# 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, thousands of cores will be available for use. This change in architecture will force engineers to rethink OS design, so that operating systems remain scalable even as the number of cores increases. Presented in this elaboration are three design challenges for operating systems on 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 2015 *International Technology Roadmap for Semiconductors 2.0* [1] gives an overview of the history of microprocessor development. Manufacturers of microprocessors used to increase the frequency of their processors with each new technology generation, but at the beginning of the current century thermals limits were encountered which stalled the increase in processor frequencies. It is no longer possible to increase both the number of transistors on a die while also increasing clock speed, because trying to do both would lead to serious problems related to heat dissipation. Chip makers decided to keep Moore's Law in effect and therefore still produce chips with an ever increasing number of transistors while frequencies do not see any more significant increase. The newly gained transistors are used for more cores on a single die.

The move away from higher clock rates to more cores is what changed the processor landscape from single-core to multi-core. As the number of transistors on integrated circuits continues to grow, it is reasonable to expect systems with hundreds, even thousands, of general purpose cores in the future [7, 20].

If the number of transistors continues to grow as expected while frequencies stay about the same, it will be up to the designers of software to ensure that this new hardware is used efficiently. Simply waiting for higher clock rates that speed up performance is not feasible 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 on top of such an OS 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. Bondi provides such a definition, focusing on two types of scalability: *Load scalability* and *structural scalability* [6]: Load scalability is 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 hamper scalability, (2) poor locality that leads to ineffective use of caches and (3) reliance on global cache coherent shared memory, something future multi-core architectures may not 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, avoiding locks should be a core goal in the design of future operating systems.

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]. Many operating systems use locks to synchronize these different parties with each other to avoid synchronization problems [20].

### 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 which they tested the performance of the 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 and then 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 researchers noticed that as they increased the number of active cores, lock contention became the greatest cost factor. It was synchronization overhead that consumed the most resources [20].

At first this benchmark may seem unrealistic, as all active cores

only request memory. Wentzlaff et al. are aware of this and argue that while it is unrealistic for all cores on a machine to request memory at the same time, it is fair to expect 16 cores out of an available 1,000 to do so. Lock contention should be a big factor on many-core systems if software design remains as it is today.

Comparable observations were also made in the development of the Corey [8] operating system, where Boyd-Wickizer et al. ran a similar benchmark on a machine with 16 cores. They measured the time it takes to acquire a spin lock and then release it again in the Linux 2.6.25 kernel. As the number of active cores increased, the time for a single lock acquire/release increased in a linear fashion. More active cores meant more lock contention.

These two examples show that using locks offers poor load scalability. Increasing the number of active cores increases the contention for locks. This results in a noticeable overhead, which only grows 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 designers used to increase the granularity of locks to combat this. They long ago stopped using a single global lock for global data structures, instead today more fine grained locks are used. This allows for higher levels of parallelism [20].

Increasing the granularity of locks as the number of cores rises is not a long term solution. This is because splitting up already parallelized code very work intensive and also prone to errors. Simply increasing the granularity of locks will not offer the desired structural scalability [20].

Another idea is increasing the performance of locks themselves. In *Non-scalable locks are dangerous* [9], Boyd-Wickizer et al. 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.

But more sophisticated locks such as the MCS lock can only be a short term remedy, 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, as they do not solve the problem of lock contention itself [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.

## 2.3 Avoiding Locks All Together

To avoid the problem of locks contention, the OS should avoid locking as much as possible. This is the approach of fos [20] and the Barrelfish operating system [4].

On fos, an OS designed for thousands of cores, every thread runs on its own core. Wentzlaff et al. 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 can fall back to traditional time sharing [20].

OS functionality, such as memory management or network communication, is split into different *servers*. Each of those servers offers a certain OS service and runs on a dedicated core. When an application wants to use OS functionality (e.g. allocate memory), it sends a message to the corresponding server to request the service [20]. The way servers offer OS functionality is similar to how microkernels implement OS services through independent processes [2, 12].

Inspired by distributed online services, servers are organized in

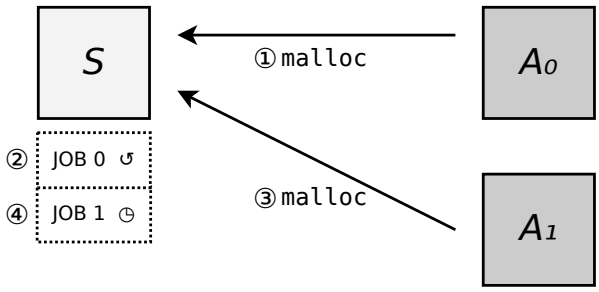


**Figure 1: Server $S$ sequentializes concurrent requests for memory from applications $A_0$ and $A_1$. ① First, application $A_0$ sends a request to server $S$. ② Because $S$ is idle, it can start processing the job right away. ③ While $S$ is still processing the first job, application $A_1$ also sends a request for memory. ④ $S$ is still busy with the first job, so the second job only gets enqueued, which can be implemented using optimistic synchronization. The second job will be processed once the first job is done.**

*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 way, rather they process requests from applications in a sequential manner. This is illustrated in Figure 1, where two applications $A_0$ and $A_1$ send a request for memory to a server core $S$. Because servers sequentialize the concurrent requests from applications and only one server thread runs on a server core, there is no need for hard synchronization on the core itself. With this approach, no locks are needed within a core, thus avoiding the scaling pitfalls of 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 some possible solutions: (1) Use a dedicated lock server that offers *notional locks*, which can be used to synchronize servers with each other. To acquire a lock, a server sends a message to the lock server to request that lock, which is identified by a unique name. The same applies for releasing that lock. Because notional locks are not expected to perform well, (2) algorithms found on distributed systems should be used instead. With high rates of replication it will hopefully be possible to service applications in a reasonable amount of time. As an alternative, (3) using a dedicated 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, including a 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 different approach to reducing the number of required locks. Here, applications have to explicitly specify which resources are shared. On today's general purpose computers, it is typical for a process 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 synchronized. Often locks are used to enforce these critical sections. But 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 message passing, all sharing is explicit by design: Sharing requires sending a message and all data that is shared has to be embedded in a message [21].

In this section, locks were introduced as 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. USE OF CACHES AND LOCALITY

This next section is on the performance impact caused by operating system and application sharing of the same processors. Context switches between OS and application are expensive and disrupt caches. As a solution, this section will introduce dedicated OS cores, which results in better locality and thus effective use of caches.

On today's multi-core machines, 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 a case study to illustrate the damage caused by OS/application sharing of cores. 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 equipped with just one processor core, they ran Debian 4 and the Apache2 web server, which received requests for a static web page.

The main take-away from this experiment was that the OS especially suffers from cache misses, more than the application does. This applied to different types of caches and different cache sizes. Cache misses related to operating system/application interference were negligible [20].

Serving static web content is something many web servers do with 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 the produced findings match 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 OS and application misses were about equal in percentage.

It is not clear how these cache misses are related to scalability. If system calls are implemented with traps, 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 [16]. If this is an actual threat to load scalability is not known as no such considerations were made for this case study [20].

## 3.2 Keeping Operating System and Application Separated

Whether poor locality has impact on scalability or not, new many-core computer architectures make it possible to envision systems that require little context switching between OS and application, as proposed in fos [20] and Barrelfish [4]. As free cores become a commodity, a new approach is to keep OS and application code separated.

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 has to manage space. Only when the number of threads exceeds the number of cores will time sharing be needed [20].

No evaluation of separating OS and application was made by Wentzlaff et al. 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: Unlike Linux, they do not rely on cache coherent shared memory as a communication medium. Cache coherent shared memory is something future many-core systems may not offer (see section 4).

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, especially because this allows the OS to take advantage of message passing networks available on many-core hardware [4].

In conclusion, sharing a core between OS and application is expensive. 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, which means 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, processes should use message passing 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 mechanisms are employed by hardware, so their implementation remains transparent to software [17, p. 1–5].

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

Cache coherent shared memory may not be available on 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. believe that global cache coherent shared memory will not be available in the future. 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 et al. 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, which implies that cache coherence 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 Message Passing 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, operating systems designed for many-core systems focus on message passing instead of shared memory for communication.

The fos does not require cache coherent shared memory for communication between OS and application or within the kernel itself. 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 performance will be comparable to cache coherent shared memory [20].

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 and a total of 16 cores. First, they had threads dedicated to a single core update the contents of a small portion of memory. The cache coherence mechanism ensured that changes were visible to all cores. They noticed that as the number of cores grew, performance worsened, exhibiting poor load

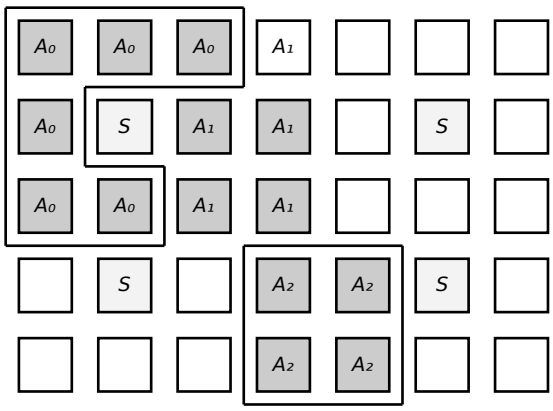


**Figure 2: An example of application-level cache coherent shared memory. Each square represents a processor core. The cores denoted $S$ are running OS services, while the cores denoted $A_i$ run an application thread with process identifier $i$. Applications $A_0$ and $A_2$ use cache coherent shared memory within the application.**

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 Bauman et al. attribute to queuing delays [4].

The idea behind message passing is not new. The Mach [2] microkernel used message passing 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 passing are dual to each other [22]. The L4 [12] microkernel did show that it is possible to implement fast message passing based on only shared memory. However it is unlikely that a system optimized to do one thing will excel doing the other thing.

In summary, message passing is a viable alternative to cache coherent shared memory. Application processes can use message passing 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 [20].

## 4.3 Islands of Cache Coherent Shared Memory

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

As such, fos allows application-level cache coherent shared memory if the hardware supports it [20]. Figure 2 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 application-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 either. Depending on the specific needs of

an application, this feature can enabled or ignored.

This approach to user-level cache coherent shared memory employed by fos is reminiscent of 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, global cache coherent shared memory is likely to disappear. Instead, message passing is an alternative that can be used 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 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 system software that remains 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 in a sequential manner. Second, as cores become a commodity, it becomes 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 and caches can be used to their fullest potential. Finally, cache coherent shared memory may not be available on many-core systems. Scaling hardware-provided cache coherence up to many-core systems is difficult and embedded many-core hardware already ships without it. Instead of cache coherent shared memory, message passing 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 from 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.