Tessellation: Refactoring the OS around Explicit Resource Containers with Continuous Adaptation

Juan A. Colmenares[♦][♭], Gage Eads[♦], Steven Hofmeyr[†], Sarah Bird[♦], Miquel Moretó[♦], David Chou[♦], Brian Gluzman[♦], Eric Roman[†], Davide B. Bartolini[♦], Nitesh Mor[♦], Krste Asanović[♦], John D. Kubiatowicz[♦]
[♦]The Parallel Computing Laboratory, UC Berkeley, Berkeley, CA, USA
[†]Lawrence Berkeley National Laboratory, Berkeley, CA, USA
[♭]Samsung Research America - Silicon Valley, San Jose, CA, USA
juan.col@samsung.com, geads@eecs.berkeley.edu, shofmeyr@lbl.gov, sbird@eecs.berkeley.edu, mmoreto@ac.upc.edu, {brian.gluzman,davidchou}@berkeley.edu, eroman@lbl.gov, {dbb,mor,krste,kubitron}@eecs.berkeley.edu

ABSTRACT

Adaptive Resource-Centric Computing (ARCC) enables a simultaneous mix of high-throughput parallel, real-time, and interactive applications through automatic discovery of the correct mix of resource assignments necessary to achieve application requirements. This approach, embodied in the Tessellation manycore operating system, distributes resources to QoS domains called *cells*. Tessellation separates global decisions about the allocation of resources *to* cells from application-specific scheduling of resources *within* cells. We examine the implementation of ARCC in the Tessellation OS, highlight Tessellation's ability to provide predictable performance, and investigate the performance of Tessellation services within cells.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management – Scheduling; D.4.7 [**Operating Systems**]: Organization and Design – Real-time and embedded systems; D.4.8 [**Operating Systems**]: Performance – Measurements, Monitors

General Terms

Multicore, parallel, quality of service, resource containers

Keywords

Adaptive resource management, performance isolation, quality of service

1. INTRODUCTION

Today's users demand lightning fast interaction with their portable devices while simultaneously displaying glitch-free multimedia and interacting with a variety of external information sources. Further, the growing number of mobile devices, sensors, and embedded systems makes efficiency a primary concern as systems demand an increasing amount of computation from the same battery capacity. Such requirements are very different from the high-throughput workloads that drove the design of operating systems in the past.

Modern applications consist of multiple interacting components, each with differing resource needs and quality-of-service (QoS) requirements. They often extend into the cloud and may include sensors and other embedded systems, requiring the operating system to provide responsiveness and performance predictability on a global scale. One example is that of a modern smartphone reproducing music retrieved on the fly from the cloud while rendering web pages with interactive multimedia contents. Other examples include Distributed Real-time Embedded (DRE) systems that control fly-by-wire airplanes and manufacturing plants; these systems need to ensure performance predictability across increasingly parallel components [13] and communication networks. Future DRE systems, such as autonomous vehicles and health-monitoring applications, will

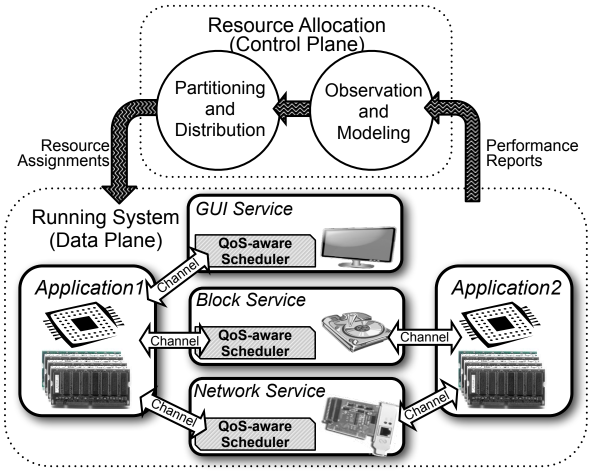

Figure 1: Adaptive Resource-Centric Computing: *Cells* provide performance isolation and guaranteed access to resources for applications and services. Cells are depicted as rounded boxes with solid lines. Resource allocations are automatically adjusted to maximize overall system utility. Resources wrapped with cells provide guaranteed services to other cells.

also have to manage large swaths of data from growing, interconnected "swarms" of sensors [22].

Thus, a new paradigm for the interaction between application components and systems software is clearly needed. Since resources are central to performance, predictability, and efficiency, we propose to utilize *Adaptive Resource-Centric Computing* (ARCC), as illustrated by Figure 1. In ARCC, resources are distributed to explicitly parallel, lightweight resource containers called *cells*, which provide stable execution environments for the software components running within them. Further, *composite* resources are constructed by wrapping cells around existing resources and exporting service interfaces. Applications can then be allocated QoS contracts from these services. This *service-oriented architecture* [34] approach enables performance predictability for a variety of more complex system services.

To reduce the burden on the programmer and to respond to changing environmental circumstances, our approach automatically adjusts resource allocations to meet application requirements. The stable environment of a cell makes it possible to experimentally observe user-defined progress metrics and predict how these metrics vary with resources – thus enabling accurate resource optimization. This approach addresses the *impedance-mismatch* between programmer and system that results from the fact that user-meaningful QoS metrics (*e.g.,* frame rate) are only indirectly related to resource allocations (*e.g.,* processor cycles).

In this paper, we illustrate the fundamental concepts of ARCC and evaluate it with Tessellation, a novel operating system for multi- and manycore systems. We base our evaluation on typical client applications, but the underlying methodology is equally applicable to embedded systems.

2. TESSELLATION ARCHITECTURE

In this section, we summarize some of the key components of Tessellation OS [11, 27], as illustrated by Figure 2. The Tessellation kernel is a thin, hypervisor-like layer that provides support for ARCC by implementing *cells* and providing interfaces for resource adaptation and cell composition. Tessellation uses *two-level scheduling* [24, 30, 11] to separate resource *allocation* from resource *usage*. This approach supports custom schedulers for efficient resource usage.

2.1 The Cell Model

Cells provide the basic unit of computation and protection in Tessellation. Cells are performance-isolated resource containers that export their resources to user level. The software running within each cell has full user-level control of the resources assigned to the cell, including CPU cores and memory pages. In the future, we plan to extend cells with multiple address spaces managed by the cell user-level runtime. We envision this facility as supporting more traditional UNIX-style processes or multi-component device drivers (*e.g.,* USB) within a cell.

Applications in Tessellation are created by composing cells via efficient and secure *channels*. Channels provide fast, user-level asynchronous message-passing between cells. Standard OS services (*e.g.,* network and file services) are hosted in cells and accessed via channels.

Tessellation OS virtualizes resources using *space-time partitioning* [36, 27, 28], a multiplexing technique that divides the hardware into a sequence of simultaneously-resident spatial partitions. With space-time partitioning, CPU cores and

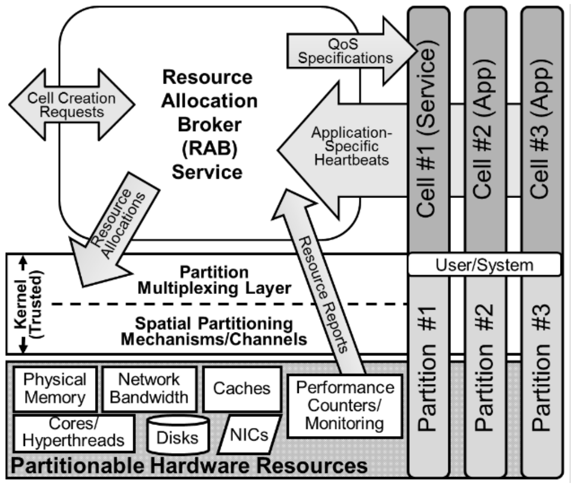

Figure 2: The Tessellation kernel implements *cells* through *spatial-partitioning*. The *Resource Allocation Broker* redistributes resources after consulting application-specific *heartbeats* and system-wide *resource reports*.

other resources are *gang-scheduled* [31, 14]. Cells thus provide to their hosted applications an environment that is very similar to a dedicated machine.

Partitionable resources include CPU cores, pages in memory, and guaranteed fractional services from other cells (*e.g.,* a throughput reservation of 150 Mbps from the network service). They may also include cache slices, portions of memory bandwidth, and fractions of the energy budget, when hardware support is available (*e.g.,* [4, 23, 38]).

Two-level scheduling [24, 30, 11] in Tessellation separates global decisions about resource allocation *to* cells (*first level*) from management and scheduling of resources *within* cells (*second level*). Resource redistribution occurs at a coarse time scale to amortize the decision-making cost and allow time for second-level scheduling decisions (made by each cell) to become effective.

The user-level runtime within each cell may utilize its resources (*e.g.,* hardware-thread contexts and memory pages) as it wishes – without interference from other cells. The cell's runtime can thus be customized for specific applications or application domains with, for instance, a particular scheduling algorithm and page replacement policy. Section 4 discusses second-level scheduling in detail.

2.2 Service-Oriented Architecture

Cells provide a convenient abstraction for building OS services with QoS guarantees. Such services can reside in dedicated cells, have exclusive control over devices, and encapsulate user-level device drivers (see Figure 1). Hence, each service can arbitrate access to its enclosed devices, and leverage its cell's performance isolation and customizable schedulers to offer service guarantees to other cells.[1] Services can shape data and event flows coming from external sources with unpredictable behavior and prevent other cells from being affected.

Two services in Tessellation that offer QoS guarantees are: the *Network Service*, which provides access to network adapters and guarantees that the data flows are processed

[1]In keeping with ARCC, we view the services offered by such *service cells* as additional resources to be managed by the adaptive resource allocation architecture.

with the agreed levels of throughput; and the *GUI Service*, which provides a windowing system with response-time guarantees for visual applications [19]. For details about the implementation and performance of these services, refer to Appendix B.

2.3 Adaptive Resource Allocation

Tessellation uses an adaptive resource-allocation approach to provide QoS guarantees to applications while maximizing efficiency in the system. The *Resource Allocation Broker* (RAB) is a *broker service* that distributes resources to cells while attempting to satisfy competing system-wide goals, such as deadlines met, energy efficiency, and throughput. Allocation decisions are communicated to the kernel and services for enforcement. The RAB Service uses system-wide goals, resource constraints, performance targets and current performance measurements as inputs to the optimization.

The RAB Service reallocates resources, for example, when a cell starts or finishes or when a cell significantly changes performance. The RAB Service can periodically adjust allocations; the reallocation frequency provides a tradeoff between adaptability (to changes in state) and stability (of user-level scheduling).

The RAB Service runs in its own cell and communicates with applications through channels. When a cell is started, it provides its QoS requirements to the RAB Service in the form of target performance goals, such as desired framerates. The RAB Service continuously monitors the cells' performance and compares it to target rates, adjusting resource allocations as required. To do this, the RAB Service utilizes two sources of information:

- Periodic performance reports, *heartbeats* [17], containing application-specific performance metrics from the cells (*e.g.,* the time to render a frame for a video app).
- System-wide performance counter values, such as cache-miss statistics and energy measurements.

The RAB Service provides a resource-allocation framework that supports rapid development and testing of new allocation policies. Section 5 demonstrates a few simple policies we developed as a proof of concept. The development time for each policy was under an hour. Using this framework we can explore the tradeoffs between enabling software components to meet their performance goals and optimizing resource distribution to achieve global objectives.

3. IMPLEMENTING THE CELL MODEL

As shown in Figure 2, the Tessellation kernel comprises two layers, the *Partition Multiplexing Layer* (or Mux Layer) and the *Spatial Partitioning Mechanisms Layer* (or Mechanism Layer). The Mechanism Layer performs spatial partitioning and provides resource guarantees by exploiting hardware partitioning mechanisms (when available) or through software emulation (*e.g.,* cache partitioning can be implemented using page coloring). Building on this support, the Mux Layer implements space-time partitioning and translates resource allocations from the RAB Service into an ordered time sequence of spatial partitions.

3.1 Types of Cells

The Mux Layer offers several time-multiplexing policies for cells to support applications (or parts thereof) with different timing requirements. Each multiplexing policy defines a cell type with a specific timing behavior. Tessellation provides: 1) non-multiplexed (non-muxed) cells with dedicated access to cores; 2) time-triggered (TT) cells, which are active during periodic time windows; 3) event-triggered (ET) cells, which are activated upon event arrivals, but never exceed their assigned fraction of processing time; and 4) best-effort (BE) cells with no time guarantees. The cell types are explained in greater detail in Appendix A.

These time-multiplexing policies allow users to easily specify the desired timing behavior for cells with a certain precision (currently 1 ms). The Mux Layer then ensures that, if feasible, a set of cells with different multiplexing policies harmoniously coexist and receive the specified time guarantees. In this way, Tessellation offers precise control over cells' timing behavior, one of the characteristics that differentiates Tessellation from traditional hypervisors and virtual machine monitors [5, 20].

3.2 Space-Time Partitioning

On each hardware thread there is a separate multiplexer (or *muxer*) that controls the multiplexing of cells on that thread. The muxers collectively implement gang scheduling [31] in a *decentralized* manner. They execute the *same* scheduling algorithm and rely on a high-precision global-time base [21] to simultaneously activate a cell on multiple hardware threads with minimum skew. In the common case, the muxers operate independently and do not communicate to coordinate the simultaneous activation of cells. The muxers thus implement an instance of *communication-avoiding* gang-scheduling.

For correct gang scheduling, the muxers need to maintain an identical view of the system's state whenever a scheduling decision is made. Hence, each muxer makes not only its own scheduling decisions but also reproduces the decisions made by other (related) muxers with overlapping schedules. In the worst case, each muxer must schedule the cell activations happening in every hardware thread in the system, but the RAB Service tries to avoid such unfavorable mappings.

The muxers implement gang-scheduling using a variant of *Earliest Deadline First* (EDF) [26], combined with the *Constant Bandwidth Server* (CBS) [3] reservation scheme in order to provide the variety of timing behaviors required by the cell types. EDF is used to implement TT cells while CBS is used for ET and BE cells. More details can be found in Appendix A.

3.3 Redistributing Resources among Cells

To request a redistribution of resources among cells (*e.g.,* resizing cells, changing timing parameters, or starting new cells), the RAB Service passes the new distribution to the Mux Layer via a system call (only accessible to this service). To implement resource-distribution changes, each muxer has two scheduler instances: one active and one inactive. The Mux Layer first validates the new resource distribution and, if successful, proceeds to serve the request. The Mux Layer next resets and prepares the inactive schedulers, and establishes the *global* time in the near future (*e.g.,* 1 ms later) at which the muxers will synchronously exchange their active and inactive schedulers. Then, the Mux Layer sends the muxers a message with the global time value and the system call returns. Finally, at the specified time, the muxers exchange their schedulers and perform other actions related to the relocation of cells (*e.g.,* re-routing device interrupts). This approach allows the Mux Layer to process resource-distribution requests almost entirely without disturbing the system's operation with only the overhead of switching schedulers and other cell-relocation actions. Note that if a subset of muxers is involved in a resource redistribution, only that subset performs the scheduler switch.

4. USER-LEVEL RUNTIME

One benefit of two-level scheduling is the ability to support different resource-management policies simultaneously. In Tessellation OS, cells provide their own, possibly highly-customized, user-level runtime system for processor (thread) scheduling and memory management. Further, each cell's runtime can control the delivery of events, such as timer and device interrupts, inter-cell message notifications, exceptions, and memory faults. This section describes the support Tessellation offers to implement user-level runtimes.

Our current Tessellation prototype includes two user-level thread scheduling frameworks: a *preemptive* one called PULSE (Preemptive User-Level SchEduling), and a *cooperative* one based on Lithe (LIquid THrEads) [33]. With either framework, a cell starts when a single entry point, `enter()`, is executed simultaneously on each core. After that, the kernel interferes with the cell's runtime only when: 1) the runtime receives events (*e.g.,* interrupts) it has registered for, 2) the cell is suspended and reactivated according to its time-multiplexing policy, and 3) the resources (*e.g.,* hardware threads) assigned to the cell change in response to RAB Service's requests. For instance, when an interrupt occurs during user-level code execution, the kernel saves the thread context and calls a registered interrupt handler, passing the saved context to the cell's user-level runtime. This way the cell's runtime can then choose whether to restore the previously running context or swap to a new one.

Since preemptive scheduling is commonly used in embedded and other types of computer systems, now we discuss PULSE, our framework for user-level preemptive scheduling, in more detail. PULSE is a simple framework, written in less than 800 lines of code (LOC). Creating a new user-level preemptive scheduler with PULSE is an easy task. The runtime must implement several callbacks, for instance: `enter()`, mentioned earlier; `tick(context)`, which is called whenever a timer tick occurs and receives the context of the interrupted thread; `yield()`, called when a thread yields; and `done()`, called when a thread terminates. The framework also provides functions for saving and restoring contexts, and other relevant operations.

PULSE's simplicity makes it easy to implement and customize schedulers – without having to patch the OS kernel, as is often the case with Linux. For example, we implemented a global round-robin scheduler with mutex and conditional-variable support, in ∼850 LOC. We also wrote a global EDF scheduler with mutex support and priority-inversion control via dynamic deadline modification (DDM) [18], in less than 1000 LOC.

To support adaptive resource allocation, user-level runtimes must adjust to changes in the number of hardware threads assigned to their host cells. PULSE propagates changes in the number of hardware threads to the user-level scheduler. In the event that hardware threads are added, PULSE informs the runtime that additional resources are available. When a cell loses hardware threads (or harts), the framework delivers the extra application contexts (from the revoked hardware threads) to the user-level scheduler's scheduling queue. If any of the saved contexts are scheduler contexts, PULSE uses a *non-preemptive auxiliary scheduler* that executes all scheduler contexts one by one until they enter application contexts. Then PULSE communicates these application contexts to the user-level scheduler with the callback `adapt(prev_num_harts, new_num_harts, context*)`. PULSE implements this procedure so that the user-level scheduler is completely unaware that adaptation is occurring until the `adapt` callback is executed. Consequently, we were able to add support for adaptation in the round-robin and global EDF schedulers simply by adding the `adapt` callback.

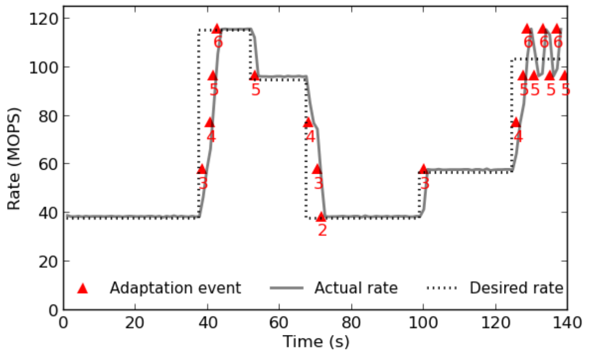

Figure 3: Adaptation of core counts while running the NAS EP benchmark. The number under each adaptation marker indicates the number of cores in the new allocation.

Although still a work in progress, Tessellation will soon support a user-level paging facility. The Resource Allocation Broker will allocate physical memory pages to cells after which the user-level runtime will manage these pages with customized page replacement and allocation policies, and use the block storage device(s) through a service interface[2] to implement paging, if desired. This approach, similar to self-paging in the Nemesis OS [16], can minimize uncontrolled interference between cells and enable better performance predictability and stronger QoS guarantees.

5. EXPERIMENTAL EVALUATION

In this section, we investigate the adaptive behavior of Tessellation OS through two simple experiments. The first shows the performance of the adaptation mechanisms in the kernel and user-level preemptive runtime when changing the number of cores assigned to a cell. The second, demonstrates the system's ability to adjust, via the RAB Service, the QoS guarantee that a service offers to a cell so that it can meet its performance goals.

5.1 Adjusting Core Allocation

This experiment is a simple feedback loop between a compute intensive application and the RAB Service. The application used was the NAS EP benchmark [2], which generates random numbers in an embarrassingly parallel manner. We chose EP because it scales perfectly and uses negligible memory; so we can easily confirm that it behaves as expected when varying number of cores.

EP was configured for 6 threads and run in a non-muxed cell with our global round-robin scheduler. The RAB Service

[2]We are currently developing a QoS-aware service that provides guaranteed client access to hard disks and other storage devices, called the Block Device Service.

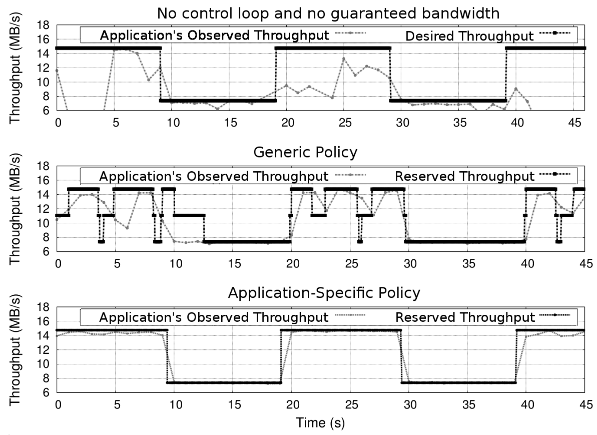

Figure 4: Adaptive network throughput no guarantees, with a general-purpose policy, and with an application-specific policy.

resided in a separate non-muxed cell running on a dedicated core. We modified EP to compute the rate R at which random numbers were generated, and set a goal of R' for the desired rate of random number generation. Each second, EP computed R and sent the value to the RAB Service.

We set 5 resource allocations, *operating points*, with core allocations of 2, 3, 4, 5, and 6. The RAB Service used a simple reactive policy to adjust the allocation of cores. When the performance is too low (*i.e.,* $R < R'$), it allocates the next larger operating point; when the performance is too high (*i.e.,* $R > R' + \epsilon$), it allocates the next smaller operating point. We used $\epsilon = 0.1$ to prevent oscillation between operating points.

We ran this experiment on an Intel system with two 2.66-GHz Xeon X5550 quad-core processors and hyper-threading disabled (*i.e.,* 8 hardware threads). The results can be seen in Figure 3, which shows the performance of EP as the desired rate R' changes periodically. There is a small lag between an adaptation event and the resulting performance change because EP only reports changes every second. We also see a noticeable lag in performance when a large change is required since the simple reactive policy shifts only one operating point at a time, and then waits for a new measurement before shifting again. Note, however, after the adaptation time it does settle on the correct rate illustrating the effectiveness of Tessellation's adaptation mechanisms. Although clearly, the reactive policy used here is too limited for general use as shown by the oscillations at $t \geq 125$ s.

5.2 Adjusting Service Guarantees

In this experiment, the RAB Service drove the Network Service to allocate network throughput to an application. We show how the RAB Service adjusted the throughput reservation for a video player against changes in the incoming video stream, so that the video player's performance goals were met, while enabling efficient use of the Network Service. We used two computers (1 Linux, 1 Tessellation), both with an Intel 3.4-GHz Core i7 quad-core processor with hyper-threading (*i.e.,* 8 hardware threads), and 4 GB of RAM, directly connected via 1-Gbps Ethernet adapters.

The Linux box was the video source and sent video frames to the Tessellation box over a TCP connection. The Linux box produced a stream of uncompressed frames at a constant rate of 24 frames per second (fps). The frame size was

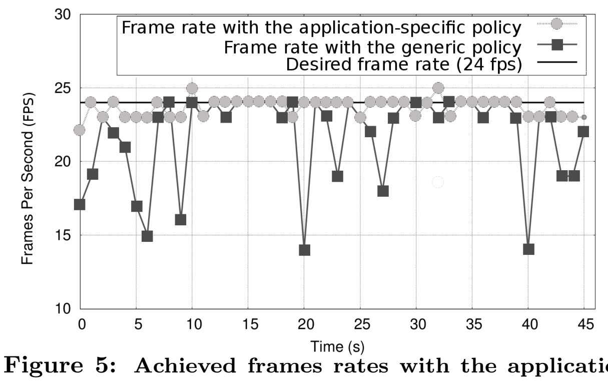

Figure 5: Achieved frames rates with the application-specific policy and the generic policy.

adjusted between 480x320 and 320x240 every 10 s to simulate changing video quality. Thus, large frames required 14.74 MB/s while small frames 7.37 MB/s.

On the Tessellation box, the video-player application received the video stream via the Network Service and passed the reconstructed frames to the GUI Service for display. The video player periodically sent performance reports to the RAB Service. Using this information, RAB Service chose the throughput reservation the Network Service provided to the video player. Additionally, two "bandwidth-hog" applications ran on the Tessellation box and contended with the video player for bandwidth by constantly sending UDP messages to the Linux box. The Network Service gave no bandwidth guarantees to these applications, which shared the excess of bandwidth. On the Tessellation box, applications and services resided in separate non-multiplexed cells.

Ideally, the RAB Service should allocate sufficient throughput for the video player to reach 24 fps without over provisioning. To this end, we experimented with two simple reactive policies: an *application-specific policy*, P_{App}, that exploits knowledge about the video stream's bandwidth demands; and a *general-purpose policy*, P_{Gen}, that operates only based on the video player's observed performance. P_{App} uses reports that explicitly specify the size of the frames being received. P_{Gen}, on the other hand, uses reports with the average frame inter-arrival time (in milliseconds per frame), and has three operating points: 15.0 MB/s, 11.0 MB/s, and 7.5 MB/s. When the RAB Service receives a performance report from the video player, the policy in use determines whether the application is falling short of its performance goal and needs higher throughput from the Network Service, or whether the applications is exceeding the performance goal by a pre-determined threshold could possibly run at a lower operating point. Both policies include hysteresis to prevent oscillations between operating points.

Figure 4 shows how the system reacts to changes in the incoming video stream for each policy type. The upper graph shows the video player's observed throughput with no guaranteed bandwidth reservation. In this case, the two "bandwidth-hog" applications consume too much bandwidth to allow the video player to receive its required amount. The lower graph shows the results of P_{App}, where the Network Service (via the RAB Service) rapidly adapts to the changing needs of the video player. For P_{Gen} (the center graph), the high-bandwidth intervals contain occasional low-reservation glitches when it probes the next lowest operating point. P_{Gen} is flexible enough for any scenario where a latency value (*e.g.,* frame inter-arrival time) is the key metric, but is not optimized for the video-player application. However, adjusting the frequency of probing, number of operating points, and hysteresis in the RAB Service can improve the system's adaptation accuracy.

Figure 5 shows the video player's observed frame rate, which is the user-meaningful metric considered here. P_{App} achieves near-constant 24 fps with glitches near the high-bandwidth to low-bandwidth transition points. As expected, P_{Gen} falls short of 24 fps during the high-bandwidth intervals when the RAB Service probes lower operating points. This experiment demonstrates that the system, composed by the video player, the RAB Service, and the Network Service, can adapt to changes in the incoming video stream.

6. CONCLUSIONS

This paper introduced *Adaptive Resource-Centric Computing* (ARCC) and described and evaluated its implementation in the Tessellation OS. The key points for ARCC are:

- Resources distributed to QoS domains called *cells*, which are explicitly parallel, light-weight containers with guaranteed, user-level access to resources;
- User-level scheduling of resources within cells; and
- Adaptive allocation and distribution of resources to cells in order to meet QoS requirements efficiently.

We showed that this approach handles a simultaneous mix of high-throughput parallel, real-time, and interactive applications. Though we restricted our focus to client applications, the embedded systems domain presents a similar complex resource management problem that we believe Tessellation OS is capable of managing. We believe that ARCC is essential for designing systems that can meet the rigorous responsiveness and efficiency demands of modern applications.

7. ACKNOWLEDGMENTS

This research is supported by Microsoft (Award #024263), Intel (Award #024894), matching U.C. Discovery funding (Award #DIG07-102270), and DOE ASCR FastOS Grant #DE-FG02-08ER25849. Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung. No part of this paper represents views and opinions of the sponsors mentioned above. We thank other Par Lab members for their collaboration and feedback. J. A. Colmenares participated in this work while he was a post-doctoral scholar at UC Berkeley. M. Moreto is supported by a MEC/Fulbright Fellowship.

8. REFERENCES

[1] Nano-X window system. http://www.microwindows.org/.

[2] NAS parallel benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[3] L. Abeni and G. Buttazzo. Resource reservations in dynamic real-time systems. *Real-Time Systems*, 27(2):123–165, 2004.

[4] B. Akesson et al. Predator: a predictable SDRAM memory controller. In *Proc. of CODES+ISSS*, 2007.

[5] P. Barham et al. Xen and the art of virtualization. In *Proc. of SOSP*, 2003.

[6] D. B. Bartolini et al. The Autonomic Operating System Research Project Achievements and Future Directions. In *Proc. of DAC*, 2013.

[7] S. Baruah et al. Implementing constant-bandwidth servers upon multiprocessors. In *Proc. of RTAS*, 2002.

[8] S. Baruah and G. Lipari. Executing aperiodic jobs in a multiprocessor constant-bandwidth server implementation. In *Proc. of ECRTS*, 2004.

[9] A. Baumann et al. The Multikernel: A new OS architecture for scalable multicore systems. In *Proc. of SOSP*, 2009.

[10] S. Boyd-Wickizer et al. Corey: an operating system for many cores. In *Proc. of OSDI*, 2008.

[11] J. A. Colmenares et al. Resource management in the Tessellation manycore OS. In *Proc. of HotPar*, 2010.

[12] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of SOSP*, 1995.

[13] P. Fischer. Multicore processors revolutionize real-time embedded systems. *Electronic Design*, December 2007.

[14] L. L. Fong et al. Gang scheduling for resource allocation in a cluster computing environment. Patent US 6345287, 1997.

[15] A. Gulati et al. mClock: handling throughput variability for hypervisor IO scheduling. In *Proc. of OSDI*, 2010.

[16] S. M. Hand. Self-paging in the nemesis operating system. In *In Proc. of OSDI*, 1999.

[17] H. Hoffmann et al. SEEC: a general and extensible framework for self-aware computing. Technical Report MIT-CSAIL-TR-2011-016, 2011.

[18] K. Jeffay. Scheduling sporadic tasks with shared resources in hard real-time systems. In *Proc. of RTSS*, 1992.

[19] A. Kim et al. A soft real-time parallel GUI service in Tessellation many-core OS. In *Proc. of CATA*, 2012.

[20] A. Kivity. kvm: the Linux virtual machine monitor. In *Proc. of OLS*, 2007.

[21] H. Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer, 1997.

[22] E. A. Lee et al. The TerraSwarm Research Center (TSRC) (A White Paper). Technical Report UCB/EECS-2012-207, EECS Department, University of California, Berkeley, Nov 2012.

[23] J. W. Lee et al. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. *SIGARCH Comput. Archit. News*, 36(3):89–100, June 2008.

[24] B. Leiner et al. A comparison of partitioning operating systems for integrated systems. In *Proc. of SAFECOMP*, 2007.

[25] J. Liedtke. On micro-kernel construction. *ACM SIGOPS Oper. Syst. Rev.*, 29:237–250, December 1995.

[26] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[27] R. Liu et al. Tessellation: Space-time partitioning in a manycore client OS. In *Proc. of HotPar*, 2009.

[28] L. Luo and M.-Y. Zhu. Partitioning based operating system: a formal model. *ACM SIGOPS Oper. Syst. Rev.*, 37(3), 2003.

[29] K. J. Nesbit et al. Multicore resource management. *IEEE Micro*, 28(3):6–16, 2008.

[30] R. Obermaisser and B. Leiner. Temporal and spatial partitioning of a time-triggered operating system based on real-time Linux. In *Proc. of ISORC*, 2008.

[31] J. Ousterhout. Scheduling techniques for concurrent systems. In *Proc. of ICDCS*, 1982.

[32] P. Padala et al. Automated control of multiple virtualized resources. In *Proc. of EuroSys*, 2009.

[33] H. Pan et al. Composing parallel software efficiently with Lithe. In *Proc. of PLDI*, 2010.

[34] M. P. Papazoglou and W.-J. Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, July 2007.

[35] B. Rhoden et al. Improving per-node efficiency in the datacenter with new os abstractions. In *Proc. of SOCC*, 2011.

[36] J. Rushby. Partitioning for avionics architectures: requirements, mechanisms, and assurance. Technical Report CR-1999-209347, NASA Langley Research Center, June 1999.

[37] B. Saha et al. Enabling scalability and performance in a large scale CMP environment. In *Proc. of EuroSys*, 2007.

[38] D. Sanchez and C. Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. *SIGARCH Comput. Archit. News*, 39(3):57–68, June 2011.

[39] A. Sharifi et al. METE: meeting end-to-end qos in multicores through system-wide resource management. *SIGMETRICS Perform. Eval. Rev.*, 39(1):13–24, June 2011.

[40] D. D. Silva et al. K42: an infrastructure for operating system research. *SIGOPS Oper. Syst. Rev.*, 40(2):34–42, 2006.

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

APPENDIX

A. CELL TYPES

Table 1 summarizes the cell types Tessellation provides. Non-multiplexed (non-muxed) cells are intended to host software components with stringent performance requirements that demand a high degree of performance isolation. Time-triggered (TT) cells are for hosting time-predictable components that can tolerate some latency. Event-triggered (ET) cells provide flexible event-handling as well as good responsiveness and resource utilization, which make them ideal for hosting OS services. Finally, best-effort (BE) cells are for components without strict timing constraints.

These time-multiplexing policies allow users to easily specify the desired timing behavior for cells with a certain precision (currently 1 ms). For example, TT and ET cells both take the parameters *period* and *active_time*, where $period > active\_time$. In the case of an ET cell, its reserved fraction of processing time is given by ($active\_time/period$).

The Partition Multiplexing Layer in the Tessellation kernel runs a separate multiplexer on each hardware thread of the system. The multiplexers, or *muxers*, control time-multiplexing of cells. The muxers collectively implement gang-scheduling using a variant of *Earliest Deadline First* (EDF) [26], combined with the *Constant Bandwidth Server* (CBS) [3] reservation scheme. The Partition Multiplexing Layer is then capable of providing all of the scheduling behaviors shown in Table 1. We chose EDF for TT cells because it enables the muxers to directly utilize the timing parameters specified for these cells. CBS, on the other hand, isolates each ET cell from other cell activations, and ensures each ET cell a fraction ($f = active\_time/period$) of processing capacity on each hardware thread assigned to the cell. Further, CBS offers ET cells responsiveness by allowing them to exploit the available slack without interfering with other cells. For short activation time (*e.g.,* for event processing in server cells), an ET cell is activated with an immediate deadline if it has not used up its time allocation.

Muxers could schedule BE cells using a hierarchical scheme, in which CBS reserves a small fraction of processing capacity for BE cells and a round-robin algorithm operates in the time slices given by CBS. For implementation simplicity, however, the muxers use only CBS to schedule BE cells. Unlike ET cells, which are activated by events, BE cells are always kept in the runnable queue. Each BE cell is given a fixed small reservation (*e.g.,* 2% with $active\_time = 5$ ms and $period = 100$ ms) to ensure that it always makes progress.

B. OS SERVICES: TWO CASE STUDIES

Tessellation OS builds upon a service-oriented architecture (see Section 2.2), in which services encapsulate devices and, in general, resources. Each service can leverage its cell's performance isolation and customizable user-level schedulers to offer service guarantees to applications and services residing in other cells. Moreover, services may exploit parallelism to reduce service times or increase service throughput.

Each service in Tessellation comes with a library to facilitate the development of client applications. The client libraries offer friendly, high-level application programming interfaces (APIs) to manage connections and interact with the services (*i.e.,* they hide most of the details of inter-cell channel communication). Those libraries also allow applications to request the QoS guarantees they need from services.

Two services at a mature development state that provide QoS guarantees are the *Graphical User Interface (GUI) Service* and the *Network Service*. They are the focus of this appendix and we describe them in detail below. Additionally, Tessellation offers a *Console Service* that prints character strings from other cells to a serial console (via a serial port). This service has exclusive access to the console device, and for a client cell, printing a string is just sending a one-way message on a dedicated channel to the console service. Thus, cells do not need to contend and wait for accessing the console device, and the influence of printing console messages on each application's behavior can be controlled independently and minimized. The Console Service has been instrumental in collecting the experimental data presented in this paper.

Other services in active development include a Block Device Service and an Object Store Service. The Block Device Service provides block-level operations with additional QoS guarantees to applications or other services, and behaves as a swap pager for individual cells. The Object Store Service provides persistent storage through a simple interface that enables applications to put (and retrieve) variable sized objects in a flat namespace. Due to space constraints, we do not discuss these services further.

Next, we discuss the implementation of Tessellation's GUI Service and Network Service, and examine their ability to provide QoS guarantees to client cells.

B.1 GUI Service

Tessellation's GUI Service [19] provides a windowing system with response-time guarantees for visual applications. It resides in a dedicated cell, which encapsulates and has sole control of the framebuffer device. Therefore, applications can only draw to the screen by using the GUI Service.

The GUI Service is a rearchitected version of the Nano-X Window System [1]. It exploits a user-level *Earliest Deadline First* (EDF) scheduler to take advantage of multiple cores and ensure that rendering jobs with earlier deadlines are scheduled sooner. The GUI Service supplements the EDF scheduler with a *resource reservation scheme*, called Multiprocessor Constant Bandwidth Server (M-CBS) [7, 8], to provide different CPU reservations to different rendering tasks – a big distinction from traditional GUI systems.

We conduct an experiment to evaluate GUI Service's ability to provide QoS guarantees to visual applications. The experiment consists of eight video clients that send 8,000 computationally intensive rendering requests (frames) to the window system, half at a rate of 30 frames per second (fps), and half at 60 fps. To compare their performance, we use both the GUI Service on Tessellation and the original Nano-X system on Linux. The test platform is equipped with an Intel 3.4-GHz Core i7 quad-core processor, 4 GB of RAM, and hyper-threading enabled.

Figure 6 shows the result of our experiment. The traditional GUI system (represented by Nano-X running on Linux) runs on a single hardware thread and misses 65% of deadlines of the 60-fps requests. By contrast, even on one hardware thread, Tessellation misses only 0.1% of deadlines (GUIServ(1) in Figure 6), because it can reallocate some of the CPU reservation from the 30-fps streams. Reallocation is not necessary when the GUI Service uses more than one hardware thread, and the overall service times roughly halve when the hardware-thread count doubles. This suggests that

Cell Type	Description	Gang-Scheduling Algorithm
Non-Multiplexed (Non-Muxed)	The cell is given dedicated access to the hardware threads and the other managed resources.	Permanent activation.
Time-Triggered (TT)	The cell is active for some time during periodic time intervals.	Earliest Deadline First (EDF) [26].
Event-Triggered (ET)	The cell is activated upon the arrival of an event. Once activated, the cell remains "runnable" and is multiplexed with other cells until its user-level runtime requests the cell to yield all the resources via the `cell_yield()` system call. Once the cell yields it does not become runnable until another event arrives.	Constant Bandwidth Server (CBS) [3].
Best-Effort (BE)	These cells have no strong guarantees, but the kernel ensures that they have the chance to be activated (*i.e.,* make progress) and are multiplexed in a fair manner among themselves.	CBS with cells always available for activation and small reservations.

Table 1: Types of cells according to the time-multiplexing policies.

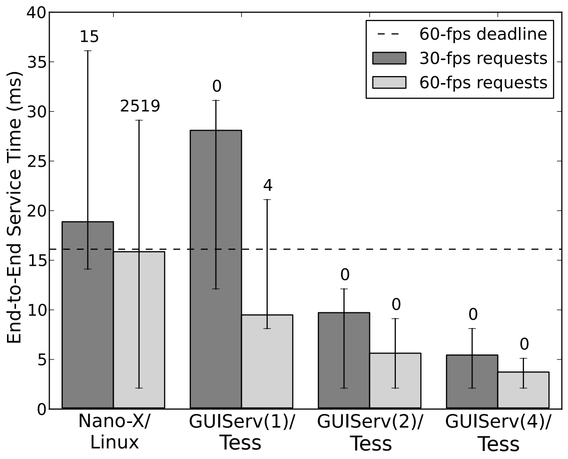

Figure 6: Service time for rendering requests. The numbers above the bars represent missed deadlines; the numbers in parentheses indicate the number of allocated hardware threads in Tessellation.

the GUI Service may scale well.

The GUI Service is a good example of the advantages of two-level scheduling in Tessellation. Implementing the GUI Service's customized scheduler on a monolithic kernel (*e.g.,* Linux) would require extensive kernel-side modifications, especially if we wanted to apply it to a single application. Modifying a general-purpose scheduler to meet the requirements of a specific application easily leads to performance issues for the other applications. However, on Tessellation, no kernel modification is required; the GUI Service's scheduler sits on top of the PULSE framework (Section 4), is only 445 lines of code, runs completely at user level, and applies only to the cell hosting the GUI Service.

So far, we have focused on a basic software-based rendering pipeline, which makes no use of acceleration capabilities in graphical processing units (GPUs). The reason is that using a CPU-based software rendering service is enough for exploring mechanisms for QoS guarantees; GPU acceleration would not add much on this side and its use is left to future work.

B.2 Network Service

Tessellation's Network Service provides access to network interface cards (NICs) through an API similar to the socket API. Its implementation is based on a modified multi-threaded version of the lightweight TCP/IP protocol stack lwIP.[3] The Network Service allows the specification of *minimum throughput reservations* for data flows between NICs and client cells. The service guarantees that the data flows are processed with *at least* the specified levels of throughput, provided it is feasible to do so with the networking and computational resources available to the service (*e.g.,* the aggregate reservation should be less than or equal to the NIC's maximum capacity). Moreover, the Network Service distributes any excess throughput proportionally among the client cells via an adaptation of the mClock algorithm [15].

The Network Service enforces QoS guarantees by restricting channel communication. When an application opens a channel to send or receive data over the network through the Network Service API, it requests both a guaranteed and a proportional bandwidth. The Network Service's access control mechanism denies guaranteed-bandwidth requests that would cause the total guaranteed bandwidth to exceed the NIC's maximum capacity. The Network Service first enforces the request for guaranteed bandwidth and uses the proportional request to assign the slack of capacity. The bandwidth-guarantee enforcement takes place at channel message granularity, where each message contains an individual remote procedure call such as `recv()` or `send()`. While we stopped at the message granularity for implementation simplicity, bringing this enforcement at finer grain (*i.e.,* at channel data granularity) is a straightforward extension.

We evaluate the Network Service in a common client use case: a user wants an uninterrupted stream of video content (*e.g.,* Hulu) in a foreground application, while a background task (*e.g.,* Dropbox) generates bursty network traffic. The video stream requires 125 KB/s, which is the bandwidth of an H.264 480p Hulu stream, while the background application runs periodically and tries to use all the available capacity. To realize this experiment, we use two identical machines, each equipped with an Intel 3.4-GHz quad-core Core i7 processor and 4 GB of RAM. Both machines are directly connected over Ethernet via their Intel Pro/1000 1-Gbps NICs. One machine acts as the video source and runs Linux 3.1.9, while the other, which is the video player to be evaluated, runs Tessellation. The Linux box runs two separate socket applications that serve data respectively to the video-player application and the background application running in Tessellation. On the Tessellation box, each of the two test applications is assigned a dedicated hardware thread, while the network service uses three dedicated hardware threads.

Figure 7 shows the results of this experiment: the foreground application receives an average 125.2 KB/s throughput, while the background application periodically uses

[3]http://savannah.nongnu.org/projects/lwip/

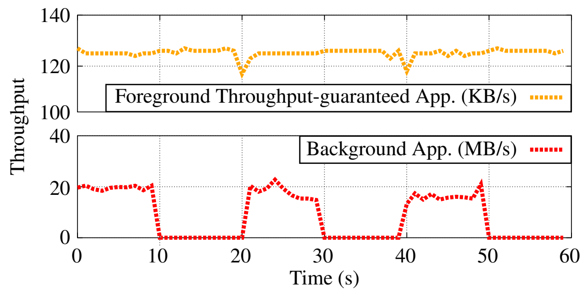

Figure 7: Guaranteeing network throughput in the presence of greedy applications. This graph shows that the greedy application is unable to deny service to the application with guaranteed service.

around 20 MB/s. The slight dip (around 5 KB/s) in the foreground application's throughput that occurs when the background application starts using the link is a result of a limitation in our modified mClock implementation. What happens is that the QoS state of previously dormant connections is out of sync with virtual time, resulting in boosted privilege for a brief instant. This problem has a known solution, described by Gulati et al. [15] but, since the effects of this issue are not disruptive, we did not put additional implementation effort in porting the solution. Despite the mClock artifact of our implementation, we achieve a standard deviation of only 1.65 KB/s for the throughput of the foreground application; this means that the user will not experience disruption in the video-streaming applications due to bandwidth-hungry background tasks.

The Network Service is another example of how the service-oriented architecture of Tessellation can help modern clients match users' needs. One of the key points is that, aside from one-time system calls to configure the NIC, the NIC driver[4] is completely contained in userspace. This structure allows the Network Service to avoid having to make relatively expensive system calls to access the transmit and receive buffers.

C. RELATED WORK

A number of research efforts have focused on the problem of adaptive resource allocation to meet QoS objectives in multi-application scenarios. Some previous work most relevant to this paper includes: AutoControl [32], SEEC [17], AcOS [6], and METE [39]. Next we present a comparison bewteen Tessellation and that previous work focusing on goals, software infrastructure, and underlying adaptation mechanisms.

AutoControl, SEEC, AcOS, METE, and other autonomic computing frameworks incorporate adaptation policies of various types. Policy-related aspects, such as formulation of the resource-allocation problem, resource-allocation decision engines using optimal control or machine learning techniques, and online performance-model estimators, are orthogonal to our discussion here. Those adaptation policies can be implemented in Tessellation's Resource Allocation Broker.

AutoControl [32] is a feedback-based resource-allocation system for shared virtualized infrastructure in data centers, where applications are hosted in virtual machines across multiple nodes. It addresses the resource-management problem at the hypervisor level, and exploits Xen [5] to allocate CPU and disk I/O bandwidth to mitigate bottlenecks. AutoControl and Tessellation have similar characteristics, despite big differences in their target computing platforms – data centers for AutoControl and a single multicore node currently for Tessellation. Tessellation implements dynamic resource allocation at the cell level, and cells resemble some aspects of virtual machines [5, 20]. Cells, however, are intended to provide better performance isolation and more precise control over their timing behavior than traditional virtual machines; besides, cells do not host a complete OS – only a user-level runtime. In addition, AutoControl and Tessellation both consider I/O services as shared resources and dynamically allocate fractions of the services to applications.

SEEC [17] is a self-aware programming model designed to facilitate the development of adaptive computing systems on multicore platforms. It supports a decoupled approach in which application programmers specify the applications' goals and report the current progress toward those goals, while system programmers separately specify the set of actions system software (*e.g.,* OS and runtime) and hardware can take to affect the applications. The SEEC framework implements an observe-decide-act (ODA) control loop to monitor applications and dynamically select actions to optimally meet their goals. SEEC currently supports three application goals: performance, accuracy, and power. As SEEC, Tessellation aims at providing a general and extensible framework for self-adapting computing, in particular through its RAB Service. Similar to SEEC's Application Heartbeat API, the RAB Service provides an API for applications hosted in cells to report their goals and performance over inter-cell channels. SEEC does not propose changes in the OS and currently uses mechanisms available in Linux to implement the actuator functions. On the contrary, our work on Tessellation focuses on rearchitecting the OS to provide better performance predictability and adaptive resource management.

AcOS [6] is a proposal for an autonomic resource-management layer to extend commodity OSs, such as Linux and FreeBSD. The authors investigate the application of autonomic-computing ideas at the OS level to automate resource allocation based on user-specified application performance goals and enforce system-level restrictions. They demonstrate different approaches to automate allocation of cores and processor time in order to meet those goals. Moreover, AcOS considers maximum processor temperature thresholds and implements a dynamic performance and thermal management (DPTM) control loop to cap temperature while still meeting the performance goals of a subset of the applications. With respect to SEEC, the scope of AcOS is closer to Tessellation's. The major difference is that AcOS only focuses on adaptation and extends commodity OSs, whereas Tessellation builds support for adaptive resource allocations into its novel cell model from the ground up. Also, neither AcOS nor SEEC in practice consider OS services as part of the resource-allocation problem.

METE [39] is a platform for end-to-end on-chip resource management for multicore processors. Its main goal is to dynamically provision hardware resources to applications to achieve performance targets. METE leverages feedback control to partition shared hardware resources among

[4]Tessellation currently supports the Intel PRO/1000 PCI (E1000) and Realtek RL8168 adapters.

co-located applications and uses an autoregressive-moving-average (ARMA) model to capture applications' performance characteristics. METE considers cores, shared cache space, and off-chip memory bandwidth as partitionable resources. Since current processors do not support partitioning of all those resources, METE has been evaluated in a simulation environment. Tessellation shares with METE the interest in exploiting hardware partitioning mechanisms to guarantee end-to-end QoS. However, while METE assumes the availability of such mechanisms, Tessellation builds support for resource partitioning in a new OS model to enable experimentation on real hardware. We exploit hardware solutions when available and develop software solutions where hardware support is absent.

Tessellation has similarities to several recent manycore OSs. The use of message-passing communication via user-level channels is similar to Barrelfish [9]. However, Barrelfish is a multikernel OS that assumes no hardware assist for cache coherence, and does not focus on adaptive resource allocation. The way Tessellation constructs user-level services is similar to fos [41]. Services in Tessellation are QoS-aware and cells are partitioned based on applications rather than physical cores. Tessellation is similar to Corey [10] in that we also try to restrict sharing of kernel structures.

Tessellation adopts a microkernel philosophy [25], in which OS services are implemented in user-space and applications interact with them via message passing. Unlike in traditional microkernels, however, each service residing in a separate cell is explicitly parallel and performance-isolated, and includes an independent user-level runtime. The runtime customization in Tessellation is influenced by Exokernel [12]. However, Tessellation tries to mitigate some of the problems of exokernels by providing runtimes and services for the applications. Tessellation has some similarities to the K42 OS [40]. Both implement some OS services in user-space, but K42 uses protected procedure calls (PPCs) to access services, where Tessellation uses user-level channels.

Tessellation shares with the Nemesis OS [16] the emphasis on ensuring QoS for multimedia applications. Nemesis also uses an approach around OS services and message passing, but on uniprocessors.

Resource partitioning has also been presented in McRT [37] and Virtual Private Machines (VPM) [29]. The concepts of VPM and cells are similar, but VPM lacks inter-cell communication and has not been implemented yet. Gang-scheduling [31, 14] is a classic concept and has also been applied to other OSs – most similarly in Akaros [35]. However, unlike other systems, Tessellation supports cells with different timing behaviors.