PACORA: Optimizing Resource Allocations for Dynamic Interactive Workloads

Abstract

1. Introduction

“Resource allocation” as the term is used here means the apportionment by an operating system of processor cores, memory pages, and various categories of bandwidth to computations that compete for those resources. The objective is to allocate resources so as to minimize a metric of responsiveness subject to the finite resources available. This definition naturally establishes resource allocation as a type of constrained optimization problem. We will focus primarily on client systems rather than servers or data centers, but nearly all of the principles discussed here are applicable in these other domains as well.

Historically,resource allocation has been rather unsystematic. 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. Memory allocation has usually employed independent machinery, and other resources such as I/O or network bandwidth have been deemed so abundant as to require no explicit management at all.

The assumptions underlying strategies of this sort no longer hold,especially for emerging client systems. First, applications increasingly differ in their ability to exploit multiple processor cores and other resources, and these differences are independent of their relative responsiveness requirements. Second, the value of application responsiveness is highly nonlinear for an increasing variety of “Quality-Of-Service” (QOS) applications like streaming media or gaming; for these real-time applications, responsiveness is approximately two-valued depending on whether performance is adequate or not. Third, power and energy are key concerns, in mobile computing for example, and available battery energy itself becomes a component of responsiveness.

2. PACORA Architecture

Resource Allocation As Optimization

The resource allocation approach taken here attempts to address the challenges described above as follows:

1. The *process* is the entity to which the operating system allocates resources. Micro-management of the resources within a process is generally application dependent and should be under the control of components of the runtime environment such as a user-mode work scheduler for processor cores or a memory garbage collector for memory pages.
2. The objective function to be minimized is the total *penalty*, which is the sum of the penalties of the runnable processes.
3. The penalty of a process is a function rather than a single value and the argument of each penalty function, the *runtime*, is an appropriate measure of the performance of the process. For example,the runtime of a process might be one of these:

   The time from a mouse click to its result;

   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.
4. The runtime of a process is measured or predicted from its history of resource usage.

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

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

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

Convex Optimization

If the penalty functions, runtime functions, and resource constraints were arbitrarily, little could be done to optimize the total penalty beyond searching at random for the best allocation. However, if resource management can be framed as a *convex optimization problem* [**?**], two benefits will accrue: an optimal solution will exist without multiple local extrema, and fast, incremental solutions will become feasible.

A constrained optimization problem will be 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 $F(\theta x+(1-\theta)y) \le \theta F(x)+(1-\theta)F(y)$ 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 $F(y) \ge F(x)+\nabla F^T(y-x)$ where ∇F is the gradient of F and its domain is an open convex set. Put another way, F is convex if its first-order Taylor approximations are always global underestimates of its true value.

A convex optimization problem is one that can be expressed in this form:

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

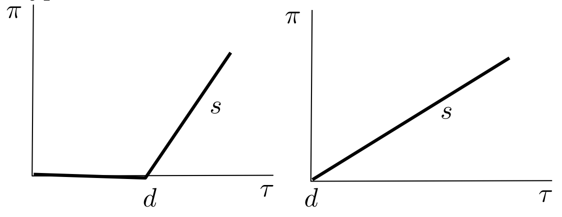
A few more facts about convex functions will be useful in what follows. First, a *concave* function is one whose negative is convex. Maximization of a concave function is equivalent to minimization of its convex negative. An affine function, one whose graph is a straight line in two dimensions or a hyperplane 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).

As a consequence, the resource management problem posed above 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 runtime functions τ_p are convex. Note that the resource constraints are all affine and can be rewritten as $\sum_{p \epsilon P} a_{p,r} - A_r \leq 0$ and $-a_{p,r} \leq 0$.

3. Application Functions

Penalty Functions

Penalty functions are generically defined as members of a family of such functions so that user preferences for a process p (elided in the discussion below) can be implemented by assigning values to a few well-understood parameters. As a process grows or diminishes in importance, its penalty function can be modified accordingly. 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. The penalty functions used in PACORA are non-decreasing piecewise linear functions of the form $\pi(\tau) = \max(0, s(\tau - d)$. Two representative graphs of this type are shown below.



The two parameters d and s define the penalty function. To guarantee it is convex and non-decreasing, s must be non-negative. The runtime τ is of course non-negative, and it may be sensible (if not strictly necessary) to convene that d is also. A service-time constrained process 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 $\tau > d$). "Softer" requirements will doubtless be the rule. For processes without service time constraints one can set $d = 0$. This defines linear behavior with s as the rate of penalty increase with runtime.

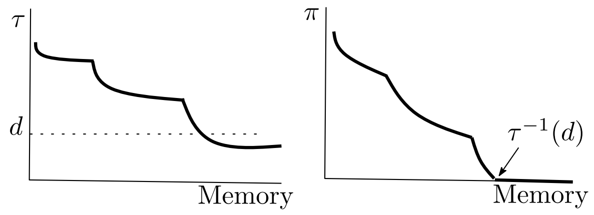
The gradient of process penalty with respect to its resource allocations is useful in controlling the optimization algorithm. By the chain rule, $\partial\pi/\partial a_r = \partial\pi/\partial\tau \cdot \partial\tau/\partial a_r$. The first term is well-defined but discontinuous at $\tau = d$ with $\partial\pi/\partial\tau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

Runtime Functions

Unlike penalty functions, which describe user preference, runtime functions measure performance of processes as functions of their resource assignments. Runtime will commonly vary with time as a process changes phase and can make better or worse use of certain resources To guarantee the objective function is convex, τ must be also, and this is at first glance a plausible requirement akin to the proverbial "Law of Diminishing Returns". An equivalent statement is that incrementally increasing the allocated quantity of a resource results in a runtime that is never better (smaller) than a first-order Taylor extrapolation from the current allocation.

Besides the value of the runtime function, its gradient or an approximation to it is useful to estimate the relative runtime improvement from each type of resource. A user-level runtime scheduler that manages allocation internal to the process may be a good source of data. Additionally, the resource manager can allocate a modest amount of a resource and measure the change in runtime. Instead of these, PACORA maintains a parameterized analytic runtime model with the partial derivatives evaluated from the model *a priori*.

There are examples of runtime versus resource behavior that violate convexity. One such example sometimes occurs in memory allocation, where "plateaus" can sometimes be seen:



Typically,these plateaus are caused by algorithm adaptations within the process to accommodate variable resource availability. The runtime is really the minimum of several convex runtime functions depending on allocation and the point-wise minimum that the process implements fails to preserve convexity. The effect of the plateaus will be a non-convex penalty as shown in the right-hand figure, and multiple extrema in the optimization problem may result.

There are several ways to avoid this problem. One is based on the observation that such runtime 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 process p from any assignment of resource r exceeding μ. Similarly, $\mu - a_{p,r} \leq 0$ exludes the opposite possibility. A binary (or even linear)search of such sub-domains could be used to find the optimal value.

PACORA adopts a simpler idea, modeling runtimes by functions that are convex by construction and do not distort runtime behavior too much. This approach is developed more fully below.

Power and Battery Energy

It is useful to designate a "process" to receive allocations of all resources that are not used elsewhere and are therefore to be powered off if possible. Process 0 plays this role in PACORA. The measure of runtime for process 0, τ_0, is artificially defined to be the total system power consumption. This function is affine and monotone nonincreasing in its arguments $a_{0,r}$.

The penalty function π_0 can now be used to keep total system power below the parameter d_0 to the extent the penalties of other processes cannot overcome its penalty slope s_0. Both s_0 and d_0 can be adjusted to reflect the current battery charge. As the battery depletes, π_0 can force other processes to slow or cease execution.

This introduction of *slack* resource allocations into Process 0 turns the resource bounds into equalities:

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

Runtime Modeling

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

- The size of the multidimensional runtime 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.

An alternative approach is to model the runtime functions using parameterized expressions that are convex by construction. For example, the runtime might be modeled as a weighted sum of component terms, one per bandwidth 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 bandwidth resource [**?**]. For example, one term might model the number of instructions executed divided by total processor MIPS, another might model storage accesses divided by storage bandwidth allocation and so forth. Such models will automatically be 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 runtime 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 $\Re^n$.

Asynchrony and latency tolerance may make runtime 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.

This scheme also accommodates non-bandwidth resources such as memory, the general idea being to roughly approximate "diminishing returns" in the runtime 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.

Sometimes a runtime component might be better modeled by a term involving a combination of resources. For example, runtime 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 begs the question of how memory affects the runtime. The effect of memory on runtime 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 the need for bandwidth has been studied by H. T. Kung [**?**], 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 process. If a process shows diminished bandwidth amplification as its memory allocation increases, we can even let

$$\begin{aligned}\alpha(m) &= \min(c_1\alpha_1(m), c_2\alpha_2(m))\\ & \quad c_1, c_2 \geq 0\end{aligned}$$

Each bandwidth amplification factor might be described by one of the functions above and included in the denominator of the appropriate component in the runtime function model. For example, the storage runtime component for the model of an out-of-core sort process 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 runtime 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.

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.

Runtime Model Convexity

We now show that the runtime 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,

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

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 runtime function for a process might be modeled by the convex function

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

where the w_j are the parameters of the model (the "quantities of work") to be learned, the b_j are the allocations of the bandwidth resources, the α_j are the bandwidth amplification functions (also to be learned), and the m_j are the allocations of the memory or cache resources that are responsible for the amplifications. This formulation allows the process runtime 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.

4. Dynamic Optimization

On-line Runtime Modeling

For the moment, assume the model norm $p = 1$ and suppose several runtime measurements have already been made using a variety of resource allocations to begin optimizing the application runtime. After enough 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 runtimes measured for the process and D is a matrix whose ith row $D_{i,*}$ contains the reciprocals of the amplified bandwidth allocations that generated the corresponding runtime measurement t_i. Estimating w given this formulation is straightforward. A least-squares solution accomplished via *QR factorization* [**?**] of D will determine the w that minimizes the residual error $\|t - Dw\|_2 = \|\varepsilon\|_2$. 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 the *residual* $Q^T \varepsilon$. Since Rw exactly equals the upper part of $Q^T t$, the upper part of $Q^T \varepsilon$ is zero. The error for each t_i can be found by premultiplying $Q^T \varepsilon$ by Q.

Suppose a different model norm p is desired. If $p = 2$, we might 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 $\|t^2 - D^2 w^2\|_2$.

Incremental Least Squares

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 process if its characteristics have changed, presumably as reflected in $Q^T \varepsilon$. . 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 [**?**] 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 maintained at 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.

Downdating makes an instructive example. A downdate 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 is still orthogonal, the non-diagonal elements of row dd must also be zero 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 had on Q^T.

There are a few ways to select the row pairs and the order of rotations that will preserve the upper triangular structure of R while zeroing almost all of a column of Q^T. Since dd is below the diagonal of R it initially contains only zeros. It therefore suffices 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 in row dd. At this point, except for a possible difference in overall sign, $R_{dd,*} = D_{dd,*}$.

Now the rows from the top down through dd of the modified matrices $Q^T t$ and R and both the rows and columns of the new Q^T are circularly shifted one position down, moving row dd to the top (and column dd of Q^T to the left edge). The following picture 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 $\tilde{R}$). These rotations are applied not just to R $(= Q^T D)$ but also to $Q^T t$ and of course to Q^T itself.

Non-negativity

The solution w to $t \approx QRw$ must have no negative components to guarantee convexity of the runtime model. If a resource allocation is associated with multiple w_j or if measured runtime increases with the allocation, then negative w_j may occur.

The need 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 [**?**], 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 that downdates and updates the *columns* of R incrementally, somewhat in the style of Algorithm 3 in [**?**]. The scheme is too complex to be adequately described here.

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, perhaps caused by small process runtime fluctuations. 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 w.

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.

Outliers and Phase Changes

Some runtime 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 runtimes τ and the measured ones t by a factor that increases exponentially with the age $g(i)$ of the ith 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.*

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

Penalty Optimization

Convex optimization is simplest when it is unconstrained. Extending the runtime model functions to all of $\Re^n$ moves the requirement that allocations must be positive into the objective function, and introducing Process 0 for slack resources turns the affine inequalities into equalities:

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

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

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

where d_r is the (constant) power dissipation associated with 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 [**?**]. 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 runtime 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 should be set to zero. Since the only constraint is variable positivity then either the variable or its gradient component will be zero at a solution point; see [**?**], page 142.

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.

5. Evaluation

6. Discussion

7. Related Work

8. Conclusion