PACORA: Optimizing Resource Allocations for Dynamic Interactive Workloads

Authors Redacted

Abstract

rewrite duplicate sentences

Modern computing systems are under intense pressure to provide guaranteed responsiveness to their workloads. Ideally, applications with strict performance requirements should be given just enough resources to meet these requirements consistently, without unnecessarily siphoning resources from other applications. However, executing multiple parallel, real-time applications while satisfying QoS requirements is a complex optimization problem and traditionally operating systems have provided little support to provide QoS to applications. As a result, client, cloud and embedded systems have all resorted to over-provisioning and isolating applications to guarantee responsiveness. We present PACORA, a resource allocation framework, which is designed to provide responsiveness guarantees to a simultaneous mix of high-throughput parallel, interactive, and real-time applications in an efficient, scalable manner. By measuring application behavior directly and using convex optimization techniques, PACORA is able to understand the resource requirements of applications and perform near optimal resource allocation with low overhead.

1. Introduction

The demands on modern systems have changed. Users demand responsive applications often containing high-quality multimedia that requires real-time guarantees. Meeting this responsiveness goal is a challenge for all types of systems including cloud systems, databases, webservers, client operating systems, and emerging distributed embedded systems. Additionally battery life and system power are extremely important, thereby forcing systems to try to find more efficient ways to meet the quality-of-service demands of their workloads.

Ideally, applications, jobs, or queries with strict performance requirements should be given just enough system resources (*e.g.,* nodes, processor cores, cache slices, memory pages, various kinds of bandwidth) to meet these requirements consistently, without unnecessarily siphoning resources from other applications. However, executing multiple parallel, real-time applications while satisfying *Quality-of-Service* (QoS) requirements is a complex optimization problem, particularly as modern hardware diversifies to include a variety of parallel architectures (*e.g.,* multicore, gpus). Historically operating systems have not provided useful mechanisms that implement stronger performance guarantees and resource allocation has been rather unsystematic, making it difficult to reason about the expected response time of an application.

Consequently, predictability has traditionally been obtained at a significant expense by designing for the worst case and over-provisioning. Evidence of this behavior can be found in current systems of all sizes. OSs describe responsiveness with a single value (usually called a *priority*) associated with a thread of computation and adjusted within the operating system by a variety of ad-hoc mechanisms. Other shared resources either employ independent machinery (*e.g.,* memory, caches), or are deemed so abundant as to require no explicit management at all (*e.g.,* I/O, network bandwidth). Priority approaches have no mechanism to understand deadlines or the resources required to meet a deadline and as such must run the highest priority applications as fast as possible on all the resources requested. As a result, interactive and realtime applications are often run needlessly fast with significantly over-provisioned resources –wasting power and energy and preventing other applications from using the resources.

Some mobile systems have gone so far as to limit which applications can run in the background [2] in order to preserve responsiveness and battery life, despite the obvious concerns this raises for user experience. Cloud computing providers routinely utilize their clusters at only 10% to 50% to keep the system responsive despite the additional operational costs of consuming electricity and the significant impact to the capital costs of the infrastructure [3,26]. In some cases, clusters only run a single application on each cluster to avoid unexpected interference. Similarly the realtime community has used completely separate systems for each application to provide QoS despite the high-cost of specialization and low utilization with this approach.

In this paper, we present PACORA, a resource allocation framework, which is designed to provide responsiveness guarantees to a simultaneous mix of high-throughput parallel, interactive, and real-time applications in an efficient, scalable manner. Unlike traditional systems, in order to provide predictable responsive times PACORA considers all resource types when making decisions–providing the complete set of resources an application will need to meet its deadline. PACORA builds application-specific models of applications through measurement to determine the resources required to meet the deadline and leverages convex optimization with these application performance models to determine the optimal amount of each resource to give each application, enabling the system to make trade-offs between application QoS/responsiveness, system performance, and energy efficiency.

We believe PACORA is applicable to many resource allocation scenarios including cloud providers determining how much to give each job to avoid violating SLAs, databases allocating resources to queries, and distributed embedded systems allocating bandwidth between devices and sensors. In this paper we choose to study client applications and implement PACORA in a general-purpose operating system because we believe this scenario has some of the most significant resource allocation challenges: the constantly changing environment requires low overhead and fast response times from PACORA; shared resources create more interference between applications; and the applications are more likely to be written by domain experts, thus less highly optimized.

Add Performance Numbers

2. PACORA Architecture

PACORA is a framework designed to determine the proper amount of each resource type (e.g., nodes, processor cores, cache slices, memory pages, various kinds of bandwidth) to assign to each application. For example, consider a scenario of a video conference, where each participant requires a separate, performance guaranteed video stream. New participants may join the conference and others leave, increasing or decreasing the number of streams running at any given time. Simultaneously, participants may be collaborating through browsers, watching shared video clips and web searching, while running compute-intensive background tasks such as updates, virus scans, or file indexing. Although it may be relatively straightforward to provide QoS guarantees for applications such as video streams, the real challenge is to do so without using static resource reservations that compromise system utilization. The purpose of PACORA is to dynamically assign resources to applications to guarantee QoS without overprovisioning and adapt these allocations as the applications in the system change.

For our purposes an application is the entity to which the system allocates resources: it can viewed as a complete application (*e.g.,* a video player), a component of an application (*e.g.,* a music synthesizer) or a process (*e.g.,* indexing) for operating systems, jobs for warehouse-scale computing, and distributed applications in distributed embedded systems.

PACORA is designed to work in systems with hierarchical scheduling. In a client system, this design just looks like classic two-level scheduling. PACORA makes allocation decisions about resources (*e.g.,* cores and memory pages), and micro-management of the resources within an application is left to user-level runtimes such as User Mode Scheduling in Windows [**?**] or Lithe [46] and memory managers. In a cloud environment, PACORA allocates resources (*e.g.,* nodes and storage) to jobs, and scheduling is left to other entities such as the MapReduce framework [16] or the operating system on the node. Separating resource allocation from scheduling enables the use of application-specific scheduling policies, which have the potential to be easier to design and more efficient than a general-purpose scheduler that has to work for everything. This approach leaves the system to focus on the problem of *how many* of each resource type to assign to each application.

PACORA takes a different approach to resource allocation than traditional systems relying heavily on application-specific functions built through measurement and convex optimization. PACORA formulates resource allocation as an optimization problem built from two types of application-specific functions: a Response Time Function and a Penalty function. The Response Time Function represents the measured performance of the application on different resources. The Penalty Function represents the user-level goals for the application (i.e., the deadline and how important it is to make that deadline). The following sections describe each of the functions and the optimization construction.

Response Time Functions

Response Time Functions (RTF) represent the expected *response time* of an application as a function of the resources allocated to the application. The response time is an application-specific measure of the performance of the application. For example, the response time of an application might be:

- The time from a mouse click to its result;
- The time to produce a frame;
- The time from a service request to its response;
- The time from job launch to job completion;
- The time to execute a specified amount of work.

The RTFs are built to be a convex function. All applications have a function of the same form but the application-specific weights are set using the performance history of the application. RTFs are designed to capture information such as how well an application scales with a particular resource. As a result, RTFs naturally support heterogeneity. Each CPU or GPU type is simply viewed as a different resource type by the system, and thus the RTFs will represent how effectively an application uses a particular type of core. Figure 1 shows two example RTFs we have created from applications we studied.

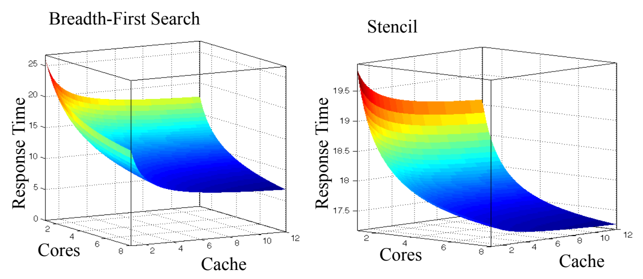

Figure 1: Response Time Functions for a breadth-first search and a stencil algorithm with 2 response dimensions: cores and cache slices.

Equation 1 below shows the form of the RTF we use in PACORA.

$$\tau(w,a) = \tau_0 + \sum_{i \varepsilon n, j \varepsilon n} \frac{w_{i,j}}{\sqrt{a_i * a_j}} \quad (1)$$

Here τ represents the response time, i and j represent resource types, n represents the total number of resource types, a_i and a_j represent the allocation of resources types i and j, and $w_{i,j}$ represents the application-specific weight for the term representing resource $i*$ resource j. The dimensionality of the function increases linearly with the number of resource types. The RTFs will be described further in Section 3.

Penalty Functions

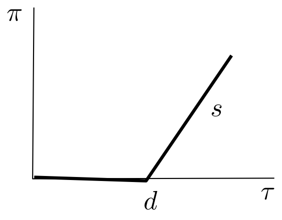

Figure 2: A penalty function with a response time constraint.

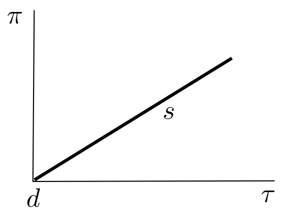

Figure 3: A penalty function with no response time constraint.

Penalty functions are designed to represent the user-level goals of the application. They are similar to the concept of priorities; however, penalty functions are a function rather than a single value and explicitly represent deadlines. Knowing the deadlines enables the system to make optimizations that are difficult in current systems such as running just fast enough to make the deadline. Like priorities, the penalty functions are set by the system in order to represent the relative importance of applications.

The penalty of the application is a function of the response time of the application. PACORA's penalty functions are non-decreasing piecewise linear functions of the form

$$\pi(\tau) = \max(0, s(\tau - d)) \quad (2)$$

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 is set to 0. Two representative graphs of this type appear in Figures 2 and 3. Penalty Functions will be described further in Section 3.

Resource Allocation As Optimization

PACORA formulates resource allocation as an optimization problem designed to minimize the total penalty of the system. This is analogous to minimizing user dissatisfaction with the user experience due to missed deadlines in a client system and minimizing the contract penalties paid for violated service-level agreements in a cloud system.

The optimization selects the allocations for all resources and resource types at once. This approach enables 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. It would difficult to provide predictable response times for applications considering the allocation of only one resource type at a time.

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

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

Here π_p is the penalty function for application p, τ_p is its response time function, $a_{p,r}$ is the allocation of resource r to application p, and A_r is the total amount of resource r available. Optimization details are described in Section 5.

Convex Optimization

If the penalty functions, response time functions, and resource constraints were arbitrary, little could be done to optimize the total penalty beyond searching at random for the best allocation. However, by framing our resource allocation problem as a *convex optimization problem* [9], two benefits accrue: an optimal solution will exist without multiple local extrema and fast, incremental solutions will become feasible.

A constrained optimization problem is convex if both the objective function to be minimized and the constraint functions that define its feasible solutions are convex functions. A function f is convex if its domain is a convex set and $F(\theta x + (1-\theta)y) \leq \theta F(x) + (1-\theta)F(y)$ for all θ between 0 and 1. A set is convex if for any two points x and y in the set, the point $\theta x + (1-\theta)y$ is also in the set for all θ between 0 and 1. If F is differentiable, it is convex if its domain is an open convex set and $F(y) \geq F(x) + \nabla F^T(y-x)$ where ∇F is the gradient of F.

Therefore, a problem will be a convex optimization problem if it can be expressed in the form:

$$\begin{aligned} \text{Minimize} \quad & f_0(x_1, \ldots x_m) \\ \text{Subject to} \quad & f_i(x_1, \ldots f_m) \leq 0, i = 1, \ldots k \\ \text{where} \quad & \forall i \quad f_i : \Re^m \to \Re \text{ is convex.} \end{aligned}$$

As a consequence, PACORA's resource allocation problem can be transformed into a convex optimization problem in the $m = |P| \cdot n$ variables $a_{p,r}$ as long as the penalty functions π_p are convex non-decreasing and the response time functions τ_p are convex. We designed our functions to meet these constraints and proofs of their convexity can be found in Apendix B. Note that the resource constraints are all affine and can be rewritten as $\sum_{p \varepsilon P} a_{p,r} - A_r \leq 0$ and $-a_{p,r} \leq 0$.

PACORA's convex formulation allows the time to solve the optimization to scale linearly with the number of resource types and the number of applications. For client operating systems with around 100 application running and 10 resource dimensions the total variables in the system to be solved is 1000 –a very small problem which can be solved in microseconds on current systems. Cloud systems could possibly have more applications running; however, the problem size scales linearly and the benefits gained could be significantly greater given the scale of the system.

Power and Energy

In PACORA, we create an artificial application to represent the interests of system power and energy. Application 0 is designated the idle application and receives allocations of all resources that are left idle, *i.e.,* not allocated to other applications. The idle resources can be powered off or put to sleep if possible to save power.

The "response time" for application 0, τ_0, is artificially defined to be the total system power consumption. The penalty function π_0 establishes a system tradeoff between power and performance that will determine which resources are allocated to applications to improve performance and which are left idle. The penalty function π_0 can be used to keep total system power below the parameter d_0 to the extent the penalties of other applications cannot overcome its penalty slope s_0. Both s_0 and d_0 can be adjusted to reflect the current battery charge in mobile devices. For example as the battery depletes, s_0 could be increased to force other applications to slow or cease execution.

Additionally Application 0 functions as *slack* variables in our optimization problem turning the resource bounds into equalities:

$$\sum_{p \varepsilon P} a_{p,r} - A_r = 0, r = 1, \ldots n. \tag{3}$$

Resources

In our client system, resources are anything that the system can provide QoS on in hardware or software: specifically we use cpus, network bandwidth, cache ways, and memory pages in our system. However, other resources could easily be added as well assuming they have QoS enforcement mechanisms. The other scenarios would have resources that perform similar functions (compute, network, capacity), but at a different scale. For warehouse-scale computing resources are more likely to be different types of nodes, network bandwidth, and storage, and distributed embedded systems would include compute devices, link bandwidth, and memory.

3. Application-Specific Functions

In this section, we describe the design of each of PACORA's application-specific functions in more detail.

Response Time Functions

A response time function describes an application's performance based on its resource assignments. These functions capture information about the value of a particular resource type to an application. How well an application scales on a particular resource is high depend on the specific application and how it was written along with performance properties of the resources, which is why we believe it is necessary to collection application-specific data for each application. Without information of this type, it would be difficult for any resource allocation system informed decisions. Thus forcing the system to be entirely reactive and most likely require a complex set of heuristics.

Model Selection While it might be possible to model response times by recording past values and interpolating among them, this idea has serious shortcomings:

- The size of the multidimensional response time function tables will be large;
- Interpolation in many dimensions is computationally expensive;
- The measurements will be noisy and require smoothing;
- Convexity in the resources may be violated;
- Gradient estimation will be difficult.

Alternatively, the resource manager could allocate a modest amount of a resource and measure the change in response time; however this is a difficult way to traverse a multidimensional space.

Instead of these, PACORA maintains a parameterized analytic response time model with the partial derivatives evaluated from the model *a priori*. Application responsiveness is highly nonlinear for an increasing variety of applications like streaming media or gaming, thus requiring many data points to represent the response times without a model. Using models each application can be described in a small number of parameters. Additionally models can be built from few data points and naturally smooth out noisy data. The resource gradient can be easily calculated which is essential for PACORA to solve the optimization problem efficiently.

PACORA models response times by functions that are convex by construction. The specific function chosen for PACORA is shown in Equation 1. In this equation, the response time is be modeled as a weighted sum of component terms, one per resource, where each term w_r/a_r is the amount of work $w_r \geq 0$ divided by a_r, the allocation of that resource [52]. For example, one term might model the number of instructions executed divided by total processor MIPS, another might model storage accesses divided by storage allocation and so forth. Asynchrony and latency tolerance may make response time components overlap partly or fully; and thus we added additional terms represent the interactions between resources. We choose this specific function because in initial application studies we found it models response time behavior accurately enough to allow the optimization to make good decisions but it also low overhead to build. Alternative models and the initial model evaluations are described in Appendix A.

Such models are automatically convex in the allocations because $1/a$ is convex for positive a and because a positively-weighted sum of convex functions is convex. It is obviously important to guarantee the positivity of the resource allocations. This can be enforced as the allocations are selected during penalty optimization, or the response time model can be made to return ∞ if any allocation is less than or equal to zero. This latter idea preserves the convexity of the model and extends its domain to all of $\Re^n$ and consequently we used this approach in our implementation. The gradient $\nabla\tau$ is needed by the penalty optimization algorithm. Since τ is analytic, generic, and symbolically differentiable it is a simple matter to compute the gradient of τ once the model is defined.

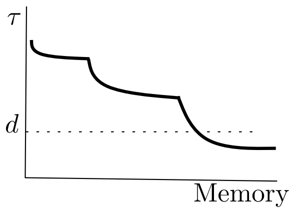

Figure 4: Response time function with some resource "plateaus".

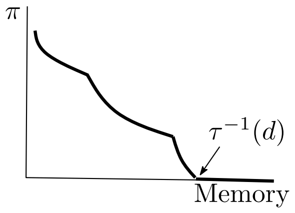

Figure 5: Net effect of the resource plateaus on the application penalty.

Convexity Assumption Forcing our response time functions to be convex assumes that the actual response time are reasonably convex; We find is a plausible requirement as applications almost completely follow the proverbial "Law of Diminishing Returns" for resource allocations.

However, there are examples of response time versus resource behavior that violate convexity. For example, we have particularly seen non-monotonic performance behavior in applications when dealing with hyperthreads or memory pages. For two of our applications 5 hyperthreads resulted in significantly worse performance than 4 or 6 hyperthreads. When studying some other applications applications we found that particular numbers of memory pages, *i.e.,* 2K, resulted in much better performance than the adjacent page allocations. One possible solution to solve this problem is keep track of points with significant error in the models and adjust resource allocations slightly to avoid these points.

Another such example sometimes occurs in memory allocation, where "plateaus" can sometimes be seen as in Figure 4. Such plateaus are typically caused by algorithm adaptations within the application to accommodate variable resource availability. The response time is really the *minimum* of several convex functions depending on allocation and the point-wise minimum that the application implements fails to preserve convexity. The effect of the plateaus will be a non-convex penalty as shown in Figure 5 and multiple extrema in the optimization problem will be a likely result. There are several ways to avoid this problem. One is based on the observation that such response time functions will at least be *quasiconvex*. Another idea is to use additional constraints to explore convex sub-domains of τ.

Overall, we found that our simple convex models still resulted in high-quality resource allocations and thus chose not to implement any of these approaches. Alternative approaches to handling non-convex behavior are described in Appendix B. Additional challenges to response time modeling are discussed in Section 7.

Data Collection and Creation Time There are several possible ways to collect the response time data for applications. A user-level runtime scheduler that schedules work internal to the application may be a good source of data or the operating system could measure progress using performance counters. In our implementation applications report their own measured values: however, this solution was chosen simply as a way to test the validity of the concept. In a production operating system, it may not be the best approach because it requires trusting applications not to lie about their performance. In a datacenter environment this may be less of a concern.

There are also many different possible times response time functions could be created. RTFs could be created in advance and distributed with the application. In the case of app stores this approach could make lot of sense since most app stores only cater to a limited number of platforms. Data could also be crowd sourced and the RTFs built in the cloud, which has the advantage making it easy to collect a diverse set of training points. However, all of these approaches are missing personalization. As a result we have chosen to implement two solutions that collect data directly from the users machine. The first approach is to collect all of the training points all the application install time and build the model then. The more advanced approached collects data continuously as the application runs and adds that data to the model training set and rebuilds the model. Although ultimately a hybrid approach may be the most effective: applications can begin with a provided generic model, and the system can improve the model over time. Section 4 describes our model creation process in detail.

Penalty Functions

Penalty functions are generically defined as members of a family of such functions so that user preferences for an application p can implemented by assigning values to a few well-understood parameters.

In traditional systems, responsiveness has been described by a single value (usually called a *priority*) associated with a thread of computation and adjusted within the operating system by a variety of ad-hoc mechanisms. However for many interactive real-time applications, performance is measured as sufficient if the deadline is met and insufficient otherwise. Priority approaches have no mechanism to understand deadlines or the resources required to meet a deadline.

We designed PACORA's penalty functions to represent deadlines and relative importance of making the deadline.

These values are represented by two parameters a deadline d and a slope s. Equation 2 shows penalty functions used in PACORA.

A response-time constrained application has a marked change in slope, namely from 0 to s, at the point $\tau = d$. In the most extreme case $s = \infty$ (implying infinite penalty for the system as a whole when $\tau > d$). For applications without response time constraints one can set $d = 0$. This defines linear behavior with s as the rate of penalty increase with response time.

To guarantee it is convex and non-decreasing, s must be non-negative. The response time τ is of course non-negative, and it may be sensible (if not strictly necessary) to convene that d is also.

The gradient of application penalty with respect to its resource allocations is necessary in controlling the optimization algorithm. By the chain rule, $\partial\pi/\partial a_r = \partial\pi/\partial\tau \cdot \partial\tau/\partial a_r$. The first term is well-defined but discontinuous at $\tau = d$ with $\partial\pi/\partial\tau =$ if $(\tau - d) \leq 0$ then 0 else s.

In a client operating system, the instantaneous management of penalty function modifications should be highly automated by the system to avoid unduly burdening the user. Most easily parameters could be set similarly to how priorities are set in some systems today. Applications are grouped into *interaction classes* based on application type and the interaction class defines the deadline and slope. In our implementation we set the parameters manually. However, future work is to experiment with alternative methods for *learning* the deadline and slope.

As a application grows or diminishes in importance, its penalty function can be modified accordingly. Penalty function adjustment is most likely to occur in transitions between operating scenarios. For example, when unplugging a device all the of the background activities could have their slopes significantly reduced to save battery life.

Power and Battery Energy

In order to control power and energy of the system, we use an artificial application named Application 0 which receives all resources not allocated to other applications. Application 0's response time function is similar to the other applications' response time functions. The function inputs are resource allocations just as with the other applications. However, the function output is system power rather than response time. To create the RTF, system power can be measured directly from on-chip energy counters in systems that they are available or from a power meter. These models can be built in advance, durning a training phase or online while the system runs, just as with the application RTFs. Alternatively, the model could be part of the operating system platform-specific information.

Although system power may not be entirely convex in reality, approximating it to be convex is reasonable because the convex model still captures the general behavior that leaving a resource idle should save some or no power. As a result, Application 0 still fills its purpose of preventing applications using additional resources that have low performance/power ratios.

The penalty function can be adjusted to represent how important saving power and energy is in the current operating scenario. For example, if a mobile device is plugged in to a charger than perhaps the penalty slope should be set very low. However if the device is unplugged the penalty function could be changed to have a deadline and then an steep slope which represents creating a power cap for the system. In this way the deadline and slope could be set to help control the minimum number of hours the battery will last.

4. Response Time Model Creation

Response Time Modeling

To create RTF models either at install time or online we use a least squares approach described a below. For the install time approach we use a genetic algorithm, Audze-Eglasis Design of Experiments [5], to select the resource vectors to use for training data. The data for all those vectors is collected and fed into the least squares algorithm. In the online approach the data comes from the application's response time history.

Model Creation using Least Squares After sufficient measurements, discovery of the model parameters w that define the function τ can be based on a solution to the over-determined linear system $t = Dw$ where t is a column vector of actual response times measured for the application and D is a matrix whose ith row $D_{i,*}$ contains the reciprocals of the amplified bandwidth allocations that generated the corresponding response time measurement t_i. Estimating w is relatively straightforward: a least-squares solution accomplished via *QR factorization* [22] of D will determine the w that minimizes (half the) square of the $r$esidual error $1/2\|Dw - t\|_2^2 = 1/2\|\varepsilon\|_2^2$. The solution proceeds as follows:

$$\begin{aligned} t &= Dw - \varepsilon \\ &= QRw - \varepsilon \\ Q^T t &= Rw - Q^T\varepsilon \end{aligned}$$

It is not always necessary to materialize the orthogonal matrix $Q^T = Q^{-1}$; the individual elementary orthogonal transformations (Householder reflections or Givens rotations) that triangularize R by progressively zeroing out partial columns of D can simultaneously be applied to t. The elements of the resulting vector $Q^T t$ that correspond to zero rows in R comprise $-Q^T\varepsilon$. Since Rw exactly equals the upper part of $Q^T t$, the upper part of $Q^T\varepsilon$ is zero. The residual error for the t_i can be found by premultiplying $Q^T\varepsilon$ by Q.

This formulation assumes a model norm $p = 1$. If a different model norm p is desired, such as $p = 2$, we could first square each measurement in t and each reciprocal bandwidth term in D and then follow the foregoing procedure. The elements of the result w will be squares as well, and the 2-norm of the difference in the squared quantities will be minimized. This is not the same as minimizing the 4-norm; what is being minimized is $1/2\|\mathrm{diag}(Dww^T D^T - tt^T)\|_2^2$.

On-line Response Time Modeling

As resource allocation continues, more measurements will become available to augment t and D. Moreover, older data may become a poor representation of the current behavior of the application if its characteristics have changed, presumably as reflected in $Q^T\varepsilon$. PACORA adopts the incremental least squares described below to replace old data and efficiently update RTFs.

Incremental Least Squares What is needed is a factorization $\tilde{Q}\tilde{R}$ of a new matrix $\tilde{D}$ derived from D by dropping a row, perhaps from the bottom, and adding a row, perhaps at the top. Corresponding elements of t are dropped and added to form $\tilde{t}$.

The matrices $\tilde{Q}$ and $\tilde{R}$ can be generated by applying Givens rotations in the way described in Section 12 of [22] to *downdate* or *update* the factorization much more cheaply than recomputing it *ab initio*. The method requires retention and maintenance of Q^T but not of D. Every update in PACORA is preceded by a downdate that makes room for it. Downdated rows are *not* always the oldest (bottom) ones, but an update always adds a new top row. For several reasons, the number of rows m in R will be at least twice the number of columns n. Rows selected for downdating will always be in the lower $m-n$ rows of R, guaranteeing that the most recent n updates are always part of the model.

Non-Negativity To guarantee convexity of the response time model, the solution w to $t \approx QRw$ must have no negative components. Intuitively, when a resource is associated with more than a single w_j or when the measured response time increases with allocation then negative w_j may occur.

A requirement for non-negative solutions to least-squares linear algebra problems is common, so much so that it has a name: *Non Negative Least Squares*, or NNLS. There are several well-known techniques [12], but since the method proposed here for online model maintenance calls for incremental downdates and updates to rows of Q^T, $Q^T t$ and R, the NNLS problem is handled with a complementary scheme based on the *active-set* method [36] that also downdates and updates the *columns* of R incrementally, roughly in the spirit of Algorithm 3 in [39]. However, PACORA's algorithm cannot ignore downdated columns of R because subsequent *row* updates and downdates must have due effect on these columns to allow their later reintroduction via column updates when necessary. /pacora solves this problem by leaving the downdated columns in place, skipping over them in maintaining and using the QR factorization.

Model Rank Preservation If care is not taken in the allocation process, the rows of R may become linearly dependent to such an extent that its rank is insufficient to determine w. This might be the result of repetitions in resource assignment updates. There are several possible ways to avoid this *rank-deficiency* problem. The characteristics of R depend on both the resource optimization trajectory and the choices made in the downdate-update algorithm. In particular, deciding whether to downdate the bottom row of R or some "younger" row will depend on whether the result would become rank-deficient. This approach decouples allocation optimization from performance model maintenance and places responsibility upon the latter to always keep enough history to determine a model.

Outliers and Phase Changes Some response time measurements may be "noisy" or even erroneous. A weakness of least-squares modeling is the high importance it gives to outlying values. On the other hand, when an application changes phase it is important to adapt quickly, and what looks like an outlier when it first appears may be a harbinger of change. What is needed is a way to discard either old or outlying data with a judicious balance between age and anomaly.

The downdating algorithm accomplishes this by weighting the errors in $\varepsilon = Q(Q^T t - Rw)$ between the predicted response times τ and the measured ones t by a factor that increases exponentially with the age $g(i)$ of the error ε_i. Age can be modeled coarsely by the number of time quanta of some size since the measurement; PACORA simply lets $g(i) = i$. The weighting factor for the ith row is then $\eta^{g(i)}$ where η is a constant somewhat greater than 1. The candidate row to downdate is the row with the largest weighted error, *i.e.*

$$dd = \arg\max_i |\varepsilon_i| \cdot \eta^{g(i)}$$

5. Dynamic Penalty Optimization

PACORA's penalty optimization algorithm dynamically decides resource allocations. The algorithm can be run periodically, when applications start or stop, when an application changes faces or when the system changes operating scenarios. One of the advantages of convex optimization is that it enables fast, incremental solutions. As a result the algorithm could terminal earlier to decrease overhead and still be moving towards an optimal solution as it runs. However we found in our implementation that the algorithm was fast enough to run to completion every time.

Convex optimization is simplest when it is unconstrained, and so we reformulated PACORA's construction to be unconstrained. Extending the response time model functions to all of $\Re^n$ moves the requirement that allocations must be positive into the objective function, and introducing Application 0 for slack resources turns the affine inequalities into equalities:

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

The only remaining constraints are those on the $a_{p,r}$. These can be removed by letting the $a_{p,r}$ be unbounded above for

$p \neq 0$ and changing the domain of τ_0 to be the whole resource allocation matrix. The definition of τ_0 might take the form

$$\begin{aligned} \tau_0 &= \sum_r \Delta_r (A_r - a_{0,r}) \\ &= \sum_r \Delta_r \sum_{p \neq 0} a_{p,r} \end{aligned}$$

where Δ_r is the (constant) power dissipation of one unit of resource r. However,if any of the allocations $a_{0,r}$ is negative then τ_0 should instead return the value $+\infty$. This modification of the objective function transforms the resource allocation problem to unconstrained convex optimization.

The penalty optimization algorithm used in PACORA is descent via backtracking line search along the negative gradient direction [9]. This algorithm rejects and refines any step that yields insufficient relative improvement in the objective function, so infinite values from infeasible allocations will automatically be avoided by the search. The negative gradient $-\nabla\pi$ of the overall objective function π with respect to the resource allocations a is computed analytically from the response time models and penalty functions. When a component of this overall gradient is negative, it means the penalty will be reduced by increasing the associated allocation if possible. The gradient search at the boundaries of the feasible region must ignore components that lead in infeasible directions; these can be detected by noting whether for some p and r, $a_{p,r} = 0$ with $(-\nabla\pi)_{p,r} > 0$. In such cases, the associated step component is set to zero.

We added an additional optimization to move along boundaries more rapidly quickly in the scenario when completely allocated resource had a large gradient. We scale all the resource allocations of that resource type down to fit into the resource constraint while leaving the resource allocations of over resources untouched.

The rate of convergence of gradient descent depends on how well the sublevel sets of the objective function are conditioned (basically, how "spherical" they are). Conditioning will improve if resource allocation units are scaled to make their relative effects on t similar. For example, when compared with processor allocation units, memory allocation units of 4MB are probably a better choice than 4 KB. In addition, penalty function slopes should not differ by more than perhaps two orders of magnitude. If these measures prove insufficient, stronger preconditioners can be used. In our implementation, we condition all resource allocations to be in the range of 0-50.

6. Evaluation

[38] [53] [6] [7] [46] [40] [15] [5]

PULSE (Preemptive User-Level SchEduling) Lithe (LIquid THrEads)

6.1. Experimental Methodology

To evaluate PACORA, we use two different implementations of the framework on the same hardware platform. Our offline framework was used to experiment with the accuracy of different types of models and test

2 implementations

6.1.1. Experimental Platform We use a prototype version of Intel's Sandy Bridge x86 processor to collect results on resource allocation and application co-scheduling. By using a real hardware prototype, we are able to run full applications for realistic time scales and workload sizes, while running a standard operating system. The processor is similar to the commercially available client chip, but with additional hardware to support way-based cache partitioning in the LLC.

The Sandy Bridge client chip has four quad-issue out-of-order superscalar cores, each of which supports two hyperthreads using simultaneous multithreading [?]. Each core has private 32 KB instruction and data caches, as well as a 256 KB private non-inclusive L2 cache. The LLC is a 12-way set-associative 6 MB inclusive L3 cache, shared among all cores using a ring-based interconnect. All three cache levels are write-back. Larger server versions of the same processor family have up to 15 MB of LLC capacity.

The cache partitioning mechanism is way-based and modifies the cache-replacement algorithm. Each core can be assigned a subset of the 12 ways in the LLC. Although all cores can hit on data stored in any way, a core can only replace data stored in one of its assigned ways. Allocation of ways among cores can be completely private, completely shared, or overlapping. Data is not flushed when the way allocation changes; newly fetched data will just be written into one of the assigned ways according to the updated allocation configuration.

We use a customized BIOS that enables the cache partitioning mechanism,

To measure on-chip energy, we use the energy counters available on Sandy Bridge to collect the energy used by the entire socket and also the total combined energy of cores, their private caches, and the LLC. The counters measure power at a $1/2^{16}$ second granularity.

6.2. Linux Analysis

and run unmodified Linux-2.6.36 for all of our experiments. We use the Linux `taskset` command to pin each application to subsets of the available HyperThreads.

To measure application performance, we use the `libpfm` library [?, ?], built on top of the `perf_events` infrastructure introduced in Linux 2.6.31, to access various performance-monitoring counters available on the machine [?].

We access these counters using the Running Average Power Limit (RAPL) interfaces [?].

6.2.1. Description of Workloads We built our workload using a wide range of codes from three different popular benchmark suites: SPEC CPU 2006 [53], DaCapo [?], and PARSEC [?]. We included some additional application benchmarks to broaden the scope of the study, and some microbenchmarks that exercise certain features of the system.

The **SPEC CPU 2006** benchmark suite [53] is a CPU-intensive, single-threaded benchmark suite, designed to stress a system's processor, memory subsystem and compiler. Using the similarity analysis performed by Phansalkar et al. [47], we subset the suite, selecting 4 integer benchmarks (astar, libquantum, mcf, omnetpp) and 4 floating-point benchmarks (cactusADM, calculix, lbm, povray). Based on the characterization study by Jaleel [?], we also pick 4 extra floating-point benchmarks that stress the LLC: GemsFDTD, leslie3d, soplex and sphinx3. When multiple input sets and sizes are available, we pick the single *ref* input indicated by Phansalkar et al. [47]. SPEC is the only benchmark suite used in many previous characterizations of LLC partitioning [?, ?, ?].

The **DaCapo** benchmark suite is intended as a tool for Java benchmarking, consisting of a set of open-source, real-world applications with non-trivial memory loads, including both client and server-side applications. We used the latest 2009 release. The managed nature of the DaCapo runtime environment has been shown to make a significant difference in some scheduling studies [?], and is also representative of the increasing relevance of such runtimes.

The **PARSEC** benchmark suite is intended to be representative of parallel real-world applications [?]. PARSEC programs use various parallelization approaches, including data- and task-parallelization. We use the version of the benchmarks parallelized with the `pthreads` library, with the exception of `freqmine`, which is only available in OpenMP. We used the full native input sets for all the experiments. Past characterizations of PARSEC have found it to be sensitive to available cache capacity [?], but also resilient to performance degradation in the face of intra-application sharing of caches [?].

We added four **additional parallel applications** to help ensure we covered the space of interest: *Browser_animation* is a multithreaded kernel representing a browser layout animation; *G500_csr* code is a kernel performing breadth-first search of a large graph for the Graph500 contest, based on a new hybrid algorithm [?]; *Paradecoder* is a parallel speech-recognition application that takes audio waveforms of human speech and infers the most likely word sequence intended by the speaker; *Stencilprobe* simulates heat transfer through a fluid using a parallel stencil kernel over a regular grid [?].

6.2.2. Response-Time Functions

6.3. Manycore OS Implementation

6.3.1. Description of Workloads

6.3.2. Response-Time Functions

6.3.3. Resource Allocation Decisions

7. Discussion

Program phase detection has targeted hardware or software reconfiguration. The overview [17] evaluates three measurement-based predictors, and while performance variation is not one of the three, the authors note that small performance variation is an indicator of correct phase identification. A similar conclusion is reached in [50].

8. Related Work

Add Gang Scheduling? Discuss Application Performance Modeling

8.1. Autonomic Systems

In much of this research area [33, 43, 56, 57] the performance models are derived from off-line measurements and are either interpolations from tables or analytic functions based on queueing theory. The utility functions typically map the number of servers each execution environment receives to its performance relative to its requirements, which may be multiple. A central arbiter maximizes total utility. The utility functions are not necessarily concave, so the arbiter must use reinforcement learning or combinatorial search to make allocations. Walsh *et al.* [58] note the importance of basing utility functions on the metrics in which QoS is expressed rather than on the raw quantities of resources. Each application has a manager that schedules the resources given to it by a global arbiter. There are other philosophical similarities to PACORA, but since the objective functions are neither continuous nor convex their optimization is difficult.

The related systems AcOS [4] and Metronome [51] feature hardware-thread based maintenance of "Heart Rate" targets using adaptive reinforcement learning techniques. AcOS also senses thermal conditions and can exploit Dynamic Voltage and Frequency Scaling (DVFS).

A survey of autonomic systems research appears in [28].

8.2. Soft Real-Time and SLAs

Soft real-time systems typically modify an existing operating system to improve responsiveness. Calandrino *et al.* [10] use working set sizes to make co-scheduling decisions and enhance soft real-time behavior.

Rajkumar *et al.* [49] propose a system Q-RAM that maximizes the weighted sum of utility functions, each of which is a function of the resource allocation to the associated application. Unlike PACORA, there is no distinction between performance and utility, and the utility functions are assumed as input rather than being discovered by the system. They are always nondecreasing in every resource and sometimes are piecewise-linear and concave; in these cases the optimal allocation is easily found by a form of gradient ascent. When the utility functions are not concave, a suboptimal greedy algorithm is proposed.

In the Redline system [60], compact resource requirement specifications are written by hand to guarantee response times. Isolation of resources is strong, as in PACORA. Scheduling is Earliest-Deadline-First. Admission control is lenient but oversubscription situations are remedied by de-admitting some of the non-interactive applications.

Jockey [19] has some similarities to PACORA: it is intended to handle parallel computation, its utility functions are concave, and it adapts dynamically to application behavior. Its performance models are obtained by calibrating either event-based simulation or a version of Amdahl's Law to computations. Jockey does not optimize total utility but simply increases processors until utility flattens for each application, *i.e.* each deadline is met. A fairly sophisticated control loop prevents oscillatory behavior.

Bodik *et al.* [8] is also like PACORA in that it builds online performance models. Initially, it uses an *exploration policy* that avoids nearly all SLA violations while it builds the model; later, it shifts to a controller based on the model it has built. The models are statistical, and bootstrapping is used to estimate performance variance. Major changes in the application model are detected and cause model exploration to resume. The models are not convex or concave in general, and all SLAs must be met with high probability.

8.3. Partitioning Mechanisms

Most hardware partitioning mechanism work has looked at shared cache structures and provided mechanisms to partition them according to a varied set of goals. Suh *et al.* [54, 55] and Qureshi and Patt [48] monitor individual applications' cache performance and use this monitoring to inform their partitioning mechanism in an attempt to reduce the total amount of cache misses and off-chip memory traffic. A wide variety of proposals exist for multicore last-level cache structures that partition the spatial resources between private and shared data, in an attempt to create a manageable trade-off between capacity for shared data and low latency for private data [11, 14, 18, 29, 41, 62, 63].

Further cache partitioning work has focused on providing QoS guarantees to applications. Early work focused on providing adaptive, fair policies that ensure equal performance degradation [32, 61], while more recent proposals have incorporated more sophisticated policy management [23, 24, 27, 31]. Other partitioning work has focused on interconnect bandwidth QoS [37] or partitioning cache capacity and bandwidth simultaneously [44]. In general, these papers focus on designing and proving the effectiveness of particular mechanisms for particular goals, without a concrete notion of a general framework in which a variety of application-specific QoS requirements can be communicated to an all-purpose resource allocator and scheduler. For example, Iyer [30] suggests a priority-based cache QoS framework, CQoS, for shared cache way-partitioning. The priorities might be specified per core, per application, per memory type, or even per memory reference. Simultaneous achievement of performance targets is not addressed.

8.4. Resource Allocation Frameworks

Guo *et al.* [24] point out that much prior work is insufficient for true QoS – merely partitioning hardware is not enough, because there must also be a way to specify performance targets and an admission control policy for jobs. The framework they present incorporates a scheduler that supports multiple execution modes.

Nesbit *et al.* [45] introduce the *Virtual Private Machines* (VPM), a framework for resource allocation and management in multicore systems. A VPM consists of a set of virtual hardware resources, both spatial (physical allocations) and temporal (scheduled time-slices). Unlike traditional virtual machines that only virtualize resource functionality, VPMs virtualize a system's performance and power characteristics, meaning that a VPM has the same performance and power profile as a real machine with an equivalent set of hardware resources.

They break down the framework components into policies and mechanisms which may be implemented in hardware or software. Critical software components of the VPM framework are VPM *modeling*, which maps high-level application objectives into VPM configurations, and VPM *translation*, which uses VPM models to assign an acceptable VPM to the application while adhering to system-level policies. A VPM scheduler then decides if the system can accommodate all applications or whether resources need to be revoked.

The VPM approach and terminology mesh well with our study, which can be seen as a specific implementation of several key aspects of the type of framework they describe (i.e. VPM modeling and translation). Nesbit *et al.* did not perform any evaluations of the modeling, translation, or scheduling processes suggested in their paper.

The Singularity operating system [1] provides process isolation through software rather than hardware. This isolation in accessibility is not the same as the performance isolation that is desirable for PACORA.

Kumar *et al.* [34] demonstrate the performance advantages of heterogeneous cores for mixed workloads using heuristic allocation strategies in both space and time.

Genbrugge and Eeckhout [21] demonstrate the importance of adapting to changes in application characteristics, in this case instructions per cycle.

Merkel and Bellosa [42] propose *Task Activity Vectors* that describe how much each application uses the various functional units; these vectors are used to balance usage across multiple cores and unbalance usage among hardware threads within each core. The intended effect is to distribute chip temperature more evenly, but the idea may be more broadly applicable, *e.g.* for heterogeneous systems.

Ganapathi *et al.* have had success using machine learning to model application performance and select the best performing configuration in [20].

I DON'T KNOW WHY [25,59] ARE HERE.

Chen *et al.* [13] propose parallel depth-first thread scheduling as an alternative to work-stealing for constructive cache sharing. The paper discusses the automation of task granularity selection to match cache capacity. This kind of tuning is particularly appropriate for a user-level runtime system.

9. Conclusion

References

[1] M. Aiken *et al.*, "Deconstructing process isolation," in *Memory Systems Performance and Correctness*, 2006.

[2] Apple Inc., "iOS App Programming Guide," http://developer.apple.com/library/ios/DOCUMENTATION/iPhone/Conceptual/iPhoneOSProgrammingGuide/iPhoneAppProgrammingGuide.pdf.

[3] L. A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[4] D. B. Bartolini *et al.*, "Acos: an autonomic management layer enhancing commodity operating systems," in *In DAC Workshop on Computing in Heterogeneous, Autonomous, 'N' Goal-oriented Environments (CHANGE), co-located with the Annual Design Automation Conference (DAC)*, 2012.

[5] S. J. Bates, J. Sienz, and D. S. Langley, "Formulation of the audze–eglais uniform latin hypercube design of experiments," *Adv. Eng. Softw.*, vol. 34, no. 8, pp. 493–506, 2003.

[6] C. Bienia *et al.*, "The parsec benchmark suite: Characterization and the parsec benchmark suite: Characterization and architectural implications," Princeton University, Tech. Rep. TR-811-08, January 2008.

[7] S. M. Blackburn *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA*, 2006, pp. 169–190.

[8] P. Bodik *et al.*, "Automatic exploration of datacenter performance regimes," in *Proc. ACDC*, 2009.

[9] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, England: Cambridge University Press, 2004.

[10] J. M. Calandrino and J. H. Anderson, "On the design and implementation of a cache-aware multicore real-time scheduler," in *ECRTS*, 2009, pp. 194–204.

[11] J. Chang and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors," in *Proc. ICS '07*. New York, NY, USA: ACM, 2007, pp. 242–252.

[12] D. Chen and R. J. Plemmons, "Nonnegativity constraints in numerical analysis," Lecture presented at the symposium to celebrate the 60th birthday of numerical analysis, Leuven, Belgium, October 2007.

[13] S. Chen *et al.*, "Scheduling threads for constructive cache sharing on CMPs," in *Proc. SPAA '07*. New York, NY, USA: ACM, 2007, pp. 105–115.

[14] S. Cho and L. Jin, "Managing distributed, shared l2 caches through os-level page allocation," in *Proc. MICRO 39*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 455–468.

[15] I. CVX Research, "CVX: Matlab software for disciplined convex programming, version 2.0 beta," http://cvxr.com/cvx, Sep. 2012.

[16] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. Available: http://doi.acm.org/10.1145/1327452.1327492

[17] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *Proc. MICRO 36*. Washington, DC, USA: IEEE Computer Society, 2003, p. 217.

[18] H. Dybdahl and P. Stenstrom, "An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors," in *Proc. HPCA '07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 2–12.

[19] A. D. Ferguson *et al.*, "Jockey: guaranteed job latency in data parallel clusters," in *Proceedings of the 7th ACM european conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 99–112. Available: http://doi.acm.org/10.1145/2168836.2168847

[20] A. Ganapathi *et al.*, "A case for machine learning to optimize multicore performance," in *HotPar09*, Berkeley, CA, 3/2009 2009. Available: http://www.usenix.org/event/hotpar09/tech/

[21] D. Genbrugge and L. Eeckhout, "Statistical simulation of chip multiprocessors running multi-program workloads," in *ISCA*, 2007.

[22] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. Baltimore, Maryland: Johns Hopkins University Press, 1996.

[23] F. Guo *et al.*, "From chaos to qos: case studies in cmp resource management," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 21–30, 2007.

[24] F. Guo *et al.*, "A framework for providing quality of service in chip multi-processors," in *Proc. MICRO '07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 343–355.

[25] J. L. Hellerstein *et al.*, *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[26] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

[27] L. R. Hsu *et al.*, "Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource," in *Proc. PACT '06*. New York, NY, USA: ACM, 2006, pp. 13–22.

[28] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing—degrees, models, and applications," *ACM Comput. Surv.*, vol. 40, no. 3, pp. 1–28, 2008.

[29] J. Huh *et al.*, "A nuca substrate for flexible cmp cache sharing," in *Proc. ICS '05*. New York, NY, USA: ACM, 2005, pp. 31–40.

[30] R. Iyer, "Cqos: a framework for enabling qos in shared caches of cmp platforms," in *Proc. ICS '04*. New York, NY, USA: ACM, 2004, pp. 257–266.

[31] R. Iyer *et al.*, "Qos policies and architecture for cache/memory in cmp platforms," in *Proc. SIGMETRICS '07*. New York, NY, USA: ACM, 2007, pp. 25–36.

[32] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proc. ASPLOS-X*. New York, NY, USA: ACM, 2002, pp. 211–222.

[33] S. Kounev, R. Nou, and J. Torres, "Autonomic qos-aware resource management in grid computing using online performance models," in *Proc. ValueTools '07*. ICST, 2007, pp. 1–10.

[34] R. Kumar *et al.*, "Single-isa heterogeneous multi-core architectures for multithreaded workload performance," in *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 2004, p. 64.

[35] H. T. Kung, "Memory requirements for balanced computer architectures," in *International Symposium on Computer Architecture*, 1986, pp. 49–54.

[36] C. L. Lawson and R. J. Hanson, *Solving Least Squares Problems*. Englewood Cliffs, NJ: Prentice Hall, 1974.

[37] J. W. Lee, M. C. Ng, and K. Asanovic, "Globally-synchronized frames for guaranteed quality-of-service in on-chip networks," in *Proc. ISCA '08*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 89–100.

[38] R. Liu *et al.*, "Tessellation: Space-time partitioning in a manycore client os," in *HotPar09*, Berkeley, CA, 03/2009 2009. Available: http://www.usenix.org/event/hotpar09/tech/

[39] Y. Luo and R. Duraiswami, "Efficient parallel nonnegative least squares on multicore architectures," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2848–2863, 2011.

[40] Mathworks, "Matlab 2009b," http://www.mathworks.com/, 2009.

[41] J. Merino *et al.*, "Sp-nuca: a cost effective dynamic non-uniform cache architecture," *SIGARCH Comput. Archit. News*, vol. 36, no. 2, pp. 64–71, 2008.

[42] A. Merkel and F. Bellosa, "Task activity vectors: a new metric for temperature-aware scheduling," in *Proc. Eurosys '08*. New York, NY, USA: ACM, 2008, pp. 1–12. Available: http://portal.acm.org/ft_gateway.cfm?id=1352594&type=pdf&coll=GUIDE&dl=GUID

[43] M. N. Bennani and D. A. Menasce, "Resource allocation for autonomic data centers using analytic performance models," in *ICAC '05*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 229–240.

[44] K. J. Nesbit, J. Laudon, and J. E. Smith, "Virtual private caches," in *Proc. ISCA '07*. New York, NY, USA: ACM, 2007, pp. 57–68.

[45] K. J. Nesbit *et al.*, "Multicore resource management," *IEEE Micro*, vol. 28, no. 3, pp. 6–16, 2008.

[46] H. Pan, B. Hindman, and K. Asanović, "Lithe: Enabling efficient composition of parallel libraries," in *HotPar09*, Berkeley, CA, 03/2009 2009. Available: http://www.usenix.org/event/hotpar09/tech/

[47] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the spec cpu2006 benchmark suite," in *ISCA*, 2007, pp. 412–423.

[48] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. MICRO 39*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 423–432.

[49] R. Rajkumar *et al.*, "A resource allocation model for qos management," in *Proc. RTSS '97*. Washington, DC, USA: IEEE Computer Society, 1997, p. 298.

[50] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," *SIGARCH Comput. Archit. News*, vol. 31, no. 2, pp. 336–349, 2003.

[51] F. Sironi *et al.*, "Metronome: operating system level performance management via self-adaptive computing," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 856–865. Available: http://doi.acm.org/10.1145/2228360.2228514

[52] A. Snavely *et al.*, "A framework for performance modeling and prediction," in *SC*, 2002, pp. 1–17.

[53] Standard Performance Evaluation Corporation, "SPEC CPU 2006 benchmark suite," http://www.spec.org.

[54] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *J. Supercomput.*, vol. 28, no. 1, pp. 7–26, 2004.

[55] G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proc. HPCA '02*. Washington, DC, USA: IEEE Computer Society, 2002, p. 117.

[56] G. Tesauro, W. E. Walsh, and J. O. Kephart, "Utility-function-driven resource allocation in autonomic systems," in *Proc. ICAC '05*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 342–343.

[57] G. Tesauro *et al.*, "On the use of hybrid reinforcement learning for autonomic resource allocation," *Cluster Computing*, vol. 10, no. 3, pp. 287–299, 2007.

[58] G. Tesauro and J. O. Kephart, "Utility functions in autonomic systems," in *Proc. ICAC '04*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 70–77.

[59] L. Wasserman, *All of Nonparametric Statistics (Springer Texts in Statistics)*. Se- caucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[60] T. Yang *et al.*, "Redline: first class support for interactivity in commodity operating systems," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 73–86. Available: http://dl.acm.org/citation.cfm?id=1855741.1855747

[61] T. Y. Yeh and G. Reinman, "Fast and fair: data-stream quality of service," in *Proc. CASES '05*. New York, NY, USA: ACM, 2005, pp. 237–248.

[62] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *Proc. ISCA '05*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 336–345.

[63] L. Zhao *et al.*, "Towards hybrid last level caches for chip-multiprocessors," *SIGARCH Comput. Archit. News*, vol. 36, no. 2, pp. 56–63, 2008.

A. Response Time Model Design

Alternative Resource Representations

Asynchrony and latency tolerance may make response time components overlap partly or fully; if the latter, then the maximum of the terms might be more appropriate than their sum. The result will still be convex, though, as will any other norm including the 2-norm, *i.e.* the square root of the sum of the squares. This last variation could be viewed as a "partially overlapped" compromise between the 1-norm (sum) describing no overlap and the ∞-norm (maximum) describing full overlap.

Sometimes a response time component might be better modeled by a term involving a combination of resources. For example, response time due to memory accesses might be approximated by a combination of memory bandwidth allocation b_{r1} and cache allocation m_{r2}. Such a model could use the geometric mean of the two allocations in the denominator, *viz.* $w_{r1,r2}/\sqrt{b_{r1}\cdot m_{r2}}$, without compromising convexity.

This scheme also accommodates non-bandwidth resources such as memory, the general idea being to roughly approximate "diminishing returns" in the response time with increasing resources. For clarity's sake, rather than using a_r indiscriminately for all allocations, we will denote an allocation of a bandwidth resource by b_r and of a memory resource by m_r. This begs the question of how memory affects the response time. The effect is largely indirect. Memory permits exploitation of temporal locality and thereby *amplifies* associated bandwidths. For example, additional main memory may reduce the need for storage or network bandwidth, and of course increased cache capacity may reduce the need for memory bandwidth. The effectiveness of cache in reducing bandwidth was studied by H. T. Kung [35], who developed tight asymptotic bounds on the bandwidth amplification factor $\alpha(m)$ resulting from a quantity of memory m acting as cache for a variety of computations. He shows that

$$\begin{aligned}\alpha(m) &= \Theta(\sqrt{m}) && \text{for dense linear algebra solvers}\\ &= \Theta(m^{1/d}) && \text{for d-dimensional PDE solvers}\\ &= \Theta(\log m) && \text{for comparison sorting and FFTs}\\ &= \Theta(1) && \text{when temporal locality is absent}\end{aligned}$$

For these expressions to make sense, the argument of α should be dimensionless and greater than 1. Ensuring this might be as simple as letting it be the number of memory resource quanta (*e.g.* hundreds of memory pages) assigned to the application. If a application shows diminishing bandwidth amplification as its memory allocation increases, this can be accommodated:

$$\alpha(m) = \min(c_1\alpha_1(m), c_2\alpha_2(m)),\ c_1, c_2 \geq 0$$

Each bandwidth amplification factor might be described by one of the functions above and included in the denominator of the appropriate component in the response time function model. For example, the storage response time component for the model of an out-of-core sort application might be the quantity of storage accesses divided by the product of the storage bandwidth allocation and $\log m$, the amplification function associated with sorting given a memory allocation of m. Amplification functions for each application might be learned from response time measurements by observing the effect of varying the associated memory resource while keeping the bandwidth allocation constant. Alternatively, redundant components, similar except for amplification function, could be included in the model to let the model fitting process decide among them.

Initial Model Evaluation

B. Problem Convexity

Convex Functions

A few more facts about convex functions will be useful in what follows. First, a *concave* function is one whose negative is convex. Maximization of a concave function is equivalent to minimization of its convex negative. An affine function, one whose graph is a straight line in two dimensions or a hyperplane in n dimensions, is both convex and concave. A non-negative weighted sum or point-wise maximum (minimum) of convex (concave) functions is convex (concave), as is either kind of function composed with an affine function. The composition of a convex non-decreasing (concave non-increasing) scalar function with a convex function remains convex (concave).

Response Time Model Convexity

We now show that the response time model including the various bandwidth amplification functions is convex in both the bandwidth and memory resources b_r and m_r given any of the possibilities listed above. Since norms preserve convexity, this reduces the question to proving each term in the norm is convex. Since all quantities are positive and both maximum and scaling by a positive constant preserve convexity,

$$\begin{aligned} & w/(b \cdot \min(c_1\alpha_1(m), c_2\alpha_2(m))) \\ & \quad = \quad \max(w/(b \cdot c_1\alpha_1(m)), w/(b \cdot c_2\alpha_2(m))). \end{aligned}$$

It only remains to show that $1/(b \cdot \alpha(m))$ is convex in b and m.

A function is defined to be *log-convex* if its logarithm is convex. A log-convex function is itself convex because exponentiation preserves convexity, and the product of log-convex functions is convex because the log of the product is the sum of the logs, each of which is convex by hypothesis. Now $1/b$ is log-convex for $b > 0$ because $-\log b$ is convex on that domain. In a similar way, $\log(1/\sqrt{b \cdot m}) = -(\log b + \log m)/2$ and $\log m^{-1/d} = -(\log m)/d$ are convex. Finally, $\log(1/\log m)$ is convex because its second derivative is positive for $m > 1$:

$$\begin{aligned} \frac{d^2}{dm^2}\log(1/\log m) &= \frac{d^2}{dm^2}(-\log\log m) \\ &= \frac{d}{dm}\left(\frac{-1}{m\log m}\right) \\ &= \frac{1+\log m}{(m\log m)^2}. \end{aligned}$$

Summing up, a response time function for a application might be modeled by the convex function

$$\begin{aligned} \tau(w,b,\alpha,m) &= \sqrt[p]{\sum_j \left(\frac{w_j}{b_j \cdot \alpha_j(m_j)}\right)^p} \\ &= \|\mathrm{diag} w d^T\|_p \end{aligned}$$

where the w_j are the parameters of the model (the quantities of work) to be learned, the components of d satisfy $d_j = 1/(b_j \cdot \alpha_j(m_j))$, the b_j are the allocations of the bandwidth resources, the α_j are the bandwidth amplification functions (also to be learned), the m_j are the allocations of the memory or cache resources that are responsible for the amplifications. This formulation allows the application response time τ to be modeled as the p-norm of the component-wise product of a vector d that is computed from the resource allocation and a learned vector of work quantities w.

Handling Quasiconvex Response Time Functions

There are examples of response time versus resource behavior that violate convexity. One such example sometimes occurs in memory allocation, where "plateaus" can sometimes be seen as in Figure 4.

Such plateaus are typically caused by algorithm adaptations within the application to accommodate variable resource availability. The response time is really the *minimum* of several convex functions depending on allocation and the point-wise minimum that the application implements fails to preserve convexity. The effect of the plateaus will be a non-convex penalty as shown in Figure 5 and multiple extrema in the optimization problem will be a likely result.

There are several ways to avoid this problem. One is based on the observation that such response time functions will at least be *quasiconvex*. A function f is quasiconvex if all of its *sublevel sets* $S_\ell = \{x | f(x) \leq \ell\}$ are convex sets. Alternatively, f is quasiconvex if its domain is convex and

$$f(\theta x + (1-\theta)y) \leq \max(f(x), f(y)), 0 \leq \theta \leq 1$$

Quasiconvex optimization can be performed by selecting a threshold ℓ and replacing the objective function with a convex constraint function whose sublevel set S_ℓ is the same as that of f. Next, one determines whether there is a feasible solution for that particular threshold ℓ. Repeated application with a binary search on ℓ will reduce the level of feasibility until the solution is approximated well enough.

Another idea is to use additional constraints to explore convex sub-domains of τ. For example,the affine constraint $a_{p,r} - \mu \leq 0$ excludes application p from any assignment of resource r exceeding μ. Similarly, $\mu - a_{p,r} \leq 0$ excludes the opposite possibility. A binary (or even linear)search of such sub-domains could be used to find the optimal value.

C. Model Update and Downdate Algorithms

C.1. Row Update and Downdate

Downdating makes an instructive example. A row downdate operation applies a sequence of Givens rotations to the rows of Q^T. The rotations are calculated to set every $Q^T_{i,dd}$, $i \neq dd$ to zero. In the end only the diagonal element $Q^T_{dd,dd}$ of column dd will be nonzero. Since Q^T remains orthogonal, the non-diagonal elements of row dd will also have been zeroed automatically and the diagonal element will have absolute value 1. These same rotations are concurrently applied to the elements of $Q^T t$ and to the rows of R $(= Q^T D)$ to reflect the effect that these transformations have on Q^T.

It is crucial to select pairs of rows and an order of rotations that preserves the upper triangular structure of R while zeroing all but the diagonal entry of the chosen column dd of Q^T. Since dd is always below the diagonal of R it initially will contain only zeros. It is therefore sufficient to rotate every non-dd row with row dd, proceeding from bottom to top. The first $m-n-1$ rotations will keep row $R_{dd,*}$ entirely zero, and the remaining n rotations will introduce nonzeros in $R_{dd,*}$ from right to left. The effect on R will be to replace zero elements by nonzero elements only within row dd. At this point, except for a possible difference in overall sign, $R_{dd,*} = D_{dd,*}$.

Now the rows from 0 down through dd of the modified matrices $Q^T t$ and R and both the rows and columns of the modified Q^T are circularly shifted by one position, moving row dd to the top (and column dd of Q^T to the left edge). The following is the result:

$$\begin{bmatrix} \pm 1 & 0 \\ 0 & \tilde{Q}^T \end{bmatrix} \begin{bmatrix} t_{dd} \\ \tilde{t} \end{bmatrix} = \begin{bmatrix} \pm D_{dd,*} \\ \tilde{R} \end{bmatrix} w - \begin{bmatrix} \pm 1 & 0 \\ 0 & \tilde{Q}^T \end{bmatrix} \begin{bmatrix} \varepsilon_{dd} \\ \tilde{\varepsilon} \end{bmatrix}$$

The top row has thus been decoupled from the rest of the factorization and may either be deleted or updated with new data.

The update process more or less reverses these steps, adding a new top row to R and t and a row and column to Q^T. Then R is made upper triangular once more by a sequence of Givens rotations that zero its sub-diagonal elements (formerly the diagonal elements of $\tilde{R}$) one at a time. These rotations are applied not just to R but also to $Q^T t$ and of course to Q^T itself.

C.2. Rank Preservation

Deciding in advance whether downdating a row of R will reduce its rank is equivalent to predicting whether one of the Givens rotations, when applied to R, will zero or nearly zero a diagonal entry of R. This is particularly easy to discover because dd, the row to be downdated, is initially all zeros in R, *i.e.* in the lower part of the matrix. In this situation a diagonal entry of R, $R_{i,i}$ say, will be compromised if and only if the cosine of the Givens rotation that involves rows dd and i is nearly zero. The result will be an interchange of the zero in $R_{dd,i}$ with the nonzero diagonal element $R_{i,i}$. $R_{dd,i}$ is zero before the rotation because R was originally upper triangular and prior rotations only involved row subscripts greater than i.

PACORA keeps track of the sequence of values in $Q^T_{dd,dd}$ without actually changing Q^T so that if the downdate at location dd is eventually aborted there is nothing to undo. It is also possible to remember the sines and cosines of the sequence of rotations so they don't have to be recomputed if success ensues. A rank-preserving row to downdate will always be available as long as R is sufficiently "tall". Having at least twice as many rows as columns is enough since the number of available rows to downdate matches or exceeds the maximum possible rank of R.

C.3. Column Update and Downdate

The active-set NNLS method is based on the idea that since the only constraints are variable positivity then for all components either the variable or its gradient will be zero at a solution point; see [9], page 142. The active set, denoted by $\mathbf{Z}$, comprises the column subscripts j for which the variable w_j is zero and the gradient v_j is positive. If a column j not currently in $\mathbf{Z}$ happens to acquire a negative w_j after a backsolve, w_j is zeroed, j is moved into $\mathbf{Z}$ and column j is downdated in R, thereby making the gradient positive. Conversely, if a column already in $\mathbf{Z}$ happens to acquire a negative gradient v_j it is removed from $\mathbf{Z}$ and updated in R, allowing it to further reduce the value of the objective function.

The set $\mathbf{Z}$ and its complement $\mathbf{P}$ are implemented as an index *idx* containing a single permutation vector of the column subscripts comprising the sorted members of $\mathbf{P}$ followed by the sorted members of $\mathbf{Z}$, and an offset defining the beginning of $\mathbf{Z}$ in the vector. For example, if columns 1, 3, and 4 are in $\mathbf{Z}$ and columns 0, 2, and 5 are in $\mathbf{P}$ then the resulting vector is [0 2 5 1 3 4] and the offset is 3. Since the offset is just the size of the set $\mathbf{P}$ it is naturally called p.

Columns are left in place in R. The columns of R belonging to $\mathbf{P}$ are denoted by R^p and those in $\mathbf{Z}$ by R^z. The updating or downdating of a column only involves modifying the index *idx* to redefine $\mathbf{P}$ and $\mathbf{Z}$ and then applying Givens rotations to (all of) the rows of R to restore R^p to upper triangular form.

After initial acquisition of data and QR factorization, each step of PACORA's NNLS algorithm combines incremental row and column downdates and updates as follows:

Algorithm C.1: INCREMENTALNNLS(t_0, d_0)

```
local $R, Q^T, Q^T t, w, v, idx, d, u, done$
$R, Q^T, Q^T t \leftarrow$ DNDTROW($R, Q^T, Q^T t, idx$)
$R, Q^T, Q^T t \leftarrow$ UPDTROW($t_0, d_0, R, Q^T, Q^T t, idx$)
$w \leftarrow$ BACKSOLVE($R, Q^T t, idx$)
$v \leftarrow$ GRADIENT($R, Q^T t, idx$)
repeat
  $done \leftarrow$ true
  $d \leftarrow \arg\min(w)$
  if $w_d < 0$
    then
      $done \leftarrow$ false
      $R, Q^T, Q^T t, idx \leftarrow$ DNDTCOL($R, Q^T, Q^T t, idx, d$)
      $w \leftarrow$ BACKSOLVE($R, Q^T t, idx$)
      $v \leftarrow$ GRADIENT($R, Q^T t, idx$)
  $u \leftarrow \arg\min(v)$
  if $v_u < 0$
    then
      $done \leftarrow$ false
      $R, Q^T, Q^T t, idx \leftarrow$ UPDTCOL($R, Q^T, Q^T t, idx, u$)
      $w \leftarrow$ BACKSOLVE($R, Q^T t, idx$)
      $v \leftarrow$ GRADIENT($R, Q^T t, idx$)
until $done$
return $(w, v)$
```

When a column indexed by d in R^p is downdated because $w_d < 0$, that column is moved from **P** to **Z** in *idx*. To restore R^p to upper triangular form, Givens rotations are applied to R at rows $R_{d,*}$ and $R_{k,*}$ where $d < k < p$. The row subscripts k are used in decreasing order from $p-1$ down to $d+1$, and each rotation zeros the subdiagonal element in R^p of the column indexed by k. As usual, these rotations are also applied to Q^T and $Q^T t$. The result in R^z is a “spike” of nonzeros in the column that was moved; it can eventually extend to the bottom of R as *row* updates occur.

Column movements from **Z** to **P** are based on the gradient v of the objective function, namely

$$\begin{aligned} v &= 1/2\nabla\|Dw - t\|_2^2 \\ &= D^T(Dw - t) \\ &= R^T Q^T(QRw - t) \\ &= R^T(Rw - Q^T t) \\ &= R^T(-Q^T\varepsilon). \end{aligned}$$

If for some column in **Z** the inner product of the corresponding spiked row in R^T and $-Q^T\varepsilon$ is negative, the column subscript must be moved to P. Updating R^p reverses the downdating steps by zeroing the spike via a sequence of Givens rotations on R between adjacent pairs of rows, starting at the bottom and ending at $m, m+1$ where m is the position of the new column in *idx*. These rotations conveniently extend the columns to the right of m in R^p by one, thus restoring R^p to upper triangular form. Once again, the rotations are also applied to Q^T and $Q^T t$.

A new gradient computation and new back-solve for w are clearly necessary after either downdates or updates to columns of R.