Building an Adaptive Operating System for Predictability and Efficiency

Gage Eads
Juan Colmenares
Steven Hofmeyr
Sarah Bird
Davide Bartolini
David Chou
Brian Glutzman
Krste Asanovic
John D. Kubiatowicz

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-137
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-137.html

July 7, 2014

Acknowledgement

This research is supported by Microsoft (Award #024263), Intel (Award #024894), matching UC 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 would like to thank other Par Lab members for their collaboration and feedback. Juan A. Colmenares participated in this work first as a post-doctoral scholar and then as a visiting researcher for Samsung at UC Berkeley.

Building an Adaptive Operating System for Predictability and Efficiency

Gage Eads[♦], Juan A. Colmenares[♭♦], Steven Hofmeyr[†], Sarah Bird[♦], Davide B. Bartolini[§] ,
David Chou[♦], Brian Gluzman[♦], Krste Asanović[♦], John D. Kubiatowicz[♦]

[♦]The Parallel Computing Laboratory, UC Berkeley, CA, USA

[†]Lawrence Berkeley National Laboratory, Berkeley, CA, USA

[♭]Samsung Research America - Silicon Valley, San Jose, CA, USA

[§]Politecnico di Milano, Italy

geads@eecs.berkeley.edu, juan.col@samsung.com, shofmeyr@lbl.gov,
slbird@eecs.berkeley.edu, davide.bartolini@polimi.it,
davidchou.24@gmail.com, brian.gluzman@berkeley.edu,
{krste, kubitron}@eecs.berkeley.edu

Abstract

We address the system-level challenge of supporting a dynamically changing, complex mix of simultaneously running applications with diverse requirements including responsiveness, throughput, and performance guarantees. In our approach, called Adaptive Resource Centric Computing (ARCC), the OS distributes resources to QoS domains called cells, which are explicitly parallel lightweight containers with guaranteed, user-level access to resources. The resource allocations are dynamically adjusted whenever applications change state or the mix of applications changes. This paper gives explicit details about our implementation of ARCC on Tessellation OS, an experimental platform for resource management on multicore-based computers. We discuss the implementation of cells, user-level scheduling, and resource allocation policies. Our running example is a realistic video conferencing scenario that incorporates parallelism, QoS guarantees, and dynamic optimization with two-level scheduling. Our system reduces reserved CPU bandwidth to 69.95% of that of a static allocation, while still meeting performance and QoS targets.

1. Introduction

Today's users expect snappy operation from high-quality multimedia and interactive applications, which need to both provide responsive user interfaces and meet stringent real-time guarantees. Moreover, the need for data security and accessibility leads to a number of compute-intensive applications, such as anti-virus scanners or file indexers, executing in the background and possibly interfering with interactive and multimedia applications. An open research challenge is understanding how an operating system (OS) should best support this dynamically changing and complex mix of applications. Addressing this challenge means being able to satisfy quality of service (QoS) requirements while making efficient use of computing resources — a potentially complex optimization problem.

For example, consider the multi-stream, multi-resolution video conference in Figure 1. This scenario integrates video streams from multiple sites, with one site featured by a larger, higher-resolution image. An outgoing video must be also encoded and forwarded to each of the remote sites. New participants may join the conference and others leave, increasing or decreasing the number of streams running at any given time. While conferencing, participants may collaborate through browsers, watch shared video clips and search the web; moreover, compute-intensive tasks, such as virus scans or file indexing, could run in the background. Each individual participant introduces a need for a separate, performance guaranteed video stream and sufficient computation to render and composite an image on the shared video screen.

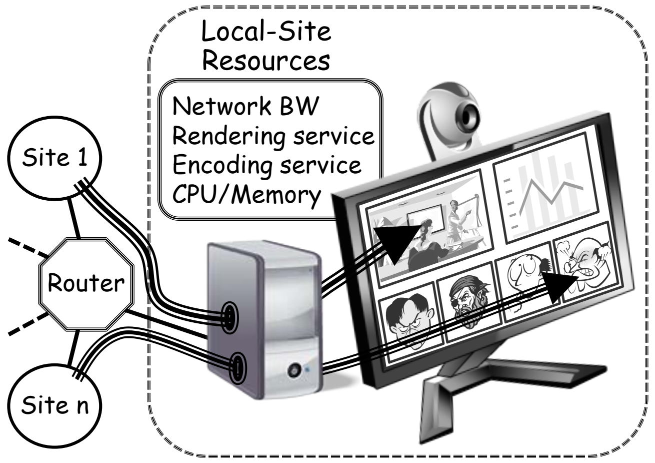


Figure 1: Multi-stream, multi-resolution video conferencing with resource-hungry background tasks. Glitch-free behavior requires both physical guarantees (*e.g.,* CPU, memory, packet bandwidth) and service-level guarantees (*e.g.,* rendering and encoding performance, remote network reservations). The challenge is choosing appropriate resource allocations.

Although providing QoS guarantees to video streams by over-provisioning resources may be relatively straightforward, the real challenge is to do so without using static resource reservations that compromise system utilization or needlessly drain energy. *Resources*, in this context, include *physical resources* such as CPU cycles or energy and *services* such as file storage or network communication. Ideally, applications with strict performance requirements should be given just enough resources to meet these requirements consistently, without siphoning resources from other applications. Further, unused resources should be identified and deactivated to save energy.

Unfortunately, applications such as video encoding exhibit a complex relationship between resources and performance. Consequently, dynamic, in-situ profiling must be part of any solution. Further, the diffusion of multicore processors complicates this task by adding on-chip spatial multiplexing and shared resources that lead to contention and interference between co-running applications. Parallelizing applications is not enough for performance; the system must balance resources among competing tasks.

In the following pages, we tackle this system-level optimization problem with a resource-centric approach that we call Adaptive Resource Centric Computing (ARCC) [20]. Our approach involves both *dynamic resource allocation* (adaptive assignment of resources to applications) and *QoS enforcement* (preventing interference between components). We illustrate our solution in the context of *Tessellation OS* [21, 20, 22], an experimental platform for resource management on multicore computers. In Tessellation, resources are distributed to QoS domains called *cells*, which are explicitly parallel, light-weight containers with guaranteed, user-level access to resources. Further, composite resources are constructed by wrapping cells around existing resources and exporting service interfaces with QoS contracts. Cells provide our essential mechanism for QoS enforcement. To reduce the burden on the programmer and to respond to changes in the environment, we automatically adjust resource allocations to meet application requirements; this framework for adaptation is one of our contributions.

Scheduling within cells functions purely at the user-level, as close to the *bare metal* as possible, improving efficiency and eliminating unpredictable OS interference. Our framework for preemptive scheduling, called *Pulse*, enables customization and support for a wide variety of application-specific runtimes and schedulers without kernel-level modifications. Pulse is highly efficient; for instance we wrote an Earliest Deadline First (EDF) scheduler [34] that runs entirely in userspace in about 800 lines of code.[1] In addition to support for timer interrupts, the Pulse API provides callbacks for adaptation events to notify schedulers when the number of available cores changes. These notifications permit resource-aware, application-specific management, which is impossible with a centralized OS approach. This capability eliminates the need to build a *one-size-fits-all* scheduler, thus sidestepping a difficult design challenge [2].

[1] In contrast, the best-known EDF kernel patch for Linux (SCHED_DEADLINE) has over 3500 modified lines in over 50 files [9].

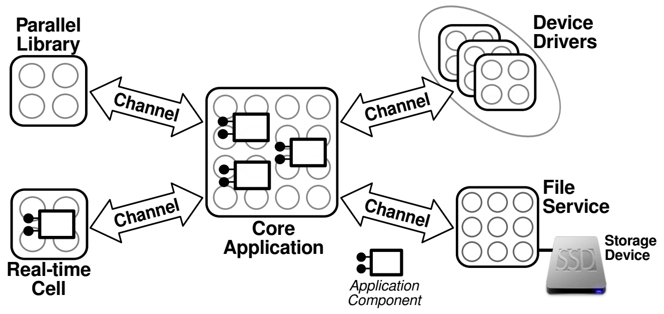


Figure 2: Applications in Tessellation are created as sets of interacting components hosted in different cells that communicate over channels. Standard OS services (*e.g.,* the file service) are also hosted in cells and accessed via channels.

By separating resource management into dynamic allocation and QoS enforcement, we are essentially adopting *two-level scheduling* [32, 37, 21]. While two-level scheduling has been investigated in the past, leveraging this concept to address the issues that emerge in the dynamic execution scenarios we outlined in this introduction raises complex challenges that are often underestimated. For this reason, we evaluate a complete, detailed solution that incorporates parallelism, QoS guarantees, and dynamic optimization with two-level scheduling. We directly address many of the issues that arise from this challenge, such as timing for interacting control loops, application resizing, and efficient gang scheduling.

We utilize the multi-stream video conference from Figure 1 as a running example throughout this paper. In one experiment, our system is able to reduce CPU bandwidth to 69.95% of that of a static allocation (efficient resource usage), while meeting the performance target.

2. Tessellation OS: An ARCC Instance

This section briefly describes the key components of Tessellation OS [21, 20, 22]. The Tessellation kernel is a thin, hypervisor-like layer that provides support for ARCC. It implements cells along with interfaces for user-level scheduling, resource adaptation, and cell composition. Tessellation currently runs on x86 hardware platforms (*e.g.,* with Intel's Sandy Bridge CPUs).

2.1. The Cell Model

Cells are the basic unit of computation and protection in Tessellation. They 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 cell's resources (*e.g.,* CPU cores, memory pages, and I/O devices).

As depicted in Figure 2, applications in Tessellation are created by composing cells via *channels*, which provide fast, user-level asynchronous message-passing between cells. Applications can then be split into performance-incompatible and mutually distrusting cells with controlled communication. Cells provide our basic mechanism for QoS enforcement;

when combined with adaptive resource allocation, they provide a complete platform on which to build a multi-application environment.

2.2. Implementing Cells

Tessellation OS implements cells on x86 platforms by partitioning resources using *space-time partitioning* [44, 35], a multiplexing technique that divides the hardware into a sequence of simultaneously-resident spatial partitions. Cores and other resources are *gang-scheduled* [38, 24], so cells provide to their hosted applications an environment that is very similar to a dedicated machine.

Partitionable resources include CPU cores, memory pages, 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 [12, 31, 41, 46]. Section 3 provides details about our implementation of cells in Tessellation.

The user-level runtime within each cell can be tuned for a specific application or application domain with a custom scheduling algorithm. Using our user-level scheduler framework, Tessellation provides pre-canned implementations for TBB [42] and a number of scheduling algorithms, including Global Round Robin (GRR), Earliest Deadline First (EDF), and Speed Balancing [27]. Others may be easily constructed if necessary. Section 4 provides a detailed discussion of user-level scheduling in Tessellation and the Pulse API.

2.3. Service-Oriented Architecture

Cells provide a convenient abstraction for building OS services with QoS guarantees. Such services reside in dedicated cells, have exclusive control over devices, and encapsulate user-level device drivers. Each service can thus arbitrate access to its enclosed devices, and leverage its cell's performance isolation and customizable scheduler to offer service guarantees to other cells. Services can shape data and event flows coming from external sources with unpredictable behavior and prevent other cells from being affected. In keeping with ARCC, Tessellation treats the services offered by such *service cells* as additional resources, and manages them with its adaptive resource allocation architecture.

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 with the agreed levels of throughput; and the *GUI Service*, which provides a windowing system with response-time guarantees for visual applications. These services are utilized in our macro-benchmarks measurements in Section 6.

2.4. Adaptive Resource Allocation

Tessellation uses an adaptive resource allocation approach to assign resources to cells. This functionality is performed by the *Resource Allocation Broker (RAB)*, as shown in Figure 3.

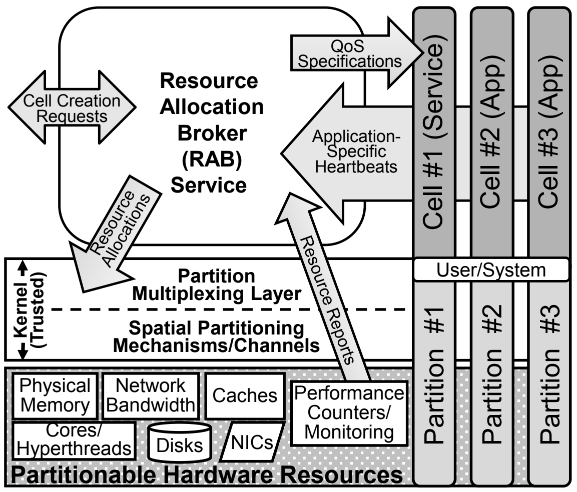


Figure 3: The Tessellation kernel implements *cells* through *spatial-partitioning*. The *Resource Allocation Broker (RAB)*redistributes resources after consulting application-specific *heartbeats* and system-wide *resource reports*.

The RAB distributes resources to cells while attempting to satisfy competing application performance targets and system-wide goals, such as deadlines met, energy efficiency, and throughput. It utilizes resource constraints, application models, and current performance measurements as inputs to this optimization. Allocation decisions are communicated to the kernel and services for enforcement. The RAB reallocates resources, for example, when a cell starts or finishes or when a cell significantly changes performance. It can periodically adjust allocations; the reallocation frequency provides a trade-off between adaptability (to changes in state) and stability (of user-level scheduling).

The RAB provides a resource allocation framework that supports rapid development and testing of new allocation policies. This framework enables us to explore the potential of an ARCC-based OS for providing QoS to individual applications while optimizing resource distribution to achieve global objectives. In Section 5, we discuss two policies we have implemented to demonstrate this potential.

3. Space-Time Partitioning Implementation

As shown in Figure 3, 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 via page coloring). Building on this support, the Mux Layer implements space-time partitioning and translates resource allocations from the RAB into an ordered time sequence of spatial partitions.

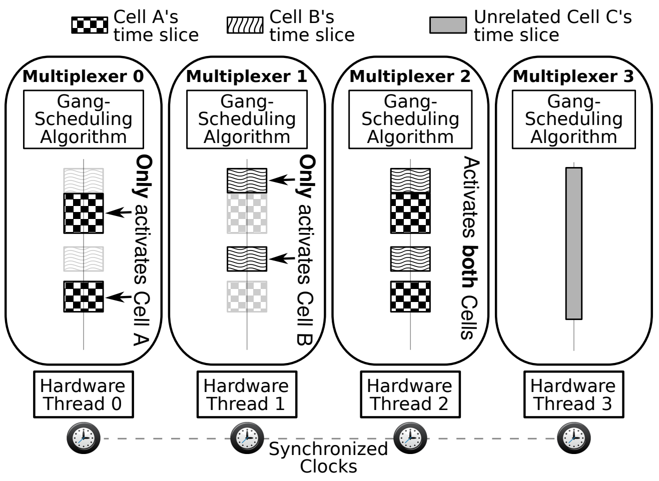


Figure 4: Operation of the Mux Layer: Gang scheduling of overlapping time-triggering cells A and B and an independent cell C. Cell A is assigned to hardware threads {0,2} and cell B to hardware threads {1,2}. Hardware thread 3 is dedicated to cell C. Multiplexers 0, 1, and 2 produce same schedule, but only activate the cells allocated to their hardware threads. Multiplexer 3, on the other hand, does not require knowledge of other multiplexers to schedule its cell.

3.1. Types of Cells

The Mux Layer offers several time-multiplexing policies for cells, some of them offering high degrees of time predictability; they are: 1) *non-multiplexed* (dedicated access to its assigned resources), 2) *time triggered* (active during predetermined and periodic time windows), 3) *event triggered* (activated upon event arrivals, with an upper bound on total processor utilization), and 4) *best effort* (without time guarantees).

These multiplexing policies allow users to specify desired timing behavior for cells within a certain precision (currently 1 ms). For example, time-triggered and event-triggered cells both take the parameters *period* and *active_time*, where $period > active\_time$; for an event-triggered cell, $(active\_time/period)$ is its reserved processing-time fraction. The Mux Layer then ensures that, if feasible, a set of cells with different multiplexing policies harmoniously coexist and receive their specified time guarantees. In this way, Tessellation offers precise control over cells' timing behavior, a characteristics that differentiates Tessellation from traditional hypervisors and virtual machine monitors [14, 29].

3.2. Cell Multiplexers and Gang Scheduling

The Mux Layer in the Tessellation kernel runs a separate multiplexer on each hardware thread of the system, as depicted in Figure 4. The multiplexers, or *muxers*, control the time-multiplexing of cells, and collectively implement gang scheduling [38, 24] in a *decentralized* manner. They execute the *same* scheduling algorithm and rely on a high-precision global-time base [30] to simultaneously activate a cell on multiple hardware threads with minimum skew. Our prototype takes advantage of Intel's TSC-invariant feature[2] and the fact that LAPIC timers are all driven by the bus frequency to have access to a high-precision global time. In the common case, the muxers operate independently and do not communicate to coordinate the simultaneous activation of cells.

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 tries to avoid such unfavorable mappings.

Muxers communicate in the *uncommon case* of events that change the scheduling of cells. Such events include when a cell yields its resources, when a device interrupt requires the activation of an event-triggered cell that has previously yielded its resources, and when the RAB requests a redistribution of resources among cells (see Section 3.3). The muxer that first receives a given event initiates the communication by propagating the event to the other (related) muxers via inter-processor interrupt (IPI) multicast. Communication between muxers can leverage message passing depending on architectural support, but often proceeds via shared memory.

The scheduling algorithm implemented by the muxers is *tickless*; *i.e.,* timer interrupts are issued only when necessary. For example, in the common case, a cell with a non-preemptive user-level runtime (*e.g.,* based on Lithe [40]) running purely computational code is interrupted only when it is to be multiplexed. A tickless environment minimizes both the direct cost (cycles spent in the kernel) and indirect cost (cache pollution) of timer-interrupt overhead [13]. For a cell with a preemptive user-level runtime, muxers make timer interrupts occur more often (*e.g.,* periodically) during the cell's timeslices to trigger user-level runtime's scheduling logic (see Section 4).

The muxers implement gang-scheduling for all of the cell types mentioned in Section 3.1 using a variant of *Earliest Deadline First* (EDF) [34], combined with the *Constant Bandwidth Server* (CBS) [11] reservation scheme. We chose EDF for time-triggered cells because it enables the muxers to directly utilize the timing parameters specified for these cells. CBS, on the other hand, isolates each event-triggered cell from other cell activations, and ensures the cell a fraction $(f = active\_time/period)$ of processing capacity on each hardware thread assigned to the cell. Further, CBS offers event-triggered 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 service cells), an event-triggered cell is activated with an immediate deadline if it has not used up its time allocation.

For implementation simplicity, the muxers use CBS to schedule best-effort cells. Unlike event-triggered cells, best-

[2] The *invariant* timestamp counter (TSC) runs at a constant rate.

```
1  void make_sched_decision(local_sched, cur_time) {
2    update_runnable_Q(local_sched, cur_time)
3    ...
4    cell_2_activate =
5      get_cell_2_activate(local_sched,
6                          cur_time)
7    next_alarm =
8      when_is_next_alarm(cell_2_activate,
9                         local_sched)
10   set_alarm_at(next_alarm)
11   send_end_of_interrupt()
12   switch_cell(cell_2_activate)
13 }
```

Listing 1: Multiplexer's function for scheduling cells.

effort cells are always kept in the runnable queue. Each best-effort 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.

Listing 1 shows pseudo-code for the multiplexer scheduling function. It is called by kernel's interrupt-handler functions (*e.g.,* the timer interrupt handler) after disabling interrupts and suspending the currently active cell (if any). The function first calls `update_runnable_Q(...)` that embeds EDF/CBS logic; this helper function updates the time-accounting variables of the cells that were just interrupted, as well as the scheduler's runnable queue. Next, in line 5 it determines the cell to activate or the need to leave an empty timeslice on the local hardware thread; in making this decision it considers cells that have been assigned to the local hardware thread and those overlapping cells that have not (see Figure 4). Then, it obtains and sets the time for the next timing interrupt in lines 8 and 10, respectively. Finally, the function signals the end of interrupt handling (line 11), and calls the non-returning function `switch_cell(...)`, which activates a cell if given, or halts the hardware thread otherwise.

We evaluate Tessellation's multiplexing accuracy by running a CPU-intensive application kernel in multiple time-multiplexed cells (*i.e.,* time-triggered, event-triggered, and best-effort cells). Each cell is run alone first to determine its performance in isolation, and then multiplexed with other cells on the same hardware threads. We observe that a time-triggered cell with 25% hardware thread reservation achieves near identical performance when run alone or with other cells. Similarly, an event-triggered cell with a 20% reservation runs the same when alone, with a best-effort cell, or with a time-triggered cell with 25% reservation. As expected, two best-effort cells scheduled on the same hardware threads each achieve 50% of their performance when run alone. Consequently, the Tessellation kernel effectively isolates CPU-bound cells sharing the same hardware threads.

We also measured Tessellation's gang-scheduling skew in a 3-core cell (with hyper-threading disabled) across 10,000 wake ups. At the cell entry point, each thread records its timestamp counter (TSC). Since the system has synchronized TSCs, we measure skew as the largest difference between any

	Max	Min	Average	Std. Dev.
TSC cycles	4980	148	2721	843
microseconds	1.468	0.044	0.802	0.249

Table 1: Gang scheduling skew results.

	Max	Min	Average	Std. Dev.
TSC cycles	29462	3600	7197	1018
microseconds	8.685	1.061	2.122	0.300

Table 2: Time multiplexing latency results.

two TSC values at each wake up point. On average, we found sub-microsecond gang-scheduling skew (see Table 1) and low time-multiplexing latency (see Table 2).

3.3. Redistributing Resources among Cells

To request a redistribution of privileged resources among cells (*e.g.,* resizing cells, changing timing parameters, or starting new cells), the RAB passes the new distribution to the Mux Layer via the `update_cells(...)` system call (only accessible to this service). The Tessellation kernel responds to requested changes by altering the scheduling of cells.

To implement 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. Next, it resets and prepares the inactive schedulers, and establishes the *global* time in the near future (with a safety margin of at least 1 ms) at which the muxers will synchronously swap their active and inactive schedulers. Until the swap occurs, no other adaptation event can take place. At the specified time, the muxers swap 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 switch.

Right before the muxers simultaneously switch their schedulers, each muxer suspends the active cell on its hardware thread (if any). Upon the first activation of each cell after the scheduler exchange, the kernel sets the *resource-redistribution event flag* for the cell if its hardware-thread count has changed. This alerts the cell's second-level scheduler to take necessary action, such as migrating threads or creating new ones, in response to the resource-redistribution event. This reactive adaptation process is discussed in detail in Section 4.4.

While running our experiments, we have observed that `update_cells(...)` returns on average after 30 μs, while the entire process described above often takes less than 2 ms with a safety margin of 1 ms for the scheduler exchange.

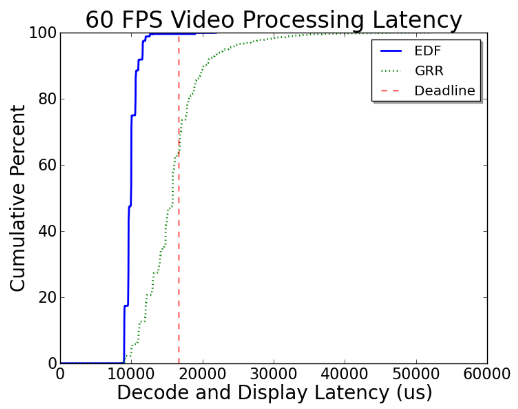


Figure 5: Cumulative distributions of frame processing latency for the 60-fps video with the global round-robin (GRR) and global early-deadline-first (EDF) schedulers.

4. Custom and Adaptive User-Level Schedulers

In Tessellation, scheduling *within* cells occurs purely at the user-level, as close as possible to the *bare metal*, to improve efficiency and eliminate OS interference. To make user-level scheduling practical for application developers, we developed *Pulse*—an efficient framework for constructing application-specific schedulers without kernel-level modifications.

In this section, we start by illustrating how applications can benefit from using a custom scheduler, as opposed to a one-size-fits-all scheduler; then, we show how to build schedulers with the Pulse framework and illustrate how Pulse handles preemption and core allocation/revocation.

4.1. Customizing a User-Level Scheduler

The search for a single, optimal OS scheduler is a hot topic in the OS community (*e.g.,* refer to the CFS versus BFS debate in Linux [2]). The advantage of Tessellation's user-level scheduling approach is that each cell can independently define its ideal scheduler, thus dispensing with the need for a complex global scheduler. In the following experiment, we highlight the advantages of choosing the "correct" scheduler.

We run a real-time video decoder on Tessellation that operates on two H.264 videos with different characteristics, using one thread per video. The videos have a resolution of 640x360 and 750x430, respectively, and we require the two videos to play at 60 and 30 frames per second (fps), respectively. The video decoder uses FFMPEG's libavcodec library [6], and we run the experiment on a Intel Core i7-3770 desktop.

We measured the number of deadlines missed (*i.e.,* decoding latency that exceeds, respectively, 1/60 and 1/30 of a second) with a global round-robin (GRR) scheduler versus a global EDF scheduler. The EDF scheduler is a natural fit, as it allows one to specify the period and deadline for both decoding tasks, while the round-robin scheduler multiplexes runnable threads with a 1-ms timeslice, regardless of deadline. Figure 5 shows the cumulative distributions of frame processing latency for both schedulers for the 60-fps video.

```
void init(num_threads, thread_func) {
  initialize(global_q)
  for (i = 0; i < num_threads; i++) {
    threads[i] = pulse_thread_alloc(stack_size)
    pulse_ctx_make(threads[i], thread_func, i)
    push(global_q, threads[i])
  }
  active_threads = num_threads
  pulse_sched_reg(sched_callbacks)
}

void enter() {
  run_next_thread()
}

void tick(ctx) {
  pulse_ctx_copy(threads[vhart_id()], ctx)
  push_lock(global_q, threads[vhart_id()])
  run_next_thread()
}

void yield(ctx) {
  pulse_ctx_copy(threads[vhart_id()], ctx)
  push_lock(global_q, threads[vhart_id()])
  run_next_thread()
}

void done() {
  atomic_dec(active_threads)
  pulse_thread_free()
  run_next_thread()
}

void run_next_thread() {
  while (atomic_read(active_threads) > 0) {
    ctx = pop_lock(global_q)
    if (ctx != NULL) {
      pulse_ctx_restore_enable_ticks(ctx)
    }
    pulse_sched_ctx_done()
  }
}

void adapt(old_n, new_n, ctx[]) {
  if (vhart_id() != 0) return
  for (i = 0; i < old_n; i++) {
    pulse_ctx_copy(threads[i], ctx[i])
    push(global_q, threads[i])
  }
}
```

Listing 2: Simple Round Robin scheduler using Pulse. Functions that are underlined are implemented in the Pulse library.

The EDF-scheduled video decoder misses no deadline for the 30-fps video, and 0.4% of the deadlines for the 60-fps video. The GRR-scheduled decoder, on the other hand, misses 7.3% and 36.3% of the 30- and 60-fps videos' deadlines, respectively. In this experiment, the EDF scheduler behaves much better with the video application because it exploits the notion of deadline. However, the EDF scheduler is difficult to use for applications without natural deadlines making round-robin schedulers necessary in many situations. Thus, cell-specific user-level schedulers are an advantage of Tessellation over traditional OS designs.

4.2. Writing Application Schedulers

Tessellation's preemptive user-level scheduling framework, Pulse, makes it easy to implement schedulers by taking care of all the details of: 1) interfacing with the kernel, 2) mapping user threads to hardware threads (harts), and 3) ensuring that application schedulers see consistent state when interrupted by adaptation events (*i.e.,* changes to the resources available to a cell). Application schedulers link with the Pulse library and must implement five callbacks:

- `enter()` is called when code first runs on a hart;
- `tick(ctx)` is called on every timer tick and is passed the active application thread context when the interrupt occurred;
- `yield(ctx)` is called when an application thread yields and is passed the application thread context that yielded;
- `done()` is called when an application thread terminates;
- `adapt(old_n, new_n, ctx[])` is called on an adaptation event and is passed the number of harts before the event (`old_n`), the number of harts after the event (`new_n`), and an array of the application thread contexts that were running on the original harts before the event.

Listing 2 shows pseudo-code for a simple Round Robin scheduler. The application calls the `init` function (line 1) in the `main` function, which runs on a single hart before the OS starts code running on all cores. Pulse's helper functions `pulse_thread_alloc`, `pulse_ctx_make` and `pulse_thread_free` serve to manage relevant data structures, that are pushed onto a global queue. Finally the scheduler is registered with Pulse via the `pulse_sched_reg` call.

The `enter` callback (line 12) is initially executed in parallel on each hart. The scheduler calls `run_next_thread` (line 34), which tries to pop a thread off the global queue and run it using the `pulse_ctx_restore` function provided by Pulse. The `tick` (line 16) and `yield` (line 22) callbacks save the application context to a scheduler data structure indexed by the hart number (`vhart_id()`), push it onto the global queue, and call `run_next_thread`.

Note that the functions used to manipulate the global queue, `push_lock` and `pop_lock`, synchronize using a global lock (not shown in the code). All locks in the scheduler implementation must use synchronization functions provided by Pulse because of the way adaptation is handled, as explained in Section 4.4. Another requirement for adaptation support is that the scheduler code should never spin without yielding control to Pulse; this is the purpose of the `pulse_sched_ctx_done` function at line 40.

Adaptation is simple for the global Round Robin scheduler. In the `adapt` callback (line 44), a single scheduler thread (on vhart 0) iterates through the provided application contexts, copies them to a local array, pushes pointers to them into the global queue, and returns to Pulse. If the number of harts has decreased, there will be fewer scheduler threads running, but since this simple implementation only depends on having one scheduler thread per hart, the change will not affect the implementation. If the number of harts increased, then new scheduler threads will be started at the `entry` callback.

While we illustrated this global Round Robin scheduler for its simplicity, Pulse allows efficient implementation of more complex scheduling algorithms. For example, we implemented a user-level EDF scheduler with support for priority inheritance synchronization mechanisms (avoiding the pitfall of priority inversion) in about 800 lines of C code.

4.3. Preemption Handling in Pulse

When a timer event occurs, the kernel saves the context of the interrupted task to userspace and clears an atomically accessible, user-writable `TICKS_ENABLED` flag before switching to a userspace handler.[3] The kernel allows the user-level scheduler to decide when it should re-enable ticks; Pulse currently does not support re-entrant schedulers, so it expects that ticks will remain disabled until the scheduler switches to an application context. Pulse provides a context switching function, `pulse_ctx_restore_enable_ticks`, that switches to an application context and re-enables ticks (line 38 in Listing 2).

Since scheduler threads can migrate to different hardware threads during adaptation events, each `TICKS_ENABLED` flag is specific to a scheduler thread. We currently use the FS segment register on x86 architectures to hold this flag.[4] Specifically, each scheduler thread is given an entry in the cell's local descriptor table (LDT) that points to a single `TICKS_ENABLED` word. Each thread's FS register is set to index the LDT appropriately such that any atomic access to `%fs:(0x0)` will correctly modify `TICKS_ENABLED`.

4.4. Adaptation Handling in Pulse

Tessellation's user-level runtimes must react to resource redistributions from the kernel. When the resources available to a cell vary (*i.e.,* an adaptation event occurs), the kernel copies all previously running contexts into user-space structures, sets a (per-cell) flag indicating that an adaptation event is in progress, and calls Pulse. To simplify our design, we do not support multiple outstanding adaptation events: an event must terminate before a new one starts; if a cell breaks this requirement, it triggers a termination failure. We claim this approach is sufficient for experimentation and avoids over-engineering, as adaptation events are expected to happen at a coarse time scale and to be handled quickly.

When Pulse responds to an adaptation event, it checks each previously running context to determine if it is a scheduler or application context (it is enough to check if ticks are disabled, indicating a scheduler context). If there are interrupted scheduler contexts, Pulse will activate an *auxiliary* scheduler to deal with all the outstanding scheduler contexts before any application contexts. The auxiliary scheduler ensures that the

[3] For non-preemptive cells, `TICKS_ENABLED` is always clear.

[4] We are investigating non-architecture specific methods to atomically control a thread-local variable.

	Cycles		Nanoseconds	
	Average	Std. Dev.	Average	Std. Dev.
1st Level	4279	358	1452	121
2nd Level	4711	335	1598	114
Total	8990		3050	

Table 3: Thread Switch Latency Results

application scheduler is never aware that it was interrupted, so that it never has to deal with inconsistent scheduler states.

If an adaptation event increases the number of available hardware threads (harts), the auxiliary scheduler has sufficient harts to simply switch to each outstanding scheduler context on a different hart and allow those to run in parallel until they switch to application contexts and re-enable ticks. Then, on the next timer tick, Pulse will determine that the adaptation event is still in progress and that the saved contexts are now purely application contexts. Pulse then saves the application contexts into an array, which it passes to the application scheduler via the `adapt` callback. When the `adapt` callback returns, Pulse clears the adaptation flag, marking the adaptation event as complete, and then calls the application scheduler's `tick` callback to resume normal scheduling.

If an adaptation event decreases the number of harts, there may not be enough harts to simultaneously run each outstanding scheduler context to completion. Consequently, the auxiliary scheduler runs the outstanding scheduler contexts in a globally *cooperative*, Round Robin manner; *i.e.,* a scheduler context runs until it either completes and transitions into an application context, or yields into Pulse, allowing other contexts to run. While a cooperative scheduler simplifies the design, it adds the requirement that application schedulers never block. For this reason, Pulse provides synchronization functions and the `pulse_sched_ctx_done` function, described in the example scheduler in Section 4.2.

Another consequence of using a Round Robin auxiliary scheduler is that a scheduler context could complete on *any* of the available harts, meaning that the underlying hart ID might change, possibly breaking an application scheduler that relies on hart IDs to index data structures. To prevent this problem and ensure that adaptation events are transparent to the application scheduler, the auxiliary scheduler virtualizes the hart IDs and makes sure that each user thread always gets the same unique ID when calling the function `vhart_id()`, regardless of the actual hart in use.

4.5. Scheduler Performance

The user-level implementation of scheduling allows for low latency thread switching. Table 3 shows thread-switch latency on a quad-core, 3.3GHz Intel Core i5-3550. We measured the time taken in both the kernel (1st level) and userspace (2nd level), for a single thread on a single core using the global Round Robin scheduler. The results are averaged over 10,000 interrupts. This cost is hence the minimum associated with a context switch, which will also be impacted by the number of running threads, cache effects, and other factors.

5. Resource Brokerage

Without automatic resource allocation, the best way to attain performance goals is to profile applications and statically provision resources for the worst case. In contrast, Tessellation's kernel and user-level scheduler mechanisms create a unique platform for investigating *dynamic* resource allocation policies. In Tessellation, we have created a Resource Allocation Broker (RAB) that utilizes information about the system and applications to provide resource allocations to the kernel and services (*e.g.,* Network Service and GUI Service). The Broker can use a variety of allocation strategies, and next we describe two approaches we have implemented. We start by explaining the basic framework.

5.1. Resource Allocation Broker (RAB)

The RAB runs in its own cell and communicates with applications through channels, as shown in Figure 3. When a cell is started, it registers with the RAB and provides an application-specific performance target, such as desired a frame rate. These metrics are provided in the form of a time in milliseconds. For example, an application with a desired rate of 30 frames per second would specify a performance target of 33 ms per frame. While the cell is running, the Broker receives periodic performance reports, called *heartbeats* [26], containing measured performance values from the cell (*e.g.,* the time to render a frame for a video application). These measured values correspond to the performance target provided by the cell when it registered with the Broker, so a value that is larger than the target would mean that the application is not meeting its goals. The RAB also provides an interface for cells to update their performance targets while they are running.

Ideally, performance targets would be inferred by the system or provided by a more trusted source than the applications themselves. However, we chose this design point since it was straightforward to implement and we wanted to focus our efforts on exploring resource allocation approaches that could take advantage of the additional information.

RAB also accesses information from system-wide performance counters, such as cache-miss statistics and energy measurements, which enables policies to take system-wide goals into account in their allocations. For example, the Broker could reduce resource allocations of non-critical applications if the system power is too high.

The RAB monitors cells' performance using heartbeats and compares it to target rates, adjusting resource allocations as required. Allocation decisions are communicated to the kernel (using `update_cells(...)`) and services (using channels) for enforcement. The reallocation frequency is adjustable. However, in most of our allocation approaches we try to avoid frequent reallocation, leaving the user-level scheduler to handle fine-grained resource management.

Next, we present two resource allocation approaches. Section 5.2 describes a control-theory based approach to adjust processor allocations. Section 5.3 describes an optimization-based approach for multi-dimensional resource allocation that uses application models. We present these policies because we believe they are good demonstrations of the potential of an adaptive OS for efficiently guaranteeing application performance; however, there may be many more such policies.

5.2. Performance-To-Allocation Policy

The first resource allocation technique [50] uses feedback controllers to automatically allocate the right amount of a single resource type to applications with performance goals. We refer to this technique as POTA (**P**erf**O**rmance-**T**o-**A**llocation). While different types of resources exist (compute, storage, I/O bandwidth), in many cases resources of a single type become the performance bottleneck; so, POTA estimates the amount of the bottleneck resource needed to maintain a desired performance level.

In Tessellation, we use POTA to estimate resource requirements for compute-bound streaming applications hosted in cells. The performance of such applications is expressed as a throughput measurement. We assume the throughput depends on the availability of compute resources (*i.e.,* cores and CPU bandwidth).

To estimate the amount of compute resources for an application to maintain the desired throughput level, we capture the relationship between resource availability and performance with a *linear model*. Then, we synthesize a controller able to estimate resource needs based on the model.

The linear model used is:

$$t(k+1) = a \cdot t(k) + b \cdot r(k)$$

where $t(k)$ and $t(k+1)$ are throughput values at control steps k and $(k+1)$ on a window of P seconds, and $r(k)$ is the fraction of compute resources the application can use during the time quantum $(k, k+1)$. The terms a and b characterize the application's workload and scalability, respectively.

For a given application, the model's parameters are estimated online using the application's measured throughput and a recursive least-squares algorithm. Two implicit parameters of the model are: the sampling period T and the time window of throughput measurements w. Intuitively, we observe the application every Tms and evaluate its performance over the last Tms. The actual value of T depends on the application and the required control granularity. We make no assumptions on the actual values, but require $w = T$.

Once the parameters a and b are identified, a proportional-integral (PI) controller uses the model and the application's heartbeats with throughput values to determine the necessary amount (without much excess) of resources for the application to meet its performance target. For brevity, we do not present the controller's canonical derivation, but its final time-domain expression is:

$$\hat{r}(k) = r(k-1) + \frac{1-p}{b} \cdot e(k) - a \cdot \frac{1-p}{b} \cdot e(k-1)$$

Here, $e(k) = \bar{t} - t(k)$ is the performance error at step k with respect to the desired throughput $\bar{t}$, $\hat{r}(k)$ is the resource need estimate at step k, $r(k)$ is the resource allocation at step k, and p is a controller parameter. Choosing $p \in (0,1)$ ensures that the closed loop is asymptotically stable and it converges without oscillations.

In Tessellation, the model and controller for each cell (hosting a given application) are implemented in the RAB. This way the RAB can establish the resource needs of the cells and redistribute the resources accordingly.

5.3. Multidimensional Optimization Policy

The second resource allocation technique is called PACORA [17], which stands for Performance-Aware Convex Optimization for Resource Allocation. PACORA formulates resource allocation as an optimization problem built from two types of application-specific functions: a response time function and a penalty function. PACORA uses convex optimization [18] to efficiently determine the ideal resource allocation across all active cells by minimizing the total penalty of the system.

Response time functions (RTF) represent the expected *response time* of a cell as a function of the resources allocated to it. The response time is an application-specific measure of the performance of the application, such as the time to produce a frame or the time from a service request to its response. This information is provided to PACORA through the heartbeats interface. By varying the resource allocations and collecting heartbeat information, PACORA can determine how well each cell scales with a particular resource. Using the performance history of the cell, we fit the model:

$$\tau(w,a) = \tau_0 + \sum_{i \in n, j \in n} \frac{w_{i,j}}{\sqrt{a_i * a_j}}$$

Here τ is the response time, i and j are resource types, n is the total number of resource types, a_i and a_j are the allocations of resource types i and j, and $w_{i,j}$ is the application-specific weight for the term representing resources i and j.

Penalty functions embody user-level goals of the application. Although similar to priorities, they are functions of the response time rather than simply values so can explicitly represent deadlines. Knowing the deadlines lets the system make optimizations that are difficult in today's systems, such as running just fast enough to make the deadline. Like priorities, the penalty functions are set by the system on behalf of the user.

PACORA's penalty functions π are non-decreasing piecewise-linear functions of the response time τ of the form $\pi(\tau) = \max(0, (\tau - d)s)$, where d represents the deadline of the application and s (slope) defines the rate the penalty increases as response time increases. For applications without response time constraints the deadline can be set to 0.

Given the response time and penalty functions, PACORA formulates resource allocation as an optimization problem designed to minimize the total penalty of the system. This approach is analogous to minimizing user dissatisfaction with the user experience due to missed deadlines. The optimization selects the allocations for all resources and resource types at once, thereby enabling the system to make tradeoffs between resource types. For example, the system could choose to allocate more memory bandwidth in lieu of on-chip cache, or one large core instead of several small cores.

A succinct mathematical characterization of this resource allocation scheme is the following:

$$\begin{aligned} \text{Minimize} \quad & \sum_{p \in P} \pi_p(\tau_p(a_{p,1} \ldots a_{p,n})) \\ \text{Subject to} \quad & \sum_{p \in P} a_{p,r} \leq A_r, r = 1, \ldots n \\ \text{and} \quad & a_{p,r} \geq 0 \end{aligned}$$

Here π_p is the penalty function for application p, τ_p is its RTF, $a_{p,r}$ is the allocation of resource r to application p, and A_r is the total amount of resource r available.

PACORA is convex by construction, so that optimization can be solved quickly with low overhead. We have observed in our experiments that solving this optimization in Tessellation takes 350 µs. PACORA also formulates RTF *creation* as a convex optimization problem, which takes 4 µs. Section 6.2 demonstrates how Tessellation uses PACORA to allocate resources in our video conference scenario.

6. Case-study Evaluation

Now we demonstrate how the polices described in Section 5 can be used to efficiently allocate resources for different parts of the video conference scenario described in Section 1. In Section 6.1, we use POTA to provide a consistent frame rate for encoding the user's outgoing video feed without over-provisioning resources, despite variable execution phases. Section 6.2 demonstrates using PACORA as the overall resource brokerage system, dividing resources between incoming video streams, a file indexer, and outgoing network data.

Our test platform is an Intel server containing two 2.66-GHz Xeon X5550 quad-core processors with hyper-threading enabled and a 1-Gbps Intel Pro/1000 Ethernet network adapter.

6.1. Leveraging POTA Policy for Stable Performance

We use POTA to allocate resources to a video-encoder application that produces an outgoing video stream. The performance requirement on the video encoder is to output 30 fps to allow a high-quality, glitch-free video experience. Figure 6 shows the frame rate achieved under two possible resource-allocation solutions conceived to meet that performance requirement.

We use the x264 video encoder from the PARSEC 2.1 benchmark suite with a downscaled (512x288 pixels) version of the original input. The encoder runs in a cell, and we adjust the encoding settings so that the encoder is able to meet the 30-fps requirement throughout the run with 8 hyperthreads fully dedicated to it, which represents our 100% allocation.

The most straightforward way to achieve the desired QoS is to statically reserve the 8 hyperthreads for the (non-multiplexed cell hosting) video encoder. Figure 6a shows the application's throughput measured with a 1-second moving average. While the encoder runs fast enough to meet the performance requirement, the output frame rate varies depending on the characteristics of the video input (*i.e.,* x264 goes through different execution phases [47]), leading to exceedingly fast performance for a significant portion of the execution. This situation is not ideal for two reasons: 1) the application uses more resources than needed during part of its execution, and 2) since the output data is consumed at constant rate, faster phases lead to increased buffering requirements.

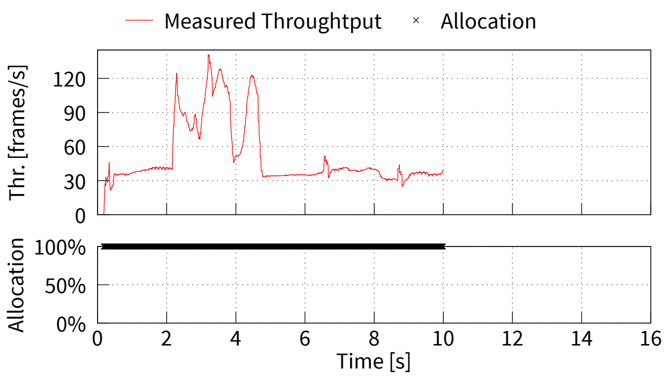
With POTA, we set the performance target to 30 fps and let the system to dynamically determine the CPU-bandwidth allocation for a time-triggered cell hosting the encoder.[5] Figure 6b shows that POTA is able to dynamically vary the amount of CPU-bandwidth granted to the video encoder and keep the performance very close to the 30-fps mark, while not over-allocating resources. In fact, POTA allocates *69.95%* of the CPU bandwidth that the static allocation uses – which leaves a resource slack that can be either employed in other useful work or idled to take advantage of power-saving mechanisms. Moreover, POTA avoids the need for a large buffer to store the excess of data produced in the faster phase.

6.2. Adapting Multiple Resources

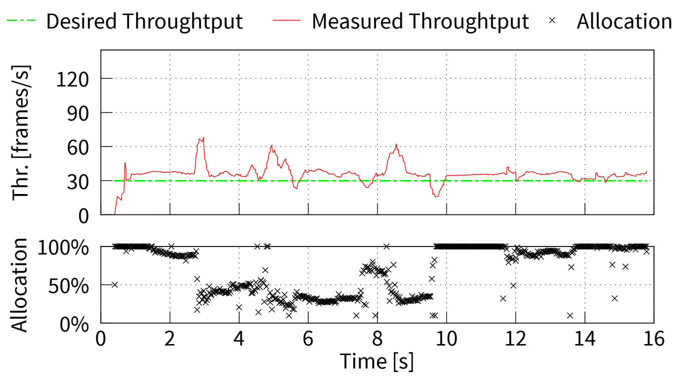
This experiment demonstrates using PACORA to allocate cores and network bandwidth to 3 applications (video player, file indexer, network hog) in our video conference scenario, and shows the complete adaptive loop in Tessellation.

Our streaming video application is a multi-threaded, network-intensive workload intended to simulate video chat applications like Skype [10] and Facetime [5]. We have 9 incoming video streams each handled by a thread in our video cell, which uses Pulse EDF scheduler (see Section 4). Each thread receives frames from the network service, decodes them using libffmpeg, and sends them to the GUI service for display. Videos are encoded offline in H.264 format using libx264, transported across the network via a TCP connection from a Linux Xeon E5-based server. We use Big Buck Bunny [1] for each video. Videos can be small or large (although only one may be large at a time) and are resized by a keyboard command. Small videos require roughly 90 kbit/s while large require 275 kbit/s of network bandwidth. Tessellation provides each video stream a separate network bandwidth allocation, and the videos share their core allocations using the EDF scheduler.

[5] Since we have only one application, the resource brokerage step is trivial: it just allocates the fraction of resources the controller requests.

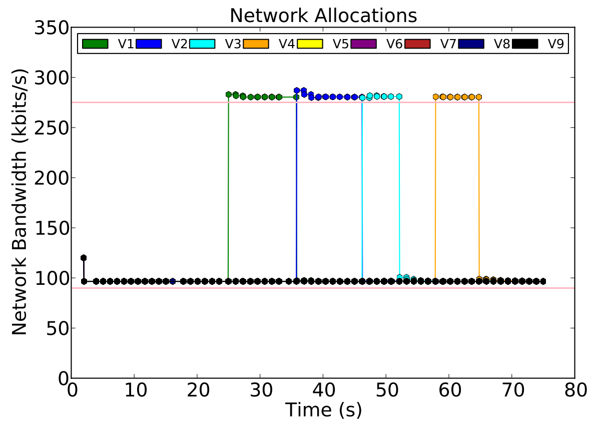


(a) Static allocation of 8 cores at 100% of the bandwidth.

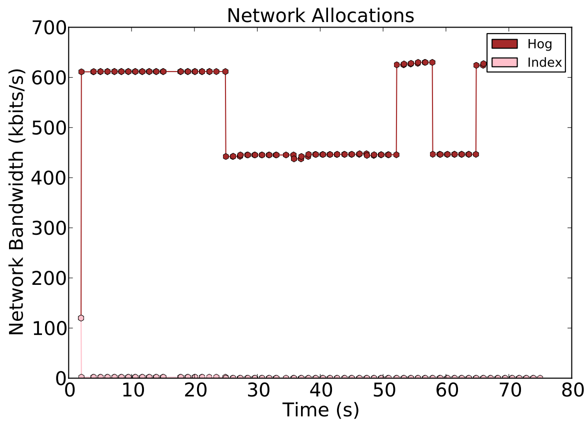


(b) Dynamic processor-bandwidth allocation with a performance goal of 30 fps on a 1-second moving average.

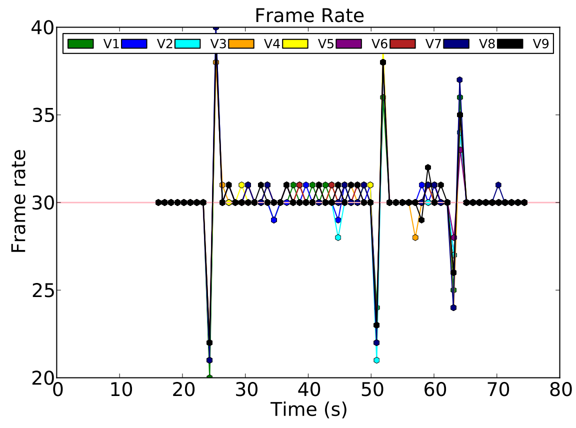
Figure 6: Two resource-allocation solutions to meet the performance requirements of a streaming application. The application is the x264 video encoder running on 8 cores. The static allocation wastes resources, while POTA does not.



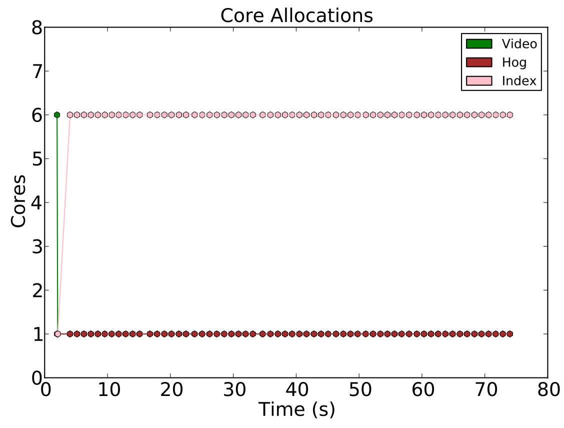
(a) Network allocations for 9 incoming videos



(b) Network allocations for file indexer and bandwidth hog



(c) Frame rate for 9 incoming videos



(d) Core allocations for incoming video, file indexer, and bandwidth hog

Figure 7: Allocations and frame rate when running 9 video-player threads, a file indexer, and a bandwidth hog together. Periodically, one of the videos becomes large causing the allocations to change. The two red lines on 7a represent the required network bandwidth for a large and small video. The red line on 7c represents the desired frame rate of 30 fps for the video-player threads.

Our file indexer application is psearchy, a pthread-based parallel application from the MOSBENCH benchmark suite [7]. It runs on top of a pthread-compatible runtime system implemented in Pulse. The TCP bandwidth hog is a simple, single-threaded application that transmits data at the fastest possible rate, similar to Dropbox [4] or an FTP server.

Figure 7 shows the allocations of the applications and video frame rates as Tessellation runs and adapts to the video resizes. The first adaptation event occurs at $t = 2$s, when PACORA changes the allocations from their initial settings to application-specific allocations. All network allocations were initially set to 120 kbits/s, and as shown in Figures 7a and 7b, PACORA changes all of the video threads to 96 kbits/s, just above the required network bandwidth for small videos. PACORA removes all bandwidth from the file indexer since it does not use network bandwidth and gives the remaining bandwidth in the system to the network hog.[6] We also see in Figure 7d that Tessellation removes cores from the video cell and gives them to the file indexer, utilizing the mechanisms described in Sections 3 and 4 to do so.

Additional resizing events occur at 25, 35, 52, 58 and 65 seconds, when videos 1, 2, 3, and 4 change size. As shown in Figures 7a and 7b, PACORA reduces the network hog's allocation in order to give sufficient bandwidth to the large video. However, when all the videos are small the bandwidth is returned to the network hog. Figure 7d shows that larger videos do not need enough additional processing power to require an increase in cores, so the core allocations do not change after the initial adaptation. Figure 7c shows that the videos do not drop below the required frame rate except when resizing.[7]

All of the runtime functions used in this experiment were built automatically from measured values using RAB's heartbeat interface. These results demonstrate that using Tessellation's kernel, Pulse, and the RAB, we are able to implement a resource-allocation approach that can efficiently assign resources to applications and adapt the resource allocations as the system changes state. As a result using an ARCC style system, we do not need to sacrifice utilization in order to provide performance guarantees.

7. Related Work

A number of research efforts have focused on the problem of adaptive resource allocation to meet QoS objectives in multi-application scenarios. Redline [52] is an adaptive resource manager that attempts to maximize application responsiveness and system utilization. Like Tessellation, Redline supports heterogeneous workloads, but Redline does not provide the same guarantees as Tessellation, since QoS-bound tasks may be demoted to best-effort if the system becomes overloaded.

[6]We have artificially limited the available cores to 8 and the available network bandwidth to 1500 kbits/s to make the resources more constrained.

[7]Glitches while resizing are an artifact of the application implementation.

Tessellation avoids this by using admission control and gang-scheduled time-multiplexing classes with guaranteed timing.

AutoControl [39] is a feedback-based resource-allocation system for shared virtualized infrastructure in data centers. It operates at the hypervisor level to allocate CPU and disk I/O bandwidth to virtual machines (VMs) in order to mitigate bottlenecks. While Tessellation's cells resemble some aspects of VMs, they are intended to provide better performance isolation and more precise control and are lighter-weight containers.

SEEC [26] is a self-aware programming model designed to facilitate the development of adaptive computing systems on multicore platforms. As with SEEC, Tessellation aims at providing a general and extensible framework for self-adapting computing. However, SEEC explores a user-level extension to commodity OSs. Tessellation, on the other hand, is built with the belief that an OS designed for resource isolation and adaptation can yield finer control. AcOS [15] is an autonomic resource management layer to extend commodity OSs. AcOS considers application performance goals and temperature thresholds to implement a dynamic performance and thermal management (DPTM) control loop to cap temperature without impairing QoS. The scope of AcOS is closer to Tessellation than SEEC is; the main difference is that AcOS leverages commodity OSs to support autonomic resource management, while Tessellation builds this support from the ground up. Moreover, neither AcOS nor SEEC in practice consider OS services as part of the resource allocation problem.

METE [48] 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 requires hardware partitioning mechanisms to provide QoS (and is evaluated only in a simulation environment), while Tessellation builds support for resource partitioning into the OS and works on real hardware.

Tessellation has similarities to several recent manycore OSs. The use of message-passing communication via user-level channels is similar to Barrelfish [16]. However, Barrelfish is a multikernel OS that assumes no hardware assistance for cache coherence, and does not focus on adaptive resource allocation. The way Tessellation constructs user-level services is similar to fos [51]. Services in Tessellation are QoS-aware and cells are partitioned based on applications rather than physical cores. Tessellation is similar to Corey [19] in that we also try to restrict sharing of kernel structures.

Tessellation adopts a microkernel philosophy [33], 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 [23]. 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 [49]. Both implement some OS services in user-space, but K42 uses protected procedure calls to access them, where Tessellation uses user-level channels.

Tessellation and Nemesis OS [25] both emphasize on ensuring QoS for multimedia applications. Nemesis also relies on OS services and message passing, but on uniprocessors.

Through mechanisms such as control groups [3], the `isolcpus` boot option, thread affinity, IRQ affinity masking, and libnuma [8], Linux supports thread performance isolation. However, achieving isolation from per-CPU kernel threads is not possible without modifying the Linux kernel [13]. Tessellation's tickless environment, however, provides guaranteed isolation between threads.

Resource partitioning has also been presented in McRT [45] and Virtual Private Machines (VPM) [36]. The concepts of VPM and cells are similar, but VPM lacks inter-cell communication and has not been implemented yet.

Gang-scheduling [38] is a classic concept and has also been applied to other OSs – most similarly in Akaros [43]. However, unlike other systems, Tessellation supports cells with different timing behaviors. Our gang-scheduling implementation and that of Fong et al. [24] both rely on high-precision synchronized clocks. But ours focuses on a single node while theirs on a cluster computing environment.

Kato and Ishikawa's work on Gang EDF [28] differs from ours in several ways. Their algorithm is centralized and focuses on a single resource type (parallel tasks on processors), while ours is decentralized, incorporates scheduling disciplines besides EDF, and considers multiple resources (*e.g.,* cache ways and memory pages). Also, they couple resource placement with scheduling, such that each scheduling decision involves a bin-packing operation. While their approach may result in better CPU utilization, ours does not suffer the performance overhead of bin-packing.

8. Discussion and Conclusion

In this paper, we describe our experiences designing and constructing the kernel- and user-level mechanisms to *efficiently* distribute guaranteed resources for predictable application performance. These mechanisms implemented in Tessellation include precise gang-scheduled privileged resources, adaptive user-level schedulers, and the framework and policies to control the resource allocation. We demonstrate this system in action running on current x86 hardware with real applications using two cutting-edge resource allocation policies, POTA and PACORA, and show that it could potentially be used to guarantee performance without sacrificing utilization. Many of the mechanisms we explore in our new OS could be retrofitted to an existing OS, and we believe the demonstrations of minimizing missed deadlines and maximizing system utilization in this paper provide a good proof-of-concept of the potential of these mechanisms in current and future operating systems.

We are now working to evolve the ideas of ARCC and Tessellation to provide efficient resource management and guarantees in distributed environments. In these scenarios, the resources are no longer entirely on-chip and could include new partitioning hardware, such as Audio Video Bridging-enabled devices, novel sensors and screens, and compute clusters.

Acknowledgments

This research is supported by Microsoft (Award #024263), Intel (Award #024894), matching UC 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 would like to thank other Par Lab members for their collaboration and feedback. Juan A. Colmenares participated in this work first as a post-doctoral scholar and then as a visiting researcher for Samsung at UC Berkeley.

References

[1] Big Buck Bunny. http://www.bigbuckbunny.org/.
[2] CFS vs. BFS Linux kernel mailing list discussion. http://lwn.net/Articles/351058/.
[3] Cgroups. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.
[4] Dropbox. https://www.dropbox.com.
[5] Facetime. http://www.apple.com/ios/facetime/.
[6] FFmpeg. http://www.ffmpeg.org/.
[7] MOSBENCH. http://pdos.csail.mit.edu/mosbench/.
[8] NUMA Policy Library. http://linux.die.net/man/3/numa.
[9] SCHED_DEADLINE. https://github.com/jlelli/sched-deadline.
[10] Skype. http://www.skype.com/.
[11] L. Abeni and G. Buttazzo. Resource reservations in dynamic real-time systems. *Real-Time Systems*, 27(2):123–165, 2004.
[12] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *Proc. of CODES+ISSS*, pages 251–256, 2007.
[13] H. Akkan, M. Lang, and L. M. Liebrock. Stepping towards noiseless Linux environment. In *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS'12, pages 7:1–7:7, June 2012.
[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 164–177, October 2003.
[15] D. B. Bartolini, R. Cattaneo, G. C. Durelli, M. Maggio, M. D. Santambrogio, and F. Sironi. The autonomic operating system research project: Achievements and future directions. In *Proceedings of the 50th Annual Design Automation Conference*, DAC'13, pages 77:1–77:10, June 2013.
[16] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 29–44, June 2009.
[17] S. Bird and B. Smith. PACORA: Performance-aware convex optimization for research allocation. In *Proc. of HotPar*, 2011.
[18] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, England, 2004.
[19] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proc. of the ACM Symp. on Operating Systems Design and Implementation (OSDI'08)*, pages 43–57, December 2008.
[20] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moretó, D. Chou, B. Gluzman, E. Roman, D. B. Bartolini, N. Mor, K. Asanović, and J. D. Kubiatowicz. Tessellation: refactoring the OS around explicit resource containers with continuous adaptation. In *Proceedings of the 50th Annual Design Automation Conference*, DAC'13, pages 76:1–76:10, June 2013.

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

[22] J. A. Colmenares, N. Peters, G. Eads, I. Saxton, I. Jacquez, J. D. Kubiatowicz, and D. Wessel. A multicore operating system with QoS guarantees for network audio applications. *Journal of the Audio Engineering Society*, 61(4):174–184, April 2013.

[23] 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.

[24] L. L. Fong, A. S. Gopal, N. Islam, A. L. Prodromidis, and M. S. Squillante. Gang scheduling for resource allocation in a cluster computing environment. Patent US 6345287, 1997.

[25] S. M. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 73–86, February 1999.

[26] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal. SEEC: a general and extensible framework for self-aware computing. Technical Report MIT-CSAIL-TR-2011-046, Massachusetts Institute of Technology, 2011.

[27] S. Hofmeyr, J. A. Colmenares, C. Iancu, and J. Kubiatowicz. Juggle: addressing extrinsic load imbalances in SPMD applications on multicore computers. *Cluster Computing*, 16(2):299–319, June 2013.

[28] S. Kato and Y. Ishikawa. Gang EDF scheduling of parallel task systems. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, RTSS'09, pages 459–468, December 2009.

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

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

[31] J. W. Lee, M. C. Ng, and K. Asanovic. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. *SIGARCH Comput. Archit. News*, 36(3):89–100, June 2008.

[32] B. Leiner, M. Schlager, R. Obermaisser, and B. Huber. A comparison of partitioning operating systems for integrated systems. In *Proc. of SAFECOMP*, pages 342–355, September 2007.

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

[34] 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.

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

[36] 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.

[37] 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.

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

[39] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proc. of EuroSys*, pages 13–26, 2009.

[40] H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with Lithe. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'10, pages 376–387, June 2010.

[41] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. *SIGARCH Comput. Archit. News*, 37(3):57–68, June 2009.

[42] J. Reiders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O'Reilly, 2007.

[43] B. Rhoden, K. Klues, D. Zhu, and E. Brewer. Improving per-node efficiency in the datacenter with new OS abstractions. In *Proc. of SOCC*, pages 25:1–25:8. ACM, 2011.

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

[45] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang. Enabling scalability and performance in a large scale CMP environment. In *Proc. of EuroSys*, pages 73–86, March 2007.

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

[47] H. Sasaki, T. Tanimoto, K. Inoue, and H. Nakamura. Scalability-based manycore partitioning. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT'12, pages 107–116, New York, NY, USA, 2012. ACM.

[48] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das. METE: meeting end-to-end qos in multicores through system-wide resource management. *SIGMETRICS Perform. Eval. Rev.*, 39(1):13–24, June 2011.

[49] D. D. Silva, O. Krieger, R. W. Wisniewski, A. Waterland, D. Tam, and A. Baumann. K42: an infrastructure for operating system research. *ACM SIGOPS Oper. Syst. Rev.*, 40(2):34–42, 2006.

[50] F. Sironi, D. Sciuto, and M. D. Santambrogio. A performance-aware quality of service-driven scheduler for multicore processors. *To appear in ACM SIGBED Review*, January 2014.

[51] D. Wentzlaff, C. Gruenwald, III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An operating system for multicore and clouds: mechanisms and implementation. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 3–14, New York, NY, USA, 2010. ACM.

[52] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Redline: First class support for interactivity in commodity operating systems. In *Proc. of 8th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'08, 2008.