QFSync: A Queue-Free Approach to Improve the Performance of Synchronization Primitives of Multithreaded Applications

Pierre Sens
Sorbonne Université, CNRS
LIP6, 4 Place Jussieu
75005 Paris, France
pierre.sens@lip6.fr

Luciana Arantes
Sorbonne Université, CNRS
LIP6, 4 Place Jussieu
75005 Paris, France
luciana.arantes@lip6.fr

Julien Sopena
Sorbonne Université, CNRS
LIP6, 4 Place Jussieu
75005 Paris, France
julien.sopena@lip6.fr

Abstract

Synchronization mechanisms such as locks and condition variables significantly impact the performance of parallel applications. While numerous algorithms have been proposed to implement these mechanisms, most rely on either thread spinning or system calls to manage waiting queues. Both approaches introduce inefficiencies: thread spinning leads to excessive CPU usage, while system calls incur costly context switches between user and kernel spaces, particularly in contended scenarios.

This paper presents QFSync, a novel queue-free synchronization approach aimed at reducing the number of system calls and eliminating unnecessary thread spinning. When a thread requires blocking, it is suspended for a short time interval using the *clock_nanosleep* system call. The thread is then awakened directly by a clock interrupt when the interval elapses.

We propose two algorithms for implementing POSIX mutex locks and condition variables using this technique. Extensive evaluations using microbenchmarks and parallel applications demonstrate that QFSync outperforms traditional POSIX implementations by significantly reducing the number of system calls.

Keywords:multi-core, synchronization, threads, parallel applications

1 Introduction

Multithreaded parallel applications on multicore machines rely on synchronization mechanisms to ensure correctness. The performance of these applications heavily depends on the latency of synchronization operations, which become increasingly critical as the number of cores and threads scales. Synchronization mechanisms serve various purposes, including controlling access to shared resources (e.g., mutex locks, semaphores), synchronizing thread execution (e.g., barriers, semaphores), and allowing threads to wait for specific conditions (e.g., condition variables).

Historically, these mechanisms were implemented in kernel space. While they provide simple programming primitives for the applications, their intensive use can significantly impact application performance due to costly context switching between user and kernel spaces [3, 12, 13]. This paper focuses on improving the efficiency of mutex (mutual exclusion locking) and condition variable synchronization mechanisms. Mutexes prevent multiple threads from accessing a shared resource (e.g. shared variable) simultaneously while condition variables allow threads to wait until a condition is met. The implementation of condition variables includes the use of mutexes.

To address these challenges, *futex* ("fast userspace mutex") was introduced in version 2.5.7 of the Linux kernel in 2003 to reduce the number of expensive system calls. Current POSIX implementations of mutex and condition variables use futex, which associates an atomic integer in user space with a kernel-space waiting queue. By using this integer, threads can synchronize themselves in user space, invoking system calls only when blocking is necessary. By minimizing interactions with the kernel, futexes are efficient in uncontended scenarios. However, when contention increases, futex-based synchronization mechanisms rely on system calls for blocking and unblocking concurrent threads.

This paper proposes a *queue-free synchronization* approach to further reduce interaction with the kernel in contended scenarios. We introduce two synchronization algorithms – one for mutex locks and another for condition variables – that eliminate the need for dedicated waiting queues. Instead, our approach leverages the existing thread waiting queues managed by the Linux scheduler. Our solution not only reduces the number of system calls but also prevents unnecessary and costly thread elections or CPU-intensive spinning of threads that should wait until synchronization conditions are met, particularly in contended scenarios. To this end, the calling thread is suspended for a short time interval using the *clock_nanosleep* system call. The thread is then awakened directly by the clock interrupt when the time expires, rather than a system call. By using nanosleep, only one system call is required to block a thread, significantly reducing the synchronization response time of applications.

The contributions of this paper are:

1. We introduce novel queue-free synchronization mechanisms that does not require dedicated waiting queues and reduces system calls overhead.

2. We propose efficient POSIX-compatible implementations of mutex locks and condition variables that use clock nanosleep system call.

3. Results from extensive evaluation experiments conducted on different multicore machines with several widely used multithreaded applications demonstrate that, in most cases, the queue-free synchronization approach can highly improve application performance.

The rest of this paper is organized as follows: Section 2 provides background on synchronization for multithreaded POSIX applications. Section 3 details our queue-free mutex and condition variables algorithms. Section 4 presents performance evaluations. Section 5 discusses related work, and Section 6 concludes the paper.

2 Background

Most POSIX multithreaded applications rely on mutual exclusion (mutex) locks and condition variables for synchronization. This section provides an overview of these fundamental mechanisms.

2.1 Mutex

A *mutex* object ensures the integrity of shared data required by multiple threads by allowing at most one thread to access it at a time. The application code that accesses shared data is called a critical section (cs). A mutex can be in one of two states: locked or unlocked. Various implementations exist, and numerous studies have proposed optimized locking algorithms for multicore architectures (see Section 5).

Regardless of the locking algorithm, the *waiting policy* –*i.e.*, how a thread waits when it cannot acquire the lock– plays a crucial role in performance. There are three main approaches for implementing this policy:

1. Blocking: The thread is suspended until the lock is available.

2. Spinning: The thread actively waits by repeatedly checking a variable in memory.

3. Hybrid (Spinning-then-park): The thread initially spins for a short period before blocking.

The *glibc* provides two implementations of pthread mutex.

POSIX default (futex-based) implementation The default mutex implementation in *glibc* uses futex-based synchronization (Algorithm 1). Each mutex variable is associated with an atomic futex integer. The algorithm uses atomic functions provided by *glibc* such as compare_and_swap (*cas*), test_and_set (*tas*), or fetch_and_sub (*fas*) to manage locking.

Initially, the futex value is set to 0, indicating that the shared data is available. The first thread acquiring the lock sets the futex to 1. Subsequent threads attempting to acquire the lock set the value to 2 and invoke *futex_wait* to block until the lock becomes available. When a process ends its critical section, it wakes up another thread only if the value is 2, avoiding unnecessary wake-ups.

Algorithm 1 POSIX mutex locking pseudo-code based on futex.

```
1:  mutex_lock(int futex)
2:      old = cas(futex, 0, 1);
3:      if old == 0 then
4:          return; // futex was idle and now is set to 1
5:      else
6:          old = tas(futex, 2); // set futex to 2
7:          while old != 0 do
8:              futex_wait(futex, 2); // Wait as long as futex
                is 2
9:              old = tas(futex, 2);

10: mutex_unlock(int futex)
11:     old = fas(futex, 1);
12:     if old==2 then
13:         old = tas(futex, 0); // Set futex to 0
14:         futex_wakeup(futex, 1); // Wakeup one thread
```

POSIX Adaptive implementation The second mutex implementation provided by *glibc* uses *adaptive spinning* and is enabled for mutexes initialized with the `PTHREAD_MUTEX_ADAPTIVE_NP` GNU extension. This adaptive mutex reduces kernel interaction by combining active spinning with blocking, following the *spinning-then-park* policy. Based on the previous number of spin loops, the mutex algorithm estimates how long the lock will be held by the current owner and performs up to 100 acquire attempts. If unsuccessful, the thread is blocked in a futex queue.

2.2 Condition variable

A condition variable allows threads to block execution until a specific condition on a shared data object (a predicate) is met. A condition variable is always used in conjunction with a mutex, ensuring safe access to the predicate.

To use a condition variable, a thread needs to lock a mutex for protecting the shared data and then verifies the predicate. If the predicate is not met, the thread waits on a condition variable associated with it. Waiting on the condition variable automatically unlocks the mutex. When another thread acquires the mutex and updates the data to a state that satisfies the predicate, it wakes up one or all waiting threads by calling *pthread_cond_signal* or *pthread_cond_broadcast* on the condition variable, respectively. Threads resume execution one at a time, since each must acquire the mutex lock.

POSIX condition variable semantics According to the POSIX specification, either *all* threads blocked on a condition variable *cond* at time t are unblocked by the first *pthread_cond_broadcast*(*cond*), called at time $t' > t$, or *a single* blocked thread is unblocked by the first *pthread_cond_signal*(*cond*), called at time $t' > t$. However, a call to *pthread_cond_signal* following *pthread_cond_broadcast* may override the broadcast, resulting in unpredictable behavior: some threads not yet awakened by the prior broadcast may remain blocked, since only one will be resumed by the signal call.

3 Queue-free mutex locking and condition variable algorithms

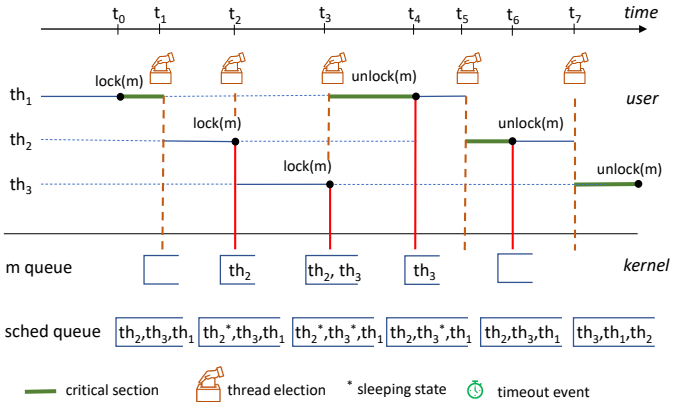
This section introduces two queue-free synchronization algorithms designed to minimize system calls and reduce unnecessary CPU spinning. We first present the principles of queue-free synchronization, followed by detailed descriptions of our mutex and condition variable implementations. We then provide a proof that our condition variable algorithm satisfies POSIX semantics.

3.1 Principles of queue-free synchronization

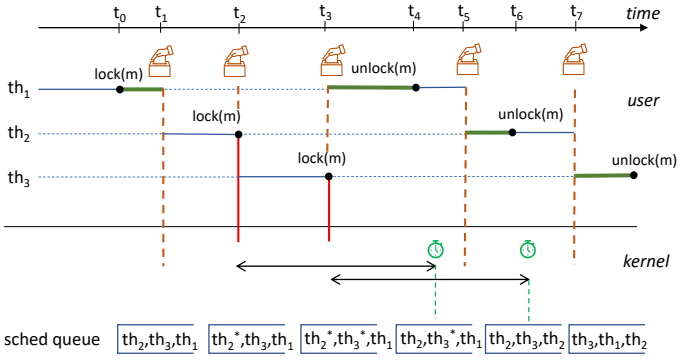
The aim is to reduce interactions with the kernel: rather than creating a dedicated lock queue in the kernel space, the proposal is to reuse already existing queues maintained by the kernel scheduler. Figure 1 illustrates this principle with three threads attempting to access the same shared variable, protected by a mutex lock m.

In the traditional approach (Figure 1a), a queue is defined in kernel space for m. When a thread t needs to wait, the *lock*(m) system call inserts t into m's queue and sets t to the sleeping state. To release m, the *unlock*(m) system call removes one of the threads from m'queue and sets it to the runnable state. This thread can then access the critical section the next time it is scheduled. In Figure 1a, at time t_0, thread th_1 directly acquires the lock m, since it is the first to request access to the critical section. At time t_1, th_2 is scheduled while th_1 is in its critical section and requests m at time t_2. A system call is then required to insert th_2 into the lock's queue and to switch it to the sleeping state. Similarly, thread th_3 is inserted in m' queue at time t_3. When th_1 releases the lock at time t_4, the unlock system call removes th_2 from the m's queue and switches th_2 to the runnable state. At time t_5, th_2 is scheduled and begins its critical section.

In the queue-free approach (Figure 1b), when a thread needs to wait, the lock system call only sets the thread to the sleeping state and requests the kernel to wake it up after a timer. When the timer expires, the kernel switches the thread to the runnable state without incurring the cost of an additional system call. The next time this thread is scheduled, if the lock is available, it can enter its critical section. In Figure 1b, at time t_2, the lock system call only sets a timer and switches th_2 to the sleeping state. Thread th_2 is directly woken up by the clock interrupt handler when the timer expires, and begins its critical section the next time it is scheduled at t_5.



(a) traditional queue lock algorithm.



(b) queue-free lock algorithm.

Figure 1: Principles of queue-free lock algorithms.

Another way to implement queue-free algorithms is to simply call the *yield* system call to release the CPU, when a thread needs to wait. However, since the waiting thread remains runnable, such an approach can significantly increase the number of context switches in a contended scenario, as we will discuss in Section 4.

3.2 Algorithms

Queue-free Mutex algorithm. Algorithm 2 presents the pseudo-code of our queue-free mutex lock algorithm based on the test and test-and-set (*ttas*) approach [2]. The latter relies on the CPU instruction $tas(v, true)$, which atomically sets v to *true* and returns its previous value, provided that the lock is not already set.

A boolean value and a timer are associated with each lock. While the lock is held (i.e., the boolean value is true), the calling thread invokes nanosleep system calls to yield the core to another thread, thus avoiding the CPU spinning (lines 10 and 11). As in the traditional *ttas* algorithm, if the value is already true, the algorithm avoids a call to the costly *tas* instruction (condition of line 10).

The timer value is dynamically adjusted during execution: it increases when the lock remains unavailable after nanosleep call (lines 13 and 14), and decreases when no sleep is required (lines 16 and 15). The increase and decrease functions are described in Section 3.3.

Algorithm 2 Test test-and-set mutex locking pseudo-code.

```
   Lock structure
 1: typedef struct {
 2:   bool val; // if true, the lock is held
 3:   int timer; // timer for nanosleep
 4: } lock_t;

 5:  init_lock(lock_t lck)
 6:    lck.val = false;
 7:    lck.timer = δ_min;

 8:  mutex_lock(lock_t lck)
 9:    int count_loop = 0;
10:    while lck.val == true || tas(lck.val,true) == true
       do
11:      nanosleep(lck.timer);
12:      count_loop++;
13:      if count_loop > 1 then
14:        increase_timer(lck.timer);
15:    if count_loop == 0 then
16:      decrease_timer(lck.timer);

17:  mutex_unlock(lock_t lck)
18:    lck.val = false;
```

Queue-free condition variable algorithm. Algorithm 3 presents the pseudo-code for our queue-free condition variable algorithm. Each condition has four fields:

- *ts* is a logical timestamp of the last signal or broadcast call on the condition.
- *flg* flag is set to `BCAST` or `SIG` when a broadcast or a signal is posted, respectively, and reset when no signal is pending.
- *pend* is the number of pending signals.
- *timer* is the current value of the sleep timer.

The algorithm uses *sync_add_and_fetch* and *sync_sub_and_fetch* glibc built-in functions which atomically perform addition and subtraction, respectively, and return the updated value. A global $CLOCK$ counter is atomically incremented each time a *cond_wait*, *cond_signal*, or *cond_broadcast* functions is called (lines 14, 31, 35).

Line 16 defines the wait condition for the *cond_wait* function. The caller must wait if (1) the function is invoked after the last signal or broadcast, or (2) no signal or broadcast is currently pending (i.e., *flg* in not set). The sleep timer is dynamically adjusted as in the mutex algorithm.

The *cond_signal* and *cond_broadcast* functions set the condition's flag (lines 29 and 33) and atomically increment the condition's timestamp (lines 31 and 35). As a result, any blocked thread *tb* exits the wait loop (lines 10-14): the wait condition becomes false because the flag is set and the condition's timestamp exceeds the timestamp of the *cond_wait* function stored in the *cl* variable. The flag tested at line 24 is true if an event has been posted on the condition. Otherwise, all pending signals have been already processed and *tb* restarts from the beginning of the waiting loop. If a signal has been sent, *tb* decrements the pending signal counter and resets the signal flag if there are no more pending signals (lines 26–27).

Algorithm 3 Condition variable synchronisation pseudo-code.

```
   Condition structure
 1: typedef struct {
 2:   int ts; // timestamp of the last cond wakeup call
 3:   int pend; // number of pending signals
 4:   int flg; // flags
 5:   int timer; // timer for nanosleep
 6: } cond_t;

 7:  init_cond(cond_t cond)
 8:    cond.ts = 0;
 9:    cond.pend = 0;
10:    cond.flg = 0;
11:    cond.timer = δ_min;

12:  cond_wait(cond_t cond, lock_t mutex)
     // mutex is assumed to be held before the call
13:    int count_loop = 0;
14:    int cl = sync_add_and_fetch(CLOCK,1);
15:    mutex_unlock(mutex);
16:    while  cl > cond.ts || cond.flg==0  do
17:      nanosleep(timer_cond);
18:      count_loop++;
19:      if count_loop > 1 then
20:        increase_timer(cond.timer);
21:    if count_loop == 0 then
22:      decrease_timer(cond.timer);
23:    mutex_lock(mutex);
24:    if cond.flg==0 then
25:      goto line 15
26:    if cond.flg==SIG &&
       sync_sub_and_fetch(cond.pend,1) == 0 then
27:      cond.flg = 0;

28:  cond_broadcast(cond_t cond)
29:    cond.flg = BCAST;
30:    cond.pend = 0;
31:    cond.ts = atomic_add_and_fetch(CLOCK,1);

32:  cond_signal(cond_t cond)
33:    cond.flg = SIG;
34:    atomic_add(cond.pend,1);
35:    cond.ts = atomic_add_and_fetch(CLOCK,1);
```

Figure 2 shows the execution of the algorithm considering three waiting threads (th1, th2, and th3) and two signaling threads (th4 and th5). Initially, $cond.ts = 0$, and the waiting threads are blocked in

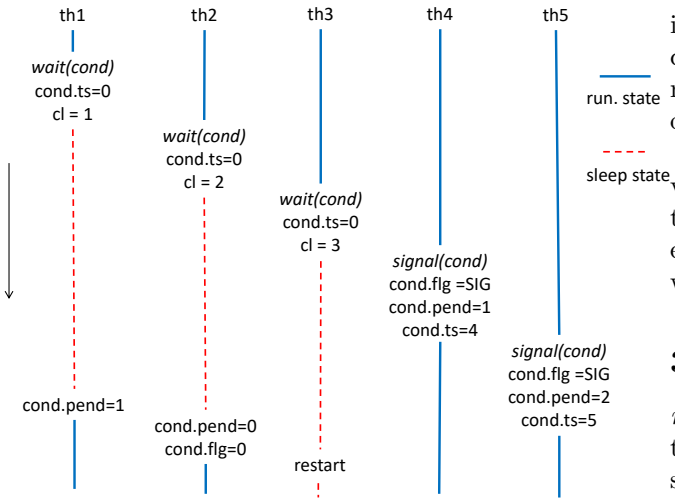


Figure 2: Example wait and signal.

the wait loop since $cond.flg = 0$.

After the two signal calls, all three waiting threads exit the loop because the condition in line 16 is false for each of them ($cl < cond.ts$ and $cond.flg \neq 0$). The first two scheduled threads, $th1$ and $th2$, decrement the pending signal counter, and flg is cleared by the second thread, $th2$. Then, the last waiting thread, $th3$, resumes execution at line 15 and remains in the wait loop until a new signal or broadcast is called, since $cond.flg = 0$.

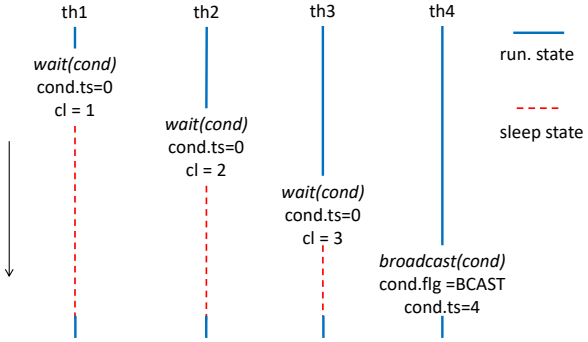


Figure 3: Example wait and broadcast.

Figure 3 illustrates the execution of the algorithm considering three waiting threads (th1, th2, and th3) and one broadcasting thread (th4). After the broadcast call, all three waiting threads exit the loop because the condition is false for each of them ($cl < cond.ts$ and $cond.flg \neq 0$). The threads then exit the *cond_wait* function and do not restart, since the condition flag has been set to `BCAST` and is therefore not cleared.

Fairness issue Test test-and-set (*ttas*) locks are commonly used to ensure mutual exclusion. However, a *ttas* lock does not guarantee fairness, since when the lock is released, any of the contending threads may acquire it, regardless of their arrival order. As there is no FIFO ordering among the threads competing for the lock, some threads may suffer from infinite starvation if others repeatedly succeed in acquiring the lock more quickly. In practice, however, once other threads stop requesting the lock, the waiting thread will eventually obtain it.

In our variable-condition algorithms, the CLOCK variable ensures that only threads already waiting for the lock before a signal or broadcast call can be awakened. Newly waiting threads are then served afterwards, thus avoiding starvation.

3.3 Timer setting

nanosleep call use the high-resolution timers (HRT) of the Linux kernel. Each lock or condition variable is associated with a timer value in nanoseconds, initialized to δ_{min}. When *nanosleep* is invoked, the thread will sleep for at least the current timer value plus a fixed delay defined by the kernel, *the timer slack*, which is around 50μsec in our configuration. This additional delay allows the kernel to coalesce wake-up events, especially when several timers expire concurrently.

The timer value is crucial to performance. Algorithm 4 shows how the timer is dynamically adjusted. If the timer is too short, the wait loop may lead to numerous calls to *nanosleep*, particularly in contented configurations where many threads compete to get the same lock. To reduce the number of loop iterations, the timer is dynamically increased exponentially: after each subsequent *nanosleep* call, the timer value is multiplied by a growth factor λ, until it reaches the maximum threshold δ_{max}.

Conversely, if the timer is too long, e.g., after a period of high contention, the calling thread may sleep excessively even though the lock is now available. To avoid such a behavior, a decrease function is invoked whenever a thread successfully acquires the lock without issuing a *nanosleep* call. This function reduces the timer by a decay factor τ, but only if the thread acquires the lock without waiting in β consecutive attempts.

Algorithm 4 Timer adjustment pseudo-code.

```
1: increase_timer(int timer)
2:     if timer < δ_max then
3:         timer = timer × λ
4: decrease_timer(int timer)
5:     if timer > δ_min && nonblocking_calls > β
       then
6:         timer = timer ÷ τ
```

3.4 Proof

We consider that time $T \in \mathbb{N}$ is characterized by the algorithm's $CLOCK$ variable.

Lemma 1. *A thread that enters cond_wait(c) function at time t will exit only after the end of the first cond_broadcast(c) call following t, assuming no cond_signal(c) call occurs between the two calls.*

Proof. Let c be a condition variable with $c.ts = x$ before time t. The timestamp $c.ts$ is set only by *cond_signal* or *cond_broadcast*, at lines 31 and 35 respectively. Let th be a thread invoking *cond_wait(c)* at time t. At line 14, $CLOCK$ is incremented, and cl is set to this new value. Since $c.ts$ was set to $CLOCK$ before t, we have $c.ts < cl$. Thus, the condition at line 16 holds, and th enters the wait loop.

The first *cond_broadcast(c)* after t sets $c.flg$ to $BCAST \neq 0$, increments $CLOCK$, and updates $c.ts$ to a value greater than cl. Consequently, th exits the wait loop and leaves *cond_wait* function unless $c.flg = 0$. However, *cond_broadcast(c)* sets the $c.flg$ flag at line 29 and $c.flg$ is cleared only at line 27 if the SIG flag is set by a *cond_signal(c)* call. Assuming no such call occurs before *cond_broadcast*, $c.flg$ remains set, and th leaves the *cond_wait* function. □

Lemma 2. *Among the threads blocked in the cond_wait(c) wait loop at time t, one will leave the cond_wait function after the first cond_signal(c) call following t, assuming no cond_broadcast(c) occurs in between.*

Proof. Let c be a condition variable with $c.ts = x$ and $c.pend = 0$, and consider the first *cond_signal* call at time $t' > t > x$. Any thread invoking *cond_wait(c)* between x and t sets $cl \in (x, t]$ and stays in the wait loop. At t', *cond_signal* sets $cs.ts = t'$, $cs.pend = 1$ and $cs.flg = SIG$. Subsequently, all waiting threads will leave the loop when next elected since $t' > t$. Among them, only one thread, tf, acquires the mutex lock, decrements $cs.pend$, clears $cs.flg$, and exists *cond_wait*. The other ones block on mutex at line 23. Once tf releases the mutex, the remaining threads one by one test $c.flg$, and whenever $c.flg = 0$ at line 15, they re-enter the wait loop. □

Theorem 1. *Algorithm 3 follows the semantic of POSIX condition variable.*

Proof. The proof is a direct consequence of lemma 1 and 2. □

4 Performance evaluation

This section describes the testbed where experiments were carried out, the metrics and applications used to evaluate the algorithms, and performance results. Using two single multithreaded applications as microbenchmarks, Sections 4.3 and 4.4 evaluate the performance of the queue-free mutex and condition variables algorithms respectively. Section 4.5 evaluates the performance of four multithreaded applications from well-known benchmarks that require mutex and/or condition variable synchronization.

4.1 POSIX implementation of mutex and conditions

The proposed locking and condition variable algorithms were implemented in a POSIX-compliant library providing the 5 functions *pthread_mutex_lock*, *pthread_mutex_unlock*, *pthread_cond_wait*, *pthread_cond_signal*, and *pthread_cond_broadcast*. These functions reuse the internal structure allocated by glibc 2.36 when condition and mutex lock variables are created. By using LD_PRELOAD, our library allows transparent interposition of pthread mutex lock and condition variables by replacing the 5 original functions implemented in glibc.

4.2 Application and system setup

Our testbed consists of two Linux-based x86 multicore nodes whose main characteristics are summarized in Table 1. The i9 machine is a non-NUMA machine used in our experiments to better understand the behavior of our algorithms without the interference of NUMA characteristics. The Xeon is a 64 NUMA machine.

Each point in figures is an average of 10 experiments.

Name	i9	iXeon
Processor	Intel i9 10900K	Intel Xeon Gold 6130
# cores	10	64
# vCPUs	20	128
# numa nodes	1	4
Memory	64 GiB	788 GiB
Linux kernel	6.12.0	5.10.0-19
Tick config.	Periodic (250 HZ) default Tickless (250 HZ) No idle tick (250 HZ) Periodic (100/1000 HZ)	No idle tick (250 HZ)

Table 1: Nodes configuration.

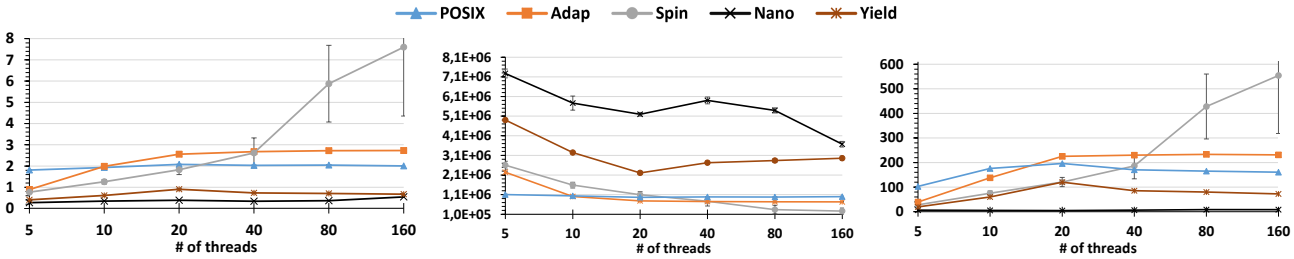
4.3 Locking evaluation

This section describes the performance results of our queue-free mutex implementation on a multithreaded application using a lock to protect a single shared counter. Each thread does the same number of iterations executing the critical section (cs) that modifies the counter. To increase the number of synchronization mechanism executions, there is no local computation outside the cs within the thread iterations.

We have considered the three following metrics:

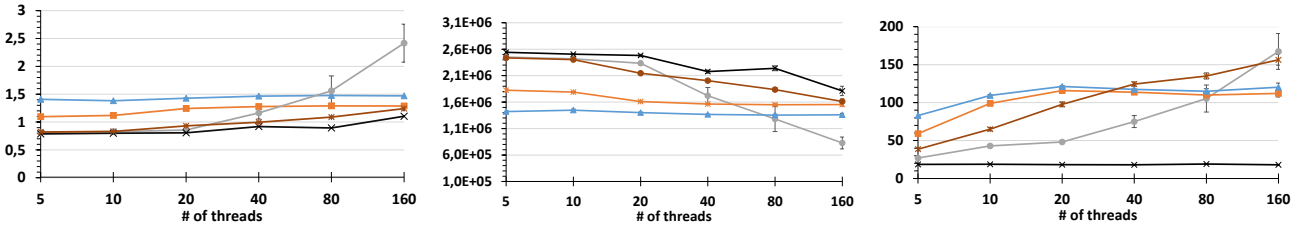
- application response time (in seconds);
- bandwidth (in locks/second);
- energy consumption (in Joule), obtained using the `perf` tool.

We compare our nanosleep queue-free mutex algorithm ('Nano' in the figures) and a version of it that uses sched_yield instead of nanosleep ('Yield') with both POSIX implementations of mutex in glibc 2.36, i.e. the default one with futex ('POSIX') and the POSIX Adaptative spinning ('Adap') based on spinning-then-park, as well as with spinlocks ('Spin').



(a) response time (sec.) (b) bandwidth (lock/sec) (c) energy (Joule)

Figure 4: Mutex lock on a single counter to obtain 20,000,000 cs overall (short cs time).



(a) response time (sec.) (b) bandwidth (lock/sec) (c) energy (Joule)

Figure 5: Mutex lock on a single counter to obtain 2,000,000 cs overall (long cs time).

4.3.1 Scalability

The experiments evaluate the performance of the mutex algorithms when the number of threads varies from 5 to 160 on the i9 20 vCPUs machine.

Figure 4 considers a short cs time: in cs, each thread only executes 3 basic operations on the shared counter. The total number of cs is 20,000,000.

As expected, the spinlock outperforms both POSIX mutex implementations only when the number of threads is less than the number of vCPUs. Moreover, increasing the number of threads does not affect the performance of the POSIX futex implementation.

Adaptive spinning has globally poor performance, mainly because of the overhead introduced by spinning steps before blocking. We also observe that both versions of our algorithm outperform all POSIX implementations. On average, the nanosleep version is 5.46 times faster than the default POSIX mutex. In terms of energy consumption, the nanosleep version consumes on average 26.05 fewer Joules than the POSIX futex implementation. It also outperforms the yield-based version by reducing the number of iterations in the wait loop. On average, the nanosleep version is 1.82 times faster than the yield one and consumes 11.86 fewer Joules.

In Figure 5, each critical section (*cs*) executes 100 times more operations on the counter than in the previous experiment. The total number of *cs* is 2,000,000. Compared to the previous configuration, we observe that the performance of Adaptive is now closer to the one of futex POSIX mutex, and that the cost of its spinning phase becomes negligible as the *cs* duration increases. Spinlock has a short response time up to 20 threads as each vCPU runs a single thread. At 40 threads or more, all vCPUs are running multiple threads and since the lock duration is longer, spinning from waiting threads on all vCPUs slows down the thread that currently holds the lock. The nanosleep implementation has the best response time but the gap is close to the two POSIX implementations: it is on average 1.64 faster than POSIX mutex. However, it consumes significantly less energy (on average 5.98 fewer Joules). This can be explained by a lower CPU *system* time, as illustrated by the results of Figure 6, whose experiments set the number of threads at 80.

System calls Tables 2 and 3 summarize the number of system calls for the 80-thread configuration for the same application of Figures 5 and 4, considering short and long *cs* time respectively. The number of system calls is captured directly within the kernel using the `bpftrace` tool, which enables fine-grained tracing without altering the application. We distinguish between successful and failed futex calls. In the latter case, the system call returns an error and does not block the calling thread.

As expected, our locking mechanism significantly reduces the number of system calls, by a factor of 1617 in the case of short *cs* duration and 403 in the case of long ones, compared to the default POSIX mutex. We also observe a high number of failed futex calls, which likely indicates race conditions on the lock: the userspace variable representing the lock state may have been reset just before the futex invocation, rendering the system call useless.

As the *cs* duration increases, the number of futex calls per *cs* rises significantly, even though, thanks to their spinning phase, Adaptive locks reduce the number of futex calls compared with default POSIX locks.

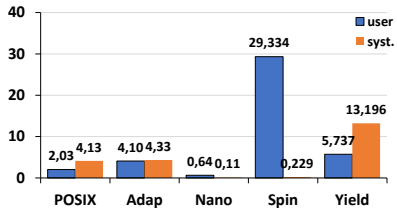


Figure 6: CPU times in sec. for 80 threads configuration to obtain 2,000,000 long critical section overall.

	POSIX	Adap	Nano
success futex	9,840,577	10,725,302	-
error futex	2,835,885	3,702,993	-
nanosleep	-	-	7,568
success futex/cs	0.4920	0.5363	-
futex/cs	0.6338	0.7214	-
nano/cs	-	-	0.000378
sys. calls	12,677,729	14,429,558	8,920

Table 2: Number of system calls for 80 threads configuration to obtain 2,000,000 cs overall (short cs time).

4.3.2 Impact of pinning in a NUMA architecture

The previous experiments were carried out on a non-NUMA architecture, letting the Linux scheduler place the threads. In NUMA configuration, the use of thread pinning could provide a better control on performance by improving localities. This section studies the performance of locking algorithms on a NUMA architecture and the impact of three pinning strategies: default Linux scheduler with no restriction to place threads (no pinning), NUMA aware, and Round-Robin (RR).

In NUMA aware pinning, a number of threads equal to the number of vCPUs is assigned sequentially to the NUMA nodes: considering that each NUMA node has 32 vCPUs, the first 32 threads are placed on the 32 vCPUs of NUMA node 0, then the next 32 threads are pinned on the vCPUs of node 1, and so on in a circular way. In RR pinning, the threads are spread on NUMA nodes: the first thread is placed on the first vCPUs of node 0, the second thread is on the first vCPUs of node 1 and so on in a circular way. NUMA aware aims to increase the locality whereas RR strategy to balance the load between NUMA nodes.

Figures 7 and 8 respectively show the response time and CPU time of the same application of Figure 5. The application runs on the iXeon machine with 4 NUMA nodes, each of them having 32 vCPUs (see Table 1). Considering the 3 pinning strategies, we have compared the performances of the default POSIX based on futexes, the Adaptive POSIX using spinning-then-park, and our mutex using nanospleep and yield.

We observe that both POSIX implementations are sensitive to the pinning strategy. As expected, by increasing the locality, the NUMA Aware is the best strategy and, at the opposite, the RR is the worst one. In default POSIX, the performances of pinning strategies converge to the same value when the number of threads increases, since greater than 128

	POSIX	Adap	Nano
success futex	47,816,089	45,174,170	-
error futex	8,493,502	7,840,753	-
nanosleep	-	-	130,168
success futex/cs	2.39	2.26	-
futex /cs	2.82	2.65	-
nano/cs	-	-	0.00651
sys. calls	56,310,858	53,016,189	131,522

Table 3: Number of system calls for 80 threads configuration to obtain 2,000,000 cs overall (long cs time).

all the vCPUs have several threads. More surprising, NUMA aware remains more efficient for Adaptive POSIX locks even when the number of threads increases. This behavior is probably due to the spinning step on the shared variable and the order in which Linux schedules the threads: due to thread creation order, it is more likely that spinning threads are located on the same NUMA node. On the other hand, we observe that our nanosleep locking outperforms the other algorithms and it is not sensitive to the pinning. The yield version is also not very sensitive to pinning, but is more unstable than nanosleep locking.

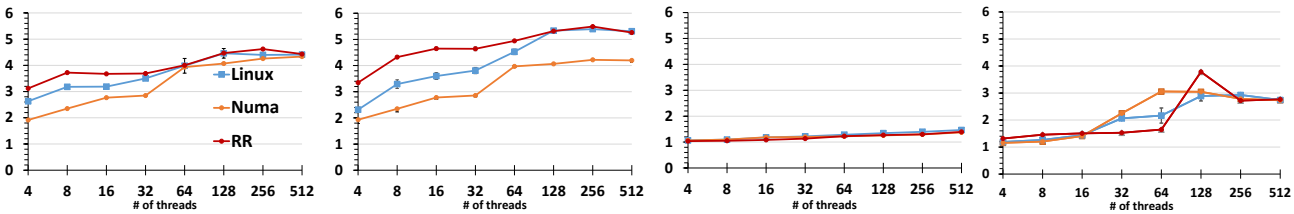
4.3.3 Impact of tick kernel policies

Linux kernel offers three scheduling-clock timer implementations [1]: periodic ticks, idle-tickless since 2.6.21 version, and full-tickless since 3.10 version. Timer strategies are configured during system boot according to kernel configuration.

In periodic tick, a constant tick frequency can be set at 100, 250, or 1000 HZ. In idle-tickless and full-tickless modes, the kernel avoids sending timer interrupts to idle CPUs and CPUs with a single runnable thread respectively.

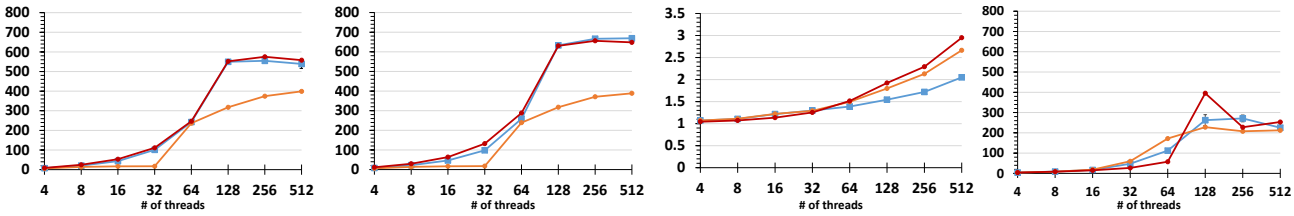
Figure 9 shows the impact of the timer strategy on the locking algorithms considering four kernel configurations on the i9 machine. The application executes 20,0000,000 critical sections, and the number of threads varies from 5 to 320. Although our nanosleep-based implementation outperforms the other algorithms (Figure 9c), it is more sensitive to the kernel tick configuration. As expected, higher frequency improves the response time of the nanosleep algorithm by reducing the waiting time when the shared resource is busy. We also observe that the Yield version becomes increasingly sensitive to the kernel configuration as the number of threads grows. Above 40 threads, all vCPUs host more than two threads, and the yield loop of waiting threads increases the load of all vCPUs, including the one currently running the lock-owning thread. This additional load, combined with higher tick frequencies, degrades the application response time.

More surprising, when the number of threads increases up to 320, the performance of the Yield version improves and converges towards that of the nanosleep version, except for the 1000hz kernel configuration which overloads the nodes. This behavior is due to the high number of runnable threads on each vCPU,



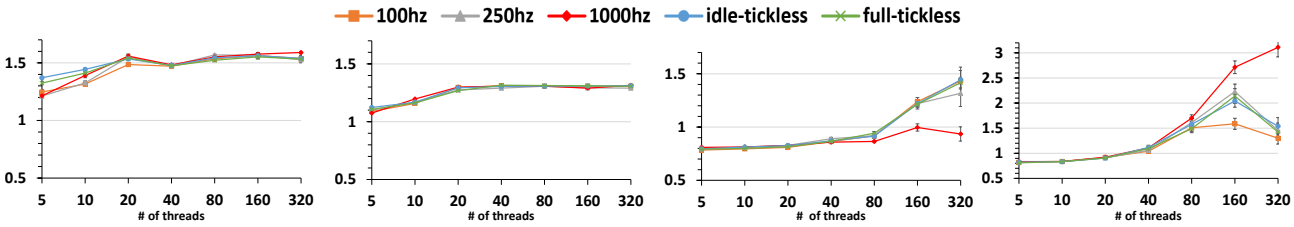
(a) POSIX mutex (b) Adaptive POSIX mutex (c) Nanosleep (d) Yield

Figure 7: Response time (sec.) to obtain 2,000,000 cs on iXeon NUMA machine.



(a) POSIX mutex (b) Adaptive POSIX mutex (c) Nanosleep (d) Yield

Figure 8: CPU time (sec.) to obtain 2,000,000 cs on the iXeon NUMA machine with three pinning strategies (Numa aware - NUMA, Round-Robin - RR , and no pinning - Linux).



(a) POSIX mutex (b) Adaptive POSIX mutex (c) Nanospleep (d) Yield

Figure 9: Response time (sec.) to obtain 20,000,000 cs with different kernel configurations (long cs time).

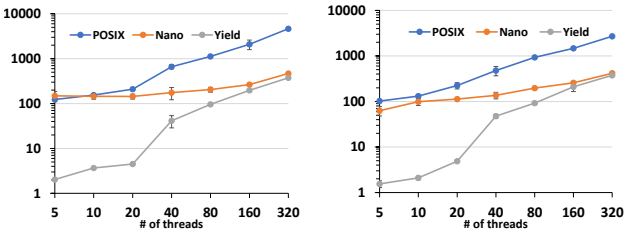
which increases the time between two elections of the same waiting thread, thereby reducing the total number of Yield calls.

4.4 Condition variable evaluation

We use a micro-benchmark to evaluate the wakeup time of our queue-free implementations of condition variable on the i9 machine. Figure 10 gives the time to wake up a set of threads blocked on the same condition variable using a simple broadcast call (Figure 10a) or a set of signal calls (Figure 10b). We observe that both queue-free implementations highly reduce the wakeup time compared to the POSIX implementation. The Yield version is the most efficient but it incurs an extra-cost CPU on non-blocking thread, as we have seen in the previous sections.

4.5 Evaluation on multi-threaded applications

We consider four multi-threaded applications, *lu*, *pca*, *radiosity*, and *raytrace* executed on the iXeon 128 vCPUs machine. The characteristics of these applications are summarized in Table 4. The number of threads varies from 8 to 256.



(a) broadcast (b) signal

Figure 10: Wakeup time (μsec.) of threads blocked on a condition variable.

Raytrace and radiosity are lock-intensive applications characterized by high lock contention, whereas pca and lu concern non-contended scenarios with fewer calls to synchronization functions. Pca does not use any condition variable.

4.5.1 Single application

Figure 11 compares the response times of applications running alone on the iXeon machine using default POSIX synchronizations with those of applications using our nanosleep algorithms.

For applications exhibiting high parallelism and low synchronization contention (lu and pca), we ob-

Applications	benchmark	# mutex	# cond	# mutex lock calls	# cond calls
lu	Splash-3 [24]	2 → 3	1	7734 → 263422	3605 → 131325
pca	Phoenix [22]	1	0	8008 → 8256	-
radiosity	Splash-3	1722 → 1780	1	54.56 M → 86.04 M	112 → 4590
raytrace	Splash-3	12 → 148	1	43.8 M	7 → 255

Table 4: Applications characteristics.

Applications	# mutex calls	POSIX				Nano		
		sys. calls	futex calls	futex error	futex/cs	sys. calls	clock_nano	nano/cs
lu	131,454	240,206	238,297	55,092	1.81	1,350,352	1,349,606	10.27
pca	8,128	14,332	9,568	4,344	1.18	9,385	4,318	0.53
radiosity	67,064,043	5,827,164	5,825,585	2,190,827	0.087	3,803,626	3,802,036	0.057
raytrace	43,790,428	38,560,960	38,560,225	15,760,105	0.881	1,350,352	1,349,606	0.033

Table 5: Applications system calls in 128 threads configuration.

serve the same behavior, as synchronization does not present a bottleneck. The short CPU system time, illustrated in Figures 12a and 12b, in both implementations confirms that few synchronization calls need to block the calling thread via system calls. Surprisingly, lu exhibits relatively high system time under the nanosleep version compared to prior observed results (e.g., with 256 threads, the nanosleep version consumes 10.38 seconds of CPU in the kernel versus 14.92 for POSIX). Even though lu performs few synchronizations, they probably last a long time, which increases the number of calls to nanosleep. Such a hypothesis is confirmed by the high number of nanosleep invocations shown in Table 5.

For high-contention applications (raytrace and radiosity), our implementation outperforms the POSIX one. On average, the response times of the nanosleep version is 1.44 times and 1.077 times lower than POSIX for raytrace and radiosity, respectively. The improvements are even more pronounced in terms of CPU usage, as shown in Figures 12c and 12d. Overall, the nanosleep implementation reduces CPU time by on average factor of 11.86, reaching up to 20.4 times for 128 threads for raytrace, and 4.27 times on average and up to 9.28 for radiosity. Such a behavior is due to the considerable amount of time spent in kernel mode, as illustrated in Figures 12c and 12d.

Table 5 reports the number of futex and nanosleep calls for the 128-thread configuration. For raytrace, we observe a high number of futex calls, indicating significant lock contention. On average,there are about 0.88 futex calls per *pthread_mutex_lock* / *pthread_mutex_unlock* call, while our nanosleep-based implementation has on average only 0.03 nanosleep calls per pthread_mutex_lock. Furthermore, the total number of system calls is reduced in both radiosity (1.53 times fewer calls) and pca (1.52 times fewer). Although radiosity performs numerous mutex lock calls, the results show low contention on the locks with only 0.087 futexes per *cs*, which explains the limited response time improvement under our nanosleep-based implementation.

4.5.2 Concurrent applications

In this experiment, we run raytrace and radiosity applications concurrently on the iXeon machine using the 128-thread configuration. Table 6 compares the average response time with that of each application running alone. As in the previous experiment, the nanosleep version outperforms the POSIX version and is less sensitive to concurrency.

Algorithms	App.	Conc.	Single	Conc./Single
Nano	Radiosity	16.35	13.14	1.24
	Raytrace	25.35	26.35	0.96
POSIX	Radiosity	18.70	14.18	1.32
	Raytrace	40.27	37.63	1.07

Table 6: Concurrent (Conc.) vs single response time (sec.) on the iXeon with 128 threads

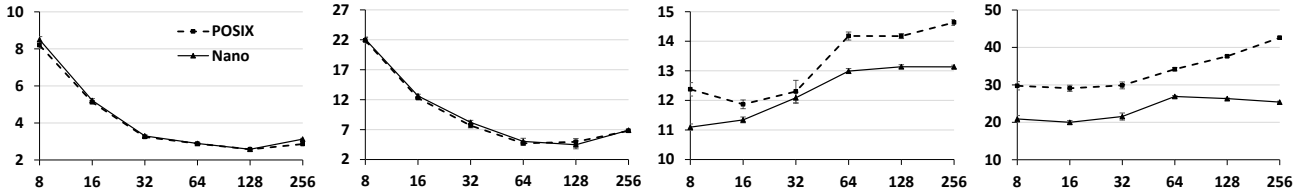
5 Related work

Many lock-based studies focus on optimizing algorithms for multicore architectures. In [12], the authors provide a comprehensive performance study of 28 state-of-the-art mutex lock algorithms, on 40 applications, on four different multicore machines.

Mutex lock Mutex lock algorithms can be classified as follows:

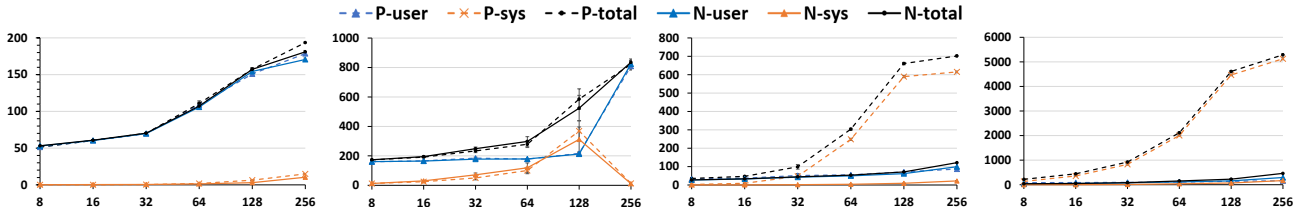
Global spinning algorithms are queue-free and rely on atomic instructions to spin on a shared lock variable. Simple spinlock, backoff spinlock, test and test-and-set (ttas) lock [2], and ticket lock [20] are well-known queue-free algorithms. While these algorithms are simple and have a very low latency in uncontended scenarios, they fail to scale because of the global spinning which significantly increases CPU usage in highly contended scenarios.

Local spinning algorithms such as MCS [25], CLH [19, 5] or Hemlock [9] improve performance under contention by minimizing cache-line movement of the lock variables between different CPU caches. MCS and CLH are queue-based: each thread inserts a record into a queue, and at most one thread busy waits at any time on its own element in the queue for MCS,



(a) lu (b) pca (c) radiosity (d) raytrace

Figure 11: Response time (sec.) of 4 applications.



(a) lu (b) pca (c) radiosity (d) raytrace

Figure 12: CPU usage (sec.) of 4 applications.

or on its predecessor's element for CLH. Hemlock improves MCS and CHL by avoiding the use of a queue, while still provides local spinning in most cases. To avoid sudden performance degradation as the number of threads increases, Malthusian Locks [6] limit the number of distinct threads requesting the lock by removing some threads from the queue.Both MCS and Hemlocks also require spinning during the unlock operation, as the unlocking thread must wait for it successor in the queue). In [10], the authors extend MCS to provide a wait-free unlock.

Delegation-based approaches improve cache locality of shared data accessed within critical sections. Waiting threads delegate the execution of their critical sections to a server thread running on a dedicated core. Ffwd [23] and RCL [18] use one or more dedicated servers, while in combining approaches [11], one of the threads acquires a global lock and executes not only its own critical section but also those of other threads. While these approaches avoid data movement by localizing shared-data access to a single core, they require significant application modifications to wrap critical sections. TCLocks [13] proposes a transparent delegation, where the lock holder transparently executes the critical section of the waiting thread without requiring changes to the application code.

Hierarchical approaches, such as ShflLock [14, 15] and CNA [8], aim to reduce cache-line contention on NUMA machines by reordering the waiting queue to avoid lock migrations among NUMA nodes.

Recently, FlexGuard [16] introduced a novel approach that dynamically switches from busy-waiting to blocking whenever a lock-holding thread is preempted by a waiting thread. To detect such preemption, FlexGuard leverages communication with the OS scheduler through eBPF.

Although local spinning, delegation-based, and dynamic switching approaches significantly improve performance, they do not aim to reduce the number of system calls, which becomes critical in contended scenarios. Furthermore, they do not provide algorithms for condition variable synchronization, commonly used in parallel applications.

Readers-writer locks Readers-writer locks enforce mutual exclusion only when at least one of the concurrent accesses is a write, thereby enhancing concurrency in read-only scenarios. Some work based on spinning aims to minimize contention on the lock variable [17, 4], while the authors of [7, 17, 21] investigate the impact of lock scheduling strategies.

6 Conclusion and future work

Synchronization overhead remains a critical bottleneck for high-performance concurrent applications on multicore systems. While futex-based implementations reduce unnecessary kernel interactions in uncontended scenarios, they still require costly system calls to manage waiting queues under contention.

This paper introduces a queue-free synchronization approach that eliminates dedicated waiting queues by leveraging the Linux scheduler's existing thread management. Threads suspend themselves using nanosleep and are awakened directly by clock interrupts rather than explicit wakeup system calls. This design reduces system calls by up to 1,617× in highly contended scenarios while maintaining POSIX compatibility. We provide implementations of mutex lock and condition variable. Extensive evaluations demonstrate significant improvements: our nanosleep-based mutex achieves 5.46× faster response time on microbenchmarks and up to 20.4× lower CPU usage on contention-intensive applications compared to standard POSIX implementations.

Acknowledgment

This work was supported by the French National Research Agency (ANR) under the project FrugalDinet, ANR-24-CE25-3083-04.

References

[1] Linux 3.10. NO_HZ: Reducing scheduling-clock ticks, https://www.kernel.org/doc/documentation/timers/no_hz.txt, 2018.

[2] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distributed Syst.*, 1(1):6–16, 1990.

[3] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Tappan Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 1–16. USENIX Association, 2010.

[4] Björn B. Brandenburg and James H. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real Time Syst.*, 46(1):25–87, 2010.

[5] Travis S. Craig. Building fifo and priority-queuing spin locks from atomic swap. 1993.

[6] Dave Dice. Malthusian locks. In Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic, editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 314–327. ACM, 2017.

[7] Dave Dice and Alex Kogan. BRAVO - biased locking for reader-writer locks. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 315–328. USENIX Association, 2019.

[8] Dave Dice and Alex Kogan. Compact numa-aware locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Dave Dice and Alex Kogan. Hemlock: Compact and scalable mutual exclusion. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, page 173–183, New York, NY, USA, 2021. Association for Computing Machinery.

[10] Rotem Dvir and Gadi Taubenfeld. Mutual exclusion algorithms with constant RMR complexity and wait-free exit code. In James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão, editors, *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, volume 95 of *LIPIcs*, pages 17:1–17:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[11] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 257–266, New York, NY, USA, 2012. Association for Computing Machinery.

[12] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. Multicore locks: The case is not closed yet. In Ajay Gulati and Hakim Weatherspoon, editors, *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 649–662. USENIX Association, 2016.

[13] Vishal Gupta, Kumar Kartikeya Dwivedi, Yugesh Kothari, Yueyang Pan, Diyu Zhou, and Sanidhya Kashyap. Ship your critical section, not your data: Enabling transparent delegation with TCLOCKS. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 1–16, Boston, MA, July 2023. USENIX Association.

[14] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and practical locking with shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 586–599, New York, NY, USA, 2019. Association for Computing Machinery.

[15] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and practical locking with shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 586–599, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Victor Laforet, Sanidhya Kashyap, Călin Iorgulescu, Julia Lawall, and Jean-Pierre Lozi. FlexGuard: Fast Mutual Exclusion Independent of Subscription. In *Proceedings of the 31st ACM Symposium on Operating Systems Principles (SOSP)*, Seoul, Korea, October 2025.

[17] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, page 101–110, New York, NY, USA, 2009. Association for Computing Machinery.

[18] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 6, USA, 2012. USENIX Association.

[19] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, page 165–171, USA, 1994. IEEE Computer Society.

[20] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[21] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Avoiding scheduler subversion using scheduler-cooperative locks. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[22] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[23] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 342–358, New York, NY, USA, 2017. Association for Computing Machinery.

[24] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016*, pages 101–111. IEEE Computer Society, 2016.

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