Online Prediction of the Running Time of Tasks *

Peter A. Dinda
Department of Computer Science
Northwestern University
pdinda@cs.northwestern.edu

Abstract. We describe and evaluate the Running Time Advisor (RTA), a system that can predict the running time of a compute-bound task on a typical shared, unreserved commodity host. The prediction is computed from linear time series predictions of host load and takes the form of a confidence interval that neatly expresses the error associated with the measurement and prediction processes—error that must be captured to make statistically valid decisions based on the predictions. Adaptive applications make such decisions in pursuit of consistent high performance, choosing, for example, the host where a task is most likely to meet its deadline. We begin by describing the system and summarizing the results of our previously published work on host load prediction. We then describe our algorithm for computing predictions of running time from host load predictions. We next evaluate the system using over 100,000 randomized testcases run on 39 different hosts, finding that is indeed capable of computing correct and useful confidence intervals. Finally, we report on our experience with using the RTA in application-oriented real-time scheduling in distributed systems.

Keywords: performance prediction, performance analysis, adaptive applications

1. Introduction

To provide consistent high performance when running on typical shared, unreserved distributed computing environments, adaptive applications must exploit the degrees of freedom such environments offer, carefully choosing how and where to run their tasks [4, 2]. To make such decisions, applications require predictions of the performance of each of the alternatives. This paper addresses one form of such application-level performance predictions.

Consider an adaptive application, such that as a distributed scientific visualization system [1, 18, 4], that needs to schedule a real-time task with known resource requirements on one of several hosts. If the application could predict the running time of the task on each of the available hosts, it could trivially choose an appropriate host to run the task. Even if no host existed on which the task could meet its original deadline, such predictions of running time would permit the application to modify the resource requirements of the task or its deadline until an appropriate host could be found.

This paper describes a system, the Running Time Advisor (or RTA), that can supply these predictions for the case of compute-bound tasks. To char-

* Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, and EIA-0130869.

© 2001 *Kluwer Academic Publishers. Printed in the Netherlands.*

```
int PredictRunningTime(RunningTimePredictionRequest  &req,
                       RunningTimePredictionResponse &resp);

struct RunningTimePredictionRequest {
  Host   host;
  double conf;
  double tnom;
};

struct RunningTimePredictionResponse {
  Host   host;
  double tnom;
  double conf;
  double texp;
  double tlb;
  double tub;
};
```

Figure 1. Running Time Advisor (RTA) API.

acterize the variability inherent to distributed systems and to the process of prediction, the RTA predicts a task's running time as a confidence interval computed to the application's requested confidence level. Confidence intervals provide a simple abstraction to the application, but still provide sufficient information to enable valid statistical reasoning in the scheduling process.

The RTA's response is computed from host load predictions, a topic we have thoroughly reported on in previous papers [5, 8, 7, 6]. We have implemented an extremely low overhead online host load prediction system based on our results and our general purpose RPS Toolkit. In this paper, we describe the algorithm the RTA uses to compute a confidence interval for the running time of a compute-bound task from such host load predictions. We then evaluate the RTA using a randomized evaluation approach.

The evaluation, in which we use a 95% confidence level, takes place in a real environment where the background load on a host is supplied by host load trace playback [9], a new technique that lets us reconstruct a realistic repeatable workload using a host load trace collected on a real machine. We use 39 traces that are described in detail in a previous paper [5] and are representative of production and research clusters, application servers, and desktops.

The main conclusion is that the RTA and its algorithm can indeed predict the running time of tasks in a useful and effective way. Furthermore, our experience with predictive real-time scheduling based on the RTA suggests that these predictions are quite useful. The software and traces described in this paper are all publicly available. [1]

2. RTA API, system, and metrics

Figure 1 presents the interface of the Running Time Advisor. A query takes the form of a `host`, a confidence level `conf` ($conf$, e.g., 95%), and a nomi-

[1] http://www.cs.nwu.edu/∼pdinda/{RPS.html,LoadTraces}.

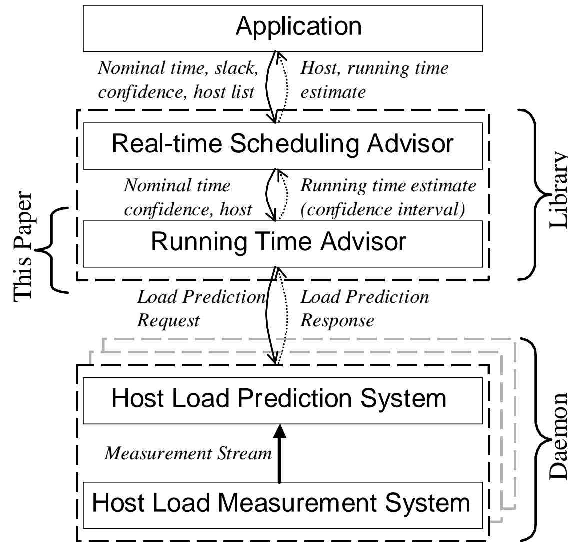


Figure 2. RTA system and context.

nal time tnom (t_{nom}), which is the running time of the task on an otherwise vacant machine. A response consists of a copy of the request's fields, the expected running time of the task texp (t_{exp}), and the upper and lower bounds of the $conf$ confidence interval for the running time, [tlb, tub] ($[t_{lb}, t_{ub}]$.) t_{exp} is a point estimate which represents the most likely running time. The actual running time, t_{act}, will likely be different from t_{exp} but be near it. The confidence interval represents a range of values around t_{exp} such that t_{act} will be in the range a fraction $conf$ of the time. Because the lower bound of the confidence interval is artificially limited due to the fact that load cannot drop below zero, the expected time is not necessarily in the middle of the confidence interval.

Figure 2 shows the structure of the RTA and the broader context of which it is a part. The system is based on the measurement, characterization, and prediction of host load, which we describe in the next section. This paper is primarily concerned with the Running Time Advisor part of the system, which predicts the running time of tasks based on the host load predictions. The Real-time Scheduling Advisor (RTSA) component of the system suggests, for interactive applications such as scientific visualizations [1, 18], the host where a just-submitted soft real-time task is most appropriately run. We will return briefly to the RTSA in Section 8, and it is also described in detail elsewhere [6, Chapter 6].

Evaluating the quality of the confidence interval, $[t_{lb}, t_{ub}]$, is a somewhat complex endeavor. Suppose we ran a wide variety of testcases with a specified confidence, say 95%. If we used the ideal algorithm for computing confidence intervals and the best possible predictor, the lengths of the tasks' confidence

intervals would be the minimum possible such that 95% of the tasks would have running times in their predicted intervals. An imperfect algorithm, such as ours, will compute confidence intervals that were larger or smaller than ideal where fewer or more than 95% of the tasks complete in their intervals. The important point is that to evaluate a confidence interval algorithm, we must measure the lengths of the confidence intervals it produces and the number of tasks which complete within these confidence intervals. To evaluate confidence intervals, we will use following two metrics:

- **Coverage**: the fraction of tasks which complete with their predicted confidence intervals
- **Span** : the average width of the confidence interval width in seconds

The ideal system will have the minimum possible span such that the coverage is 95%. We will also briefly touch on how well the t_{exp} point estimate predicts t_{act}. Generally, an adaptive application will use the confidence intervals.

3. Measurement and prediction of host load

The Running Time Advisor's predictions of running time are computed from the nominal time of the task, t_{nom}, and predictions of host load. The host load measure that we use is the Digital Unix 5 second load average. Conceptually, the kernel samples the length of the run queue at some frequency, computes an exponential average over the samples with a time constant of five seconds, and then presents this measure to applications. Our load sensor samples this measure at a rate of 1 Hz, which is twice the empirically determined kernel frequency. It is this discrete-time signal that we predict.

We began our investigation of host load by creating a public archive of a large family of long host load traces taken on a wide variety of machines. There are 39 traces, each roughly one week long, and sampled at the appropriate 1 Hz frequency. The traces include production cluster machines at the Pittsburgh Supercomputing Center (PSC), research cluster machines in the Computing, Media, and Communication Lab (CMCL) at Carnegie Mellon University (CMU), big memory application servers in the CMCL, and desktop workstations at CMU. We studied the statistical properties of these traces and presented a detailed description of the traces and the results in an earlier paper [5].

We evaluated different predictive models on the traces in a large scale randomized study. Surprisingly, despite the complex statistical properties that we identified in our earlier study, which included self-similarity and epochal behavior, simple linear time series models proved to be most appropriate for host load prediction. In fact, the best all around model for host load predictions of 1-30 seconds into the future, in terms of predictive power and low

overhead, was the AR(16) model [8]. In this paper, we shall present results that use the AR(16) model, as well as the simple LAST model (last measurement is prediction for all future measurements), and the MEAN model (prediction is the long-term arithmetic average of the load signal.)

It is important to note that linear time series models do not merely provide point predictions for future values of the load signal. Such models also provide a characterization of their prediction errors, and how prediction errors for different time horizons are related. This characterization takes the form of a covariance matrix, and is critical to computing a confidence interval for the running time, as we shall see in the next section. To compute the covariance matrix for the LAST model, we treat it as an AR(1) with a pole at unity. To compute the covariance matrix for MEAN, we compute the autocovariance function of the signal.

We developed the RPS toolkit to simplify the construction of online prediction systems. RPS provides components which can be linked together at run-time using different communication methods [7]. Such a composition forms a prediction system. We implemented the online host load measurement and prediction systems of Figure 2 using RPS. These systems have have extremely low overhead, consuming much less than one percent of the CPU time of a typical desktop host to predict the load on that host.

4. RTA algorithm

We relate the running time of a task, t_{exec}, to the average load it experiences while it runs using the following continuous-time model:

$$\frac{t_{exec}}{1 + \frac{1}{t_{exec}} \int_0^{t_{exec}} z(t)dt} = t_{nom}$$

Here $z(t)$ is the load signal, shifted such that $z(0)$ is the value of the signal at the current time, t_{now}. We introduce this shift to simplify the presentation of our algorithm, and to conform to the Box-Jenkins notation for time series analysis. This simplification does not affect the results. t_{nom} is the nominal running time of the task, which quantifies the CPU demand (or "size") of the task as its running time on an unloaded machine.

This continuous-time equation is basically a fluid model of a priority-less host scheduler. We will use this simple model to describe our estimation procedure. However, real schedulers incorporate priorities that can change over time. We assume that the majority of the workload runs at similar priorities. In particular, we assume that there are no processes whose priorities have been drastically increased or decreased, such as with the Unix "nice" facility. Ultimately, we will relax this assumption slightly and model the temporary priority boosts that most Unix implementations give processes immediately

after they become unblocked. Given this extension, the procedure we outline in this section works quite well.

Continuous-time: The above equation is somewhat unwieldy to discretize and use, so, before we continue, let's define the *available time function*

$$at(t) = \frac{t}{1 + al(t)} \; , \; t > 0 \tag{1}$$

which depends on the *average load function*

$$al(t) = \frac{1}{t} \int_0^t z(\tau) d\tau \; , \; t > 0 \tag{2}$$

$at(t)$ represents the available CPU time over the next t seconds, which is inversely related to the average load during that interval, $al(t)$. As the average load increases, the available time decreases. t_{exec} is then the minimum t for which $at(t) = t_{nom}$. Using this available time function function makes it easier to explain how our algorithm estimates the running time of a task, and, of course, the available time function is offered directly through the API described in Section 2.

Discrete-time: In our system, z is not a continuous-time signal, but rather it is a discrete-time approximation of the continuous-time signal with a sampling interval of Δ seconds, $z_{t+i}, i = 1, 2, \ldots, \infty$, where z_{t+1} represents $z(t)$ for $0 < t \leq \Delta$, and so on. We approximate $z(t)$ as $z(t) = z_{\lceil t/\Delta \rceil}$. This lets us write a discrete-time approximation of $at(t)$ and $al(t)$:

$$at_i = \begin{cases} 0 & i = 0 \\ \frac{i\Delta}{1+al_i} & i > 0 \end{cases} \tag{3}$$

$$al_i = \frac{1}{i} \sum_{j=1}^{i} z_{t+j} \; , \; i > 0 \tag{4}$$

at_i is the the time available during the next $i\Delta$ seconds and al_i is the average load that will be encountered over the next $i\Delta$ seconds. We then estimate the available time $at(t)$ by linear interpolation:

$$at(t) = at_{\lfloor t/\Delta \rfloor} + \frac{t - \lfloor t/\Delta \rfloor}{\Delta} (at_{\lceil t/\Delta \rceil} - at_{\lfloor t/\Delta \rfloor}) \tag{5}$$

Using host load predictions: Given these definitions, we substitute the predicted load signal $\widehat{z}_{t+i}$ for z_{t+i} resulting in the predicted average load $\widehat{al}_i$,

and continue substituting back to give the predicted available time $\widehat{at}_i$ and its corresponding continuous-time approximation:

$$\widehat{at}_i = \begin{cases} 0 & i = 0 \\ \frac{i\Delta}{1+\widehat{al}_i} & i > 0 \end{cases} \tag{6}$$

$$\widehat{al}_i = \frac{1}{i}\sum_{j=1}^{i} \widehat{z}_{t+j}, i > 0 \tag{7}$$

$$\widehat{at}(t) = \widehat{at}_{\lfloor t/\Delta \rfloor} + \frac{t - \lfloor t/\Delta \rfloor}{\Delta}(\widehat{at}_{\lceil t/\Delta \rceil} - \widehat{at}_{\lfloor t/\Delta \rfloor}) \tag{8}$$

Then, the expected running time of the task, t_{exp}, is simply the smallest t for which $\widehat{at}(t) = t_{nom}$.

Confidence intervals: Because host load predictions are not perfect, we also report the running time or available time as a confidence interval, computed to a user-specified confidence level. The better the predictions are, the narrower the confidence interval is.

The predicted load signal is $\widehat{z}_{t+i} = z_{t+i} + a_{t+i}$, where z_{t+i} is the real value of the signal and a_{t+i} is the i-step-ahead prediction error term which we summarize with a variance $\sigma^2_{a,i}$. Our uncertainty in estimating the available time at_i is due to our uncertainty in estimating the average load al_i, which is due in turn to these error terms and their variance. To represent this uncertainty in the form of a confidence interval, we must push the underlying error variances through the above equations to arrive at a variance for the average load al_i.

Notice that the average load (Equation 7) sums the estimates $\widehat{z}_{t+i}$. Rewriting the equation, we can see that

$$\widehat{al}_i = al_i + \frac{1}{i}\sum_{j=1}^{i} a_{t+j} \tag{9}$$

By the central limit theorem, then, $\widehat{al}_i$ will become increasingly normally distributed with increasing i. Given that the errors a_{t+i} are of zero mean, $\widehat{al}_i$ has an expected value of al_i and a variance that depends on the sum of the prediction errors a_{t+i}:

$$\widehat{al}_i \sim N\left(al_i, Var\left\{\frac{1}{i}\sum_{j=1}^{i} a_{t+j}\right\}\right) \tag{10}$$

It is important to note that for short jobs or large Δ, this normality assumption may be invalid. We will evaluate the system later and determine whether the results of the assumption are reasonable.

Suppose the user requests a confidence interval at 95% confidence. We can then compute a confidence interval for al_i (for $i > 0$):

$$[al_i^{low}, al_i^{high}] = \widehat{al}_i \mp \frac{1.96}{\sqrt{i}} \sqrt{Var\left\{\sum_{j=1}^{i} a_{t+j}\right\}} \tag{11}$$

What this means is that we predict that al_i will be in the range $[al_i^{low}, al_i^{high}]$ with 95% probability. The 1.96 is the number of standard deviations of a standard normal needed to capture 42.5% of values. al_i^{low} is set to the maximum of the computed value from above and zero. This is important because the average load cannot drop below zero, although the prediction errors can make that appear to be the case.

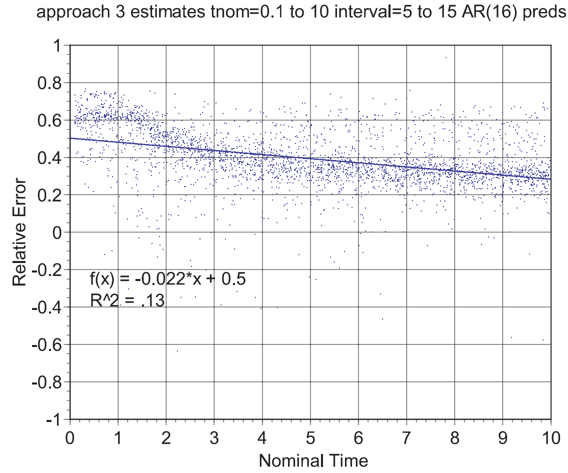
We can now back-substitute these upper and lower bounds of the confidence interval into $at(t)$, resulting in upper and lower confidence intervals for $at(t) = [at^{low}(t), at^{high}(t)]$. Then the confidence interval on the running time is $[t_{lb}, t_{ub}]$, where t_{lb} is the minimum t for which $at^{high}(t) = t_{nom}$ and t_{ub} is the minimum t for which $at^{low}(t) = t_{nom}$.

Correlated prediction errors: Given the discussion of the previous section, we must still determine the variance of a sum of consecutive prediction errors in Equation 11 to compute the confidence interval. This is one of the subtler issues in converting from load predictions to running time predictions.

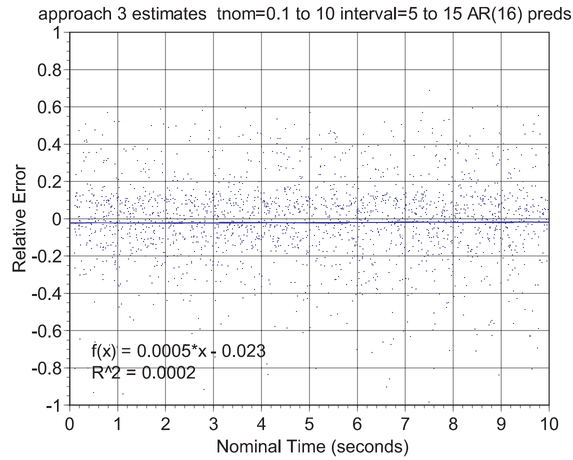
To correctly compute the variance of the sum of load predictions, we must compute the covariance of each of the prediction errors with each of the other prediction errors and then sum all i^2 terms of this covariance matrix. Entry j, k of this matrix is $CoVar\{a_{t+j}, a_{t+k}\} = \sigma_{a,j}\sigma_{a,k}$ and is the covariance of the j-step-ahead prediction with the k-step-ahead prediction. Notice that variances of the individual predictions are simply the diagonal elements of the matrix.

The prediction errors' correlation over lead time is akin to a signal's autocorrelation over time. Recall that an autocorrelation sequence is simply a normalized autocovariance sequence. The covariances are easily computed from the autocovariance sequence. In particular, $\sigma_{a,j}\sigma_{a,k} = CoVar\{a_{t+j}, a_{t+k}\} = AutoCoVar\{a_t, a_{|j-k|}\}$.

The host load prediction system uses the algorithm of Box, et al to compute the autocovariance sequence for any linear model [3, pp. 159–160]. Since the LAST predictor is simply an AR(1) model with $\phi_0 = 1$, its autocovariances can also be computed using Box, et al's method. In the case of the MEAN predictor, the autocovariances are simply the autocovariances of the signal itself.



(a) Without load discounting



(b) With load discounting

Figure 3. Relative error and load discounting.

The variance of the sum of the first i prediction errors, which we will write as $\sigma^2_{s,i}$ is then computed as

$$\sigma^2_{s,i} = Var\left\{\sum_{j=1}^{i} a_{t+j}\right\} = \sum_{j=1}^{i}\sum_{k=1}^{i}\sigma_{a,j}\sigma_{a,k} \quad (12)$$

To avoid communicating the whole covariance matrix, the sum is computed by the host load prediction system.

Load discounting: Figure 3(a) shows the result of running many tasks whose t_{nom} times vary from 100 ms to 10 s using the algorithm described thus far.

The background load was from a machine in the Pittsburgh Supercomputing Center's Alpha cluster, while the model used was an AR(16). The figure plots the relative error $((t_{exp} - t_{act})/t_{exp})$ of the predictions versus the nominal time of the task.

Notice that relative error is always positive and increases markedly as nominal time decreases. For one second tasks, the running time is over-predicted by 80%. The confidence intervals were also skewed, with far too many points falling below the lower bounds of the intervals. The problem is that the Digital Unix scheduler gives an "I/O boost" to the priority of a process when a blocking I/O operation completes. Over time, the process's priority will "age" to its baseline level. The result is that the awakened process will get more than its fair share of the CPU until its priority has degraded. The shorter the task, the more this mechanism will benefit it, and the more inaccurate our running time estimate will be, as we can see in the figure.

Our solution to this problem is load discounting. We exponentially decayed the load predictions $\widehat{z}_{t+j}$, the discounted load, $\widehat{zd}_{t+j}$, being

$$\widehat{zd}_{t+j} = (1 - e^{-j\Delta/\tau_{discount}})\widehat{z}_{t+j} \tag{13}$$

How quickly the initial load discount decays depends on the setting of $\tau_{discount}$. We determined the value of $\tau_{discount}$ empirically by again running a large number of randomized testcases as described above and varying $\tau_{discount}$ randomly in the range 0 to 10 seconds. This gives us a relative error as a function of $\tau_{discount}$, which, turns out to be linear. We fitted a line to the points and determined that it crossed zero relative error at $\tau_{discount} = 4.5$ seconds. This value is used throughout this paper and seems to be a property of the operating system that can be computed offline.

Figure 3(b) shows the result of using load discounting. As can be seen, the appropriate $\tau_{discount}$ value has eliminated the dependence of the relative error on the nominal time and has further reduced the average relative error to almost exactly zero, which we would also expect from these point estimates.

Load discounting is an effective solution to the priority boosts that most Unix schedulers give to processes that have become unblocked. It is important to note, however, that other priority problems remain. For example, a background process which has had its priority significantly reduced (e.g., a process which has has been "reniced") but which remains compute bound will result in artificially exaggerated predictions. Similarly, a process with high priority will result in predictions that are too low.

5. Experimental infrastructure

Our infrastructure hardware consists of two Alphastation 255 hosts connected with a private network. Both machines run Digital Unix 4.0D. One host is

referred to as the measurement host while the other is called the recording host. The hosts have no other load on them.

The recording host runs software that interrogates the components running on the measurement host and then submits tasks to it. The measurement host runs the following components: a host load playback tool to provide a background workload, a host load sensor, one or more host load prediction systems, and a spin server.

The playback tool uses a new technique in which the workload is generated according to a load trace [9]. With no other work on the host, this background load results in the host's load signal repeating that of the load trace. The host load sensor provides an interface for the recording host to request the latest load measurement on the host.

The reader may wonder why we do not use a synthetic workload generator. There are two reasons. First, because the behavior of host load is so complex, creating a model that captures its relevant properties is a hard problem. Furthermore, in the prediction context, it is not clear a priori what the relevant properties of the workload are. The second reason is that the predictability of a synthetic workload is inherent in the model used to generate it. Simply put, our predictor may very well "discover" our workload model and produce overstated results. The beauty of using host load trace playback is that the predictor is operating on a real workload, and yet the workload is repeatable.

The host load prediction systems (described in Section 3) provide an interface for the recording host to request the latest host load predictions using an experiment-specific prediction model.

The spin server runs tasks—it takes requests to compute (using a busy loop) for some number of CPU-seconds and then returns the wall-clock time that the task took to complete. It looks like a CORBA ORB [21]. The busy loop is carefully calibrated, and the server monitors the amount of CPU it has consumed as it computes. The relative error is much less than 1%.

6. Evaluation methodology

To evaluate the RTA given a particular traced host, we started up the experimental infrastructure described in Section 5 on the measurement and recording hosts. The host load playback tool was set to replay the selected trace (all 39 were used). The host load sensor was configured to run at 1 Hz. Three host load prediction systems were started: MEAN, LAST, and AR(16). The systems were configured to fit to 300 measurements (5 minutes) and to refit themselves when the absolute error for a one-step-ahead prediction exceeds 0.01 or the average measured one-step-ahead mean squared error exceeds the estimated one-step-ahead mean squared error by more than 5%. The minimum interval between refits was 30 seconds and the maximum interval

before the measured mean squared error was tested was 300 seconds. These parameters were found based on previous experiments [8].

The prediction and measurement software were then permitted to quiesce for at least 600 seconds. Then 3000 consecutive testcases were run on the recording host, each according to this procedure:

1. Wait for a delay interval, $t_{interval}$, randomly selected from a uniform distribution from 5 to 15 seconds.

2. Get the current time t_{now}.

3. Select the task's nominal time, t_{nom} randomly from a uniform distribution from 100 ms to 10 seconds.

4. Select a random host load prediction system from among MEAN, LAST, AR(16).

5. Use the `PredictRunningTime` API to compute the expected running time t_{exp} and the 95% confidence interval $[t_{lb}, t_{ub}]$ using the latest predictions from the selected host load prediction system.

6. Run the task on the spin server and retrieve its actual running time, t_{act}.

7. Record the timestamp t_{now}, the prediction system used, the nominal time t_{nom}, the expected running time t_{exp}, the confidence interval $[t_{lb}, t_{ub}]$, and the actual running time t_{act}.

After all 3000 testcases were run, their records were imported into a database table corresponding to the trace.

It takes approximately 13 hours to complete 3000 testcases. To evaluate the running time predictions, we mined the database of 114,000 testcases. The results of this analysis are described in the following section.

7. Evaluation results

In examining our testcases, we wanted to answer several questions. The most important of these is (1) does our system provide useful predictions of task running times in the form of valid confidence intervals? In addition we wanted to understand (2) how the choice of underlying host load predictor affects performance, and (3) how that performance depends on the nominal time of the task?

To address these questions, we looked at the testcases in several ways. First, we measured the quality of the confidence intervals independently of the nominal time of the task. For each trace, we computed the confidence

interval metrics of coverage and span. Then we compared the different predictors based on these per-trace metrics. Next, we conditioned this comparison on the nominal time of the task, dividing the range in to small, medium, and large tasks. Finally, we hand-classified each trace based on the relationship of the performance metrics and the nominal time. This resulted in five classes. We then developed a recommendation for each class. In the following discussion, when we refer to a "significant" difference, we mean that the difference is significant at a 95% confidence level.

In the following, we will begin by showing exemplars from each of the five classes of behavior we saw. The goal is to give the reader a visual idea of the performance of this system on hosts with different behaviors. After discussing each class, we will generalize our results and explain the conclusions we reached.

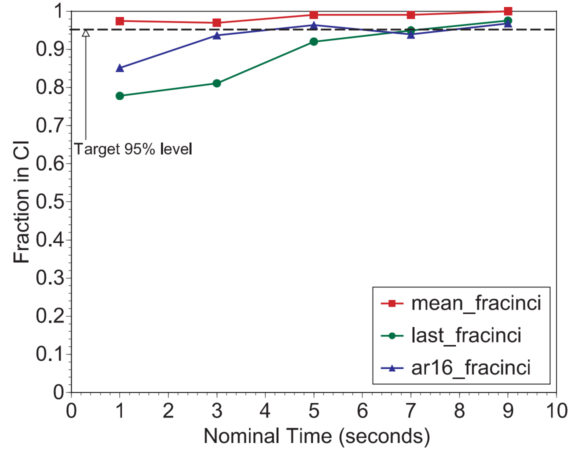
7.1. Classes of behavior

For each individual load trace, we plotted our performance metrics versus the nominal time t_{nom}. When we did this, we found that an interesting pattern emerged. By visual inspection, the results for the 39 traces could be placed into five classes. It is enlightening to examine representatives of each of these classes in detail, and doing so permits us to make recommendations for each of them.

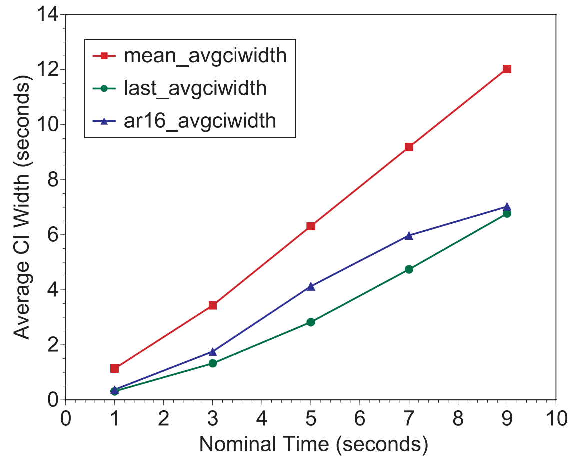
Class I: Class I, which we also call the "typical low load host" class represents the most common behavior by far that we have encountered. The class consists of 29 of the 39 hosts (76%). A representative of class I is plotted in Figure 4. Each point in the graph represents the average of about 200 testcases and represents a 2 second span of nominal time, extending from one second before the point's x-coordinate to one second after. The remainder of the figures in the paper have the same semantics.

The main characteristics of the class are the following. The coverage is only slightly dependent on the nominal time, increasing slightly for all predictors as the nominal time increases. The MEAN predictor typically has almost 100% coverage and is closely followed by the AR(16) and then the LAST predictor. The LAST and AR(16) predictors have significantly narrower spans than the MEAN predictor, with AR(16) producing slightly wider spans than LAST.

We believe that the AR(16) is the best predictor for this most common class of host. The coverage is nearly as good as MEAN and is typically near the target 95% point, while LAST tends to lag behind, especially for smaller tasks. Furthermore, the span of AR(16) is typically half that of MEAN and only slightly wider than LAST. In most hosts, then, a better predictor produces much narrower accurate confidence intervals.



(a) Coverage

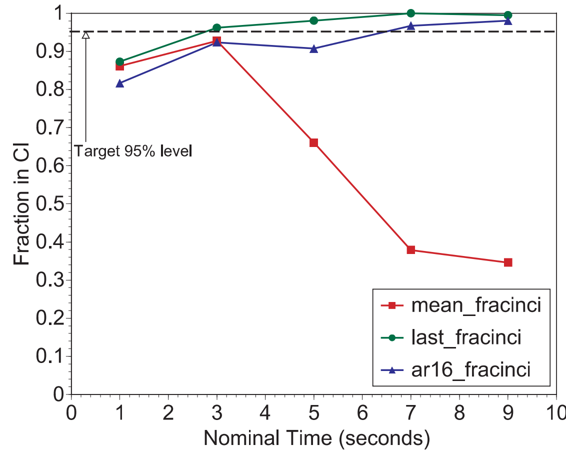


(b) Span

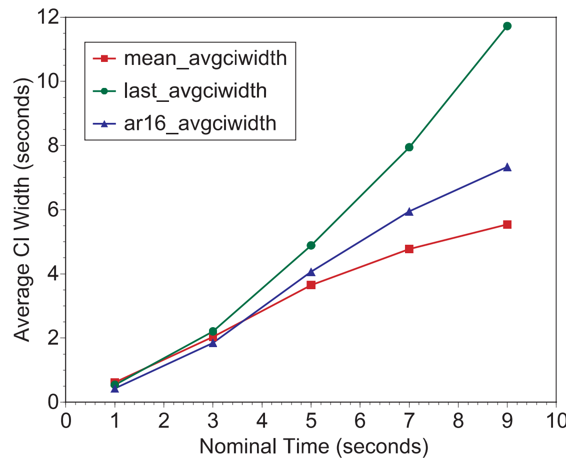
Figure 4. Coverage and Span, Class I hosts

Class II: Class II hosts, which we refer to as being in the "atypical low load host" class, represent the second most common behavior among our traces. The class consists of 4 of the 39 hosts (10%). An exemplar of Class II is plotted in Figure 5.

An important distinguishing feature of this class is that the coverage of the MEAN predictor drops precipitously with increasing nominal time because the span of its confidence interval is not sufficiently large. In contrast, LAST and AR(16) compute slightly larger confidence intervals which result in excellent coverage that increases with increasing nominal time. LAST and AR(16) have similar coverage (in this example LAST is slightly ahead, in other cases AR(16) is slightly ahead).



(a) Coverage

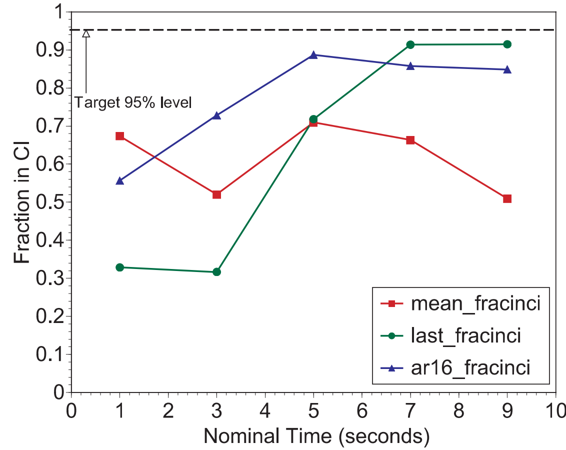


(b) Span

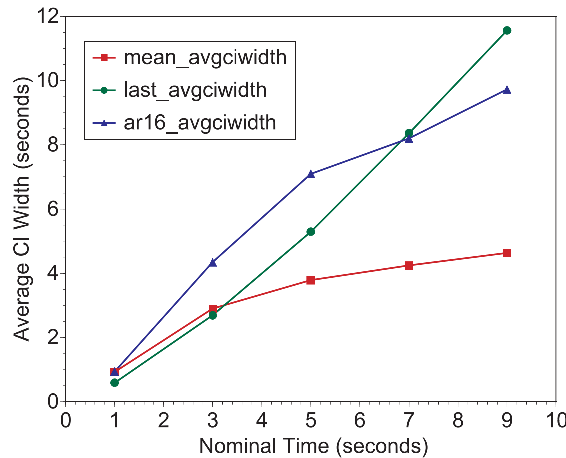
Figure 5. Coverage and Span, Class II hosts

In terms of computing confidence intervals, either AR(16) or LAST seems adequate for producing confidence intervals for this class of host. Compared to MEAN, both produce significantly larger spans that result in much better coverage. Computationally, AR(16) is almost as inexpensive as LAST.

Class III: The remainder of the five host classes all contain high load hosts. There does not seem to be a "typical" behavior on a high load host, so we will simply enumerate these classes. Class III, which we also call "high load 1", consists of a 3 of the 39 hosts (8%). Figure 6 plots the performance metrics as a function of the nominal time for an exemplar.



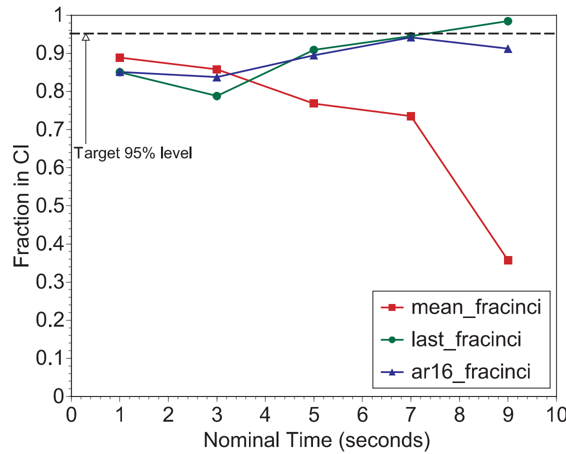
(a) Coverage



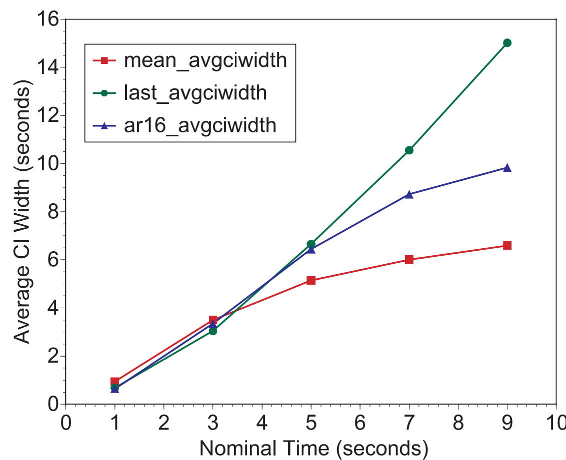
(b) Span

Figure 6. Coverage and Span, Class III hosts

Compared to the low load hosts, this high load 1 host displays much more complex behavior. The predictor with the best coverage depends strongly on the nominal time. For very short tasks, MEAN is slightly better than AR(16), which is much better than LAST, although the coverage is quite poor with all three predictors. For medium size tasks, AR(16) provides the best coverage, followed at a distance by MEAN and LAST, which become interchangeable. For large tasks, AR(16) and LAST have similar coverage, with AR(16) lagging slightly, while MEAN's coverage is far behind. In terms of the span, AR(16) and LAST both compute much wider (and thus more appropriate) confidence intervals than MEAN, which explains why their coverage is so much better. MEAN is unable to understand the dynamicity of this kind of



(a) Coverage



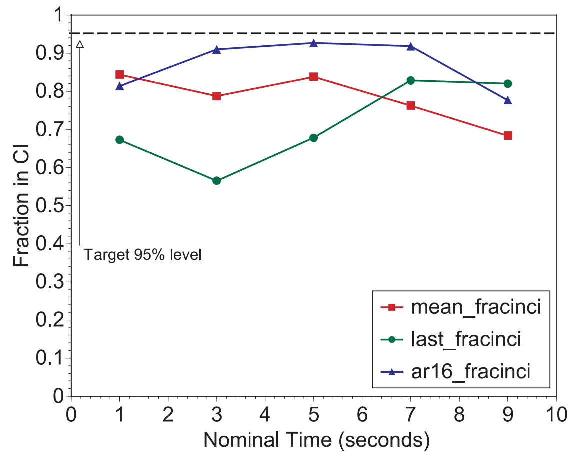
(b) Span

Figure 7. Coverage and Span, Class IV hosts

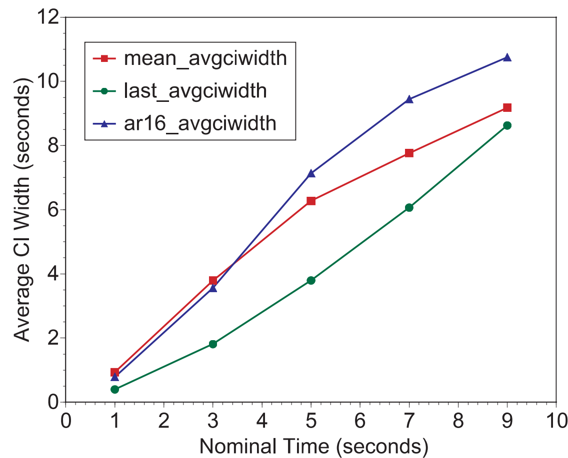
host. Predictably, for the nominal times where AR(16) is preferable to LAST, it has a larger span.

In terms of computing accurate confidence intervals, the best predictor is highly dependent on the nominal time. For very short tasks, MEAN or AR(16) is preferable, but either has rather poor coverage. For medium tasks, AR(16) produces the best performance. For large tasks, LAST is best. Clearly, there is room for improvement on this class of hosts.

Class IV: This class, which we also refer to as the “high load 2” class, contains two hosts (5%). Figure 7 plots the performance of the predictors on a representative trace.



(a) Coverage



(b) Span

Figure 8. Coverage and Span, Class V hosts

We can see that the coverage of LAST and AR(16) are virtually identical here and both increase slowly with nominal time. MEAN has similar coverage for small tasks, but then behaves increasingly poorly, with coverage decreasing rapidly with nominal time. In terms of the span, LAST grows much more quickly than MEAN with increasing nominal time, while AR(16) is almost exactly in between them. For very short nominal times the spans are all identical.

In terms of computing confidence intervals, AR(16) clearly produces the best results for this class of hosts, getting coverage identical to that of LAST with a span that is often half as wide.

Class V: Class V, which we also refer to as the "high load 3" class, consists of a single host (2.5%). Figure 8 plots the performance of the predictors on that host.

In terms of coverage, AR(16) is clearly the winner here, especially for medium sized tasks. It achieves its reasonable coverage (the goal is 95%) by computing slightly larger confidence intervals than MEAN. LAST computes confidence intervals that are far too small, resulting in abysmal coverage.

AR(16) is clearly the preferable predictor for this class of hosts in terms of computing confidence intervals.

7.2. Generalized results

The class-by-class analysis of the preceding section makes it clear what the LAST and AR(16) predictors generally provide quite different performance results than the simple MEAN predictor, and that performance can vary with nominal time. In this section, we will generalize the results we saw over all of the testcases and traces.

The RTA works: Looking at the exemplars of the five classes, one can see that the RTA system works, for the most part, as advertised. This inference is supported by a broader examination of the testcases. With almost every load trace in our study, the coverage of either the AR(16) or LAST predictor is very close to the target 95% coverage. Furthermore, these predictors, especially AR(16), do so with reasonable spans.

Of course, there are some rare cases (consider the Class IV exemplar) where coverage is significantly lower than desired. Also, it may be possible to reduce spans even further while maintaining the same coverage. These issues point to questions of the inherent predictability of distributed systems: what is the optimal predictability and the optimal characterization of prediction error that can be achieved? Without answers to these questions, it is difficult to know if performance *can* be better in these rare cases or if the span can be reduced. However, even without answering these questions, this study provides evidence for the effectiveness of the RTA.

LAST and AR(16) produce better coverage on heavily loaded hosts: On nine of our traces, LAST and AR(16) simultaneously produce better coverage and worse spans than MEAN. These nine traces correspond to the hosts that are more heavily loaded, and thus, correspondingly, exhibit greater variability in load [5]. The LAST and AR(16) predictors are better able to "understand" such hosts and compute appropriately wider confidence intervals compared to MEAN. These wider confidence intervals result in a far greater chance of a task's actual running time falling within its computed confidence interval. This is precisely the behavior that we want. Our goal is that 95% of tasks

fall within their confidence intervals. With the AR(16) predictor, of the 39 cases, only 5 cases have coverage less than 90%, and only one less that 85%, whereas with the MEAN predictor, only one of the high load traces is better than 85%. The percentage point gain from MEAN to AR(16) can be as much as 30%, and it is typically around 15%.

Two effects are at work here. First, the predictions of the LAST and AR(16) predictors depend most strongly on recent measurements. The MEAN predictor, on the other hand, always presents the long term mean of the signal. As a result, the LAST and AR(16) predictors will respond much more quickly during the period after an epoch transition (Section 3) and before a model refit happens. This means that their predictions, and thus the center point of the confidence interval will much more likely be in the right place in these situations

The second effect results from how the confidence interval length is computed. Recall that with the MEAN predictor the autocovariance of the signal is used to compute the confidence interval, while for the LAST and AR(16) predictors it is the autocovariance of their prediction errors that is used. On a high load, high variability host, an epoch transition is more likely than on a low load, low variability host to make the “old” autocovariance of the signal fail to characterize the new epoch well. The structure of the autocovariance of the prediction errors will not change at all, although the individual predictions may be less accurate.

LAST and AR(16) produce better spans on lightly loaded hosts: For those hosts which have lower load and variability, the LAST and AR(16) predictors produce significantly narrower confidence intervals than MEAN while still capturing an appropriate number of tasks within their computed confidence intervals. On average, the confidence intervals are shrunk by 2-3 seconds while the fraction of tasks within their confidence intervals shrinks by about 5%. Since for these lightly loaded hosts, the MEAN predictor results in coverages that are significantly larger than the target 95%, this is not an unreasonable tradeoff. Essentially, on average, for these low load hosts, moving from MEAN to AR(16) reduces coverage by about 5% while decreasing the span by 2-3 seconds (about 33%).

AR(16) performs better than LAST: At this point, we have shown that the RTA does indeed compute reasonable confidence intervals for task running times and that it does so more accurately when using a more sophisticated predictor than MEAN. Now we would like to know whether we should prefer the LAST predictor or the AR(16) predictor. We have already pointed out some of the differences between these two.

If we consider the aggregate performance of the different predictors on each of the traces and compare LAST and AR(16) we see that the confidence

intervals computed using AR(16) generally include more of their tasks than those computed using LAST. Using the AR(16) predictor, only five of the traces are at less than 90% and only one less than 85%. Using LAST, 9 are less than 90%, while four are less than 85%. This gain is due to AR(16) predictors producing wider confidence intervals on heavily loaded hosts. There is a corresponding performance gain on lightly loaded hosts, where AR(16) produces narrower confidence intervals than LAST because it is able to appropriately relax its coverage even more than LAST.

In essence, the use of AR(16) instead of LAST brings coverage closer to the target coverage, from above or below, through adjusting span size accordingly. In moving from LAST to AR(16), we either see a large increase in coverage, on the order of 10 percentage points or more, combined with a span increase of 2-3 seconds, or there is a slight decline of 5 percentage points or less combined with a span decrease of 1-2 seconds.

Performance is slightly dependent on the nominal time: For very small tasks, especially those on the order of the measurement period (1 second) or smaller, coverage is worse than for larger tasks. This is not too surprising given the normality assumption we make about the sum of load predictions. With these tiny tasks, the sum is over a single prediction and the point prediction error is *not* usually normally distributed. As tasks increase beyond 1-2 seconds in duration, coverage improves to near optimal. For very long tasks, we see a decline in performance on some hosts. Generally, then, as the nominal time increases, coverage improves slightly.

Obviously, the quality of our predictions should be as independent of the nominal time as possible. In fact, the level of dependence we noted is quite low. Consider Figures 4–8, our exemplars of the five behavior classes. Note that the coverage of AR(16), for example, is only slightly dependent on nominal time for the vast majority of the traces in our study. It is only in class III (8% of the traces) that we see unhappy behavior out of AR(16).

Not surprisingly, spans grow with nominal times. A longer task requires looking over a longer prediction horizon, which introduces more error. Generally, the span grows linearly with the nominal time of the task, with AR(16) having the flattest slope.

Predictions of expected running time behave similarly: For space reasons, we have not talked about the quality of the t_{exp} point predictions, concentrating instead on the quality of the confidence intervals $[t_{lb}, t_{ub}]$. The quality of the the t_{exp} predictions, as measured using the R^2 metric [14, pp. 226–228] behaves in similar ways to the span and coverage metrics for the confidence intervals. Generally, the system is able to achieve R^2 in excess of 0.9. For low load hosts, R^2 is typically even higher.

8. Experience with predictive real-time scheduling

Earlier, we introduced the Real-time Scheduling Advisor (RTSA) as a client of the Running Time Advisor. We'll briefly revisit it here as an example of this adaptation advisor can benefit from the RTA. An in-depth study of the RTSA is available elsewhere [6, Chapter 6] and it is the subject of a forthcoming paper.

The goal of the RTSA is to help the application select a host on which a real-time task, eligible to run immediately, is likely to finish on time. The application supplies to the RTSA the nominal time of the task (t_{nom}), a confidence level ($conf$), a slack factor (sf), and a list of hosts on which the task may be run. The goal is that $t_{act} \leq (1 + sf)t_{nom}$. The RTSA returns an appropriate host and the RTA prediction of the running time on that host.

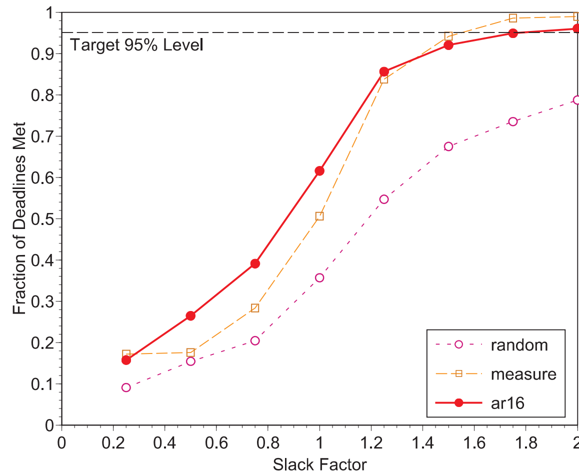
The RTSA determines the appropriate host by getting RTA predictions for each host on the list, selecting the subset of hosts for which $t_{ub} \leq (1 + sf)t_{nom}$, and then returning a randomly selected host in that subset. Recall that t_{ub} is the upper limit of the $conf$ confidence interval for the running time. If the subset is null, then the host with the lowest t_{exp} (point estimate of the running time) is returned. Because the RTSA also returns the RTA prediction for the host, it can quite naturally inform the application *whether* it is possible to meet the deadline given the constraints.

Figure 9 shows an example of the performance of the RTSA. Scheduling is being done to a collection of four hosts playing back relatively difficult-to-predict traces. The randomly generated tasks range in size from 0.1 to 10 seconds. Figure 9(a) shows the fraction of tasks that meet their deadline as a function of the slack factor. It compares the predictive RTSA (denoted "ar16" after the underlying host load prediction model) to randomly selecting hosts ("random"), and to selecting hosts with the lowest measured load ("measure"). We can see that the predictive scheme works especially well when tightly constrained. When loosely constrained, the predictive RTSA's performance converges on $conf$ while measurement-based RTSA's performance conferences on 1.0.

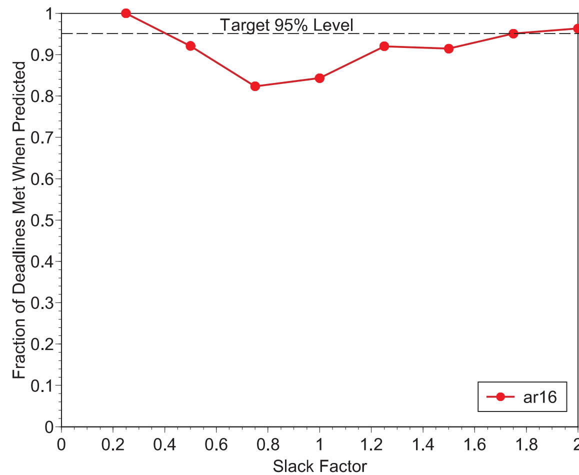
Figure 9(b) shows the fraction of deadlines met when the RTSA has predicted that they will be met through the returned RTA prediction. Clearly, by using the RTA, the predictive RTSA is able not only to provide reliable high level adaptation advice to the application, but it is also able to signal that what the application asks is likely to be impossible.

9. Related work

Work on the explicit prediction of the dynamic behavior of distributed systems, particularly to support adaptive applications, has a surprisingly short



(a) fraction of deadlines met



(b) fraction of deadlines met when predicted

Figure 9. RTSA scheduling performance versus slack factor.

history. The parallel computing community has studied application-level load balancing for some time [23, 25, 10], but this work has treated prediction only implicitly. The operating systems community has studied existing workloads [20, 11, 16, 13] to support distributed load sharing, and developed innovative system-level scheduling policies based on queueing theoretic models [13]. In contrast to these two threads, our work is done entirely at the user level and considers prediction explicitly. Our goal is to provide high level predictive services in shared, unreserved distributing computing environments that are useful to applications make different kinds of scheduling and adaptation decisions.

The notion of such application-level scheduling is due to Berman and Wolski [2]. It has been shown that application-level scheduling is feasible not only in tradition parallel load balancing, but also in distributed object systems like CORBA [29] and distributed interactive applications such as scientific visualization [1, 18]. It is also becoming increasingly clear that Grid [12] applications will have to rely on adaptation. Our work supports adaptation frameworks by providing performance predictions on which statistically valid scheduling decisions can be made. The system described in this paper has been incorporated in one such framework, BBN's QuOiN [29].

Starting in the late 90s, research began on how to build scalable systems for measuring the dynamic properties of distributed environments, leading to such well known systems as the Network Weather Service [27] (NWS) and Remos [19]. Over time, NWS incorporated time series prediction for host and network load [27, 28], while Remos did the same, using our work (the RPS toolkit [7]). With regard to this paper, we and the NWS group have independently demonstrated that host load prediction is feasible and have independently reached similar conclusions about what form of predictive model is preferable [28, 8].

This paper demonstrates that host load prediction is not only feasible, but also useful, in that such predictions permit us to cheaply and scalably predict the running time of tasks as confidence intervals. This form of higher level prediction is extremely useful in adaptation. In other work, we have shown, for example, that these predictions allow us to make scheduling decisions to meet real-time goals with high probability [6, Chapter 6]. In general, our confidence intervals can provide support for many forms of stochastic scheduling [24].

Real-time scheduling work dates back to the early 1970s [17] and includes work on collaboratively scheduled distributed real-time systems [15, 26]. Perhaps closest to the spirit of the RTSA is the work of Ramamritham, et al [22].

10. Conclusion and future work

We have described an algorithm that can estimate, on a typical shared, unreserved host running a commodity operating system, the running time of a compute-bound task given the task's CPU demand and time series predictions of the load on the host. A prediction of running time is presented to the application as a confidence interval, which enables the application to make statistically valid decisions based on the prediction. We summarized how our host load prediction system works (the full details are presented elsewhere), and then showed how we implemented the algorithm on top of it. We then evaluated the composite system using a large number of randomized

testcases. The main conclusion is that the algorithm, when paired with an appropriate predictive model for host load, does indeed compute valid confidence intervals. Experience with predictive real-time scheduling suggests that these intervals have real value in adaptive applications.

We are currently working on a similar system to predict communication times. The goal is to be able to predict, again as a confidence interval, how long it will take to transfer a given number of bytes between two hosts. In addition to predicting resource supply, we are also very interested in predicting the resource demand of applications. Finally, we are studying how to automatically learn models of resource schedulers with hopes to improve on, for example, the simple Unix scheduler model we used here.

Acknowledgements

This paper benefited from discussions with David O'Hallaron, Bruce Lowekamp, Peter Steenkiste, Jaspal Subhlok, Thomas Gross, Dean Sutherland, David Bakken, and Nancy Miller. Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, and EIA-0130869

References

1. Aeschlimann, M., P. Dinda, L. Kallivokas, J. Lopez, B. Lowekamp, and D. O'Hallaron: 1999, 'Preliminary Report on the Design of a Framework for Distributed Visualization'. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*. Las Vegas, NV, pp. 1833–1839.
2. Berman, F. and R. Wolski: 1996, 'Scheduling From the Perspective of the Application'. In: *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing HPDC96*. pp. 100–111.
3. Box, G. E. P., G. M. Jenkins, and G. Reinsel: 1994, *Time Series Analysis: Forecasting and Control*. Prentice Hall, 3rd edition.
4. Dinda, P., B. Lowekamp, L. Kallivokas, and D. O'Hallaron: 1999, 'The Case for Prediction-Based Best-Effort Real-Time Systems'. In: *Proc. of the 7th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 1999)*, Vol. 1586 of *Lecture Notes in Computer Science*. San Juan, PR: Springer-Verlag, pp. 309–318. Extended version as CMU Technical Report CMU-CS-TR-98-174.
5. Dinda, P. A.: 1999, 'The Statistical Properties of Host Load'. *Scientific Programming* **7**(3,4). A version of this paper is also available as CMU Technical Report CMU-CS-TR-98-175. A much earlier version appears in LCR '98 and as CMU-CS-TR-98-143.
6. Dinda, P. A.: 2000, 'Resource Signal Prediction and Its Application to Real-time Scheduling Advisors'. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Available as Carnegie Mellon University Computer Science Department Technical Report CMU-CS-00-131.
7. Dinda, P. A. and D. R. O'Hallaron: 1999, 'An Extensible Toolkit for Resource Prediction In Distributed Systems'. Technical Report CMU-CS-99-138, School of Computer Science, Carnegie Mellon University.

8. Dinda, P. A. and D. R. O'Hallaron: 2000a, 'Host Load Prediction Using Linear Models'. *Cluster Computing* **3**(4). An earlier version of this paper appeared in HPDC '99.
9. Dinda, P. A. and D. R. O'Hallaron: 2000b, 'Realistic CPU Workloads Through Host Load Trace Playback'. In: *Proc. of 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR2000)*, Vol. 1915 of *Lecture Notes in Computer Science*. Rochester, New York, pp. 246–259.
10. Dusseau, A. C., R. H. Arpaci, and D. E. Culler: 1996, 'Effective Distributed Scheduling of Parallel Workloads'. In: *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. pp. 25–36.
11. Eager, D. L., E. D. Lazowska, and J. Zahorjan: 1988, 'The Limited Performance Benefits of Migrating Active Processes for Load Sharing'. In: *SIGMETRICS '88*. pp. 63–72.
12. Foster, I. and C. Kesselman (eds.): 1999, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann.
13. Harchol-Balter, M. and A. B. Downey: 1996, 'Exploiting Process Lifetime Distributions for Dynamic Load Balancing'. In: *Proceedings of ACM SIGMETRICS '96*. pp. 13–24.
14. Jain, R.: 1991, *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc.
15. Kurose, J. F. and R. Chipalkatti: 1987, 'Load sharing in Soft Real-Time Distributed Computer Systems'. *IEEE Transactions on Computers* **C-36**(8), 993–1000.
16. Leland, W. E. and T. J. Ott: 1986, 'Load-balancing Heuristics and Process Behavior'. In: *Proceedings of ACM SIGMETRICS*, Vol. 14. pp. 54–69.
17. Liu, C. L. and J. W. Layland: 1973, 'Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment'. *Journal of the ACM* **20**(1), 46–61.
18. Lopez, J. and D. O'Hallaron: 2000, 'Runtime Support for Adaptive Heavyweight Services'. In: *Proc. of 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR2000)*, Vol. 1915 of *Lecture Notes in Computer Science*. Rochester, NY, pp. 221–234.
19. Lowekamp, B., N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok: 1998, 'A Resource Monitoring System for Network-Aware Applications'. In: *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC)*. pp. 189–196.
20. Mutka, M. W. and M. Livny: 1991, 'The Available Capacity of a Privately Owned Workstation Environment'. *Performance Evaluation* **12**(4), 269–284.
21. Object Management Group: 1999, 'The Common Object Request Broker: Architecture and Specification (version 2.3.1)'. Technical report, Object Management Group.
22. Ramamrithham, K., J. A. Stankovic, and W. Zhao: 1989, 'Distributed Scheduling of Tasks with Deadlines and Resource Requirements'. *IEEE Transactions on Computer Systems* **38**(8), 1110–1123.
23. Rinard, M., D. Scales, and M. Lam: 1993, 'Jade: A High-Level Machine-Independent Language for Parallel Programming'. *IEEE Computer* **26**(6), 28–38.
24. Schopf, J. M. and F. Berman: 1999, 'Stochastic Scheduling'. In: *Proceedings of Supercomputing '99*. Also available as Northwestern University Computer Science Department Technical Report CS-99-03.
25. Siegell, B. and P. Steenkiste: 1994, 'Automatic Generation of Parallel Programs with Dynamic Load Balancing'. In: *Proceedings of the Third International Symposium on High-Performance Distributed Computing*. pp. 166–175.
26. Stankovic, J., K. Ramamritham, D. Niehaus, M. Humphrey, and G. Wallace: 1999, 'The Spring System: Integrated Support for Complex Real-Time Systems'. *Real-Time Systems Journal* **16**(2/3).
27. Wolski, R.: 1997, 'Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service'. In: *Proceedings of the 6th High-Performance*

Distributed Computing Conference (HPDC97). pp. 316–325. extended version available as UCSD Technical Report TR-CS96-494.

28. Wolski, R., N. Spring, and J. Hayes: 1999, 'Predicting the CPU Availability of Time-shared Unix Systems'. In: *Proceedings of the Eighth IEEE Symposium on High Performance Distributed Computing HPDC99*. pp. 105–112. Earlier version available as UCSD Technical Report Number CS98-602.
29. Zinky, J. A., D. E. Bakken, and R. E. Schantz: 1997, 'Architectural Support for Quality of Service for CORBA Objects'. *Theory and Practice of Object Systems* **3**(1), 55–73.

Author's Vitae

Peter A. Dinda

Peter Dinda is an assistant professor in the Department of Computer Science at Northwestern University. He holds a B.S. in electrical engineering from the University of Wisconsin and a Ph.D. in computer science from Carnegie Mellon University. His research centers on the intersection of interactive applications and high performance computing, and in particular on statistical signal processing approaches to analyzing and predicting the dynamic behavior of such systems. He is a member of ACM and IEEE.