Osprey: Operating System for Predictable Clouds

Jan Sacha, Jeff Napper, Sape Mullender
Bell Laboratories
Alcatel-Lucent
Antwerp, Belgium

Jim McKie
Bell Laboratories
Alcatel-Lucent
Murray Hill, NJ

***Abstract*—Cloud computing is currently based on hardware virtualization wherein a host operating system provides a virtual machine interface nearly identical to that of physical hardware to guest operating systems. Full transparency allows backward compatibility with legacy software but introduces unpredictability at the guest operating system (OS) level. The time perceived by the guest OS is non-linear. As a consequence, it is difficult to run real-time or latency-sensitive applications in the cloud. In this paper we describe an alternative approach to cloud computing where we run all user applications on top of a single cloud operating system called Osprey. Osprey allows dependable, predictable, and real-time computing by consistently managing all system resources and exporting relevant information to the applications. Osprey ensures compatibility with legacy software through OS emulation provided by libraries and by porting runtime environments. Osprey's resource containers fully specify constraints between applications to enforce full application isolation for real-time execution guarantees. Osprey pushes much of the state out of the kernel into user applications for several benefits: full application accounting, mobility support, and efficient networking. Using a kernel-based packet filter, Osprey dispatches incoming packets to the user application as quickly as possible, eliminating the kernel from the critical path. A real-time scheduler then decides on the priority and order in which applications process their incoming packets while maintaining the limits set forth in the resource container. We have implemented a mostly complete Osprey prototype for the x86 architecture and we plan to port it to ARM and PowerPC and to develop a Linux library OS.**

***Keywords*-cloud; virtualization; predictability; real-time;**

I. Introduction

A growing number of network services today are built on cloud computing infrastructure. Cloud computing is typically based on hardware virtualization wherein a host operating system (virtual machine monitor, or hypervisor) provides a virtual machine interface nearly identical to that of physical hardware. Services with strict latency or timing requirements, such as telecommunications systems, are difficult to run in current clouds: Hardware virtualization introduces unpredictability at the guest operating system (OS) level that violates key assumptions the guest system makes about the hardware it is running on. The time perceived by the guest system in a virtual machine is non-linear when sharing hardware, making it difficult to provide reliable timing guarantees. Real-time scheduling is only possible if the guest system cooperates with the host operating system, which violates full transparency.

Although hardware virtualization allows backward compatibility with legacy software, full transparency comes at a high cost. In addition to real-time scheduling problems, virtualization also hides the real properties of redundant system components such as the storage devices, compute nodes, and network links of different systems to the level of the datacenter.

In paravirtualization [1], [2], [3], the virtual machine interface is close but not identical to the physical hardware and the guest operating system is aware of virtualization. However, paravirtualization has been introduced to simplify the hypervisor and improve application performance rather than provide predictable performance and dependability. For instance, the hypervisor might provide real-time information to the guest OS but it does not cooperate in scheduling and thus the guest OS is not able to provide real-time guarantees to the applications.

In this paper we describe an alternative approach to using hardware virtualization for cloud computing. We abandon the split between host and guest operating systems to run all user applications on top of a single cloud operating system called Osprey. We trade full virtualization transparency for predictable, real-time application execution and fault-tolerance.

Osprey provides a virtual machine interface at the system call level as opposed to instruction set (ISA) level in hardware virtualization. This approach, known as OS virtualization [4], [5], incurs lower overhead compared to hardware virtualization, while both approaches introduce similar benefits to cloud computing including application sandboxing, server consolidation, checkpoint/restart, and migration. Hardware virtualization introduces unnecessary overhead because an application runs on top of two operating systems (host and guest) which have separate sets of drivers, data buffers, resource management policies, etc. that often conflict.

In this paper we claim that OS virtualization allows dependable, predictable, and real-time computing since the operating system can consistently manage all system resources while exporting relevant information to the application. In Osprey, we combine OS virtualization with library

operating systems [6], [7], [8], which implement legacy OS personalities in the user space providing compatibility with legacy software. We implement library OSes on top of the Osprey kernel to enable high performance, real-time guarantees, and scalability on modern, massively multicore hardware. In the following sections we describe the key features that allow Osprey to meets these goals.

II. Predictable performance

We are expecting that the CPU core count in future hardware is going to gradually increase and memory access is going to be less and less uniform. Consequently, an operating system for future clouds should be able to execute real-time processes efficiently on multi-core NUMA hardware.

Similar to the multikernel model [9], we partition most of the kernel state and physical memory in Osprey between CPU cores in order to reduce synchronization cost, increase parallelism, maximize the efficiency of CPU memory caches, and reduce the load on CPU interconnects. In particular, we pin user processes to cores and we run an independent scheduler on each core. A load balancer occasionally migrates processes between cores. Given a large number of cores in the system and a smart process placement heuristic, we expect cross-core process migrations to be relatively infrequent. Each core is assigned its own memory allocation pool to reduce cross-core memory sharing. Parts of the kernel address space are shared between the cores but there is also a region that is private for each core.

Osprey supports several types of locks and semaphores for synchronization. However, since most kernel data structures are partitioned between cores, access to these data structures can be protected using cheap locks which disable interrupts and do not require busy waiting. We use global system locks infrequently and keep critical code sections to a necessary minimum in order to avoid wasting CPU cycles on lock contention and to eliminate unknown wait times to provide real-time guarantees.

Osprey generally attempts to minimize the number of traps, interrupts, and context switches since they have an adverse effect on system performance and real-time guarantees. Clock and inter-core interrupts are only used if a process needs to be preempted. Interrupts from I/O devices are made as short as possible and merely wake driver tasks which run at their own priority and have their own resource budgets. We also redirect I/O interrupts to a *noise core* so that all other cores can run uninterrupted. Further, we disable clock interrupts if a core has only one runnable process. The kernel uses an inter-core interrupt to activate the scheduler if necessary.

The Osprey kernel is fully mapped in using large memory pages to avoid page faults and reduce TLB misses. Osprey also does not swap application memory to disk since swapping incurs a high performance penalty which greatly decreases application predictability.

Inter-process communication, as well as user-kernel communication, is performed using shared-memory asynchronous message queues. Traditional trap-based system calls are replaced in Osprey by these message queues that allow user processes to batch multiple kernel requests to reduce context switches. Processes need to trap to the kernel only in cases where the process fills up its queue or waits for an incoming message. Message-based system calls also allow the kernel to handle requests on a different core while the user process is running in parallel, maximizing throughput.

Since writing event-driven applications is considered difficult, Osprey provides a lightweight, cooperative user-level threading library on top of asynchronous system calls. Osprey threads require almost no locking. When a thread yields or blocks waiting for a message, the library schedules another runnable thread, allowing the application to perform multiple system calls in parallel without ever trapping or blocking the process.

III. Application isolation

Cloud applications often share hardware in order to run in a cost-effective manner, amortizing the costs of execution over multiple applications. It is crucial for a cloud operating system to isolate applications from each other in order to provide them with enough resources to meet their Service-Level Agreements (SLA). Current SLAs offered by cloud providers specify only an annual uptime because fine-scale performance is very hard to guarantee with virtual machines sharing hardware resources.

Each application in Osprey runs in a resource container that controls resource consumption by the application. We distinguish between two types of resources: preemptive and non-preemptive. Preemptive resources, such as CPU time or network interface bandwidth, can be freely given to the application and taken back by the kernel when needed. Resource containers thus specify the minimum amount of preemptive resources guaranteed for the application.

Non-preemptive resources, such as memory, cannot be easily taken back once granted (note that we abandon swapping due to its high cost), and thus resource containers limit the maximum resource consumption. If the kernel does not over-provision, these limits are equivalent to guarantees. Further, an application can ensure receiving enough non-preemptive resources by allocating them upfront.

A resource container in Osprey describes the maximum amount of CPU time the application is allowed to consume and the maximum delay the application can tolerate (deadline) when receiving an external event such as a network packet. A real-time scheduler guarantees that all applications obey the limits set in their resource containers. Beside time-division multiplexing, resource containers can also divide entire cores between applications. This way of managing CPU resources is more coarse-grained compared to time

sharing, but it introduces less overhead and enables more predictable computing.

A resource container also defines the application's memory budget, both for the main memory (RAM) and external storage (e.g., disk space). Using techniques such as cache coloring—potentially with support from future hardware—a resource container could also control the use of CPU caches (TLB, L1, L2, L3) and I/O bus cycles by user applications. Finally, a resource container also limits the maximum number and total bandwidth of network packets that an application is allowed to send.

An Osprey resource container can also provide a private namespace to the application by translating between global system identifiers used by the kernel and local names used by the application. Namespace isolation improves application security and provides configuration independence.

Since cloud applications can share physical network interfaces, Osprey needs to manage the use of low-level network addresses by user applications. As with current cloud systems, Osprey will multiplex low-level network protocols in the kernel, giving the illusion that every application has an exclusive access to the network. However, this illusion can always be broken to verify whether applications share physical hardware in order to guarantee fault-tolerance.

Isolation is further enhanced in Osprey by moving most of the traditional operating system's functionality to user space. This approach allows Osprey to enforce simple application isolation and accounting. Osprey applications require very little kernel state and spend very little time in the kernel because they run entire network protocol stacks and most file systems in user space.

IV. Backward compatibility

In order to successfully deploy a new cloud infrastructure, it is important to be able to execute legacy applications. There are several ways of supporting legacy applications in Osprey: legacy system emulation, library operating systems, and porting native runtime environments. These methods differ in the degree of compatibility with legacy code and provide a tradeoff between simple migration and modifying the code to take advantage of Osprey services.

The most powerful mechanism to execute unmodified binary applications is based on legacy system emulation. In this approach, a legacy application is associated with a peer process, which we call a buddy operating system (buddyOS), that handles all system traps from the legacy applications such as system calls and page faults and provides kernel file systems for devices and processes such as `/dev` and `/proc`. Although developing operating system emulators requires a significant engineering cost, it has been demonstrated to be feasible even for the largest commercial operating systems [10].

A second mechanism supports unmodified legacy code by relinking the source code to a library operating systems (libOS) implementation. A libOS is a user-level library that implements the legacy system call functions (and possibly some system libraries) and translates them all to native Osprey system calls. It also provides legacy file systems. Compared to a system emulation for binary compatibility, a libOS is more efficient because it runs in the same address space as the user application. It has been shown that a libOS can successfully run sophisticated commercial applications [6], [7], [11].

Finally, Osprey will implement native runtime environments for some programming languages such as Java and Python or runtime environments like POSIX that allows Osprey to run unmodified legacy applications (binaries, bytecode, or scripts) written in these languages or environments. This approach would allow Osprey to provide an interface to user applications similar to commercial platforms such as Google App Engine.

Importantly, buddyOSes, libOSes, and runtime environments will be implemented natively against the Osprey kernel API taking advantage of Osprey's scalability and predictable (real-time) performance. The compatibility layers will extend Osprey's functionality but will avoid as much as possible redundant resource management. In particular, they will rely on the Osprey scheduler, resource containers, and memory management.

V. Efficient networking

A cloud operating system needs to support efficient networking for user applications. Ideally, a cloud operating system should provide: low-latency communication in order to maximize application responsiveness, maximum throughput in order to fully utilize hardware, and support for mobility events such as migration to new hosts and network address changes.

In order to address these goals in Osprey, we run all network protocol stacks in user space together with the user application. Such a design has a number of advantages. By dispatching incoming packets from the network interface to the user application as quickly as possible we eliminate the kernel from the critical path and allow user applications to receive packets with a minimum delay. A real-time scheduler decides on the priority and order in which applications can inspect their incoming packets while maintaining the limits set forth in the resource container. Similarly, the scheduler serializes outgoing packets to the hardware from different applications.

User-level networking also improves modern multi-core hardware utilization. By generating and consuming network packets on their respective application cores we parallelize network traffic processing and improve memory cache efficiency since each packet is mostly accessed on one core.

User-level networking simplifies operations in cloud environments such as application checkpointing, suspension, resumption, and migration. In Osprey, much of the state

that is maintained by the kernel in traditional operating systems is moved to user space libraries. Pushing all the application-specific state (connection status, packet buffers, etc.) into the user application greatly simplifies application marshaling and unmarshaling by the kernel. Further, it allows customized network protocols stacks to respond to mobility and fault events in an application appropriate way.

Finally, we envisage that the main bottleneck when delivering high-volume traffic from the network interface to the user application is the host's memory bus. Indeed, it has been shown that more and more applications (especially multimedia and telecommunication) are memory-bound. In order to achieve high throughput, Osprey thus supports zero-copy networking.

VI. Checkpoint and restart

A cloud computing infrastructure needs to support application checkpointing and restarting for reasons such as fault tolerance and energy saving. Checkpoint and restart functionality also enables application migration if the target application can be restarted on a new machine.

Osprey supports application checkpoint, restart, and migration in two ways. First, due to library and buddy operating systems, it stores almost all application state in the user space and can thus efficiently save it and restore it.

Second, resource containers in Osprey isolate application namespaces preventing name clashes when applications are restarted or migrated. Private namespaces also decouple applications from non-portable, host-specific names, such as `/dev/tty7`, and help virtualize resources by providing aliases, such as `/dev/console`, which can be remapped upon migration.

VII. Related work

Arguably, many ideas described in this paper are not new. However, the mechanisms and techniques that we use in Osprey have mostly been studied in isolation while we combine them into one consistent architecture for dependable cloud computing. In the following section we outline the systems that had the strongest impact Osprey's design.

The original resource containers [12] were introduced to manage resource consumption by user applications. In traditional operating systems protection domains and resource principals coincided with processes and the kernel provided fairness by treating all processes (or threads) alike. Resource containers essentially allow the kernel to divide resources between arbitrary activities which may span multiple processes and threads and may involve both user-space and kernel-space resource consumption.

A Jail [13] is a facility originally introduced in FreeBSD which allows partitioning the system into administrative domains with separate users, processes, file systems, and network addresses. While resource containers provide performance isolation, jails provide security isolation.

Zap [14] introduced a Process Domain (POD) abstraction which essentially provides a private namespace to the containing processes. It is implemented as a thin layer between user processes and the kernel which translates names used in system calls such as process identifiers, IPC keys, filenames, device names, and network addresses to pod-local names. By decoupling pods from global system names Zap prevents name clashes when pods are checkpointed, restarted, and migrated.

A technique that combines resource containers, jails, and pods, and thus provides performance, security, and configuration isolation, is known as operating system (OS) virtualization [4], [15]. OS virtualization has been implemented in major operating system, for example in the form of Zones [16] in Solaris and VServers [4] and Virtual Private Servers (VPS) [17] in Linux. A VPS is transparent to user applications and provides an environment almost identical to a non-virtualized Linux system running natively on hardware. VPS enables Application Checkpointing and Restart (ACR) [17] in Linux.

Scout [18] and its Linux implementation called SILK [19] is an architecture that allows efficient network traffic prioritization and processing. SILK classifies every incoming packet using a packet filter [20] and assigns it to a path, i.e., a chain of processing elements such as network protocol engines and user applications. Every path is scheduled based on its priority. Osprey achieves a similar goal by early demultiplexing network traffic using a packet filter and handing packets to user-level libraries which run together with the application and are subject to scheduling and resource container policies.

Several operating systems address modern and future massively multicore hardware. Interestingly, all these systems are based on three simple observations. First, centralized or shared data structures are avoided to reduce synchronization overhead. Second, sharing between cores is minimized, and thereby locality is maximized, to enable efficient use of CPU caches and interconnects. Finally, the kernel exposes information about hardware to user applications to allow them to optimize their performance.

Corey [21] is an operating system that allows users to explicitly decide which data structures, such as file descriptors, sockets, and address space ranges, are shared between processes and threads. By avoiding unnecessary sharing it reduces synchronization overhead and improves caching performance. Corey also allows applications to dedicate cores to execute kernel functions, for example to manage physical devices, to reduce cross-core sharing.

The Factored Operating System (fos) [22] requires each core to specialize and execute a single user or kernel process only. Processes do not migrate between cores and communicate by exchanging messages. Fos thus replaces time sharing with space sharing.

Similarly, NIX [23] introduces three types of cores which

can execute either a single user process, a group of kernel tasks, or a mixture of kernel tasks and user processes. In the former case, the core does not receive any interrupts and forwards system calls using messages to kernel cores which allows user processes to achieve fully predictable performance.

We apply some of these principles in Osprey. Our tasks and processes are pinned to cores and cores are divided between applications using resource containers. If a core runs only one user process, we disable clock interrupts and let the process run undisturbed. Interrupts from I/O devices can be handled on a noise core, and similarly, system call requests can be forwarded through message queues to kernel cores.

Barrelfish [9] is an operating system based on the multikernel design principle where each core maintains its own scheduler and communicates with other cores using messages. There is no shared memory between cores apart from message queues. We adopt the multikernel design in Osprey by partitioning most of the kernel state, running an independent scheduler on each core, and using message queues for cross-core communication. Unlike Barrelfish, Osprey provides real-time guarantees and uses inter-core interrupts to enforce scheduling priority.

Akaros [24] exposes information about physical cores and memory to user applications to allow them to optimize their performance. Similar to FlexSC [25], it supports asynchronous system calls and provides a threading library which minimizes context switches and improves application performance and throughput. Osprey has a similar scheduling model, but in contrast to Akaros, Osprey threads are cooperative which allows them to avoid locks. Since cooperative threads cannot be preempted, Osprey must suspend the entire process (all threads) on a page fault while Akaros can redirect the fault to the thread scheduler and run another thread inside the process. However, Osprey does not swap pages out to disk and hence a page can fault at most once.

K42 [26] is the closest to Osprey operating system that we are aware of. K42 addresses high scalability by decentralizing its data structures, avoiding locks, and reducing cross-core memory sharing. It also moves most system functionality to user-level libraries and server processes. It has an object-oriented architecture designed for extensibility and customization. Unlike Osprey, it runs performance monitors and adaptive algorithms that perform dynamic system updates and reconfiguration. K42 has only one compatibility layer (for Linux) while Osprey aims to provide multiple operating system personalities.

VIII. Current status

We have implemented an Osprey prototype for the x86 32-bit and 64-bit architectures. The prototype consists of a kernel, a suite of user-space libraries, and several user programs. The kernel contains an Earliest Deadline First (EDF) real-time scheduler, memory management, low-level hardware management, basic system call handlers, and a packet filter. The libraries implement efficient message queues, cooperative user-level threading, and a number of network protocols. We also implemented essential device drivers, including USB and high-speed (10G) Ethernet drivers, and a framework for efficient zero-copy networking. We have ported an Inferno library OS [11] to Osprey, and we are currently working on a Plan 9 libOS [27]. We are also actively working on filesystems, resource-containers, and checkpoint/restart support. In the mid-term future we plan to port Osprey to PowerPC, ARM, and possibly MIPS, and to provide a Linux compatibility layer.

IX. Conclusion

Osprey is an operating systems for predictable cloud performance. It uses an alternative approach to using hardware virtualization for cloud computing where all user applications run on top of a single operating system. Full transparency is broken to enable competing applications to meet real-time and fault-tolerant constraints. Resource containers fully specify constraints between applications to enforce full application isolation to provide predictability. Osprey pushes much of the state that is typically in the kernel into user applications for several benefits: efficient networking, mobility support, and full application accounting. Close compatibility with legacy software is supported through OS emulation provided by libraries or by porting runtime environments. We have developed an Osprey prototype and plan to thoroughly evaluate and compare it with existing cloud computing architectures.

References

[1] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg, "The performance of μ-kernel-based systems," in *Proceedings of the 16th Symposium on Operating Systems Principles*. ACM, 1997, pp. 66–77.

[2] A. Whitaker, M. Shaw, and S. D. Gribble, "Scale and performance in the denali isolation kernel," *SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 195–209, Dec. 2002.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th Symposium on Operating Systems Principles*. ACM, 2003, pp. 164–177.

[4] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *Proceedings of the 2nd EuroSys European Conference on Computer Systems*. ACM, 2007, pp. 275–287.

[5] O. Laadan and J. Nieh, "Transparent checkpoint-restart of multiple processes on commodity operating systems," in *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2007, pp. 25:1–25:14.

[6] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library OS from the top down," in *Proceedings of the 16th Conference on Architectural support for programming languages and operating systems*. ACM, 2011, pp. 291–304.

[7] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski, "Libra: a library operating system for a JVM in a virtualized execution environment," in *Proceedings of the 3rd Conference on Virtual Execution Environments*. ACM, 2007, pp. 44–54.

[8] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: an operating system architecture for application-level resource management," in *Proceedings of the 15th Symposium on Operating Systems Principles*. ACM, 1995, pp. 251–266.

[9] 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 *Proceedings of the 22nd Symposium on Operating Systems Principles*. ACM, 2009, pp. 29–44.

[10] B. Amstadt and M. K. Johnson, "Wine," *Linux Journal*, August 1994.

[11] S. M. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. W. Trickey, and P. Winterbottom, "The inferno operating system," *Bell Labs Technical Journal*, vol. 2, pp. 5–18, 1997.

[12] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: a new facility for resource management in server systems," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*. USENIX Association, 1999, pp. 45–58.

[13] P.-H. Kamp and R. N. M. Watson, "Jails: Confining the omnipotent root," in *Proceedings of the 2nd International SANE Conference*, 2000.

[14] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: a system for migrating computing environments," *SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 361–376, Dec. 2002.

[15] O. Laadan and J. Nieh, "Operating system virtualization: practice and experience," in *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*. ACM, 2010, pp. 17:1–17:12.

[16] D. Price and A. Tucker, "Solaris zones: Operating system support for consolidating commercial workloads," in *Proceedings of the 18th USENIX Conference on System Administration*. USENIX Association, 2004, pp. 241–254.

[17] S. Bhattiprolu, E. W. Biederman, S. Hallyn, and D. Lezcano, "Virtual servers and checkpoint/restart in mainstream linux," *SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 104–113, Jul. 2008.

[18] D. Mosberger and L. L. Peterson, "Making paths explicit in the scout operating system," in *Proceedings of the 2nd Symposium on Operating Systems Sesign and Implementation*. ACM, 1996, pp. 153–167.

[19] A. Bavier, T. Voigt, M. Wawrzoniak, L. Peterson, and P. Gunningberg, "SILK: Scout paths in the linux kernel," Department of Information Technology, Uppsala University, Sweden, Tech. Rep., Feb. 2002.

[20] J. Mogul, R. Rashid, and M. Accetta, "The packer filter: an efficient mechanism for user-level network code," in *Proceedings of the 11th ACM Symposium on Operating Systems Principles*. ACM, 1987, pp. 39–51.

[21] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: an operating system for many cores," in *Proceedings of the 8th Conference on Operating Systems Design and Implementation*. USENIX Association, 2008, pp. 43–57.

[22] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 76–85, Apr. 2009.

[23] F. J. Ballesteros, N. Evans, C. Forsyth, G. Guardiola, J. McKie, R. Minnich, and E. Soriano, "Nix: An operating system for high performance manycore computing," *To appear in Bell Labs Technical Journal*, 2012.

[24] B. Rhoden, K. Klues, D. Zhu, and E. Brewer, "Improving per-node efficiency in the datacenter with new OS abstractions," in *Proceedings of the 2nd Symposium on Cloud Computing*. ACM, 2011, pp. 25:1–25:8.

[25] L. Soares and M. Stumm, "FlexSC: flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th Conference on Operating Systems Design and Implementation*. USENIX Association, 2010, pp. 1–8.

[26] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig, "K42: building a complete operating system," in *Proceedings of the 1st EuroSys European Conference on Computer Systems*. ACM, 2006, pp. 133–145.

[27] R. Pike, D. L. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, "Plan 9 from bell labs," *Computing Systems*, vol. 8, no. 2, pp. 221–254, 1995.