Processes and Resource Management in a Scalable Many-core OS*

Kevin Klues, Barret Rhoden, Andrew Waterman, David Zhu, Eric Brewer
Computer Science Division, University of California at Berkeley
{*klueska, brho, waterman, yuzhu, brewer*}*@eecs.berkeley.edu*

Abstract

The emergence of many-core architectures necessitates a redesign of operating systems, including the interfaces they expose to an application. We propose a new operating system, called ROS, designed specifically to address many limitations of current OSs as we move into the many-core era. Our goals are (1) to provide better support for parallel applications for both high-performance and general purpose computing, and (2) to scale the kernel to thousands of cores. In this paper, we focus on the process model and resource management mechanisms of ROS. We expand the traditional process model to include the notion of a 'many-core' process designed to naturally support parallel applications. Additionally, we discuss our method of resource management that builds on the ideas of Space-Time Partitioning presented in our previous work [16]. Central to our design is the notion that protection domains should *not* necessarily be coupled with resource management, and resource partitioning is *not* necessarily coupled with resource allocation.

1 Introduction

Current operating systems were originally designed for uniprocessor systems. These systems have had SMP support for years, but it was initially added with a small number of nodes in mind. Today, Linux supports up to 4096 nodes in theory [15], but not in practice [8], and the system is an evolution from the original SMP design. The kernel itself has been modified to scale past previous bottlenecks, but there has been nothing fundamentally different for the many-core era. For example, every node potentially runs all of the kernel code and participates equally in the management of resources and scheduling. Nor is it built with explicit support for parallel processes.

Parallel applications are performance sensitive to the underlying state of the machine and to any OS processing that occurs. It is essential for the application to be able to gather information about the state of the system and to make requests that influence the decisions made by the OS [3, 4]. Additionally, the system must contain mechanisms to ensure performance isolation between competing applications.

In light of these problems, future many-core operating systems will need to provide better support for parallel applications and scale well to a large number of cores. One solution that has gained traction in recent years is the use of virtual machine monitors (VMMs) to partition an underlying machine's physical resources. As pointed out by Roscoe et al. [23], this approach has a number of limitations that make it unappealing as a solution as we move forward. VMMs introduce an extra layer that negatively impacts performance and increases complexity, especially with respect to process and memory management. Indeed, the original Disco [9] paper that kickstarted the VM trend acknowledges that virtual machines only emerged as a solution because we don't yet know how to build scalable operating systems.

Thus, we believe the many-core transition is a rare opportunity to revisit basic OS technologies with a chance for high impact. We propose a new many-core OS, called ROS, designed from the ground up to provide better support for parallel applications and improved kernel scalability; an early prototype of ROS already runs simple, parallel applications up to 16 cores on Intel Core i7 and 64 cores on our FPGA-based hardware emulator [25]. Although a number of other operating systems have begun to emerge that address similar issues [7, 8, 31], each focuses it attention on different aspects in the space, and none of them jointly addresses our two stated goals.

In this paper, we focus specifically on a new process abstraction we call the 'many-core' process (MCP) and discuss how we manage resources given this abstraction. Central to our design is the notion that protection domains are not necessarily coupled with resource management. Furthermore, resources are partitioned as a means of provisioning (as opposed to allocation), giving the system more freedom to utilize these resources when not in use. These abstractions make it possible to provide performance isolation and scalability without the overhead of using a virtual machine style interface.

Given this model, it is possible to naturally build a number of applications that are either awkward or impossible to implement using current operating system abstractions. Examples include (1) a class of HPC applications that contain an interactive or real-time element, (2) a parallel browser structured for better security and resource isolation [30], and (3) a more flexible resource management scheme for web applications. In Section 4, we describe each of these examples in more detail and explain how they would be implemented using MCPs and resource partitioning.

*Research supported by Microsoft Award #024263 and Intel Award #024894 and by matching funding from U.C. Discovery (Award #DIG07-102270).

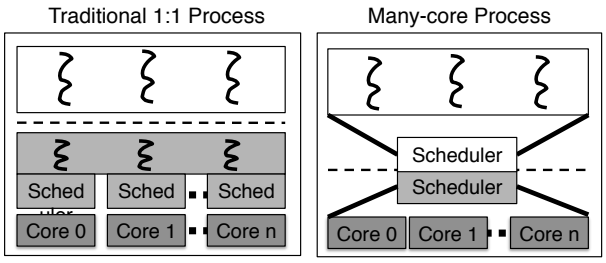

Figure 1: On the left is a traditional 1:1 threading model with each user-space thread mapped onto a kernel thread scheduled by the kernel. On the right is our MCP model with the kernel scheduling physical cores and user-space scheduling the threads that run on them.

2 A New Process Abstraction

Processes in ROS share many similarities with traditional processes. They run in the lowest privilege mode, are the unit of protection and control in the kernel, communicate with each other, and are executing instances of a program.

We extend this model in the following ways. These changes are motivated by what is lacking in current SMP operating systems: direct support for parallel processes and kernel scalability. We call this new abstraction the 'many-core' process (MCP).

- Resources, such as cores and memory, are explicitly granted and revoked. The kernel exposes information about a process's current resource allocation and the system's utilization, and allows the process to make requests based on this information.
- Most cores running a process's address space are gang scheduled; scheduling decisions are made at a coarser granularity than with today's systems.
- There are no kernel tasks or threads underlying each thread of a process, unlike in a 1:1 or M:N threading model. As shown in Figure 1, there is only one kernel object per process, regardless of the number of cores a process runs on.
- Traditionally blocking system calls are asynchronous and non-blocking. A process will not lose a core (or other granted resource) without being informed.

The OS provides the notion of a virtual multiprocessor, similar to scheduler activations [3]. A process can request a block of cores, and the OS guarantees that all cores allocated will run simultaneously [11]. Additionally, the OS will not send unexpected interrupts to cores allocated to a process, except when necessary for preemption by higher priority processes.

There are many advantages to this approach for parallel applications. Classic shared-memory synchronization methods, such as spin-locks, depend on gang scheduling for reasonable performance. By not servicing interrupts on gang-scheduled cores, we eliminate jitter in finely tuned applications' runtimes. Since blocking system calls are asynchronous, I/O concurrency is decoupled from processing concurrency and processes do not need to request extra cores for I/O processing. For memory, processes have knowledge of which virtual pages are resident and page faults are reflected back to the process, allowing it to maintain control over its cores. Finally, the kernel scheduler does not decide which user thread runs at a given time, removing its ability to adversely impact performance through a poor decision. A user-level scheduler, such as Lithe [21], can schedule its contexts on the cores granted by the kernel.

We differ from scheduler activations in that there is only one kernel object per process and that we alert userspace before any of its cores are actually revoked. Processes specify how they wish to be informed about these revocations (as well as any other system events, such as page faults or memory pressure). Whenever the kernel revokes a resource, it will notify the process by running a handler specified by the process. We discuss resources in more detail in Section 3.

These changes enable the kernel to scale well as the number of cores increases. Since we do not have a kernel task underlying every userspace context, we can manage the entire parallel process as one entity. We only need one process descriptor, instead of n. Because we are not scheduling n independent kernel tasks, we can remove per-core run queues and avoid the overhead of load-balancing them. Instead of each core deciding what to run, a subset of the cores will direct the execution of processes throughout the system. This limits any consensus or decision making to a small number of cores, instead of coordinating with n other cores. This model could extend to heterogeneous hardware, where cores with more processing power direct the execution of several smaller cores.

The changes to the process abstraction also result in memory savings. The kernel does not need a stack for every context of every process, which will be important for parallel processes that request large amounts of cores. The core kernel is event based and uses (essentially) continuations to store per-task state, as previously done in Capriccio [28]. This enables many concurrent kernel events, such as blocking I/Os, without requiring large amounts of memory for stack space.

Not all processes need to run in parallel on their cores all the time. A process is gang-scheduled at a coarse granularity; however, applications may want fast response times for short blocks of code. For example, UI events and packet acknowledgements need to execute quickly, and may want to avoid the overhead of context

switching in all of a process's cores. To support this, processes can register handlers that will run independently from the gang, with access to the entire address space. These handlers will run for certain events, but with a very short timeslice and possibly with the gang descheduled. This decouples bulk processing from interactive responses. Additionally, many processes need neither multiple cores nor resource guarantees, and the MCP abstraction has a single-core state to handle this. A more detailed explanation of the MCP model is beyond the scope of this paper.

3 Managing Resources

We introduce a general mechanism for partitioning resources for guaranteed use by a process[1]. A resource is defined as anything sharable in the system, including cores, RAM, cache, on-and off-chip memory bandwidth, access to I/O devices, etc[2]. These resources can be partitioned in both space and time, allowing processes to declare their resource needs in both dimensions [16]. For example, a process might indicate that it needs exclusive access to 25 cores 50% of the time, or that it requires 75% of the on-chip memory bandwidth 25% of the time. For the purposes of this paper, we focus our attention solely on the spatial dimension of partitioning.

In our model, resource parititons serve to provision a set of resources to a group of processes rather than allocate them to a particular process. When creating a resource parition, the system does not actually hand out resources to the requesting process. Instead, it provides a guarantee that the resources contained in that partition will be made available to the process whenever it puts in a request for them. In the mean time, the system can allocate these resources to other processes, under the assumption that they may be revoked at any time. Of course, processes can always request more resources than have been provisioned for them, but there is no guarantee that they will be able to hold onto those resources if the system becomes over-provisioned. Treating resource partitions in this way leads to better utilization of system resources and reduces the hard problem of deciding when to revoke a resource from a process to the simpler problem of deciding which processes are allowed to create a resource partition in the first place (i.e. admission control). Scheduling policy decisions such as which process to revoke a resource from, or how to discourage applications from hoarding resources are beyond the scope of this paper.

[1]Although we currently use processes as the entity to which we allocate resources, any reasonable resource principle could be used [5].

[2]We implicitly assume that a mechanism exists to control the partitioning of resources. Some of these mechanisms may already be available (e.g. cores, RAM, caches via page coloring), while others may only be available in future hardware (e.g. on-chip memory bandwidth).

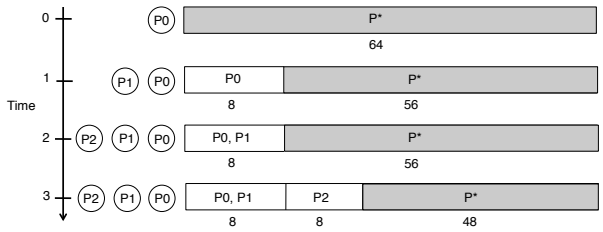

Figure 2: Example partitioning of 64 cores to a set of processes over time. At each time step some number of processes exist in the system (as indicated by the circles on the left), and some number of partitions are set up to contain a list of processes that can pull from them (as indicated by the bars on the right). P* is shorthand for the list of all processes that currently exist in the system.

Using our model, resource partitions may be created in one of two ways: (1) programmatically by some process in the process hierarchy, or (2) by a system administrator who wants fine-grained control over how resources are partitioned throughout the system. Since each partition represents a provision of resources (rather than an allocation), an entire list of processes can be associated with each partition rather than just a single process. This list denotes the set of processes that can pull resources from that partition when making a request. Each resource is partitioned separately, and the ability to associate a process with a given partition is unconstrained, barring any explicit security policy enforced by the system. By associating multiple processes with each provision of resources, we essentially decouple the notion of a protection domain from a resource partition. As we show in our example web application in Section 4, this decoupling helps simplify application design and improve performance.

As a simple example, consider Figure 2, which shows how one might spatially partition a set of 64 cores among 3 competing processes. At time 0, there is only one process in the system, P0, and a single partition containing all cores. At time 1, a second process, P1, has been created and an 8 core partition has been set up from which only P0 can pull. At time 2, P1 has been added to the list of processes associated with the 8 core partition, and a third process P2 has appeared. Finally, at time 3, a second 8 core partition has been created with P2 being the only process which can pull resources from it. Given this example, if all cores were currently allocated to process P2 at the end of Time 3, and a request for 8 cores came in from process P0, those cores would be immediately revoked from P2 and given to P0. No matter how many requests came in from other processes, however, P2 would still be able to hold onto at least 8 cores, no matter how many others it had to give up.

In addition to partitioning, allocating, and revoking resources, the system is also responsible for billing processes for the resources they have been allocated. To avoid being unnecessarily billed, processes are encouraged to voluntarily yield resources back to the system when they are no longer in use. So long as their partitions do not change, handing them back to the system should not adversely affect their performance. Moreover, giving them back may actually provide opportunities for increased energy efficiency [14]. Although not currently implemented, we plan to support a scheme similar to the one used by Resource Containers [5].

4 Examples

Here we present several parallel applications that will benefit greatly from the efficiency of our process model and the flexibility of our resource management scheme.

Music Application: First we discuss an interactive HPC application that relies on real-time sensing, synthesis, and playback of a musical instrument. This application is highly interrupt driven, requires a large amount of resources, and only needs access to its resources for very short durations. However, it requires low-latency access to its resources once it requests them. Using an MCP, the music application can declare its resource needs by provisioning their use at startup and only request them whenever an interrupt comes in. While it holds its resources it has exclusive access to them. Whenever the application becomes idle, the system is able to allocate its resources to other processes running in the system. In general, real-time applications with unpredictable, bursty loads can benefit from this model.

Parallel Browser: Browsers are emerging as *the* dominant application platform for everyday client computing. One of the challenges that current browsers face is the ability to provide quality of service to ensure application performance and prevent resource exhaustion. Moreover, they typically rely on trust models that require not only performance isolation, but also sandboxing and other security features. Recently, Meyerovich et al. have been investigating ways of parallelizing web browsers for use on many-core architectures [13]. Using MCPs and the ability to provision resources, the operating system can provide quality of service for different components of the application as well as independently isolate them for security reasons. Additionally, by using kernel-managed processes to run each component of the browser, misbehaving plugins can be uniquely identified and destroyed by an external process (e.g. the shell).

Web Applications: Although we did not specifically design our system to support web applications, its flexible resource management policy can help solve a variety of issues related to their usage. Specifically, the ability to associate multiple untrusting processes with a single resource partition offers several useful features. For example, web applications relying on a database server can run that server in a separate process while still allowing it to share its resource partition. This ensures that the server always has some resources available to service its requests. In a more complex scenario, two web applications might be sharing access to a single database server with one of them constantly overloading the server with requests. These requests might block the other application from ever making progress. In this situation, the blocked application could temporarily allow the server to allocate resources from its own resource partition to service its requests.

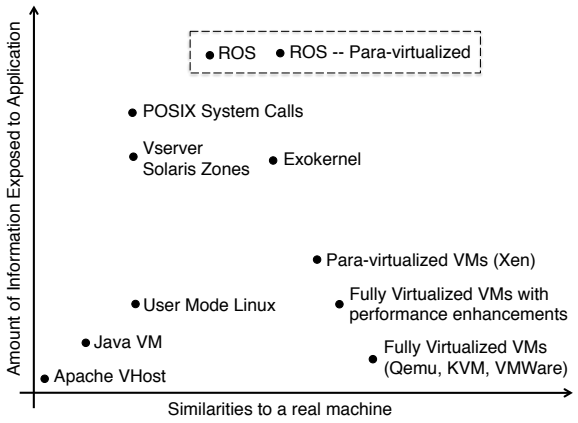

Figure 3: Summary of System Interfaces For Resource Multiplexing

5 Related Work

Many previous solutions have been proposed to multiplex and partition system resources. We categorize them based on the interface they present and the amount of information they expose to the client of the interface. The name of the client depends on the interface (e.g. a process for a system call interface or a guest OS for a virtual machine). Information exposure includes details of the underlying system and actual resource usage, such as resident memory pages and CPU utilization, both for a client and for other clients in the system. Parallel applications are sensitive to the underlying state of the system and require this information exposure. Figure 3 summarizes these approaches along two axes.

Some solutions reveal very little about the state of the overall system to applications. Apache VHost [1] and language-based virtual machines [2] abstract away the underlying hardware completely. Hardware-based virtual machines (including paravirtualized machines) [6,9]

have an interface much closer to the hardware. In fact, it is often their design goal to avoid revealing knowledge about the existence of other applications on the system. Because of this property, data centers use hardware virtual machines such as VMware and Xen to run different applications in different VMs to obtain better security and performance isolation. Typically, each guest OS is assigned a share of system resources. The applications running in a single guest OS instance can only use the resources assigned to it. This unnecessarily binds the protection domain to resource management. Additionally, guest OSs have very little information about overall system utilization or even their own resources. Some VMs with performance enhancements provide mechanisms such as the balloon driver [29], which indicates memory pressure to the guest.

Exokernel [10] allows the application writers to specialize a lib-os to suit the application's needs. We are heavily influenced by its principle of exposing information about the underlying system to the application writers to allow them to make the best decisions. However, this information is limited to a specific lib-OS; a lib-OS is unaware of the overall resource utilization of the system. Compared to this previous work, ROS strives to maintain an interface closer to a traditional operating system. Our system interface exposes information about system utilization and an application's resources, and it allows applications to make requests based on that information.

Systems such as VServer [24] and Solaris Zones [22] provide resource partitioning within an operating system. Their information exposure is similar to a traditional POSIX system, such as the POSIX memory API, procfs, and top. However, they fully isolate other aspects of the operating system such as information about resource usage and namespace management. This is appropriate for the purpose of server consolidation, but for a single user client OS, we consider a shared namespace and configuration to be more useful. A unified namespace simplifies a desktop environment with applications that may interact with each other. Zones provides physical resource partitioning, but it is static, hides all knowledge of overall system resources, and is unable to utilize idle resources for background tasks.

Our model for resource partitioning is grounded in a large body of prior work, mostly generated by research on multi-core operating systems and the real-time community [12, 18, 19, 32]. In particular, our model provides an abstraction similar to that of a *Software Performance Unit* (SPU) [26], with a few distinctions. SPUs rely on processes to actively share unused resources with background tasks, instead of allowing the system to manage these resources. We do not grant a process access to the resources contained in its resource partitions, until the process explicitly requests them. Instead, resource partitions serve as a provision of resources similar to the notion of a 'reserve' [17]. However, they are designed for predictable applications that have a predetermined schedule.

The notion that protection domains should be decoupled from resource management is similar to resource containers [5]. Resource containers allow accurate accounting of resource utilization, independent of the execution context. Our resource provisioning schemem is complementary to their approach.

Recently, a number of other operating systems have been proposed for many-core platforms. Corey [8] is an exo-kernel based on kernel scalability. The Multikernel [7] focuses on scalability in a NUMA and possibly heterogeneous environment, where they scale by distributing state and running message passing algorithms to achieve consensus. Helios [20] focuses on seemlessly running on heterogeneous hardware. We do not explicitly focus on NUMA or heterogenity, but there is nothing that prevents our techniques from working in that environment. None of these share our focus of directly supporting parallel applications.

6 Conclusion

We propose a new OS for many-core with direct support for parallel applications and a scalable kernel. Our design is based on the notion of a 'many-core' process abstraction and the decoupling of protection domains from resource partitioning. Furthermore, we provide a resource management scheme based on resource provisioning, which enables system-wide, efficient accounting and utiliziation of resources. We provide a number of example applications which can benefit directly from our new design, and believe that many more will emerge as we continue to iterate on our design. One application scenario that seems promising is the introduction of spot pricing from Amazon EC2 [27]. Under this model, customers have the ability to bid money for access to spare virtual machine resources, but they must be prepared to be evicted at any time if the system becomes overprovisioned. Our operating system will be able to support virtual machines as clients, and our model of resource management seems like a perfect fit for differentiated services.

7 Acknowledgements

We would like to thank all members of the Berkeley Par Lab OS group and several people at Lawrence Berkeley National Labs for their valuable contributions to the ideas in this paper.

References

[1] Apache VHost Documentation. http://httpd.apache.org/docs/2.0/vhosts/.

[2] Java Virtual Machine Specification. http://java.sun.com/docs/books/jvms/.

[3] T. E. Anderson et al. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, 1991.

[4] K. Asanović, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. elson Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10), 10/2009 2009.

[5] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of the ACM Symp. on Operating Systems Design and Implementation (OSDI)*, 1999.

[6] P. Barham et al. Xen and the art of virtualization. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, 2003.

[7] 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, New York, NY, USA, 2009. ACM.

[8] S. Boyd-Wickizer et al. Corey: an operating system for many cores. In *Proc. of the ACM Symp. on Operating Systems Design and Implementation (OSDI)*, 2008.

[9] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, 1997.

[10] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, 1995.

[11] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[12] A. Gupta and D. Ferrari. Resource partitioning for real-time communication. *IEEE/ACM Trans. Netw.*, 3(5):501–508, 1995.

[13] C. G. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodik. Parallelizing the Web Browser. In *HotPar '09: Proceedings of the Workshop on Hot Topics in Parallelism*. USENIX, March 2009.

[14] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis. Integrating concurrency control and energy management in device drivers. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 251–264, New York, NY, USA, 2007. ACM.

[15] C. Lameter. Extreme high performance computing or why microkernels suck. In *In Proceedings of the Ottawa Linux Symposium*, June 2007.

[16] R. Liu, K. Klues, S. Bird, S. Hofmeyr†, K. Asanović, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client os. In *HotPar '09: Proc. 1st Workshop on Hot Topics in Parallelism*, March 2009.

[17] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. In *In Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, 1994.

[18] A. K. Mok, X. A. Feng, and D. Chen. Resource partition for real-time systems. In *RTAS '01: Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*, page 75, Washington, DC, USA, 2001. IEEE Computer Society.

[19] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore resource management. *Micro, IEEE*, 28(3):6–16, May-June 2008.

[20] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234, New York, NY, USA, 2009. ACM.

[21] H. Pan, B. Hindman, and K. Asanović. Lithe: Enabling efficient composition of parallel libraries. In *Proc. of HotPar*, 2009.

[22] D. Price and A. Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *LISA '04: Proceedings of the 18th USENIX conference on System administration*, pages 241–254, Berkeley, CA, USA, 2004. USENIX Association.

[23] T. Roscoe, K. Elphinstone, and G. Heiser. Hype and virtue. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[24] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, 2007.

[25] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, D. Patterson, and K. Asanovic. RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors. In *4th Workshop on Architectural Research Prototyping (WARP-2009), at 36th International Symposium on Computer Architecture (ISCA-36)*, June 2009.

[26] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 181–192, New York, NY, USA, 1998. ACM.

[27] W. Vogels. All things distributed: Werner vogels' weblog on building scalable and robust distributed systems., December 2009. `http://www.allthingsdistributed.com/2009/12/amazon_ec2_spot_instances.html`.

[28] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *SOSP '03*, 2003.

[29] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[30] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal os construction of the gazelle web browser. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, August 2009.

[31] 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.

[32] D. Wright. Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor. In *Proceedings of the Linux Clusters: The HPC Revolution conference*, Champaign - Urbana, IL, June 2001.