A Prediction-based Real-time Scheduling Advisor

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

Abstract

The real-time scheduling advisor (RTSA) is an entirely user-level system that an application running on a typical shared, unreserved distributed computing environment can turn to for advice on how to schedule its compute-bound soft real-time tasks. Given a list of hosts, a description of the CPU demands of the task, the deadline, and a confidence level, the RTSA will recommend one of the hosts and predict, as a confidence interval, the running time of the task on that host. The RTSA is based on a scalable and extensible shared resource prediction system based on statistical time series analysis. In this paper, we first describe how the RTSA builds on this underlying system to provide its service, and then we evaluate its performance using a randomized methodology based on real background workloads, determining the effect of different factors. We also compare it with a random approach and a measurement-based approach.

1. Introduction

Consider distributed interactive applications such as interactive scientific visualizations [1]. Such applications generate aperiodic tasks with soft real-time deadlines that follow from their users' responsiveness requirements, but they typically must run these tasks on shared computing environments that are managed by uncoordinated commodity operating systems that do not provide resource reservations, admission control, or other basic facilities upon which distributed real-time systems [16, 10, 11, 14, 17] are built. Furthermore, the tasks are in active competition with unknown background workloads introduced at will by other users. However, these applications do include adaptation mechanisms through which they can change their behavior in response to changing resource availability [3]. Several adaptation frameworks have also been proposed to generalize these application-specific mechanisms [19, 12, 2, 13].

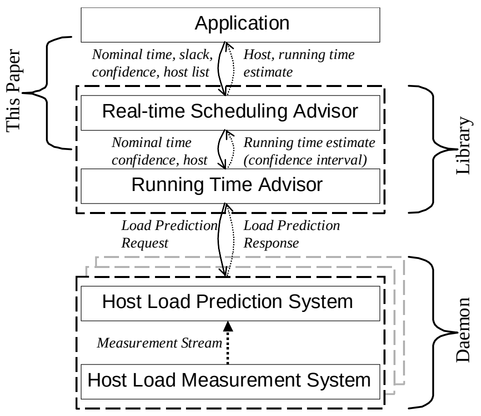
While adaptation mechanisms abound, there are few control mechanisms that can make use of them to provide real-time behavior. This paper describes the design and performance of one such control mechanism, the real-time scheduling advisor, or RTSA. The RTSA is an entirely user level tool that operates on behalf of a single application and responds to a simple query. Given a list of homogeneous hosts, the CPU demands of a compute-intensive task, a deadline, and a confidence level, the RTSA returns the host where the task is most likely to meet its deadline and a prediction of the running time of the task on that host.

The goal of the RTSA is *not* load-balancing or load-sharing, but rather to help its client application meet deadlines and to tell it when deadlines can not be met. While the system we consider here is targeted at compute-intensive tasks (so transfer costs are negligible) and homogeneous hosts, we do not believe this to be an intrinsic limitation of our approach. We are working to relax these assumptions.

The RTSA is implemented on top of the running time advisor (or RTA). Given a task's CPU demands, predictions of the load on a host, and a confidence level, the RTA predicts, as a confidence interval, the running time of the task on the host. The details of load measurement, prediction, and how the RTA computes its predictions of running time have been thoroughly documented [4, 6, 7, 5]. The Network Weather Service also provides load prediction [18]. The RTSA is similar in spirit to the focused addressing algorithm described by Ramamritham, et al [15], but it is based on sophisticated prediction techniques, makes no assumptions about host cooperation, and is designed to run on commodity operating systems.

Figure 1 shows how the different components of the system tie together. Every part of the system runs entirely at user level. The reasons for the complexity of the system are scalability and extensibility. Measurements and predictions of host load (and of other resources) are intended to be shared among all applications. There needs to be only a single sensor and predictor per resource. Furthermore, different application-level performance predictions (e.g., running time) can be computed from the same resource-level predictions. Finally, different adaptation advisors (e.g., the RTSA) can make use of the same lower-level information in pursuit of different goals.

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



Application
This Paper
Nominal time, slack, confidence, host list
Host, running time estimate
Real-time Scheduling Advisor
Nominal time confidence, host
Running time estimate (confidence interval)
Running Time Advisor
Library
Load Prediction Request
Load Prediction Response
Host Load Prediction System
Measurement Stream
Host Load Measurement System
Daemon

Figure 1. RTSA and context.

This paper concentrates on the RTSA component. We show how the RTSA uses the RTA's predictions to answer application level queries. We then evaluate RTSA performance in considerable detail, using a task model based on a Dv [1] scientific visualization application, and background workload (the competition) reproduced [8] from traces of a large set of hosts [4].

The primary purpose of our evaluation is to quantify the performance of the prediction-based strategy (called AR(16) after the underlying predictor), but we also compare AR(16) with a purely random strategy (RANDOM) and a purely measurement-based strategy (MEASURE). We make this comparison not because they are serious competitive algorithms, but because in an important sense AR(16) is an intermediate design. Like MEASURE, it operates greedily on behalf of the application, and, like RANDOM, it introduces beneficial randomness to avoid synchronization with other advisors. Unlike either, it can qualify its advice, reliably telling the application when the deadline can not be met. We find that these benefits come at a tiny additional cost over MEASURE.

The main conclusion that we draw is that it is feasible for an application to reliably achieve soft real-time goals in a typical distributed computing environment using the RTSA. While the RTSA can provide no guarantees to the application, it is empirically clear that its advice is quite accurate. The software and traces discussed in this paper are available from http://www.cs.northwestern.edu/∼pdinda.

2. Real-time scheduling advisor

The RTSA presents a simple interface to the application, one that can be implemented using several different low-overhead scheduling strategies. Figure 2 shows the API of the real-time scheduling advisor. `RTSARequest` expresses the scheduling problem: Choose a host from `hosts` such that a task with nominal running time `tnom`

```
int RTSAAdviseTask(RTSARequest  &req,
                   RTSAResponse &resp);
struct RTSARequest {
   double   tnom;
   double   sf;
   double   conf;
   Host     hosts[];
};
struct RTSAResponse {
   double                         tnom;
   double                         sf;
   double                         conf;
   Host                           host;
   RunningTimePredictionResponse runningtime;
};
struct RunningTimePredictionResponse {
   Host   host;
   double tnom;
   double conf;
   double texp;
   double tlb;
   double tub;
};
```

Figure 2. RTSA interface.

(t_{nom}), if started now, will complete in time $(1 + sf)t_{nom}$ or less with confidence `conf` ($conf$). We refer to `sf` (sf) as the slack factor. The RTSA's response consists of a copy of the request's t_{nom}, sf, and $conf$ values, the selected host, and an estimate of the task's running time, `RunningTimePredictionResponse`. This structure, which is returned by the RTA, includes the expected running time of the task, `texp` (t_{exp}), and the $conf$-level confidence interval for the running time, (`tlb`, `tub`) ((t_{lb}, t_{ub})).

It is important to note that the scheduling problem may not have a solution because of a lack of resources. If this is the case, the advisor will select the host which minimizes the running time of the task. It is the application's responsibility to verify, by using the `runningtime` field, whether the task is predicted to meet its deadline or not.

The `RTSAAdviseTask` call is implemented thus:

1 Construct an RTA request from $conf$ and t_{nom}.
2 Use the RTA to predict the running time of the task on each of the hosts as a confidence interval.
3 Find the subset of the hosts whose confidence interval upper bound is less than the deadline, $t_{ub} \leq (1 + sf)t_{nom}$. These are the *possible hosts*.
4 If there are no possible hosts, add the host with the minimum expected running time (t_{exp}) to the set of possible hosts.
5 Select a host at random from the set of possible hosts and return it and its corresponding `RunningTimePredictionResponse` to the caller via the `RTSAResponse`.

The RTSA tries to select a deadline-meeting host for its client application while it simultaneously tries to avoid contending with RTSAs operating on behalf of other applications by introducing randomness into its decisions. The amount of randomness possible depends on the load on the hosts and the slack factor the application permits. When load increases to the level where deadlines can not be met, the RTSA can inform the application of this, at which point we expect the application will back off and increase sf.

The above describes the prediction-based scheduling strategy, which is parameterized by the host load predictor that is used. In keeping with results of our study of host load prediction we use the AR(16) predictor [7] and thus we refer to this as the AR(16) strategy. We also studied two additional scheduling strategies: RANDOM and MEASURE. RANDOM simply recommends a randomly selected host. There is little chance of contention among RANDOM-based advisors. AR(16) degenerates to RANDOM when all the hosts are lightly loaded or slack factors are high. The MEASURE strategy measures the current load on each of the hosts and then selects the host with the minimum load. Obviously, it is very prone to contention. AR(16) degenerates to MEASURE when all the hosts are heavily loaded or slack factors are low.

RANDOM obviously has no overhead. MEASURE uses a host load sensor running on each of the hosts which introduces about 0.5% CPU usage. AR(16) adds a prediction filter to this sensor with unmeasurable additional overhead [6]. The CPU demands of the RTA and RTSA are tiny and the measurement and prediction messages are small. The RTA and RTSA do work only in response to a scheduling request from the application.

We evaluate the RTSA using three metrics. Our first metric is the probability that a deadline is met, which we estimate as the *fraction of deadlines met* for a randomly selected set of tasks. Ideally, the fraction of deadlines met will be the maximum feasible given the resources available, or the confidence level requested by the application, whichever is lower. The performance of the RTSA has more complexity than this metric captures, however, as the metric conflates failures due to prediction with failures due to a lack of resources or an overly low slack factor.

Our second metric, the *fraction of deadlines met when predicted*, addresses this. This metric is an estimate of the probability that a deadline will be met given that the RTSA claims it can be met—it tells us how trustworthy the advisor is. Recall that the application can rephrase its request and negotiate with the RTSA until some mapping can be found where the deadline will be met. This metric tells us how fruitful this process is likely to be. Ideally, the fraction of deadlines met when predicted will be identical to the confidence level requested by the application. For the non-predictive strategies, it is identical to the first metric, and so we do not present it.

Our final metric, the *average number of possible hosts*, is an estimate of the expected number of hosts on which the deadline can be met. It is a measure of the randomness a strategy introduces. Ideally, this metric will be as high as possible without affecting the other metrics. As it grows, the probability of RTSAs interfering declines. For the MEASURE strategy, it is one. For the RANDOM strategy, it is the number of the available hosts. For the AR(16) strategy, it depends on the factors mentioned above.

3. Workloads and scenarios

Performance evaluations are highly dependent on the workloads used. In effect, the purpose of the RTSA is to match its client's foreground workload with the background workloads on the hosts by predicting the background workloads and assigning tasks accordingly. In our evaluation, we use a simple parametric model to generate the tasks, but the background workload on each host is trace-based. The set of hosts to which the RTSA can schedule (the *scenario*) is also an important factor in RTSA performance.

The tasks we schedule arrive consecutively, with a delay between completion and the next arrival selected from $U(5, 15)$ seconds, a uniform distribution from 5 to 15 seconds. Each task is compute-bound with $t_{nom} \sim U(0.1, 100)$ seconds and $sf \sim U(0, 2)$. This model is chosen to be appropriate for the isosurface extraction stage of an interactive visualization pipeline [3]. $conf$ is set to 95% in all cases. We assume here that migration costs are negligible or can be factored out. For this specific application, data is inherently remote and thus migration happens regardless of where the task is executed.

Our background workloads are derived from a large family of long host load traces taken on a wide variety of machines. The host load measure is the Digital Unix 5 second load average. 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.

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 at Carnegie Mellon (CMU), big memory application servers at CMU, and desktop workstations at CMU. The characteristics of the traces are discussed in detail elsewhere [4].

We generate a background workload on a host by replaying one of these traces using a new technique called host load trace playback [8]. With no other work on the host, this background load results in the host's load signal repeating that of the load trace. When foreground work is added to the host, the two workloads share the machine according to the kernel's scheduling policy, and the foreground workload is slowed down accordingly by it.

Scenarios are represented by the group of traces being

Scenario	Description
4LS	4 hosts with high load/small epochs
4SL	4 hosts with low load/large epochs
4MM	2 hosts with high load/small epochs.
	2 hosts with low load/large epochs
5SS	5 hosts with low load/small epochs
4MS	2 hosts with high load/small epochs,
	2 hosts with low load/small epochs
4SM	2 hosts with low load/small epochs,
	2 hosts with low load/large epochs
2CS	2 large memory compute servers
2MP	2 very predictable hosts

Figure 3. Scenarios used in evaluation

played back. For the most part, we chose scenarios based on how our load traces are clustered by their statistics, the dominant statistics being the mean load and the mean epoch length of traces. We classified each trace as having "low" or "high" mean load, and "small" or "large" mean epoch lengths. The traces in a scenario could all be of low load, high load, or a "mixed" combination. Similarly, the scenario's traces could all be of small epochs, large epochs, or a mixed combination. The crossproduct forms nine classes. Unfortunately, because we have no traces which are simultaneously of high mean load and long mean epoch length, only six of the classes were realizable. Figure 3 lists these scenarios, which are constructed from the PSC traces.

We added two additional scenarios, also shown in the figure. In 2CS, the RTSA chooses between two compute servers. 2MP, based on CMU traces, includes two highly predictable hosts, which we use as an environment to test how multiple RTSAs might interact.

4. Evaluation

We evaluated the RTSA using a dedicated cluster, generating a background workload on each host using the trace appropriate to the scenario. Using the cluster, we ran large numbers of tasks randomized with respect to their starting time, the slack factor, the nominal time, and the strategy used. For each task, the RTSA's advice was followed, the task was executed on the suggested host, and the results logged. Using the testcases produced by applying this methodology, we then estimated the values for the three performance metrics under different constraints on the scenario, slack factor, and nominal time.

In the following, we begin by documenting our methodology. Next, we present the results for the representative 4LS scenario in considerable detail. This leads into a discussion of contention effects. Finally, we summarize our results. Except where noted, the differences we note are significant to $p = 0.05$, assuming the sample is random.

Our infrastructure consisted of dedicated Alphastation 255 hosts, each running Digital Unix 4.0D. One host is referred to as the recording host while the others are called measurement hosts. The recording host runs the RTSA and RTA and is where tasks are submitted. The measurement hosts are the hosts from which the RTSA must choose. Each measurement host runs the following components: the load playback tool, a load sensor, one or more prediction systems, and a cycle server. The configuration of the prediction systems is discussed elsewhere [7, 5]. The cycle 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.

To evaluate the RTSA given a particular scenario, we started up the infrastructure with a particular scenario, allowed it to quiesce for at least 600 seconds and ran a stream of consecutive tasks chosen according to the workload model. We scheduled each task using a randomly chosen strategy and record the strategy used, the arrival time (t_{now}), the nominal time (t_{nom}), the slack factor (sf), the predicted running time ([t_{lb}, t_{ub}], t_{exp}), and the actual running time (t_{act}). For the 4LS scenario, we ran 16,000 tasks. For each of the other scenarios we ran about 8,000 tasks. The information recorded for each task forms a *testcase*. Testcases ran at a rate of 222 per hour. Notice that this procedure allows us to make only unpaired comparisons [9, pp. 209–212] between the strategies.

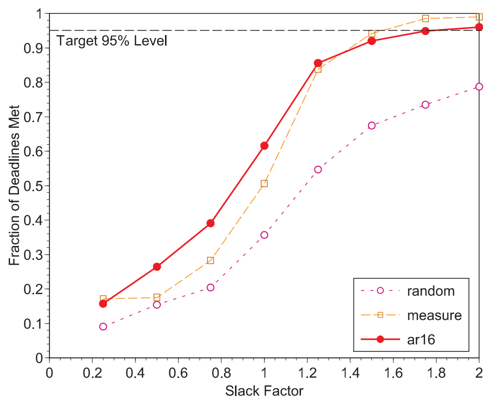
4LS scenario in detail

The performance of the RTSA on the 4LS scenario is representative of its performance on the other scenarios we examined. The primary difference is that the dependence of the metrics on the slack factor is more clearly visible because all of the hosts have high load.

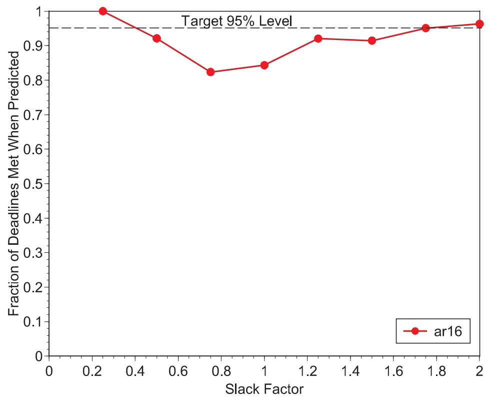
Figure 4 illustrates how the performance metrics vary with slack factor for each of the scheduling strategies using the 4LS scenario. A point represents the interval between it and the preceding point and contains about 667 testcases.

The fraction of deadlines met (Figure 4(a)) is a function of the quality of the scheduling strategy and the relationship of the slack factor to the resources available. When sf is very low, performance is dominated by the fact that it is exceedingly rare for a machine to have sufficiently low load to meet the deadline. There is no real benefit to prediction here. As sf increases, prediction begins to provide benefit over simple measurement.

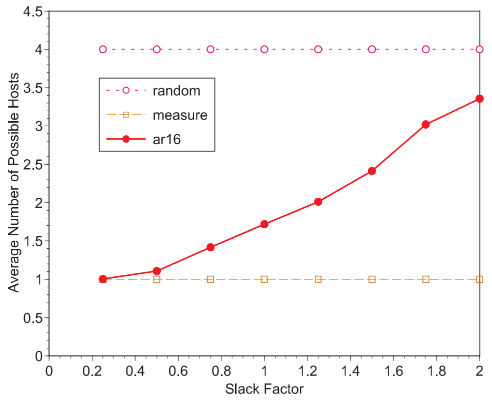
The benefit is maximal when sf and the available resources are "evenly matched", when the probability that there exists a host on which the deadline can be met reaches 50%. We refer to the sf for which this is true as the *critical slack factor*. Beyond the critical slack factor, the benefit of prediction over simple measurement begins to decline. For



(a) fraction of deadlines met



(b) fraction of deadlines met when predicted



(c) number of possible hosts

Figure 4. Performance versus slack factor, 4LS scenario, all nominal times.

AR(16), the fraction of deadlines met converges on the target level as sf continues to increase, while the metric converges on 1 for the MEASURE and RANDOM strategies.

Figure 4(b) shows how the fraction of deadlines met when predicted depends on the slack factor. Because the RANDOM and MEASURE strategies do not report a predicted running time to the application, we have excluded them from this graph. It is important to mention again that one of the benefits of the AR(16) strategy is that it does provide this additional feedback. The goal of this metric is to evaluate the quality of this feedback.

The curve in Figure 4(b) is remarkably different from the previous graph in that the dependence on slack factor has been drastically reduced. Ideally, this curve would be a flat line at the target confidence level, so there is still room for improvement. Interestingly, the non-ideal "dip" in the curve occurs around the critical slack factor. At low sf, a deadline-meeting host is rare, but the predicted running time on such as host is likely to be accurate due to its low load. At high sf, deadline-meeting hosts are plentiful, and so miss-predicting a running time leaves plenty of hosts available. Near the critical slack factor, however, predictor quality becomes critical.

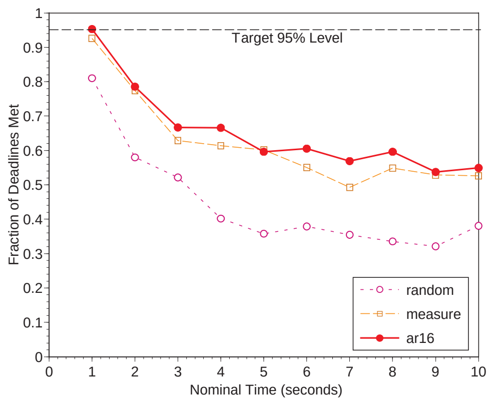
Prediction error is actually independent of the slack factor (the RTA is unaware of sf), but near the critical slack factor, the differences in quality between different predictors are highlighted the most. We also studied other predictors and found that the "correct" choice of AR(16) was important in this regime. Better predictors would lead to less of a dip in performance around the critical slack factor.

Figure 4(c) shows how the number possible hosts in the RTSA's advice varies with the slack factor for the three strategies. At low sf, the randomness of AR(16) converges with that of MEASURE, while at high sf, it converges with RANDOM. Notice that even near the critical slack factor, AR(16) chooses from approximately two hosts on average, introducing at least that degree of randomness. Prediction-based strategies such as AR(16) are a useful middle ground, introducing as much randomness as is possible given the slack, while working to meet the individual task's goals.

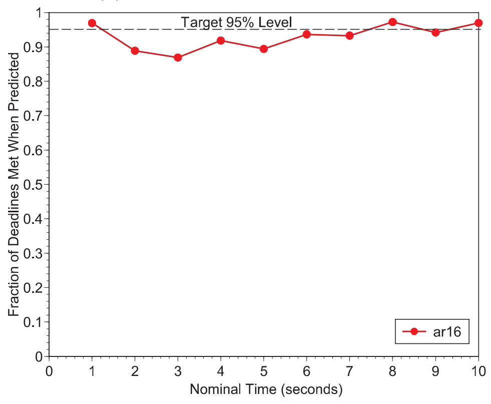
Figure 5 shows the dependence of the performance metrics on the nominal time for the 4LS scenario. To produce the figure, testcases of all slacks from 0 to 2 were aggregated. A point represents all nominal times between it and the preceding point and contains about 533 testcases.

Figure 5(a) shows the fraction of deadlines met as a function of t_{nom}. The DEC Unix scheduler gives a priority boost to tasks as they finish I/O operations, and this benefits shorter tasks more than longer tasks. The other load on a host, even if it is considerable as with the 4LS hosts, has little effect on a sub-second task, and thus most of these tasks meet their deadlines, even using very simple strategies. For longer tasks, this other load becomes increasingly visible, and the chance of a deadline being met declines. Notice, however, that the strategy can not be too simple—RANDOM lags behind even for tiny tasks.

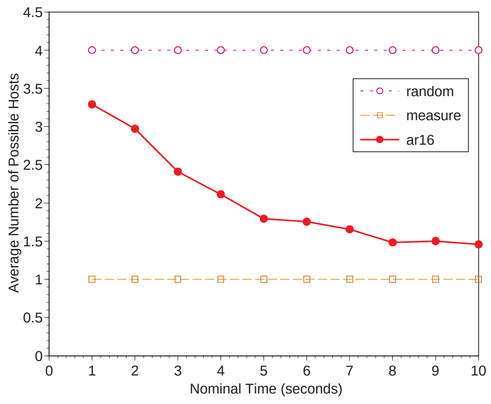
Figure 5(b) shows how the fraction of deadlines met when predicted varies with the nominal time of the task in the 4LS scenario. Once again, RANDOM and MEASURE have been elided because they are not predictive. For the



(a) fraction of deadlines met

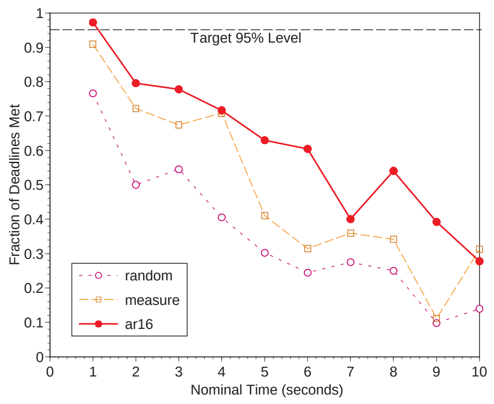


(b) fraction of deadlines met when predicted

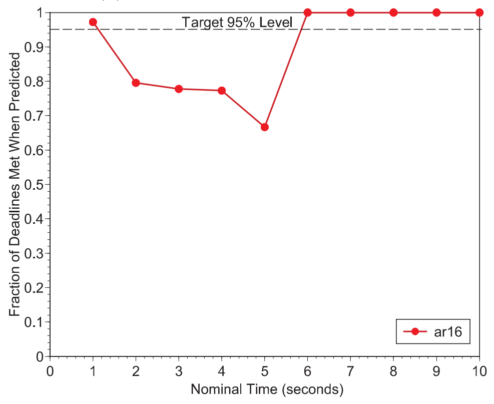


(c) number of possible hosts

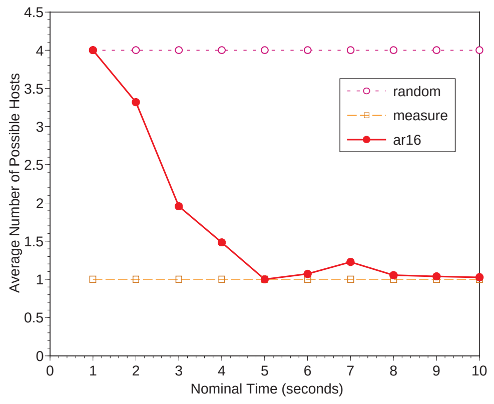
Figure 5. Scheduling performance versus nominal time, 4LS scenario, all slack factors.



(a) fraction of deadlines met



(b) fraction of deadlines met when predicted



(c) number of possible hosts

Figure 6. Scheduling performance versus nominal time, 4LS scenario, critical slack.

AR(16) strategy, the metric is largely independent of the nominal time of the task and close to the target level, although there is a slight dip for 2–6 second tasks.

Figure 5(c) shows how the number of possible hosts varies with the nominal time. Not surprisingly, because short tasks are essentially oblivious to other load on any host, and because the RTA is able to predict this obliviousness, the RTSA is able to introduce considerable randomness into its choice of hosts using the AR(16) strategy. For very small tasks, the choice is nearly as random as that of the RANDOM strategy. As tasks increase in size, there are fewer options as to where to schedule them, and so the advisor constrains the randomness it introduces.

Figure 6 shows the dependence of the performance metrics on the nominal time for the 4LS scenario near the critical slack factor. Each point on the graph represents approximately 67 testcases. This is a sufficiently small number of testcases that some of the differences we note are only statistically significant at $p > 0.05$. For all of the graphs AR(16) is better than RANDOM at a 95% confidence level.

Near the critical slack, AR(16) is much better at meeting deadlines than the RANDOM or MEASURE strategies. As Figure 6(a) shows, AR(16) results in 20 to 40% more deadlines being met than RANDOM, and 5 to 30% more deadlines being met than MEASURE. As before, the fraction of deadlines met declines with increasing nominal time.

Compared to its value over all slack factors (Figure 5(b)), the dependence of the fraction of deadlines met when predicted on the nominal time is considerably different near the critical slack factor (Figure 6(b)). In particular, the dip in performance for medium-sized tasks is deeper and we see more extreme behavior. As the nominal time increases, the I/O boost shields us less and we become more sensitive to bad predictions. Near the critical slack factor, these errors are magnified—even small errors can lead AR(16) erroneously to conclude that the deadline can be met.

Figure 6(c) shows how the number of possible hosts depends on the nominal time near the critical slack factor. The graph is similar to the overall dependence presented in Figure 5(c) in that the amount of randomness introduced is very high for small tasks and declines as task size increases.

Contention

An important concern facing measurement- or prediction-based RTSAs is that of contention between different advisors. All RTSAs observe the same measurements and predictions of signals that characterize resource availability, but they do not coordinate their scheduling decisions. This lack of coordination helps to make measurement- or prediction-based RTSAs scalable, but it is conceivable that it might synchronize their actions, leading to performance degradation for all of them.

Two factors ameliorate the chance of this. First, requests from interactive applications are unlikely to become synchronized or even correlated because they are aperiodic and arrive in response to user actions. If multiple interactive applications using RTSAs are running, each is responding to a different user, and we would expect that the users' actions are not synchronized. The second factor is limited to prediction-based strategies such as AR(16). These strategies convert excess slack into randomness in their scheduling decisions, which can serve to break any synchronization that may be starting. The number of possible hosts metric is our measure of this randomness.

To see if these two factors prevented repeated collisions in practice, we ran testcases using four competing RTSAs on the two host 2MP scenario. We found that there was no favored client—using AR(16), each RTSA had about the same performance in terms of our three metrics. However, this was also the case for MEASURE. That AR(16) didn't do better than MEASURE here suggests that the primary factor in avoiding contention is that task submission is not synchronized over time or between clients.

Summary of results

Our main result is that the prediction-based AR(16) strategy is indeed an effective RTSA implementation. This result is based on studying each of the scenarios in Figure 3, including the 4LS scenario we described above. The remainder of this section summarizes the conclusions of our study and our recommendations.

The MEASURE and AR(16) strategies significantly outperform the RANDOM strategy at all slack factors and all nominal times in terms of the fraction of deadlines that are met. AR(16) almost always performs at least as well as MEASURE and considerably improves on its performance near the critical slack factor. As one might expect, performance declines for all the strategies as the nominal time of the task increases.

Beyond meeting deadlines with higher probability than MEASURE, AR(16) is the only strategy which can indicate to the application if it believes a deadline can be met. We measure the effectiveness of this using the fraction of deadlines met when predicted metric. Ideally, this number would be exactly equal to the target confidence level. We find that with AR(16) it is close and is relatively independent of slack factor and nominal time. This means that if AR(16) claims that the deadline can be met on the host it chooses, the application can be confident that if it sends the task to that host, it will indeed meet its deadline.

AR(16) is also able to introduce appropriate amounts of randomness into its scheduling decisions, and this randomness can help to avoid synchronization between multiple independent RTSAs. The distribution of missed deadlines is identical for the three strategies.

Our overall conclusion is that using the AR(16) strategy produces the best results in terms of all three metrics. It is clear that RANDOM is insufficient—a MEASURE strategy, at least, is indicated. AR(16) has little additional overhead over a MEASURE strategy, and it has several benefits. It can increase the probability that the deadline is met in most cases. It can inform the application, accurately, whether the deadline can be met. Finally, it can introduce appropriate randomness into its scheduling decisions.

5. Conclusion and future work

We described the interface and implementation of a real-time scheduling advisor that is based on the prediction of host load and is designed to serve an individual distributed interactive application by recommending hosts for the application's soft real-time tasks. Because the RTSA operates in a shared, unreserved computing environment that it does not control, deadlines can be missed due both to RTSA errors and to a lack of resources.

To better understand the performance of the RTSA, we conducted a randomized performance evaluation of the system using a trace-based background workload and a synthetic foreground workload model. In addition to the prediction-based RTSA strategy we also considered a purely random strategy, and a strategy based on measurement of load. In contrast to the random and measurement-based strategies, the prediction-based strategy is able to provide the application with a critical (and correct) additional bit of feedback: whether the deadline will be met given its choice of host.

The main conclusion is that the prediction-based strategy is highly effective. The prediction-based strategy is able to increase the probability that a deadline will be met over that of the measurement-based strategy, and well over that of the random strategy. Furthermore, the prediction-based strategy's additional feedback is quite good—when the application is told that the deadline can be met, the chances it will actually be met are very high. Finally, the prediction-based strategy introduces randomness into its decision-making at a level appropriate to the difficulty of the application's request. This randomness helps to keep independent RTSAs from synchronizing.

We are currently working on extending the RTSA and its underlying software to support tasks with significant communication requirements.

References

[1] M. Aeschlimann, P. Dinda, L. Kallivokas, J. Lopez, B. Lowekamp, and D. O'Hallaron. 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)*, pages 1833–1839, Las Vegas, NV, June 1999. CSREA Press.

[2] F. Berman and R. Wolski. Scheduling from the perspective of the application. In *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing HPDC96*, pages 100–111, August 1996.

[3] P. Dinda, B. Lowekamp, L. Kallivokas, and D. O'Hallaron. 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)*, volume 1586 of *Lecture Notes in Computer Science*, pages 309–318. Springer-Verlag, San Juan, PR, 1999. Extended version as CMU Technical Report CMU-CS-TR-98-174.

[4] P. A. Dinda. The statistical properties of host load. *Scientific Programming*, 7(3,4), 1999. 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.

[5] P. A. Dinda. Online prediction of the running time of tasks. *Cluster Computing*, 2002. To appear, earlier version in HPDC 2001, summary in SIGMETRICS 2001.

[6] P. A. Dinda and D. R. O'Hallaron. An extensible toolkit for resource prediction in distributed systems. Technical Report CMU-CS-99-138, School of Computer Science, Carnegie Mellon University, July 1999.

[7] P. A. Dinda and D. R. O'Hallaron. Host load prediction using linear models. *Cluster Computing*, 3(4), 2000. Earlier version in HPDC 1999.

[8] P. A. Dinda and D. R. O'Hallaron. Realistic CPU workloads through host load trace playback. In *Proc. of 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR2000)*, volume 1915 of *Lecture Notes in Computer Science*, Rochester, New York, May 2000. Springer-Verlag.

[9] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc., 1991.

[10] J. F. Kurose and R. Chipalkatti. Load sharing in soft real-time distributed computer systems. *IEEE Transactions on Computers*, C-36(8):993–1000, August 1987.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[12] J. Lopez and D. O'Hallaron. Runtime support for adaptive heavyweight services. In *Proc. of 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR2000)*, volume 1915 of *Lecture Notes in Computer Science*, pages 221–234, Rochester, NY, 2000. Springer-Verlag.

[13] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997. To Appear.

[14] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[15] K. Ramamrithham, J. A. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computer Systems*, 38(8):1110–1123, August 1989.

[16] D. C. Schmidt, A. Gokhale, T. H. Harrison, and G. Parulkar. A high-performance endsystem architecture for real-time CORBA. *IEEE Communication Magazine*, 14(2), February 1997.

[17] J. Stankovic, K. Ramamritham, D. Niehaus, M. Humphrey, and G. Wallace. The spring system: Integrated support for complex real-time systems. *Real-Time Systems Journal*, 16(2/3), May 1999.

[18] R. Wolski, N. Spring, and J. Hayes. Predicting the CPU availability of time-shared unix systems. In *Proceedings of the Eighth IEEE Symposium on High Performance Distributed Computing HPDC99*, pages 105–112. IEEE, August 1999. Earlier version available as UCSD Technical Report Number CS98-602.

[19] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):55–73, April 1997.