Modern devices must juggle the competing requirements of parallel high-throughput, interactive, and multimedia applications while conserving energy where possible. Each requires some combination of CPU, cache, memory, and network bandwidth, and allocating resources among applications is a complex optimization problem. Application requirements are often conflicting, and users want predictable responsiveness, and long battery life, despite dynamically changing resources. Modern OSes do not adequately tackle this problem: system-wide power goals do not consider the priorities of various applications, and can hurt the user experience. We present a resource adaptation system, developed in the Tessellation OS, that models and schedules resources for energy consumption, in order to automatically find low-energy and high-performance resource configurations for realistic workloads. Our results show that in a two app scenario with differing priorities, our system automatically finds a resource allocation that meets both performance requirements or (when power saving is more important) reduces system power by 7 W and continues to meet the high priority application’s performance target.

1 INTRODUCTION

Users have an insatiable appetite for responsive user interfaces and high-quality multimedia with stringent real-time guarantees on platforms with limited battery life or power budget. Modern applications consist of multiple interacting components, each with differing resource needs and QoS requirements. Such applications frequently extend into the cloud, requiring responsiveness and performance predictability on a global scale. Further, with the prevalence of multicore devices, users expect better performance or responsiveness as the core count increases. Meeting these expectations requires not only parallelizing client applications but striking the right balance of resource allocation among competing software components.

Mobile devices such as the iPad 3 and Kindle Fire (whose batteries are 42.5 Watt-hours and 16 Watt-hours, respectively) have potentially short single charge lifetimes if energy is not intelligently managed. Likewise in computing datacenters, the authors of the Exascale Computing Study consider energy and power management the "most pervasive" of all challenges to reaching exascale computing. The study’s authors predict a maximum system power of 154.8 MW for a 600 rack datacenter in 2020!

Executing multiple parallel, real-time applications while satisfying QoS requirements and minimizing energy consumption is a complex optimization problem. Ideally, applications with strict performance requirements should be given just enough resources to meet these requirements consistently, without unnecessarily siphoning resources from other applications. For just two applications with separate performance requirements, considering resources of CPU frequency, time multiplexing, and number of cores, the number of possible resource configurations is:

\[ O\left(\text{Number of frequency states} \times \left(\frac{100}{\text{CPU utilization granularity}} \times \text{Number of cores}\right)^2\right) \]

Within that resource state space, it is likely that if there exists one point that satisfies every application’s performance requirements, there are many points that do. Thus a power management system should find a system that both meets performance requirements and minimizes power consumption.

Modern OSes provide CPU frequency, idle state, in some cases power capping control either directly (e.g. Linux’s `CPUFreq` daemon) or indirectly (e.g. Windows’ power plans). These mechanisms lack fine-grained control: they offer system-wide power control, not per-app. A typical use-case scenario is some interactive or multimedia foreground application with a performance target (e.g. 30 FPS for a video application) running concurrently or in
parallel with background throughput applications. In such a case, system-wide power reductions will likely cause the foreground app to miss its performance targets, resulting in an unacceptable user experience.

Our power management system offers per-application power control via a simple interface, such that the OS can reduce power consumed by low-priority applications and continue to meet high-priority applications' performance (typically latency) targets (if feasible). For each application, the user defines a penalty function, which is a unitless measure of the system penalty associated with a certain performance. Users can define performance targets by setting zero penalty if the performance meets or beats the target, and can differentiate apps by varying the penalty function slope if the target is not met. The performance is measured offline and is a function of allocated resources. Crucially, this allows the resources we control are socket frequency, cell time multiplexing, and number of cores per cell (though others, such as network bandwidth, are possible with our framework).

The rest of the paper is organized as follows. Sections 2 through 4 introduce the three main components of our adaptive system: the Tessellation OS, the resource partitioning algorithm, and the user-level scheduling, respectively. Section 5 explores our experimental evaluation and data. Section 6 discusses planned future work, and the paper concludes in section 8.

2 SYSTEM ARCHITECTURE

We implemented our adaptive resource system on top of Tessellation OS, as it provides two key guarantees:

- Apps have guaranteed access to resources, which are self-managed
- Resource partitions are dynamic and periodically re-generated by the Resource Allocation Policy (RAP) Service

Tessellation [3, 7] embodies an Adaptive Resource Centric Computing (ARCC) approach, illustrated in Figure 1. This approach enables a simultaneous mix of interactive, real-time, and high-throughput parallel applications by automatically discovering the mix of resource assignments that maximizes the net system utility for the end user—an indicator of how well the applications are meeting their requirements and the whole system is satisfying the user's needs. Tessellation treats resources as "first-class citizens" that can be assigned to application components in order to provide predictable performance, which is essential for live-performance musical and other time-sensitive applications. Resources are distributed to QoS domains called cells, which are light-weight containers with guaranteed, user-level access to resources.

The Tessellation kernel is a thin software layer that provides support for ARCC by implementing cells and providing interfaces for resource adaptation and cell composition. Cells are implemented through space-time partitioning [8, 7]. The stable performance of cells permits two-level scheduling [9, 3], which separates resource allocation from usage. In addition, the performance-isolated environment of a cell makes it possible to experimentally observe application performance metrics (e.g., completion time and throughput) and predict how these metrics vary with resources—thus enabling accurate resource optimization.

2.1 The Cell Model

Cells provide the basic unit of computation and protection in Tessellation. They are performance-isolated resource containers that export their resources to user level. Once resources (e.g., CPU cores and memory pages) have been assigned to cells, the Tessellation kernel gives cells full control over the usage of the resources allocated to them. Cells control their resource usage at user-level (i.e., outside the kernel) and the kernel is just minimally involved.

Applications in Tessellation are created by composing cells that communicate via efficient and secure channels, which enable fast asynchronous message-passing communication at user level (see Figure 2). Channels allow an application component in a cell to access OS services (e.g., network and file services) and to interact with other application components residing in other cells.

Space-Time Partitioning: Tessellation divides the hardware into a set of simultaneously-resident (spatial) partitions (see Figure 3). 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 guaranteed cache-memory units, portions of memory bandwidth, and fractions of the energy budget, when the required hardware mechanisms are available (e.g., [2, 6, 11, 12]).

Tessellation OS virtualizes partitions and exports them to applications and OS services through the cell abstraction. Partitions are virtualized via strictly controlled time-multiplexing. The kernel implements a scheduling policy...
that coordinates cell switching and ensures that each cell has simultaneous access to the entire “gang” of resources assigned to it. In other words, CPU cores and other resources are gang-scheduled [10, 4] such that cells are unaware of this multiplexing; i.e., unexpected virtualization of physical resources does not occur.

Tessellation provides several time-multiplexing policies for cells, some of them offering high degrees of time predictability; they are: 1) no multiplexing (cell given dedicated access to its assigned resources), 2) time triggering (cell active during predetermined and periodic time windows), 3) event triggering (cell activated upon event arrivals, but its contribution to the total utilization never exceeds its assigned fraction of processor time), and 4) best effort (cell with no time guarantees).

**Two-Level Scheduling:** Two-level scheduling 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 to 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 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 CPU scheduling algorithm.

### 2.2 Service-Oriented Architecture

Cells provide a convenient abstraction for building OS services (such as network interfaces, file systems, and windowing systems) with QoS guarantees. Such services can reside in dedicated cells, have exclusive control over devices, and encapsulate user-level device drivers (see Figure 1). Each service can thus arbitrate access to its enclosed devices and leverage the cell’s performance isolation and customizable QoS-aware schedulers to offer service-time guarantees to applications and services residing in other cells.\(^1\) Services may exploit parallelism to reduce service times or increase service throughput. Further, services can shape data and event flows coming from external sources with unpredictable behavior and prevent other cells from being affected.

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).

Two examples of services in Tessellation that offer QoS guarantees to client cells are the network service and the GUI service. The former in particular is key to low-latency networked music applications.

**Network Service:** This service provides access to network adapters through an API similar to the socket API [13] found in Linux and other Unix-like OSs. The network service is implemented using the lightweight TCP/IP protocol stack lwIP [1]. This service allows the specification of **minimum throughput reservations** for data flows between network adapters and client cells. It 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 total system throughput). Moreover, the network service distributes any excess throughput proportionally among the client cells via an adaptation of the mClock algorithm [5].

### 2.3 Adaptive Resource Allocation

Tessellation can use adaptive resource allocation to provide QoS guarantees to applications while maximizing efficiency in the system. The **Resource Allocation Policy (RAP) Service** is an implementation of the control plane of Figure 1; it encapsulates the decision-making logic to distribute resources to cells. The RAP Service runs in its own cell and communicates with applications and other services through channels. It decides how resources should be divided among cells in the system by monitoring other cells, adapting their resources in response to changing conditions, and controlling the admission of new cells into the

---

\(^1\)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.
system.\(^2\) The RAP Service communicates the allocation decisions to the kernel via a system call and to services through their QoS Specification interfaces over channels.

Note that the user can specify the set of resources to be given to a cell and indicate to the RAP Service that such resource allocation cannot change. This feature is very valuable to musical performers. It gives them complete control over the resources allocated to their musical applications. Further, it gives performers high confidence that their applications will behave as expected because they can recreate the execution conditions in which the applications were tested, profiled, and optimized.

3 POLICY SERVICE

The Resource Allocation Policy (RAP) Service is the “brains” of the adaptive resource-centric computing loop. It is responsible for data collection, performance modeling, and resource partitioning. The RAP Service communicates with each connected application’s second-level runtime system to receive performance reports, models, and targets. While the RAP Service could be implemented in a number of ways, we currently use the convex-optimization based PACORA algorithm.

Currently the RAP Service controls three resources that give it direct control over the performance and power usage of our applications. They are number of cores, core frequency, and time multiplexing of the application running on a core.

3.1 Resources

The resource allocation service controls the time multiplexing parameters of all cells. For a gang-scheduled system such as Tessellation, this requires choosing three parameters: number of cores, cell active time, and cell period. For simplicity, we set the maximum period to one second and the active time granularity to one millisecond, which is the minimum currently supported by Tessellation. We utilize the Enhanced Intel SpeedStep (EIST) technology to control the clock speed of processing elements. By writing a P-state value to a machine specific register, the socket can dynamically change its clock speed. From an initial experiment where we stepped through each available P-state, we saw frequencies ranging from 48 percent to 100 percent on our evaluation machine, which gave a total system power reading of 150W to 250W, respectively, according to our wall power meter.

3.2 PACORA algorithm

PACORA is a resource allocation system that constructs the resource allocation problem as a convex optimization problem. The PACORA algorithm minimizes the sum of each application’s penalty functions, which is an indicator of the app’s importance. The argument to the penalty function is the performance model - a measure of the application’s performance. The performance model is a function of the application’s resources. By providing isolated, guaranteed resources, we can ensure that pre-generated performance models match their performance in the presence of other applications. Our power-management algorithm is an extension of PACORA, in which system power is modeled as a competing application.

Performance Models: The RAP Service generates application performance models offline by running them in isolation with a set of operating points chosen from the large space by a genetic algorithm. The RAS collects heartbeats, which are 64-bit unsigned integer performance measures in a user-defined metric. The metric may vary: what is crucial to the algorithm is that the resulting performance model is convex.

Second-Level Runtime Interface: The RAP Service communicates with applications through their second-level runtime via a pre-defined interface. Performance models are communicated via the perf\_func\_t struct, which has members runtime\_target, penalty\_slope, and model\_constants. The application can send penalty, deadline, or model updates at any time. Performance reports, or “heartbeats”, contain an array of uint64\_t data points and a num\_values member. Every message is prepended with a char message\_type, which distinguishes the various message types.

Power Management in PACORA: Within the PACORA framework, we model system power as a competing application, denoted the System Power App. Rather than having a performance function that is a function of resources, the System Power App’s performance function is the system power if the resources it receives are idled/unused. For instance, if PACORA allocate four of a machine’s cores to the System Power App, those cores are left unused. (Note: while the same example applies for a cpu timeslice, it does not apply to CPU frequency.) Further, one can control the relative importance of power saving via the System Power App’s penalty function - in particular, a high sloped penalty function will force PACORA to idle more resources, thus reducing overall power. We control which application is a candidate to lose resources via their own penalty functions: the app which will incur the least penalty from losing a core, for instance, will be the victim if one is to be removed from the system.

4 USER-LEVEL SCHEDULING

A major benefit of Two-Level Scheduling is its ability to support different resource-management policies simultaneously. Tessellation OS allows cells to provide their own, possibly highly-customized user-level runtime system for processor scheduling and memory management. Further, each cells runtime can control over the delivery of events (e.g., inter-cell messages), timer interrupts, exceptions, and faults. In this section, we describe the support TL provides

---

\(^2\)The RAP Service refuses to admit new cells whose resource requirements are incompatible with existing obligations to other cells.
for the easy implementation of new, efficient, cell user-level runtimes.

4.1 PULSE

We use PULSE (Preemptive User-Level SchEduling), a framework for creating preemptive user-level schedulers. PULSE’s simplicity makes it easy to customize existing schedulers and to experiment with novel schedulers – without having to patch the OS as would be the case in other systems, such as Linux. PULSE simplifies the creation of user-level schedulers by only requiring the user to define four callback functions (enter(), tick(), yield(), done()). Thread- and preemption-management utility functions, such as pulse_thread_alloc() or pulse_ctx_restore(), are defined in the PULSE API so that scheduler writer not be burdened by machine-level implementation issues. In order to be compatible with the adaptive resource loop, we extended PULSE to manage hardware thread adaptation (exposed to the user via a fifth callback function, adapt()).

An adaptive second-level runtime must support the addition and revocation of resources, and in particular hardware threads. While it is straightforward to migrate software thread contexts onto additional hardware threads, recovering thread contexts from revoked hardware threads is not. In the revocation case, simply placing the thread contexts from lost cores into the runnable queue will not work. The contexts may have been running in the scheduler, and critically - may have held a lock, in which deadlock could occur. Consider the case of adapting from three to one hardware thread: if the software context from the second core was running held the lock to a global round robin queue, the scheduler on the remaining core could not access the runnable application contexts until that lock is released. The solution is to run scheduler contexts to a stable point - i.e. one in which the context is not locally managing an application context, holds a lock, or is executing a critical section - and discard them. At the end of this process, all application contexts should be safely returned to an execution queue. We denote the management of scheduler contexts after an adaptation event the auxiliary scheduler.

The auxiliary scheduler is initiated on all remaining cores when the kernel sets the cell’s ADAPTATION_EVENT flag. First, the zeroth core - which Tessellation guarantees to exist - places interrupted application and scheduler contexts into separate queues. If no scheduler contexts were interrupted, adaptation handling is complete. Otherwise, each remaining core attempts to grab a scheduler context to run. If it is unsuccessful, it waits at a barrier for the other cores. Once the scheduler contexts run, there are three ways they can return to the auxiliary scheduler:

- reaching a stable point, where stable is defined as being outside any critical sections and not managing any application contexts.
- yielding, in case it fails to grab a lock.
- receiving a timer interrupt, which is typical when the scheduler context was about to restore an application context.

In order to avoid deadlock in this process, it is necessary to use locks which yield to the auxiliary scheduler if acquiring the lock is unsuccessful. Once control has returned to the auxiliary scheduler, the process repeats: the context is determined to be application or scheduler, and if scheduler then it is run to completion. So long as the scheduler is guaranteed to reach a stable point, this process will complete. Finally, once the auxiliary scheduler completes, the accumulated application contexts are returned to the scheduler via the adapt() callback function.

In the worst case, the auxiliary scheduler executes the scheduler contexts of N lost cores until interrupt. Thus, the worst case adaptation recovery latency is $(\# \text{ of lost cores}) \times (\text{timer interrupt period})$.

5 EXPERIMENTAL EVALUATION

In order to evaluate the usefulness of our system, we recreate a somewhat realistic usage scenario: a foreground multimedia application with strict latency guarantees, coupled with a low-priority background processing application.

We ported four different applications to Tessellation OS for our experiment. One is a parallel browser renderer animation application, that exploits data and pipeline parallelism. We use this application as the foreground app with a target performance of 50,000 shapes drawn per second. The other three, swaptions, fluidanimate, and streamcluster, are all PARSEC benchmarks. Swaptions uses Monte Carlo simulations to compute the prices of a portfolio of swaptions. Fluidanimate simulates an incompressible fluid for interactive animation purposes with the Smoothed Particle Hydrodynamics method. And finally, streamcluster solves online clustering problems. Unfortunately, due to unforeseen last minute code problems, we were unable to use fluidanimate or streamcluster, so we restrict the background processing application to swaptions only. Swaptions’ performance target is 64 swaptions processed every 7 seconds. The performance targets were chosen to be sufficiently difficult to meet on our evaluation machine such that PACORA must use a majority of resources.

We performed all of our experiments on a manycore Sandy Bridge server with 32 hyperthreads, two sockets with 8 physical cores per socket. While implementing the P-state control, we discovered that P-states can only be modified per package. This means that we can control the P-state for each socket but every core on one socket will be forced to have the same P-state.

To prioritize the two apps, we gave the (foreground) browser application a penalty function slope of 7 and the (background) swaptions application a penalty function slope of 1. Each application’s performance model was built with ten data points. We run these two applications with a System Power App with three different penalties: 1, 10, and 100. Because the mix of apps and resources are static,
PACORA converges to a final resource allocation within its first three executions of the convex optimization algorithm. We set the performance targets to their latency inverses: i.e., 3 seconds per 150,000 shapes drawn for the browser and 7 seconds per 64 swaptions processed for swaptions. The results of the experiments are shown in Figure 4.

As shown, PACORA allocates enough resources to meet the performance requirements for two of the three experiments. For all three experiments, PACORA allocates 7 cores with 100% utilization and 100% CPU frequency to the animation app. Through the three experiments, swaptions’ resources are: 11 cores at 59% utilization, 8 cores at 82% utilization, and 4 cores at 53% utilization. In all cases, CPU frequency was kept at 100% on the sockets in use - however, in the experiment with System Power penalty slope of 100, the applications are consolidated onto one core and the unutilized socket frequency is set to the lowest possible value (48% of maximum). As expected, the lower penalty app swaptions has resources taken away as the System App penalty increases. Interestingly, PACORA expected runtime for swaptions was 6979 ms, 7384 ms, and 13592 ms respectively, while the actual runtimes were 3729.6 ms, 3921.67 ms, and 10114.7 ms. While PACORA successfully lowered swaptions’ performance in an effort to save system power, the second experiment shows that power actually increased slightly. However, power drops by about 6 W for the third experiment. The difference between PACORA’s expected runtime and actual for the animation have about the same difference as those of swaptions. This is due to imperfect models. In future work, we intend to explore the model fidelity versus num of data points used to construct it. Despite the imperfect models, we have demonstrated that PACORA can save power by revoking resources from less important applications, while continuing to meet performance requirements of high priority applications. Due to time constraints, we were unable to run a planned simulated annealing algorithm across the resource space to determine the optimal (in terms of power) resource configuration.

6 FUTURE WORK

Due to time constraints, we were not able to complete a few tasks. Of these tasks, we found the following the most important.

Comparison System: We planned to run experiments on some realtime operating systems to compare the effectiveness of Pacora in allocating resources to meet deadlines and to show we can meet just as many deadlines while saving power. We originally had two systems in mind: Wind River Linux and RTLinux. When we contacted Wind River about publishing results comparing Tessellation with their version of linux, they responded that it would be against the End User License Agreement. For RTLinux, we simply did not have enough time.

Realistic Applications: One of the main goals of our experiments was to simulate a realistic workload for a system. We had planned to run some network applications such as an FTP server and a multithreaded network video conferencing application because they are much more representative of applications a user would be interested in running. We were not able to achieve these goals because we did not have the proper driver ported to Tessellation for interacting with the NIC.

More "Knobs" to Turn: We think we would have finer control over the system’s ability to meet deadlines and the amount of power used by the system if we had more "knobs" to control, as in there are more resources that our policy service can adjust. Two resources we had in mind are cache partitioning and, if the network driver is implemented, network bandwidth, which would be especially useful with the FTP server and video conferencing application. We also would like to investigate the use of ACPI C-states, also known as CPU idling, to save further power when cores are idled.

Alternative Power Saving Model: In our current implementation of incorporating energy into the policy service, we use a fake cell called the System Power App. and we allocate resources to it like any other cell. The difference is that this cell will idle the resources it receives, thus reducing the power consumption of the system.

Another approach we designed to make the policy service energy aware is to generate runtime models and power models for the applications. A runtime model is the performance as a function of the resource configuration. Similarly, the power model is the power consumption as a function of the resource configuration. We can take the point-wise product of the two models to create a power-delay product.

7 SUMMARY AND CONCLUDING REMARKS

In this paper, we explored the use of an adaptive resource allocation system that used performance modeling and convex optimization to determine the correct partition of resources. We showed that in a two-app system, our
system can - depending on the relative priority of power saving - allocate resources to meet performance requirements of both a high- and low-priority app, or divert resources from the low-priority app to reduce power by 7 W while still meeting the high-priority application’s stringent deadline. This system is implemented on Tessellation, because it provides the primitives - particularly the Cell’s resource guarantees and the dynamic adaptation of those resources - make itself amenable to such an adaptation loop. We extended Tessellation to support a new resource - CPU frequency via Intel’s EIST hardware. Also, in order to support applications on such a system, we extended the PULSE user-level scheduling framework to support addition and revocation of cores. We also added power management, via the System Power App, to PACORA such that it can make power/performance tradeoffs according to user-defined penalty functions.

8 REFERENCES


