Design Challenges of Scalable Operating Systems for Many-Core Architectures

Andreas Schärtl
Friedrich-Alexander-Universität Erlangen-Nürnberg
andreas.schaertl@fau.de

ABSTRACT

Computers will move from the multi-core reality of today to many-core. Instead of only a few cores on a chip, we will have thousands of cores available for use. This new architecture will force engineers to rethink OS design. It is the only way for operating systems to remain scalable even as the number of cores increases. Presented here are three design challenges of operating systems for many-core architectures: (1) Locks which do not scale, (2) poor locality offered by the traditional approach of sharing processor cores between application and OS and (3) no more cache coherent shared memory available to the OS. This elaboration discusses why these challenges impact scalability, introduces proposed solutions and evaluates them.

1. INTRODUCTION

The most recent *International Technology Roadmap for Semiconductors 2.0* [1] describes the development of microcontroller. Manufacturers of microprocessors used to increase the frequency of their processors with each new technology generation. This was possible because of Moore's Law. At the beginning of the new century a physical cap was encountered in the form of thermal limits. It became clear that it would no longer be possible to increase both the number of transistors on a die while also increasing clock speed. Trying to do both would cause serious problems related to heat dissipation. Chip makers decided to keep Moore's Law in effect. They still produce chips with an ever increasing number of transistors, while frequencies do not see any more significant increase.

This is what changed our landscape from a world of single core to that of multi-core. As the number of transistors on integrated circuits continues to grow (Moore's Law remains in effect), it is reasonable to expect systems with hundreds, even thousands, of general purpose cores in the future [7, 20].

The number of transistors will continue to grow for the foreseeable future while frequencies will stay about the same. It is up to the designers of software to ensure that this new hardware is used efficiently, because simply waiting for higher frequencies that speed up performance is not possible anymore [18].

Looking at operating systems, it is essential that they scale with this new hardware. They have to to manage the resources of the upcoming many-core systems efficiently, otherwise it will be impossible for an application running on such a base to take full advantage of the newly available parallel processing power.

This elaboration is about scalability. As such, it is helpful to consider a definition of scalability. In his 2000 paper *Characteristics of Scalability and Their Impact on Performance* [6] Bondi gives a definition for scalability. He focuses on two types of scalability: *Load scalability* and *structural scalability*. Bondi defines load scalability as the ability of a system to continue effective operation while the workload increases. Systems offering poor load scalability show degraded relative performance when load increases. Structural scalability is a statement about future developments. A system with good structural scalability will be able to grow with the needs of tomorrow. On the other hand, a system with poor structural scalability will require lots of effort to keep up to date with current developments.

Wentzlaff et al. identified three challenges for system software on many-cores hardware: (1) Locks on OS data structures that impede scalability, (2) poor locality that leads to ineffective use of caches and (3) reliance on cache coherent shared memory, something future multi-core architectures will probably not be able to offer. Faced with these challenges they designed the fos operating system, an OS designed to scale on many-core hardware [20].

This elaboration focuses on those three problems identified by Wentzlaff et al. Section 2 discusses locks, caches and locality are topic of section 3 and finally section 4 is about reliance on cache coherent shared memory.

2. SCALABILITY ISSUES OF LOCKS

This section introduces the first challenge for operating systems on many-cores: Locks impede OS scalability. As the number of cores contending for a lock increases, more and more time is wasted waiting for locks. Because traditional approaches to kernel development will not offer enough scalability, a more long-term solution is introduced: Avoiding locks as a core design goal.

One job of any OS is to distribute hardware resources to the application processes and applications may need exclusive access to one or more of these resources at a given time [19, p. 6–7]. Operating systems today use locks to synchronize these different parties with each other to avoid race conditions and similar problems [20]. critical section

2.1 Locks may not Scale

Work on scalable operating systems has shown that there is reason to believe that locks will not offer the desired scalability [8, 20].

Wentzlaff et al. motivate fos with a case study. In it, they test the performance of the physical page allocator in Linux 2.6.24.7. For this case study, they used a machine with 16 Intel cores and a total of 16 GB RAM. Each core allocated one gigabyte each. Then, every core touched the first byte on every page of this gigabyte, ensuring that the kernel actually maps the pages into physical memory. In each run, they modified the number of active cores participating in the benchmark. Differences in performance depending on core count were made visible. In summary, the researches made two conclusions: First, (1) as they increased the number of cores, lock

contention became the biggest cost factor. It was the synchronization overhead that consumed the most resources. (2) When more than eight cores are active in the benchmark, the total execution time actually increased. The authors argued that synchronization overhead caused by locks is the reason for this decline in performance [20].

At first this benchmark may seem unrealistic. All active cores are busy requesting memory at high speed. But the authors are aware of this and make a counterargument. Having all cores on a system allocate considerable amounts of memory at the same time is not very realistic. However it is fair to expect 16 cores out of an available 1,000 to request memory at the same time. Lock contention will be a big factor on many-core systems if software design remains as it is today.

Such observations were also made in the development of the Corey operating system [8]. The authors ran a similar benchmark on a machine with 16 cores. In it, the time taken for acquiring and then releasing a kernel lock was timed. As the number of active cores increased, the time for a single lock acquire/release increased in a linear fashion.

From these two examples we see that using locks offers poor load scalability. Increasing the number of active cores increases the contention for locks. This results in a noticeable overhead, that only grow as core counts increase. OS design indeed needs to consider the challenge of lock overhead.

2.2 Short Term Remedies

Locks impede scalability. Operating systems used to increase the granularity of locks to combat this. They long ago stopped using a single global lock for their global data structures. Instead today more fine grained locks are used. This allows for higher levels of parallelism [20].

So one may suggest that using ever more fine grained locks is enough. But this is not a long term solution: It will not be able to continue if the number of cores grows as expected. This is because splitting up already parallelized code very work intensive and also prone to errors [20]. This shows that simply increasing the granularity of locks will not offer the desired structural scalability.

Another idea could be to increase the performance of locks themselves. In *Non-scalable locks are dangerous* [9] the authors replaced simple spin locks in the Linux kernel with more modern MCS locks [14]. They observe that (1) the required changes are straight-forward and (2) MCS locks offer the desired load scalability in their benchmarks. These benchmarks use up to 28 cores.

Using more sophisticated locks such as the MCS lock can be a short term remedy, but not a long-term solution. Better performing locks are not an improvement in structural scalability. Rather they are an aid that can postpone more fundamental changes in architecture for later [9].

These two traditional approaches can offer some improvements. But it remains unclear how to design an OS with locks that offers both good structural and load scalability. It is thus reasonable to list locks as a problem.

2.3 Avoiding Locks All Together

Using locks impedes scalability. To avoid this problem, the kernel could avoid locking as much as possible. This was proposed for fos [20] and the Barrelfish operating system [4].

On fos, an OS designed for thousands of cores, every thread runs on its own core. The designers assume that cores will be so plentiful that dedicating a core to just one thread is reasonable without running into limitations posed by core count. Should that happen regardless, the OS falls back to traditional time sharing. Servers realize typical OS functionality and run on dedicated system cores. Inspired by online services they are organized in fleets of cores offering the same functionality. One a system with thousands of cores there will be multiple server cores offering a single system service, such as memory allocation or network communication [21].

Now for how this results in less locks. Servers do not work in an preemptive manner. Rather, server cores process requests from application cores in a sequential manner. Only the server thread is running on a server core so there is no need for synchronization within the core itself. With this approach, no locks are needed within a core. This avoids the scaling pitfalls of using locks [20].

Global data structures in the kernel remain. The available physical memory is finite, there needs to be some kind of synchronization between servers in a fleet that does not rely on hardware locks. Wentzlaff et al. propose two possible solutions: (1) Use a dedicated lock server that offers *notional locks*, which can be used to synchronize the servers with each other. But because notional locks are not expected to perform well, algorithms found on distributed systems should be used instead. With high rates of replication it will hopefully be able to service applications in a reasonable amount of time. As an alternative, (2) using a dedicated core as transaction server is proposed. The individual servers in the fleet would process the requests and then only send the result to the transaction server [20].

When fos was first introduced, its implementation was still in an early state. As such, it was not possible to evaluate the performance of system services implemented as fleets. Evaluation was possible only in 2011, when some basic servers were implemented. It included a basic network stack, a page allocation service and a read-only file system. Comparing the performance of these services to a standard Linux kernel showed comparable performance and better scalability. For low core counts, fos suffered some overhead losses compared to Linux (especially the file system implementation). But as the number of cores increased, fos showed better load scalability than Linux [21]. In summary, even though some new overhead is introduced, this approach does seem promising.

Corey [8] takes a less drastic approach to reducing the number of required locks. Here, applications have to explicitly specify which resources are shared. On today's systems it is typical for a program to consist of multiple threads of execution. All of these threads share an address space and can mutate state in that address space. As such, kernel data structures, such as the page table of a process, need to be protected with locks. If only one thread accesses these structures, using a global lock is not actually required. In Corey, this does not happen: Per default, resources are not shared. Only after issuing a system call indicating the desire to share, e.g. a page of memory, will it be possible to do so. Only then will these resources be protected by locks. As the system knows exactly which resources are shared and which are not, these locks can be very fine grained as supposed to a coarse global lock. This approach reduces the number of required locks in the kernel.

One of the goals of Corey was to make sharing explicit, so that no resources are wasted by assuming data to be shared that isn't. It is interesting that fos achieves this goal as well. Because fos only relies on messaging, all sharing is explicit by design: Sharing requires sending a message and all data that is shared has to be part of a message [21].

In this section, locks were introduced a threat to scalability. Traditional approaches to OS development can only offer short term solutions. In the long run, major architecture changes are required: These changes try to avoid locking as much as possible to reduce the damage caused by lock contention.

3. CACHES AND LOCALITY

This next section elaborates the performance impact of having operating system and application share the same processing core. Context switches are expensive and disrupt caches. As a solution, this section will introduce dedicated OS cores, which mean a high level of locality and great use of caches.

On today's multi-core machines the OS and applications typically share the same cores with each other. Processor cores, however, only have one set of caches, registers and only one translation lookaside buffer (TLB). Exploiting both caches and TLB is crucial to good performance. But with every context switch from application to OS and vice versa, caches and TLBs lose effectiveness [16].

3.1 Damage Caused by Context Switches

Wentzlaff et al. conducted another case study. Using a modified version of the x86_64 emulator QEMU, it was possible to measure cache miss rates and attribute them to either misses of (1) operating system code, (2) application code or (3) operating system/application interference. On this emulator they tested Debian 4 running the Apache2 web server, which received requests for a static web page. In this test they noticed that the OS suffers from cache misses, much more than the application does. Cache misses related to operating system/application interference were negligible. These OS cache misses are because of cache interference. The context switch between OS and application causes this [20].

Serving static web content is something many web servers will do using the very same software used in this experiment. It is likely that these cache misses occur a lot on real-life multi-core systems today.

Wentzlaff et al. finish their case study with a note that these findings reproduce the results from a similar experiment made in 1988 [3] by Agarwal et al. It should be noted that Agarwal also contributed to the paper introducing fos. In the 1988 paper, OS misses also made up a noticeable portion while OS/application interference was negligible. However the difference between OS and application code was not as pronounced, rather they were about equal in numbers.

It is not immediately clear how these cache misses are related to scalability. Imagine a many-core system *A* that uses the traditional approach of context switching to another such system *B* that does away with context switching in whatever way. Assume that system *B* does not introduce any new limits to scalability in its implementation. As the number of cores increases, I do not see how system *A* or *B* will suddenly decrease in relative performance as the number of cores grows. It seems that there is no threat to structural scalability posed by context switching. About load scalability, it is likely that an increase in load of system calls will pose an ever more damage to performance as the processor is occupied with context switches. Unfortunately, no such considerations were made by the authors of this case study.

3.2 Keeping Operating System and Application Separated

Poor locality does not pose a challenge to structural scalability. New many-core computer architectures, however, make it possible to envision systems that require little context switching between OS and application, as proposed in fos [20] and Barrelfish [4]. This increases load scalability, because caches can be used to their fullest potential. As free cores become a commodity, a new approach is to keep OS and application code separated on different cores.

The fos splits OS and application threads onto different cores. Server processes dedicated to a core offer OS services, in fact every thread runs on a dedicated core. While a scheduler before had to manage the resource time, now the scheduler is occupied with managing space. It has to distribute threads onto the physical cores. Only when the number of threads exceeds the number of cores will time sharing be needed [20].

No evaluation was made until 2011. Then, the performance of single-core sharing was compared to that of multi-core communication. These tests used Linux, not fos, as the kernel. Among others, they tested the performance of a web server, directory traversal and compiling a C library project. The results were that for most use cases, separating OS and application on different cores does improve performance. Especially when OS and application run on the same chip, so they can share L3 caches [5]. If these findings also apply to more distributed systems like fos is not clear.

Dedicating cores to system functionality is something other operating systems with scalability in mind have also done. Corey allows applications to dedicate cores to kernel tasks, such as communicating with a network interface. Compared to a stock Linux kernel, Corey was able to display improved network performance in a synthetic benchmark [8].

The Barrelfish OS also uses dedicated cores, in a fashion like fos. Baumann et al. argue that dedicated cores are a natural fit for many-core architectures. Also, it allows exploiting the available messaging networks. Especially because Baumann et al. believe that message passing is getting cheaper. In addition, message passing also offers better support for heterogeneous chip layouts, something Barrelfish has to handle [4].

In conclusion, we saw that sharing the resources of a core imposes high costs in the form of cache misses. As the number of cores rises, enough are available so that a subset of them can be dedicated to OS services alone. Such a system is able to take full advantage of caches, resulting in more efficient use of the available hardware.

4. RELIANCE ON GLOBAL CACHE COHERENT SHARED MEMORY

Now for the final challenge. Some researches believe that many-core architectures will not offer cache coherent shared memory [20, 4, 11]. Instead, threads and processes should use messaging for communication. Because cache coherent shared memory is a useful tool that simplifies parallelizing certain kinds of applications, operating systems should still offer it to applications, assuming the hardware supports it.

Many contemporary computer systems offer cache coherent shared memory. Operating systems running on such hardware can assume that (1) there exists a single global address space and that (2) the caches of individual cores can be kept in sync using cache coherence protocols. These cache coherence mechanism are employed by hardware and remain transparent to software [17, p. 1–5].

4.1 Cache Coherent Shared Memory may not be Available on Many-Core Systems

The conveniences of cache coherent shared memory may not continue when we move to many-core systems. This is a trend observable in current embedded platforms. There, a global cache coherent shared memory address space is not available. Instead, cores are able to communicate using message queues [15, 20].

Baumann et al. also expect global cache coherent shared memory to be a thing of the past. They argue that while it has been a useful feature before, now it is essential that future OS designs are able to perform without cache coherent shared memory, exactly because it is possible that these operating systems will have to run on hardware without cache coherent shared memory [4].

But why is it that as the number of cores increases, it becomes

ever more expensive to implement cache coherence on hardware? Choi see three problems with scaling cache coherence to the core counts of many-cores. (1) Overhead both related to power consumption and latency, (2) a very complex implementation that is prone to errors and (3) extra space overhead as lots of state needs to be maintained [11]. These three points illustrate that it is hard to scale cache coherence up to many cores. Cache coherence therefore offers poor structural scalability.

It is not undisputed that cache coherence will disappear in the future. Some argue that hardware-provided cache coherence is way too useful to be abandoned and propose new mechanisms that are supposed to keep cache coherence alive even as core count increases [13].

4.2 Messaging Instead of Cache Coherent Shared Memory

Some assume that many-core architectures will not offer cache coherent shared memory. However Borkar explains that they do offer on-die networks. These networks connect the different cores with each other. Organized in ring or mesh topologies, they act in a packet-switching manner [7]. Because cache coherent shared memory may not be available but message networks will be available, operating systems designed for many-core systems focus on messaging instead of shared memory for communication.

The fos does not require cache coherent shared memory in kernel space. Interaction between OS and application also do not require cache coherent shared memory. Instead, all such communication is done with message passing over the on-die network. Message passing does introduce new latency, but Wentzlaff et al. are hopeful that these costs are worth the investment, because using only inter-core communication avoids the need for context switches. Remember that on fos all threads run on their own core with a subset of all cores serving OS functionality. If an OS used traps to trigger system calls before, now it can use inter-core communication. With it goes the need for an expensive context switch (around 100 to 300 cycles on modern processors). The alternative, inter-core messaging, is cheaper. Sending a message is expected to cost in the range of 15 to 45 cycles, judging from modern embedded architectures [20]. How these processor cycles translate to real-life performance is not topic in the original paper.

Baumann et al. make similar assumptions, and are able to back up their claims by comparing performance of cache coherent shared memory access with message passing. For these benchmarks, they used an AMD machine with 4 CPUs with 4 cores each, totaling 16 cores. First, they had threads dedicated to a single core update the contents of a small portion of memory. The cache coherence mechanism on hardware reacted and published these changes to the other cores. They noticed that as the number of cores grew, performance worsened, exhibiting poor load scalability. Comparing the performance between one core updating the value and 16 cores, performance dropped by a factor of 40. Second, they tested inter-core communication, which performed better. Just sending a message showed no degradation caused by the number of cores at all. Where delays did occur is when considering not just sending a message, but also processing it. Here, a linear increase of time is observed, something the authors attribute to queuing delays [4].

The idea behind message passing is not new. The Mach [2] microkernel used messaging for communication between applications and OS in 1986. Development on it lead to the believe that communication through shared memory and communication through explicit message are dual to each other [22]. The L4 [12] microkernel did show that it is possible to implement fast messaging based on only shared memory. However it is unlikely that a system optimized to do one thing will excel doing the other thing.

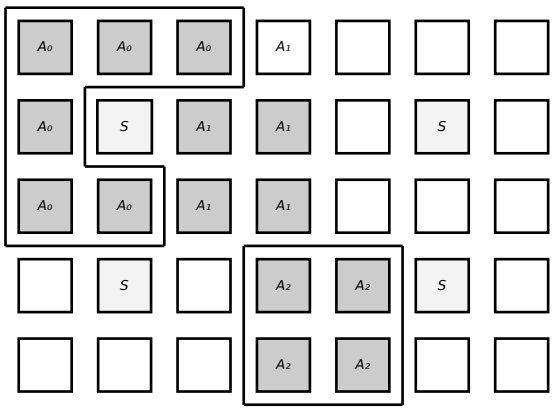

Figure 1: An example of application-level cache coherent shared memory. Each square represents a processor core. The cores denoted S are running OS services, while the cores denoted A_i run an application thread with process identifier i. Applications A_0 and A_2 use cache coherent shared memory within the application.

In summary, message passing is a viable alternative to cache coherent shared memory. Application processes can use messaging instead of other mechanisms such as traps to communicate with the OS. Designing an OS like this increases structural scalability, because no re-engineering is required as core count increases.

4.3 Islands of Cache Coherent Shared Memory

OS designers are willing to sacrifice global cache coherent shared address space for application-kernel communication and data structures internal to the kernel. But there seems to be consensus that applications should have cache coherent memory available to them, as long as this is supported by the hardware [21].

As such, fos allows application-level cache coherent shared memory if the hardware supports it [20]. Figure 1 illustrates a possible core layout on an OS in the vein of fos. Here, two applications (denoted A_0 and A_2) take advantage of user-level cache coherent shared memory. The OS server cores denoted S do not use cache coherent shared memory, they only communicate through explicit message passing. Application A_1 does not need cache coherent shared memory. Depending on the specific needs of an application, this feature can enabled or omitted.

This approach to user-level cache coherent shared memory employed by fos is reminiscent to the Hive [10] operating system from 1995. In Hive, cores are split up into individual cells. Within each of those cells, the cores have cache coherent shared memory available to them. To communicate with other cells, network packets are sent. The motivation for Hive, however, was to build a reliable OS. Faults in hardware or software stay within a cell and do not affect the whole system.

In summary, we saw that global cache coherent shared memory is likely to disappear. Instead, messaging is an alternative that cores can use for communication. If offered by the hardware platform, cache coherent shared memory should still be available to applications, albeit on a smaller scope.

5. CONCLUSION

The continuing trend for many-core systems will force OS designers to face a number of new challenges and take advantage of changes in architectures, if they want to build systems that remain scalable even as load and core count increases. First, locks should be avoided because lock contention is a threat to scalability. While there are some short term solutions (e.g. modern locks that perform better), a more long-term approach is to design the OS in a way that avoids locks as much as possible. A proposed way to do this is to split up OS services onto dedicated cores that process request for OS functionality (system calls) in a sequential manner. Second, as cores become a commodity, it is possible to split up application and operating system, so that they do not need to share cores with each other anymore. To do this, a core is dedicated to every thread on the system. Because only one thread runs on one a core, no context switching occurs. Caches and TLB can be used to their fullest potential. Finally, cache coherent shared memory may not be available on many-core systems. Scaling cache coherence up to many-core systems has many difficulties and embedded many-core hardware already ships without it. Instead of cache coherent shared memory, messaging can be used for OS/application communication, the dual to shared memory. While the kernel may have to forgo cache coherent shared memory, applications can still benefit on it, as long as the hardware offers support for it.

6. REFERENCES

[1] The international technology roadmap for semiconductors 2.0 Executive Report. 2015.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. 1986.

[3] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst.*, 6(4):393–431, Nov. 1988.

[4] R. I. Andrew Baumann, Paul Barham and T. Harris. The multikernel: A new OS architecture for scalable multicore systems. In *22nd Symposium on Operating Systems Principles*. Association for Computing Machinery, Inc., October 2009.

[5] A. Belay, D. Wentzlaff, and A. Agarwal. Vote the OS off your core. 2011.

[6] A. B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 195–203, New York, NY, USA, 2000. ACM.

[7] S. Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[8] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai, et al. Corey: An operating system for many cores. In *OSDI*, volume 8, pages 43–57, 2008.

[9] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[10] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 12–25, New York, NY, USA, 1995. ACM.

[11] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 155–166. IEEE, 2011.

[12] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[13] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012.

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

[15] J. Shalf, J. Bashor, D. Patterson, K. Asanovic, K. Yelick, K. Keutzer, and T. Mattson. The MANYCORE revolution: will HPC lead or follow. *SciDAC Review*, 14:40–49, 2009.

[16] L. Soares and M. Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 33–46. USENIX Association, 2010.

[17] D. J. Sorin, M. D. Hill, and D. A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[18] H. Sutter. The free lunch is over. *Dr. Dobb's Journal*, 30(3), Feb. 2005.

[19] A. S. Tanenbaum and H. Bos. *Modern operating systems*. Pearson Prentice Hall, 3rd international edition, 2009.

[20] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, Apr. 2009.

[21] D. Wentzlaff, C. Gruenwald III, N. Beckmann, A. Belay, H. Kasture, K. Modzelewski, L. Youseff, J. E. Miller, and A. Agarwal. Fleets: Scalable services in a factored operating system. 2011.

[22] M. Young, A. Tevanian, R. Rashid, D. Golub, and J. Eppinger. The duality of memory and communication in the implementation of a multiprocessor operating system. *SIGOPS Oper. Syst. Rev.*, 21(5):63–76, Nov. 1987.