Resource Management in the Tessellation Manycore OS *

Juan A. Colmenares, Sarah Bird, Henry Cook, Paul Pearce, David Zhu, John Shalf†, Steven Hofmeyr†, Krste Asanović, and John Kubiatowicz

The Parallel Computing Laboratory (UC Berkeley) and †Lawrence Berkeley National Laboratory
{juancol, slbird, hcook, pearce, yuzhu}@eecs.berkeley.edu, {jshalf, shofmeyr}@lbl.gov,
{krste, kubitron}@eecs.berkeley.edu

Abstract

Tessellation is a manycore OS targeted at the resource management challenges of emerging client devices, including the need for real-time and QoS guarantees. It is predicated on two central ideas: *Space-Time Partitioning* (STP) and *Two-Level Scheduling*. STP provides performance isolation and strong partitioning of resources among interacting software components, called *Cells*. Two-Level Scheduling separates *global* decisions about the allocation of resources *to* Cells from *application-specific* scheduling of resources *within* Cells. We describe Tessellation's Cell model and its resource allocation architecture. We present results from an early prototype running on two different platforms including one with memory-bandwidth partitioning hardware.

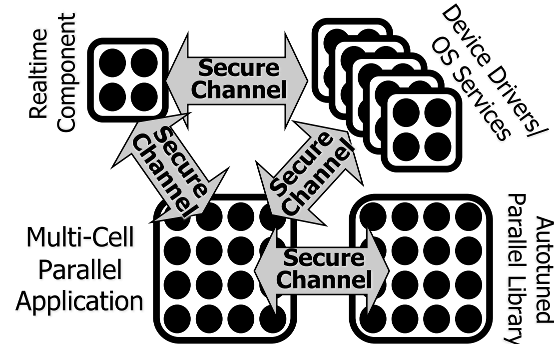

Figure 1: Decomposing an application into a set of communicating components and services running with QoS guarantees within Cells. Tessellation provides Cells that host device drivers and OS services.

1 Introduction

The trend toward *manycore* systems (with 64 or more cores) presents serious challenges for client devices. Users will expect better performance from applications as the number of cores increase; this expectation will be challenging to meet since it requires parallelizing client applications (which are often not very scalable) and exploiting parallelism that is likely to be fragile and easily disturbed by interference. Further, tomorrow's applications will consist of variety of components – each of which presents complex and differing resource requirements. In addition to best-effort computation, users have an increasing appetite for responsive user interfaces and high-quality multimedia (*e.g.*, multi-party videoconferencing, multi-player gaming, and music composition) with stringent real-time requirements; such needs are not well supported by today's commodity operating systems.

We believe that the advent of manycore is an opportunity to fundamentally restructure operating systems to support a simultaneous mix of interactive, real-time, and high-throughput parallel applications. Our hypothesis is that a much wider variety of performance goals can be met by structuring the operating system around resource distribution, performance isolation, and QoS guarantees; such structuring is natural in a manycore environment.

This paper investigates the combination of two complementary ideas embodied in our new OS, called *Tessellation*: *Space-Time Partitioning* and *Two-Level Scheduling*. Space-Time Partitioning (STP) [21], exploits novel software layering and hardware mechanisms (when available) to support a model of computation in which applications are divided into performance-isolated, gang-scheduled *Cells* communicating through secure channels; see Figure 1.

Two-Level Scheduling separates global decisions about the allocation of resources *to* Cells from application-specific scheduling of resources *within* Cells. The resource distribution process (first level) is one of the novel elements of our approach and is discussed in detail in Section 3. Once resources have been assigned to Cells, STP guarantees that user-level schedulers within Cells (second level) may utilize resources as they wish – without interference from other Cells or from the OS. It is the separation of resource distribution from usage that we believe makes Two-Level Scheduling more scalable than other approaches and better able to meet the demands of parallel client applications.

2 Overview of Tessellation

Tessellation is a manycore OS focused on resource guarantees. Here we summarize key aspects of Tessellation.

2.1 Space-Time Partitioning

A *spatial partition* (or *partition*) is a performance-isolated unit of resources maintained through a combination of software and hardware mechanisms. Managed resources include gang-scheduled hardware thread contexts, guaranteed fractions of shared resources (*e.g.*, cache or memory bandwidth), access to OS services, and fractions of the energy budget. Tessellation divides the hardware into a set of simultaneously-resident partitions

*Research supported by Microsoft Award #024263, Intel Award #024894, matching U.C. Discovery funding (Award #DIG07-102270), and DOE ASCR FastOS Grant #DE-FG02-08ER25849.

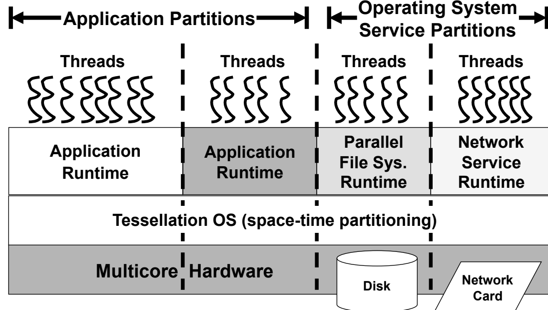

Figure 2: *Space-Time Partitioning* in Tessellation: a snapshot in time with four spatial partitions.

as shown in Figure 2. Partitioning varies with the needs of the OS and applications – hence the "time" component of the term Space-Time Partitioning (STP).

Support for STP consists of a combination of hardware and software mechanisms. The "Partition Mechanism Layer" of Tessellation, which enforces partition boundaries, has some similarities to a hypervisor [1, 4, 16] but with a crucial difference: its sole task is to provide performance-isolated, QoS-guaranteed containers for applications and OS services. Although Tessellation runs on existing multicore systems, it can also exploit hardware enhancements when available.

2.2 The Cell Model

Tessellation exports STP to applications and OS services through an abstraction called a *Cell*. A Cell is a container for parallel software components providing guaranteed access to resources, *i.e.*, the performance and behavior of an isolated machine. Resources are guaranteed as space-time quantities, such as "4 processors for 10% of the time" or "2 GB/sec of bandwidth". Although Cells may be time-multiplexed, hardware thread contexts and resources are gang-scheduled such that Cells are unaware of this multiplexing. In other words, unexpected virtualization of physical resources does not occur.

Resources allocated to a Cell are owned by that Cell until explicitly revoked[1]. Once the Cell is mapped, a user-level scheduler is responsible for scheduling hardware contexts and other resources. There is no paging of physical memory unless a paging library is linked into the user-level runtime. Further, each Cell's runtime has control over the delivery of events such as inter-cell messages, timer interrupts, exceptions, and faults.

Inter-Cell Communication: Inter-cell communication occurs through *channels*. A channel provides performance and security isolation between Cells. The setup and tear-down of a channel is privileged and strictly controlled by the OS. Once constructed, a channel provides fast asynchronous communication at user-level.

[1] Tessellation notifies the user-level scheduler of revocations, giving it a chance to adjust accordingly.

Utilizing Cells for OS Services: Cells provide a convenient abstraction for building OS services such as device drivers, network interfaces, and file systems. Tessellation adopts a philosophy similar to that of microkernels [8]. Unlike traditional microkernels, however, multiple components can be mapped to the hardware simultaneously – allowing rapid inter-domain communication. Further, each interacting component is explicitly parallel and performance-isolated from other components.

Partitioning OS functionality into a set of interacting Cells provides predictable and reliable behavior due to limited interaction with the rest of the system. QoS guarantees on shared services can be enforced by restricting channel communication. Alternatively, the capacity of overloaded services can be increased by resizing Cells.

2.3 Two-level Scheduling in Tessellation

Tessellation separates global decisions about resource allocation from local decisions about resource usage. The result is a good match to the abundant resources present in a manycore system: The resource allocation process can focus on the impact of resource *quantities* on Cell execution – leaving the fine details about how to utilize these resources to application-specific schedulers.

Distributing Resources to Cells: The resource allocator, described in Section 3, distributes partitionable resources among Cells and exercises the option to reserve or deactivate resources to guarantee future responsiveness or to optimize energy consumption. Tessellation changes resource distribution infrequently to amortize the cost of the decision-making process and to minimize interference with application-level scheduling.

Scheduling Within a Cell: The Cell-level scheduler runs at user-level and manages all resources within the Cell. Performance isolation between Cells guarantees that applications can get predictable and repeatable behavior, simplifying performance optimization and real-time scheduling. Central to Tessellation's approach are runtime frameworks, such as Lithe [23], that produce composable, application-specific schedulers. Via Lithe, Tessellation supports a variety of parallel programming models in a uniform and composable way.

3 Resource-Allocation Architecture

Tesselation strikes a balance between maximizing resource utilization to achieve performance and selectively idling resources to provide QoS guarantees. Decision-making logic is packaged into a *Policy Service* that distributes resources to Cells by combining system-wide goals, resource constraints, and performance targets with current performance measurements (see Figure 3). The results are passed as a *Space-Time Resource Graph* (STRG) to the Tessellation kernel for QoS enforcement.

3.1 Space-Time Resource Graph

The Space-Time Resource Graph (STRG) is central to our approach. It contains the current distribution of system resources to Cells and is updated by the Policy Service to reflect the changing needs of the system. As it changes, copies of the STRG are passed to the Tessellation kernel for validation and implementation.

Each leaf of the STRG represents an admitted Cell and contains the current resource assignments for that Cell, including guaranteed fractions of caches and memory bandwidth, the Cell activation policy (see Section 3.3), and a list of QoS guarantees for system services. It also indicates the Cell's desire to receive excess resources.

Interior nodes of the STRG group Cells into *resource groups*, providing a mechanism for assigning resources to related Cells. For instance, resource groups provide an obvious mechanism for distributing resources to multiple Cells that are part of a single application or service domain[2]. As another example, Tessellation's *pre-allocation* mechanism allows resources to be reserved for future use by a particular Cell while exported to other Cells in a resource group as temporary (revocable) excess resources.

3.2 Policy Service

The Policy Service admits new Cells into the system, monitors existing Cells, and adapts resources in response to changing conditions. In this section, we describe the main components of the Policy Service.

Admission Control: Without admission control it would be impossible to provide QoS guarantees. As a replacement for traditional `fork()` or `spawn()` operations, Tessellation supports Cell creation and destruction through an interface with the *Admission Control* module. Admission Control refuses to admit new Cells whose resource requirements are incompatible with existing obligations to other Cells. It also rejects resizing requests that are incompatible with such obligations.

As shown in Figure 3, the Admission Control module can handle simple requests by directly modifying the STRG. Simple requests include ones that merely consume excess resources and do not involve revocation of resources. More complex admission requests are forwarded to the *Resource Allocation and Adaptation* (RAAM) module (described below). In principle, the RAAM module can consider a wide variety of rearrangements of resources to satisfy requests – constrained only by policies, QoS requirements, and user preferences. Should the RAAM module decide that a request cannot be satisfied, however, the Admission Control module will forward the rejection back to the original requester, at which point the user may need to become involved.

[2]Resource groups can be used to provide several of the same benefits as Resource Containers [3].

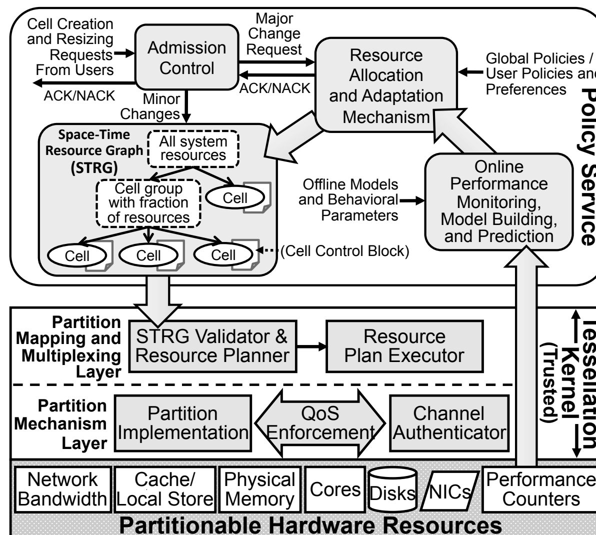

Figure 3: The *Policy Service* updates a system-wide *Space-Time Resource Graph* (STRG) based on QoS requirements, policies, and performance measurements. STRGs passed to the kernel are validated and used to distribute resources to Cells.

Resource Allocation and Adaptation: The heart of the Policy Service is the *Resource Allocation and Adaptation Mechanism* (RAAM). As seen by the large arrows in Figure 3, the RAAM is part of a system-wide adaptive loop. It enables software components within Cells to achieve their performance goals while optimizing resource distribution across the system according to global policies. RAAM allows each Cell to register one or more resource-allocation policies, thereby injecting application requirements and user preferences into the decision-making process.[3] RAAM updates the STRG to reflect changes in resource allocation.

RAAM consults a variety of situational information about the state of the system, including performance measurements and models of system performance as a function of resources. The simplest of this information involves updates from performance counters to reflect message volume across channels or energy usage. Another source of information is periodic *performance reports* from the application [12, 14]. A performance report contains Cell-specific performance metrics (*e.g.*, the amount of progress made on application-specific deadlines). These reports can be combined with application-supplied information about what a meaningful unit of work would be and how often it must be completed (*i.e.*, frames/second) to adjust allocations.

We are investigating a variety of techniques for resource adaptation. Although one *could* blindly increase the resources given to a Cell until performance goals are met, such a strategy is unlikely to work for complex con-

[3]We intend to investigate use of a new declarative language for describing resource-allocation policies.

figurations of Cells nor is it capable of incorporating subtleties in the level of importance of some Cells over others. Instead, we believe that two components are necessary: first, accurate models of the performance of Cells (or the change in performance) as a function of resources and second, a framework in which to drive the juggling of resources among Cells. One framework that we are investigating is a form of convex optimization over resources that attempts to minimize an "urgency" derived from the degree to which Cells miss their deadlines [26].

Modeling Application Behavior: The process of determining how client application performance varies with low-level resources is often labor-intensive and error-prone. Except in very special circumstances, programmers are unlikely to know "exactly" how low-level resources affect their performance goals. For example, the programmer may want a given *frame rate* but have no idea how much memory bandwidth is required to meet that rate This conundrum is a form of "impedance mismatch" between the units of hardware resources and the programmer-specified QoS requirements [17].

It is therefore desirable to figure out how to *automatically* construct predictive models of application performance. Tessellation's performance isolation should permit us to build highly accurate models. However, we must decide when and how these models will be built.

One solution is to profile the applications in advance. This solution could be viable for application distribution models like the iTunes Application Store or the Android Market. The application distributor can profile the applications for the limited platforms it supports and provide predictive models when the application is downloaded.

Another option leverages the Cloud by requesting every user to record performance, resource, and platform statistics. This approach is currently being used by Microsoft Research to generate performance models of Microsoft applications for developers [13].

A more general solution is to have the operating system profile the applications online and use the locally collected information to make resource decisions. While the operating system could potentially run the applications over a set of configurations when the application is first installed to build the models, it may be more practical to simply start with a generic model which is refined over time with on-line training methods.

The source of the data used to create the models is somewhat orthogonal to the type of models used. Adopting a particular model type in turn influences the specific method used for searching over all models to find the best resource distribution across all Cells. We are currently interested in two types of models. Models of the first type are based on a discrete, sparse set of predetermined operational points (*e.g.*, [12]). Models of the second type use continuous functions to capture performance trade-offs across a large number of allocations (*e.g.*, [5, 9]). The effectiveness of linear and quadratic models has been shown in [7].

3.3 Partition Mapping and Multiplexing

The *Partition Mapping and Multiplexing Layer* (or *Mapping Layer*), translates the resource specifications of the Policy Service (expressed in the STRG) into an ordered sequence of spatial partitions for the underlying *Partition Mechanism Layer* (mentioned in Section 2). The Mapping Layer makes no policy decisions, but rather implements the policy decisions given by the Policy Service.

The Mapping Layer comprises two main components: the *Planner* and the *Plan Executor*. When the Planner receives a new STRG from the Policy Service, it first validates that this STRG does not violate basic security or QoS requirements[4], then generates a future plan for distributing resources to Cells. The Planner invokes an operation similar to bin-packing to assign Cells and resources to future partition time-slices.

The Plan Executor implements the resulting resource plan. It can modify the plan being executed in predefined ways to accommodate more dynamic resource-allocation and time-multiplexing actions (*e.g.*, activation of a Cell upon the arrival of an event or redistribution of excess resources among Cells).

In implementing the STRG, the Mapping Layer implements a variety of *Cell activation policies*. Examples include the *Pinned Policy* (Cell given dedicated access to cores), the *Time-Triggered Policy* (Cell active during predetermined time-windows for real-time predictability), and the *Time-fraction Policy* (Cell active for a specified fraction of the time). Most Cell activation policies are *non-preemptive*: once a Cell is activated it is not suspended until its time-slice expires. The one exception is that Cells can be given best-effort resources that may be preempted by Cells with higher priority.

4 Experimental Evaluation

In this section, we examine the potential for performance isolation in the Tessellation prototype. The prototype was derived from an early version of the ROS kernel [19], supplemented with support for cell time-multiplexing and second-level preemptive scheduling. This prototype contains 22,000+ lines of code and runs on both Intel x86 platforms and RAMP Gold [27, 28]. RAMP Gold is an FPGA-based simulator that models up to 64 in-order 1-GHz SPARC V8 cores, a shared memory hierarchy, and hardware partitioning mechanisms. The Intel system used in our experiments is equipped with dual 2.67-GHz Xeon X5550 quad-core processors.

[4]We are exploring how to remove as much of the Policy Service from the trusted computing base as possible.

	2 Cores	15 Cores	63 Cores
Intel activate	1.57 μs	8.26 μs	N/A
RAMP activate	0.69 μs	1.88 μs	5.37 μs
Intel suspend	1.58 μs	17.59 μs	N/A
RAMP suspend	1.19 μs	5.91 μs	34.10 μs

Table 1: Mean activation and suspension latencies for cells of varying size. Here, core 0 was dedicated to Cell management.

Cell Activation and Suspension: Table 1 summarizes the overhead of activating and suspending a Cell with varying core counts on both RAMP Gold and our Intel system. These numbers are preliminary. The overhead is small relative to the time scale of Cell time-multiplexing (*e.g.*, 100 ms), but is still larger than we would like.

Performance Isolation: Our 64-core RAMP Gold platform simulates a mechanism that can dedicate fractions of off-chip memory bandwidth to Cells [20]. We illustrate Tessellation's use of this mechanism by creating three Cells as follows: $Cell_1$ is given 32 cores and 50% of memory bandwidth (*i.e.*, 6.4GB/s); $Cell_2$ 16 and 25%; and $Cell_3$ 15 and 25%. $Cell_1$ contains the PARSEC *streamcluster* [6], selected for its significant memory capacity and bandwidth requirements. Other PARSEC benchmarks run in the remaining Cells.

We first activate $Cell_1$ by itself. Next, we activate all Cells and run the benchmarks concurrently both with and without memory bandwidth partitioning. $Cell_1$ takes 5.70M, 6.12M and 11.59M core-cycles on average to complete, for the three experiments. The respective standard deviations are 0.30M, 0.95M and 1.17M. These results show that Tessellation can provide significant performance isolation.

Spatial Partitioning: Here we evaluate the potential of spatial partitioning using Tessellation on the RAMP Gold 64-core machine. We take pairs of PARSEC applications, placing each application in a Cell. Cores are assigned in groups of 8, page colors in sets of 16, and memory bandwidth in units of 3.4GB/s (a combinatorial total of 54 valid allocations). We then evaluate *all* possible spatial allocations for the two Cells. We also evaluate the case in which one Cell is assigned the entire machine and run to completion followed by the other Cell; this is a favorable time-multiplexing scenario as there is no overhead from repeated context switches.

For some pairs, time-multiplexing is better than any possible spatial-partitioning. However, for many pairs, the optimal spatial partition is substantially better thanks to disjoint resource requirements, reduced interference between pairs, or imperfect application scaling. Figure 4 illustrates the performance of several pairs. It shows that spatial partitioning can provide significant performance benefits. However, naïve spatial divisions are likely to be detrimental to performance, meaning that the Policy Service must be judicious in assigning resources to Cells.

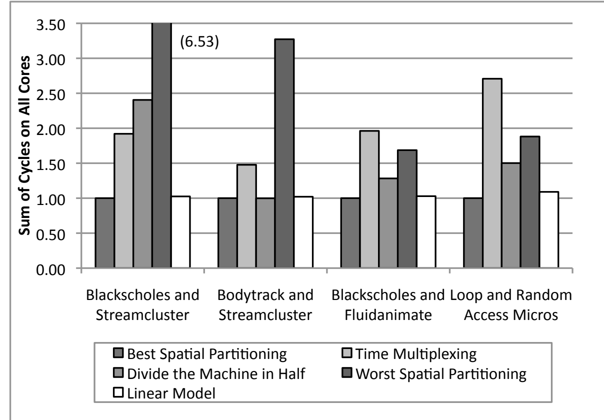

Figure 4: Performance of spatial partitioning compared with a favorable time-multiplexing. Performance is in cycles (lower is better) and results are normalized to the best spatial partition.

We also found that a *simple* linear approximation model, derived from only 10 sample points, allowed us to make resource-allocation decisions within 10% of optimal every time. These results encourage further research on approaches to resource allocation for Cells and tradeoffs between complexity, overhead and performance.

5 Related Work

Tessellation is influenced by virtual machines, exokernels, and multiprocessor runtime systems [1, 2, 4, 10, 15, 16, 18, 24]. Other recent manycore operating systems projects, such as Corey [11], Barrelfish [25], and fos [29], share some structural aspects such as distributed OS services. This body of work mainly focus on improving OS scalability and, contrary to Tessellation, does not attempt to also provide QoS guarantees.

Nesbit et al. [22] introduce Virtual Private Machines (VPM), another framework for resource allocation and management in multicore systems. The concepts of VPM and Cell are similar, but the VPM framework does not include an equivalent communication mechanism to our inter-cell channel.

6 Conclusion

We presented Tessellation, a new manycore OS for client devices that provides real-time and QoS guarantees. Tessellation is predicated on two central ideas, namely *Space-Time Partitioning* (STP) and *Two-Level Scheduling*. In this paper, we discussed Tessellation's Cell model, explored its resource-allocation architecture, and examined results from an early prototype.

7 Acknowledgment

We thank Kevin Klues, Barret Rhoden, Andrew Waterman, and Eric Brewer for making ROS source code available, their collaboration, and many lively discussions.

References

[1] VMWare ESX. http://www.vmware.com/products/vi/esx/.

[2] K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson. Adaptive scheduling with parallelism feedback. In *Proc. of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, pages 100–109, March 2006.

[3] 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.

[4] P. Barham et al. Xen and the art of virtualization. In *Proc. of the 19th ACM Sympsium on Operating Systems Principles (SOSP'03)*, pages 164–177, October 2003.

[5] M. N. Bennani and D. A. Menasce. Resource allocation for autonomic data centers using analytic performance models. In *Proc. 2nd Int'l Conference on Autonomic Computing (ICAC 2005)*, June 2005.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. of the 17th Int'l Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[7] S. Bird. Software Knows Best: A Case for Hardware Transparency and Measurability. Master's thesis, EECS Department, University of California, Berkeley, May 2010.

[8] D. L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23:35–43, 1990.

[9] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Automatic exploration of datacenter performance regimes. In *Proc. of the 1st Workshop on Automated Control for Datacenters and Clouds (ACDC'09)*, June 2009.

[10] T. L. Borden, J. P. Hennesy, et al. Multiple operating systems on one processor complex. *IBM Systems Journal*, 28(1):104–123, January 1989.

[11] S. Boyd-Wickizer et al. Corey: an operating system for many cores. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, December 2008.

[12] J. A. Colmenares, K. Kim, C. Lim, Z. Zhang, and K.-W. Rim. Real-time component-based software architecture for qos-adaptive networked multimedia applications. In *Proc. of the 13th IEEE Int'l Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC'10)*, May 2010.

[13] B. Doebel, P. Nobel, E. Thereska, and A. Zheng. Towards versatile performance models for complex, popular applications. *SIGMETRICS Perform. Eval. Rev.*, 37(4):26–33, 2010.

[14] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. *ACM SIGOPS Operating Systems Review*, 34(5):247–260, 1999.

[15] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, December 1995.

[16] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM SIGOPS Operating Systems Review*, 33(5):154–169, 1999.

[17] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *Proc. of the 40th Annual IEEE/ACM Int'l Symposium on Microarchitecture (MICRO'07)*, pages 343–355, 2007.

[18] J. Jann, L. M. Browning, et al. Dynamic reconfiguration: Basic building blocks for autonomic computing on ibm pseries servers. *IBM Systems Journal*, 42(1), January 2003.

[19] K. Klues, B. Rhoden, Y. Zhu, A. Waterman, and E. Brewer. Processes and resource management in a scalable many-core OS. In *Proc. of the 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar'10)*, June 2010.

[20] J. W. Lee, M. C. Ng, and K. Asanović. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *Proc. of the 35th ACM/IEEE Int'l Symposium on Computer Architecture (ISCA 2008)*, pages 89–100, June 2008.

[21] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *Proc. of the 1st USENIX Conference on Hot Topics in Parallelism (HotPar'09)*, March 2009.

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

[23] H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with Lithe. In *Proc. of the SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI)*, June 2010.

[24] B. Saha et al. Enabling scalability and performance in a large scale CMP environment. In *Proc. of the ACM SIGOPS European Conference on Computer Systems (EuroSys'07)*, March 2007.

[25] A. Schüpbach et al. Embracing diversity in the barrelfish manycore operating system. In *Proc. of the Workshop on Managed Many-Core Systems (MMCS'08)*, June 2008.

[26] B. Smith. Operating system resource management (keynote talk). In *24th IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS 2010)*, April 2010.

[27] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, K. Asanović, and D. Patterson. RAMP Gold: An FPGA-based architecture simulator for multiprocessors. In *Proc. of the 47th Design Automation Conference (DAC 2010)*, June 2010.

[28] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson. A case for FAME: FPGA architecture model execution. In *Proc. of the 37th ACM/IEEE Int'l Symposium on Computer Architecture (ISCA 2010)*, June 2010.

[29] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.