Small Kernels Like Big Locks

Submitted to EuroSys'15, submission number 146, 10 pages of 12 allowed

Abstract

The trade-off between coarse- and fine-grained locking is a well understood issue in operating systems. Course-grained locking provides lower overhead under low contention, fine grained locking provides higher scalability under contention, though at the expense of implementation complexity and reduced best-case performance.

We revisit the big-kernel-lock end of this trade-off in the context of microkernels and tightly-coupled cores with shared caches and low inter-core migration latencies. We evaluate performance on two architectures: x86 and ARM MPCore, in the former case also utilising transactional memory (Intel TSX). Our thesis is that on such hardware, a well-designed microkernel, with short system calls, can take advantage of course-grained locking on modern hardware, and still achieve scalability comparable to fine-grained locking.

1. Introduction

Waste of processing power resulting from lock contention has been an issue since the advent of multiprocessor computers, and has become a mainstream computing challenge since multicores became commonplace. Much research is directed to understanding and achieving scalability to large numbers of processor cores, where lock contention is inevitable and must be minimised [Clements et al. 2013]. It is now taken as given that locks must be fine-grained, ideally protecting individual accesses to shared data structures, and that shared data structures must be minimised, or, in the extreme case of a multikernel [Baumann et al. 2009], avoided alltogether.

Such work is typically performed under the assumption of high processor counts, with dozens or hundreds of concurrent hardware execution contexts, and long latencies of communication between such contexts.

While that is clearly an important scenario, we claim that it is not the only one that merits attention. There are now huge numbers of low-end multiprocessors, especially embedded systems such as smartphones. These present completely different points in the design space, with low to moderate core counts (2–8) and very low latency inter-core communication latencies (order of 10 cycles, often less than the L2 hit latency). One must suspect that what works best for high-end platforms may not work best for such systems.

Furthermore, even for high-end systems, a homogenous approach to locking may not necessarily be the best one. Systems with large processor counts usually consist of a large number of small clusters, where a cluster has similar properties to the embedded multicores, i.e. shared L2 or L3 cache and very small inter-core communication latencies. The best design for such a system may well turn out to be based on a clustered kernel design, which combines per-cluster shared data structures with an inter-cluster shared-nothing approach [von Tessin 2012].

Besides the characteristics of the processor architecture, operating system structure is obviously an important factor as well. In a monolithic kernel, such as Linux, typical system-call duration ranges from tens of microseconds to milliseconds. In such a system, course-grained locking will quickly lead to contention even on small and closely-coupled clusters. In contrast, a well-designed microkernel is essentially a context-switching engine, and average system-call duration is of the order of a microsecond. It is not obvious that fine-grained locking has any benefits for such a kernel running on a closely-coupled cluster.

This discussion may seem academic at first, given that fine-grained locking techniques are well-known and widely implemented, so why not use them anyway? There are, in fact, strong reasons to stick with course-grained locking as long as possible: It is now possible to formally verify a microkernel, but for the foreseeable future, this remains unfeasible for a kernel that exhibits true concurrency [Klein et al. 2014]. It is therefore important to understand the performance impact of a big-lock design, which keeps concurrency out of the kernel and thus enables verification. Furthermore, each lock acquisition has a cost, even when not contented. While insignificant compared to the overall system-call cost in a system like Linux, in a microkernel with a average syscall-cost of a few 100 cycles, it adds significant overhead.

We therefore perform a detailed examination of the scalability of the seL4 microkernel[1] on closely-coupled clusters on two vastly different hardware platforms (an x86-based desktop and an ARM-based embedded processor) under different locking regimes. We show the following outcomes.

- We compare several different lock types on both architectures, and show ARM-based multiprocessors are indifferent to lock choice for realistic lock/unlock intervals. The x86-based processor is more sensitive, and we identify good lock choices for those processors.
- We implement 3 microkernel variants on ARM and four in x86, with one being to the best of our knowledge, the first kernel almost entirely inside a single hardware transactional memory transaction. The others being two coarse grained and one fined grained.
- We compare the effect of lock choice on IPC microbenchmark performance for a uniprocessor, and on potential IPC throughput using multiple cores.
- The Redis key/value store is used to provide an I/O intensive workload to evaluate performance at the macro level.

We show that the choice of concurrency control in the kernel is clearly distinguishable for extreme synthetic benchmarks. For a real I/O (and thus system call intensive) workload, only minor performance differences are visible, and coarse grain locking is preferred (or the equivalent elided with hardware transactional memory) over extra kernel complexity of fine-grained locking.

2. Background

The best locking granularity is determined by a trade-off involving multiple factors. As long as there is no contention, taking and releasing locks is pure overhead, which is minimised by having just a single lock, the *big kernel lock* (BKL). Each lock adds some overhead which degrades the best-case (i.e. uncontended) performance.

As long as the total number of locks is small, this baseline overhead is usually small compared to the basic system-call cost. However, on a well-designed microkernel, where system calls tend to be very short (100s of cycles) this overhead might matter.

In the case of contention, fine-grained locking can significantly reduce contention, if it enables unlocked execution of the majority of code. In a BKL kernel, contention can be expected to be noticeable as soon as the *kernel time* (fraction of time spent inside the kernel) is not small compared to the *think time* (fraction of time spent in user mode).

The amount of kernel time depends on the profile of system calls executed, and thus on the workload. On a monolithic kernel, most system services are provided by the kernel, especially I/O, and consequently I/O-intensive workloads tend to have high kernel time. On a microkernel, system services are provided by server processes running in user mode, and the kernel provides communication between clients and servers. On a well-designed microkernel, such as the ones of the L4 family, kernel time is dominated by context switches [Liedtke 1995]. The total number of kernel calls is higher than in a monolithic kernel (at least twice as high, as every server invocation invokes the kernel twice) but the average system-call latency is a fraction of that of a monolithic kernel.

Hence, a BKL is a more credible design for a microkernel than for a monolithic OS. In this paper we investigate whether a BKL approach can work for a microkernel on a closely-coupled cluster of processor cores. Closely-coupled architectures are unlikely to scale to large core counts, so high-end scalability is not our concern and we perform our investigation on quad-core platforms. For maximum validity of our results, we examine two very different architectures: an x86-based desktop processor and an ARM-based processor aimed at embedded devices, especially smartphones.

2.1 ARM platform

Our embedded processor is an ARM multicore system on a chip (SoC). Specifically, we examine the Sabre Lite, which is based on a Freescale i.MX 6Q, a SoC featuring a quad-core ARM Cortex-A9 MPCore processor.

The cores run at a 1 GHz clock rate and each has split level-1 (L1) data and instruction caches, each 4-way-associative and 32 KiB in size. The cores share a 1 MiB unified 16-way-associative level-2 (L2) cache, which is the last-level cache. Typical L1 access time is 1–2 cycles, L2 access time is 8 cycles [ARM 2010]. We measured the access time to 1 GiB of main memory to be 51 cycles.

The platform also features a Gigabit Ethernet controller, though the theoretical maximum performance is limited to 470 Mb/s (total for transmit and receive) due to an internal bus throughput limitation [Freescale 2013].

The MPCore architecture features a *snoop control unit* (SCU) between the private L1 data caches and the shared L2 cache. The SCU implements a variation of MESI cache coherence, with the following adaptions.

- The SCU duplicates the tag bits of the L1 data caches to enable checking of remote caches without accessing them.
- Clean data is copied directly from L1 to L1 (ARM terms this *direct data intervention*).
- Dirty data (i.e. the *modified* state in MESI) is migrated directly from one core's L1 to another core's L1, without first writing the data back to the shared L2 (termed *migratory lines*).

2.2 x86 platform

As an x86 platform we use a desktop Dell Optiplex 9020. This desktop features a Q87 Express chipset with a Intel

[1] Open-source seL4 is available from http://sel4.systems/.

Core i7-4770 processor. This is a quad core processor with a clock rate of 3.4 GHz, including 8 hardware threads, although we disable hyperthreading. The processor features 3 levels of cache. Each core has private L1 instruction and a data caches, each 32 KiB in size and 8-way associative. Each core furthermore has a non-inclusive, 8-way 256 KiB L2 cache. And an inclusive, 16-way, 8 MiB L3 cache is shared between all the cores.

The platform also includes 16 GB of main memory and a 82574L Gigabit Ethernet controller.

The Core i7-4770 is a representative of the Haswell microarchitecture . It thus has support for Intel's TSX *restricted transactional memory* (RTM) implementation of hardware transactional memory.

Intel's RTM provides 3 additional instructions; XBEGIN, XEND, and XABORT. Code successfully executed between XBEGIN and XEND instructions will appear to have completed atomically and is thus called a transactional region. If there exist any memory conflicts during the execution of the transactional region then the transaction will abort and return to the instruction just after the XBEGIN. One can additionally issue an XABORT during a transactional region to explicitly abort.

RTM is implemented by taking advantage of existing cache coherency protocols to identify sets of cache lines written to and read by different cores on the CPU. This has two important consequences; memory conflicts are captured at a cache line granularity and a transaction must fit inside a hardware-limited L1 cache size. The latter consequence is an indication that larger kernels will probably not be viable when wrapped inside a RTM transaction as they are less likely to fit within this cache size.

Finally, a RTM transaction is not guaranteed to complete even when the transaction is small enough and has no memory conflicts. A variety of scenarios can result in an abort, of which many are left unspecified. Of particular interest to our work is many interactions with hardware registers that trigger aborts, but are clearly unavoidable when executing OS code.

Given transactions have no guarantees of progress, the developer must ensure there exists a fallback method of synchronization that ensures progress in the presence of repeated aborts. A commonly implemented technique is to fallback to a regular lock in the case of repeated aborts, together with testing the lock upon entry to an RTM region to ensure the lock is free and in the read set of initial of transaction attempts.

3. Small-scale multicore locking

David et al. [2013] has recently examined synchronisation on modern many-core machines. We revisit and extend their work as an introduction to microkernel synchronisation on small-scale multicore machines. The motivation is two-fold: Firstly, we examine locking specifically targeted at microkernels and their workloads on multicore processors; and secondly, we highlight the performance differences between the two processors (and thus architectures) of interest.

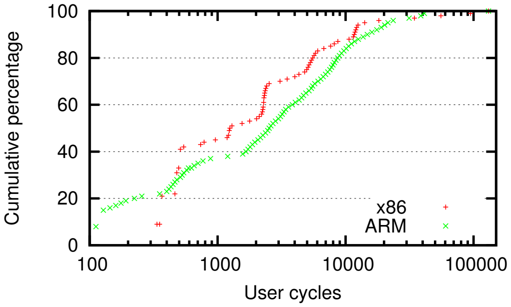


Figure 1: CDF of think times (user cycles between syscalls) collected from YCSB benchmark described in Section 6.

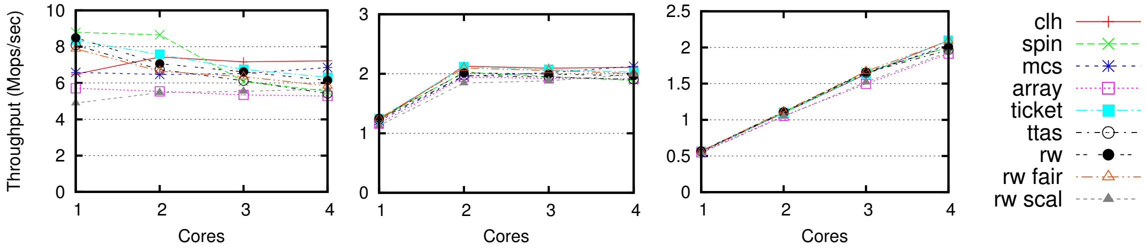
We port the synchronisation performance analysis suite of David et al. to the ARM platform. From the suite, we select the following synchronisation primitives of interest for small-scale locking: scalable queue-based CLH (clh) and MCS (mcs) locks, scalable array-based lock (array) [Craig 1993; Mellor-Crummey and Scott 1991a; Anderson 1990], fair but non-scalable ticket lock (ticket) [Mellor-Crummey and Scott 1991a], and unfair and non-scalable spinlock (spin) and a test-and-test-and-set lock (ttas) [Anderson 1990]. We do not use back-off for any of the locks.

For reasons explained in Section 4.2, our microkernel implementation requires a fair reader-writer lock. We therefore add two such locks to the benchmark suite: a fair, non-scalable (rw fair) and a fair, queue-based, scalable reader-writer lock (rw scal) [Mellor-Crummey and Scott 1991b].

3.1 Lock thoughput benchmarks

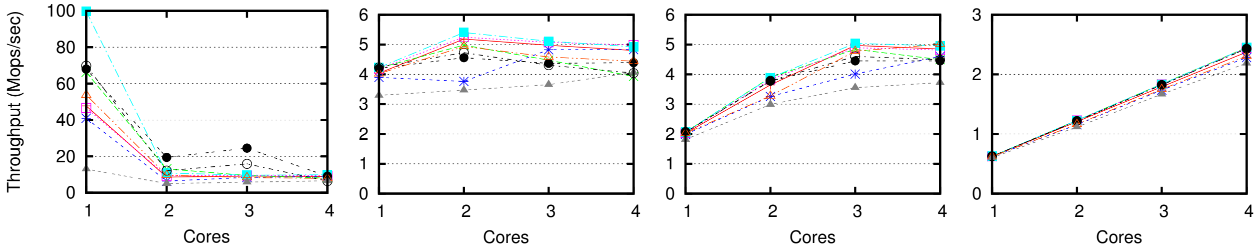
We compare lock performance across CPU architectures and lock implementations via a modified lock-throughput benchmark from the existing suite [David et al. 2013]. Our variant measures the number of successful lock acquisitions per second. The critical section within the lock updates 4 cache lines, which corresponds to the number of cache lines shared across the cores in a fast microkernel system call (seL4 IPC). The benchmark is parametrised by *hold time*, a delay while holding the lock in CPU cycles, and *pause time*, the number of cycles of delay prior to attempting to re-acquire the lock.

We choose the parameters of interest based on analysis of the system-call-intensive macro-benchmark introduced in Section 6. Figure 1 shows the cumulative distribution function (CDF) of the number of cycles spent in user mode between system calls plus kernel entry and exit costs ("think times"). We can see that about 20% of think times are below 300 cycles for ARM, and below 400 cycles for x86. 1,250 cycles represent the 40th percentile threshold on ARM and 50% of x86 think times, while 5,000 cycles represent about 65% on ARM and 75% on x86. We use those values as pause



(a) hold=0, pause=0 (b) hold=300, pause=300 (c) hold=300, pause=1250

Figure 2: Lock throughput on ARM.



(a) hold=0, pause=0 (b) hold=300, pause=400 (c) hold=300, pause=1250 (d) hold=300, pause=5000

Figure 3: Lock throughput on x86.

times in our benchmarks. We use a hold time of 300 cycles, which is approximately the time of an IPC operation in seL4 [Elphinstone and Heiser 2013]. For investigating raw lock overhead, we also use a hold time of zero.

Figure 2 shows the results for ARM (with the uninteresting (300,5000) case omitted), and Figure 3 for x86. On the ARM the relative standard deviation is less than 0.5% in all cases, while on x86 it is quite large in some cases, as shown in Figure 4 for the (0,0) scenario.

In the extreme (0,0) throughput benchmark, the majority of execution is within the lock code and critical section, with the outer loop being the only parallelisable code. With little parallelisable code, Amdahl's law predicts a perfectly scalable lock would exhibit a throughput that is independent of the number of cores that try to access it. On ARM, Figure 2(a) shows scalable locks approach this ideal Non-scalable locks show some degradation, with spinlocks being the worst offender with about 25% degradation in when the core count exceeds two.

This simple benchmark cannot be used in isolation to evaluate locks against each other, as the (0,0) case is highly sensitive to not only to the lock implementation, but also the benchmark harness, and compiler's ability to optimise the combination of both. However, the figure also clearly shows that the best-case acquisition time differs by almost a factor two between the cheapest (spinlock) and the dearest (array, which on ARM uses a software-implemented remainder operation). The ticket lock performs well across all scenarios at this scale. The CLH lock outperforms the MCS lock in our statically allocated locking scenario, though the MCS has the advantage of supporting stack-based allocation of queue nodes.

More interesting is the influence of the hardware architecture (and the memory architecture is likely a bigger factor than the ISA in this case). Here, the contrast between the ARM and x86 platforms is striking, as indicated by the vastly different graphs of Figure 2(a) and Figure 3(a). While the ARM comes close to ideal behaviour, the x86 shows a significant drop in throughput once cache line transfer occurs between cores, with throughput dropping from about 10 times that of ARM in the best case (single core) to roughly the same as ARM in the contented case. This is mostly a reflection of the relatively long L1-to-L1 cache-line transfer latency, which is very fast on the ARM (faster than an L2 access).

On x86, we observe that the TTAS and RW locks perform better than the other lock types for the 2- and 3-core (0,0) scenarios. Figure 4, showing the relative standard deviations across ten runs, provides some insight. Obviously, for a single core the variance of the results is essentially zero, as we have (fully deterministic) sequential execution. For two and three cores, however, the variance is huge in the case of the unfair lock types. These allow many repeated acquisition on the same core, which helps throughput as it avoids the latency of cache-line migrations. The exact

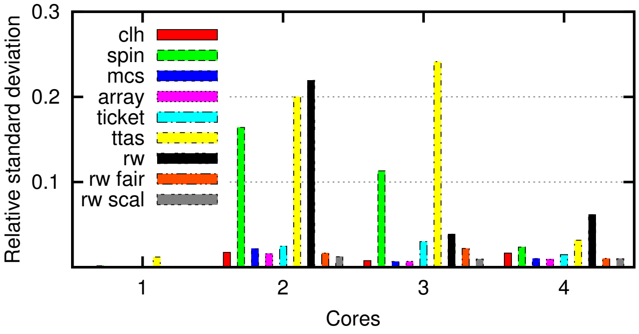


Figure 4: x86 relative standard deviation for hold=0, pause=0

acquisition order depends on unspecified (and potentially truly random) hardware behaviour.

It is interesting to note that for the spinlock and TTAS this effect vanishes when all four cores are used, the locks are behaving much more deterministically when all cores are equally competing for the lock. We do not know the reason for this, but it could well be that Intel optimised the bus arbiter for this case.

The results for the more realistic set of hold and pause times show ARM is quite insensitive to lock choice, even in the low think time (300,300) case shown in Figure 2(b), and a think time of 1,250 cycles (Figure 2(c)) is enough to achieve perfect scalability. In contrast, the x86 needs the higher 75th percentile think time (300,5000) case (Figure 3(d)) for the workload to dominate differences in lock overhead and achieving scalability.

We summarise our findings as follows.

- The ARM is fairly insensitive to lock choice in benchmarks with realistic hold and pause times even under high contention.
- The low L1-L1 cache latency on the ARM enables essentially perfect lock behaviour.
- x86 is much more sensitive to lock choice, with lock choice affecting performance for realistic hold and pause times.
- For small-scale multicore locking on both architectures, simple ticket locks are generally among the better performing locks. Of the scalable locks, CLH locks outperform MCS locks.

4. Microkernel implementation

As we use seL4 as our microkernel testbed, we will now summarise its relevant features, Klein et al. [2014] presents more details. seL4 is event-based, with a single kernel stack. To aid verification, seL4 uses a two-phase system call structure, where the first phase confirms the pre-conditions required for system call execution, and the second phase executes the system call without failure. Blocking operations are handled by re-starting the system call and thus re-confirming the preconditions prior to continuing execution.

The kernel executes with interrupts disabled in order to avoid dealing with concurrency. As the kernel features some long-running operations (destruction of kernel objects that may have derived objects), it achieves usable interrupt latencies through explicit preemption points, where the kernel polls for pending interrupts, and restarts the operation if there are any [Blackham et al. 2012]. The restart allows interrupts to be triggered from outside the kernel, prior to continuing the original operation. While a requirement for verification, this concurrency-free design has traditionally been used in L4 kernels even before formal verification became a possibility. This was to achieve high best-case performance, and has been used on other systems as well [Ford et al. 1999].

seL4 supports two versions of IPC: synchronous (rendezvous) message passing with a payload of up to a few hundred bytes, and asynchronous notification with semantics similar to a binary semaphore. IPC operates via port-like objects called *endpoints*.

4.1 Big kernel lock

The BKL is the natural, minimal extension of the existing seL4 design to multicores, as it is easy to implement and mostly preserves the in-kernel assumption of no concurrency. The kernel entry and exist code which saves and restores the user-state, and sets up safe kernel execution remains outside of the BKL, the rest of the kernel of the kernel is protected by the BKL.

This design is not entirely sufficient – the following assumption no longer holds on a multicore kernel, even when the BKL is held:

> Except for the currently executing thread's TCB and page table, all other TCBs and page tables are quiescent, and can be mutated or deleted.

This is no longer true as other cores (executing at user-level) implicitly depend on their thread's TCB and page table to transition to kernel-mode via the kernel entry code to compete for the BKL. We address this by modifying the kernel to explicitly check for quiescence of these two data structures prior to deletion. For example, if a TCB is associated with a remotely running thread, we mark the TCB as pending quiescence, trigger an inter-processor interrupt (IPI) to the remote core, release the BKL, and then wait for quiescence. The thread on the remote core will enter the kernel, acquire the BKL, mark itself as quiescent, and another thread will be chosen, thus releasing the BKL. The original thread can now obtain the BKL on its core and continue the attempted operation on the now quiescent target TCB.

This design, which is partially driven by the existing event-driven code base, is a valid design choice thanks to the short duration of most system calls in the microkernel, it would result in poor scalability on any other kind of system.

The only other changes required are (i) enabling TLB shootdown and (ii) introducing per-core idle threads. In order to minimise inter-core cache-line migrations, we also introduce per-core kernel stacks, scheduler queues and current thread pointers, even though access is serialised by the BKL.

Our BKL also inherits some improvements targeting hardware transactional memory support (see Section 4.3 for details), specifically moving context switch related hardware operations to after BKL release. These changes are not required for a BKL kernel, but potentially improve performance by moving these operations into the parallelisable part of the kernel.

Based on the small-scale multicore locking experiments, we implement two variants of the BKL: ticket locks and CLH queue locks. Ticket locks perform better under low contention, while CLH locks scale better.

4.2 Fine-grained locking

To compare the course-grained BKL with more complex but more scalable fine-grained locking, we first replace the BKL with a fair reader-writer lock.[2] The writer-lock is equivalent to the course-grained BKL in avoiding concurrency in the microkernel. We can selectively migrate performance-critical parts of the microkernel into the reader-lock, exposing those parts to concurrency in return for improved scalability.

At minimum, we retain the allocation and freeing of kernel objects within the writer lock. These are generally heavyweight operations associated with process creation or destruction. The benefit is that existing memory safety is retained while holding the reader lock, though contents of the objects themselves may be exposed to concurrency.

The frequent kernel operations of IPC and interrupt delivery (via IPC) do not require allocation, and thus can be moved into the reader lock. IPC mutates the state of TCBs, endpoints, and (potentially) the scheduler queues (depending on whether optimisations apply that avoid queue updates during IPC). We add ticket locks to each of these data structures to synchronise IPC within the reader lock. A typical synchronous IPC now involves the kernel reader lock, two TCB locks, and one endpoint lock. Lock contention during IPC is now limited to cases where IPC involves a shared destination or endpoint, or general contention with the kernel writer lock. Independent activities on independent cores result in no lock contention. We avoid deadlocks resulting from locking TCBs by identifying the TCBs involved prior to locking (made possible by memory safety provided by the reader lock), and then locking them in order of their memory addresses.

Except for IPC and interrupt delivery, all other seL4 system calls are synchronised using the writer-lock in our present prototype.

4.3 Hardware transactional memory

On x86, the combination of Intel's RTM extensions and a small microkernel allows us to explore optimistically executing the majority of the kernel without concurrency control. The two-phased system call structure of seL4, where the first phase validates the pre-conditions for execution and the second phase executes without failure, together with the absence of blocking of an event-based kernel, enables us to surround almost the entire kernel with the transaction primitives of Figure 5.

```
beginTransaction()
{
  while ((status = _xbegin()) !=
         _XBEGIN_STARTED ) {
    txnAttempts++;
    if (txnAttempts >=
        RTM_ATTEMPTS_THRESHOLD) {
      /* Give up on transaction */
      break;
    }
    /* wait for lock to be free */
    /* before trying txn again */
    while(LockTest());
  }

  if (status == _XBEGIN_STARTED) {
    /* confirm the lock is free */
    if (LockTest()) {
      _xabort('L');
    }
  } else {
    /* fallback on BKL */
    lockAcquire();
  }
}

endTransaction()
{
    if ( (txnInside = _xtest()) ) {
        _xend();
    } else {
        lockRelease();
    }
}
```

Figure 5: Kernel transaction pseudo code.

The main microkernel changes needed to support running the microkernel in a transaction (in addition to the changes described in the BKL variant) is to move CPU operations that trigger transaction aborts to after a successful transaction. Note that all L4-like microkernels run device drivers run at user level, thus avoiding by design one common im-

[2] We found a reader-biased lock results in starvation in our experiments.

pediment to running the kernel in a transaction. The remaining problematic operations are as follows.

- Context-switch-triggered page-table register (CR) loading and segment-register loading.
- IPI triggering for inter-core notifications.
- Interrupt management for user-level device drivers, which consists of masking and acknowledging interrupts prior to return to the user-level handler.

The key insight here as to why it is safe to move these operation outside of the transaction is that the two-phase kernel ensures the system call requiring these operations is guaranteed to succeed prior executing these operations, and that these operations are local to a core and thus are not exposed to concurrency issues from other cores.

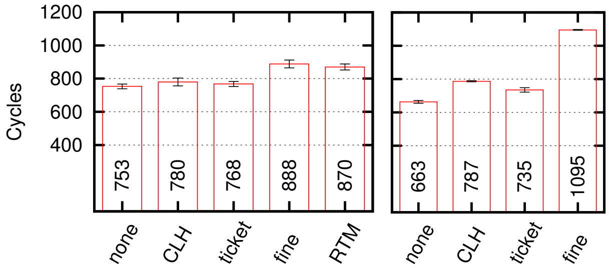
5. Microbenchmarks

To evaluate our multicore microkernel variants, we use two IPC microbenchmarks and a macrobenchmark consisting of Redis and the Yahoo cloud serving benchmark, as described in the following sections. The platforms under test have been already been described in Section 2.1 and Section 2.2.

5.1 Single-Core IPC Microbenchmarks

An emphasis on high IPC performance has a long history in the L4 community [Elphinstone and Heiser 2013], as IPC performance is a key contributor to overall system performance in microkernel-based systems. We therefore start our analysis by examining the overhead introduced by each variant of concurrency control.

We use the traditional “ping-pong” benchmark for measuring best-case IPC performance: a pair of threads on a single core does nothing other than sending messages to each other. Figure 6 shows the result for each kernel variant.



(a) x86 (b) ARM

Figure 6: Raw round-trip IPC cycle cost for different seL4 locking mechanisms. Error bars indicate standard deviations.

The figure shows that on x86, the overhead of a single lock (BKL) is small (2% for the ticket lock, 3.6% for CLH) compared to a baseline uniprocessor kernel with no concurrency primitives (“none”). With fine-grained locking, however, the overhead is 18%. The cost of uncontended transactions is 16% higher.

On the ARM, the cost of a single lock is significantly higher (11% for the ticket lock), while fine-grained locking significantly increases IPC cost by 65%! The higher synchronisation costs for ARM relate to its partial store order memory model which requires memory barriers (`dmb` instructions) to preserve memory access ordering. In our experience, the barriers cost from 6 cycles up to 19 cycles depending on architectural state. For this benchmark, CLH requires 6 barriers, ticket 4, and fine grained 16. Thus the big difference in IPC costs can be mostly attributed to the extra barriers.

Together these results confirm that there is a significant performance cost of fine-grained locking (even ignoring the fact that it makes formal verification intractable). This provides good justification for sticking with the BKL if sufficient scalability can be achieved.

5.2 Multicore IPC Microbenchmarks

In order to assess the impact of concurrency control more fully, we repeat the IPC microbenchmarks with multiple cores. Specifically, we run a copy of the ping-pong benchmark on each core, with the cores executing completely independently.

This benchmark produces extreme contention on the kernel (zero think-time). However, *none of the kernel datastructures are contended*, as each core’s pair of threads accesses disjoint kernel objects (TCBs and IPC endpoints) during their syscalls. We expect (ignoring overheads) that fine-grained locking should scale perfectly in this benchmark. It also allows us to run an (unsafe!) kernel implementation without any locking as a performance baseline.

The benchmark consists of a two second warm-up followed by sampling total IPCs during a one second interval to give total IPC round-trip throughput per second. The number of active cores is varied from 1 to 4. Each benchmark is repeated 16 times and we report the mean and standard deviation (as error bars) in Figure 7.

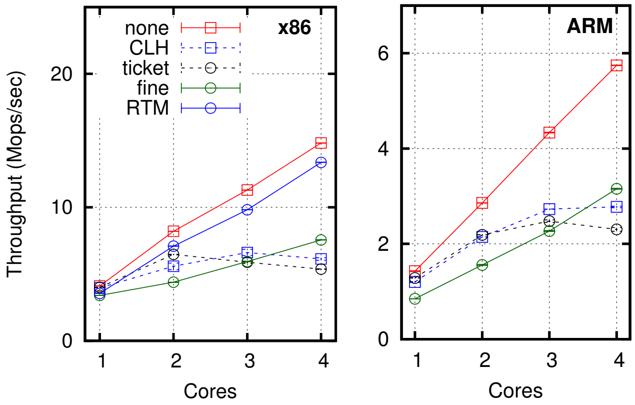


Figure 7: Synchronous intra-core IPC round-trip throughput on x86 (left) and ARM (right), error bars are too small to be visible.

Note that for the single-core case, these throughput figures are slightly less than the inverse of those of Figure 6, as the latter are pure kernel times, while the throughput figures include minimal userland code.

We see, as expected, that the “none” baseline case for x86 and ARM scales near perfectly with $R^2 > 0.99$ for a least-squares fit linear regression. While featuring the highest overheads, RTM (x86 only) and fine-grained locking also scale linearly with $R^2 > 0.99$.

The most striking observation is that RTM on x86 performs near the (unsafe) baseline performance. RTM manages to exploit the fundamental independent execution of the cores.

For fine-grained locking, the scalability does not compensate for the higher overhead until 4 cores are involved when compared to the BKL variants, on both x86 and ARM. Both CLH and ticket locks out-perform fine-grained locking in this extreme low think-time benchmark for less than 4 cores. When comparing CLH with ticket, we that the scalable CLH scales better than tickets, but performs slightly worse under low contention.

In summary, we expect with more realistic think-times, that the BKL variants will outperform fine-grained locking.

6. Redis-based Macrobenchmark

To explore the performance characteristics of our microkernel variants under more realistic loads (i.e. realistic user-level think-times) we ported the lwIP TCP/IP stack [lwIp] and the Redis key-value store [Redis] to our platforms.

Redis benchmarks explanation

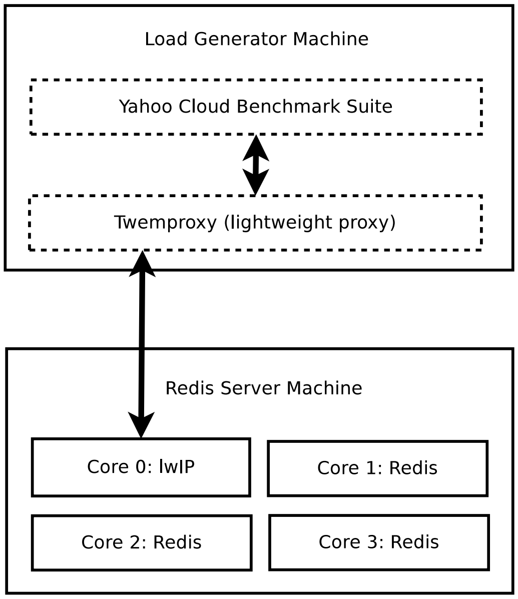


Figure 12: Redis-based benchmark architecture.

Our system under test is shown in Figure 12. The system runs seL4 with one core dedicated to lwIP and the Ethernet device driver, including handling all I/O interrupts. The remaining cores run instances of Redis. Note that we run Redis as volatile instances (we disabled file system access), as our prototype lacks file system support. To evaluate scalability, we vary the number of active cores from one to four. In the one core case, we run an lwIP thread and one Redis thread on the same core. For the remaining cases, we have a single lwIP thread pinned to a single core, and $N - 1$ Redis instances where N is the number of cores under investigation.

We evaluate performance using Yahoo! Cloud Serving Benchmarks (YCSB) [Cooper et al. 2010], running on a dedicated load generator machine, with a dedicated Gigabit Ethernet link between the load generator and the machine under test. The load generator machine uses twemproxy to shard the workload across the remote redis instances [twemproxy].

YSCB consists of several workloads. We use the same A–E benchmarks as presented in Cooper et al. [2010], which are as follows.

A: update-heavy workload (50/50 read and writes) using zipfian distribution for record selection in the store.

B: read-mostly workload (95/5) with zipfian distribution.

C: read-only workload with zipfian distribution

D: new records inserted, then most recently inserted record are read.

E: short ranges of records are queried, where record selection is zipfian, but the number of records in the range is uniformly distributed.

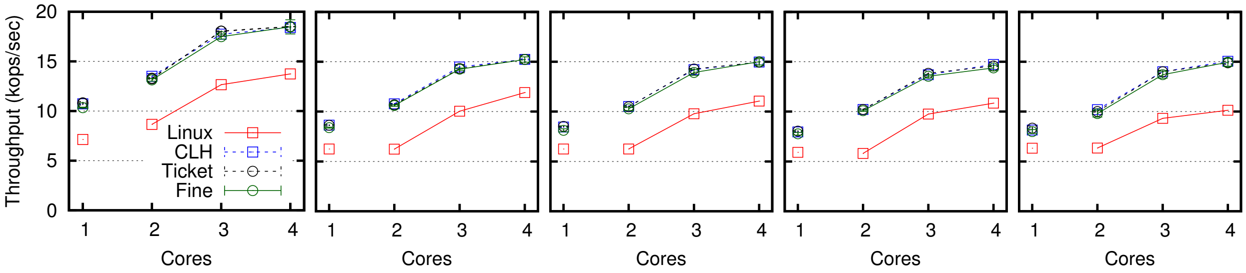
We tuned the number of operations (`operationcount`) for each workload and architecture to produce approximately 30 seconds of run time for each workload. We also set `recordcount` to 32000 to reduce the working set size to fit within our prototypes memory limitations.

For each combination of kernel variant, cpu architecture, number of cores, and workload, we run YCSB three times and report mean and standard deviation. We have also instrumented our kernel variants to record idle time to obtain CPU utilisation for each run.

6.1 Results

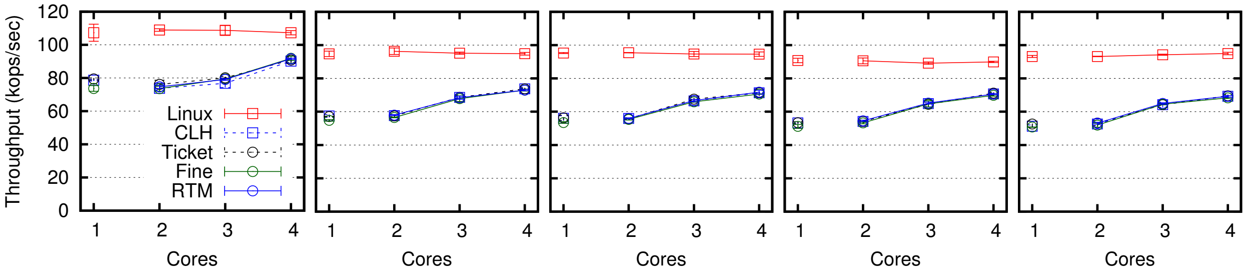
Figures Figure 8 and Figure 9 show the throughput results for ARM and x86 architectures. We report mean throughput in thousand of operations per second, for varying number of core, for each system. Standard deviations are shown as (nearly indistinguishable) error bars. Note the single-core benchmark is different from the others in having redis and lwIP share the same core, which we indicate in the graphs by keeping the single core data points unconnected.

We note that our system out-performs Linux on ARM, and under performs compared to Linux on x86. This is mostly due to our immature Ethernet driver on our x86 prototype which, for example, does packet checksumming in software. We compare our performance with Linux to



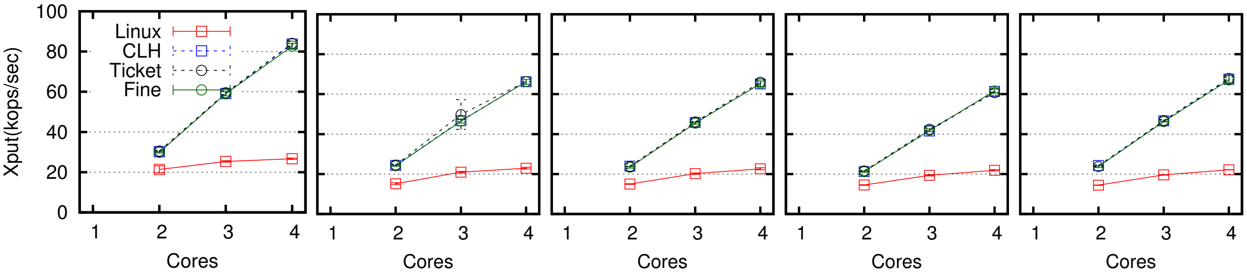
(a) Redis A (b) Redis B (c) Redis C (d) Redis D (e) Redis E×50

Figure 8: YCSB Redis benchmarks on ARM. Benchmark E scaled by factor 50.



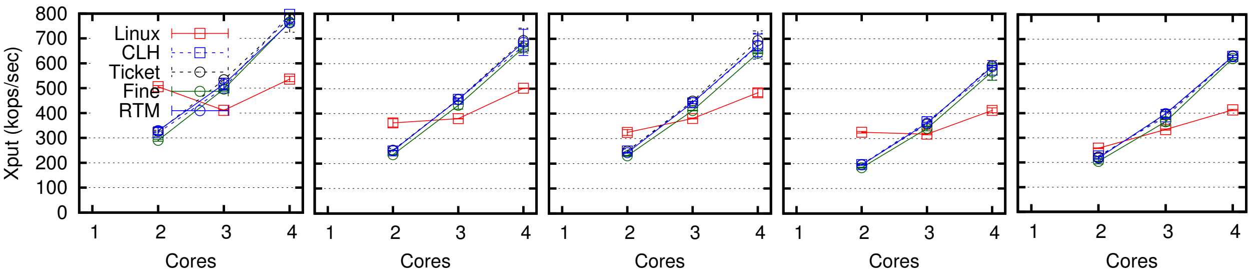
(a) Redis A (b) Redis B (c) Redis C (d) Redis D (e) Redis E×50

Figure 9: YCSB Redis benchmarks on x86. Benchmark E scaled by factor 50.



(a) Redis A (b) Redis B (c) Redis C (d) Redis D (e) Redis E×50

Figure 10: YCSB Redis throughput divided by average Redis core utilisation on ARM.



(a) Redis A (b) Redis B (c) Redis C (d) Redis D (e) Redis E×50

Figure 11: YCSB Redis throughput divided by average Redis core utilisation on x86.

demonstrate our system is capable of realistic performance levels, not to draw specific conclusions relative to Linux. We also note that Linux throughput on x86 is limited by the load generator, not the machine under test.

We make the following observations when comparing the throughput on ARM for different kernel configurations.

- For low numbers of cores, the benchmark results are consistent with the microbenchmarks in that a ticket-based BKL slightly outperforms the CLH, which slightly outperforms the fine-grained lock. While the relative performance is consistent across workloads, the difference at some points lies within a standard deviation of the results.
- For 4 cores, the greater scalability of fine grained locking overcomes some of the overhead of extra locking, but only manages to match the performance of the BKL variants.

On x86, we see the throughput is not discriminating between the difference kernel variants. Further investigation reveals that overall system throughput is limited by throughput of the lwIP core, and that the Redis cores have significant idle time.

To compare efficiency of similar throughput, we present Xput, i.e. throughput divided by average utilisation of the redis cores (the lwIP core was not discriminating). We only present configurations were redis runs on a dedicated core, i.e. the two to four core configurations. Figure 8 and Figure 9 shows the results in thousand of operations per second.

On ARM we see that Xput scales linearly with added cores while not significantly distinguishing between kernel variants. Each locking variant is as efficient as the other across all the workloads. On x86 we see kernel locking variant does affect Xput, with the BKL variants consistent outperforming fine-grained locking, though admittedly by a small margin. Finally, RTM maintains an Xput similar to other BKL variants.

7. Discussion

We have investigated the synchronisation properties of two closely coupled multicore processors, the Cortex A9 and the Intel Core i7-4770. We see that the ARM processor is very tolerant of lock choice given its low inter-core transfer costs. While differences are visible at extreme locking frequencies, they are likely indistinguishable at the frequency likely to occur in a microkernel. However, lock choice on x86, given its higher inter-core latencies is visible at realistic locking frequencies.

Course-grained versus fine-grained locking is distinguishable under extreme microkernel system call load. We see (as expected) that uncontended locks exhibit performance commensurate with their overhead. Surprisingly, it takes all for cores for the scalability of fine-grained locking to overcome the lower overhead of coarse-grained locking, and only under an extreme microbenchmark with near-zero think-time between each system call.

In our I/O intensive (system call intensive) macrobenchmark, we see that on ARM, coarse grained locking is at least equivalent performance to fine-grained lock, with the benefit of much less kernel implementation complexity.

To our knowledge, we're are the first to implement lock elision for a BKL kernel using Intel's RTM. We show in microbenchmarks, that a theoretically embarrassingly parallel application scales perfectly with little overhead and no serialisation. In our realistic macro benchmark, which is less parallel, RTM has similar results to other BKL variants indicating RTM to be an effective locking solution in both scenarios. RTM will still however be sensitive to abort rates, resulting in a higher overhead with the possibility of deteriorating into a slower BKL if the workload is not well suited for parallelisation.

8. Conclusions

Results show that the big kernel lock still has its place in one corner of the OS-architecture design space. In the OS dimension, this is the end domain of well-designed microkernels with predominantly short system calls. In the architecture dimension it is that of embedded-style closely-coupled multicores which a shared L2 cache and correspondingly low cache-migration and inter-core interrupt latencies, as well as core clusters of high-end processors with similar properties.

Our results show that in this scenario the BKL scales almost perfectly, with very low dependence on the actual lock implementation. They furthermore show that fine-grained locking, besides not improving scalability, adds overhead, and is outperformed by the much simpler BKL approach. The results, obtained on 4-core clusters, are likely to remain approximately valid for 8 cores, the likely limit of close coupling.

We believe that these results provide an important insight driving the design of microkernels for embedded multicores and clustered manycores. Specifically, as formal verification technology is presently unable to deal with the large degree of concurrency present in a kernel using fine-grained locking, our results imply that a verified BKL multicore kernel can be achieved without penalising performance.

References

Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *Trans. Parall. & Distr. Syst.*, 1 (1), Jan 1990.

AMBA Level 2 Cache Controller (L2C-310) Technical Reference Manual. ARM Ltd., r3p1 edition, 2010. ARM DDI 0246E.

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *22nd SOSP*, Big Sky, MT, USA, Oct 2009.

Bernard Blackham, Yao Shi, and Gernot Heiser. Improving interrupt response time in a verifiable protected microkernel. In *7th EuroSys*, pages 323–336, Bern, Switzerland, Apr 2012. doi: 10.1145/2168836.2168869.

Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *SOSP*, pages 1–17, Farmington, PA, USA, Oct 2013.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. Indianapolis, IN, USA, Jun 2010.

Travis S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report UW-CSE-93-02-02, Department of Computer Science and Engineering, University of Washington, 1993.

Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*, pages 33–48, Farmington, PA, USA, Nov 2013.

Kevin Elphinstone and Gernot Heiser. From L3 to seL4 – what have we learnt in 20 years of L4 microkernels? In *SOSP*, pages 133–150, Farmington, PA, USA, Nov 2013.

Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the Fluke kernel. In *3rd OSDI*, pages 101–115, New Orleans, LA, USA, Feb 1999. USENIX.

Freescale. *i.MX 6Dual/6Quad Applications Processor Reference Manual*, rev. 1 edition, Apr 2013.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *Trans. Comp. Syst.*, 32(1):2:1–2:70, Feb 2014. doi: 10.1145/2560537.

Jochen Liedtke. On μ-kernel construction. In *15th SOSP*, pages 237–250, Copper Mountain, CO, USA, Dec 1995.

lwIp. lwip. http://www.nongnu.org/lwip/.

John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronisation on shared-memory multiprocessors. *Trans. Comp. Syst.*, 9:21–65, 1991a.

John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *3rd PPOPP*, pages 106–113, 1991b.

Redis. Redis. http://redis.io.

twemproxy. twemproxy. https://github.com/twitter/twemproxy.

Michael von Tessin. The clustered multikernel: An approach to formal verification of multiprocessor OS kernels. In *2nd SFMA*, pages 1–6, Bern, Switzerland, Apr 2012.