Predicting the CPU Availability of Time-shared Unix Systems on the Computational Grid [*]

Rich Wolski,[†] Neil Spring,[‡] and Jim Hayes[§]

Abstract

In this paper we focus on the problem of making short and medium term forecasts of CPU availability on time-shared Unix systems. We evaluate the accuracy with which availability can be measured using Unix load average, the Unix utility `vmstat`*, and the Network Weather Service CPU sensor that uses both. We also examine the autocorrelation between successive CPU measurements to determine their degree of self-similarity. While our observations show a long-range autocorrelation dependence, we demonstrate how this dependence manifests itself in the short and medium term predictability of the CPU resources in our study.*

1 Introduction

Improvements in network technology have made the distributed execution of performance-starved applications feasible. High-bandwidth, low-latency local-area networks form the basis of low-cost distributed systems such as the HPVM [14], the Berkeley NOW [9], and various Beowulf [25] configurations. Similarly, common carrier support for high-performance wide-area networking is fueling an interest in aggregating geographically dispersed and independently managed computers. These large-scale *metacomputers* are capable of delivering greater execution performance to an application than is available at any one of their constituent sites [4]. Moreover, performance-oriented distributed software infrastructures such as Globus [12], Legion [18], Condor [26], NetSolve [7], and HPJava [6] are attempting to knit vast collections of machines into *computational grids* [13] from which compute cycles can be obtained in the way electrical power is obtained from an electrical power utility.

[*]Supported by NSF grant ASC-9701333 and Advanced Research Projects Agency/ITO under contract #N66001-97-C-8531.

[†]email: rich@cs.utk.edu

[‡]email: nspring@cs.washington.edu

[§]email: jhayes@cs.ucsd.edu

One vexing quality of these ensemble systems is that their performance characteristics vary dynamically. In particular, it is often economically infeasible to dedicate a large collection of machines and networks to a single application, particularly if those resources belong to separate organizations. Resources, therefore, must be shared, and the contention that results from this sharing causes the deliverable performance to vary over time. To make the best use of the resources that are at hand, an application scheduler (be it a program or a human being) must make a *prediction* of what performance will be available from each.

In this paper, we examine the problem of predicting available CPU performance on Unix time-shared systems for the purpose of building dynamic schedulers. In this vein, the contributions it makes are:

- an exposition of the *measurement error* associated with popular Unix load measurement facilities with respect to estimating the available CPU time a process can obtain,
- a study of one-step-ahead forecasting performance obtained by the Network Weather Service [29, 30] (a distributed, on-line performance forecasting system) when applied to CPU availability measurements,
- an analysis of this forecasting performance in terms of the autocorrelation present between consecutive measurements of CPU availability, and,
- verification of this analysis through its application to longer-term forecasting of CPU availability.

Our results are somewhat surprising in that they demonstrate the possibility of making short and medium term predictions of available CPU performance despite the presence of long-range autocorrelation and potential self-similarity. Since chaotic systems typically exhibit self-similar performance characteristics, self-similarity is often interpreted as an indication of unpredictability. The predictions we obtained, however, exhibited a mean absolute error of less than 10%, typically making them accurate enough for use in dynamic process scheduling.

In the next section (Section 2), we detail the accuracy obtained by three performance measurement methodologies when applied to a collection of compute servers and workstations located in the U.C. San Diego Computer Science and Engineering Department. Section 3 briefly describes the Network Weather Service (NWS) [30, 29] – a distributed performance forecasting system designed for use by dynamic schedulers. The NWS treats measurement histories as time series and uses simple statistical techniques to make short-term forecasts. Section 3 also presents the analysis the forecasting errors generated by the NWS forecasting system for the resources we monitored in this study. Finally, we conclude with a discussion of these results in the context of dynamic scheduling for metacomputing and computational grid systems, and point to future research directions in Section 4.

2 Measurements and measurement error

For dynamic process scheduling, the goal of CPU prediction is to gauge the degree to which a process can be expected to occupy a processor over some fixed time interval. For example, if "50% of the CPU" is available from a time-shared CPU, a process should be able to obtain 50% of the time-slices over some time interval. Typically, the availability percentage is used as an expansion factor [11, 23, 15, 1] to determine the potential execution time of a process. If only 50% of the time-slices are available, for example, a process is expected to take twice as long to execute as it would if the CPU were completely unloaded[1]. We have used *CPU availability* to successfully schedule parallel programs in shared distributed environments [24, 2]. Other scheduling systems such as Prophet [27], Winner [1], and MARS [15] use Unix load average to accomplish the same purpose.

In this section, we detail the error that we observed while measuring CPU availability. We report results gathered by monitoring a collection of workstations and compute servers in the Computer Science and Engineering Department at UCSD. We used the CPU monitoring facilities currently implemented as part of the Network Weather Service [30]. Each series of measurements spans a 24 hour period of "production" use during August of 1998. Despite the usual summertime hiatus from classwork, the machines in this study experienced a wide range of loads as many of the graduate students took the opportunity to devote themselves to research that had been neglected during the school year. As such, we believe that the data is a representative sample of the departmental load behavior.

[1]This relationship assumes that the execution interval is sufficiently large with respect to the length of a time-slice so that possible truncation and round-off errors are insignificant.

2.1 Measurement methods

For each system, we use the NWS CPU monitor to periodically generate three separate measurements of current CPU availability. The first is based on the Unix load average metric which is a one-minute smoothed average of run queue length. Almost all Unix systems gather and report load average values, although the smoothing factor and sampling rates vary across implementations. The NWS sensor uses the utility `uptime` to obtain a load average reading without special access privileges[2]. Using a load average measurement, we compute the available CPU percentage as

$$available\_cpu_{load\_avg} = (\frac{1.0}{Unix\_load\_avg + 1.0}) * 100\% \quad (1)$$

indicating the percentage of CPU time that would be available to a newly created process. Fearing load average to be insensitive to short-term load variability, we implemented a second measurement technique based on the utility `vmstat` which provides periodically updated readings of CPU $idle$ time, consumed $user$ time, and consumed $system$ time, presented as percentages.

$$available\_cpu_{vmstat} = (idle + \frac{user}{rp+1} + W * \frac{system}{rp+1}) \quad (2)$$

where rp is a smoothed average of the number of running processes over the previous set of measurements, and W is a weighting factor equal to $user$ time. The rationale is that a process is entitled to the current idle time, and a fair share of the user and system time. However, if a machine is used as a network gateway (as was the case at one time in the UCSD CSE Department) user-level processes may be denied CPU time as the kernel services network-level packet interrupts. In our experience, the percentage of system time that is shared fairly is directly proportional to the percentage of user time, hence the W factor.

Lastly, we implemented a *hybrid* sensor that combines Unix load average and `vmstat` measurements with a small probe process. The probe process occupies the CPU for a short period of time (currently 1.5 seconds) and reports the ratio of the CPU time it used to the wall-clock time that passed as a measure of the availability it experienced. The hybrid runs its probe process much less frequently than it measures $available\_cpu_{load\_avg}$ and $available\_cpu_{vmstat}$ as these quantities may be obtained much less intrusively. The method (`vmstat` or load average) that reports the CPU availability measure closest to that experienced by the probe is chosen to generate all measurements until the next

[2]The current implementation of the NWS runs completely without privileged access, both to reduce the possibility of breeching site security and to provide data that is representative of the performance an "average" user can obtain. For a more complete account of the portability and implementation issues associated with the NWS, see [30].

probe is run. In the experiments described in this paper, the NWS hybrid sensor calculated $available\_cpu_{load\_avg}$ and $available\_cpu_{vmstat}$ every 10 seconds (6 times per minute), and probed once per minute. The method that was most accurate with respect to the probe was used for the subsequent 5 measurements each minute.

We are interested in the deliverable performance available for a full-priority Unix process that occupies the CPU for an appreciable amount of time. However, both Unix load average and the `vmstat` method are unable to sense the presence of lower priority or "nice" processes. It is our experience that users frequently run low-priority, background processes, particularly on shared resources to try and soak up any unused CPU time without arousing the ire of the departmental system administrators. To overcome this problem, the hybrid sensor uses the difference between the probe process and the most accurate method as a *bias* value and adjusts all subsequent measurements by this value. The assumption is that the probe will not be affected by lower-priority processes and hence will be able to bias the skewed measurements derived from the Unix load average or `vmstat`.

The advantage of using load average, `vmstat`, and the NWS-hybrid to derive measurements of CPU availability is that they are relatively non-intrusive. Both `vmstat` and `uptime` read protected Unix devices to access performance statistics maintained in the kernel. Presumably, these are not heavy-weight operations in most Unix implementations. Indeed, we notice little difference in CPU availability if two instances of either method are executing, which indicates that the load they generate is not measurable given their relative sensitivities. The NWS-hybrid, however, uses a short term spinning process which (in this study) executes once per minute. We have determined though experimentation that the shortest probe duration that is useful is 1.5 seconds. The overhead, then is $1.5/60$ seconds or 2.5%.

2.2 Measurement accuracy

To determine the accuracy of these three methods, we compare the readings they generate with the percentage of CPU cycles obtained by an independent ten-second, CPU-bound process which we will refer to as the *test process*. The *test process* executes and then reports the ratio of CPU time it received (obtained through the `getrusage()` system call) to total execution time (measured in wall-clock time) as the percentage of the CPU it was able to obtain. Measurement error, then, is defined as

$$Measurement\ Error_t = \qquad (3)$$
$$|Measurement_t - Test\ Process\ Observation_t|$$

where $Measurement_t$ is the measurement of CPU availability taken at time t, and $Test\ Process\ Observation_t$ is the availability observed by a *test process* at time t. To avoid possible contention between the sensors and the test-process, we use the measurement taken most immediately before the *test process* executes as $Measurement_t$.

Host Name	Load Average	vmstat	NWS Hybrid
thing2	9.0%	11.2%	11.1%
thing1	6.4%	7.5%	6.1%
conundrum	34.1%	32.7%	4.4%
beowulf	6.3%	6.5%	7.5%
gremlin	4.0%	3.2%	4.1%
kongo	12.8%	12.9%	41.3%

Table 1. Mean Absolute Measurement Errors during a 24-hour, mid-week period

Table 1 details the measurement errors we observed for different hosts at UCSD. Each column shows the mean absolute difference between the CPU availability percentage quoted by the corresponding measurement method, and the availability percentage observed by the *test process*.

The hosts *thing1*, *thing2*, and *conundrum* are interactive workstations used for research by graduate students, while *beowulf*, *gremlin*, and *kongo* are general departmental servers available to faculty and students. Most of the errors are reasonably small and fairly equivalent across methods, given the dynamic nature of the systems we monitored. An error of 10% or less, for example, is considered useful for scheduling [23]. The notable exceptions are *conundrum* and *kongo*. On *conundrum*, a background process was running with Unix `nice` priority of 19 in an attempt to use otherwise unused CPU cycles. However, the *test process* runs with full priority, pre-empting the background process. The Unix load average and `vmstat` methods do not consider process priority, however, and record the system as being busy. The probe bias used by the NWS-hybrid method, however, correctly recognizes the priority difference and yields a more accurate measurement.

On *kongo* the NWS-hybrid performs dismally. During the monitor period, a long-running, full-priority process was executing on kongo. Typical Unix systems increase the rate at which process priority degrades while executing as a function of their CPU occupancy. A long-running process, therefore, will be temporarily evicted in favor of a short-running, full-priority process like the probe used by the NWS-hybrid sensor. The 1.5 second execution time of the probe is not long enough for it to contend with the long-running process, so the NWS-hybrid method does not sense its presence. The ten-second *test process*, however, executes long enough to share the processor with the resident long-running process and, consequently, receives a fraction better measured by both load average and `vmstat`. It is possible to increase the probe time of the NWS-hybrid sensor, with a corresponding increase in intrusiveness. We are working on less intrusive techniques to address this problem.

For the purposes of predicting availability, the measurement error we observe serves as a upper bound on the accuracy of our forecasts. That is, we do not expect to forecast with greater accuracy that we can measure. In general, at UCSD, the measurement errors we can obtain from these three methodologies are small enough so that measurements prove useful for scheduling. However, obtaining an accurate measure is complicated by the process priority mechanisms employed by Unix, and care must be taken when choosing a measurement methodology.

Host Name	Load Average	vmstat	NWS Hybrid
thing2	**8.9%** (9.0%)	**8.6%** (11%)	**10%** (11%)
thing1	**6.4%** (6.4%)	**7.0%** (7.5%)	**5.3%** (6.1%)
conundrum	**34%** (34%)	**32%** (32%)	**4.3%** (4.4%)
beowulf	**6.2%** (6.3%)	**6.8%** (6.5%)	**6.9%** (7.5%)
gremlin	**4.0%** (4.0%)	**2.6%** (3.2%)	**3.0%** (4.1%)
kongo	**12%** (12%)	**12%** (12%)	**41%** (41%)

Table 2. Mean **True Forecasting Errors** and Corresponding Measurement Errors (in parentheses) for UCSD Hosts during a 24-hour, mid-week period

3 Forecasting

Forecasting, in this setting, is the prediction of the CPU availability that the *test process* will observe. We treat histories of measurements generated by the each of the methods described in Section 2 as statistical time series. In this section, we discuss our methodology for using these time series to predict CPU availability, then compare the predictions generated with both subsequent measurements and subsequent *test process* observations to understand the error involved in the processes of prediction and forecasting. In Section 3.1 we discuss autoregressive and self-similar characteristics of these time series, and describe the effect of these characteristics on the accuracy of the predictions. In Section 3.2 we discuss the implications these characteristics have on predictions made for a longer time frame, and present additional results to show the increase in prediction error.

In previous work describing the NWS [29, 30, 31], we have proposed a methodology for making one-step-ahead predictions using computationally inexpensive time-series analysis techniques. Rather than use a single forecasting model, the NWS applies a collection of forecasting techniques to each series, and dynamically chooses the one that has been most accurate over the recent set of measurements. This method of dynamically identifying a forecasting model has been shown to yield forecasts that are equivalent to, or slightly better than, the best forecaster in the set [29]. To be efficient, each of the techniques must be relatively cheap to compute. We have borrowed heavily from methodologies used by the digital signal processing community [19] in our implementation. A complete description of each method and its relative advantages is provided in [29], [19], and [16]. Briefly summarized, each method uses a "sliding window" over previous measurements to compute a one-step-ahead forecast based either on some estimate of the mean or median of those measurements.

To evaluate the accuracy of each forecast, we examine two forms of error. The first, given by Equation 4, compares a forecast for a specific time frame to the *test process* observation that is eventually made in that time frame. We term this form of error the *true forecasting error* as it represents the actual error a scheduler would observe. Note that, in the one-step-ahead case, the time at which the forecast is generated occurs immediately before the time frame in which the *test process* runs, hence the subscripts on the terms $Forecast_{t-1}$ and $Test\ Process\ Observation_t$ respectively. To distinguish the amount of error that results from measurement inaccuracy from error introduced by prediction, we also compute the *one step ahead prediction error* as given by Equation 5. This error represents the inaccuracy in predicting the next measurement that will be gathered in a particular series capturing the predictability of the series.

$$True\ Forecasting\ Error_t = \qquad (4)$$
$$|Forecast_{t-1} - Test\ Process\ Observation_t|$$
$$one\ step\ ahead\ prediction\ error = \qquad (5)$$
$$|Forecast_{t-1} - Measurement_t|$$

where $Forecast_{t-1}$ is the NWS forecast of CPU availability made at time $t-1$ for time t and $Test\ Process\ Observation_t$ and $Measurement_t$ are defined in Section 2.

Table 2 shows both the mean true forecasting error in boldface type and the mean measurement error (defined in Equation 3 and presented in Table 1) in parentheses.If the true forecasting errors and measurement errors are approximately the same, the process of predicting what the next measurement will be is not introducing much error. Table 3 illustrates this observation further. In it, we show the mean one-step-ahead prediction error, using the NWS forecasting techniques, for each measurement method on each of the systems that we studied. On each of these systems, the one-step-ahead prediction error is less than 5%. It is somewhat surprising that the one-step-ahead prediction error does not contribute more to the overall inaccuracy associated with predicting the *test process* values.

The instances in which forecast accuracy is better than measurement accuracy are curious. An analysis of the measurement and forecasting residuals is inconclusive with respect to the significance of this difference. Since the effect

Host Name	Load Average	vmstat	NWS Hybrid
thing2	1.2%	4.9%	1.8%
thing1	1.7%	3.1%	2.8%
conundrum	0.4%	0.2%	0.2%
beowulf	1.8%	3.1%	3.5%
gremlin	1.0%	2.1%	2.0%
kongo	0.1%	0.1%	0.1%

Table 3. Mean Absolute One-step-ahead Prediction Errors during a 24-hour, mid-week period

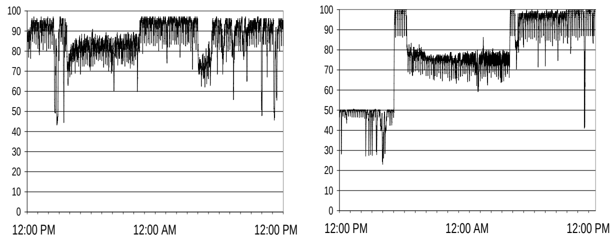

Figure 1. CPU Availability Measurements (using Unix Load Average) for *thing1* (left) and *thing2* (right).

is generally small, however, we omit that analysis in favor of brevity and make the less precise observation that measurement and forecasting accuracy are approximately the same.

3.1 CPU autocorrelation and predictability

A plot of the autocorrelations as a function of previous lags reveals that CPU availability changes slowly with respect to time and hence can be predicted relatively accurately in the short term. Figure 1 shows time series plots of CPU availability measurements using Unix load average taken from *thing1* and *thing2*. In Figure 2 we show the first 360 autocorrelations for each series.

From both the time series and the plot of the autocorrelations, it is clear that events occurring even hours apart are correlated. However, the slow rate of decay in the autocorrelation function is suggestive of self-similarity, and self-similarity is often a manifestation of an unpredictable, chaotic series [5]. Recent studies of network packet traffic [20], World-Wide-Web traffic [8], network protocol performance [22], transmitted video traffic [3], and networked file systems [17] all point to self-similarity as an inherent property of modern distributed systems. Of particular interest is the work by Dinda and O'Halloran [10] in which the authors rigorously analyze Unix load average data from a large number of computational settings. The focus of their analysis is on the degree of self-similarity and long-range autocorrelation present in a set of traces taken from a large population of machines. In almost all cases, their work shows that CPU load (for the cases that they examined) is self-similar.

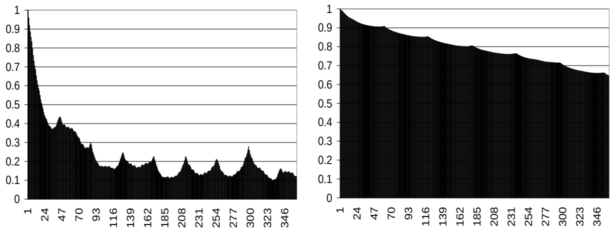

Figure 2. CPU Availability Autocorrelations using Unix Load Average for *thing1* (left) and *thing2* (right).

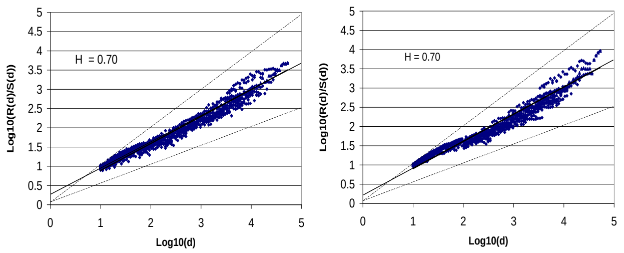

Figure 3. Pox Plot of CPU Availability using Unix Load average from *thing1* (left) and *thing2* (right)

While there are several ways to characterize the degree of self-similarity in a series, the most common techniques estimate the Hurst parameter for the series. We defer to the work of Mandelbrot, Taqqu, Willinger, Leland, and Wilson [21, 20, 28], as well as the references cited in the previous paragraph, for a thorough exposition of the Hurst effect, its relationship to self-similarity, and various techniques for Hurst parameter estimation. For the purposes of determining self-similarity, though, it is enough to show that the Hurst parameter H of a series is likely to be between 0.5 and 1.0. To estimate H for the data we gathered as part of this study, we use R/S analysis [21, 3] and pox plots [20]. Briefly, for a given set of observations $(X_k : k = 1, 2, ..., n)$, with sample mean $\overline{X}(n)$ and sample variance $\overline{S^2}(n)$, the *rescaled adjusted range statistic* or R/S statistic is calculated as $R(n)/S(n) = [max(0, W_1, W_2, ...W_n) - min(0, W_1, W_2, ...W_n)]/S(n)$ where $W_k = (X_1 + X_2 + ... + X_k) - k\overline{X}(n)(k >= 0)$. The expected value $E[R(n)/S(n)] \sim cn^H$ for some some constant c as $n \rightarrow \infty$ where H is the Hurst parameter for the series. By partitioning the series of length N into non-overlapping segments of length d, and calculating $R(d)/S(d)$ for each segment, as $(1 \leq d \leq N)$ we obtain $\lfloor d/N \rfloor$ samples of $R(d)/S(d)$. Plotting $log_{10}(R(d)/S(d))$ versus $log_{10}(d)$ for each of these samples yields a pox plot for the series. Figure 3 shows pox plots of CPU availability measurements using Unix load average for *thing1* and *thing2* over a one-week long period.

The two dotted lines in each figure depict slopes of 0.5 and 1.0. By inspection, any sort of "best fit" line for this data is likely to have a slope greater that 0.5 and less than 1.0, hence we can conclude that the Hurst parameter H falls somewhere in this range. In the figures, we show a least-squares regression line (solid) for the average $log_{10}(R(d)/S(d))$ value for each value of $log_{10}(d)$. The slope of this line estimates the Hurst parameter as 0.70 for both *thing1* and *thing2* in the figure. In the second column of Table 4, we give the Hurst parameter estimations for each of the hosts in our study using this technique. The pervasiveness of these observations across our set of experiments supports the previous work of Dinda and O'Halloran. We, therefore, surmise that the CPU availability exhibits long-range autocorrelation and is either self-similar (as noted in [10]) or short-term self-similar as described in [17].

3.2 Longer term prediction of CPU availability

Despite the long-range autocorrelation present in CPU availability measurements, the data in Tables 2 and 3 show that one-step-ahead CPU availability is relatively predictable. The slowly decaying autocorrelation between measurements means that recent history is often a good predictor of the short-term future. That is, self-similarity does not imply short-term unpredictability.

Self-similarity does mean, however, that averaging values over successively larger time scales will not produce time series which are dramatically smoother. For a self-similar series $X_1, X_2, ...X_n$, with Hurst parameter H, and aggregation level m, the averaged series $X^{(m)}$ has variance $Var(X^{(m)}) \sim cm^{-\beta}, H = 1 - \beta/2$ as $m \to \infty$ where $X^{(m)}$ is the series $X_k^{(m)} = \frac{X_{km-m+1} \dots X_{km}}{m}, k >= 1$. In other words, the variance of the average values of $X^{(m)}$ decreases more slowly than the aggregation level m increases as $m \to \infty$.

It is an estimate of average CPU availability (and not a one-step-ahead prediction for the next 10 second time frame) that is most useful to a scheduler, as process execution time may be span minutes, hours, or days. By the relationship shown above, we would expect the variance associated with a prediction of the average availability over interval m to be no worse than that for a one-step-ahead prediction. Table 4 compares the variance of an aggregated series $X^{(m)}$ where m corresponds to a five-minute interval with that of the original series for each of the hosts in our study. Except for *kongo* and the NWS hybrid sensor running on *conundrum*, the variance in each aggregated series is lower, as expected[3].

[3]The *conundrum* case is curious as it appears self-similar, but the variance increases as the series is aggregated. We are studying this case to determine why this experiment defi es the conventional analysis. The *kongo* value for the NWS hybrid sensor, however, is due to the leading constant c in the expression $Var(X^{(m)}) \sim cm^{-\beta}$. Additional aggregation of this series reveals a decreasing variance.

Host	Est.	Load Average		vmstat		NWS Hybrid	
	H	orig.	**300s**	orig.	**300s**	orig.	**300s**
thing2	0.70	0.0348	**0.0338**	0.0431	**0.0351**	0.0321	**0.0315**
thing1	0.70	0.0081	**0.0062**	0.0103	**0.0048**	0.0147	**0.0090**
conund.	0.79	0.0002	**0.0001**	0.0003	**0.0000**	0.0006	**0.0009**
beowulf	0.82	0.0058	**0.0039**	0.0063	**0.0019**	0.0151	**0.0057**
gremlin	0.71	0.0038	**0.0023**	0.0034	**0.0011**	0.0032	**0.0001**
kongo	0.69	0.0001	**0.0001**	0.0001	**0.0001**	0.0004	**0.0008**

Table 4. Variance of Original Series and 5 Minute Averages (in bold face)

Host Name	Load Average	vmstat	NWS Hybrid
thing2	**2.4%** (1.2%)	* **1.7%** (4.9%)	* **1.3%** (1.8%)
thing1	**4.9%** (1.7%)	**3.5%** (3.1%)	**3.9%** (2.8%)
conundrum	**0.7%** (0.4%)	**0.2%** (0.2%)	**0.3%** (0.2%)
beowulf	**3.4%** (1.8%)	* **2.3%** (3.1%)	**4.5%** (3.5%)
gremlin	**2.6%** (1.0%)	* **1.2%** (2.1%)	* **1.3%** (2.0%)
kongo	**0.2%** (0.1%)	**0.1%** (0.1%)	**0.2%** (0.1%)

Table 5. Mean Absolute One-step-ahead Prediction Errors for 5 Minutes Aggregated during a 24-hour, mid-week period. Unaggregated error, from Table 3, is parenthesized.

Note that the decrease in variance resulting from aggregation does not necessarily imply that the aggregated series is more predictable. Table 5 shows the mean absolute one-step-ahead prediction error (Equation 4) for each of the aggregated series using the NWS forecasting methodology. The one-step-ahead prediction for the aggregated series is typically less accurate than for the original series. For the cases denoted by a * in the table, however, the aggregated prediction is more accurate than the corresponding one-step-ahead, 10 second, prediction. We hypothesize that smoothing may be more effective for certain time frames (aggregation levels) than for others. The prediction error, therefore, may improve for these aggregation levels, and the smoothed series may be predicted more accurately. This hypothesis supports the similar observations made in [10] and [17] regarding the smoothness of aggregated series. In general, however, the improvement should be small and there is no trend as a function of aggregation level that we can detect.

To gauge the true forecasting error in the aggregated case, we examine the difference between the forecasted value and the the value observed by a *test process*. This new *test process* runs for 5 minutes at a time, every 60 minutes. The new forecasted value is derived from the averaged series X^{30}, which is calculated as $X_t^{30} = \frac{X_{30t-30+1} \dots X_{30t}}{30}$ for t varying from 1 to the number of entries in each trace, counting by 30. Since we obtain a measurement every 10 seconds, each X^{30} value is an average the measurements taken over five minutes. We then consider a one-step-ahead forecast of each X^{30} value as a prediction of the average availability during the succeeding five minute period. To

Host Name	Load Average	vmstat	NWS Hybrid
thing2	6.6%	5.3%	6.5%
thing1	5.6%	5.2%	6.7%
conundrum	3.0%	7.4%	10.1%
beowulf	6.0%	11.4%	11.1%
gremlin	4.3%	2.9%	8.3%
kongo	2.1%	1.9%	28.5%

Table 6. Mean True Forecasting Errors for 5 Minute Average CPU Availability

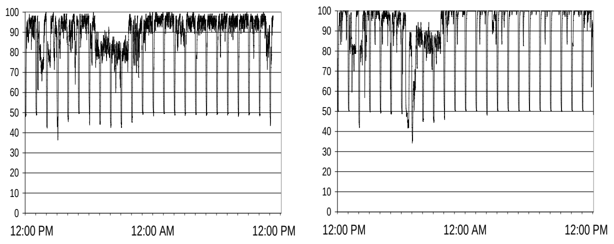

Figure 4. 5 Minute Aggregated CPU Availability using Unix Load average from *thing1* (left) and *thing2* (right).

calculate the aggregate true forecasting error, shown in Table 6, we again use Equation 5, where t now represents 5 minutes. Again, a problem with the bias value used by the NWS hybrid sensor causes the large discrepancy on kongo.

Note that we execute the *test process* only once every 60 minutes to prevent the load induced by them from driving away potential contention. We feared that a more frequent execution of the *test process* might cause other users to abandon the hosts we were monitoring in favor of more lightly loaded alternatives.

Figure 4 shows the Unix load average CPU availability measurements during the 24-hour experimental period for hosts *thing1* and *thing2* as example traces. From this figure, it is clear that the systems experienced load during the test period (the apparent periodic signal results from the intrusiveness of the 5 minute *test process*). Despite the variance in each series, however, the average true forecasting error for a program that occupies the CPU for five minutes is between 5% and 6%.

4 Conclusions and future work

From the data presented in Sections 2 and 3 we make the following observations about the workstations and computational servers in the UCSD Computer Science and Engineering department during the experimental period:

- Using conventional, non-privileged, Unix utilities the greatest source of error in making a one-step-ahead prediction of CPU availability comes from the process of *measuring* the availability of the CPU and not from *predicting* what the next measurement value will be.
- Traces of CPU availability exhibit long-range autocorrelation structures and are potentially self-similar.
- Short-term (10 seconds) and medium-term (5 minute) predictions of CPU availability (including all forms of error) can be obtained that are, on the average, between 5% and 12%.

In the context of process scheduling, these results are encouraging and somewhat surprising. Often, researchers assume that CPU loads vary to such a degree as to make dynamic scheduling difficult or impossible. While we certainly observe variation which is sometimes large, the series that are generated are fairly predictable. Moreover, the measurement and forecast error combined are small enough so that effective scheduling is possible. In [24], for example, we used considerably less accurate measurements to achieve performance gains that were better than 100% in some cases.

Another important realization is that long-range autocorrelation and self-similarity do not necessarily imply short-term unpredictability. Much of the previous excellent analysis work has focused on identifying and explaining self-similar performance behavior, particularly of networks. In these domains, short-term predictability may not be as important as predicting the long-term. While it is true that long-term predictions would be useful in a process scheduling context, short-term predictability also has utility.

Lastly, our observations coincide with those made recently by Dinda and O'Halloran [10] with respect to observed autocorrelation structure and Unix load average measurements. This is fortuitous since several large-scale metacomputing systems [18, 12, 26] use Unix load average to perceive system load. We extend this previous work by attempting to quantify the measurement error inherent in using load average as a measure of CPU availability, and by quantifying the effectiveness of the current NWS forecasting techniques on this type of data.

In future studies, we wish to expand the types of resources we consider to shared-memory multiprocessors, and collections of workstations that are combined using specialized networks (e.g. the Berkeley NOW [9]). We will also extend our set of experimental subjects to include workstations and computational servers in different production environments.

References

[1] O. Arndt, B. Freisleben, T. Kielmann, and F. Thilo. Scheduling parallel applications in networks of mixed uniprocessor/multiprocessor workstations. In *Proceedings of ISCA*

11th Conference on Parallel and Distributed Computing, September 1998.

[2] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing 1996*, 1996.

[3] M. Bernan, R. Serman, and M. Taqqu. Long-range dependence in variable-bit-rate video traffi c. *IEEE Transactions on Communications*, March 1995.

[4] S. Burnett and S. Fitzgerald. Metacomputing supports large-scale distributed simulations, 1998. available from http://www.cacr.caltech.edu/sfexpress/sc98.html.

[5] A. Cambel. *Applied Chaos Theory*. Academic Press, 1993.

[6] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. Hpjava: Data parallel extensions to java. In *ACM Workshop on Java for High-Performance Network Computing*, 1998.

[7] H. Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. In *Proc. of Supercomputing'96, Pittsburgh*. Department of Computer Science, University of Tennessee, Knoxville, 1996.

[8] M. Crovella and A. Bestavros. Self-similarity in world wide web traffi c: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5, December 1997.

[9] D. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, and F. Wong. Parallel computing on the berkeley now. In *9th Joint Symposium on Parallel Processing*, 1997.

[10] P. Dinda and D. O'Halloran. The statistical properties of host load. In *to appear in the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98) and CMU Tech. report CMU-CS-98-143*, 1998.

[11] S. Figueira and F. Berman. Modeling the effects of contention on the performance of heterogeneous applications. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, August 1997.

[12] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1997. to appear.

[13] I. Foster and C. Kesselman, editors. *The Grid – Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[14] B. Ganguly, G. Bryan, M. Norman, and A. Chien. Exploring structured mesh refi nement (samr) methods with the illinois concert system. In *Proceedings of SIAM conference on Parallel Processing*, March 1997.

[15] J. Gehrinf and A. Reinfeld. Mars - a framework for minimizing the job execution time in a metacomputing environment. *Proceedings of Future general Computer Systems*, 1996.

[16] C. Granger and P. Newbold. *Forecasting Economic Time Series*. Academic Press, 1986.

[17] S. Gribble, S. Manku, D. Roselli, and E. Brewer. Self-similarity in fi le systems. available from http://www.cs.berkeley.edu/˜gribble.

[18] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds. Legion: The next logical step towrd a nationwide virtual computer. Technical Report CS-94-21, University of Virginia, 1994.

[19] R. Haddad and T. Parsons. *Digital Signal Processing: Theory, Applications, and Hardware*. Computer Science Press, 1991.

[20] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of ethernet traffi c. *IEEE/ACM Transactions on Networking*, February 1994.

[21] B. B. Mandelbrot and M. S. Taqqu. Robust R/S Analysis of Long-run serial Correlation. In *Proceedings of the 42nd Session of the ISI*, volume 48, pages 69–99, 1979.

[22] K. Park, G. Kim, and M. Crovella. On the effect of traffi c self-similarity on network performance. In *Proceedings of the SPIE International Conference on Performance and Control of Network Systems*, November 1997.

[23] J. Schopf and F. Berman. Performance prediction in production environments. In *Proceedings of IPPS/SPDP 1998*, 1998. available from http://www.cs.ucsd.edu/users/jenny/TechReports/ipps98.ps.

[24] N. Spring and R. Wolski. Application level scheduling: Gene sequence library comparison. In *Proceedings of ACM International Conference on Supercomputing*, July 1998.

[25] T. Sterling, D. Becker, and D. Savarese. Beowulf: A parallel workstation for scientifi c computation. In *Proceedings of the International Conference on Parallel Processing*, 1995.

[26] T. Tannenbaum and M. Litzkow. The condor distributed processing system. *Dr. Dobbs Journal*, February 1995.

[27] J. Weissman and A. Grimshaw. A framework for partitioning parallel computations in heterogeneous env ironments. *Concurrency: Practice and Experience*, 7(5), August 1995.

[28] W. Willinger, M. Taqqu, R. Sherman, and D. Wilson. Self-similarity through high-variability: statistical analysis of ethernet lan traffi c at the source level. In *SIGCOMM'95 Conference on Communication Architectures, Protocols, and Applications*, 1995.

[29] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1998. also available from http://www.cs.ucsd.edu/users/rich/publications.html.

[30] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems (to appear)*, 1998.

[31] R. Wolski, N. Spring, and C. Peterson. Implementing a performance forecasting system for metacomputing: The network weather service. In *Proceedings of Supercomputing 1997*, November 1997.