For a Microkernel, a Big Lock Is Fine

Sean Peters, Adrian Danis, Kevin Elphinstone, Gernot Heiser
NICTA and UNSW Australia
{sean.peters, adrian.danis, kevin.elphinstone, gernot}@nicta.com.au

Abstract

It is well-established that high-end scalability requires fine-grained locking, and for a system like Linux, a big lock degrades performance even at moderate core counts. Nevertheless, we argue that a big lock may be fine-grained enough for a microkernel designed to run on closely-coupled cores (sharing a cache), as with the short system calls typical for a well-designed microkernel, lock contention remains low under realistic loads.

1. Introduction

For synchronising access to shared kernel state, the simplest approach is a *big kernel lock* (BKL), which is taken upon kernel entry and not released until kernel exit. The BKL has a reputation of poor performance owing to contention, even on moderate processor counts [Lehey 2001]. For achieving high-end scalability, minimising contention is essential [Clements et al. 2013], making a big lock unsuitable.

However, there are scenarios which inherently do not deal with high core counts. For example, SoCs designed for embedded systems only feature low to moderate core counts for now (up to 8), and optimisations designed for high-end scalability may be sub-optimal for such processors. Furthermore, for a microkernel that only provides fundamental mechanisms rather than general system services, high-end scalability is best achieved by a shared-nothing *multikernel* design [Baumann et al. 2009], which consequently avoids all locking in the kernel in favour of inter-kernel message passing.

The multikernel is not optimal for small numbers of cores that share caches, as the required inter-kernel communication is far more expensive than the communication through a shared cache between closely-coupled cores. For the seL4 microkernel [Klein et al. 2009], which is a general-purpose platform and presently mostly deployed on systems with a low to moderate core count, we therefore favour a *clustered multikernel* approach [von Tessin 2012]: a cluster of cores which shares a cache shares all kernel state, while across clusters sharing nothing.

The optimal locking strategy for such a system is far from obvious: seL4 is designed such that all frequently-used system calls (message-passing IPC and interrupt handling) are very short, resulting in much lower contention than on a monolithic kernel such as Linux. As shared-cache processor clusters are unlikely to scale to large core counts, a big lock might well be the best design.

We investigate the locking-granularity trade-offs for a microkernel running on cores sharing a cache (either as one cluster of a clustered-multikernel design or because the whole chip is closely-coupled). We find that in this scenario, the BKL remains competitive for at least 8-core clusters, which means that it remains the favoured design for microkernels, in stark contrast to monolithic OSes.

2. What's the Attraction of the Big Lock?

One might ask why we bother with the big lock at all. In order to compare its performance with fined-grained locking, we obviously need to implement the latter, so why not just keep it and stop worrying about scalability?

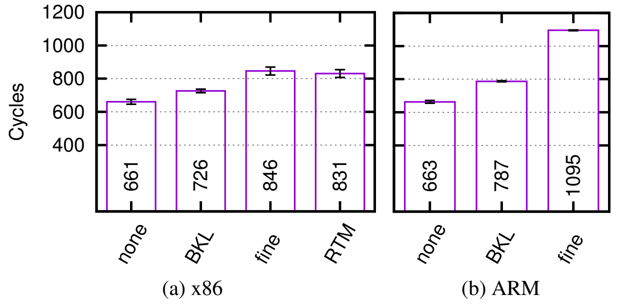

Figure 1: Raw round-trip IPC cycle cost for various locking strategies. Error bars indicate standard deviations.

There are two main concerns which favour the big lock: overhead and correctness.

APSys '15, July 27 - 28, 2015, Tokyo, Japan.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3554-6/15/07…$15.00.
http://dx.doi.org/10.1145/2797022.2797042

2.1 Lock overhead

Each lock introduces overhead, as demonstrated in Figure 1. Here we measure the (hot cache) round-trip IPC costs on a single core with different locking strategies applied to the seL4 kernel on the x86 and ARM architectures (see Section 4.1 for platform details). "BKL" and "fine" refer to applying single-lock and fine-grained locking, respectively, while "RTM" refers to the use of hardware transactional memory for synchronisation. "None" is identical to "BKL" with the lock code compiled out. We will discuss details of the lock implementations later (Section 3).

The figure shows that the overhead of a single lock is 65 cycles, or 10%, on x86 and 124 cycles, or 20% on ARM. The overhead of fine-grained locking is three times as high (for a total of 4 lock acquisitions per system call), about 30% for x86 and 60% for ARM over the single-core performance. The extra cost of fine-grained locking is clearly significant, and IPC costs are critical to the performance of microkernel-based systems [Liedtke et al. 1997]. It is *a priori* unclear whether the reduced contention makes this overhead worthwhile.

Similarly for the RTM lock: it is inherently more expensive. As hardware-supported transactions protect at a cache-line granularity, RTM should also reduce contention compared to the BKL, and the question is whether this will outweigh the higher baseline cost.

2.2 Correctness

Fine-grained locking introduces concurrency into the kernel, and concurrency is notoriously hard to get right [Godefroid and Nagappan 2008]. Even for the small, 9 kLOC codebase of seL4 (of which only about 1,500 LOC is scalability-critical IPC or interrupt-handling code) we learned that it is hard to get an optimised fine-grained locking implementation right. In fact, even after months of work we are still not convinced that we have it 100% correct.

seL4 is designed as a high-assurance system for safety- or security-critical uses, so correctness is the number-one concern. The kernel has been comprehensively formally verified for functional correctness and security enforcement [Klein et al. 2014], but this applies only to the single-core version. Verification of a single-lock version is feasible [von Tessin 2013], but our present verification approaches cannot deal with the concurrently-executed code fine-grained locking produces. Our verification team estimates that extending the verification to a kernel version with fine-grained locking will far exceed the cost already paid for verifying the single-core version.

3. Locking seL4 state

3.1 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 exit code, which saves and restores the user-state to a per-core kernel stack and sets up safe kernel execution, remains outside of the BKL, while the rest of the kernel is protected by the BKL.

This design is not entirely sufficient – the following invariant, used in the verification, 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.

User-level code executing on other cores implicitly depends on the running thread's TCB and page table to transition to kernel-mode via the kernel entry code to compete for the BKL. The invariant therefore no longer holds. We address this by modifying the kernel to ensure remote cores are not dependent on any TCB or page-table undergoing deletion. Our prototype currently takes a naïve approach of triggering remote cores to enter the kernel idle loop (which has a permanent TCB and page-table) using an IPI.

The BKL 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 shoot-down and (ii) introducing per-core idle threads. In order to minimise inter-core cache-line migrations, we also introduce per-core scheduler queues and current-thread pointers, even though access is serialised by the BKL.

To reduce contention (and enable the use of transactional memory, see Section 3.3.1) we further minimise the amount of locked code by moving context-switch-related hardware operations after the BKL release.

Our BKL implementation uses a single CLH lock [Craig 1993]. A CLH lock is a scalable queue-based lock that spins on a local cache-line when waiting. We also experimented with ticket locks, but they are non-scalable and do not provide a performance benefit [Boyd-Wickizer et al. 2012].

3.2 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 big reader lock [Corbet]. The lock allows all reader cores to proceed in parallel as they access only local state to obtain a read lock. Data structures now exposed to concurrent writes are protected using individual fine-grained locks.

IPC mutates the state of TCBs, endpoints, and (potentially) the scheduler queues (depending on whether optimisations apply that avoid queue updates during IPC [Elphinstone and Heiser 2013]). We add ticket locks (i.e. fine-grained locks) to each of these data structures for synchronising 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 performing IPC 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.

We use a write-lock selectively in our present prototype to avoid significant code changes for synchronising concurrency for paths that are not performance critical. Typical OS functionality (e.g. networking) is implemented at user-level, and thus most microkernel system calls are used much less frequently than IPC or interrupt delivery. Thus the choice of a writer lock to synchronise less frequent system calls is both pragmatic and performance-neutral.

At minimum, this design retains the de-allocation of kernel objects within the writer lock. The benefit is that existing memory safety is retained while holding the reader lock, though contents of the objects themselves are be exposed to concurrency for improved scalability.

3.3 Hardware transactional memory

3.3.1 Architectural support

Starting with the Haswell microarchitecture, Intel processors feature an implementation of *restricted transactional memory* (RTM), called Intel TSX. We use this to implement our RTM lock.

TSX provides 3 new 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 are any memory conflicts during the execution of the transactional region, the transaction will abort and jump to the instruction specified by the `XBEGIN`. A program can explicitly abort a transaction by issuing an `XABORT` instruction.

TSX takes 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 the hardware-limited L1 cache size. The latter consequence is an indication that it is probably not feasible to wrap a complete monolithic kernel into an RTM transaction, as it is unlikely to fit within the L1.

Owing to the implementation of TSX, the RTM lock logically locks a dynamic set of individual L1 cache lines, and as such is a fairly extreme case of fine-grained locking, which should result in much reduced contention (assuming a sane layout of kernel data structures).

Note that an RTM transaction is not guaranteed to complete, even when the transaction is small enough and has no memory conflicts. A variety of (hardware-implementation specific and frequently unspecified) scenarios can result in an abort. Of particular interest to our work are certain interactions on specific registers that trigger aborts, but are clearly unavoidable when executing OS code.

Given transactions have no guarantees of progress, the developer must ensure that there exists a fallback method of synchronisation that ensures progress in the presence of repeated aborts. We use a commonly implemented technique of falling back to a regular lock for the code fragment in the case of repeated aborts. To avoid races between a transaction-protected and lock-protected fragment, our transactions test the lock upon entry to an RTM code fragment to ensure the lock is free and in the read set of the transaction attempt. A change in lock state by a competing thread will trigger the desired abort, and allow the fragment to synchronise via the lock.

3.3.2 RTM lock implementation

The TSX extensions, combined with the small size of the kernel, allow us to optimistically execute the majority of the code without concurrency control. This is enabled by the two-phased system call structure of seL4 [Klein et al. 2009], where the first phase validates the pre-conditions for execution and the second phase is guaranteed to execute without failure. Also important for this design is the event-based design of the kernel, which avoids blocking. We bracket almost the entire kernel with the transaction primitives.

In addition to the changes described in Section 3.1, we need to move any TSX-specific abort-triggering CPU operations after the transaction. Many of those do not occur in seL4, as most aborting operations are typical for device drivers, which are user-level programs in seL4. The remaining problematic operations are:

- context-switch-triggered page-table register (CR3) 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 is that it is safe to move these operation outside of the transaction, because the two-phase kernel ensures the system call which requires these operations is guaranteed to succeed once the execution phase is entered, and that these operations are local to a core and thus are not exposed to concurrent access from other cores.

4. Evaluation

4.1 Platforms

4.1.1 x86 platform

As an x86 platform we use a desktop Dell Optiplex 9020, which features a Q87 Express chipset with an Intel Core i7-4770 processor. This is a quad-core processor with a clock rate of 3.4 GHz and two hardware threads each, giving 8 hardware threads in total. It is also a representative of the Haswell microarchitecture, and as such supports Intel's TSX, which we use for the RTM lock implementation (see Section 3.3).

The processor features three 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 private, non-inclusive, 8-way 256 KiB L2 cache. An inclusive, 16-way, 8 MiB L3 cache is shared between all cores.

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

4.1.2 ARM platform

Our ARM platform is the Sabre Lite, which is based on a Freescale i.MX 6Q SoC, featuring a quad-core ARM Cortex-A9 MPCore processor.

The cores run at a 1 GHz clock rate and have private, split L1 caches, each 4-way-associative and 32 KiB in size. The cores share a 1 MiB, unified, 16-way-associative L2 cache, which is the last-level cache. Typical L1 access time is 1–2 cycles, L2 access time is 8 cycles [ARM 2010]. The platform has 1 GiB of main memory, we measure the access time 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*).

4.2 Microbenchmarks

As IPC performance is a key contributor to overall system performance in microkernel-based systems, and thus optimising IPC performance has a long history in the L4 community [Elphinstone and Heiser 2013]. The traditional benchmark for best-case IPC performance is "ping-pong": a pair of threads on a single core does nothing other than sending messages to each other. This allows us to asses the basic cost of our lock implementations, i.e. the pure acquisition and release cost, without any contention.

We extend single-core ping-pong to multiple cores (including hyperthreads). Specifically, we run a copy of ping-pong on each hardware thread, with all hardware threads executing completely independently and unsynchronised.

This benchmark produces extreme contention on the kernel (almost zero user-level execution time). However, *none of the kernel data structures are contended*, as each hardware thread's pair of software threads accesses disjoint kernel objects (TCBs and IPC endpoints) during their syscalls. Hence, while maximising contention on the BKL, fine-grained locking and RTM can be expected to scale perfectly.

The lack of concurrent accesses to shared data furthermore allows us to run this benchmark for a baseline that shows optimal performance of a theoretical, perfect, zero-overhead fine-grained lock: We use an (unsafe!) kernel implementation without any locking.

4.3 Macro-benchmark: Redis

Multicore ping-pong presents an unrealistic case with no user-level work that is a worst case for the BKL. In order to assess the BKL scalability, and the significance of the overheads of the fine-grained schemes, we look for a "realistic worst-case" scenario, i.e. a benchmark which produces as high a system-call rate as can be expected under realistic conditions.

None of the usual embedded-system benchmarks produce significant syscall loads on the microkernel, we therefore use a server-style benchmark. Note that the nature of the benchmark is completely irrelevant for this exercise, all that counts is the rate and distribution of kernel entries. The relevant operations are IPC and interrupt handling, as all other microkernel operations deal with resource management that is relatively infrequent.

In fact, the seL4 equivalence of a syscall in a monolithic system is sending an IPC message to a server process and waiting for a reply (i.e. two microkernel IPCs per monolithic OS syscall). Similarly, an interrupt, which in a monolithic OS results in a single kernel entry, produces two for the microkernel-based system, as the interrupt is converted by the kernel into a notification to the driver (one kernel entry), and the driver acknowledges to the kernel with another syscall.

In order to hammer our kernel, we use a simple server scenario, consisting of the Redis key-value store [Redis]. It receives client requests from the network, using a (user-level) Ethernet driver and a port of the lwIP TCP/IP stack [lwIp] to run as a usermode process.

When running on a single core, the setup consists of three processes: driver, network stack and Redis.

For multicore setups, Core 0 has the same configuration, while all other cores (or hardware threads) run their own copies of lwIP and Redis. All interrupts go to the Ethernet driver on Core 0, which de-multiplexes incoming packets to the network stacks based on port number.

Note that we run Redis as volatile instances (we disabled file system access), as our prototype lacks file system support.

We evaluate performance using the Yahoo! Cloud Serving Benchmarks (YCSB) [Cooper et al. 2010], running on a dedicated pair of load generator machines, with a dedicated Gigabit Ethernet network between the load generators and the machine under test. On each load generator we run one instance of the YCSB benchmark per Redis instance on the target machine. All YCSB benchmark instances start simultaneously and are tuned to perform a number of operations that would result in approximately 30 seconds of run time. We also increase the `recordcount` to 32000, to entirely fill our prototype's memory limitations.

YSCB consists of several workloads. We show the results of workload A as presented in Cooper et al. [2010]. This workload is an update-heavy workload (50/50 read and writes) that uses zipfian distribution for record selection in the store.

We also include a Linux-based Redis configuration for comparison. We use Linux 3.13.0, configured at run-time using /sys/devices/system/cpuX/online to use one to eight cores, together with a Redis instance per core.

5. Results

5.1 Microbenchmarks

We collect results from 16 one-second runs of the ping-pong benchmarks and calculate the mean and standard deviation.

5.1.1 Single-Core IPC Microbenchmarks

Single-core ping-pong results are shown in Figure 1, which was already discussed early-on, showing the contention-free locking cost: about 10% overhead for the BKL and 30% for fine-grained locking and transactions on x86, and about twice that on ARM.

The higher synchronisation costs on the ARM processor relate to its partial-store-order memory model. It 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 micro-architectural state. Our implementation of BKL executes 6 barriers on this benchmark, while 16 are needed with fine-grained locking. These barriers explain most of the overhead.

As mentioned in Section 2, the significant cost of fine-grained locking provides a motivation for sticking with the BKL as long as possible, even if verification tractability was no issue.

5.1.2 Multicore IPC Microbenchmarks

We run multicore ping-pong with a two-second warm-up, followed by sampling total IPCs during a one second interval to give total IPC round-trip throughput per second. We repeat each benchmark 16 times and we report the mean and standard deviation (as error bars) in Figure 2.

In the figure, core-counts > 4 for x86 correspond to the use of hyperthreading: Cores$= 4+i$ means i cores have both hardware threads enabled, while the remaining ones just use a single hardware thread.

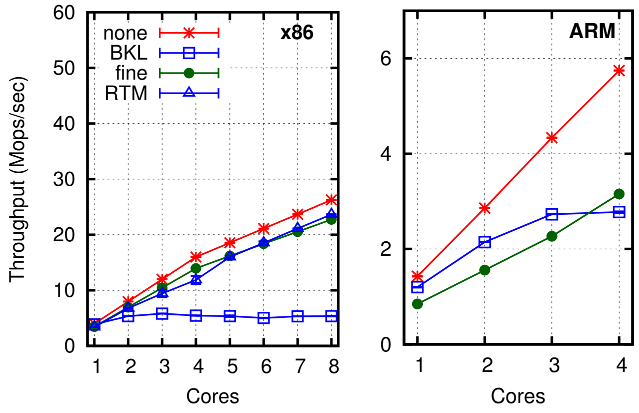

Figure 2: Synchronous intra-core IPC round-trip throughput on x86 (left) and ARM (right), error bars are too small to be visible.

Note that for the single-core case, these throughput figures are slightly less than the inverse of those of Figure 1, as the latter are pure best-case kernel times, while the throughput figures include minimal userland code.

The "none" case here corresponds to the theoretical optimal case (using the unsafe lock-free implementation). As expected, this case scales perfectly on both architectures, although hyperthreading adds somewhat less throughput than real cores.

Fine-grained locking also scales perfectly, as the locks are not contented in this case. Throughput is somewhat degraded (compared to the baseline), owing to the lock overhead.

On x86 we observe that RTM behaves identically to fine-grained locking. This is also expected: as explained in Section 3.3.1, RTM is logically an extreme case of fine-grained locking, and the baseline lock overhead is that same as for the fine-grained locks according to Figure 1.

The BKL variant serialises the IPC path across all the available hardware threads and has very little parallelism available to take advantage of the available hardware. Its performance plateaus after 3 cores.

On ARM, we see similar behaviour, except the higher overhead for fine-grained locking is readily visible, and the BKL variant outperforms fine-grain locking up to 3 cores, prior to plateauing.

Remember that this benchmark measures a pathological case of no work done at user level, it represents an unrealistic worst-case scenario for the BKL.

5.2 Macrobenchmark

We run the Redis benchmark on our x86 platform. For each combination of kernel variant and number of cores, we run YCSB three times and report mean and standard deviation. We instrument the kernel to record idle time within the idle loop for obtaining CPU utilisation for each run.

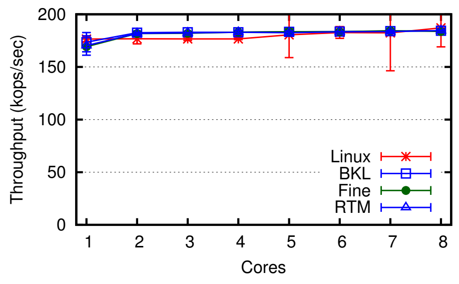

Figure 3: YCSB Redis A benchmark.

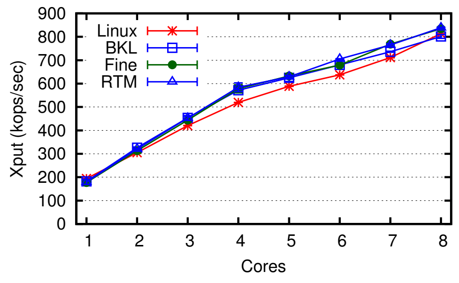

Figure 4: Redis throughput divided by average core utilisation.

Figure 3 shows mean throughput for varying number of cores. Standard deviations are shown as (except for Linux nearly indistinguishable) error bars.

The figure shows that throughput is independent of the kernel variant. Further investigation reveals that overall system throughput is limited by the network bandwidth, and that the all cores have significant idle time.

To compare efficiency of similar throughput, we show in Figure 4 the normalised “Xput” value, which we define as throughput divided by average utilisation of all cores. This represents the throughput for a fixed processing cost, and is thus the right figure of merit to compare on differently loaded processors.

We see that all seL4 variants perform very similarly. The BLK kernel drops slightly (again, barely outside the error bars) below that of the other seL4 variants, but does not show the performance cliff indicative of significant contention. The results indicate that for 8-way parallelism, and likely beyond, the choice of lock is essentially irrelevant to performance.

6. Related Work

Writing parallel and scalable code is a topic almost as old as computing itself. Cantrill and Bonwick [2008] provide some historical context and motivation for concurrent software, together with words of wisdom to tackle the difficulties of writing high-performance and correct concurrent software. We adhere to their advice by avoiding parallelising complex software (i.e. splitting the BKL) as our data shows it is unwarranted.

Recent complementary work evaluates the scalability of various synchronisation primitives [David et al. 2013] on many-core processors. The authors reinforce that scalability is a function of the hardware, with scalability best when access is restricted to a single socket with uniform memory access – exactly our area of interest.

Hardware transactional memory is utilised in TxLinux [Rossbach et al. 2007] to implement *cxspinlocks*, a combination of co-operative spinlocks and transactions capable of supporting device I/O and nesting. A small microkernel needs neither, as I/O is at user-level, and it can be designed to avoid complex, nested, fine-grained locks.

Patches for Linux to utilise Intel TSX have been made available [Kleen]. To our knowledge, no performance data was released. Eliding existing fine-grained locking does nothing to reduce kernel complexity. We elide the whole microkernel, providing favourable performance while retaining simplicity.

7. Conclusions

Our initial results show that the big kernel lock remains an attractive approach for synchronising access to shared state in a microkernel, at least for hardware sharing caches and moderate core counts. On our realistic (but tough) benchmarks, more sophisticated locking schemes fail to demonstrate performance advantages.

While at odds with scalability folklore, this result is fundamentally neither surprising, nor is it limited to kernel code. A big lock can be expected to scale well, as long as two conditions are met: (1) lock acquisition and hold times are short (i.e. inter-core cache latencies and critical sections are small), and (2) inter-lock times (i.e. concurrent execution phases) are long in comparison. What may be a bit more surprising is that these conditions can be met in an OS kernel: they will not be met for a monolithic kernel under realistic loads, but they can be met for a well-designed microkernel.

We intend to explore this space further by running macrobenchmarks on ARM and also on closely-clustered platforms with higher core counts. It might be possible that we need to limit cluster size, and move to the clustered-multikernel design even on large shared-cache multiprocessors. This would be an interesting trade-off to explore.

References

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 *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, US, October 2009.

Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Pro-*

ceedings of the 2012 Ottawa Linux Symposium, Ottawa, CA, July 2012.

Bryan Cantrill and Jeff Bonwick. Real-world concurrency. *ACM Queue*, 6(5), September 2008.

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 *ACM Symposium on Operating Systems Principles*, pages 1–17, Farmington, PA, US, October 2013.

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

J. Corbet. Big reader locks. http://lwn.net/Articles/378911/.

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 *ACM Symposium on Operating Systems Principles*, pages 33–48, Farmington, PA, US, November 2013.

Kevin Elphinstone and Gernot Heiser. From L3 to seL4 – what have we learnt in 20 years of L4 microkernels? In *ACM Symposium on Operating Systems Principles*, pages 133–150, Farmington, PA, USA, November 2013.

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

Patrice Godefroid and Nachiappan Nagappan. Concurrency at Microsoft – an exploratory survey. In *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, Princeton, NJ, US, July 2008.

Andi Kleen. RFC: Kernel lock elision for TSX. https://lkml.org/lkml/2013/3/22/630.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

Greg Lehey. Improving the FreeBSD SMP implementation. In *Proceedings of the 2001 USENIX Annual Technical Conference, FREENIX Track*, Boston, MA, US, June 2001.

Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for extensibility). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 28–31, Cape Cod, MA, USA, May 1997.

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

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

Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. TxLinux: Using and managing hardware transactional memory in an operating system. In *ACM Symposium on Operating Systems Principles*, Stevenson, WA, US, October 2007.

Michael von Tessin. The clustered multikernel: An approach to formal verification of multiprocessor OS kernels. In *2nd Workshop on Systems for Future Multi-core Architectures*, pages 1–6, Bern, Switzerland, April 2012.

Michael von Tessin. *The Clustered Multikernel: An Approach to Formal Verification of Multiprocessor Operating-System Kernels*. PhD thesis, School of Computer Science and Engineering, UNSW, Sydney, Australia, Sydney, Australia, December 2013.