

# Resource Management in the Tessellation Manycore OS \*

J. A. Colmenares, S. Bird, H. Cook, P. Pearce, D. Zhu, J. Shalf<sup>†</sup>, K. Asanović, and J. Kubiatowicz

Parallel Computing Lab, Computer Science Division, University of California at Berkeley

<sup>†</sup>Lawrence Berkeley National Laboratory

{juancol, slbird, hcook, pearce, yuzhu}@eecs.berkeley.edu, jshalf@lbl.gov, {krste, kubitron}@eecs.berkeley.edu

## Abstract

Tessellation is a manycore OS predicated on two central ideas: *Space-Time Partitioning* (STP) and *Two-Level Scheduling*. STP exploits novel hardware and software mechanisms to provide performance isolation and strong partitioning of resources (such as cores or memory bandwidth) among interacting software components, called “Cells”. *Two-Level Scheduling* separates *global* decisions about the allocation and distribution of resources to Cells from *application-specific* scheduling of resources *within* Cells. We describe Tessellation’s Cell model, its resource allocation architecture, and basic policies for resource management. We present results from our prototype running on both an 8-core Nehalem machine and an FPGA-emulation of a 64-core machine with memory-bandwidth partitioning hardware.

## 1 Introduction

Current trends suggest that highly parallel *manycore* systems (with 64 or more cores) will soon be mainstream. Increasing core counts presents serious challenges for applications attempting to harvest parallel resources and for the operating systems supporting them. Unlike servers, which exploit parallelism across independent transactions from multiple users, single-user clients will require parallelized applications to benefit from a manycore platform. Future client devices will run a mix of interactive, real-time, and batch applications simultaneously. In fact, the applications of tomorrow are likely to consist of variety of components – each of which presents complex and differing resource requirements.

We believe that operating systems of the future will require a radically new structure. One basic assumption is that the parallelism exhibited by client applications will be more fragile than that of high-performance computing applications; consequently, it is important not to disturb their execution with unexpected variations in resource availability or unexpected events such as interrupts. Further, real-time and QoS requirements are easier to meet in a stable and noiseless environment.

To this end, this paper investigates the combination of two complementary ideas: *Space-Time Partitioning* and *Two-Level Scheduling* as embodied in our

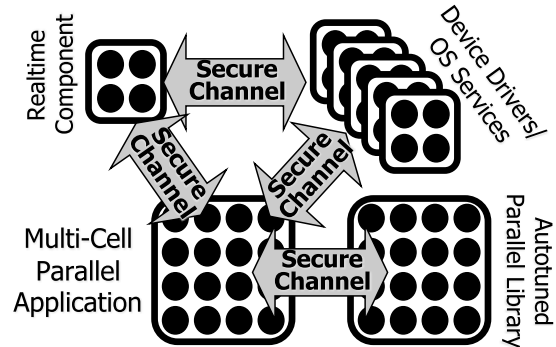


Figure 1: Decomposing an application into a set of communicating components and services running with QoS guarantees within Cells. Tessellation provides Cells that host device drivers and OS services.

new OS, called “Tessellation.” *Space-Time Partitioning* (STP) [16], 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. STP provides guaranteed fractions of resources (*e.g.* processors, memory bandwidth, or system services) to Cells. In addition, STP offers a natural way to incorporate protection, fault-containment, and security policies.

Complementing STP, *Two-Level Scheduling* separates global decisions about the allocation and distribution of resources *to* Cells from application-specific scheduling of resources *within* Cells. The global resource distribution process (*i.e.* the first level) is one of the novel elements of our approach and is discussed in detail later. Once resources have been assigned to Cells, STP guarantees that user-level schedulers within Cells (*i.e.* the 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 based on Space-Time Partitioning (STP) and Two-Level Scheduling. This section provides an overview of STP, the Cell model, and Two-Level Scheduling in Tessellation.

\*Research supported by Microsoft Award #024263 and Intel Award #024894 and by matching funding from U.C. Discovery (Award #DIG07-102270). Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

## 2.1 Space-Time Partitioning

A *spatial partition* (or “partition” for short) 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 (cache slices, memory bandwidth), access to OS services, and fractions of the power or energy budget. At any point in time, Tessellation divides the hardware into a set of simultaneously-resident partitions; over time, partitioning varies with the needs of the OS and applications – hence the “time” component of the term “STP.”

Support for STP in Tessellation consists of a combination of software and hardware mechanisms; see Figure 2. The partition-enforcing “Partition Mechanism Layer” of Tessellation has some similarities to a hypervisor but with a crucial difference: its sole task is to provide performance-isolated, QoS-guaranteed containers for applications and OS services. Decisions about the time evolution of hardware partitions based on global constraints and policies are made by the “first level” of Two-Level Scheduling, mentioned previously. 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. Resources allocated to a Cell are owned by that Cell exclusively until they are explicitly revoked, enabling application-specific optimizations.

Unexpected virtualization of physical resources does not occur within a Cell. Once the Cell is mapped, a Cell-level scheduler (running at user-level) is responsible for mapping threads to hardware contexts. Other resources are also explicitly managed by Cell-level schedulers. For example, there is no paging of physical memory unless a paging library is linked in to the user-level runtime, and each cell’s runtime has control over the delivery of events such as inter-cell messages, timer interrupts, exceptions, and faults. Cells start with a single protection domain; however, they can create additional domains as desired.

One possible objection to this paradigm is that the burden placed on application developers is increased since they are required to explicitly manage resources. However, parallel application programmers are already accustomed to linking their programs against specific user-

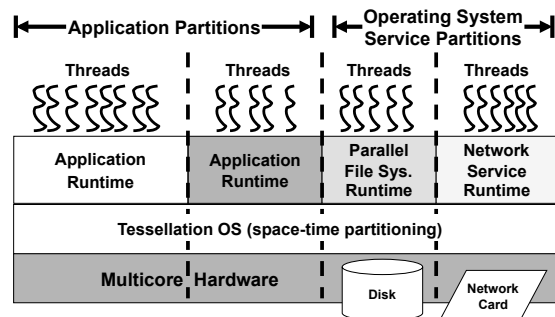


Figure 2: *Space-Time Partitioning* in Tessellation: a snapshot in time with four spatial partitions.

level thread libraries, some of which change system behavior or runtime environment semantics [11].

**Inter-Cell Communication:** *Channels* provide a controlled mechanism for inter-cell communication. A channel has clearly defined semantics and 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, however, a channel provides fast asynchronous communication with user-level operations.

**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 [7]. 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

Centralized scheduling works in a uniprocessor environment where applications have a limited number of threads and expect to be time-multiplexed onto the single CPU. In contrast, Tessellation separates global decisions about the allocation and distribution of resources from local decisions about the usage of resources. The result is more scalable when abundant parallel resources are utilized by interacting parallel components: 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 within Cells.

**Scheduling Resources To Cells:** Tessellation makes changes in resource distribution infrequently to amortize the cost of the global decision making process while further allowing time for application-level scheduling decisions to be effective. The resource allocator, described in detail 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.

**Scheduling Within a Cell:** The Cell-level scheduler runs as a user-level library and manages threads and cores within a Cell. Since a Cell’s runtime has control over the delivery of system events, it can disable events, or poll for specific ones. During resource revocation, Tessellation informs the user-level scheduler, giving it a chance to adjust accordingly.

Performance isolation between cells guarantees that other components of the system cannot interfere with a scheduler’s use of resources. Consequently, applications can get predictable and repeatable performance within a Cell, simplifying performance optimization and real-time scheduling. Central to Tessellation’s approach are runtime frameworks, such as Lithe [18], that enable construction of composable, application-specific schedulers. Via Lithe, Tessellation supports a variety of parallel programming models in a uniform and composable way.

### 3 Resource-Allocation Architecture

Tessellation adopts a layered approach to resource allocation; see Figure 3. The modularity increases overall flexibility, portability and verifiability. The layers are:

1. *Policy Layer:* This layer makes resource-allocation decisions for Cells. It considers system-wide goals, resource constraints, and performance targets for Cells. The Policy Layer exploits a variety of policies for adaptive resource allocation and admission control. It generates the *Space-Time Resource Graph (STRG)*, which contains in-memory representations of the admitted Cells along with descriptions of their allocated system resources.
2. *Mapping and Time-Multiplexing Layer:* This layer distributes the system resources among Cells based on the STRG. It time-multiplexes Cells at a *coarse* granularity – if at all – according to the time parameters stored in the STRG.
3. *Mechanism Layer:* This layer implements spatial partitioning for performance isolation, as described earlier. It utilizes whatever hardware isolation mechanisms are natively provided and *emulates* missing mechanisms when necessary.

In the following subsections we investigate the Policy Layer and the Mapping and Time-Multiplexing Layer in greater detail.

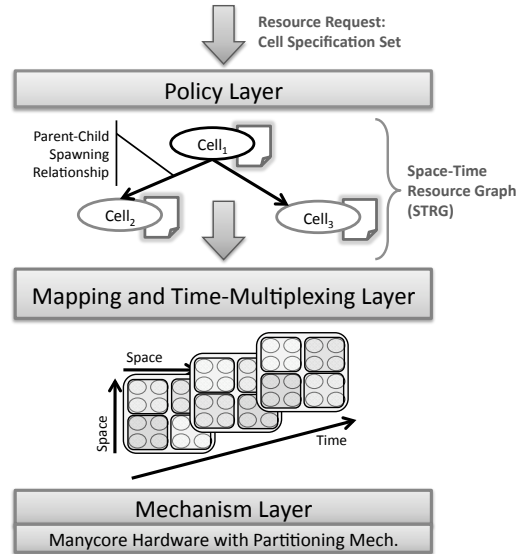


Figure 3: Resource Allocation in Tessellation: the *Policy Layer* encodes resource decisions as a STRG. The graph is realized by the *Mapping/Time-Multiplexing* and the *Mechanism* Layers.

#### 3.1 Policy Layer

The Policy Layer is packaged as a *service* running within a dedicated Cell. It distributes resources to Cells by combining global policies, resource requests, and performance requirements. Further, it will refuse to admit new Cells whose resource requirements are incompatible with existing resource obligations. In addition to incorporating global policies, the Policy Layer allows each Cell to register one or more resource-allocation policies, thereby injecting application requirements and user preferences into the decision-making process<sup>1</sup>.

The Policy Layer generates a *Specification Set* for each Cell, organized in a *Space-Time Resource Graph (STRG)*. The Specification Set of a Cell includes: 1) the required physical resources, 2) the time-multiplexing policy and its parameters, and 3) a list of QoS demands for system services. Specification Sets are generated based on requests made during the creation of the Cell, the current behavior of the Cell, and global policies. Further, the Policy layer will update Specification Sets to adapt to changing circumstances.

**Admission Control:** As a replacement for traditional `fork()` or `spawn()` operations, Tessellation supports Cell creation and destruction through an interface with the Policy Layer. Without true admission control it would be impossible to provide real QoS guarantees. Consequently, the Policy Layer performs admission control whenever a Cell is created by confirming that new resource needs are compatible with existing guarantees.

<sup>1</sup>We intend to investigate use of a new declarative language for describing policies.

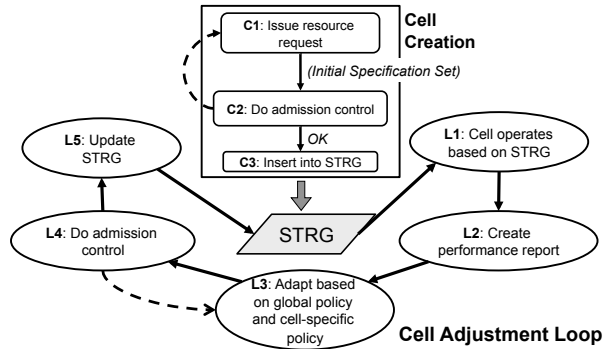


Figure 4: High-level view of *Admission Control* and the *Adaptive Resource-Allocation Loop* in the Policy Layer.

Figure 4 shows the sequence of steps for the initial allocation of resources during the creation of a Cell (shown as states **C1-C3**). When admission of a new Cell is denied, several options are possible. One option is to reflect back to the application or the users. Another is to adapt by taking resources from one or more preexisting Cells; such revocations are acceptable if doing so will not violate the other Cell’s QoS constraints.

**Adaptive Resource Allocation:** Except in very special circumstances, programmers are unlikely to know exactly how low-level resources affect their performance goals. This conundrum is a form of “impedance mismatch” between the units of hardware resources (such as processors or bandwidth) and the programmer-specified QoS requirements. As a simple example, the programmer may want a given *frame rate* but have no idea which resources to request to meet this specification.

One solution to this problem would be to profile applications in advance and have them make low-level requests to the Policy layer; while plausible, this open-loop technique is not all that robust in the face of changing resources. Another option would be to perform on-line adaptation of resources based on preferred QoS metrics. It is this later idea that we will explore next.

Figure 4 illustrates the adaptive resource-allocation loop implemented by the Policy Layer (states **L1-L5**). This figure illustrates an important idea, namely that of the *performance report*, which is generated by a Cell-level runtime system for the Policy Layer. A performance report contains performance-counter values and/or metrics of how current application progress meets QoS goals. It may consist of both global metrics of system performance (e.g., power consumed by a Cell), and Cell-specific performance metrics (e.g., whether application-specific deadlines are being met).

The Policy Layer adapts resource allocations continuously in an attempt to help Cells achieve their best possible performance under current conditions. Those decisions result from a process that applies a *set of rules*

(adaptation policies) to the performance reports.

We are currently interested in two types of adaptation policies. The first represents user preferences as trajectories through predetermined operational points and follows those trajectories to maximize the user-perceived performance subject to currently observed conditions (e.g., [12]). The second consists of model-based adaptation policies (e.g., [5,8]). These policies rely on application-specific models that capture performance trade-offs across a large number of allocations.

### 3.2 Mapping and Time-Multiplexing

The Mapping and Time-Multiplexing Layer distributes the system resources among Cells and time-multiplexes them. Both functions are performed according to the parameters stored in the STRG. In other words, this layer makes no policy decisions, but rather implements the policy decisions given by the Policy Layer.

The Mapping and Time-Multiplexing Layer implements a *bin-packing like operation* in order to allocate physical resources to Cells. It implements a group of basic *time-multiplexing policies*:

- *Pinned Policy*: The Cell is *not* subject to time-multiplexing. Cells hosting important services (e.g., the network service) and applications with stringent performance requirements will use this policy.
- *Time-triggering policy*: The Cell is active during predetermined time-windows specified by the activation period (*ap*) and the length of the time-slice (*tsl*). This policy is for real-time Cells.
- *Time-fraction policy*: The Cell is active for a specified fraction of the time (e.g. 30%) and with a minimum time-slice. Most general-purpose applications will use this policy.
- *Message-triggered policy*: A message-triggered Cell provides flexible event-handling. It is activated upon the arrival of messages in designated input channels and may remain active or may be time-multiplexed while incoming messages remain.

Most time-multiplexing 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 (e.g., Message-triggered Cells).

## 4 Experimental Evaluation

In this section, we examine the potential for performance isolation in the Tessellation prototype. Currently, Tessellation contains over 22,000 lines of code and runs on both Intel x86 platforms and RAMP Gold [3], an FPGA-based simulator that models up to 64 in-order issue 1-GHz SPARC V8 cores, a shared memory hierarchy, and hardware partitioning mechanisms. The Intel system used in

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

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

our experiments is a dual socket system equipped two 2.67-GHz Xeon X5550 quad-core processors.

**Cell Activation and Suspension:** Table 1 summarizes the overhead of activating and suspending a Cell with varying cores counts on both RAMP Gold and our Intel System. This experiment helps validate that STP partitioning in Tessellation works properly and gives insight into the performance. These numbers are preliminary. The overhead is small relative to the time scale of Cell time-multiplexing (*i.e.*, greater than 10ms), but is still larger than we would like.

**Performance Isolation:** Our RAMP Gold platform simulates a mechanism (based on globally synchronized frames (GSF) [15]) that permits dedication of fractions of off-chip memory bandwidth to Cells. Using the RAMP Gold 64-core machine, with private L1s and a shared last level cache, we create three Cells not subject to time-multiplexing, with the following resource allocations:  $Cell_1$  is given 32 cores and 50% of memory bandwidth (*i.e.*, 6.4GB/s);  $Cell_2$  16 cores and 25% of memory bandwidth;  $Cell_3$  15 cores and 25% of memory bandwidth.  $Cell_1$  contains the PARSEC *streamcluster* and is our baseline for performance [6]. *streamcluster* was selected for its significant memory capacity and bandwidth requirements. Other PARSEC benchmarks are run in the remaining Cells to potentially interfere with  $Cell_1$ .

We first activate only  $Cell_1$  and run *streamcluster* multiple times. We then activate all th Cells and run the benchmarks concurrently. Finally, we run the benchmarks again, but disable the memory bandwidth partitioning mechanism, thereby allowing the Cells to compete for resources.  $Cell_1$  takes 5.70M, 6.12M and 11.59M core-cycles on average to complete, for the alone, isolated and un-isolated cases respectively. The respective standard deviations are 0.30M, 0.95M and 1.17M. These results show that Tessellation provides significant performance isolation. The isolation is not perfect, but it does reduce the degradation caused by the other Cells from 103% to 8% in this example, as well as making the performance more predictable.

**Spatial Partitioning:** Using the RAMP Gold 64-core target machine, we evaluate the potential of spatial partitioning using Tessellation. We take pairs of PARSEC applications, placing each application in a Cell. We then evaluate *all* possible spatial allocations for the two Cells.

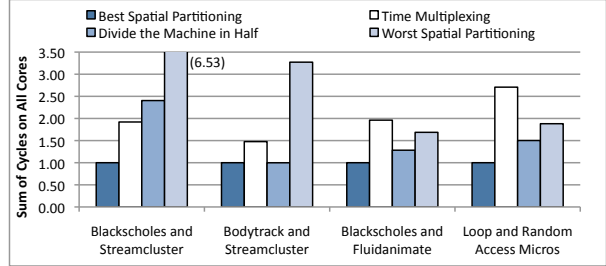


Figure 5: Performance of spatial partitioning compared with time-multiplexing. Performance is in cycles (lower is better) and results are normalized to the best spatial partition.

Cores are assigned in batches of 8, page colors in batches of 16, and memory bandwidth in batches of 3.4GB/s (a combinatorial total of 54 valid allocations). We also evaluate the time-multiplexing schedule in which one Cell is assigned the entire machine and run to completion followed by the next cell. This final schedule is favorable for time-multiplexed solution, as there is no overhead from repeated context switches.

For some pairs, the time-multiplexing solution was 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 5 illustrates the performance of several pairs. These results show that spatially partitioning greatly increase scheduler effectiveness. However, naive spatial divisions are likely to be detrimental to performance, even if optimal spatial divisions exist, meaning that our scheduler must be judicious in its assignment of resources to Cells.

## 5 Related Work

Tessellation is influenced by virtual machines, exokernels, and multiprocessor runtime systems [1, 2, 4, 9, 13, 14, 19]. Other recent manycore operating systems projects, such as Corey OS [10] and Barrelfish OS [20], share some structural aspects such as distributed OS services.

Nesbit et al. [17] 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

*Space-Time Partitioning* (STP) and *Two-Level Scheduling* are important principles for creating scalable operating systems that can provide performance guarantees to applications on manycore architectures. Tessellation is a new manycore operating system based on those principles. In this paper, we have discussed Tessellation’s Cell

model, its resource-allocation architecture, and some basic policies for resource management.

We have a working prototype of Tessellation. Our experimental results show that, when combined with hardware mechanisms, Tessellation can provide sufficient performance isolation to applications.

## 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 *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 100–109, New York, NY, USA, 2006. ACM.
- [3] K. Asanovic, D. A. Patterson, Z. Tan, A. Waterman, R. Avizienis, and Y. Lee. Ramp gold: An fpga-based architecture simulator for multiprocessors. In *Proc. of the 4th Workshop on Architectural Research Prototyping (WARP-2009)*, 2009.
- [4] P. Barham et al. Xen and the art of virtualization. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, 2003.
- [5] M. N. Bennani and D. A. Menasce. Resource allocation for autonomic data centers using analytic performance models. In *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 *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.
- [7] D. L. Black. Scheduling support for concurrency and parallelism in the mach operating system. *IEEE Computer*, 23:35–43, 1990.
- [8] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Automatic exploration of datacenter performance regimes. In *In Proc. First Workshop on Automated Control for Datacenters and Clouds (ACDC'09)*, June 2009.
- [9] T. L. Borden, J. P. Hennesy, et al. Multiple operating systems on one processor complex. *IBM Systems Journal*, 28(1):104–123, Jan. 1989.
- [10] 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.
- [11] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. CCS-TR-99-157, IDA Center for Computing Sciences, 1999.
- [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 *To appear in Proc. 13th IEEE Int'l Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC)*, May 2010.
- [13] 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.
- [14] J. Jann, L. M. Browning, et al. Dynamic reconfiguration: Basic building blocks for autonomic computing on ibm pseries servers. *IBM Systems Journal*, 42(1), Jan. 2003.
- [15] J. W. Lee, M. C. Ng, and K. Asanovic. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *Proc. of the International Symp. on Computer Architecture (ISCA)*. IEEE Computer Society, 2008.
- [16] 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 first USENIX Conference on Hot Topics in Parallelism (HotPAR)*, 2009.
- [17] 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.
- [18] H. Pan, B. Hindman, and K. Asanović. Lith: Enabling efficient composition of parallel libraries. In *Proc. of HotPar*, 2009.
- [19] 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)*, 2007.
- [20] A. Schüpbach et al. Embracing diversity in the barrellish manycore operating system. In *Proc. of the Workshop on Managed Many-Core Systems (MMCS)*, 2008.