usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Unlocking Energy

Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis,
École Polytechnique Fédérale de Lausanne (EPFL)

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

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.

Unlocking Energy

Babak Falsafi *Rachid Guerraoui* *Javier Picorel* *Vasileios Trigonakis* [*]
{first.last}@epfl.ch
EcoCloud, EPFL

Abstract

Locks are a natural place for improving the energy efficiency of software systems. First, concurrent systems are mainstream and when their threads synchronize, they typically do it with locks. Second, locks are well-defined abstractions, hence changing the algorithm implementing them can be achieved without modifying the system. Third, some locking strategies consume more power than others, thus the strategy choice can have a real effect. Last but not least, as we show in this paper, improving the energy efficiency of locks goes hand in hand with improving their throughput. It is a win-win situation.

We make our case for this throughput/energy-efficiency correlation through a series of observations obtained from an exhaustive analysis of the energy efficiency of locks on two modern processors and six software systems: Memcached, MySQL, SQLite, RocksDB, HamsterDB, and Kyoto Kabinet. We propose simple lock-based techniques for improving the energy efficiency of these systems by 33% on average, driven by higher throughput, and without modifying the systems.

1 Introduction

For several decades, the main metric to measure the efficiency of computing systems has been *throughput*. This state of affairs started changing in the past few years as *energy* has become a very important factor [17]. Reducing the power consumption of systems is considered crucial today [26, 30]. Various studies estimate that datacenters have contributed over 2% of the total US electricity usage in 2010 [36], and project that the energy footprint of datacenters will double by 2020 [1].

Hardware techniques for reducing energy consumption include clock gating [38], power gating [52], as well as voltage and frequency scaling [28, 56]. Software techniques include approximation [16, 27, 57], consolidation [18, 21], energy-efficient data structures [23, 32], fast active-to-idle switching [44, 45], power-aware

[*]Author names appear in alphabetical order.

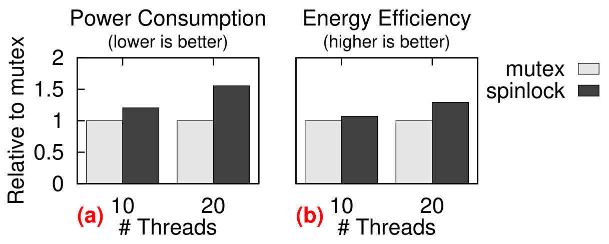

Figure 1: Power consumption and energy efficiency of `CopyOnWriteArrayList` with mutex and spinlock.

schedulers [48, 55, 59], and energy-oriented compilers [63, 64]. Basically, those techniques require changes in hardware, installing new schedulers or runtime systems, or even rebuilding the entire system.

We argue that there is an effective, complementary approach to saving energy: Optimizing *synchronization*, specifically its most popular form, namely *locking*. The rationale is the following. First, concurrent systems are now mainstream and need to synchronize their activities. In most cases, synchronization is achieved through locking. Hence, designing locking schemes that reduce energy consumption can affect many software systems. Second, the lock abstraction is well defined and one can usually replace the algorithm implementing it without any modification to the rest of the system. Third, the choice of the locking scheme can have a significant effect on energy consumption. Indeed, the main consequence of synchronization is having some threads *wait* for one another–an opportunity for saving energy.

To illustrate this opportunity, consider the average power consumption of two versions of a `java.util.concurrent.CopyOnWriteArrayList` [6] stress test over a long-running execution–Figure 1(a). The two versions differ in how the lock handles contention: Mutexes use *sleeping*, while spinlocks employ *busy waiting*. With sleeping, the waiting thread is put to sleep by the OS until the lock is released. With busy waiting, the thread remains active, polling the lock until the lock is finally released. Choosing sleeping as the *waiting strategy* brings up to 33% benefits on power. Hence, as we pointed out, the choice of locking strategy can have a significant effect on power consumption.

Accordingly, privileging sleeping with mutex locks seems like the systematic way to go. This choice, however, is not as simple as it looks. What really matters is not only the power consumption, but the amount of energy consumed for performing some work, namely *energy efficiency*. In the Figure 1 example, although the spinlock version consumes 50% more power than mutex, it delivers 25% higher energy efficiency (Figure 1(b)) for it achieves twice the throughput. Hence, indeed, locking is a natural place to look for saving energy. Yet, choosing the best lock algorithm is not straightforward.

To finalize the argument that optimizing locks is a good approach to improve the energy efficiency of systems, we need locks that not only reduce power, but also do not hurt throughput. Is that even possible?

We show that the answer to this question is positive. We argue for the POLY[1] conjecture: *Energy efficiency and throughput go hand in hand in the context of lock algorithms*. POLY suggests that we can optimize locks to improve energy efficiency without degrading throughput; the two go hand in hand. Consequently, we can apply prior throughput-oriented research on lock algorithms almost as is in the design of energy-efficient locks as well.

We argue for our POLY conjecture through a thorough analysis of the energy efficiency of locks on two modern Intel processors and six software systems (i.e., Memcached, MySQL, SQLite, RocksDB, HamsterDB, and Kyoto Kabinet). We conduct our analysis in three layers. We start by analyzing the hardware and software artifacts available for synchronization (e.g., pausing instructions, the Linux `futex` system calls). Then, we evaluate optimized variants of lock algorithms in terms of throughput and energy efficiency. Finally, we apply our results to the six software systems. We derive from our analysis the following observations that underlie POLY:

Busy waiting inherently hurts power consumption. With busy waiting, the underlying hardware context remains active. On Intel machines, for example, it is not practically feasible to reduce the power consumption of busy waiting. First, there is no power-friendly pause instruction to be used in busy-wait loops. The conventional way of reducing the cost of these loops, namely the x86 `pause` instruction, actually increases power consumption. Second, the `monitor/mwait` instructions require kernel-level privileges, thus using them in user space incurs high overheads. Third, traditional DVFS techniques for decreasing the voltage and frequency of the cores (hence lowering their power consumption) are too coarse-grained and too slow to use. Consequently, the power consumption of busy waiting can simply not be reduced. The only way is to look into sleeping.

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

Sleeping can indeed save power. Our Xeon server has approximately 55 Watts idle power and a max total power consumption of 206 Watts. Once a hardware context is active, it draws power, regardless of the type of work it executes. We can save this power if threads are put to sleep while waiting behind a busy lock. The OS can then put the core(s) in one of the low-power idle states [5]. Furthermore, when there are more software threads than hardware contexts in a system, sleeping is the only way to go in locks, because busy waiting kills throughput.

However, going to sleep hurts energy efficiency. The `futex` system call implements sleeping in Linux and is used by pthread mutex locks. In most realistic scenarios, the `futex`-call overheads offset the energy benefits of sleeping over busy waiting, if any, resulting in worse energy efficiency. Additionally, the spin-then-sleep policy of mutex is not tuned to account for these overheads. The mutex spins for up to a few hundred cycles before employing `futex`, while waking up a sleeping thread takes at least 7000 cycles. As a result, it is common that a thread makes the costly `futex` call to sleep, only to be immediately woken up, thus wasting both time and energy. We design MUTEXEE, an optimized version of mutex that takes the `futex` overheads into account.

Thus, some threads have to go to sleep for long. An unfair lock can put threads to sleep for long periods of time in the presence of high contention. Doing so results in lower power consumption, as fewer threads (hardware contexts) are active during the execution. In addition, lower fairness brings (i) better throughput, as the contention on the lock is decreased, and (ii) higher tail latencies, as the latency distribution of acquiring the lock might include some large values.

Overall, on current hardware, every power trade-off is also a throughput and a latency trade-off (motivating the name POLY[1]): (i) sleeping vs. busy waiting, (ii) busy waiting with vs. without DVFS or `monitor/mwait`, and (iii) low vs. high fairness.

Interestingly, in our quest to substantiate POLY, we optimize state-of-the-art locking techniques to increase the energy efficiency of our considered systems. We improve the systems by 33% on average, driven by a 31% increase in throughput. These improvements are either due to completely avoiding sleeping using spinlocks, or due to reducing the frequency of sleep/wake-up invocations using our new MUTEXEE scheme.

We conduct our analysis on two modern Intel platforms as they provide tools (i.e., RAPL interface [4]) for accurately measuring the energy consumption of the processor. Still, we believe that POLY holds on most modern multi-cores. On the one hand, without explicit hardware support, busy waiting on any multi-core exhibits similar behavior. On the other hand, `futex` implementations are alike regardless of the underlying platform, thus the overheads of sleeping will always be significant. However, should the hardware provide adequate tools for fine-grained energy optimizations in software, POLY might need to be revised. We discuss the topic further in §8.

In summary, the main contributions of this paper are:

- The POLY conjecture, stating that we can simply, yet effectively optimize lock-based synchronization to improve the energy efficiency of software systems.
- An extensive analysis of the energy efficiency of locks. The results of this analysis can be used to optimize lock algorithms for energy efficiency.
- Our lock libraries and benchmarks, available at: `https://lpd.epfl.ch/site/lockin`.
- MUTEXEE, an improved variant of pthread mutex lock. MUTEXEE delivers on average 28% higher energy efficiency than mutex on six modern systems.

It is worth noting that POLY might not seem surprising to a portion of the multi-core community. Yet, we believe it is important to clearly state POLY and quantify through a thorough analysis the reasons why it is valid on current hardware. As we discuss in §8, our results have important software and hardware ramifications.

The rest of the paper is organized as follows. In §2, we recall background notions regarding synchronization and energy efficiency. We describe our target platforms in §3 and explore techniques for reducing the power of synchronization in §4. We analyze in §5 the energy efficiency of locks and we use our results to improve various software systems in §6. We discuss related work in §7, and we conclude the paper in §8.

2 Background and Methodology

Lock-based Synchronization. Locks ensure mutual exclusion; only the holder of the lock can proceed with its execution. The remaining threads wait until the holder releases the lock. This waiting is implemented with either *sleeping* (*blocking*), or *busy waiting* (*spinning*) [49].

With sleeping, the thread is put in a per-lock wait queue and the hardware context is released to the OS. When the lock is released, the OS might wake up the thread. With busy waiting, threads remain active, polling the lock in a spin-wait loop.

Sleeping is employed by the pthread mutex lock (MUTEX). MUTEX builds on top of `futex` system calls, which allow a thread to wait for a value change on an address. MUTEX might first perform busy waiting for a limited amount of time and if the lock cannot be acquired, the thread makes the `futex` call.

The locks which use busy waiting are called *spinlocks*. There are several spinlock algorithms, such as test-and-set (TAS), test-and-test-and-set (TTAS), ticket-lock (TICKET) [46], MCS (MCS) [46], and CLH (CLH) [22]. Spinlocks mostly differ in their busy-waiting implementation. For example, TAS spins with an atomic operation, continuously trying to acquire the lock (*global spinning*). In contrast, all other spinlocks spin with a load until the lock becomes free and only then try to acquire the lock with an atomic operation (*local spinning*).

Energy Efficiency of Software. *Energy efficiency* represents the amount of work produced for a fixed amount of energy and can be defined as *throughput per power* (TPP, $\#operation/Joule$). Higher TPP represents a more energy-efficient execution. We use the terms energy efficiency and TPP interchangeably. Alternatively, energy efficiency can be defined as the energy spent on a single operation, namely *energy per operation* (EPO, $Joule/operation$). Note that TPP = 1/EPO.

Experimental Methodology. We prefer TPP over EPO because both throughput and TPP are "higher-is-better" metrics. Recent Intel processors include the RAPL [4] interface for accurately measuring energy consumption. RAPL provides counters for measuring the cores', package, and DRAM energy. We use these energy measurements to calculate average power. Our microbenchmark results are the median of 11 repetitions of 10 seconds. When we vary the number of threads, we first use the cores within a socket, then the cores of the second socket, and finally, the hyper-threads.

3 Target Platforms

We describe our two target platforms and then estimate their maximum power consumption.

Platforms. We use two modern Intel processors:

Name	Type	#Cores	L1	L2	LLC	Mem	TDP
Xeon	server	10	32KB	256KB	25MB	128GB	115W
Core-i7	desktop	4	32KB	256KB	8MB	16GB	77W

In the interest of space and clarity of explanation, we focus in the paper on the results of our server. Note that the results on Core-i7 are in accordance with the ones on Xeon. Our server is a two-socket Intel Ivy Bridge (E5-2680 v2), henceforth called *Xeon*. Xeon runs on frequencies scaling from 1.2 to 2.8 GHz due to DVFS and uses the Linux kernel 3.13 and glibc 2.13. Our desktop is an Intel Core i7 (Ivy Bridge–3770K) processor, henceforth called *Core-i7*. Core-i7 runs on frequencies scaling from 1.6 to 3.5 GHz due to DVFS and runs the Linux kernel 3.2 and glibc 2.15. Both platforms have two hardware threads per-core (hyper-threads in Intel's terminology). Intel turbo boost is disabled for all the experiments.

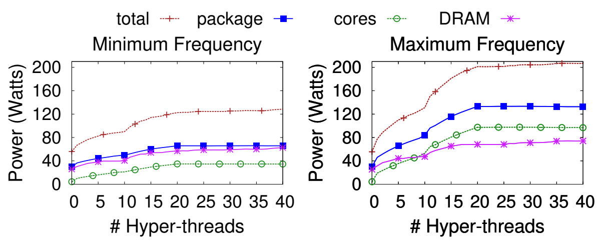

Figure 2: Power-consumption breakdown on Xeon.

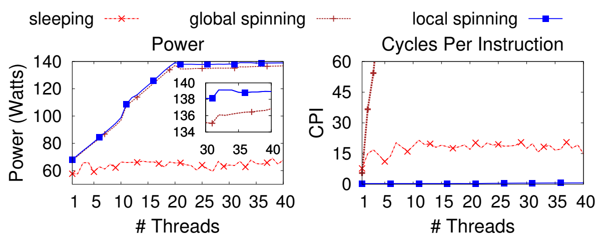

Figure 3: Power consumption and CPI while waiting.

3.1 Estimating Max Power Consumption

We estimate the maximum power that Xeon can consume, using a memory-intensive benchmark that consists of threads sequentially accessing large chunks of memory from their local node. Figure 2 depicts the total power and the power of different components on Xeon, depending on the number of active hyper-threads and the voltage-frequency (VF) setting.

Idle Power Consumption. The 0-thread points represent the idle power consumption, which accounts for the static power in cores and caches, and DRAM background power, and is the power that is consumed when all cores are inactive.[2] In both min and max frequency settings the total idle power is 55.5 Watts as the VF setting only affects the active power.

Power of Active Cores. Activating the first core of a socket is more expensive than activating any subsequent due to the activation of the uncore package components. In particular, it costs 6.4 and 13.6 Watts in package power on the min and max VF settings, respectively. The second core costs 2.3 and 5.6 Watts. We perform more experiments (not shown in the graphs) with data sets that fit in L1, L2, and LLC. The results show that the package power is not vastly reduced on any of these workloads, indicating that once a core is active, the core consumes a certain amount of power that cannot be avoided.

Attribution of Power to Cores, Package, and Memory. Notice the breakdown of total power to package/core[3] and DRAM power. DRAM power has a smaller dynamic range than package and core power. On the max VF setting, DRAM power ranges from 25 to 74 Watts, while the range of package power is from 30 to 132 Watts, and core power from 4 up to 96 Watts.

Implications. The power consumption of Xeon ranges from 55 up to 206 Watts. Out of the 206 Watts, 74 Watts are spent on the DRAM memory. Locks are typically transferred within the processor by the cache-coherence protocol, thus limiting the opportunities for reducing power to package power (30-132 Watts). Additionally, once a core is active, the core draws power, regardless of the type of work performed. Consequently, the opportunity for reducing power consumption in software is relatively low and mostly has to do with (i) using fewer cores, by, for example, putting threads to sleep, or (ii) reducing the frequency of a core.

[2]Still, the OS briefly enables a few cores during the measurements.

[3]The package power includes the core power.

4 Reducing Power in Synchronization

In this section, we evaluate the costs of busy waiting and sleeping, and examine different ways of reducing them.

4.1 Power: The Price of Busy Waiting

We measure the total power consumption of the three main waiting techniques (i.e., sleeping, global spinning, and local spinning–see §2) when all threads are waiting for a lock that is never released. Figure 3 shows the power consumption and the cycles per instruction (CPI). CPI represents the average number of CPU cycles that an instruction takes to execute.

Two main points stand out. First, in this extreme scenario, sleeping is very efficient because the waiting threads do not consume any CPU resources. Second, local spinning consumes up to 3% more power than global spinning. This behavior is explained by the CPI graph: Global spinning performs atomic operations on the shared memory address of the lock, resulting in a very high CPI (up to 530 cycles). In local spinning, every thread executes an L1 load each cycle, whereas, in global spinning, storing over coherence occurs once the atomic operation is performed, each 530 cycles on average.

4.2 Reducing the Price of Busy Waiting

We reduce the power consumption of busy waiting in different ways: (i) we examine various ways of pausing in spin-wait loops, (ii) we employ DVFS, and (iii) we use `monitor/mwait` to “block” the waiting threads.

Pausing Techniques. Busy waiting with local spinning is power hungry, because threads execute with low CPI. Hence, to reduce power, we must increase the loop’s CPI. We take several approaches to this end (Figure 4).

Any instruction, such as a `nop`, that the out-of-order core can hide, cannot reduce the power of the spin loop. According to Intel’s Software Developer’s Manual [4], “*Inserting a pause instruction in a spin-wait loop greatly reduces the processor’s power consumption.*” A `pause` (*local-pause*) increases CPI to 4.6. However, not only

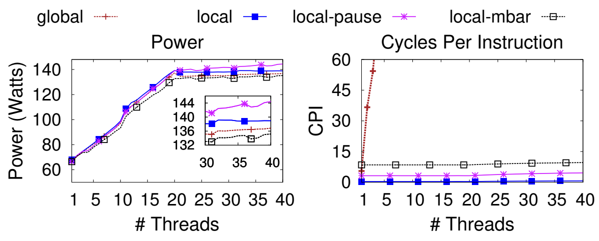

Figure 4: Power consumption and CPI while spinning.

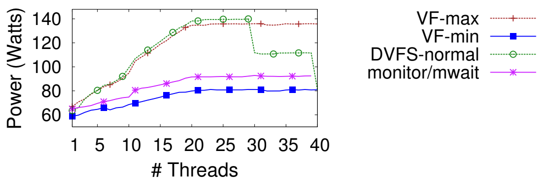

Figure 5: Power consumption of busy waiting using DVFS and `monitor/mwait`.

does it not "greatly reduce" power, but it even increases the power consumption by up to 4%.[4]

In general, the reason behind the very low CPI of local spinning is the aggressive execution mechanisms of modern processors that allow instructions to execute both speculatively and out of the program order. This results in one out of three of the retired operations being a memory load (the other two are a test and a conditional jump). Without appropriate pausing, the spin loop retires one memory load per cycle.

A way to avoid the speculative execution of the load is to insert a full, or a load, memory barrier. That way, the loads only execute once the previous load retires and the instructions that depend on it, test and jump, are stalled as well. The results (*local-mbar*) show that the barrier reduces the power consumption of local spinning to the point that becomes less expensive than global spinning (*global*). Additionally, *local-mbar* consumes up to 7% less power than *local-pause*. It is worth noting that *local-mbar* consumes less power than *local-pause* even for low thread counts (e.g., 5% on 10 threads). In the rest of the paper, we use a memory barrier for pausing in spin loops.

Dynamic Voltage and Frequency Scaling (DVFS). An intuitive way of lowering the power consumption of an active core is to reduce the voltage-frequency (VF) point via DVFS (see §3). Figure 5 shows that spinning on *VF-min* consumes up to 1.7x less power than on the *VF-max* setting. Still, DVFS is currently impractical for dynamically reducing power in busy waiting.

First, to trigger the VF change with DVFS, we need to write on a certain per-hardware context file of the `/sys/devices` directory (more details about DVFS can be found in [62]). Hence, the VF-switch operation is slow: We measure that it takes 5300 cycles on Xeon. If DVFS is used while busy waiting, this overhead will be on the critical path when the lock is acquired and the thread must switch back to the maximum VF point.

Second, both hyper-threads of a physical core share the same VF setting–the higher of the two. If a hyper-thread lowers its VF setting, the action will have no effect unless the second hyper-thread has the same or lower VF setting. Consequently, using DVFS with hyper-threading is tricky, and as the *DVFS-normal* line shows, the power consumption drops only when both hyper-threads lower their VF points.

Monitor/mwait. The `monitor/mwait` [4] instructions allow a hardware context to block and enter an implementation-dependent optimized state while waiting for a store event. In detail, `monitor` allows a thread to declare a memory range to monitor. The hardware thread then uses `mwait` to enter an optimized state until a store is performed within the address range. Essentially, `mwait` implements sleeping in hardware and can be used in spin-wait loops: The hardware sleeps, yet the thread does not release its context.

These instructions require kernel privileges. We develop a virtual device and overload its *file_operations* functions to allow a user program to declare and wait on an address, similar to [14]. A thread can wake up others with a user-level store. However, threads pay the user-to-kernel switch and system-call overheads for waiting.

Figure 5 includes the power of busy waiting with `monitor/mwait`. These instructions can reduce power consumption over conventional spinning up to 1.5x. However, similarly to DVFS, using `monitor/mwait` has two shortcomings. First, `monitor/mwait` can be only used in kernel space. The overloaded file operation takes roughly 700 cycles. The best case wake-up latency from `mwait`, with just one core "sleeping," is 1600 cycles. In comparison, "waking up" a locally-spinning thread takes two cache-line transfers (i.e., 280 cycles). Second, programming with `monitor/mwait` on Intel processors can be elaborate and limiting. The `mwait` instruction blocks the hardware context until the thread is awaken. In oversubscribed environments (i.e., more threads than hardware contexts), `monitor/mwait` will likely exacerbate the "livelock" issues of spinlocks (see §6). Blocked threads might occupy most hardware contexts, thus preventing other threads from performing useful work.

Implications. Busy waiting drains a lot of power because cores execute at full speed. Neither of the two platforms provides sufficient tools for reducing power consumption in a systematic way. Pausing techniques, such as `pause`, can even increase the power of busy waiting.

[4]We speculate that one of the reasons for this increase in power is that `pause` gives a hint to the core to prioritize the other hyper-thread.

Techniques that can significantly reduce power, such as DVFS and `monitor/mwait`, are not designed for user-space usage as they require expensive kernel operations. Hence, sleeping is currently the only practical way of reducing the power consumption in locks.

4.3 Latency: The Price of Sleeping

In Linux, sleeping is implemented with `futex` system calls. A `futex`-sleep call puts the thread to sleep on a given memory address. A `futex` wake-up call awakes the first N threads sleeping on an address ($N = 1$ in locks). The `futex` calls are protected by kernel locks. In particular, the kernel holds a hash table (array) of locks and `futex` operations calculate the particular lock to use by hashing the address. Given that the array is large (approximately $256 * \#cores$ locks), the probability of false contention is low. However, operations on the same address (same MUTEX) do contend on kernel level.

We use a microbenchmark where two threads run in lock-step execution (synchronized at each round with barriers). One makes `futex`-sleep calls and the second makes wake-up calls on the same `futex`, after waiting for some time. A `futex`-sleep call (i.e., enqueuing behind the lock and descheduling the thread) takes around 2100 cycles.[5] This sleep latency is not necessarily on the critical path: The thread sleeps because the lock is occupied. However, the latency to wake up a thread and the one for the woken-up thread to be ready to execute are on the critical path. Figure 6 contains the wake-up call and the turnaround latencies, depending on the delay between the invocation of the sleep call and the wake-up call. The turnaround latency is the time from the wake-up invocation until the woken-up thread is running.[6]

The turnaround time is at least 7000 cycles and is higher than the wake-up call latency. Apart from the approximately 2700 cycles of the wake-up call, the woken-up thread requires at least 4000 more cycles before executing. Concretely, once the wake-up call finishes, the woken-up thread pays the cost of idle-to-active switching and the cost of scheduling.[7]

Figure 6 further includes two interesting points. First, for low delays between the two calls, the wake-up call is more expensive as it waits behind a kernel lock for the completion of the sleep call. Second, when the delay between the calls is very large (>600K cycles), the turnaround latency explodes, because the hardware context sleeps in a deeper idle state [41].

Finally, the results in Figure 6 use just two threads and thus represent the best-case latencies, with minimal or no contention at the kernel level. With more threads, a wake-up invocation is likely to contend with `futex` sleep calls, all serialized using a single kernel lock.

[5]Estimated as the required delay between sleep and wake-up calls for the wake-up calls to almost always find the other thread sleeping.

[6]The wake-up call latency is directly measured in our microbenchmark, while the turnaround time is estimated as the duration of the sleep call, reduced by the delay between the sleep and wake-up calls.

[7]When the core is constantly active due to multiprogramming, the turnaround latency only includes the scheduling delays.

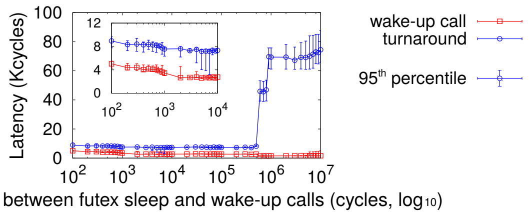

Figure 6: Latency of different `futex` operations.

Implications. `futex` operations have high latencies and consume energy, as a non-negligible number of instructions are executed. Handing over a lock with a `futex` wake-up call requires at least 7000 cycles. Even on rather lengthy critical sections (e.g., 10000 cycles), this latency is prohibitive; it almost doubles the execution time of the critical section. In this case, the energy benefits of sleeping will not easily compensate the performance losses. In short critical sections, invoking `futex` calls will have detrimental effects on performance.

4.4 Reducing the Price of Sleeping

Sleeping can save energy on long waiting durations. We estimate when sleeping reduces power consumption with two threads:

Period between wake-up calls (cycles)	1024	2048	4096	8192
Power (Watts)	72.03	69.18	68.75	68.02

The first thread sleeps on a location, while the second periodically wakes up the first thread. We vary the period between the wake-up invocations, which essentially represents the critical-section duration in locks. The results confirm that if a thread is woken up more frequently than the `futex`-sleep latency, power consumption is not reduced. The thread goes to sleep only to be immediately woken up by a concurrent wake-up call. When these "sleep misses" happen, we lose performance without any power reduction. Once the delay becomes larger than the sleep latency (i.e., approximately 2100 cycles on Xeon), we start observing power reductions.

Reducing Fairness. We show two problems with `futex`-based sleeping: (i) high turnaround latencies, and (ii) frequent sleeps and wake ups do not reduce power consumption. To fix both problems simultaneously, we recognize the following trade-off: We can let some threads sleep for long periods, while the rest coordinate with busy waiting. If the communication is mostly done via busy waiting, we almost remove the `futex` wake-up calls from the critical path. Additionally, we let threads sleep for long periods, a requirement for reducing power consumption in sleeping.

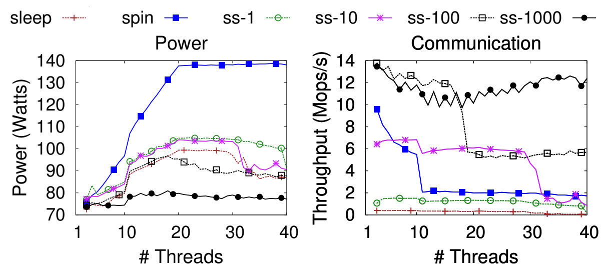

Figure 7: Power and communication throughput of sleeping, spinning, and spin-then-sleep for various Ts.[8]

This optimization comes at the expense of fairness. The longer a thread sleeps while some others progress, the more unfair the lock becomes. We experiment with the extreme case where only two threads communicate via busy waiting, while the rest sleep. Each active thread has a "quota" T of busy-waiting repetitions, after which it wakes up another thread to take its turn. Figure 7 shows the power and the communication rate (similar to a lock handover) of sleeping, busy waiting, and spin-then-sleep (*ss*-T) with various Ts on a single `futex`. T is the ratio of busy-waiting over `futex` handovers.

Figure 7 clearly shows that the more unfair an execution (i.e., for large Ts), the better the energy efficiency. First, larger values of T result in lower power, because the sleep and wake-up `futex` calls become infrequent, hence the sleeping threads sleep for a long duration. For example, on 10 threads with $T = 1000$, threads sleep for about 2M cycles. In comparison, with only sleeping, the sleep duration is less than 90000 cycles. Second, spin handovers face minimal contention, as only two threads attempt to "acquire" the cache line. Consequently, because most handovers (99.9%) happen with spinning, the latency is very low, resulting in high throughput.

Implications. Frequent `futex` calls will hurt the energy efficiency of a lock. A way around this problem is to reduce lock fairness in the face of high contention, by letting only a few threads use the lock as a spinlock, while the remaining threads are asleep.

5 Energy Efficiency of Locks

We evaluate the behavior of various locks in terms of energy efficiency and throughput, heavily relying on the results of §4. We first introduce MUTEXEE, an optimized version of MUTEX.

5.1 MUTEXEE: An Optimized MUTEX Lock

In §4, we analyze the overhead of `futex` calls. Additionally, we show how we can trade fairness for energy efficiency. MUTEX does not explicitly take these trade-offs into account, although it is an unfair lock.

	MUTEX	MUTEXEE
lock	for up to ∼ **1000 cycles** spin with `pause`	for up to ∼ **8000 cycles** spin with `mfence`
	if still busy, sleep with `futex`	if still busy, sleep with `futex`
unlock	release in user space (`lock->locked = 0`)	release in user space (`lock->locked = 0`)
		wait in user space
	wake up a thread with `futex`	wake up a thread with `futex`

Table 1: Differences between MUTEX and MUTEXEE.

In particular, MUTEX by default attempts to acquire the lock once before employing `futex`. MUTEX can be configured (with the `PTHREAD_MUTEX_ADAPTIVE_NP` initialization attribute) to perform up to 100 acquire attempts before sleeping with `futex`.[9] Still, threads spin up to a few hundred cycles on the lock before sleeping with `futex` (the exact duration depends on the contention on the cache line of the lock). This behavior can result in very poor performance for critical sections of up to 4000 cycles. In brief, threads are put to sleep, although the queuing time behind the lock is less than the `futex`-sleep latency. Additionally, to release a lock, MUTEX first sets the lock to "free" in user space and then wakes up one sleeping thread (if any). However, a third concurrent thread can acquire the lock before the newly awaken thread T_{aw} is ready to execute. T_{aw} will then find the lock occupied and sleep again, thus wasting energy, creating unnecessary contention, and breaking lock fairness.

To fix these two shortcomings, we design an optimized version of MUTEX, called MUTEXEE. Table 1 details how MUTEXEE differs from the traditional MUTEX. The "wait in user space" step of unlock requires further explanation. MUTEXEE, after releasing the lock in user space, but before invoking `futex`, waits for a short period to detect whether the lock is acquired by another thread in user space. In such case, the unlock operation returns without invoking `futex`. The waiting duration must be proportional to the maximum coherence latency of the processor (e.g., 384 cycles on Xeon).

Moreover, MUTEXEE operates in one of two modes: (i) *spin*, with ∼ 8000 cycles of spinning in the lock function and ∼ 384 in unlock, and (ii) *mutex*, with ∼ 256 cycles in lock and ∼ 128 in unlock (used to avoid useless spinning). MUTEXEE keeps track of statistics regarding how many handovers occur with busy waiting and with `futex`. Based on those statistics, MUTEXEE periodically decides on which mode to operate in: If the `futex`-to-busy-waiting handovers ratio is high (>30%), MUTEXEE uses mutex, otherwise it remains in spin mode.

[8] The performance collapse of *spin* is due to contention, while of *ss*-10 and *ss*-100 due to the high idle-to-active switching costs (see Figure 6).

[9] For brevity, in our graphs we show the default MUTEX configuration (i.e., without `PTHREAD_MUTEX_ADAPTIVE_NP`). We choose the default MUTEX version because: (i) it is the default in our systems (§6), and (ii) we thoroughly compare the two versions and conclude that for most configurations MUTEX is slightly faster without the adaptive attribute.

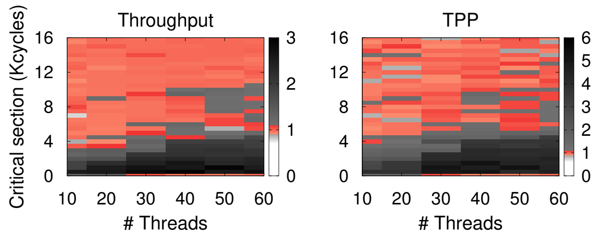

Figure 8: Throughput and TTP ratios of MUTEXEE over MUTEX on various configurations with a single lock.

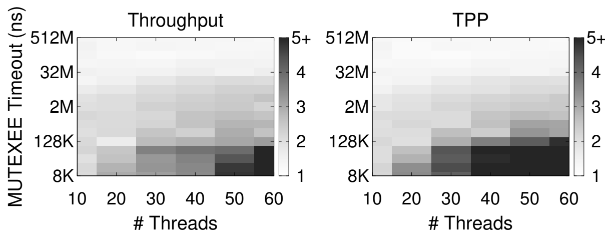

Figure 10: Throughput and TTP ratios of MUTEXEE without over with timeouts depending on the timeout.

Our design sensitivity analysis for MUTEXEE (not shown in the graphs) highlights three main points. First, spinning for more than 4000 cycles is crucial for throughput: MUTEXEE with 500 cycles spin behaves similarly to MUTEX. Second, the "wait in user space" functionality is crucial for power consumption (and improves throughput): If we remove it, MUTEXEE consumes similar power to MUTEX. Finally, the spin and mutex modes of MUTEXEE can save power on lengthy critical sections.

Fine-tuning MUTEXEE. The default configuration parameters of MUTEXEE should be suitable for most x86 processors. Still, these parameters are based on the latencies of the various events that happen in a `futex`-based lock, such as the latency of sleeping or waking up. Accordingly, in order to allow developers to fine-tune MUTEXEE for a platform, we provide a script which runs the necessary microbenchmarks and reports the configuration parameters that can be used for that platform.

Comparing MUTEXEE to MUTEX. Figure 8 depicts the ratios of throughput and energy efficiency of MUTEXEE over MUTEX on various configurations on a single lock. MUTEXEE indeed fixes the problematic behavior of MUTEX for critical sections of up to 4000 cycles. While MUTEX continuously puts threads to sleep and wakes them up shortly after, MUTEXEE lets the threads sleep for larger periods and keeps most lock handovers `futex` free. Of course, the latter behavior of MUTEXEE results in lower fairness as shown in Figure 9. Up to 4000 cycles, MUTEXEE achieves much lower 95th percentile latencies than MUTEX, because most lock handovers are fast with busy waiting. However, the price of this behavior is a few extremely high latencies as shown in the 99.99th percentile graph. These values are caused by the long-sleeping threads and represent the trade-off between lock fairness and energy efficiency. As the critical section size increases, the behavior of the two locks converges: Both locks are highly unfair.

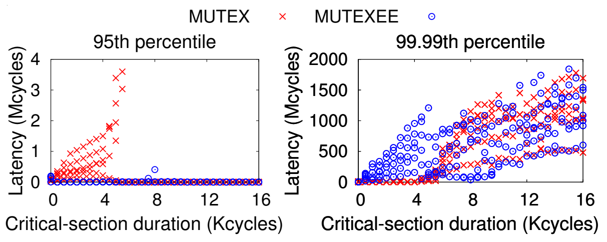

Figure 9: 95th and 99.99th percentile latency of a single MUTEX and MUTEXEE on various configurations.

Reducing MUTEXEE's Tail Latencies. MUTEXEE purposefully tries to reduce the number of `futex` invocations by handing the lock over in user space whenever possible. Therefore, it might let some threads sleep while the rest keep the lock busy, resulting in high tail latencies. A straightforward way to limit the tail latencies of MUTEXEE, so that threads are not allowed to remain "indefinitely" asleep, is to use a timeout for the `futex` sleep call. Once a thread is woken up due to a timeout, the thread spins until it acquires the lock, without the possibility to sleep again.[10] Controlling this timeout essentially controls the maximum latency of the lock (given that the sleep duration is significantly larger than the critical sections protected by that lock).

Figure 10 depicts the relative performance of MUTEXEE without over with timeouts for a single lock with 2000 cycles critical sections. For an 8 μs timeout, MUTEXEE delivers up to 14x lower throughput and 24x lower TPP than without timeouts. In general, for timeouts shorter than 16-32 ms, both throughput and TPP suffer, representing the clear trade-off between fairness and performance. For example, with 20 threads, MUTEXEE with a 4 ms timeout compares to the rest as follows:

Lock	**Throughput**	**TPP**	**Max Latency**
	Kacq/s	Kacq/Joule	Mcycles
MUTEX	317	4.0	2.0
MUTEXEE	855	10.9	206.5
MUTEXEE timeout	474	6.5	12.0

Depending on the application, the developer can decide whether to use timeouts and choose the timeout duration for MUTEXEE. For brevity, in the rest of the paper, we use MUTEXEE without timeouts. As we show in §6, we do not observe significant tail-latency increases due to MUTEXEE in real systems.

[10] Of course, one can design more elaborate variants of this protocol.

	MUTEX	TAS	TTAS	TICKET	MCS	MUTEXEE
Throughput	11.88	16.88	16.98	16.97	12.04	13.32
TPP	174.31	248.14	249.41	249.24	176.72	195.48

Table 2: Single-threaded lock throughput and TPP.

5.2 Evaluating Lock Algorithms

We evaluate various lock algorithms under different contention levels in terms of throughput and TPP.

Uncontested Locking. It is common in systems that a lock is mostly used by a single thread and both the acquire and the release operations are almost always uncontested. Table 2 includes the throughput (Macq/s) and the TPP (Kacq/Joule) of various lock algorithms when a thread continuously acquires and releases a single lock. We use short critical sections of 100 cycles.

The trends in throughput and TPP are identical as there is no contention. The locks perform inversely to their complexity. The simple spinlocks (TAS, TTAS, and TICKET) acquire and release the lock with just a few instructions. MUTEX performs several sanity checks and also has to handle the case of some threads sleeping when a lock is released. MUTEXEE is also more complex than simple spinlocks due to its periodic adaptation. The queue-based lock, MCS, is even more complex, because threads must find and access per-thread queue nodes.

Contention–Single (Global) Lock. We experiment with a single lock accessed by a varying number of threads. This experiment captures the behavior of highly-contended coarse-grained locks. We use a fixed critical section of 1000 cycles.

Figure 11 contains the throughput and the TPP results. On 40 threads, MUTEX delivers 73% lower TPP than TICKET: 63% less throughput and 5.8% more power. The throughput difference is due to (i) the global spinning of MUTEX, and (ii) the `futex` calls, even if they are infrequent. The power difference is mainly because of the pausing technique. MUTEX spins with `pause`, while TICKET uses a memory barrier. With `pause` instead of a barrier, TICKET consumes 4 Watts more.

Moreover, MUTEXEE maintains the contention levels and the frequency of `futex` calls low, regardless of the number of threads. This results in stable throughput and

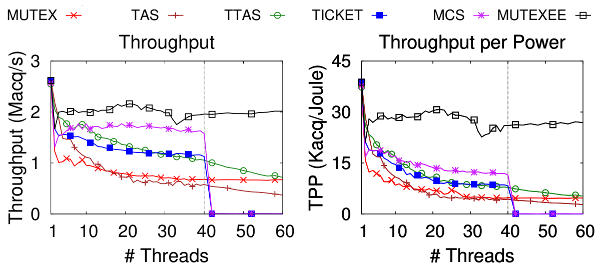

Figure 11: Using a single (global) lock.

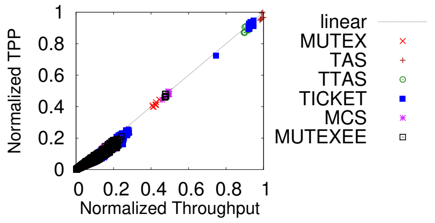

Figure 12: Correlation of throughput with energy efficiency (TPP) on various contention levels.

TPP because neither contention, nor the number of active hardware contexts increases with the number of threads. This behavior comes at the expense of high tail latency: On 40 threads, MUTEXEE has an 80x higher 99.9th percentile latency than MUTEX.

Regarding spinlocks, TAS is the worst in this workload. This behavior is due to the stress on the lock, which makes the release of TAS very expensive. Moreover, for up to 40 threads, the queue-based lock (MCS) delivers the best throughput and TPP. Queue-based locks are designed to avoid the burst of requests on a single cache line when the lock is released. On more than 40 threads, fairness shows its teeth. As Xeon has 40 hardware threads, there is oversubscription of threads to cores. TICKET and MCS, the two fair locks, suffer the most: If the thread that is the next to acquire the lock is not scheduled, the lock remains free until that thread is scheduled.

Finally, throughput and TPP are directly correlated: The higher the throughput, the higher the energy efficiency. Still, MUTEXEE delivers higher TPP by achieving both better throughput and lower power than the rest.

Variable Contention. Figure 12 plots the correlation of throughput with TPP on a diverse set of configurations. We vary the number of threads from 1 to 16, the size of critical section from 0 to 8000 cycles, and the number of locks from 1 to 512. At every iteration within a configuration, each thread selects one of the locks at random. The results are normalized to the overall maximum throughput and TPP, respectively.

Most data points fall on, or very close to, the linear line. In other words, most executions have almost one-to-one correlation of throughput with TPP. The bottom-left cluster of values represents highly-contended points. On high contention, there is a trend below the linear line, which represents executions where throughput is relatively higher compared to energy efficiency. These results are expected, as on very high contention sleeping can save power compared to busy waiting, but still, busy waiting might result in higher throughput.

If we zoom into the per-configuration best throughput and TPP, the correlation of the two is even more profound. On 85% of the 2084 configurations, the lock with the best throughput achieves the best energy efficiency

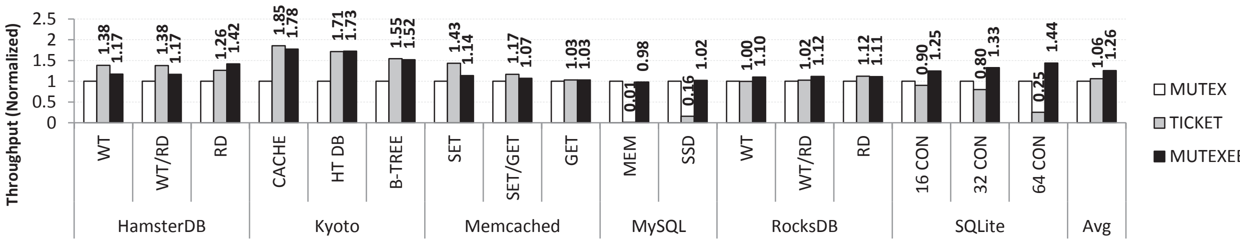

Figure 13: Normalized (to MUTEX) throughput of various systems with different locks. (Higher is better)

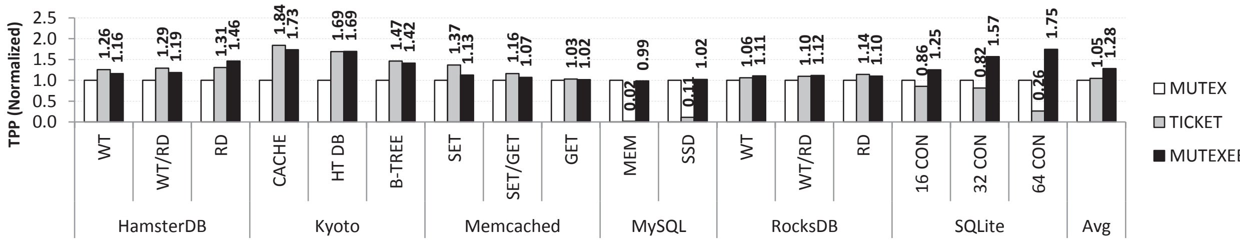

Figure 14: Normalized (to MUTEX) energy efficiency (TPP) of various systems with different locks. (Higher is better)

as well. On the remaining 15%, the highest throughput is on average 8% better than the throughput of the highest TPP lock, while the highest TPP is 5% better than the TPP of the highest throughput lock.

Finally, MUTEXEE delivers much higher throughput and TPP than MUTEX; on average, 25% and 32% higher throughput and TPP, respectively. MUTEX is better than MUTEXEE in just 4% of the configurations (by 9% on average, both in terms of throughput and TPP).

5.3 Implications

The POLY conjecture states that energy efficiency and throughput go hand in hand in locks. Our evaluation of POLY with six state-of-the-art locks on various contention levels shows that, with a few exceptions, POLY is indeed valid. The exceptions to POLY are high contention scenarios, where sleeping is able to reduce power, but still results in slightly lower throughput than busy waiting on the contended locks.

For low contention levels, energy efficiency depends only on throughput, as there are no opportunities for saving energy. In these scenarios, even infrequent `futex` calls reduce both throughput and energy efficiency.

For high contention, sleeping can reduce power consumption. However, the frequent `futex` calls of MUTEX hinder the potential energy-efficiency benefits due to throughput degradation. MUTEXEE is able to reduce the frequency of `futex` calls either by avoiding the ones that are purposeless, or by reducing fairness. MUTEXEE achieves both higher throughput and lower power than spinlocks or MUTEX for high contention levels.

6 Energy Efficiency of Lock-based Systems

We modify the locks of various concurrent systems to improve their energy efficiency. We choose the set of systems so that they use the pthread library in diverse ways, such as using mutexes or reader-writer locks, building on top of mutexes, or relying on conditionals. Note that we do not modify anything else other than the pthread locks and conditionals in these systems.

Table 3 contains the description and the different configurations of the six systems that we evaluate. All benchmarks use a dataset size of approximately 10 GB (in memory), except for the MySQL SSD configuration that uses 100 GB. We set the number of threads for each system according to its throughput scalability.

6.1 Results

Figures 13-14 show the throughput and the energy efficiency (TPP) of the target systems with different locks.

HamsterDB [3] Version: 2.1.7 # Threads: 4	An embedded key-value store. We run three tests with random reads and writes, varying the read-to-write ratio from 10% (WT), 50% (WT/RD), to 90% (RD).
Kyoto [7] Version: 1.2.76 # Threads: 4	An embedded NoSQL store. We stress Kyoto with a mix of operations for three database versions (CACHE, HT DB, B-TREE).
Memcached [8] Version: 1.4.22 # Threads: 8	An in-memory cache. We evaluate Memcached using a Twitter-like workload [40]. We vary the get-to-set ratio from 10% (WT), 50% (WT/RD), to 90% (RD). The server and the clients run on separate sockets.
MySQL [9] Version: 5.6.19	An RDBMS. We use Facebook's LinkBench and tuning guidelines [2] for an in-memory (MEM) and an SSD-drive (SSD) configurations.
RocksDB [10] Version: 3.3.0 # Threads: 12	A persistent embedded store. We use the benchmark suite and guidelines of Facebook for an in-memory configuration [11]. We run 3 tests with random reads and writes, varying the read-to-write ratio from 10% (WT), 50% (WT/RD), to 90% (RD).
SQLite [12] Version: 3.8.5	A relational DB engine. We use TPC-C with 100 warehouses varying the number of concurrent connections (i.e., 8, 32, and 64).

Table 3: Software systems and configurations.

For brevity, we show results with MUTEX, TICKET, and MUTEXEE. The remaining local-spinning locks are similar to TICKET (TAS is less efficient–see §5).

Throughput and Energy Efficiency. In 16 out of the 17 experiments, avoiding the overheads of MUTEX improves energy efficiency from 2% to 184%. On average, changing MUTEX for either TICKET or MUTEXEE improves throughput by 31% and TPP by 33%. The results include three distinct trends.

First, in some systems/configurations (i.e., Memcached and HamsterDB) sleeping can "kill" throughput. For instance, on the SET workload on Memcached, MUTEXEE allows for a few sleep invocations, resulting in lower throughput than TICKET.

Second, in some systems/configurations (i.e., MySQL and RocksDB) MUTEX is less of a problem. Both of these systems build more complex synchronization patterns on top of MUTEX. MySQL handles most low-level synchronization with customly-designed locks. Similarly, RocksDB employs a write queue where threads enqueue their operations and mostly relies on a conditional variable. Therefore, altering MUTEX with another algorithm does not make a big difference.

Finally, in MySQL and SQLite sleeping is necessary. Both these systems oversubscribe threads to cores, thus spinlocks, such as TICKET, result in very low throughput. A spinning thread can occupy the context of a thread that could do useful work. Additionally, on the SSD, TICKET consumes 40% more power than the other two, as it keeps all cores active. The fairness of TICKET exacerbates the problems of busy waiting in the presence of thread oversubscription: TTAS (not shown in the graph) has roughly 6x higher throughput than TICKET, but it is still much slower than MUTEX and MUTEXEE.

Overall, in five out of the six systems, the energy-efficiency improvements are mostly driven by the increased throughput. SQLite is the only system where the lock plays a significant role in terms of both throughput and power consumption. With MUTEXEE, SQLite consumes 15% and 18% less power than with MUTEX with 32 and 64 connections, respectively.

Tail Latency. MUTEXEE can become more unfair than MUTEX (see §5). Figure 15 includes the QoS of four systems in terms of tail latency. For most configurations, the results are intuitive: Better throughput comes with a lower tail latency. However, there are a few configurations that are worth analyzing.

First, MUTEXEE's unfairness appears in the RD configuration of HamsterDB, resulting in almost 20x higher tail latency than MUTEX, but also in 46% higher TPP. Second, TICKET has high tail latencies on all oversubscribed executions as a result of low performance.

Finally, MUTEXEE on SQLite achieves better throughput and lower power than MUTEX, without increasing tail latencies. TPC-C transactions on SQLite have latencies in the scale of tens of ms. Each transaction consists of multiple accesses to shared data protected by various locks. MUTEXEE does indeed increase the tail latency of individual locks, but these latencies are in the scale of hundreds of μs and do not appear in the transaction latencies. However, this low-level unfairness brings huge contention reductions. For instance, on 64 CON, the SQLite server with MUTEX puts threads to sleep for 472 μs on average, compared to 913 μs with MUTEXEE. The result is that with MUTEX, SQLite spends more than 40% of the CPU time on the `_raw_spin_lock` function of the kernel due to contention on `futex` calls. In contrast, MUTEXEE spends just 4% of the time on kernel locks, and 21% on the user-space lock functions.

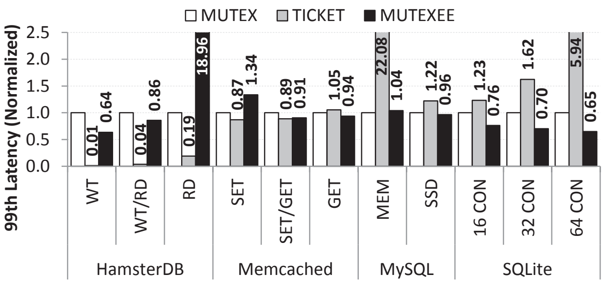

Figure 15: Normalized (to MUTEX) tail latency of various systems with different locks. (Lower is better)

Implications. Changing MUTEX in six modern systems results in 33% higher energy efficiency, driven by a 31% increase in throughput on average. Clearly, the POLY conjecture (i.e., throughput and energy efficiency go hand in hand in locks) holds in software systems and implies that we can continue business as usual: To optimize a system for energy efficiency, we can still optimize the system's locks for throughput.

Additionally, we show that MUTEX locks must be redesigned to take the latency overheads of `futex` calls into account. MUTEXEE, our optimized implementation of MUTEX, achieves 26% higher throughput and 28% better energy efficiency than MUTEX. Furthermore, the unfairness of MUTEXEE might not be a major issue in real systems: MUTEXEE can lead to high tail latencies only under extreme contention scenarios, that must be avoided in well engineered systems.

In conclusion, we see that optimizing lock-based synchronization is a good candidate for improving the energy efficiency of real systems. We can modify the locks with minimal effort, without affecting the behavior of other system components, and, more importantly, without degrading throughput.

7 Related Work

Lock-based Synchronization. Lock-based synchronization has been thoroughly analyzed in the past. For instance, many studies [13, 15, 34, 42, 46] point out scalability problems due to excessive coherence with traditional spinlocks and propose alternatives, such as hierarchical spin- and queue-based locks [34, 43, 54]. Prior work [20, 25] sacrifices short-term fairness for performance, however, it does not consider sleeping or energy efficiency. Similarly to MUTEXEE, Solaris' mutex locks offer the option of "adaptive unlock," where the lock owner does not wake up any threads if the lock can be handed over in user space [60]. David et al. [24] analyze several locks on different platforms and conclude that scalability is mainly a property of the hardware. Moreshet et al. [47] share some preliminary results suggesting that transactional memory can be more energy efficient than locks. Wamhoff et al. [62] evaluate the overheads of using DVFS in locks and show how to improve performance by boosting the lock owner. Our work extends prior synchronization work with a complete study of the energy efficiency of lock-based synchronization.

Spin-then-sleep Trade-off. The spin-then-sleep strategy was first proposed by Ousterhout [49]. Various studies [19, 35, 39] analyze this trade-off and show that just spinning or sleeping is typically suboptimal. Franke et al. [29] recognize the need for fast user-space locking and describe the first implementation of `futex` in Linux. Johnson et al. [33] advocate for decoupling the lock-contention strategy from thread scheduling. At first glance, MUTEXEE might look similar to their load-control TP-MCS [31] lock (LC). However, the two have some notable differences. LC relies on a global view of the system for load control, while MUTEXEE performs per-lock load control. LC's global load control can result in "unlucky" locks having their few waiting threads sleep for at least 100 ms, although there is low lock contention–sleeping threads are not woken up by a lock release, but only because of a decrease in load or 100 ms timeout. Finally, in contrast to MUTEXEE, LC might waste energy, because on low system load, no thread is blocked, even if the waiting times are hundreds ms.

Energy Efficiency in Software Systems. There is a body of work that points out the importance of energy-efficient software. For instance, Linux has rules to manage frequency and voltage settings [50]. Further work proposes OS facilities for managing and estimating power [48, 55, 58, 59, 65, 66]. Other frameworks approximate loops and functions to reduce energy [16, 57]. Moreover, compiler-based [63, 64] and decoupled access-execute DVFS [37] frameworks trade off performance for energy. In servers, consolidation [18, 21] collocates workloads on a subset of servers, and fast transitioning between active-to-idle power states allows for low idle power [44, 45]. Psaroudakis et al. [53] achieve up to 4x energy-efficiency improvements in database analytical workloads, using hardware models for power-aware scheduling. Similarly, Tsirogiannis et al. [61] analyze a DB system and conclude that the most energy-efficient point is also the best performing one. Our POLY conjecture is a similar result for locks. Nevertheless, while they evaluate various DB configurations, we study the spin vs. sleep trade-off. To the best of our knowledge, this is the first paper to consider the energy trade-offs of synchronization on modern multi-cores.

8 Concluding Remarks

In this paper, we thoroughly analyzed the power/performance trade-offs in lock-based synchronization in order to improve the energy efficiency of systems. Our results support the POLY conjecture: Energy efficiency and throughput go hand in hand in lock algorithms. POLY has important software and hardware ramifications.

For software, POLY conveys the ability to improve the energy efficiency of systems in an simple and systematic way, without hindering throughput. We indeed improved the energy efficiency of six popular software systems by 33% on average, driven by a 31% increase in throughput, These improvements are mainly due to MUTEXEE, our redesigned version of pthread mutex lock, that builds on the results of our analysis.

For hardware, POLY highlights the lack of adequate tools for reducing the power consumption of synchronization, without significantly degrading throughput. We performed our analysis on two modern Intel platforms that are representative of a large portion of the processors used nowadays. We argue that our results apply in most multi-core processors, because without explicit hardware support for synchronization, the power behavior of both busy waiting and sleeping will be similar regardless of the underlying hardware.

Our analysis further points out potential hardware tools that could reduce the power of synchronization. In brief, a truly energy-friendly `pause`, fast per-core DVFS, and user-level `monitor/mwait` can make the difference. In fact, industry has already started heading towards these directions. Intel includes on-chip voltage regulators on the latest processors, but all the cores share the same frequency. Similarly, the recent Oracle SPARC M7 processor includes a variable-length pause instruction and user-level `monitor/mwait` [51].

Acknowledgments. We wish to thank our shepherd, Leonid Ryzhyk, and the anonymous reviewers for their fruitful comments. We also want to thank Dave Dice for his useful feedback on an early draft of this paper. This work has been partially supported by the European ERC Grant 339539–AOC.

References

[1] America's Data Centers Consuming and Wasting Growing Amounts of Energy. `http://www.nrdc.org/energy/data-center-efficiency-assessment.asp`.

[2] Facebook LinkBench Benchmark. `https://github.com/facebook/linkbench`.

[3] HamsterDB. `http://hamsterdb.com`.

[4] Intel 64 and IA-32 Architectures Software Developer Manuals. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[5] Intel Xeon Processor E5-1600/ E5-2600/E5-4600 Product Families. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-1600-2600-vol-1-datasheet.pdf`.

[6] Java CopyOnWriteArrayList Data Structure. `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html`.

[7] Kyoto Cabinet. `http://fallabs.com/kyotocabinet`.

[8] Memcached. `http://memcached.org`.

[9] MySQL. `http://www.mysql.com`.

[10] RocksDB. `http://rocksdb.org`.

[11] RocksDB In-memory Workload Performance Benchmarks. `https://github.com/facebook/rocksdb/wiki/RocksDB-In-Memory-Workload-Performance-Benchmarks`.

[12] SQLite. `http://sqlite.org`.

[13] AGARWAL, A., AND CHERIAN, M. Adaptive Backoff Synchronization Techniques. ISCA '89.

[14] ANASTOPOULOS, N., AND KOZIRIS, N. Facilitating Efficient Synchronization of Asymmetric Threads on Hyper-threaded Processors. IPDPS '08.

[15] ANDERSON, T. The Performance of Spin Lock Alternatives for Shared-memory Multiprocessors. *IEEE TPDS* (1990).

[16] BAEK, W., AND CHILIMBI, T. Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation. PLDI '10.

[17] BARROSO, L., AND HÖLZLE, U. The Case for Energy-proportional Computing. *IEEE Computer* (2007).

[18] BENINI, L., KANDEMIR, M., AND RAMANUJAM, J. *Compilers and Operating Systems for Low Power*. Kluwer Academic Publishers, 2003.

[19] BOGUSLAVSKY, L., HARZALLAH, K., KREINEN, A., SEVCIK, K., AND VAINSHTEIN, A. Optimal Strategies for Spinning and Blocking. *J. Parallel Distrib. Comput.* (1994).

[20] CALCIU, I., DICE, D., LEV, Y., LUCHANGCO, V., MARATHE, V. J., AND SHAVIT, N. NUMA-Aware Reader-Writer Locks. PPoPP '13.

[21] CHASE, J., ANDERSON, D., THAKAR, P., VAHDAT, A., AND DOYLE, R. Managing Energy and Server Resources in Hosting Centers. SOSP '01.

[22] CRAIG, T. Building FIFO and Priority-Queuing Spin Locks From Atomic Swap. Tech. rep., 1993.

[23] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. ASPLOS '15.

[24] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. SOSP '13.

[25] DICE, D., MARATHE, V., AND SHAVIT, N. Lock Cohorting: A General Technique for Designing NUMA Locks. PPoPP '12.

[26] ESMAEILZADEH, H., BLEM, E., S. AMANT, R., SANKARALINGAM, K., AND BURGER, D. Dark Silicon and the End of Multicore Scaling. ISCA '11.

[27] ESMAEILZADEH, H., SAMPSON, A., CEZE, L., AND BURGER, B. Architecture Support for Disciplined Approximate Programming. ASPLOS '12.

[28] FLEISCHMANN, M. LongRun Power Management. Tech. rep., 2001.

[29] FRANKE, H., RUSSELL, R., AND KIRKWOOD, M. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. OLS '02.

[30] HARDAVELLAS, N., FERDMAN, M., FALSAFI, B., AND AILAMAKI, A. Toward Dark Silicon in Servers. *IEEE Micro* (2011).

[31] HE, B., III, W. N. S., AND SCOTT, M. L. Preemption Adaptivity in Time-Published Queue-Based Spin Locks. HiPC '05.

[32] HUNT, N., SANDHU, P., AND CEZE, L. Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures. INTERACT '11.

[33] JOHNSON, F., STOICA, R., AILAMAKI, A., AND MOWRY, T. Decoupling Contention Management from Scheduling. ASPLOS '10.

[34] KÄGI, A., BURGER, D., AND GOODMAN, J. R. Efficient Synchronization: Let Them Eat QOLB. ISCA '97.

[35] KARLIN, A. R., LI, K., MANASSE, M. S., AND OWICKI, S. Empirical Studies of Competitive Spinning for a Shared-memory Multiprocessor. SOSP '91.

[36] KOOMEY, J. Growth in Data Center Electricity Use 2005 to 2010. Analytics Press Report.

[37] KOUKOS, K., BLACK-SCHAFFER, D., SPILIOPOULOS, V., AND KAXIRAS, S. Towards More Efficient Execution: A Decoupled Access-execute Approach. ICS '13.

[38] LI, H., BHUNIA, S., CHEN, Y., ROY, K., AND VIJAYKUMAR, T. DCG: Deterministic Clock-Gating for Low-Power Microprocessor Design. *IEEE TVLSI* (2004).

[39] LIM, B.-H., AND AGARWAL, A. Waiting Algorithms for Synchronization in Large-scale Multiprocessors. *ACM TOCS* (1993).

[40] LIM, K., MEISNER, D., SAIDI, A., RANGANATHAN, P., AND WENISCH, T. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. ISCA '13.

[41] LO, D., CHENG, L., GOVINDARAJU, R., BARROSO, A., AND KOZYRAKIS, C. Towards Energy Proportionality for Large-Scale Latency-Critical Workloads. ISCA '14.

[42] LOZI, J., DAVID, F., THOMAS, G., LAWALL, J., AND MULLER, G. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. ATC '12.

[43] LUCHANGCO, V., NUSSBAUM, D., AND SHAVIT, N. A Hierarchical CLH Queue Lock. ICPP '06.

[44] MEISNER, D., GOLD, B., AND WENISCH, T. PowerNap: Eliminating Server Idle Power. ASPLOS '09.

[45] MEISNER, D., AND WENISCH, T. DreamWeaver: Architectural Support for Deep Sleep. ASPLOS '12.

[46] MELLOR-CRUMMEY, J., AND SCOTT, M. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM TOCS* (1991).

[47] MORESHET, T., BAHAR, R. I., AND HERLIHY, M. Energy Implications of Multiprocessor Synchronization. SPAA '06.

[48] MUTHUKARUPPAN, T., PATHANIA, A., AND MITRA, T. Price Theory Based Power Management for Heterogeneous Multi-Cores. ASPLOS '14.

[49] OUSTERHOUT, J. Scheduling Techniques for Concurrent Systems. ICDCS '82.

[50] PALLIPADI, V., AND STARIKOVSKIY, A. The Ondemand Governor. OLS '06.

[51] PHILLIPS, S. M7: Next Generation SPARC. Hot Chips '14.

[52] Powell, M., Yang, S., Falsafi, B., Roy, K., and Vijaykumar, T. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. ISLPED '00.

[53] Psaroudakis, I., Kissinger, T., Porobic, D., Ilsche, T., Liarou, E., Tözün, P., Ailamaki, A., and Lehner, W. Dynamic Fine-Grained Scheduling for Energy-Efficient Main-Memory Queries. DaMoN '14.

[54] Radovic, Z., and Hagersten, E. Hierarchical Backoff Locks for Nonuniform Communication Architectures. HPCA '03.

[55] Ribic, H., and Liu, Y. Energy-Efficient Work-Stealing Language Runtimes. ASPLOS '14.

[56] Rotem, E., Naveh, A., Moffie, M., and Mendelson, A. Analysis of Thermal Monitor Features of the Intel Pentium M Processor. TACS Workshop at ISCA '04.

[57] Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., and Grossman, D. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. PLDI '11.

[58] Shen, K., Shriraman, A., Dwarkadas, S., Zhang, X., and Chen, Z. Power Containers: An OS Facility for Fine-Grained Power and Energy Management on Multicore Servers. ASPLOS '13.

[59] Singh, K., Bhadauria, M., and McKee, S. Real Time Power Estimation and Thread Scheduling via Performance Counters. *SIGARCH CAN* (2009).

[60] Sun Microsystems. Multithreading in the Solaris Operating Environment. `http://home.mit.bme.hu/~meszaros/edu/oprendszerek/segedlet/unix/2_folyamatok_es_utemezes/solaris_multithread.pdf`.

[61] Tsirogiannis, D., Harizopoulos, S., and Shah, M. Analyzing the Energy Efficiency of a Database Server. SIGMOD '10.

[62] Wamhoff, J., Diestelhorst, S., Fetzer, C., Marlier, P., Felber, P., and Dice, D. The TURBO Diaries: Application-controlled Frequency Scaling Explained. ATC '14.

[63] Wu, Q., Martonosi, M., Clark, D., Reddi, V., Connors, D., Wu, Y., Lee, J., and Brooks, D. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. MICRO '05.

[64] Xie, F., Martonosi, M., and Malik, S. Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits. PLDI '03.

[65] Xu, C., Lin, F., Wang, Y., and Zhong, L. Automated OS-Level Device Runtime Power Management. ASPLOS '15.

[66] Zhai, Y., Zhang, X., Eranian, S., Tang, L., and Mars, J. Happy: Hyperthread-Aware Power Profiling Dynamically. ATC '14.