Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2009-059 November 20, 2009

A Unified Operating System for Clouds and Manycore: fos

David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski,, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

A Unified Operating System for Clouds and Manycore: fos

David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski,
Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal
CSAIL, Massachusetts Institute of Technology

Abstract. Single chip processors with thousands of cores will be available in the next ten years and clouds of multicore processors afford the operating system designer thousands of cores today. Constructing operating systems for manycore and cloud systems face similar challenges. This work identifies these shared challenges and introduces our solution: *a factored operating system (fos) designed to meet the scalability, faultiness, variability of demand, and programming challenges of OS's for single-chip thousand-core manycore systems as well as current day cloud computers.* Current monolithic operating systems are not well suited for manycores and clouds as they have taken an evolutionary approach to scaling such as adding fine grain locks and redesigning subsystems, however these approaches do not increase scalability quickly enough. fos addresses the OS scalability challenge by using a message passing design and is composed out of a collection of Internet inspired servers. Each operating system service is factored into a set of communicating servers which in aggregate implement a system service. These servers are designed much in the way that distributed Internet services are designed, but provide traditional kernel services instead of Internet services. Also, fos embraces the elasticity of cloud and manycore platforms by adapting resource utilization to match demand. fos facilitates writing applications across the cloud by providing a single system image across both future 1000+ core manycores and current day Infrastructure as a Service cloud computers. In contrast, current cloud environments do not provide a single system image and introduce complexity for the user by requiring different programming models for intra- vs inter-machine communication, and by requiring the use of non-OS standard management tools.

1 Introduction

The next decade will bring single microprocessors containing 100's, 1000's, or even tens of 1000's of computing cores. Computational clusters and clouds built out of these multicore microprocessors will offer unprecedented quantities of computational resources. Scaling operating systems(OS), internal OS data structures, and managing these resources will be a tremendous challenge. Contemporary OS's have been designed to run on a small number of reliable cores and are not equipped to scale up to 1000's of cores or to tolerate frequent errors.

While manycore processors and cloud computing provide great opportunities, each also presents many challenges to OS's. Manycore processors are forcing OS's to embrace parallelism in order to not become a bottleneck for the system. With increasing transistor counts and shrinking transistor size, the probability of faults in manycore systems is increasing. The increased likelihood of faults requires OS's to gracefully handle such faults and continue executing. On cloud systems, some challenges for the system software include matching resources to the requested load and the sheer scale of the

number of resources being managed. Developing applications for current cloud systems requires the developer to address more system level concerns than are required for programming traditional multiprocessor computers.

Upon close inspection, clouds and future manycore processors impose the same challenges upon the OS. We identify four main challenges for future OS design: scalability, elasticity of demand, faults, and large system programming difficulty. This work examines these challenges in both cloud and manycore systems. We raise the question of how these challenges can be addressed in a unified manner by a single solution. fos is a new OS designed to tackle these challenges while leveraging the potential of future systems.

This work extends fos [19] to run on Infrastructure as a Service (IaaS) cloud computers. We take the factored OS design and apply it to OS scalability challenges for current day multicore processors, future multicore/manycore processors, and IaaS Cloud infrastructures built out of multicore processors. fos is being built as the first prototypical cloud-aware OS. Furthermore, it is designed to provide a single system image across clusters and clouds built out of contemporary and future multicore microprocessors. This is in contrast to current solutions which require a cluster or cloud to be utilized as set of distinct machines, each running its own OS. Also, by attacking the scalability and manageability problem at the OS level, fos can support a broader range of applications than existing automatic management frameworks. Current frameworks (like those provided by RightScale [16]) only target specific application domains such as web servers.

This paper is organized as follows. Section 2 explores the challenges for OS design brought on by manycore and cloud computing. Section 3 describes the design of fos. Section 4 describes how fos's design solves these challenges. We briefly describe fos's current implementation state in Section 5. And finally, section 6 places fos in context with previous systems.

2 Common Challenges

2.1 Scalability

The number of transistors which fit onto a single chip microprocessor is exponentially increasing [14]. In the past, new hardware generations brought higher clock frequency, larger caches, and more single stream speculation. Single stream performance of microprocessors has fallen off the exponential trend[4]. In order to turn increasing transistor resources into exponentially increasing performance, microprocessor manufacturers have turned to integrating multiple processors onto a single die [20, 18]. On the other hand, current OS's were designed for single processor or small number of processor systems. The current multicore revolution promises drastic changes in fundamental system architecture, primarily in the fact that the number of general-purpose schedulable processing elements is drastically increasing. Therefore multicore OS's need to embrace scalability and make it a first order design constraint. In our previous work [19], we investigated the scalability limitations of contemporary OS design including: locks, locality aliasing, and reliance on shared memory.

Concurrent with the multicore revolution, cloud computing and IaaS systems have been gaining popularity. This emerging computing paradigm has a huge potential to

transform the computing industry and programming models [7]. The number of computers being added by cloud computing providers has been growing at a vast rate driven largely by user demand for hosted computing platforms. The resources available to a given cloud user is much higher than is available to the non-cloud user. Cloud resources are virtually unlimited for a given user, only restricted by monetary constraints. Example public clouds and IaaS services include Amazon's EC2 [1] and Rackspace's Cloud Server [2]. It is clear that scalability is a major concern for OS's in both single machine and cloud systems.

2.2 Elasticity of Demand

We define elasticity as the aspect of a system where parameters such as demand, workloads, and available resources change dynamically over time. In manycore systems the number of cores available is much larger than in single and or low core count system. Furthermore, the load on a manycore system translates into number of cores being utilized. Thus the system must manage the number of cores to match the demand of the user. For example, in a 1,000 core manycore system, the demand can require from one core to 1,000 cores. Therefore, multicore OS's need to manage the number of live cores in contrast to managing whether a single core is utilized or idle.

In cloud systems, user demand can grow much larger than in the past. Additionally, this demand is often not known ahead of time by the cloud user. It is often the case that users wish to handle peak load without over-provisioning. In contrast to cluster systems where the number of cores is fixed, cloud computing makes more resources available on-demand than was ever conceivable in the past.

A major commonality between cloud computing and manycore systems is that the demand is not static. Also, the variability of demand is much higher than in previous systems because the amount of available resources can be varied over a much broader range in contrast to single core or fixed sized cluster systems.

2.3 Faults

Dealing with software and hardware faults is another common challenge for future manycores and cloud systems. In manycore system, hardware faults are becoming more common. As the hardware industry is continuously decreasing the size of transistors and increasing their count on a single chip, the chance of faults is rising. With 100-1000s of cores per chip, the software components in the system must gracefully support cores dying and bit flips. In this regard, fault tolerance in modern OS's designed for manycore is becoming an essential requirement.

In addition, faults in large-scale cloud systems are common. Furthermore, cloud applications usually share cloud resources with other users and applications in the cloud. Although each users' application is encapsulated in a virtual container (virtual machines in EC2 model), performance interference from other cloud users and applications can potential impact the quality of service provided to the application.

Programming for massive systems is likely to introduce software faults. Due to the inherent difficulty of writing multithreaded and multiprocess applications, the likelihood of software faults in those applications is high. Furthermore, the lack of tools to debug and analyze those software systems makes these software faults hard to understand and challenging to fix. In this respect, dealing with software faults is another common challenge that programming for manycores and cloud systems share.

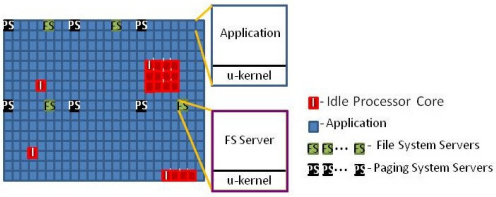

Fig. 1. OS and application clients executing on the fos-microkernel

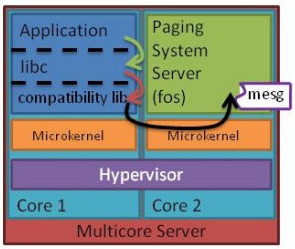

Fig. 2. Anatomy of a system call on fos.

2.4 Difficultly in Programming Large Systems

Contemporary OS's which execute on multiprocessor systems have evolved from uniprocessor OS's. This evolution was achieved by adding locks to the OS data structures. There are many problems with locks such as, choosing correct lock granularity for performance, reasoning about correctness, and deadlock prevention. Ultimately, programming efficient large-scale lock-based OS code is difficult and error prone. Difficulties of using locks in OSs is discussed in more detail in [19].

Developing cloud applications, composed of several components to be deployed across many machines is a difficult task. The prime reason for this is that current Infrastructure as a Service cloud systems impose an extra layer of indirection through the use of virtual machines. While on multiprocessor systems, the OS manages resources and scheduling, on cloud systems, much of this complexity, arising from the fragmented view of the resource pool, is pushed into the application.

Furthermore, there is not a uniform programming model for communicating within a single multicore machine and between machines. The current programming model requires a cloud programmer to write a threaded application to utilize intra-machine resources while socket programming is used to communicate with components of the application executing on different machines.

Scalability, elasticity of demand, faults, and difficult programming model are common issues for emerging Manycore and Cloud systems. In the remainder of this paper we present the design of fos and demonstrate how it leverages the potential of these emerging systems while solving these challenges.

3 Design of a Factored Operating System

In order to create an OS to tackle 1,000+ core processors and cloud computing datacenters, we propose a factored OS (fos) design. fos is an operating system which takes scalability and adaptability as *the* first order design constraints. Unlike most previous OS's where a subsystem scales up to a given point and beyond that point, the subsystem needs to be redesigned, fos ventures to develop techniques to build OS services which scale, up and down, across a large range ($>$ 3 orders of decimal magnitude) of core count.

3.1 Design Principles

In order to achieve the goal of scaling over multiple orders of magnitude in core count, fos utilizes the following design principles:

– Space multiplexing replaces time multiplexing.

    - Scheduling becomes a layout problem not a time multiplexing problem.
    - OS runs on distinct cores from applications.
    - Spatially partitioned working sets; OS doesn't interfere with application's cache.
- OS is factored into function specific services.
    - Servers collaborate and communicate only via message passing.
    - Servers are bound to a core.
    - Applications communicate with servers via message passing.
    - Servers leverage ideas from Internet servers.
- OS adapts resource utilization to changing system needs.
    - Each service utilization is measured.
    - Highly loaded system servers are provisioned more cores.
    - OS closely manages resources utilized.
- Faults are detected and handled by OS
    - OS services are monitored by watchdog process.
    - Name server reassigns communication channel on faults.

In the near future, we believe that the number of cores on a single chip or the number of cores in a cloud under one administrative domain will be on the order of the number of active threads in a system. When this occurs, the reliance on temporal multiplexing of resources will be removed, thus fos replaces traditional time multiplexing with space multiplexing. By scheduling resources spatially, traditional scheduling problems are transformed into layout and partitioning problems.

Spatial multiplexing is taken further as fos is factored into function specific services. Each system service is then distributed into a fleet of cooperating servers. All of the different function specific servers collaborate to provide a service such as a file system interface. Each server is bound to a particular processor core and communicates with other servers in the same service fleet via messaging only. When an application needs to access a service provided by the OS, the application messages the closest core providing that service, found by querying the name server. Name lookups returned by the name server are cached in the microkernel.

In order to build a scalable OS, the fos server is inspired by Internet servers. fos servers leverage ideas such as extensive caching, replication, spatial distribution, and lazy update, which have allowed Internet services to scale up to millions of users. Writing fos servers can be more challenging than writing traditional shared memory OS subsystems. We believe that the *shared nothing, message passing only, approach* will encourage OS developers to think carefully about what data is being shared, which will ultimately lead to more scalability.

A factored OS environment is composed of three main components. A thin microkernel, a set of servers which together provide system services which we call the OS layer, and applications which utilize these services. The lowest level of software management comes from the microkernel. A portion of the microkernel executes on each processor core. The microkernel controls access to resources (protection), provides a communication API and code spawning API to applications and system service servers, and maintains a name cache used internally to determine the location (physical machine and physical core number) of the destination of messages. Applications and

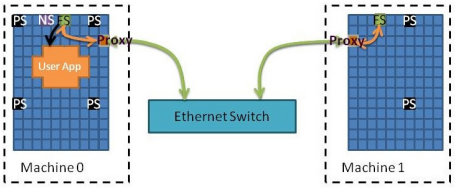

Fig. 3. Servers on differing machines message eachother by having messages proxied over Ethernet

system servers execute on separate cores on top of the microkernel and execute on the same core resources as the microkernel as shown in Figure 1.

fos is a full featured OS which provides many services to applications such as resource multiplexing, management of system resources such as cores, memory, and input-output devices, abstraction layers such as file-systems and networking, and application communication primitives. In fos, this functionality is provided by the OS layer. The OS layer is composed of fleets of function specific servers. Each OS function is provided by one or more servers. Each server of the same type is a part of a function specific fleet. Naturally there are differing fleets for different functions. For instance there is a fleet which manages physical memory allocation, a fleet which manages the file system access, and a fleet which manages process scheduling and layout. Each server executes solely on a dedicated processor core. Servers communicate only via the messaging interface provided by the microkernel layer.

fos provides the ability for applications to have shared memory if the underlying hardware supports it. The OS layer does not internally utilize shared memory, but rather utilizes explicit message based communication. When an application requires OS services, the underlying communication mechanism is via microkernel messaging. While messaging is used as the communication mechanism, a more traditional system call interface is exposed to the application writer. A small translation library (shown in Figure **??**) is used to turn system calls into messages from application to an OS layer server.

Applications and OS layer servers act as peers. They all run on top of the fos-microkernel and communicate via the fos-microkernel messaging API. The fos-microkernel does not differentiate between applications and OS layer servers executing under it. The code executing on a single core is called a fos client. Figure 1 has a conceptual model of applications and the OS layer, as implemented by fleets of servers.

In the cloud, fos provides a uniform naming and messaging interface across clouds of multicore processors. Messaging a neighboring core is the same as messaging a core on a different machine. The name server takes care of much of the complexity of determining where a resource resides. When a message is sent between machines, the local name server identifies a cloud message proxy server as the destination of a message. The local cloud message proxy server packages up the message and sends it via the network interface to the required machine where the message is un-encapsulated and send to the recipient server as shown in Figure 3.

4 fos Solution

4.1 Scalability

To enable applications and the OS to scale, fos incorporates several approaches. First, the OS is factored by service being provided. By factoring by system service, scalability

is increased because each service can run independently. After factoring by service, each service is factored again into a fleet of spatially distributed servers. Each server within a fleet executes on its own core thereby increasing the parallelism available. By spatially distributing system servers, locality can be exploited by both reducing communication cost and increasing data access locality.

A given OS service is composed of multiple replicated servers, collectively known as a fleet. Within this fleet, new servers can be added or removed to allow for service load to scale up or down. A fleet of servers communicates amongst themselves. This intra-fleet communication is done via message passing. Differing approaches are used to increase the system service throughput. For example the physical page allocation service can divide the pages based off the address across the fleet or separate page allocation servers call allocate from the same pool by using a consensus protocol.

Applications and the OS have largely different working sets. To address this problem and avoid implicitly sharing hardware structures such as caches and translation lookaside buffers (TLBs), fos executes the OS on separate processing cores from the application. Also, different portions of the OS are factored apart such that different OS working sets do not interfere in implicitly accessed data structures.

To facilitate scaling across the cloud, fos's messaging API provides a transparent communication mechanism. The messaging API is transparent with respect to whether a message's destination is on the same machine or on a different machine. As a result, it is easy to move OS services to different machines as needed. Furthermore, fos is designed to manage multiple machines thereby enabling the scaling up of an OS across the cloud. OS data structures contain meta-data of not only which core a resource resides on, but also which machine. fos's OS APIs contain core and machine specifiers thereby enabling a OS server on one machine to manage the resources for cores on a different machine.

4.2 Elasticity of Demand

Architectural features such as disabling cores, frequency scaling, low power modes, and provisioning VMs provide several different options for configuring cores. As the optimal decision for these parameters depends on a global view of the system, the OS is generally responsible for setting these parameters. First, fos solves elasticity of demand by dynamically varying the overall number of cores utilized. Second, fos dynamically allocates an optimal number of cores to system servers in an on-demand basis. Each of the OS services provides a feedback mechanism to the scheduler. The scheduler uses the feedback from several system services and application load to make an informed decision about how to adapt the system. In particular the scheduler may decide to spawn more servers if a particular system service is under heavy load, or reduce the number under light load. Feedback about communication patterns will allow the scheduler to make informed decisions about process migration to co-locate processes with a high amount of communication. Likewise the adaptability of fos's scheduler allows it to co-locate processes with their data based on access patterns. Migration is particularly beneficial when it reduces high-latency communication costs as in the multi-machine case. In example, migrating communicating processes on separate machines onto one machine.

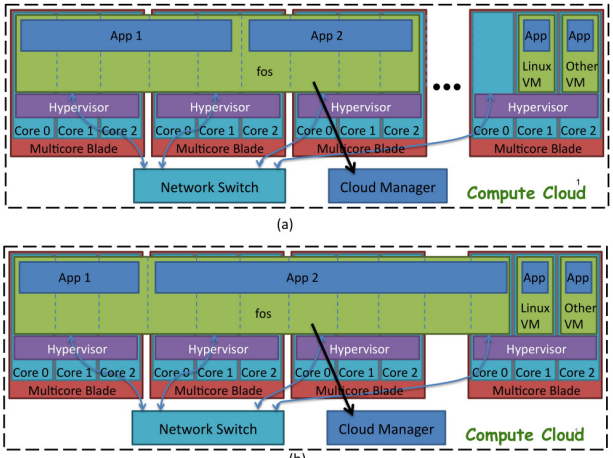

Fig. 4. fos provides a single system image across all the cloud nodes. Two snapshots in time demonstrate how changing demand by the application is handled by provisioning different numbers of VMs to grow and shrink cloud resources as needed.

fos addresses elasticity of demand by providing a single system image OS which can grow and shrink across a cloud. Figure 4 demonstrates fos's single system image and dynamic resources acquisition and release in the cloud. By utilizing a single system image, fos is able to gain a more coherent and global view of the entire system when compared to batch or cluster administration systems. It accomplishes this by having OS system services communicate across machine boundaries. Also, by allowing communication of other machines' meta-data so readily, the OS is able to make an informed decision about load balancing and process migration.

4.3 Faults

fos is designed to be fault tolerant by using feedback from running servers to detect when a fault has occurred and takes advantage of the replicated design to recover from such faults. The basic idea is similar to a watchdog timer in that the monitor is expecting a periodic message identifying the server as running properly. When one of these messages is not received, the server is assumed to have faulted and recovery action is taken. This design provides a common mechanism for handling both hardware and software faults.

Once the system has detected the fault, fos can recover from the fault as it was designed with this scenario in mind. First a new server is spawned and joins the fleet taking over responsibility for the resources and computation previously assigned to the faulting server. Next the name server is updated to route messages destined for the old server to the new server. This allows the fault to be transparent to any application or system server trying to interact with the faulting server. In addition to fault detection and recovery, the factored design leads to fault isolation.

fos is built out of replicated servers which makes replacing a faulting server similar to adding a server to a fleet. In the cloud, it is common for a whole machine to fail. fos's fault detection and recovery extends to this eventuality as well.

4.4 Difficultly in Programming Large Systems

fos is a cloud-aware OS that provides a single system image for cloud applications. fos simplifies programmability and allows cloud applications to utilize the power of cloud computing by handling the dynamic nature of the cloud workloads.

Unlike traditional cloud and cluster systems, fos provides a single system image to the application. This means that the application interface is that of a single machine while the OS implements this interface across several machines in the cloud. The OS dynamically decides how to distribute the set of applications and system servers across various machines in the cloud.

Many of the traditional parallel applications generally use a fixed number of threads or processes defined as a parameter at the start of the application. The decision for the number of threads is often decided by the user in an effort to fully utilize the parallel resources of the system or to meet peak demand of a particular service. fos uses the replicated server model which allows additional processing units to be dynamically added during runtime allowing the system to achieve a better utilization for dynamic workloads and alleviating the user from such concerns.

5 fos Progress

fos has been implemented as a Xen Para-Virtualized Machine (PVM) OS. We decided to implement fos as a PVM OS in order to support the cloud computing goals of this project, as this allows us to run fos on EC2 and Eucalyptus cloud infrastructure. It also simplifies the driver model as the Xen PVM interface abstracts away many of the details of particular hardware. fos and the underlying design does not require a hypervisor to execute but are simply used for convenience. fos is currently a multitasking multiprocessor OS executing on real x86_64 hardware. We have a working microkernel, messaging layer, naming layer, protected memory management, a spawning interface, a basic system server, and we are now expanding our collection of system servers. Our next step is to extend fos across the cloud with an addition of a cloud message proxy server in order to allow fos messaging between machines.

Some preliminary results have been gathered comparing system call time in fos and Linux as provided below.

Benchmark	Cycles
null system call (Linux)	1789
null system call (fos)	20031

6 Related Work

There are several classes of systems which have similarities to fos: traditional microkernels, distributed OS's, and cloud computing infrastructure.

Traditional microkernels include Mach [3] and L4 [12]. fos is designed as a microkernel and extends the microkernel design ideas. However, it is differentiated from previous microkernels in that instead of simply exploiting parallelism between servers which provide different functions, this work seeks to distribute and parallelize within a server for a single high-level function. fos also exploits the spatial-ness of massively multicore processors by spatially distributing servers which provide a common OS function.

Like Tornado [11] and K42 [5], fos explores how to parallelize microkernel based OS data structures. They are differentiated from fos in that they require SMP and

NUMA shared memory machines instead of loosely coupled single-chip massively multicore machines and clouds of multicores. Also, fos targets a much larger scale of machine than Tornado/K42. The recent Corey [9] OS shares the spatial awareness aspect of fos, but does not address parallelization within a system server and focuses on smaller configuration systems. Also, fos is tackling many of the same problems as Barrelfish [8] but fos is focusing more on how to parallelize the system servers as well as addresses the scalability on chip and in the cloud.

fos bears much similarity to a distributed OS such as Amoeba [17], Sprite [15], or Clouds [10]. One major difference is that fos communication costs are much lower when executing on a single massive multicore, and the communication reliability is much higher. Also, when fos is executing on the cloud, the trust model and fault model is different than previous distributed OS's where much of the computation took place on student's desktop machines.

fos differs from existing cloud computing solutions in several aspects. Cloud (*IaaS*) systems, such as Amazon's Elastic compute cloud (EC2) [1], provide computing resources in the form of virtual machine (VM) instances and Linux kernel images. fos builds on top of these virtual machines to provide a single system image across a IaaS system. With the traditional VM approach, applications have a poor control over the co-location of the communicating applications/VMs. Furthermore, IaaS systems do not provide a uniform programming model for communication or allocation of resources. Cloud aggregators such as RightScale [16] provide automatic cloud management and load balancing tools, but they are application specific while fos provides these features in an application agnostic manner. Platform as a service (*PaaS*) systems, such as Google AppEngine [6] and MS Azure [13], represent another cloud layer which provides APIs that applications can be developed for. PaaS systems often provide automatic scale up/down and fault-tolerance as features, but are typically language-specific. fos tries to provide these benefits in a application agnostic manner.

References

1. Amazon Elastic Compute Cloud (Amazon EC2), 2009. http://aws.amazon.com/ec2/.
2. Cloud hosting products - Rackspace, 2009. http://www.rackspacecloud.com/cloud_hosting_products.
3. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, pages 93–113, June 1986.
4. V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the International Symposium on Computer Architecture*, pages 248–259, June 2000.
5. J. Appavoo, M. Auslander, M. Burtico, D. M. da Silva, O. Krieger, M. F. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. K42: an open-source linux-compatible scalable operating system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.
6. Google appengine – http://code.google.com/appengine/.
7. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

8. A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.
9. S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. D. Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, Dec. 2008.
10. P. Dasgupta, R. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. Applebe, J. M. Bernabeu-Auban, P. Hutto, M. Khalidi, and C. J. Wileknloh. The design and implementation of the Clouds distributed operating system. *USENIX Computing Systems Journal*, 3(1):11–46, 1990.
11. B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 87–100, Feb. 1999.
12. J. Liedtke. On microkernel construction. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 237–250, Dec. 1995.
13. Microsoft azure. http://www.microsoft.com/azure.
14. G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, Apr. 1965.
15. J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, Feb. 1988.
16. Rightscale home page. http://www.rightscale.com/.
17. A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 558–563, May 1986.
18. S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 98–99, 589, Feb. 2007.
19. 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, 2009.
20. D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal. On-chip interconnection architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, Sept. 2007.