble***

https://www.usenix.org/conference/atc16/technical-sessions/presentation/guiroux

This paper is included in the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Open access to the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16) is sponsored by USENIX.

Multicore Locks: The Case is not Closed Yet

Hugo Guiroux[†⋆] Renaud Lachaize[†⋆] Vivien Quéma[†‡⋆]
[†]*Université Grenoble Alpes* [‡]*Grenoble INP*
[⋆]*LIG (CNRS UMR 5217)*

Abstract

NUMA multicore machines are pervasive and many multithreaded applications are suffering from lock contention. To mitigate this issue, application and library developers can choose from the plethora of optimized mutex lock algorithms that have been designed over the past 25 years. Unfortunately, there is currently no broad study of the behavior of these optimized lock algorithms on realistic applications. In this paper, we attempt to fill this gap. We perform a performance study of 27 state-of-the-art mutex lock algorithms on 35 applications. Our study shows that the case is not yet closed regarding locking on multicore machines. Indeed, our conclusions include the following findings: (i) at its optimized contention level, no single lock is the best for more than 52% of the studied workloads; (ii) every lock is harmful for several applications, even if the application parallelism is properly tuned; (iii) for several applications, the best lock changes when varying the number of threads. These findings call for further research on optimized lock algorithms and dynamic adaptation of contention management.

1 Introduction

Today, multicore machines are pervasive and many multithreaded applications are suffering from bottlenecks related to critical sections and their corresponding locks. To mitigate these issues, application and library developers can choose from the plethora of optimized mutex lock algorithms that have been designed over the past 25 years but there is currently no clear study to guide this puzzling choice for realistic applications. In particular, the most recent and comprehensive empirical performance evaluation on multicore synchronization [9], 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 9 algorithms, does not consider hybrid spinning/blocking waiting policies, omits emerging approaches (e.g., load-control algorithms described in §2) and provides a modest coverage of hierarchical locks [14, 5, 6], a recent and efficient approach. Furthermore, most of the observations are based on microbenchmarks. 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 lock was designed [21, 26], or mostly based on microbenchmarks [14, 12].

The present paper provides a broad performance study on Linux/x86 of 27 state-of-the-art mutex lock algorithms on a set of 35 realistic and diverse applications (the PARSEC, Phoenix, SPLASH2 suites, MySQL and an SSL proxy). We make a number of observations, several of which have not been previously mentioned: (*i*) about 60% of the studied applications are significantly impacted by lock performance; (*ii*) no single lock is systematically the best, even for a fixed number of contending cores; (*iii*) worse, at their optimized contention level (individually tuned for each application), the best locks never dominate for more than 52% of the lock-sensitive applications; (*iv*) any of the locks is harmful (i.e., significantly inefficient compared to the best one) for at least several workloads; (*v*) across all the lock-sensitive applications, there is no clear performance hierarchy among the locks, even at a fixed number of contending cores; (*vi*) for a given application, the best lock varies according to both the number of contending cores and the machine; (*vii*) unlike previous recommendations [9] advocating that standard Pthread mutex locks should be avoided for workloads using no more than one thread per core, we find that, with our studied workloads, the current Linux implementation of these locks actually yields good performance for many applications with this pattern. Moreover, we show that all these results hold even when each configuration, i.e., each *(application, lock)* pair, is tuned to its optimal degree of parallelism. From our performance study, we draw two main conclusions. First, specific lock algorithms should not be hardwired into the

code of applications. Second, the observed trends call for further research both regarding lock algorithms and runtime support for parallel performance and contention management.

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 [26] or libslock [9]) would not have solved the problem, as this would still have required 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. Therefore, we implemented LiTL[1], a low-overhead library that allows transparent interposition of Pthread mutex lock operations and support for mainstream features like condition variables, without any restriction on the application-level locking discipline.

The remainder of the paper is organized as follows: §2 presents a taxonomy of existing lock designs and the list of algorithms covered by our study. §3 describes our experimental setup and the studied applications. §4 describes the LiTL library. §5 exposes the main results from our empirical observations. §6 discusses related works and §7 concludes the paper.

2 Lock algorithms

2.1 Background

The body of existing works on optimized lock algorithms for multicore architectures is rich and diverse and can be split into the following five categories:

1) Flat approaches correspond to simple algorithms (typically based on one or a few shared variables accessed by atomic instructions) such as: simple spinlock [33], backoff spinlock [2, 30], test and test-and-set (TTAS) lock [2], ticket lock [30], partitioned ticket lock [11], and standard Pthread mutex lock.
2) Queue-based approaches correspond to locks based on a waiting queue in order to improve fairness as well as the memory traffic, such as: MCS [30, 33] and CLH [7, 29, 33].
3) Hierarchical approaches are specifically aimed at providing scalable performance on large-scale NUMA machines, by attempting to reduce the rate of lock migrations among NUMA nodes. This category includes HBO [32], HCLH [28], FC-MCS [13], HMCS [5], AHMCS [6] and the algorithms that stem from the *lock cohorting* framework [14]. A cohort lock is based on a combination of two lock algorithms (similar or different): one used for the global lock and one used for the local locks (there is one local lock per NUMA node); in the usual C-L_A-L_B notation, L_A and L_B respectively correspond to the global and the node-level lock algorithms. The list includes C-BO-MCS, C-PTL-TKT and C-TKT-TKT (also known as Hticket [9]). The *BO*, *PTL* and *TKT* acronyms respectively correspond to backoff lock, partitioned ticket lock, and standard ticket lock.
4) Load-control approaches correspond to algorithms that aim at limiting the number of threads that concurrently attempt to acquire a lock, in order to prevent a performance collapse. These algorithms are derived from queue-based locks. This category includes MCS-TimePub[2] [19] and so-called *Malthusian algorithms* like Malth_Spin and Malth_STP[3] [12].
5) Delegation-based approaches correspond to algorithms 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 high lock contention. This category includes Oyama [31], Hendler [20], RCL [26], CC-Synch [15] and DSM-Synch [15].

Another important design dimension is the *waiting policy* used when a thread cannot immediately obtain a requested lock [12]. There are three main approaches: (i) spinning on a memory address, (ii) immediate parking (i.e., blocking the thread) either for a fixed amount of time or until the thread gets a chance to obtain the lock, and (iii) spinning-then-parking (STP), a hybrid strategy using a fixed or adaptive threshold [22]. 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 queue-based (from categories 2–4 above) and the standard Pthread mutex locks. Besides, note that the GNU C library for Linux provides two versions of Pthread mutex locks: the default one uses 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 [23].

2.2 Studied algorithms

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

[1] LiTL: Library for Transparent Lock interposition.

[2] MCS-TimePub is mostly known as MCS-TP but we use MC-TimePub to avoid confusion with MCS_STP.

[3] Malth_Spin and Malth_STP correspond to MCSCR-S and MCSCR-STP, respectively, but we do not use the latter names to avoid confusion with other MCS locks.

Name Total #cores	**A-64** 64	**A-48** 48	**I-48** 48 (no hyperthreading)
Server model	Dell PE R815	Dell PE R815	SuperMicro SS 4048B-TR4FT
Processors Microarchitecture Core clock Last-level cache (per node)	4× AMD Opteron 6272 Bulldozer / Interlagos 2.1 GHz 8 MB	4× AMD Opteron 6344 Piledriver / Abu Dhabi 2.6 GHz 8 MB	4× Intel Xeon E7-4830 v3 Haswell-EX 2.1 GHz 30 MB
Interconnect	HT3 - 6.4 GT/s per link	HT3 - 6.4 GT/s per link	QPI - 8 GT/s per link
Memory #NUMA nodes (#cores/node)	256 GB DDR3 1600 MHz 8 (8)	64 GB DDR3 1600 MHz 8 (6)	256 GB DDR4 2133 MHz 4 (12)
Network interfaces (10 GbE)	2× 2-port Intel 82599	2× 2-port Intel 82599	2-port Intel X540-AT2

Table 1: Hardware characteristics of the testbed platforms.

the application/OS behavior, code modifications, or fragile performance tuning: HCLH, HBO, FC-MCS, and all the delegation-based locks (see Dice et al. [14] for detailed arguments).

Our study considers 27 mutex lock algorithms that are representative of both well-established and state-of-the-art approaches. We use the *_Spin* and *_STP* suffixes to differentiate variants of the same algorithm that only differ in their waiting policy. The *-LS* tag corresponds to optimized algorithms borrowed from libslock [9]. Our set includes ten flat locks (Backoff, Partitioned ticket, Phtread, Pthread adaptive, Spinlock, Spinlock-LS, Ticket, Ticket-LS, TTAS, TTAS-LS), seven queue-based locks (Alock-LS, CLH-LS, CLH_Spin, CLH_STP, MCS-LS, MCS_Spin, MCS_STP), seven hierarchical locks (C-BO-MCS_Spin, C-BO-MCS_STP, C-PTL-TKT, C-TKT-TKT, Hticket-LS, HMCS, AHMCS), and three load-control locks (Malth_Spin, Malth_STP, MCS-TimePub).

3 Experimental setup and methodology

3.1 Testbed and studied applications

Our experimental testbed consists of three Linux-based servers whose main characteristics are summarized in Table 1. All the machines run the Ubuntu 12.04 OS with a 3.17.6 Linux kernel (CFS scheduler), glibc 2.15 and gcc 4.6.3. For our comparative study of lock performance, we consider (i) the applications from the PARSEC benchmark suite (emerging workloads), (ii) the applications from the Phoenix 2 MapReduce benchmark suite, (iii) the applications from the SPLASH2 high-performance computing benchmark suite[4], (iv) the MySQL database running the Cloudstone workload, and (v) SSL proxy, an event-driven SSL endpoint that processes small messages. In order to evaluate the impact of workload changes on locking performance, we also consider so called "long-lived" variants of four of the above workloads denoted with a "_ll" suffix. Note that six of the applications cannot be evaluated on the two 48-core machines because, by design, they only accept a number of threads that correspond to a power of two: facesim, fluidanimate (from PARSEC), fft, ocean_cp, ocean_ncp, radix (from SPLASH2).

Most of these applications use a number of threads equal to the number of cores, except the three following ones: dedup (3× threads), ferret (4× threads) and MySQL (hundreds of threads). Two thirds of the applications use Pthread condition variables.

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

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 [12]) rely on a fixed threshold for the spinning time that corresponds to the duration of a round-trip context switch [22] — 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. Generally, in the experiments presented in this paper, we study the performance impact of a lock for a given contention level, i.e., the number of threads of the application. We vary the contention level at the granularity of a NUMA node (i.e., 8 cores for the A-64 machine, 6 cores for the A-48 machine, and 12 cores for the I-48 machine). For most of the experiments detailed in the paper, the application threads are not pinned to specific cores. The impact of pinning is nonetheless discussed in §5.3.

Finally, each experiment is run at least five times and we compute the average value. Overall, we observe little variability for most configurations. For all experiments, the considered application-level performance metric is the throughput (operations per time unit).

4 The LiTL lock interposition library

In order to carry out the lock comparison study, we have developed LiTL, an interposition library for Linux/x86 allowing transparently replacing the lock algorithm used for Pthread mutexes. We describe its design, implementation, and assess its performance.

4.1 Design

The design of LiTL does not impose any restriction on the level of nested locking and is compatible with arbitrary locking disciplines (e.g., hand-over-hand locking [33]). The pseudo-code of the main wrapper functions of the LiTL library is depicted in Figure 1.

```
// return values and error checks
// omitted for simplification

pthread_mutex_lock(pthread_mutex_t *m) {
    optimized_mutex_t *om = get_optimized_mutex(m);
    if (om == null) {
        om = create_and_store_optimized_mutex(m);
    }
    optimized_mutex_lock(om);
    real_pthread_mutex_lock(m);
}

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);
    optimized_mutex_lock(om);
    real_pthread_mutex_lock(m);
}

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

Figure 1: Overview of the pseudocode for the main wrapper functions of LiTL.

General principles The primary role of LiTL is to maintain a mapping structure 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 implies that LiTL must keep track of the lifecycle of all the application's 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. For example, in the case of an MCS lock, this parameter corresponds to the record to be inserted in (or removed from) the lock's waiting queue. In the general case, a struct cannot be reused nor freed before the corresponding lock has been released. For instance, an application may rely on nested critical sections (i.e., a thread T must acquire a lock L_2 while holding another lock L_1). In this case, T must use a distinct struct for L_2 in order to preserve the integrity of L_1's struct. In order to gracefully support the most general cases, LiTL systematically allocates exactly one struct per lock instance and per thread.

Supporting condition variables 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. We therefore use the following strategy: our wrapper for `pthread_cond_wait()` internally calls the true `pthread_cond_wait()` function. To issue this call, we need to hold a real Pthread mutex lock (of type `pthread_mutex_t`). This strategy (depicted in the pseudocode of Figure 1) does not introduce high contention on the internal Pthread lock. Indeed, for workloads that do not use condition variables, the Pthread lock is only requested by the holder of the optimized lock associated with the critical section. Furthermore, 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. Note that the latter claim also holds for workloads that rely on `pthread_cond_broadcast()` because the Linux implementation of this call 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.

Support for specific lock semantics 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 such semantics. Although not yet implemented, LiTL could easily support blocking requests with timeouts for the so-called "abortable" locks (e.g., MCS-Try [34] and MCS-TimePub [19]). Moreover, support for optional Pthread

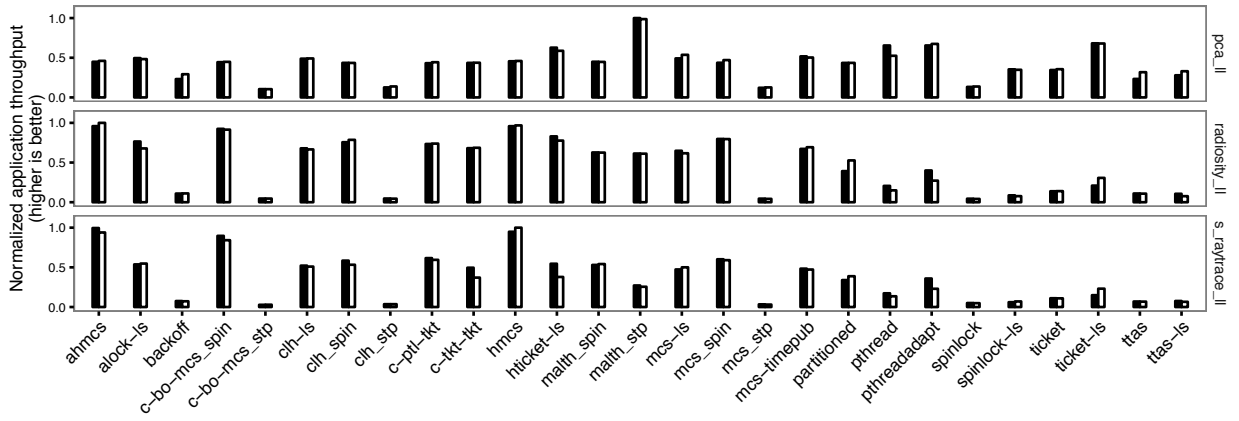

Figure 2: Performance comparison (throughput) of manually implemented locks (black bars) vs. transparently interposed locks using LiTL (white bars). The throughput is normalized with respect to the best performing configuration for a given application (**A-64 machine**).

mutex behavior like reentrance and error checks[5] could be easily integrated in the generic wrapper code by managing fields for the current owner and the lock acquisition counter.

4.2 Implementation

The library relies on a scalable concurrent hash table (CLHT [10]) 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 [9, 1, 26]. Other algorithms are implemented from scratch based on the description of the original papers. For algorithms that are based on a parking or on a spinning-then-parking waiting policy, our implementation directly relies on the `futex` Linux system call.

Finally, the source code of LiTL relies on preprocessor macros rather than function pointers. Indeed, we have observed that the use of function pointers in the critical path introduced a surprisingly high overhead. Moreover, all data structures are cache-aligned in order to mitigate the impact of false sharing.

4.3 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 between both configurations: (i) the achieved performance is close and (ii) the general trends for the different locks remain stable.

We selected three applications: pca_ll, radiosity_ll and s_raytrace_ll. These three applications are particularly lock-intensive and the last two use Pthread condition variables. Therefore, all three represent an unfavorable case 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: black bars correspond to manually implemented locks, whereas white bars correspond to transparently interposed locks using LiTL. In addition, Table 2 summarizes the performance differences for each application: number of locks for which each version performs better and, in each case, the average gain and the relative standard deviation.

We observe that, for all of the three applications, the results achieved by the two versions of the same lock are very close: the average performance difference is below 5%. Besides, Figure 2 highlights that the general trends observed with the manual versions are preserved with the interposed versions. We thus conclude that using LiTL to study the behavior of lock algorithms in an application yields only very modest differences with respect to the performance behavior of a manually modified version.

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

		pca_ll	radiosity_ll	s_raytrace_ll
Manual	Winners	10	17	19
	Average Gain	2%	3%	4%
	Rel. Dev.	4%	4%	5%
LiTL	Winners	17	10	8
	Average Gain	2%	3%	3%
	Rel. Dev.	2%	5%	3%

Table 2: Detailed statistics for the performance comparison of manually implemented locks vs. transparently interposed locks using LiTL (**A-64 machine**).

5 Performance study of lock algorithms

In this section, we use LiTL to compare the behavior of the different lock algorithms on different workloads and at different levels of contention. In the interest of space, we do not systematically report the observed standard deviations. However, in order to mitigate the impact of variability, when comparing the performance of two locks, we consider a margin of 5%: lock A is considered better than lock B if B's achieved performance is below 95% of A's. Besides, in order to make fair comparisons, the results presented for the Pthread locks are obtained using the same library interposition mechanism as with the other locks.

Note that some configurations are not tested because of specific restrictions. First, streamcluster, streamcluster_ll, and vips cannot use CLH-based locks or Hticket-LS as they do not support trylocks semantics. Second, we omit the results for most locks with MySQL: given the extremely large ratio of threads to cores, most locks yield performance close to zero. Third, some applications, e.g., dedup and fluidanimate, run out of memory for some configurations.

Finally, for the sake of space, we do not report all the results for the three studied machines. We rather focus on the A-64 machine and provide summaries of the results for the A-48 and I-48 machines. Nevertheless, the entire set of results can be found in a companion technical report [18].

The section is structured as follows. §5.1 provides preliminary observations that drive the study. §5.2 answers the main questions of the study regarding the observed lock behavior. §5.3 discusses additional observations.

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 3 shows two metrics for each application and for different numbers of nodes on the A-64 machine: 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. For the moment, we only focus on the relative standard deviations at the maximum number of nodes (*max nodes*—highest contention) given in the 5th column (the detailed results from this table are discussed in §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). We observe that about 60% of the applications are impacted by locks. We observe similar trends on the three studied machines (see Table 4).

In the remainder of this study, we focus on lock-sensitive applications.

	Gain 1 node	R.Dev. 1 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	**683%**	**56%**	**970%**	**55%**	**683%**	**56%**
facesim	**10%**	**2%**	**771%**	**76%**	**14%**	**3%**
ferret	**1%**	**0%**	**349%**	**58%**	**107%**	**25%**
fft	8%	2%	11%	3%	9%	2%
fluidanimate	**48%**	**11%**	**302%**	**28%**	**133%**	**20%**
fmm	**26%**	**7%**	**42%**	**12%**	**42%**	**11%**
freqmine	7%	2%	6%	1%	6%	1%
histogram	7%	2%	20%	5%	12%	3%
kmeans	9%	3%	12%	2%	12%	2%
linear_regression	**9%**	**2%**	**228%**	**22%**	**49%**	**10%**
lu_cb	11%	2%	5%	1%	5%	1%
lu_ncb	17%	5%	8%	2%	8%	2%
matrix_multiply	**7%**	**3%**	**643%**	**51%**	**372%**	**38%**
mysqld	**30%**	**9%**	**174%**	**38%**	**122%**	**34%**
ocean_cp	**17%**	**4%**	**129%**	**15%**	**22%**	**5%**
ocean_ncp	**21%**	**5%**	**118%**	**14%**	**18%**	**4%**
pca	**12%**	**3%**	**358%**	**31%**	**47%**	**8%**
pca_ll	**19%**	**5%**	**665%**	**47%**	**100%**	**20%**
p_raytrace	2%	0%	1%	0%	2%	0%
radiosity	**3%**	**1%**	**91%**	**13%**	**13%**	**4%**
radiosity_ll	**8%**	**2%**	**2299%**	**71%**	**180%**	**29%**
radix	2%	1%	8%	2%	8%	2%
s_raytrace	**4%**	**1%**	**1929%**	**62%**	**126%**	**29%**
s_raytrace_ll	**4%**	**1%**	**3343%**	**79%**	**157%**	**26%**
ssl_proxy	**37%**	**6%**	**1309%**	**63%**	**58%**	**11%**
streamcluster	**13%**	**3%**	**1087%**	**56%**	**13%**	**3%**
streamcluster_ll	**23%**	**4%**	**1305%**	**55%**	**56%**	**12%**
string_match	5%	2%	11%	2%	11%	2%
swaptions	8%	2%	10%	2%	10%	2%
vips	**2%**	**1%**	**334%**	**32%**	**8%**	**2%**
volrend	**7%**	**1%**	**161%**	**21%**	**24%**	**5%**
water_nsquared	**10%**	**2%**	**94%**	**14%**	**94%**	**14%**
water_spatial	**24%**	**5%**	**98%**	**15%**	**96%**	**15%**
word_count	4%	1%	17%	3%	12%	2%
x264	4%	1%	6%	2%	5%	2%

Table 3: For each application, performance gain of the best vs. worst lock and relative standard deviation (**A-64 machine**).

	A-64	A-48	I-48
# tested applications	39	33	33
# lock-sensitive applications	23	19	17

Table 4: Number of tested applications and number of lock-sensitive applications (**all machines**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	252	129	89	95	229	200	204	125	117	75	96	119	119	106	110	113	80	136	120	126	147	118	141	121	145	197
facesim	412	908	425	172	55	888	895	78	460	328	324	379	711	71	1k	948	87	26	895	91	67	726	35	919	462	489	530
ferret	134	176		46		170	174		109	63	100	108	57		194	192			173					182	34		7
fluidanimate	-	72			9	-	-	-				-	7	53	8	12	54	7				16		13	11	6	65
fmm							15	12																			
histogram	95	88	90	95	95	87	92	92	84	79	94	90	90	88	89	85	109	84	89	125	88	107	87	105	102	97	104
linear_regression	44	227	12	21	132	67	45	34	7	49	44	15	25	8	51	47	24		50	10	8	38	8	21			27
matrix_multiply		259								92	287	66			62				7				64		65		55
mysqld	-	-	-	-		-	-	-	-	-	-	-	-		-	-		25	-			-	-	-	-	-	-
ocean_cp	107	97	114	81	70	103	124	121	89	92	96	73	87	75	111	114	82	45	103	72	73	234	49	136	60	106	173
ocean_ncp	93	99	90	73	69	90	93	79	76	90	81	73	84	85	73	92	95	61	98	97	85	206	56	89	57	93	186
pca	77	79	163	42	370	69	44	148	40	34	68	49	37		49	55	134	19	50	97	36	229	80	116	35	160	130
pca_ll	91	81	219	14	582	74	41	321	23	16	88	31	7	21	58	41	403		21	195	114	513	168	108	51	206	476
radiosity																	69						21		10		53
radiosity_ll		12	413		1k	13	10	699	33	19			7		13	11	792	18	48	157	71	987	164	296	97	411	615
s_raytrace		18	185		1k		66	460		14	13	16		7			436		100	88	14	269	50	134	149	195	154
s_raytrace_ll	19	96	781	17	2k	110	107	1k	83	180	15	170	68	161	108	88	1k	118	178	371	185	1k	308	495	301	857	881
ssl_proxy	44	69	695	33	1k	107	61	1k	61	103	608	78	36	52	95	99	1k	73	87	268	195	2k	268	360	139	718	957
streamcluster	2k	2k	4k	2k	2k	-	-	-	1k	2k	1k	-	4k	16k	4k	3k	16k	1k	1k	2k	3k	9k	2k	5k	4k	4k	7k
streamcluster_ll	421	246	829	410	497	-	-	-	266	275	250	-	816	4k	774	590	4k	301	275	446	450	2k	585	1k	615	718	1k
vips	64	56	22	400	32	-	-	-	331	189	131	-	229	18	46	51	18	21	60	20	21	20	23	37	28	22	26
volrend	52	88	97	62	99	72	82	123	50	62	52	59	69	128	79	86	109	82	83	131	162	222	114	74	70	108	154
water_nsquared																											
water_spatial																											

Table 5: For each *(application, lock)* pair, performance gain (in %) of the optimized configuration over the max-node configuration. The background color of a cell indicates the number of nodes (1, 2, 4, 6, or 8 nodes) for the optimized configuration: 1 | 2 | 4 | 6 | 8. Dashes correspond to untested cases. (**A-64 machine**).

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 machines, we consider 1, 2, 3, and 4 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.

The results for the A-64 machine are displayed in Table 5. For each *(application, lock)* pair, the corresponding cell indicates the performance gain of the optimized configuration with respect to the max-node configuration. The background color of a cell indicates the number of nodes for the optimized configuration. In addition, Table 6 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 5) that the performance gain of the optimized configuration is often extremely large. 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.

	A-64	A-48
1 Node	11%	9%
2 Nodes	28%	24%
4 Nodes	27%	21%
6 Nodes	7%	9%
8 Nodes	27%	37%

	I-48
1 Node	33%
2 Nodes	14%
3 Nodes	8%
4 Nodes	45%

Table 6: Breakdown of the *(application, lock)* pairs according to their optimized number of nodes (**all machines**).

In light of the above observations, the main questions investigated in the study (§5.2) will be considered from two complementary angles: (i) comparing locks at a fixed number of nodes, and (ii) comparing locks at their optimized configurations (i.e., with possibly a different number of nodes for each). The first angle offers insight for situations in which the degree of parallelism cannot be adjusted, while the second is useful for scenarios in which more advanced application tuning is possible.

5.2 Main questions

5.2.1 How much do locks impact applications?

Table 3 shows, for each application, the performance gain of the best lock over the worst one at 1 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 impact of locks on the performance of applications depends on the number of nodes. **At 1 node, the impact of locks on lock-sensitive applications is moderate**. 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 3343%. Finally, **at the optimized number of 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 5), which considerably increases the observed performance gaps between locks. We observe the same trends on the A-48 and I-48 machines (see the companion technical report [18]).

5.2.2 Are some locks always among the best?

Table 7 shows 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 for the A-64 machine. The results are shown for three configurations: 1 node, max nodes, and opt nodes. Besides, Table 8 displays, for each machine (at 1 node, max nodes and opt nodes) the following metrics aggregated over the different locks: the min and max coverage, the average coverage, and the relative standard deviation of the coverage.

	Number of nodes		
Locks	1	Max	Opt
ahmcs	67%	24%	52%
alock-ls	52%	4%	30%
backoff	83%	30%	26%
c-bo-mcs_spin	74%	22%	39%
c-bo-mcs_stp	62%	12%	29%
clh-ls	63%	5%	37%
clh_spin	68%	5%	37%
clh_stp	63%	16%	21%
c-ptl-tkt	57%	22%	35%
c-tkt-tkt	74%	22%	39%
hmcs	65%	22%	48%
hticket-ls	63%	16%	37%
malth_spin	61%	9%	26%
malth_stp	54%	29%	29%
mcs-ls	74%	4%	30%
mcs_spin	70%	22%	48%
mcs_stp	79%	21%	29%
mcs-timepub	54%	38%	29%
partitioned	70%	22%	39%
pthread	50%	21%	29%
pthreadadapt	58%	33%	29%
spinlock	65%	26%	30%
spinlock-ls	57%	30%	35%
ticket	74%	22%	39%
ticket-ls	74%	13%	35%
ttas	83%	26%	43%
ttas-ls	65%	0%	9%

Table 7: For each lock, fraction of the lock-sensitive applications for which the lock yields the best performance for three configurations: 1 node, max nodes, and opt nodes (**A-64 machine**).

# nodes	Coverage	A-64	A-48	I-48
1	[min; max]	[50%; 83%]	[27%; 83%]	[44%; 89%]
	Avg.	66%	66%	62%
	Rel. Dev.	9%	15%	12%
Max	[min; max]	[0%; 38%]	[0%; 42%]	[5%; 50%]
	Avg.	19%	17%	24%
	Rel. Dev.	10%	12%	11%
Opt	[min; max]	[9%; 52%]	[0%; 47%]	[5%; 50%]
	Avg.	34%	21%	28%
	Rel. Dev.	9%	13%	12%

Table 8: Statistics on the coverage of locks for three configurations: 1 node, max nodes, and opt nodes (**all machines**).

We make the following observations (Table 8). **No lock is among the best for more than 89% of the applications at 1 node and for more than 52% of the applications both at max nodes and at the optimal number of nodes**. We also observe that the average coverage is much higher at 1 node than at max nodes, and slightly higher at the optimized number of nodes than at max nodes. This is directly explained by the observations made in §5.2.1. First, at 1 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 the optimized number of nodes. We observe the same trends on the A-48 and I-48 machines (see the companion technical report [18]).

5.2.3 Is there a clear hierarchy between locks?

Table 9 shows pairwise comparisons for all locks, at max nodes on the A-64 machine. In each table, cell $(rowA, colB)$ contains 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. For example, Table 9 shows that for 38% of the applications, AHMCS performs at least 5% better than Backoff at the optimized number of nodes. Similarly, the table shows that Backoff is at least 5% better than AHMCS for 29% of the applications. From these two values, we can conclude that the two above mentioned locks perform very closely for 33% of the applications. At the end of each line (resp. column), the table also shows the mean of the fraction of applications for which a lock is better (resp. worse) than others. Besides, the latter two metrics are summarized for the three machines in Table 10.

We observe that **there is no clear global performance hierarchy between locks**. More precisely, for most pairs of locks (A, B), there are some applications for which A is better than B, and vice-versa (Table 9). The only marginal exceptions are the cells having 0% for value. This corresponds to pairs of locks (A, B) for which A

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		19	38	48	29	22	17	61	19	48	5	33	33	43	38	38	48	52	24	38	43	57	48	33	33	43	38	36
alock-ls	19		39	30	26	16	16	58	17	22	9	26	39	30	22	26	43	30	9	39	43	48	39	35	30	35	39	30
backoff	29	35		30	26	37	37	58	26	26	35	32	35	26	35	30	52	30	17	35	39	30	26	4	22	0	39	30
c-bo-mcs_spin	33	48	43		35	37	32	74	22	17	39	32	39	48	39	9	48	13	22	39	39	39	43	48	39	35	65	38
c-bo-mcs_stp	33	43	35	22		42	32	74	17	22	30	21	22	25	26	26	42	21	13	33	33	39	26	26	22	26	61	31
clh-ls	22	21	37	42	32		16	47	26	26	16	26	37	37	16	32	47	26	16	42	47	53	47	47	42	42	47	34
clh_spin	22	32	32	32	26	32		53	21	37	21	42	32	26	32	21	47	32	11	37	37	47	42	32	42	37	47	33
clh_stp	33	32	5	16	11	37	16		26	16	26	26	16	11	21	16	11	5	11	11	11	21	21	11	26	11	32	18
c-ptl-tkt	19	35	35	39	30	32	21	68		26	22	26	26	43	30	26	57	39	17	39	35	48	35	30	30	35	57	35
c-tkt-tkt	24	39	35	26	39	32	26	74	26		30	32	48	65	43	17	57	22	9	39	43	39	43	39	43	35	65	38
hmcs	14	30	39	35	22	42	32	74	17	39		32	39	35	35	26	52	39	26	39	39	48	39	30	30	30	52	36
hticket-ls	17	16	47	32	26	21	32	74	11	21	5		32	42	11	26	53	32	11	42	42	53	42	37	26	47	58	33
malth_spin	14	35	22	22	26	26	16	63	13	17	22	16		22	22	13	39	17	4	35	35	35	39	17	13	17	48	25
malth_stp	24	35	22	35	21	32	37	58	17	17	26	21	4		22	17	33	25	9	33	29	35	22	17	17	17	48	26
mcs-ls	24	17	35	35	35	21	26	63	13	17	17	16	35	26		17	39	17	4	39	43	43	35	30	17	35	48	29
mcs_spin	29	43	35	26	39	37	32	68	26	17	39	47	39	43	43		43	22	22	35	39	35	43	39	30	39	61	37
mcs_stp	29	35	9	22	21	32	32	42	22	9	30	26	17	17	26	9		12	17	21	25	17	17	13	17	13	39	22
mcs-timepub	33	39	35	22	33	42	37	68	17	9	30	32	39	29	22	9	38		13	29	33	30	35	30	30	30	57	32
partitioned	24	39	26	39	43	32	32	68	26	22	39	53	52	43	35	35	61	35		43	48	48	43	26	43	35	65	41
pthread	29	39	22	26	25	37	32	58	22	17	39	26	30	25	35	26	46	25	13		21	39	13	17	13	17	43	28
pthreadadapt	29	43	22	35	21	37	37	53	30	26	35	26	26	25	35	30	42	25	17	21		22	22	17	17	17	43	29
spinlock	29	39	9	26	17	37	32	53	35	13	39	32	43	35	35	22	39	17	22	26	30		26	13	30	9	35	29
spinlock-ls	29	39	26	30	35	26	26	63	26	30	35	16	30	30	30	30	48	30	22	43	30	48		26	13	26	57	33
ticket	29	35	9	26	26	32	32	63	26	22	35	32	30	26	30	26	48	22	13	26	39	30	26		22	0	39	29
ticket-ls	19	22	30	26	39	26	32	68	26	26	22	11	35	39	22	26	52	26	26	35	48	43	39	30		30	52	33
ttas	24	35	4	26	22	37	26	63	26	17	35	32	30	26	30	30	52	17	17	30	35	30	26	4	26		30	28
ttas-ls	19	17	9	17	13	21	16	42	13	13	4	5	22	22	9	22	30	9	13	17	22	30	17	13	4	9		17
average	25	33	27	29	28	32	28	62	22	22	26	28	32	32	29	23	45	25	15	33	36	39	33	26	26	26	49	

Table 9: For each pair of locks *(rowA, colB)* at the optimized number of nodes, score of lock A vs lock B: percentage of applications for which lock A performs at least 5% better than B (**A-64 machine**).

	Better			Worse		
Lock	A-64	A-48	I-48	A-64	A-48	I-48
ahmcs	36%	40%	52%	25%	28%	25%
alock-ls	30%	42%	37%	33%	25%	32%
backoff	30%	29%	23%	27%	33%	45%
c-bo-mcs_spin	38%	47%	46%	29%	25%	15%
c-bo-mcs_stp	31%	25%	38%	28%	44%	25%
clh-ls	34%	46%	32%	32%	32%	38%
clh_spin	33%	38%	33%	28%	34%	37%
clh_stp	18%	11%	8%	62%	72%	71%
c-ptl-tkt	35%	44%	54%	22%	26%	13%
c-tkt-tkt	38%	42%	51%	22%	27%	15%
hmcs	36%	50%	52%	26%	21%	17%
hticket-ls	33%	45%	42%	28%	25%	17%
malth_spin	25%	36%	31%	32%	37%	35%
malth_stp	26%	20%	28%	32%	53%	36%
mcs-ls	29%	43%	35%	29%	22%	26%
mcs_spin	37%	38%	36%	23%	33%	23%
mcs_stp	22%	23%	20%	45%	59%	52%
mcs-timepub	32%	38%	34%	25%	34%	29%
partitioned	41%	42%	38%	15%	32%	23%
pthread	28%	33%	34%	33%	43%	35%
pthreadadapt	29%	34%	34%	36%	38%	36%
spinlock	29%	35%	20%	39%	44%	49%
spinlock-ls	33%	41%	38%	33%	30%	31%
ticket	29%	23%	17%	26%	44%	53%
ticket-ls	33%	40%	28%	26%	24%	35%
ttas	28%	28%	24%	26%	34%	44%
ttas-ls	17%	27%	20%	49%	42%	52%

Table 10: For each lock, at the optimized number of nodes, mean of the fraction of applications for which the lock is better (resp. worse) than other locks (**all machines**).

never yields better performance than *B*. The results at max nodes (not shown due to lack of space) 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 and I-48 machines (see the companion technical report [18]).

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 11 displays, for the A-64 machine, the performance gain brought by the best lock with respect to each of the other locks for each application at max nodes (top part) and at the optimized number of nodes for each lock (bottom part). For example, the top part of the table shows that for the dedup application, the best lock (0%, here Spinlock-LS) is 598% better than the Alock-LS lock. The gray cells highlight values greater than 15%. Thus, for each lock in a column, the number of grey cells corresponds to the number of applications for which the lock is beaten by a gap of 15% or more by the best lock(s) for this application. In addition, Table 12 displays, for each machine, the fraction of applications that are significantly hurt by a given lock.

On the three machines, we observe that, **both at max**

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	
dedup	-	598	4	135	137	970	575	576	27	11	145	130	130	129	123	127	128	105	14	6	2	2	0	4	0	5	579	
facesim	298	701	323	107	25	680	687	52	333	224	234	273	531	40	771	710	52	0	685	56	44	572	6	719	340	368	409	
ferret	329	297	10	84	0	261	312	0	286	228	255	291	196	0	349	317	0	4	314	0	1	10	0	331	84	9	11	
fluidanimate	-	301	0	57	65	-	-	-	35	14	72	-	36	95	50	40	94	50	14	5	12	26	0	17	15	9	201	
fmm	41	37	15	3	26	38	39	33	30	0	35	32	16	14	32	2	0	0	14	25	23	2	25	15	27	17	34	
histogram	1	2	8	3	4	3	3	12	2	0	2	0	0	1	5	1	14	1	4	19	2	18	3	11	5	8	12	
linear_regression	32	228	24	20	108	57	31	62	0	52	28	11	17	0	49	46	56	3	39	15	0	83	15	32	9	19	49	
matrix_multiply	9	559	5	26	7	18	9	3	24	136	608	642	5	3	639	27	2	0	33	3	3	5	637	3	633	5	630	
mysqld	-	-	-	-	30	-	-	-	-	-	-	-	-	0	-	-	7	173	-	97	102	-	-	-	-	-	-	
ocean_cp	31	18	37	22	16	27	38	38	24	29	29	15	23	27	27	43	32	0	24	11	19	129	5	55	5	38	81	
ocean_ncp	27	28	29	30	9	25	27	28	12	28	16	10	20	22	14	36	37	11	29	31	27	118	0	25	2	29	93	
pca	65	69	155	46	357	61	48	220	40	38	59	39	38	0	43	58	214	23	45	110	39	252	75	110	23	157	112	
pca_ll	47	38	251	24	664	25	51	511	30	24	41	0	18	36	17	50	526	15	27	206	68	584	128	128	17	241	338	
radiosity	14	12	0	0	1	13	9	0	8	1	7	9	9	12	10	1	91	0	1	0	0	1	33	0	19	0	71	
radiosity_ll	0	47	801	9	2k	50	16	2k	35	45	3	28	59	63	62	12	2k	44	76	567	267	2k	396	614	193	825	1k	
s_raytrace	2	24	536	17	2k	9	75	1k	8	27	18	38	26	64	16	0	1k	13	122	230	122	714	118	412	225	554	471	
s_raytrace_ll	6	82	1k	18	3k	96	87	3k	68	169	0	164	84	291	99	69	3k	111	157	639	335	2k	428	813	332	1k	1k	
ssl_proxy	0	18	532	1	1k	47	16	879	9	41	379	20	16	35	43	47	900	29	36	293	153	1k	249	271	85	539	735	
streamcluster	45	24	153	13	63	-	-	-	7	13	3	-	210	1k	183	118	979	6	0	90	133	505	33	290	166	177	395	
streamcluster_ll	61	6	188	20	55	-	-	-	0	17	6	-	234	1k	202	133	1k	34	13	77	102	518	65	263	139	155	411	
vips	41	38	4	333	17	-	-	-	267	145	101	-	177	0	28	28	1	3	37	0	2	3	1	16	8	4	10	
volrend	2	28	41	9	34	16	25	58	1	9	0	6	17	63	22	26	47	24	24	78	104	161	58	24	16	51	92	
water_nsquared	94	48	2	2	9	58	35	35	7	0	14	10	7	6	9	3	2	7	4	6	7	0	6	4	6	4	37	
water_spatial	97	49	2	11	7	63	40	39	4	5	8	4	8	5	5	9	9	10	1	0	0	2	1	1	0	1	41	
dedup	-	378	10	199	193	682	443	436	36	23	237	183	153	152	161	160	158	174	16	16	9	0	10	3	10	3	451	
facesim	2	4	6	0	6	4	4	12	1	0	4	2	2	8	3	1	7	4	3	7	13	7	3	5	3	4	6	
ferret	88	47	6	29	0	37	53	0	89	106	82	92	93	0	56	46	0	3	55	0	0	7	0	56	41	6	7	
fluidanimate	-	133	0	50	51	-	-	-	35	14	64	-	28	27	39	25	26	40	14	5	12	9	0	4	3	3	83	
fmm	41	35	15	3	26	38	21	19	30	0	33	32	16	14	32	2	0	0	14	25	23	1	25	15	27	17	34	
histogram	0	5	9	1	2	6	3	11	6	6	1	1	1	3	6	4	4	5	6	2	3	9	5	3	0	4	5	
linear_regression	2	12	24	11	0	5	1	35	4	14	0	8	5	4	10	11	39	14	4	16	4	48	19	22	15	25	30	
matrix_multiply	9	83	5	22	7	18	9	3	24	23	83	348	5	3	357	23	2	0	24	3	3	5	349	3	343	5	372	
mysqld	-	-	-	-	31	-	-	-	-	-	-	-	-	0	-	-	8	121	-	96	96	-	-	-	-	-	-	
ocean_cp	5	0	7	12	13	4	2	4	10	12	10	11	9	21	0	11	20	14	2	7	15	14	18	9	9	12	10	
ocean_ncp	3	1	6	17	1	3	3	12	0	5	0	0	2	3	3	10	10	8	2	4	7	11	0	4	2	5	5	
pca	2	4	6	13	6	4	12	41	10	12	4	3	11	7	5	12	47	13	6	17	12	17	7	7	0	8	1	
pca_ll	6	5	51	49	54	0	48	100	46	48	3	5	53	55	3	46	71	51	45	43	8	53	17	51	7	53	5	
radiosity	10	9	0	0	1	10	8	0	6	1	7	9	7	10	8	1	13	0	1	0	0	1	10	0	9	0	11	
radiosity_ll	0	31	75	9	53	32	5	180	1	22	3	28	49	59	42	1	165	22	19	159	114	120	88	80	49	80	83	
s_raytrace	2	5	123	16	74	9	5	123	5	11	5	19	26	53	14	0	117	12	10	75	94	120	45	119	30	121	125	
s_raytrace_ll	2	6	79	16	74	7	4	157	5	10	0	11	25	72	9	3	150	11	6	79	74	75	48	75	23	76	78	
ssl_proxy	3	4	17	12	23	5	7	30	0	3	0	0	26	31	9	9	23	11	7	57	27	20	40	19	15	15	16	
streamcluster	11	9	6	0	4	-	-	-	8	1	7	-	10	10	9	1	2	5	7	12	7	2	2	8	8	7	9	
streamcluster_ll	30	29	31	0	9	-	-	-	15	31	28	-	54	47	46	42	39	41	27	36	55	46	2	33	41	31	35	
vips	4	7	3	4	7	-	-	-	3	3	5	-	2	2	5	2	3	3	3	0	1	4	0	2	2	3	5	
volrend	2	4	9	2	2	3	4	8	3	2	0	1	5	8	4	3	7	4	3	17	18	23	12	8	4	10	15	
water_nsquared	94	48	2	2	9	58	35	35	7	0	14	10	7	6	9	3	2	7	4	6	7	0	6	4	6	4	37	
water_spatial	95	49	2	11	7	63	40	39	4	5	8	4	8	5	5	9	9	10	1	0	0	2	1	1	0	1	41	

Max nodes

Opt nodes

Table 11: For each application, at max nodes (top part) and at the optimized number of nodes (bottom part), performance gain (in %) obtained by the best lock(s) with respect to each of the other locks. The grey background highlights cells for which the performance gains are 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**).

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 the optimized number of nodes than at max nodes. This is explained by the fact that several of the locks create performance collapses at max nodes, which does not occur at the optimized number of nodes. Moreover, we observe that, for each lock, the performance gap to the best lock can be significant (Table 11).

5.3 Additional observations

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. For example, in the case of the facesim application, there are 18% of the pairwise performance comparisons between locks that change when moving from a 1-node configuration to a 2-node configuration. Similarly, there are 95% of pairwise comparisons that change at least once when considering

Lock	A-64		A-48		I-48	
	Max	Opt	Max	Opt	Max	Opt
ahmcs	62%	24%	56%	39%	39%	33%
alock-ls	87%	39%	61%	39%	58%	58%
backoff	61%	35%	68%	53%	58%	53%
c-bo-mcs_spin	61%	35%	53%	58%	47%	32%
c-bo-mcs_stp	71%	38%	80%	65%	55%	45%
clh-ls	84%	37%	73%	40%	69%	62%
clh_spin	84%	32%	60%	47%	62%	56%
clh_stp	79%	58%	87%	87%	81%	75%
c-ptl-tkt	52%	30%	53%	42%	47%	26%
c-tkt-tkt	61%	26%	58%	42%	53%	26%
hmcs	61%	26%	37%	37%	37%	16%
hticket-ls	58%	32%	44%	38%	50%	50%
malth_spin	78%	43%	63%	53%	53%	53%
malth_stp	54%	38%	65%	60%	55%	55%
mcs-ls	78%	30%	63%	47%	58%	58%
mcs_spin	70%	26%	63%	53%	58%	58%
mcs_stp	67%	46%	70%	65%	70%	60%
mcs-timepub	42%	25%	65%	55%	50%	50%
partitioned	61%	26%	68%	47%	63%	47%
pthread	62%	50%	60%	55%	60%	55%
pthreadadapt	58%	38%	55%	50%	55%	50%
spinlock	65%	39%	68%	58%	63%	53%
spinlock-ls	57%	39%	58%	42%	58%	47%
ticket	74%	39%	79%	63%	74%	63%
ticket-ls	65%	39%	58%	47%	63%	47%
ttas	61%	35%	68%	53%	63%	58%
ttas-ls	87%	57%	78%	61%	74%	68%

Table 12: For each lock, at max nodes and at the optimized number of nodes, fraction of the applications for which the lock is harmful (**all machines**).

the 1-node, 2-node, 4-node and 8-node configurations.

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 and I-48 machines (see the companion technical report [18]).

Applications	% of pairwise changes between configurations			
	1/2	2/4	4/8	1/2/4/8
dedup	16%	6%	12%	19%
facesim	18%	38%	81%	95%
ferret	0%	74%	26%	87%
fluidanimate	5%	6%	24%	32%
fmm	33%	10%	19%	45%
histogram	19%	32%	24%	55%
linear_regression	58%	40%	57%	95%
matrix_multiply	16%	27%	45%	54%
mysqld	33%	20%	7%	40%
ocean_cp	54%	53%	72%	94%
ocean_ncp	52%	54%	56%	86%
pca	44%	60%	29%	89%
pca_ll	31%	38%	23%	73%
radiosity	11%	49%	65%	83%
radiosity_ll	66%	28%	14%	92%
s_raytrace	1%	70%	32%	96%
s_raytrace_ll	21%	69%	24%	99%
ssl_proxy	62%	12%	21%	78%
streamcluster	68%	21%	32%	88%
streamcluster_ll	60%	28%	31%	90%
vips	2%	3%	82%	82%
volrend	16%	27%	44%	85%
water_nsquared	23%	24%	13%	52%
water_spatial	12%	10%	10%	29%

Table 13: For each application, percentage of pairwise changes in the lock performance hierarchy when changing the number of nodes (**A-64 machine**).

Impact of the machine. Table 14 shows the number of pairwise lock inversions observed between the machines (both at max nodes and at the optimized number of nodes). More precisely, 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.

# nodes	A-64 vs. A-48	A-48 vs. I-48	A-64 vs. I-48
Max	38%	36%	38%
Opt	30%	29%	31%

Table 14: For each pair of machines, at max nodes and at opt nodes, percentage of pairwise changes in the lock performance hierarchy (**all machines**).

A note on Phtread locks. The various results presented in this paper show that the current Linux **Pthread locks perform well (i.e., are among the best locks) for a significant share of the studied applications**, thus providing a different insight than recent results, which were mostly based on synthetic workloads [9]. Beyond the changes of workloads, these differences may 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 adaptive locks perform slightly better than standard Pthread locks.

Impact of thread pinning. As explained in §3.2, all the above-described experiments were run without any restriction on the placement of threads, leaving the corresponding decisions to the Linux scheduler. However, in order to better control CPU allocation and improve locality, some developers and system administrators use pinning to explicitly restrict the placement of each thread to one or several core(s). The impact of thread pinning may vary greatly according to workloads and can yield both positive and negative effects [9, 27]. In order to assess the generality of our observations, we also performed the complete set of experiments 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 first N nodes of the machine. We chose thread-to-node pinning rather than thread-to-core pinning because we observed that the former generally provided better

performance for our studied applications, especially the ones using more threads than cores. The detailed results of our experiments with thread-to-node pinning are available in the companion technical report [18]. Overall, we observe that **all the conclusions presented in the paper still hold with per-node thread pinning**.

6 Related work

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 [1], liblock [25, 24, 26], and libslock [9]. 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 [3, 33, 36]. Moreover, among the above works, none of them provides a simple and generic solution for supporting Pthread condition variables. The authors of liblock [26] have proposed an approach 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.

Several research works have leveraged library interposition to compare different locking algorithms on legacy applications (e.g., Johnson et al. [21] and Dice et al. [14]) but, to the best of our knowledge, they have not publicly documented the design challenges to support arbitrary application patterns, nor disclosed the corresponding source code and the overhead of their interposition library has not been discussed.

Several studies have compared the performance of different multicore lock algorithms, either from a theoretical angle or based on experimental results [4, 33, 9, 24, 14]. In comparison, 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. [4] and Lozi et al. [26] but 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 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. Other synchronization-related studies like the one from Gramoli [16] have a different scope and focus on concurrent data structures, possibly based on other facilities than locks.

Finally, some tools have been proposed to facilitate the identification of locking bottlenecks in applications [35, 8, 26]. These publications are orthogonal to our work. We note that, among them, the profilers based on library interposition can be stacked on top of LiTL.

7 Conclusion and future work

Optimized lock algorithms for multicore machines are abundant. However, there are currently no clear guidelines and methodologies helping developers to select the right lock for their workloads. In this paper, we have presented a broad study of 27 locks algorithms with 35 applications on Linux/x86. To perform that study, we have implemented LiTL, an interposition library allowing the transparent replacement of lock algorithms used for Pthread mutex locks. From our study, we draw several conclusions, including the following ones: at its optimized contention level, no single lock dominates for more than 52% of the lock-sensitive applications; any of the locks is harmful for at least several applications; for a given application, the best lock varies according to both the number of contending cores and the machine that executes the application. These observations call for further research on optimized lock algorithms, as well as tools and dynamic approaches to better understand and control their behavior.

The source code of LiTL and the data sets of our experimental results are available online [17].

Acknowledgments

We thank the anonymous reviewers and our shepherd, Tim Harris, for their insightful comments on ealier drafts of this paper. 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. Finally, this work has been partially supported by: LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01), EmSoc Replicanos and AGIR CAEC projects of Université Grenoble-Alpes and GrenobleINP, and the INRIA/LIG Digitalis project.

References

[1] AL BAHRA, S. Concurrency Kit, 2015. `http://concurrencykit.org`.

[2] ANDERSON, T. E. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transaction on Parallel and Distributed Systems* (Jan. 1990), 6–16.

[3] AUSLANDER, M., EDELSOHN, D., KRIEGER, O., ROSENBURG, B., AND WISNIEWSKI, R. Enhancement to the MCS Lock for Increased Functionality and Improved Programmability. U.S. Patent Application Number 20030200457 (abandoned), October 2003.

[4] BOYD-WICKIZER, S., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. Non-scalable Locks are Dangerous. In *Proceedings of the Linux Symposium* (Ottawa, Canada, July 2012).

[5] CHABBI, M., FAGAN, M., AND MELLOR-CRUMMEY, J. High Performance Locks for Multi-level NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)* (2015), ACM.

[6] CHABBI, M., AND MELLOR-CRUMMEY, J. Contention-conscious, Locality-preserving Locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)* (2016), ACM.

[7] CRAIG, T. S. Building FIFO and Priority-Queuing Spin Locks from Atomic Swap. Tech. Rep. TR 93-02-02, University of Washington, 1993.

[8] DAVID, F., THOMAS, G., LAWALL, J., AND MULLER, G. 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* (2014), OOPSLA '14, ACM.

[9] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)* (2013), ACM.

[10] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. 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)* (2015), ACM.

[11] DICE, D. Brief Announcement: A Partitioned Ticket Lock. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)* (2011), ACM.

[12] DICE, D. Malthusian Locks, november 2015. `http://arxiv.org/abs/1511.06035`.

[13] DICE, D., MARATHE, V. J., AND SHAVIT, N. Flat-Combining NUMA Locks. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)* (2011), ACM.

[14] DICE, D., MARATHE, V. J., AND SHAVIT, N. Lock Cohorting: A General Technique for Designing NUMA Locks. *ACM Transactions on Parallel Computing 1*, 2 (Feb. 2015), 13:1–13:42.

[15] FATOUROU, P., AND KALLIMANIS, N. D. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)* (2012), ACM.

[16] GRAMOLI, V. 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'15)* (2015), ACM.

[17] GUIROUX, H., LACHAIZE, R., AND QUÉMA, V. LiTL source code and data sets, 2016. `https://github.com/multicore-locks`.

[18] GUIROUX, H., LACHAIZE, R., AND QUÉMA, V. Multicore Locks: the Case is not Closed Yet. Technical report, 2016. Available from `https://github.com/multicore-locks`.

[19] HE, B., SCHERER, W. N., AND SCOTT, M. L. Preemption Adaptivity in Time-published Queue-based Spin Locks. In *Proceedings of the 12th International Conference on High Performance Computing (HiPC'05)* (2005), Springer-Verlag.

[20] HENDLER, D., INCZE, I., SHAVIT, N., AND TZAFRIR, M. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)* (2010), ACM.

[21] JOHNSON, F. R., STOICA, R., AILAMAKI, A., AND MOWRY, T. C. Decoupling Contention Management from Scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)* (2010), ACM.

[22] KARLIN, A. R., LI, K., MANASSE, M. S., AND OWICKI, S. Empirical Studies of Competitve Spinning for a Shared-memory Multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP'91)* (1991), ACM.

[23] KYLHEKU, K. What is PTHREAD_MUTEX_ADAPTIVE_NP?, 2014. `http://stackoverflow.com/a/25168942`.

[24] LOZI, J.-P. *Towards More Scalable Mutual Exclusion for Multicore Architectures*. PhD thesis, UPMC, Paris, July 2014. `http://www.i3s.unice.fr/~jplozi/documents/lozi-phd-thesis.pdf`.

[25] LOZI, J.-P., DAVID, F., THOMAS, G., LAWALL, J., AND MULLER, G. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In *Proceedings of the 2012 USENIX Annual Technical Conference* (2012), USENIX Association.

[26] LOZI, J.-P., DAVID, F., THOMAS, G., LAWALL, J., AND MULLER, G. Fast and Portable Locking for Multicore Architectures. *ACM Transactions on Computer Systems 33*, 4 (Jan. 2016), 13:1–13:62.

[27] LOZI, J.-P., LEPERS, B., FUNSTON, J., GAUD, F., QUÉMA, V., AND FEDOROVA, A. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)* (2016), ACM.

[28] LUCHANGCO, V., NUSSBAUM, D., AND SHAVIT, N. A Hierarchical CLH Queue Lock. In *Proceedings of the 12th International Conference on Parallel Processing (Euro-Par'06)* (2006), Springer-Verlag.

[29] MAGNUSSON, P. S., LANDIN, A., AND HAGERSTEN, E. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing* (1994), IEEE Computer Society.

[30] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions on Computer Systems 9*, 1 (Feb. 1991), 21–65.

[31] OYAMA, Y., TAURA, K., AND YONEZAWA, A. 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)* (1999), World Scientific.

[32] RADOVIC, Z., AND HAGERSTEN, E. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)* (2003), IEEE Computer Society.

[33] SCOTT, M. L. *Shared-Memory Synchronization*. Morgan & Claypool Publishers, 2013.

[34] SCOTT, M. L., AND SCHERER, W. N. Scalable Queue-based Spin Locks with Timeout. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP'01)* (2001), ACM.

[35] TALLENT, N. R., MELLOR-CRUMMEY, J. M., AND PORTERFIELD, A. Analyzing Lock Contention in Multithreaded Applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)* (2010), ACM.

[36] WANG, T., CHABBI, M., AND KIMURA, H. Be My Guest — MCS Lock Now Welcomes Guests. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)* (2016), ACM.