# Tessellation: Space-Time Partitioning in a Manycore Client OS[*]

Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr[†], Krste Asanović, John Kubiatowicz
*Computer Science Division, University of California at Berkeley*
[†]*Lawrence Berkeley National Laboratory*
{*rfl, klueska, slbird*}*@eecs.berkeley.edu, shofmeyr@lbl.gov,* {*krste, kubitron*}*@eecs.berkeley.edu*

## Abstract

We argue for *space-time partitioning* (STP) in manycore operating systems. STP divides resources such as cores, cache, and network bandwidth amongst interacting software components. Components are given unrestricted access to their resources and may schedule them in an application-specific fashion, which is critical for good parallel application performance. Components communicate via messages, which are strictly controlled to enhance correctness and security. We discuss properties of STP and ways in which hardware can assist STP. We introduce Tessellation, a new operating system built on top of STP, which restructures a traditional operating system as a set of distributed interacting services. In Tessellation, parallel applications can efficiently coexist and interact with one another.

## 1 Introduction

All major vendors have ceased the relentless pursuit of individual CPU performance and have instead started doubling the number of CPUs per chip with each generation. Highly parallel *manycore* systems will soon be the mainstream, not just in large machine room servers but also in small client devices, such as laptops, tablets, and handhelds. This emergence of ubiquitous multiprocessing presents both a challenge and an opportunity. On the one hand, multiprocessing has achieved only limited success for general computing; the challenge is to find innovative and compelling ways in which to use an ever increasing number of CPUs. On the other hand, the presence of vast CPU resources presents an opportunity to fundamentally change the assumptions and structure of systems software.

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; a user may run multiple web applications, such as Gmail and Facebook, while listening to MP3 music files and video chatting with friends. In addition, battery life is a critical issue for client devices, requiring energy to be a first-class resource that is actively managed by the operating system.

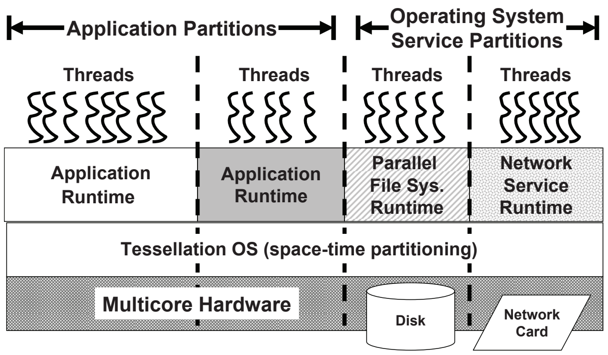

Figure 1: *Space-Time Partitioning* in Tessellation: a snapshot in time with four spatial partitions.

This paper argues that *space-time partitioning* (STP) is crucial for manycore client operating systems. A *spatial partition* (or "partition" for short) is an isolated unit containing a subset of physical machine resources such as cores, cache, memory, guaranteed fractions of memory or network bandwidth, and energy budget. *Space-time partitioning* virtualizes spatial partitions by time-multiplexing whole partitions onto available hardware, but at a coarse-enough granularity to allow efficient user-level scheduling within a partition.

Space-time partitioning leads to a restructuring of systems services as a set of interacting distributed components. We propose a new "exploded OS" called *Tessellation*, structured around space-time partitioning and *two-level scheduling* between the global and partition runtimes. Tessellation, shown in Figure 1, implements scheduling and resource management at the partition granularity. Applications and OS services run within their own partitions and have exclusive control of the scheduling of resources (*e.g.* cores, cache, memory) within their partitions. Partitions are lightweight, and can be resized or suspended with similar overheads to a process context switch.

## 2 A Case For Space-Time Partitioning

In this section, we make the case for space-time partitioning as a central component of a manycore operating system. One of the key tenets of our approach is that resources given to a partition are either exclusive (such as cores or private caches) or guaranteed via a quality-

[*]Research supported by Microsoft Award #024263 and Intel Award #024894 and by matching funding from U.C. Discovery (Award #DIG07-102270). This work has also been in part supported by an NSF Graduate Research Fellowship. 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.

of-service (QoS) contract (such as a minimum fraction of network or memory bandwidth). During a scheduling quantum, components within a partition are given unrestricted access to their resources via a standardized API that provides a thin abstraction layer, sufficient for portability and ease of programming. This API includes operations to configure the execution context of an application, such as setting up virtual memory translations and assigning threads to specific cores. It also includes mechanisms for constructing secure, restricted channels with other partitions, a topic we will explore in Section 3.

Effectively, components within a partition run as close as possible to the "bare metal" hardware with minimal OS interference. They flexibly control their resources by scheduling them in whatever way they choose. Thus, we expect that application runtimes and access to OS services will be provided as libraries (similar to a libOS [8]).

## 2.1 Partitioning for Performance

The need for space-time partitioning becomes clear as we consider the diverse application workload on a manycore client. Unlike high-performance computing (HPC) environments, where a single large application executes at a time, typical client environments will have many parallel applications interacting with one another and competing for shared resources.

The challenge of exploiting parallelism for client manycore environments is daunting. Successful parallelism strategies, *e.g.*, bulk-synchronous data parallelism or streaming pipeline parallelism, work best when the performance of components is predictable to enable load balancing. If co-scheduled applications or services generate interrupts that cause unpredictable resource availability, performance will suffer. Also, the variety of applications on a client will exhibit different forms of parallelism and incompatible performance goals (*e.g.*, interactive response vs. guaranteed throughput). Consequently, a single monolithic scheduler is unlikely to be sufficient for this diverse environment.

Spatial partitions with exclusive access to resources can provide a stable environment in which application-specific runtimes tailored to different parallel programming models and resource management techniques can execute without interference from incompatible OS scheduling policies. Further, space-time partitioning can provide both *functional* and *performance* isolation between simultaneously running applications with different performance goals.

## 2.2 Partitioning for Energy

Energy is a crucial resource for the client environment. Spatial partitioning provides an operating system's analog of clock gating (used for power reduction at the hardware level). With proper hardware support, whole partitions could exist in low power states until requests are made of them, at which point they could become enabled only long enough to satisfy a set of incoming requests.

Further, by carefully allocating resources to partitions, the kernel can control the power consumption of each software component. For instance, a portable device encountering a low-energy condition could choose to devote a minimal fraction of available memory and compute bandwidth to a crucial function such as cellphone service, while greatly restricting the resources available to non-crucial functions. Consequently functions such as background email synchronization would slow down or stop while continuing to support reception of calls.

## 2.3 Partitioning for QoS

A system with space-time partitioning can enforce QoS guarantees through performance isolation and strict control of inter-partition communication.

For instance, an IP network component could receive sufficient CPU and memory resources to guarantee its ability to handle external network traffic at line speed while performing intrusion detection, packet classification, and demultiplexing. Resources allocated to this partition are dedicated, ensuring that it can achieve its performance goals.

Another example is a file system service that receives a guaranteed resource allocation to provide service-level QoS guarantees to other partitions. The file system expresses these service-level QoS guarantees to other partitions in terms of the number of incoming requests that can be satisfied per unit time. This request bandwidth can be enforced by the controlled communication mechanisms, exported by an OS kernel, to limit message traffic across partition boundaries.

## 2.4 Partitioning for Correctness

Isolation of resources and inter-partition communication leads to enhanced security and correctness. Partitions are vulnerable at their boundaries. By enforcing predetermined communication patterns, the kernel can prevent many classes of attack and incorrect behaviors. Even if dynamic inter-partition communication patterns are desired, the partitioning mechanism can require each new connection to be vetted and approved by the kernel before communication is possible.

Further, the isolation properties of a partition could be exploited for *fast restart* of software components. Fast restart would be achieved by making sure that each partition has persistent state that is undo-able through logging or transaction support. Since partitions are isolated from one another, it is conceivable that a faulty service partition could be rapidly rebooted and restored to a functioning state without revoking its resources or disturbing other partitions.

## 2.5 Partitioning for Hybrid Behavior

An efficient inter-partition communication mechanism would enable the splitting of applications into multiple *sub-application* partitions. For instance, a parallel web browser [11] might utilize auxiliary partitions to hold untrusted plugins, thereby gaining additional security. Or, an application such as a virtual-reality game or music synthesis program might isolate realtime components from background model-building components via partitioning. Separating such components would simplify scheduling since each component can be configured/scheduled independently. Realtime sub-partitions can be given higher priority with stronger resource allocation guarantees than best-effort sub-partitions. A kernel API would provide support for application runtimes to configure sub-partition resource allocation, priority, and scheduling.

For heterogeneous architectures, partitions can encompass components with different computational capacity; for instance, some partitions may consist of "fat" cores while others include cores best adapted for SIMD computation. In this case, application partitioning can reflect the computational needs of an application.

## 3 Interaction Between Partitions

The simplest view of a partition is an isolated set of computational resources, but STP becomes particularly interesting when we consider partitions as interacting components within a larger system. In this section, we elaborate on three important mechanisms for selectively relaxing partition-level isolation: inter-partition communication, cross-partition quality-of-service guarantees, and partition virtualization.

## 3.1 Inter-Partition Channels

Partitions provide a natural framework for encapsulating major components of applications and the operating system. However, splitting an application across multiple partitions requires that these partitions be able to communicate with one another. If poorly designed, cross-partition communication could become a weak point in the security or QoS guarantees provided by the system as a whole. We believe that a misbehaving or compromised partition should not affect the QoS or correctness of other partitions, except in a very restricted fashion along previously-established inter-partition communication channels.

Our current model of inter-partition communication is via message passing. Message passing makes it easier to reason about the security of the system because messages are read-only and explicitly relinquished by the sender at the time of transmission. Message channels (that may be encrypted) must be authorized by a trusted code base (*e.g.*, as with partition tagging in HiStar [18]) before they can be used. The efficient implementation of message channels including hardware support is an ongoing topic of research, as is the channel interface; we may eventually relax our inter-partition communication model to include restricted forms of shared memory.

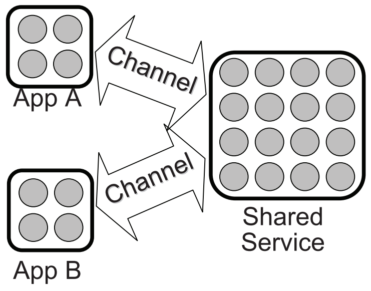

Figure 2: Request messages to shared service partitions must be tracked and controlled to meet QoS guarantees.

Each unidirectional message channel consists of a source endpoint and a destination endpoint. Since the destination partition of a message may be descheduled at the time that a message is sent, each destination endpoint must consist of resources (*e.g.*, a queue in memory) that persist at all times. There may be a variety of semantics for what happens upon a message enqueue depending on the scheduling policy for destination endpoints. For instance, if the partition is not currently running, the kernel scheduler could schedule the destination partition at some future (or the next) time quantum, or immediately schedule the destination partition. To notify the destination partition of the message, the kernel could either generate an interrupt to the destination partition (once it is running), or the destination partition could periodically poll for new messages. We are experimenting with batching most application and OS events in hopes of eliminating many of the traditional uses for interrupts.

## 3.2 Cross-Partition Quality of Service

Partitions only communicate with one another or the outside world through messaging. Consequently, monitoring and restricting the flow of messages provides an important mechanism for enforcing QoS.

For instance, suppose that applications A and B are granted equal quantities of some global resource (*e.g.*, energy, network bandwidth, or execution time). In the presence of a shared service (see Figure 2), two things must be accomplished to enforce QoS: First, the shared service must be profiled or otherwise analyzed to ascertain a relationship between incoming request rate and resource usage. Second, requests from applications A and B must be monitored and potentially suppressed at the source to prevent either application from denying the other its fair share of the shared service. Such source suppression is an analog of mechanisms that have been proposed to guarantee QoS for network bandwidth [12].

## 3.3 Partition Virtualization

Partitions can be virtualized and multiplexed on physical hardware by a global partition manager. Partition resources *must be* gang scheduled to provide application runtimes with full knowledge and control over their resources to enable them to implement optimized scheduling and resource allocation internally. To allow application-level scheduling, the time quantum at which partitions are run should be much longer than that for traditional OS processes. Some partitions with strict resource demands, *e.g.*, highly utilized services or hard realtime applications, may be pinned to cores or multiplexed at a predictable rate. Other partitions may only be scheduled in response to messages sent to that partition. For example, rarely used services could be ready to run but only scheduled when an application makes a request.

When starting, an application that requires resource guarantees must express its resource and QoS requirements to a global partition manager in charge of actually allocating out those resources; such requirements may be discovered through profiling, analysis, or online tuning. If the demands can be met without violating other applications' guarantees, then the new application can start (it is *admitted*). However, if admitting the application would cause oversubscription—resulting in failure to meet all of the current QoS contracts—then the application must be denied. The user will be informed that the new application can not be admitted without changing the current set of running applications.

## 4 Space-Time Partitioning in Tessellation

The Tessellation Kernel (shown in Figure 3) is a thin, trusted layer that implements resource allocation and scheduling at the partition granularity. Tessellation exploits a combination of hardware and software mechanisms to perform space-time partitioning and provides a standardized API for applications to configure resources and construct secure restricted communication channels between partitions. The Tessellation Kernel is much thinner than traditional monolithic kernels or even hypervisors, and avoids many of the performance issues with microkernels by providing OS services in spatially-distributed active partitions through secure messaging (thus avoiding context switches).

## 4.1 Hardware Partitioning Mechanisms

In this section, we discuss hardware mechanisms (existing and proposed) to aid in partitioning and reduce the overhead of time-multiplexing a partition. The *Partition Mechanism Layer* (next section) combines these mechanisms to provide a uniform partitioning API.

Cores (CPUs) are controlled by restricting the ability of applications to run threads on the cores. Physical memory is partitioned using conventional mechanisms such as page tables and TLBs.

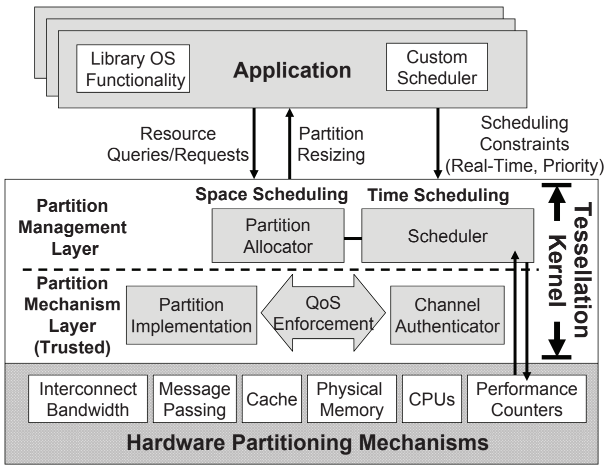

Figure 3: Components of a Tessellation system. The Tessellation Kernel utilizes a combination of hardware and software mechanisms to enforce partition boundaries.

We need hardware support to partition shared resources, *e.g.*, caches and network/memory bandwidth, in order to provide performance isolation. Performance isolation enables the operating system to run arbitrary applications simultaneously (spatially distributed), while providing a simple performance model to each application. Shared caches should be partitionable flexibly and dynamically for efficient utilization. Previous works suggest some possible mechanisms to partition caches such as [17] or [9]. Similarly, hardware support is needed to partition interconnect bandwidth. Lee et. al. describe one possible solution for providing QoS in on-chip networks [12]. Such mechanisms can be extended to provide QoS for off-chip memory usage.

Ideally, we also envision having hardware support to secure message channels between partitions. Mechanisms such as message tagging would allow partitions to initiate communication without the intervention of privileged software. Support to track and regulate the volume of inter-partition communication would enable efficient implementations of service-level QoS.

So far, we have described mechanisms for *enforcing* spatial partitions. Tessellation also uses hardware performance counters to *monitor* application performance, resource usage, and energy consumption. Performance information can be collected for all resources within a partition to determine the runtime behavior of an application during its different phases. This information can then be used to adjust resource allocations for better utilization. Ideally, future hardware would support gathering and reporting statistics at the partition granularity.

## 4.2 The Partition Mechanism Layer

The lowest layer of the Tessellation Kernel is called the *Partition Mechanism Layer*, as shown in Figure 3. This thin, machine-dependent layer is responsible for configuring available hardware mechanisms to enforce dynamic hardware partitions. When mechanisms are not available directly in hardware, the Partition Mechanism Layer implements them (a form of paravirtualization). It provides a machine-independent interface to the policy layer (next section). By separating mechanisms from policy, we can easily verify the correctness of the mechanism layer, which is critical to functional and performance isolation.

In our current experimental platform, cores are partitioned in software by controlling the ability of application runtimes to place a thread onto a particular core; the kernel proceeds only if the core belongs to the requesting partition. Lacking hardware support for messaging, message channels are constructed by mapping message FIFOs in protected shared memory between partitions.

## 4.3 The Partition Manager

The heart of the Tessellation Kernel is a policy layer, called the *Partition Manager*. The Partition Manager schedules and allocates resources to applications and service partitions, while balancing system performance with energy consumption, to achieve high-level objectives, such as battery life or system responsiveness. Resource allocation, scheduling of partitions, and partition virtualization is described more in Section 3.3.

To determine *which* resources and *how much* of those resources should be allocated to each partition, the Partition Manager needs an abstract understanding of the machine topology; the Partition Mechanisms Layer should provide this abstract specification in a machine-independent way. Fortunately, future single-chip many-cores will have lower communication latencies than existing symmetric multiprocessors (SMPs) which simplifies the tradeoff of either minimizing the latency of private data access per thread or minimizing communication latency between threads. Consequently, to first order, the Partition Manager will only need to determine *how much* of a resource to allocate rather than *which* resource to allocate.

Since the optimal number of partition resources required by an application can vary over time, the Partition Manager should dynamically adjust allocations to optimize system efficiency. The partition resizing API implemented by application runtimes (shown in Figure 3 and described further in Section 4.4) permits the Partition Manager to resize running application partitions. Applications that do not implement this interface prevent the Partition Manager from dynamically changing the size of a spatial partition to improve performance or reduce energy consumption (although such partitions can still be time-multiplexed).

The Partition Manager uses hardware performance counters, as described in Section 4.1, to determine the best resource allocation for each partition. The Partition Manager uses these counters to dynamically build a model correlating resource allocation with estimated performance on a partition basis; This is in the spirit of Quereshi's work [14], however, our goal is to speed up an entire parallel application, not just one of its threads (which may not improve application performance). We are currently exploring policies for resource allocation trading off system performance with energy consumption.

The Partition Manager time-multiplexes partitions onto the machine. Partitions whose configurations fit on the machine can be scheduled simultaneously. The performance isolation provided by partitioning eliminates the need to worry about how the applications may interfere with each other. Unallocated resources can be left in a low-power state, or used to run an application that can make reasonable progress with only those resources.

## 4.4 Application Use of Partitions

Tessellation provides both *fixed-sized partitions* and *dynamically-sized partitions* to applications. Applications written for fixed sized partitions query Tessellation for available resources and request a certain amount. The Partition Manager can choose to grant the original request or select a different allocation of resources. Once created, a fixed-sized partition is guaranteed to have that number of resources when scheduled. Applications that support dynamically-sized partitions implement the partition resizing API to allow the dynamic addition and removal of resources during runtime. This protocol is based on a callback mechanism, similar to scheduler activations [3].

Tessellation exports three additional interfaces. One allows applications to dynamically request and return resources to the Partition Manager. Another allows applications to specify scheduling constraints such as real time deadlines or priorities. A third allows applications to initiate secure channels with other partitions.

The Tessellation effort includes work on user-level libraries for exporting traditional I/O and system call interfaces to applications running in partitions. We provide a version of NewLibC [1] that implements system services by sending messages to service partitions; initially, these service partitions are running complete copies of operating systems such as FreeBSD. We are also collaborating closely with efforts to provide frameworks for constructing custom user-level runtime systems, such as Lithe [13].

## 5 Related Work

Tessellation is influenced by virtual machines, exokernels, and multiprocessor runtime systems. Similar to Xen [5] and VMware ESX [2], Tessellation virtualizes machine resources. Unlike these systems, Tessellation virtualizes resources at the partition granularity, guaranteeing that CPUs within a partition are scheduled simultaneously. Similar to the Exokernel [8], Tessellation implements system services extensibly at user-level, allowing, for instance, applications to choose a user-level runtime best suited for the application. However, Tesselation avoids the shortcomings of the Exokernel by providing a thin abstraction layer that ensures portability without restricting an application's access to partition resources. Further, Tessellation can support multiple heterogeneous runtimes simultaneously through isolation.

Unlike LPAR [6] and DLPAR [10], hardware partitioning within Tessellation is lightweight and flexible; the overhead of resizing or multiplexing partitions is of the order of a process context switch. Space-time partitioning complements threading runtimes such as McRT [15]. The sequestered mode of McRT, which runs directly on bare-metal resources and acts as a light weight threading system, is exactly the type of environment that Tessellation's spatial partitioning can provide.

Existing operating systems (e.g. Linux, BSD, or Windows) operate at the granularity of individual CPUs, and therefore use the thread abstraction to make resource allocation and scheduling decisions. We claim that spatial partitions provide a more natural abstraction for supporting multiple parallel applications within a client manycore device. Further, spatial partitions act as a natural abstraction for implementing resource allocation and accounting frameworks such as resource containers [4] and energy-aware scheduling policies such as [19].

Researchers have started to explore the space of manycore operating systems. Corey OS [7] is a manycore OS that achieves scaling by giving programmers control over the sharing of kernel data structures. We believe that Tessellation's distributed OS services approach combined with space-time partitioning helps with scaling. Barrelfish OS [16] is geared towards manycore platforms and shares our high-level goal of structuring the OS as a distributed system.

## 6 Conclusions and Future Work

We argued for *space-time partitioning* (STP) as the primary abstraction for resource management on manycore client devices, and presented a new OS, called Tessellation, based on STP. We assert that STP is crucial for the programmability and performance of parallel applications in a multi-application environment and serves as a natural primitive for energy management, fault containment, security, and service-level QoS.

We are currently implementing Tessellation OS with resource partitioning and secure channels; several components exist already. Initially, device drivers and I/O will be provided by FreeBSD running in a separate partition. Soon thereafter, we will build partition-aware file systems and TCP/IP interfaces. We are investigating efficient dynamic resource allocation policies, QoS mechanisms, and portable representations of resources and communication topology. We will soon have several secure channel designs, composable user-level scheduling, and energy management. STP allows a restructuring of traditional operating systems—an exciting prospect that we look forward to reporting on at a future date.

## 7 Acknowledgements

We would also like to thank the rest of the ParLab OS group and several people at LBNL for their invaluable contribution to the ideas in this paper.

## References

[1] NewLibC. http://sourceware.org/newlib/.

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

[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] 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.

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

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

[7] 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.

[8] 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.

[9] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proc. of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2007.

[10] 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.

[11] C. G. Jones et al. Parallelizing the web browser. In *Proc. of HotPar*, 2009.

[12] 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)*, 2008.

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

[14] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. of the IEEE/ACM International Symp. on Microarchitecture (MICRO)*, 2006.

[15] 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.

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

[17] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared l2 caches on multicore systems in software. In *Proc. of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2007.

[18] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proc. of the ACM Symp. on Operating Systems Design and Implementation (OSDI)*, 2006.

[19] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: managing energy as a first class operating system resource. *SIGPLAN Not.*, 37(10):123–132, 2002.