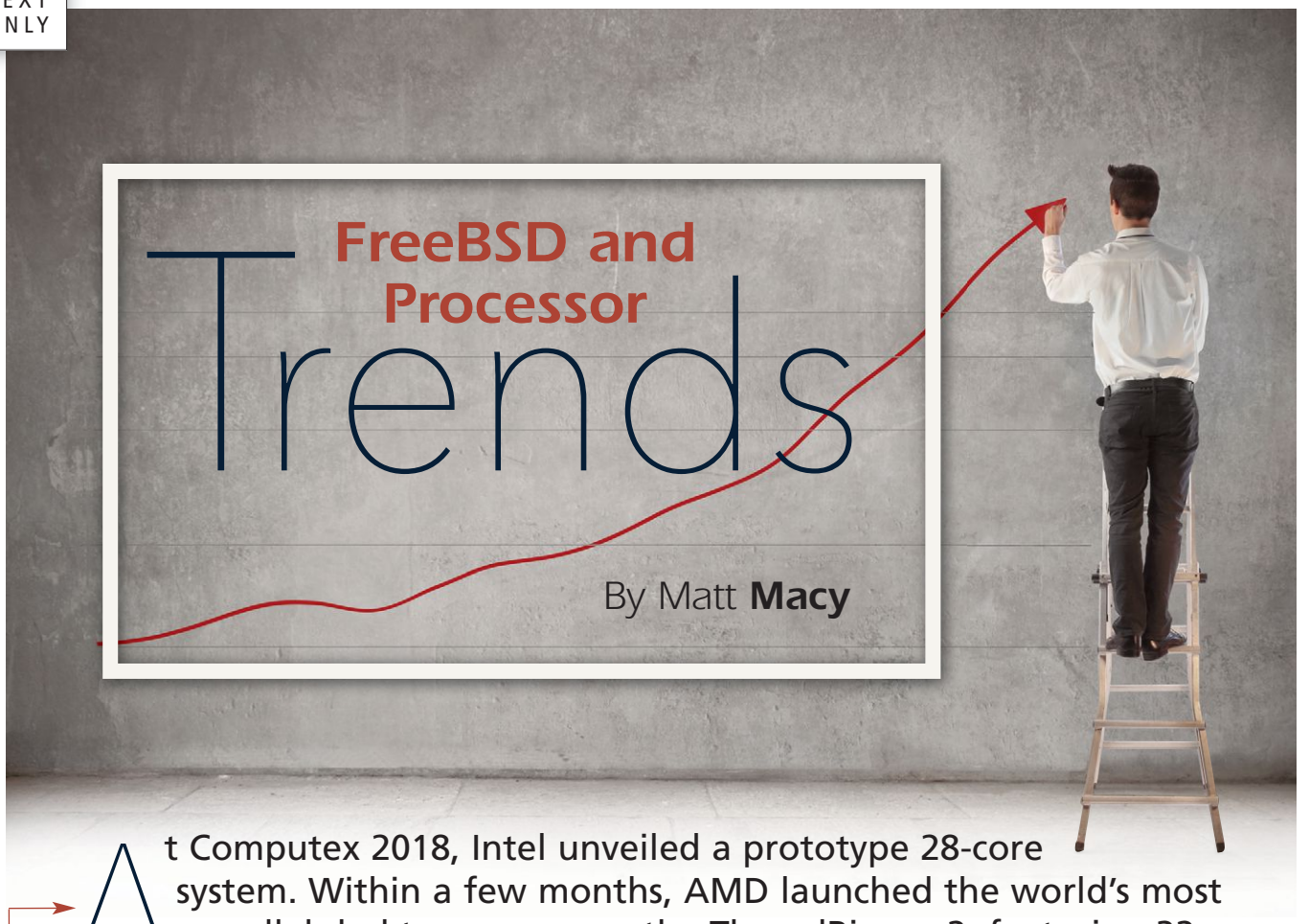
SEE TEXT ONLY

FreeBSD and Processor Trends

By Matt **Macy**

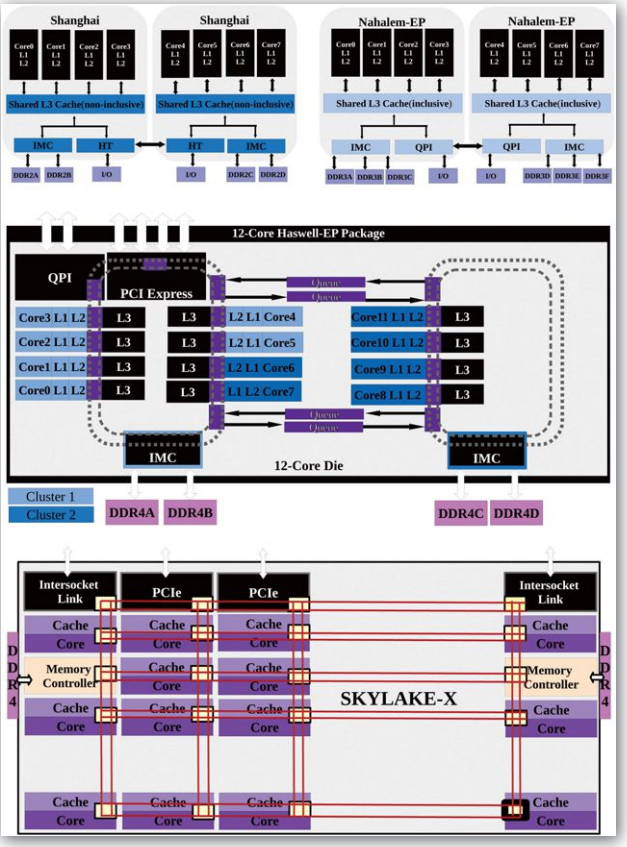


At Computex 2018, Intel unveiled a prototype 28-core system. Within a few months, AMD launched the world's most parallel desktop processor, the ThreadRipper 2, featuring 32 cores (64 hardware threads). AMD's EPYC2 is in the lab and rumored to be 64 cores (128 hardware threads), bringing 256 hardware threads to a commodity server dual socket system. Historically, FreeBSD has existed at the "knee" of the hardware commodity curve. In order to maintain its relevance in the server space, FreeBSD needs to keep pace with the latest processor developments.

Processor Evolution

As core count has increased, the designs have gotten steadily more complicated. AMD's Shanghai and Intel's Nehalem used a broadcast bus for handling cache coherence. Intel's Haswell later changed this to multiple rings on chip. And with Skylake, Intel has moved to a mesh **(see right)**.

AMD has taken a yield-centric focus to scaling up by reusing the same design across its product



line. Each chip consists of two core complexes (CCX), and an EPYC package consists of four chips (often referred to as "chiplets") **(see lower right)**.

Defining Scalability

Scalability can be defined on a number of axes [Culler, 1999]:

- Problem-Constrained **`strong scaling`** - The user wants to use a larger machine to solve the same problem faster. As the number of processors available to complete a task increases, the extent to which the time completes the problem decreases:

$$\text{Speedup}_{PC}(n \text{ processors}) = \frac{\text{Time}(1 \text{ processor})}{\text{Time}(n\text{processors})}$$

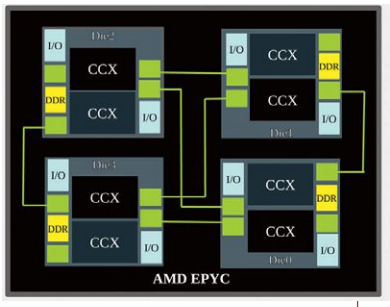
- Time-Constrained **`weak scaling`** - the time to execute a given workload remains constant; user wants to solve the largest problem possible. It is the degree to which the amount of work accomplished increases as the number of processors increases:

$$\text{Speedup}_{TC}(n \text{ processors}) = \frac{\text{Work}(n \text{ processors})}{\text{Work}(1 \text{ processor})}$$

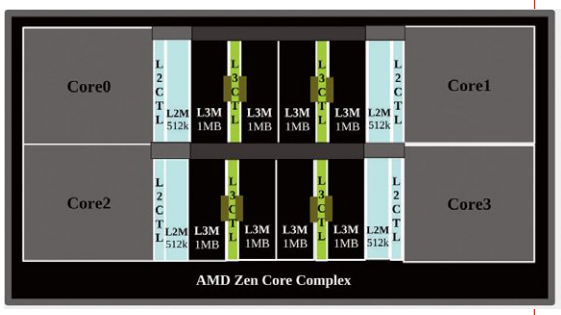
- Memory-Constrained - The user wants to solve the largest problem that will fit in memory.

$$\text{Speedup}_{MC}(p \text{ processors}) = \frac{\text{Work}(p \text{ processors})}{\text{Time}(p \text{ processors})} \times \frac{\text{Time}(1 \text{ processor})}{\text{Time}(1 \text{ processor})} = \frac{\text{Increase in Work}}{\text{Increase in Execution Time}}$$

In this article, scalability refers to time-constrained scalabilty ("weak scaling"), which will be characterized by the aggregate number of operations performed during benchmarks. Performance bottlenecks are application- and work-load specific. Therefore it is problematic to extrapolate actual application performance from these scalability measurements. Nonetheless, the OS impact on any given workload can be characterized as a combination of the average time per system call and the impact of scheduling decisions. System call overhead can be captured by simple microbenchmarks. Scheduling decisions are harder to measure but one can measure them to a limited degree by measuring workloads with varying scheduler restrictions (i.e., limiting the set of CPUs the scheduler can use) or by comparing single socket results with dual/multi-socket results.



It is important for the reader to understand that the purpose of microbenchmarks is not to measure workloads themselves. They are a means to observe the scaling of individual OS services to measure scalabilty in isolation. These measurements are only predictive of performance on real world workloads to the extent to which a workload uses the individual service being measured.



What Makes Scaling Difficult—Serialization and Scheduling

If *n* threads are attempting to perform an operation, serialization overhead can roughly be defined as the extent to which the throughput per thread declines from 1 to 1/*n*, as each thread waits to acquire the same lock. **`Scheduling overhead`** is more difficult to define. In an ideal world a given thread would only ever run on one core; any other threads that it communicated with would be on the same "core complex" - sharing an L3 cache so that IPIs (inter processor interrupts—a facility to allow a cpu to interrupt other cpus) and cache coherency traffic would not have to traverse an interconnect and any misses could be refilled without going to memory. Unfortunately, in practice, this is impossible in the general case. CPUs are commonly oversubscribed and the scheduler cannot infer relationships between threads in different processes. The further away one cpu is from another, the poorer the performance for latency-sensitive operations such as networking and synchronous IPC. And the larger the distance between two cpus, the greater cost of refilling the caches when a thread is migrated.

The scaling challenges stem from three factors:

- memory latency
- limits to coherency traffic
- shared globally unique resources

To a large degree the scaling solutions are all a combination of:

- per cpu resources
- relaxing constraints
- distinguishing between existence guarantees and mutual exclusion

Memory latency and the bounds on coherency traffic are fundamental to the evolution of computer hardware over the last decade. What were once design artifacts seen only in high-end systems are now an important consideration even in consumer CPUs like AMDs ThreadRipper. The shared memory programming model is becoming an increasingly leaky abstraction. Cache coherence logic in processors provides the single-writer /multiple-reader `SWMR` guarantees that programmers are all accustomed to [Sorin, 2011]. However, at its limit, the observed performance is defined by the actual implementation of a distributed memory with all updates performed by message passing [Hacken, 2009], [Molka, 2015]. Today, message latency and bandwidth are dominant factors in observed performance.

Implementation issues impacted by the increasing number of hardware threads are:

- locking granularity
- using locks to provide existence guarantees
- using atomic references to provide existence guarantees
- poor cache locality between L3 caches or NUMA domains.

Locking Granularity

Locking granularity refers to how many operations are protected by a single lock. The "Big Kernel Lock" or "BKL" in Linux or "Giant" in FreeBSD initially encompassed the entire kernel in a single lock. This evolved into locks for individual subsystems, then individual data structures, and finally fields in data structures. Even with fine-grained locking, the case of a widely referenced global resource (memory, routing table entry, etc.) that can only be accessed one at a time occurs. In general, locking granularity in FreeBSD is already relatively fine-grained. Nonetheless, between FreeBSD 11 and FreeBSD 12 there are numerous examples of work done to reduce lock contention, either by increasing locking granularity, moving to per-cpu resources, or reducing the frequency with which global updates occur.

Locking for Existence Guarantees

One can use a lock to guarantee that entries in a system global or process global structure have not been freed while in use. One example in FreeBSD 11.x vs FreeBSD 12.x is how existence is guaranteed for connection state within the per protocol hash table. FreeBSD 11.x guarantees a thread that any connection found in the table is valid by requiring that all table readers do a shared (for read) acquisition of a per-table reader/writer lock. This allowed multiple simultaneous readers while preventing any table updates. Although conceptually straightforward, this comes at a substantial price and the guarantee is stronger than required. FreeBSD 12.x weakens the guarantee to provide that any connection found during a lookup had not been freed. Lookups are protected with epoch and updates are serialized with a mutex. Connection state lookup still returns the connection locked to guarantee existence past lookup. However, once the lock is acquired, lookup now checks that the connection has not had the `INP_FREED` flag set. If the flag is set, this indicated that connection is pending free. In this case, we drop the lock and return `NULL` as if no connection had been found. This change adds some additional complexity to readers, but in exchange we no longer require a global atomic for the rwlock [App. A] and updates can proceed in parallel with lookups (lookups no longer block on updates and vice versa). This change provided a 10–20x reduction in time spent in lookups on a loaded multi-socket server.

Atomic Refcounts for Existence Guarantees

Atomically updating a reference counter for an object peforms better than using a lock to serialize updates. Updates can proceed fully in parallel with ownership changes. Each new thread or object holding a pointer to the object increments the reference. When the reference is removed from the object or the thread's reference goes out of scope the reference is decremented. When the count goes to zero the referenced object is freed. Nonetheless it does not scale as the cost of coherency traffic rises. For an object frequently referenced by many threads the coherency traffic invalidating and migrating the cache line between L2 and L3 caches quickly becomes a bottleneck. There are two separate issues to address here:

- Is reference counting necessary here?
- Can anything be done to make reference counting cheaper?

Perhaps suprisingly, for stack local references, reference counting isn't actually necessary. SMR "Safe Memory Reclamation" techniques such as Epoch Based Reclamation [Fraser, 2004], Hazard Pointers [Michael, 2004; Hart, 2006], RCU [McKenney, 2011], scalable parallel sections [Wang, 2016], etc., can allow us to provide existence guarantees without any shared memory modifications. And reference counting can, in many cases, be made much cheaper.

Recent work in UDP expanded the scope of objects tied to the network stack's epoch structure. Epoch structure is now also used to guarantee existence of interface addresses. This now means that references to them that are stack local no longer need to update the object's refcount.

The observed reference count can safely be different from the "true" reference count if we can safely handle zero detection correctly. The different approaches to scalable reference counts rely on this insight. Although there are other approaches to this in the literature [Ellen, 2007], the ones I consider most interesting are Linux's percpu refcount [Corbet, 2013] and Refcache [Clements, 2013]. The former is a per-cpu counter that degrades to a traditional atomically updated reference count when the initial reference holder "kills" the perpcpu refcount. Its advantage is that it is simple and can be extremely lightweight provided that the life cycle of the object closely mirrors that of the initial reference holder. It does not work well if the object substantially outlives the initial owner. Refcache maintains a per cpu cache of reference updates and flushes them when there is conflict or at the end of an "epoch." In this case an "epoch" is 10 milliseconds. Zero detection is done by putting the object on a per-cpu "review" list when its global reference count reaches zero. The global reference count can be assumed to be the true reference count when it has remained at zero for two "epochs." Refcache doesn't rely on an initial reference holder with a closely correlated life cycle to avoid a degraded state. In some respects this makes it much more general. However, potential multiple passes through the review queue can add substantial overhead to the zero detection process. The latency between initial candidate for free and final release makes it unsuitable for objects with a high rate of turnover. For example, a ten millisecond backlog of network mbufs or VM page structures could incur punitive overhead.

Cache Locality

A simple example of designing for cache locality is packing structures contiguously as opposed to a linked list so that the prefetcher can furnish the next element as a thread iterates through them. At high operation rates, the way in which fields are ordered within a structure can make a measurable performance difference. A 45% increase in brk calls per second was measured when a reorganization of the core memory allocation structure reduced from three to two the number of cache lines for the most commonly accessed fields. Once serialization bottlenecks are eliminated, kernel performance is determined by the frequency of cache misses.

Minimal sharing and cache misses are easily definable ideals for serialization and locality. However, algorithmically defining optimal scheduling for an arbitrary hitherto unseen workload is an unrealizable ideal. Even where the information is present, data structure knowledge and sharing afforded to the scheduler slows down scheduling decisions. When sharing an L2 cache, two communicating threads with a small working set will benefit while two communicating threads with a larger working set will be adversely impacted. A particularly egregious example of where FreeBSD falls down due to thread scheduling on multiple sockets is measured throughput on TCP connections to localhost. On a ~3Ghz single socket system FreeBSD can achieve 50-60Gbps, partly by offloading network processing to *the* `netisr` thread. On a dual socket system of the same clock speed, the measured throughput drops to 18–32Gbps. In the worst case, two communicating processes are on one socket and the `netisr` thread is on the other. Therefore the notification for every packet has to cross the interconnect between sockets. At least on a dual socket, Linux does network processing inline (i.e., no service thread) when doing TCP to localhost. On a single socket Linux would achieve lower throughput than FreeBSD does. However, it achieves a consistent 35Gbps provided both processes are scheduled on the same socket. There are a number of issues to address here:

- the existence of only one `netisr` thread for the entire system
- where the sender, receiver, and `netisr` thread should be scheduled
- how to convey to the scheduler that the three different threads are communicating with each other.

Nonetheless, the key insight here is that latency can determine usable bandwidth and poor scheduling decisions can have a devastating impact on performance when we move from single socket to dual socket.

For users who understand in advance what workloads they will be running, the situation is manageable. The `cpuset` command allows one to assign processor sets to processes, restricting the choices that the scheduler has available to it.

Measuring Scalability

For purposes of this article, scaling measurements will be limited to running the "Will-it-scale" system call microbenchmark suite [Blanchard, 2013] on FreeBSD 11, FreeBSD 12 and Ubuntu 18 (Linux-4.15.1) on two systems—a dual socket EPYC 7601 (2x32 cores @2.2Ghz) and a dual socket Intel Xeon 6130 (2x16 cores @2.1Ghz). The two cannot be directly compared as the EPYC 7601 is a top bin processor retailing for 130% more than the mid-level Xeon 6130. Nonetheless, the more complex EPYC is likely to show very different scaling characteristics and a higher penalty for poor locality.

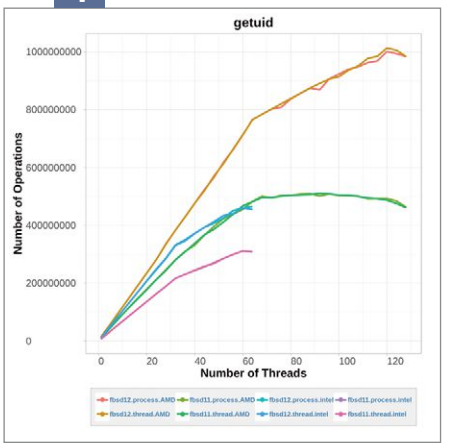
The first thing of note is that the multithreaded variants of most benchmarks scale much more poorly than their multi-process counterparts.The shared address space, file descriptor array, and proc structure all require added locking for the multithreaded case. In the cases where the benchmarks do scale, the curve typicaly flattens at the halfway mark. This is because we move from a regime of one benchmark thread or process per core to oversubscribing the cores and using both hardware threads **(1)**.

There are a number of notable improvements going from FreeBSD 11 to FreeBSD 12.The getuid benchmark shows that system call overhead has been reduced by more than 50%. Page fault performance has improved by 20–80x **(2, 3, 4)**.

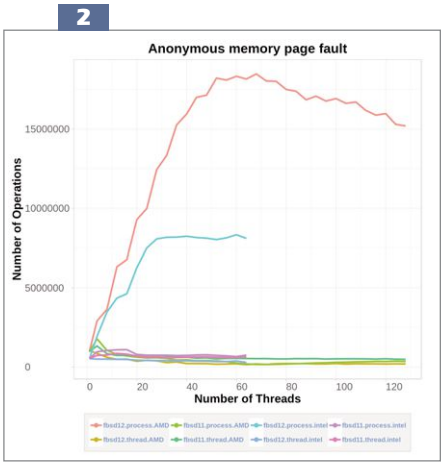
Anonymous memory mmap/munmap of 128MB has improved substantially and currently outperforms Linux **(5, 6)**.

Unix domain socket performance has improved by 19x. Performance previously flattened out at eight hardware threads, but now continues to increase up to 128 hardware threads **(7)**.

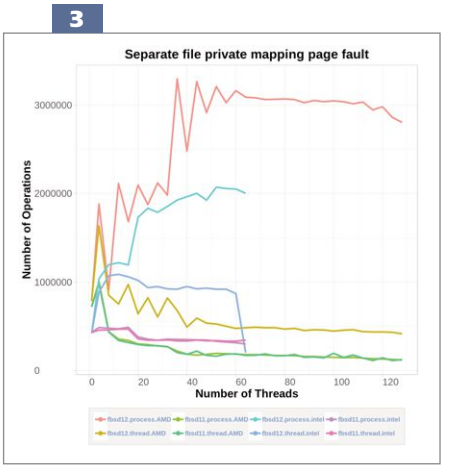
1



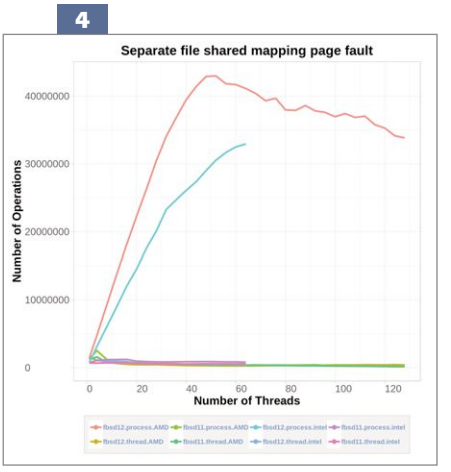
2



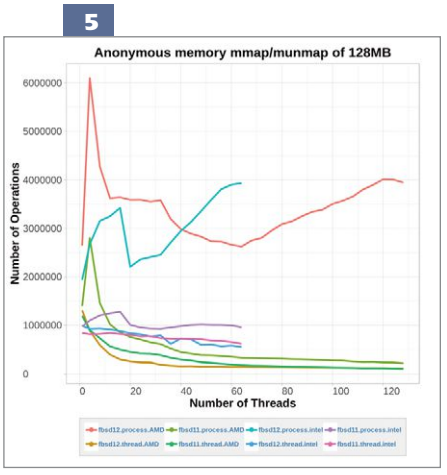
3



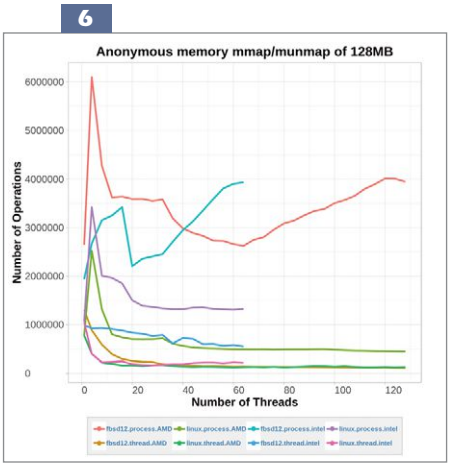
4



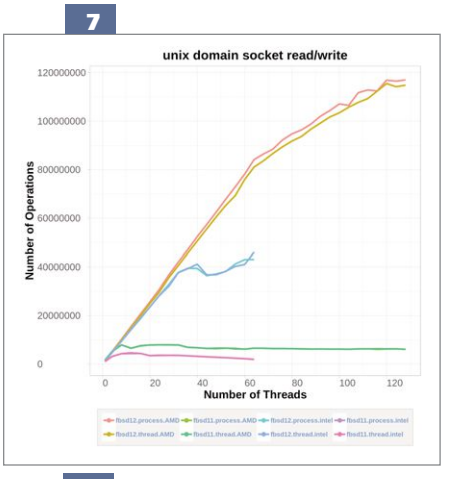
5



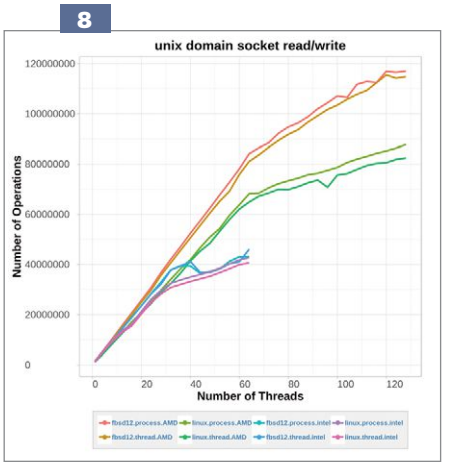
6



7



8



At least as of Linux 4.15 FreeBSD actually scales better than Linux on UNIX sockets **(8)**.

Separate file read still peaks at 32 hardware threads, but it's an 8x improvement **(9, 10)**.

Unfortunately, there are a few areas where underinvestment shows through quite clearly. Although there was an improvement in the brk benchmark from changing handling of swap reservations **(11)**, there are several other system-wide serialization points. Linux scales near linearly here, and at its peak is capable of performing 32x as many brk ops/s **(12)**.

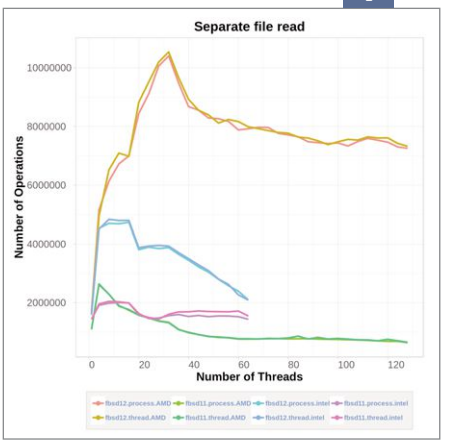
Arguably this isn't that big a deal in practice due to its relative lack of prominence in real world workloads. As a class, the most unsettling difference between FreeBSD and Linux is in filesystem operations. Linux scales near linearly in many cases where FreeBSD stops scaling at four hardware threads **(13, 14, 15, 16)**.

Given more time we would have provided benchmarks with more real world workloads such as the nginx web server serving small static objects, memcached, PostgreSQL, etc.

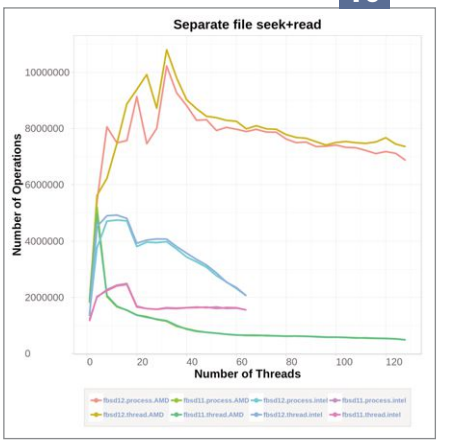
Alternative Approaches

Where do we go from here? Benchmarks can identify how well a system performs but are specific to one workload and configuration. Microbenchmarks are useful for identifying system bottlenecks. However, they don't provide a way to systematically guarantee the absence of scaling limitations. Is there a way to more consistently identify issues? Up until recently the answer was no. However, work in 2014 by Clements [Clements, 2014] built on the notion of disjoint-access-parallel memory systems [Israeli, 94] to rigorously identify limitations in the scalability of both software interfaces and their implementations. This work proposes

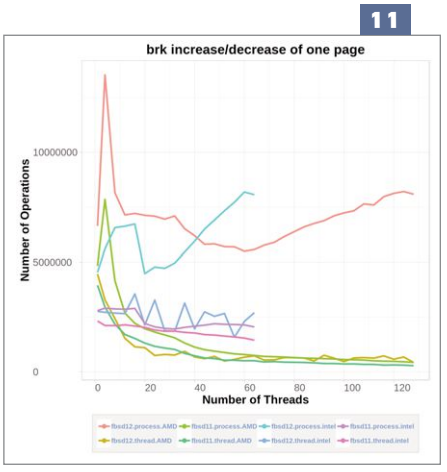
9



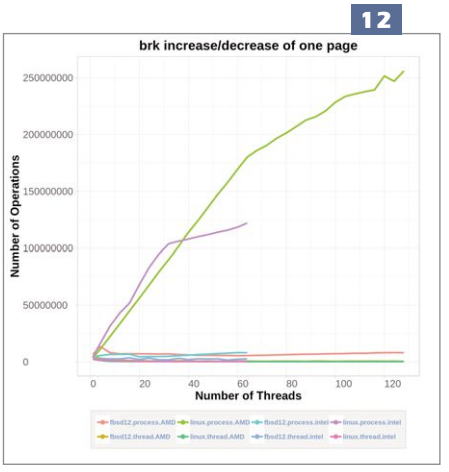
10



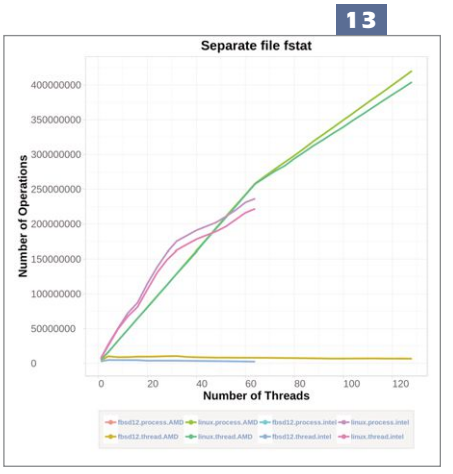
11



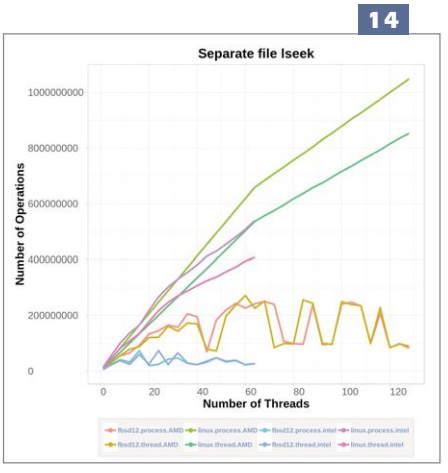
12



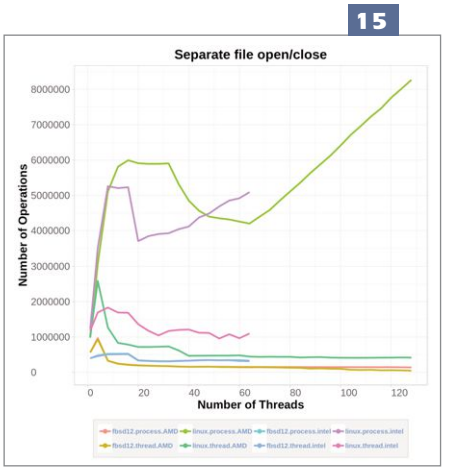
13



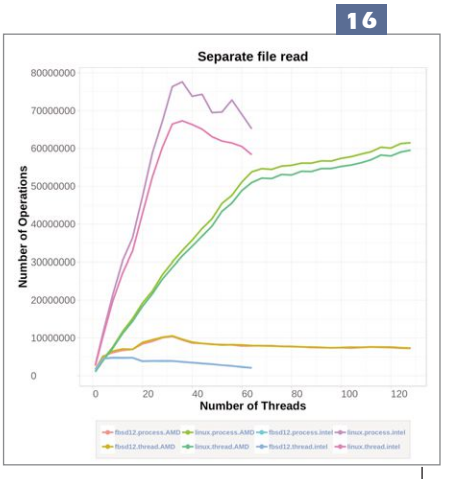
14

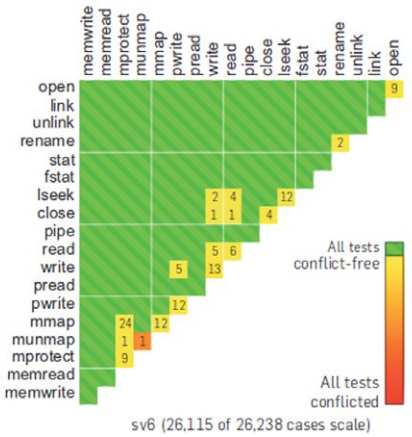


15

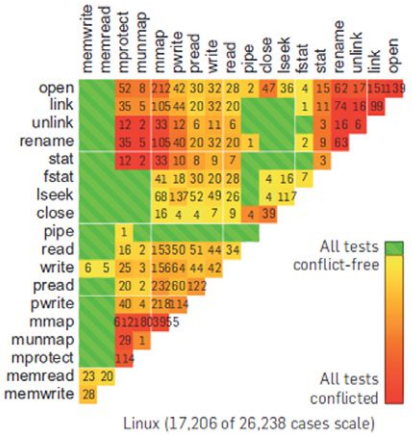


16





sv6 (26,115 of 26,238 cases scale)



Linux (17,206 of 26,238 cases scale)

the *scalable commutativity rule*, which says, in essence, that whenever interface operations commute, they can be implemented in a way that scales. The intuition behind this is simple: when operations commute, their results (both the values returned and any side effects) are independent of order.

He starts by observing that many scalability problems lie not in the implementation, but in the design of the software interface. An interface definition that does not permit two operations to commute enforces serialization between two calls. The POSIX definition of the *open* system call requires that it return the lowest available file descriptor. This means that two calls to open of different files need to be serialized on file descriptor allocation. Some other system calls that have unnecessarily unscalable interfaces are: fork (when immediately followed by exec), stat, sigpending, and munmap.

This is an interesting observation, but the real contribution of the work is developing a tool called *COMMUTER* which:

1. takes a symbolic model of an interface and computes precise conditions for when that interface's operations commute.
2. uses these conditions to generate concrete tests of sets of operations that commute according to the interface model, and thus should have a conflict-free implementation according to the commutativity rule.
3. checks whether a particular implementation is conflict-free for each test case.

He applied this to 18 POSIX system calls to generate 26,238 test cases and used these to compare Linux with sv6, a research OS developed by his group. He found that on Linux 3.8 17,206 cases scale vs 26,115 on sv6. The collection of test cases that failed to scale can be used as a starting point for redesigning subsystems just as the will-it-scale benchmarks have enabled us to identify a much narrower set of issues **(see diagrams left)**.

Porting COMMUTER to work with FreeBSD would be an interesting avenue for future work. ●

BIBLIOGRAPHY

[Attiya, 2011] Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P., Michael, M. M., Vechev, M. 2011. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: 487-498; http://doi.acm.org/10.1145/1926385.1926442

[Blanchard, 2013] Will-It-Scale benchmark suite. https://github.com/ScaleBSD/will-it-scale.

[Boyd, 2010] S. Boyd-Wickizer, A. T. Clements, . Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI), Vancouver, Canada, Oct. 2010.

[Corbet, 2013] Corbet, J.Per-CPU reference counts, July 2013. https://lwn.net/Articles/557478/

[Clements, 2013] Clements, A. T., M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In Proceedings of the ACM EuroSys Conference, Prague, Czech Republic, April 2013.

[Clements, 2014] Clements, A. T. The scalable commutativity rule: Designing scalable software for multicore processors, Ph.D. dissertation, Massachusetts Institute of Technology, Jun. 2014. [Online]. Available: https://pdos.csail.mit.edu/papers/aclements-phd.pdf

[Culler, 1999] Culler, D. Singh, J. P., Gupta, A. Parallel Computer Architecture - A Hardware / Software Approach, Morgan Kaufman, 1999

[Ellen, 2007] F. Ellen, Y. Lev, V. Luchango, and M. Moir. SNZI: Scalable nonzero indicators. In Proceedings of the 26th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Portland, OR, Aug. 2007.

[Fraser, 2004] Fraser, K. Practical lock-freedom, Ph.D. Thesis, University of Cambridge Computer Laboratory, 2004

[Hackenberg, 2009] D. Hackenberg, D. Molka, and W. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. MICRO 2009, pages 413–422.

[Hart, 2007] Hart, T. E., McKenney, P. E., Demke Brown, A., Walpole, J. 2007. Performance of memory reclamation for lockless synchronization. Journal of Parallel and Distributed Computing 67(12): 1270-1285; http://dx.doi.org/10.1016/j.jpdc.2007.04.010

[Herlihy, 2008] Herlihy, M., Shavit, N. 2008. The Art of Multiprocessor Programming. San Francisco: Morgan Kaufmann Publishers Inc.

[Israeli, 1994] Israeli, A., Rappoport, L. Disjoint-access-parallel implementations of strong shared memory primitives. In Proceedings of the 13th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (Los Angeles, CA, August 1994), 151–160.

[McKenney, 2011] McKenney, P. E. 2011. Is parallel programming hard, and, if so, what can you do about it? kernel.org; https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html

[Michael, 2004] Michael, M. M. Hazard pointers: safe memory reclamation for lock-free objects, IEEE Trans. Parallel Distrib. Syst. 15 (6) (2004) 491–504.

[Molka, 2015] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E Nagel. 2015. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In Parallel Processing (ICPP), 2015 44th International Conference on.IEEE, 739–748.

[Sorin, 2011] D. J. Sorin, M. D. Hill, and D. A. Wood. A Primer on Memory Consistency and Cache Coherence. Morgan and Claypool, 2011.

[Wang, 2016] Q. Wang, T. Stamler, and G. Parmer, "Parallel sections: Scaling system-level data-structures," in Proceedings of the ACM EuroSys Conference, 2016
Appendix A - FreeBSD Serialization Primitives
mutex

APPENDIX A - FreeBSD Serialization Primitives

mutex

Mutex is the most straightforward primitive; ownership is acquired by a thread doing a compare and swap operation on it with a pointer to the acquiring thread's structure. If the lock already contains a thread pointer, it is owned; if not it is free. There are two classes, `MTX_DEF` and `MTX_SPIN`. A `MTX_SPIN` mutex is considered "heavyweight" because it disables interrupts. It can be acquired in any context. While it is held (and with interrupts disabled in general) a thread can only acquire other `MTX_SPIN` locks and cannot allocate memory or sleep. While holding a `MTX_DEF` lock a thread can do anything that would not entail sleeping (acquire either types of mutex, rwlocks, non-sleepable memory allocations, etc.). While waiting to acquire a `MTX_SPIN` mutex a thread will "spin" polling the lock for release by its current holder. While waiting to acquire a `MTX_DEF` a thread will "adaptively spin" on it, polling for release if the current holder is running and being enqueued a `turnstile` if the current holder has been preempted. `Turnstile`s are facility for priority propagation allowing blocked threads to "lend" their scheduler priority to the current lock holder as a mechanism for avoiding priority inversion. In other words, if the blocked thread is higher priority the lockholder's scheduler priority will be elevated to that of the blocked thread.

rwlock

The rwlock extends the semantics of the `MTX_DEF` mutex by supporting two modes – single writer and multiple readers. Its implementation is similar to that of mutex with some additional state assigned to the lower bits of the lock field. In single writer mode it behaves the same as a mutex would. In reader mode, multiple readers can acquire the lock and writers are blocked until all readers drop the lock. In this mode we can no longer efficiently track the lockholders' state so we cannot propagate priority and it is not possible for an acquirer to know if all holders are running. Thus a thread can only spin speculatively.

It supports an arbitrary number of readers, so it's the first primitive developers have traditionally reached for when trying to guarantee existence of fields during a table lookup. However, every read acquisition and release involves an atomic update of the lock. When the lock is shared across core complexes (and thus updates entail cache coherency traffic between LLCs to transition the previous holder's cacheline from modified/exclusive to invalid) its use can quickly become very expensive.

sx

The sx lock is logically equivalent to the rwlock with the critical difference being that a lockholder can sleep. Blocked readers and writers are maintained on sleepqueues and priority propagation is not done.

rmlock

Like the rwlock and sx, the rmlock "read mostly lock" is a reader/writer lock. Its critical difference is that acquisition for read is extremely fast and does not involve any atomics. Acquisition for write is extremely expensive. In its current incarnation it involves a system wide IPI to all other cpus. This is actually a reasonable primitive for guaranteeing existence if updates are infrequent enough. It is easy to reason about, having the same semantics as familiar rwlocks.

lockmgr

Lockmgr is something of an anachronism. It has some unique features required by the VFS (virtual file system) layer. It is generally not a bottleneck in today's code and its idiosyncracies are outside the scope of what I hope to touch on.

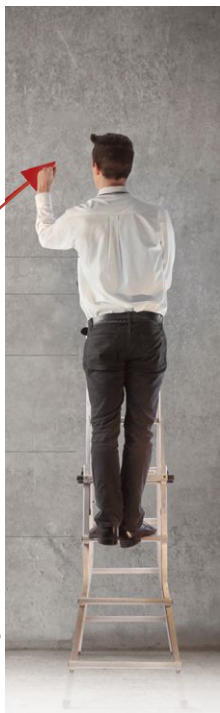
epoch

The epoch primitive allows the kernel to guarantee that structures protected by it will remain live while a thread is in an epoch section. Executing `do_stuff()` in an epoch section looks something like:

```
epoch_enter(global_epoch);
do_stuff(); ...
epoch_exit(global_epoch);
```

A thread deleting an object referenced within an epoch section can either synchronously wait for all threads in an epoch section during the current epoch plus a grace period by calling `epoch_wait(epoch)`, or it can enqueue the object to be freed at a later time using `epoch_call(epoch, context, callback)`, allowing a service thread to confirm – at lower cost than a synchronous operation – that a grace period has elapsed. In many respects the read side of epoch has similar characteristics to the read side of an `rmlock`. However, it does not provide a mutual exclusion guarantee. Modifications to an epoch protected data structure can proceed in parallel with readers. Modifications do typically need to be explicitly serialized with respect to each other. Thus a mutex is used to protect a writer against other writers. Although its implementation and the performance trade-offs are completely different from Linux's RCU, it largely supports the same programming design patterns. There are two variants of epoch, preemptible and non-preemptible. A non-preemptible epoch is lighter weight but does not permit the calling thread to acquire any lock type other than `MTX_SPIN` mutexes. Epoch is new in FreeBSD 12. It is essentially in-kernel scaffolding built around ConcurrencyKit's epoch (Epoch Based Reclamation) API.

Matt Macy is a Consulting Kernel Engineer and FreeBSD committer. Recently he has focused on networking performance, kernel scalability, and ZFS.



APPENDIX B Will-It-Scale Results

The complete results for Will-It-Scale for FreeBSD 11 vs FreeBSD 12 and FreeBSD 12 vs Ubuntu 18 (Linux 4.15) can be found at: https://github.com/ScaleBSD/scalebsd.github.io/tree/master/media/freebsd_processor_scaling