A New Metric for Robustness with Application to Job Scheduling

Darin England, Jon Weissman, and Jayashree Sadagopan
Department of Computer Science and Engineering
University of Minnesota, Twin Cities
{england,jon,sjaya}@cs.umn.edu

Abstract

Scheduling strategies for parallel and distributed computing have mostly been oriented toward performance, while striving to achieve some notion of fairness. With the increase in size, complexity, and heterogeneity of today's computing environments, we argue that, in addition to performance metrics, scheduling algorithms should be designed for robustness. That is, they should have the ability to maintain performance under a wide variety of operating conditions. Although robustness is easy to define, there are no widely used metrics for this property. To this end, we present a methodology for characterizing and measuring the robustness of a system to a specific disturbance. The methodology is easily applied to many types of computing systems and it does not require sophisticated mathematical models. To illustrate its use, we show three applications of our technique to job scheduling; one supporting a previous result with respect to backfilling, one examining overload control in a streaming video server, and one comparing two different scheduling strategies for a distributed network service. The last example also demonstrates how consideration of robustness leads to better system design as we were able to devise a new and effective scheduling heuristic.

1. Introduction

As its name suggests, high-performance computing pertains to the execution of large, scientific codes. Users want results quickly, and job schedulers have evolved to accommodate them. There is no disputing the fact that performance in job scheduling is paramount. Techniques such as backfilling [15] and gang scheduling [6] are designed to get users' jobs out of the queue and running as fast as possible. At the same time, we note that computing systems are becoming larger, more distributed, and more heterogeneous. Huge data sets and large problem instances lead to very long-running execution times that are inherently unpredictable. Communication latencies contribute to more uncertainty. Given these circumstances, we argue that it is increasingly important for computing systems to be robust. That is, systems should be able to *maintain* performance despite various uncertainties in the operating environment. It is important to emphasize that these uncertainties are outside the system's control. Consider the case of a flash crowd to a news service when disaster strikes, a distributed denial-of-service attack, or the researcher who submits a 1 million by 1 million dense matrix computation. These events occur with enough frequency to warrant consideration in system design. This work presents a new way to measure how well a system responds to such events.

We have an intuitive notion of robustness, but let us be a bit more precise by employing the following definition that we have adapted from the works of Ali et. al. [1] and Carlson and Doyle [3].

> "Robustness is the persistence of certain specified system features despite the presence of perturbations in the system's environment."

Indeed, robustness is a desirable property, but how can we evaluate it? We seek a way to characterize the robustness of a system and obtain a corresponding quantitative metric. In this article we introduce a methodology for achieving this goal. In view of our definition above, we will be quite specific with respect to the performance features that we want the system to maintain and with respect to the type of disturbance (or perturbation) that is applied. What is unique (and in our opinion quite necessary) about our methodology is that it takes into account the full distribution of performance observations, as opposed to just using average measurements. This is important for two reasons: (1) The uncertainty present in todays' computing systems effect large deviations in performance observations, and (2) The presence of large deviations (or outliers) can greatly affect the average of a set of observations.

The use of stochastic information to guide system design is not new [8–10, 14, 17, 18], and in the next section we discuss the relevant related work. Results from some of the prior work are restricted in the sense that they exploit the Markov property in order to make the mathematics tractable. In reality, inter-arrival times and execution times may not be exponentially distributed. A nice property of our methodology is that it is straightforward to apply regardless of the underlying probability distribution from which the observations are taken. Constructing the empirical distribution functions, an easy task, is at the heart of the technique.

The rest of the paper is organized as follows. In section 2 we discuss related work in the areas of robustness, performance evaluation, and scheduling. In section 3 we present our methodology for characterizing and measuring the robustness of a system to a specific disturbance. Section 4 presents the application of our methodology to three different problems: backfilling jobs on a partitionable supercomputer, serving requests for a streaming video service, and routing requests for a network service to distributed computing nodes. We provide concluding remarks in section 5.

2. Related Work

2.1. System Design

Gribble [7] makes a strong argument for considering robustness in the design of complex systems. They provide an example of a distributed application which performs well under normal operating circumstances. However, unforeseen interactions among the loosely coupled components lead to poor performance and even corruption of the system. Gribble's work gives general suggestions for how to incorporate robustness into the design of complex systems, including admission control, system introspection, and adaptive control. However, they do not provide a mechanism for measuring the robustness of the resulting system.

2.2. Measuring Robustness

Most closely related to our work, Ali et. al. [1] present a metric for the robustness of a resource allocation. Their procedure consists of identifying the performance features that cause system degradation and quantitatively specifying the bounds on those features. Then the unpredictable elements (called perturbation parameters) and their impact on the system are assessed by finding a mathematical relationship between the perturbation parameters and the system performance features. The robustness measurement of a system is determined by finding the smallest variation in the perturbation parameters that cause degradation of the system to violate the performance bounds. In general, the problem of finding the smallest acceptable variation, called the robustness radius, is cast as a mathematical optimization problem. The authors of this work do a good job of defining the robustness metric and they present the application of the metric to three example systems. However, the specification of the bounds on the performance features seems somewhat arbitrary and may only apply to systems for which acceptable performance is well-defined. Our metric for robustness is generally applicable and is considerably easier to compute.

Although not advertised as a way to measure robustness, another alternative view to traditional performance measurements is *performability*. The idea behind performability is that measurements of either performance or availability alone do not indicate the real usefulness of a system, especially for multiprocessor and distributed systems. When used in a modeling context, performance measurements that ignore failure and repair overestimate the capacity of a system. Similarly, availability measurements do not consider lower levels of performance and hence underestimate the capacity of a system. Performability, as described in [14, 18], measures the amount of useful work that a system can complete as it undergoes various levels of degradation. In this way the work of Smith et. al. [18] could be viewed as a way to measure robustness; however, their method requires that the system be modelled as a Markov Reward Model. Then, transient and steady-state methods are applied to view the accumulated performance. The modeling and solution techniques require significant work and there exist systems for which the assumptions of the Markov property do not hold. Our robustness metric can be used in any analytical model for which the CDFs of the disturbance and performance can be calculated or estimated. It can also be applied to the empirical CDFs of actual system data.

2.3. Scheduling

With respect to scheduling, Schopf and Berman [17] examine the use of the probability distribution of an application's execution time in order to achieve good performance on multi-user clusters. Their focus is on data parallel applications where the amount of computation performed is directly related to how much data is assigned to a processor. The scheduling policy tries to assign data to each processor so that all processors finish at about the same time, a policy known as time balancing. However, execution times vary because the computing resources are shared, and as a consequence, some machines are more heavily loaded than others. The main idea is to assign more data to machines with smaller variability in performance. The reasoning is such: suppose that a machine is fast but exhibits high variability in performance. More data will be assigned to this machine because it is fast; however, its high variability will have a large impact on the overall application execution time. Through experiments with a distributed application, Schopf and Berman show that it is possible to obtain better performance and predictability when using a stochastic scheduling approach.

Several researchers have found that the execution times of many computational tasks follow a heavy-tailed probability distribution [4, 8, 10, 13]. In [9], Harchol-Balter et. al. present a policy for assigning distributed servers to such computational tasks. Their policy, called SITA-E (Size Interval Task Assignment with Equal Load), balances the workload by computing ranges for execution times and then assigning tasks whose execution times fall within the same range to the same server. Thus, a priori knowledge of the execution times is required for the assignment. They show through both analysis and simulation that their assignment policy performs well when the variability of the execution times is high, i.e. when the execution times follow a heavy-tailed distribution. The performance metric of interest in [9] is mean waiting time. Our work differs in that we are concerned with the maintenance of a performance metric as variability is introduced into the workload. We also make the important distinction that although we assume knowledge of the distribution of such variability, e.g. execution times, we do not require a priori knowledge of the execution times of individual tasks.

3. Measuring Robustness

3.1. Measuring Performance Degradation

A property such as robustness is easy to define, but difficult to measure in a quantitative manner. Perhaps this is why there are no commonly used robustness metrics. To begin we first note that robustness is not performance. This is not to say that the two properties are mutually exclusive, but rather they have different meanings in the context of computing systems. It is possible for a system to perform quite well under normal operating circumstances and yet exhibit catastrophic failure when subjected to slight disturbances [3]. Alternatively, if a system's operating environment is known to be highly variable, one might prefer a system with less average performance, but robust to a wide variety of operating conditions. Figure 1 shows an overview of our situation. We observe a specific performance metric such as packets per second, waiting time, or utilization under normal system operation and with a disturbance applied. The idea is to measure the amount of performance degradation relative to the size of the disturbance. Typical performance metrics for some representative computing systems are shown in Table 1. We list these metrics in order to show that our methodology for measuring robustness applies to a wide variety of systems.

Consider the cumulative distribution function (CDF) of the performance metric. Given a set of performance observations, the CDF is simply the proportion of observations

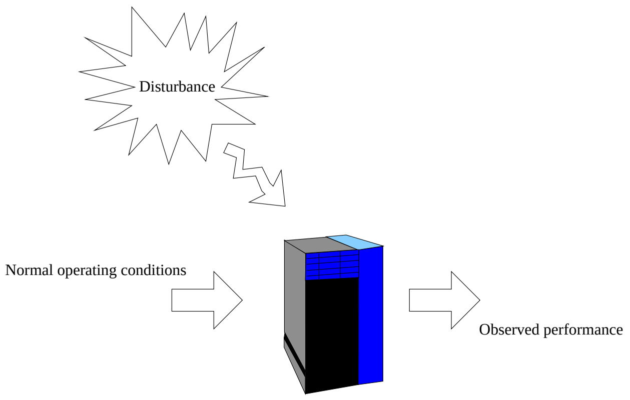

Figure 1. Measuring Robustness

System	Typical performance metric
Web server	Maximum number simultaneous connections
FTP server	Bytes per second
Media server	Delay/jitter
Network router	Packets per second
Supercomputer/cluster	Utilization

Table 1. System Performance Metrics

less than or equal to a certain value. Let X be an observation of the system's performance. Let $F(x) = P(X \leq x)$ be the CDF of performance under normal operating conditions and let $F^*(x) = P(X \leq x)$ be the CDF of performance with perturbations applied. If the the system is robust with respect to the type of perturbation applied, then $F^*(x)$ should be very similar to $F(x)$. The closer the two functions agree, the more robust the system. However we expect the perturbations to have some effect on performance, and this is what we measure. In the case where small observations of the performance metric are good, e.g. waiting time or communication delay, we expect $F(x)$ to be greater than $F^*(x)$. That is, for a given value of the performance metric, x, we expect a greater proportion of the observations to be less than x when the system is in normal operation (see Figure 2). To measure the degradation in performance, we measure the maximum distance between the two functions, $F(x)$ and $F^*(x)$, as shown in Figure 2. In fact this distance, which we denote as δ, is the well-known Kolmogorov-Smirnov (K-S) statistic, which is computed as

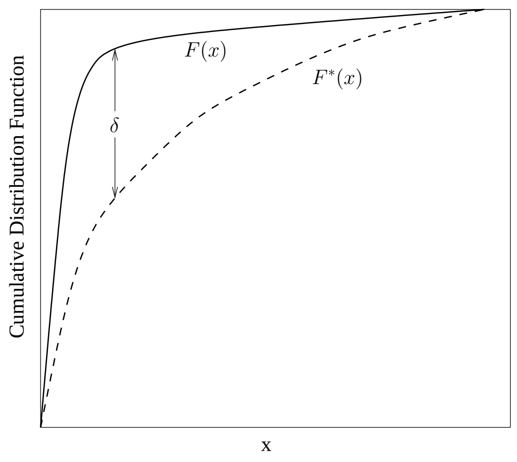

Figure 2. The Kolmogorov-Smirnov Statistic δ

follows[1]

$$\delta = \sup_{-\infty<x<\infty} F(x) - F^*(x).$$

Normally, the K-S statistic is employed to either accept or reject the hypothesis that two sets of observations come from the same underlying probability distribution. Given the distance δ and the sample sizes, one may compute the probability that a distance of δ would actually occur if the two samples really came from the same distribution. Thus the typical use of the K-S statistic is to make a binary decision: either we accept or reject the hypothesis that the two sets of observations are statistically equivalent. For our purposes we will make further use of δ by using its magnitude as an indication of the amount of performance degradation. A nice property of δ is its invariance to the scale used for the x-axis. (Recall that the CDF is the proportion of observations less than or equal to a given value x.) For example, δ will be the same whether we use x or $\log x$ as the scale for the abscissa. We also note that δ will always be in the range $[0, 1]$.

However the main reason we use δ to measure the amount of performance degradation is because it takes into account the entire distribution of performance observations, and not just averaged measurements. This is important because, as mentioned in the Introduction, averages are greatly affected by the presence of outliers. δ, on the other hand, is not. Indeed, a robust statistic is one that is not affected by the presence of outlying data points. This is robustness on another level; we are using a robust statistic to measure the robustness of a system.

1 The statistic is defined using the supremum instead of the more familiar maximum. This is due to the fact that the maximum is not well-defined for some sets of numbers.

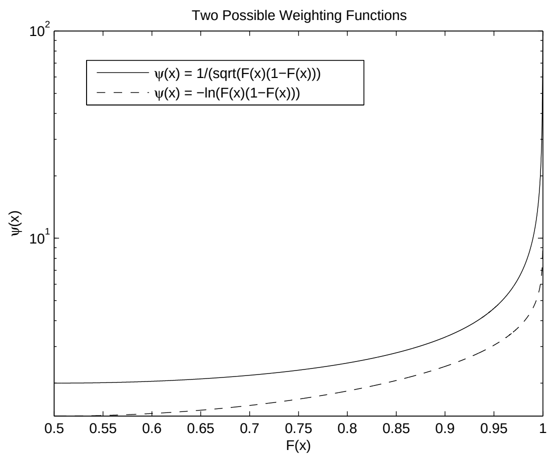

Figure 3. Weighting Functions for the K-S Statistic δ

3.2. Adjusting the K-S Statistic δ

The K-S statistic δ has its *own* probability distribution. Although any particular observed value of δ is invariant with respect to the x-axis, its probability distribution is not. It is known that δ is most sensitive at values near the median, where $F(x) = .5$, and that δ is underestimated near the tails of the CDF [2]. This is particularly important for our work for two reasons. The first reason is that values at the right tail of the CDF represent large values of the performance metric, which represent a larger impact to the system. The second reason is that many real-world computing systems exhibit the heavy-tailed phenomenon. This is the notion that a small percentage of the observations constitute the bulk of the total disturbance or performance. Well-known examples include the distribution of file sizes requested by HTTP and FTP clients, and the distribution of UNIX process lifetimes [10]. One way to compensate for this underestimation is to weight δ as a function of $F(x)$, with the weight increasing toward the right tail. Denote this weighting function by $\psi(F(x)) = \psi(x)$. Then the adjusted K-S statistic is given by

$$\delta_w = \sup_{-\infty<x<\infty} (F(x) - F^*(x))\psi(x).$$

We considered two different weighting functions, both of which are based on the fact that the quantity $F(x)(1-F(x))$ is greatest at $F(x) = 0.5$. The magnitudes of these two functions, $\psi(x) = 1/\sqrt{F(x)(1-F(x))}$, and $\psi(x) = -\ln(F(x)(1-F(x)))$, are shown in Figure 3. (Note the logarithmic scale for the ordinate.) Choosing a proper weighting function is somewhat subjective – we want to compen-

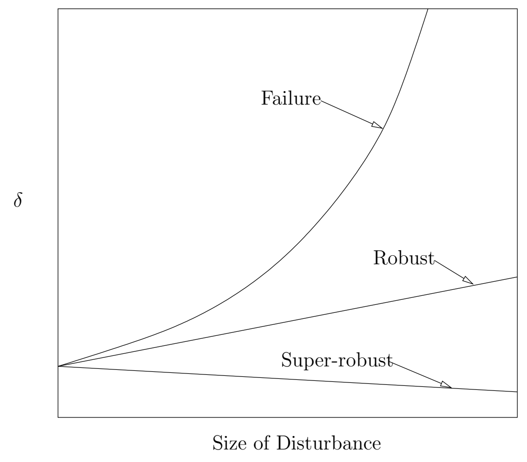

Figure 4. Characterizing the Robustness of a System

sate δ for underestimation at the right tail, but not by too much. After some experimentation, we chose the function $\psi(x) = -\ln(F(x)(1 - F(x)))$ to apply in the computation of δ_w. We mention here that we only apply the weight when δ occurs to the right of the median since these values have the largest impact on the system. Thus if δ occurs to the left of the median, then $\psi(x) = 1$.

3.3. Characterizing Robustness

So far we have described our technique for measuring performance degradation. However the measurement, δ_w, corresponds to just one level of disturbance. A robust system has the connotation of being able to withstand various levels of disturbance. With this in mind, we can characterize the robustness of a system by measuring δ_w as a function of the size of the applied perturbations. If the system is robust, then δ_w will show a graceful increase as the size of the disturbance increases. Systems that are sensitive to the applied disturbance will exhibit large, possibly nonlinear increases in δ_w. This idea is illustrated in Figure 4. It is also possible for δ_w to decrease as the disturbance is increased, in which case the term “disturbance” is really a misnomer since the applied perturbations caused the system to perform better, i.e. $F(x) < F^*(x)$ for all values of x. When this happens we could say the system is super-robust. Although it seems counter-intuitive that perturbations outside the system’s control would be beneficial, our first example in the next section exhibits this phenomenon.

4. Application to Job Scheduling

In this section we apply our methodology for measuring and characterizing the robustness of a system to a specific disturbance. The first example, backfilling jobs on parallel machines, uses trace data from a real system. The second example concerns overload control in a streaming media server, for which the data was collected from an actual implementation. Finally, the last example is more theoretical in nature. Based on data from a simulation, it pertains to routing requests in a distributed network service.

4.1. Backfilling on Supercomputers

The basic premise of fairness in job scheduling on parallel machines is the notion of first-come-first-served. However a job that requires a large number of processors may delay many smaller jobs while it waits for its required number of processors to become free. Backfilling was devised to remedy this situation. Using this technique, a job may move ahead of other jobs in the queue and begin processing as long as it does not delay the *first* job in the queue[2]. This form of backfilling is commonly employed in parallel job schedulers such as Maui and IBM Load-leveller. In order to work, the algorithm requires that users submit run time estimates with their job submissions. It is well-known that these estimates are notoriously inaccurate [12, 15]. The run time estimates are generally inflated because users do not want the system to kill their jobs in case the actual run time exceeds the estimate. It has also been observed that users simply do not take the time to provide good estimates [12]. An interesting feature of backfilling that was reported in [15] is that over-estimation of run time estimates actually benefits the system. The reason is that the algorithm can find more “holes” in the schedule to move jobs forward when the estimated run times are large. By ”benefit the system” we mean that metrics such as average queue time and utilization are improved.

As a first application of our methodology, We tested the robustness of the backfilling algorithm to the inaccuracy of user run time estimates. The backfilling algorithm that we implemented is exactly the one described in [15] for aggressive backfilling. The parallel workload we used is from the San Diego Supercomputer Center Blue Horizon machine, which is available from [16]. This machine is a 144 node IBM SP with 8 processors per node. The log is for the period from April 2000 to January 2003. In this experiment, the performance metric in which we are interested is the distribution of queue times and the disturbance is the amount of

2 This is known as aggressive backfilling. Conservative backfilling, which is not widely used, requires that the job that moves ahead may not delay *any* job that was ahead of it in the queue

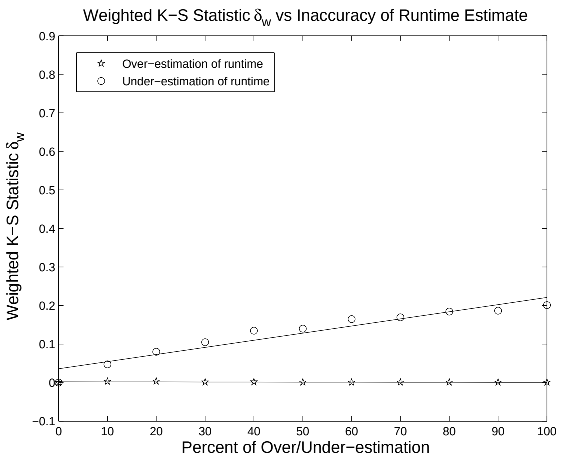

Figure 5. Robustness of Backfilling to Inaccuracy of User Runtime Estimates

inaccuracy in the user run time estimates. Based on the analysis presented in [15], we expected the system to be robust against over-estimation, but not against under-estimation. We tested both. First we ran the backfilling algorithm on the workload log using the exact run times as the estimates. Then, we made further runs increasing the run time estimates in increments of 10% up to 100%. To be clear, the run time of every job was consistently over-estimated by a certain percentage of its actual run time for the purpose of backfilling. We ran the experiment for under-estimation of run times as well. The robustness of the algorithm to over-estimation and to under-estimation is presented in Figure 5.

As expected, the backfilling algorithm performs well when run times are over-estimated. In fact, the slope of the (least-squares) fitted line is negative, which indicates that over-estimation of run times actually benefits the distribution of queue times (i.e. the system is super-robust). We also found that backfilling is very robust to under-estimation, although the performance does degrade as the amount of under-estimation increases. Note that the inaccuracy, under- or over-estimation, applies to the job being considered for backfilling as well as to the jobs that are in progress. Thus the scheduler does not know when a job will actually finish until it does so. We also mention that our simulation did not kill any jobs due to under-estimation of run time. This example illustrates how our methodology can be applied to measure the robustness of a system to a specific disturbance. Our results support those of a previous work and also indicate that backfilling is robust to inaccurate run time estimates in general. It seems that with respect to backfilling what the scheduler doesn't know won't hurt it.

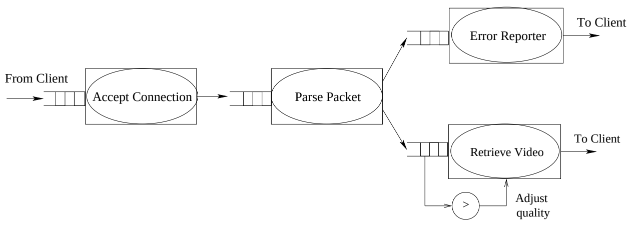

Figure 6. SEDA-based Video Server

4.2. Streaming Video Server

The second application of our methodology is to measure the robustness of a streaming video server to increased load. Used in a separate project to study adaptive overload control, this particular server is based on the SEDA [19] architecture. SEDA (staged event-driven architecture) is a framework designed for high concurrency and robustness to large variations in load [19]. An application built on SEDA consists of a series of stages, each stage having its own queue. This allows for fine-grained control of separate sections of an application, including resource allocation and error reporting. The stages for our video server are shown in Figure 6.

User satisfaction with a streaming video service depends on jitter, or delay, between frames. In order to avoid jitter, the server limits the number of threads that can run simultaneously in the video retrieval stage. This can cause a queue of requests to build up quickly when the server is experiencing overload or when the requested videos are large. The end-to-end delay seen by the user can be controlled by limiting the queue delay and the video retrieval duration, assuming a relatively constant transmission time. This is the motivation behind the overload control mechanism. The server stores pre-encoded videos of varying quality. High quality videos take longer to transmit and consume more server resources than low quality videos. Based on the current system load, the server decides which quality of video to deliver to the client. The system load is measured by the number of requests in the queue at the video retrieval stage (see Figure 6). This technique has the advantage that it reduces both the queue time and the video retrieval time. The compromise is the quality of video delivered to the client when the system is experiencing heavy load. Hence the system is better suited for applications where the user values quick response over video quality, for example a news on-demand video server.

In this experiment, we measure the robustness of the system as the load on the server is increased. The performance feature that we wish to maintain is response time, the elapsed time between the client's request and the arrival of the first video byte. Although the system is capable of dropping requests at extremely high loads, the threshold for doing so is set high enough to ensure that all requests are processed. Both the video objects and the requests for them (the workload) were generated using GISMO [11], a synthetic streaming media load generator. The details of the load generator are presented in [11]. Here we briefly describe the relevant parameters for our experiments. Video file sizes follow a lognormal distribution, which is somewhat similar to a normal distribution, but is skewed to the right. This allows the probability of having larger file sizes than are possible under a normal distribution with the same mean. The popularity of any particular video is determined by a Zipf-like distribution. This is a power-law distribution wherein a few videos are extremely popular while most videos are rarely requested. The interarrival times of individual requests for the video service are generated from a truncated Pareto distribution, which is also a power-law distribution. The purpose of generating the interarrival times in this manner is to simulate periods of heavy activity mixed with periods of relatively light activity.

There are two versions of the server: one version contains the overload control mechanism, and the other version does not. The server and client are executed on separate machines in a local area network. To inject load on the server, the client generates requests for the video service for a duration of 90 seconds, increasing the total number of requests for each experiment. Figure 7 shows the robustness of each version of the video server to increased load. When the load on the server is low, both systems give comparable performance. As the number of simultaneous requests increases, the queue begins to build up. The video server with the overload control mechanism adapts to this situation by degrading the quality of video being served. The gradual increase in the weighted K-S statistic indicates a more graceful degradation in response time than the version of the server that does not use overload control. Of course this result is expected since the overload control mechanism was designed to handle large variations in load. But what we want to show is that the application of our methodology provides a quantitative way to measure the robustness of the overload control mechanism.

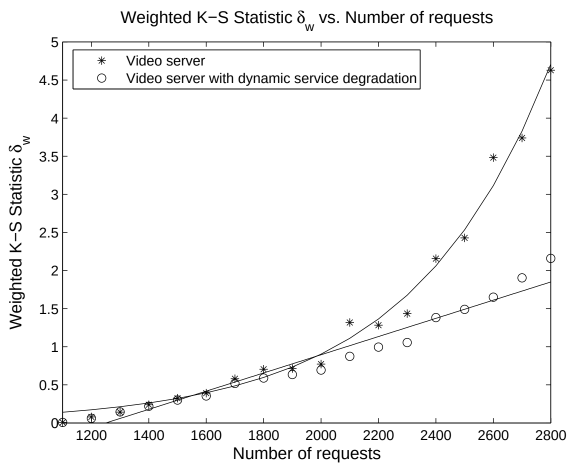

Figure 7. Robustness of Video Server to Increased Load

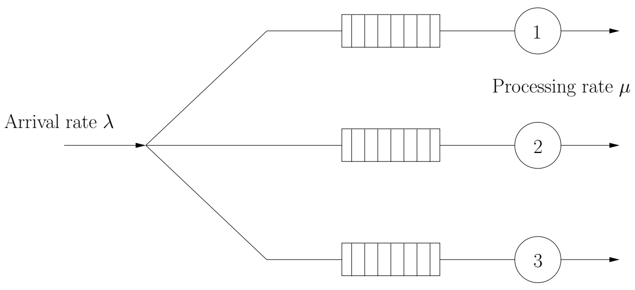

Figure 8. Network Service Hosted on Three Resources

4.3. Distributed Network Service

In this section we apply the robustness metric to a simulated network service. Requests for the service arrive according to a Poisson process at the rate of 2 requests per minute. The service is hosted on three computing nodes and requests are routed to one of the three nodes according to a scheduling policy. Each node contains its own separate queue to hold requests before processing (see Figure 8) and the performance feature of interest is the amount of time requests spend in the queue. Naturally, it is desirable for the queue times to be as small as possible. Suppose that under normal operating conditions the execution times of the requests are exponentially distributed with a mean of 4 minutes. However, the computing nodes are time-shared with other applications and during periods of heavy load the execution times follow a heavy-tailed Weibull distribution with shape parameter $\alpha = .42$ and scale parameter $\beta = 1.35$, which also has a mean of 4 minutes, but much higher variance. The nature of this distribution is such that most execution times are very short; however, there are a few execution times that are very long in duration. We would like to know which of two scheduling policies is more robust to long-running execution times. The two polices to be considered are:

Shortest Queue Route incoming requests to the node with the least number of queued requests.

Alternate Method Route requests to the node with the most recently started execution.

It has been shown that when the execution times are exponentially distributed the shortest queue policy is optimal for this type of distributed queueing system [5]. This is not necessarily the case in the presence of very long execution times (i.e. as the tail of the distribution becomes heavier). The philosophy behind the alternate scheduling policy centers on recognition of the decreasing failure rate of the Weibull distribution (with $\alpha < 1$). Under this distribution, the longer a request has been in execution the more likely it is to execute even longer [8]. Thus the policy attempts to avoid sending requests to nodes that are already processing long-running requests. We use the term "attempt" because individual execution times are not known a priori.

To test the robustness of the two scheduling policies to the presence of very long executions we simulated the system for different values the Weibull shape parameter α. Holding the scale parameter β constant while decreasing α increases the right tail of the Weibull distribution. In this way we simulated increasing levels of disturbance in the form of execution times of very long duration. Each of the two scheduling policies was simulated for 10,000 requests for each level of α. The robustness characteristics of both scheduling policies are shown in Figure 9. Naturally, the performance of both policies degrade with the increasing presence of extremely long execution times, but the alternate method is clearly more robust as evidenced by the linear increase in δ_w. Compare this to the steep, nonlinear increase in δ_w for the shortest queue policy. The fitted curve is a polynomial of degree 4, which seemed to best describe the trend.

Let us explicate the conclusions from this example. We cannot simply state that the alternate policy is more robust than shortest queue in any case. What we can state, with a quantitative measurement, is that the alternate policy is more robust than shortest queue *in the presence of long-running execution times*. We view this detailed nature of the metric as an attribute, not a limitation. Indeed, claiming that a system is robust without stating the type of disturbance it can withstand is so generic as to be almost useless. This example further shows how consideration of robustness in system design can lead to new strategies. Although our alternate method is a heuristic, and thus not provably optimal, we have shown that it performs better than shortest queue as the disturbance increases, and performs as good as shortest queue when the level of disturbance is low.

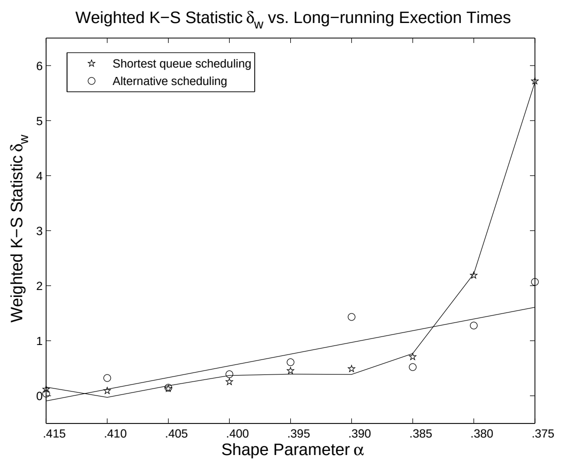

Figure 9. Robustness of Scheduling Strategy to Long-running Executions

5. Conclusion

Robustness is a desirable property for computing systems. It is not the same thing as performance, but rather, it measures the ability of a system to maintain performance in the presence of adverse operating conditions. This notion is important for today's large, complex high-performance computing systems because they are increasingly subject to uncertainty in load, communication latency, and even resource availability. While acknowledging that performance is the namesake in high-performance computing, we have argued that it also important to consider robustness in system design. Robust systems are able to withstand disturbances and maintain performance features. Ideally, computing systems should achieve both performance and robustness.

We presented a new methodology to characterize and quantitatively measure the robustness of a system to a specific disturbance. The technique is easy to understand and easy to apply: measure the degradation in performance as reflected in the cumulative distribution functions and plot this measurement against increasing levels of the disturbance. In this way we consider the entire distribution of performance observations and we do not rely on averages that can be greatly affected by the presence of outlying data points, itself another level of robustness. We also presented three example applications of our methodology. The first example tested the robustness of the aggressive backfilling algorithm to the inaccuracy that is inherent in users' estimates of run time. Using trace data from the SDSC Blue Horizon machine we incrementally over-estimated (and under-estimated) user run times and then measured the resulting change in the distribution of queue times. Characterization of the system's robustness to inaccurate run time estimates showed that over-estimation actually improves queue times while underestimation degrades them, albeit only slightly. The second application was to assess the robustness of an overload control mechanism in a streaming media server. The server was implemented on the SEDA framework and the workload was created using the GISMO load generator. By employing the overload control mechanism the server was shown to be robust against very high load, albeit at the cost of degraded video quality as seen by the client.

The final example application of our methodology compared the robustness of two different scheduling policies to the presence of long-running execution times in a distributed network service. Again, the performance metric of interest was queue time. We showed that when the execution times follow a heavy-tailed probability distribution it is more robust to avoid routing requests to nodes that are (likely to be) processing long-running requests. Furthermore, taking robustness into consideration led to a new scheduling heuristic, one that performs much better than the traditional strategy as the disturbance grows and also performs well under normal operating conditions. The examples demonstrate the utility of our methodology. Characterizing and measuring the robustness of a system in this way is intuitive and easy to apply. The results can then be used as one measure of the overall usefulness of a system.

6. Acknowledgements

The authors would like to acknowledge the support of the National Science Foundation under grants CNS-0305641 and ITR-0325949, the Department of Energy's Office of Science under grant DE-FG02-03ER25554, and the Minnesota Supercomputing Institute for Digital Simulation and Advanced Computation and the Digital Technology Center at the University of Minnesota.

References

[1] S. Ali et al. Measuring the robustness of a resource allocation. *IEEE Transactions on Parallel and Distributed Systems*, 15(5), May 2004.

[2] T. W. Anderson and D. A. Darling. Asymptotic theory of certain "goodness of fit" criteria based on stochastic processes. *The Annals of Mathematical Statistics*, 23(2):193–212, June 1952.

[3] J. M. Carlson and J. Doyle. Highly optimized tolerance: Robustness and design in complex systems. *Physical Review Letters*, 84:2529–2532, 2000.

[4] H. Chen, C. Gomes, and B. Selman. Formal models of heavy-tailed behavior in combinatorial search. In *Proceedings of the Seventh International Conference on Principles and Practices of Constraint Programming*, 2001.

[5] A. Ephremides, P. Varaiya, and J. Walrand. A simple dynamic routing problem. *IEEE Transactions on Automatic Control*, AC-25(4):690–693, Aug. 1980.

[6] D. G. Feitelson. Packing schemes for gang scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162, pages 89–110. Springer-Verlag, 1996. Lecture Notes in Computer Science.

[7] S. D. Gribble. Robustness in complex systems. In *Proceedings IEEE Eighth Workshop on Hot Topics in Operating Systems*, pages 21–26, May 2001.

[8] M. Harchol-Balter. The effect of heavy-tailed job size distributions on computer system design. In *In Proceedings of ASA-IMS Conference on Applications of Heavy Tailed Distributions in Economics, Engineering and Statistics*, June 1999. Washington, DC.

[9] M. Harchol-Balter, M. E. Crovella, and C. D. Murta. On choosing a task assignment policy for a distributed server system. In *Proceedings of Performance Tools 1998*, pages 231–242, 1998. Springer-Verlag Lecture Notes in Computer Science.

[10] M. Harchol-Balter and A. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3), 1997.

[11] S. Jin and A. Bestavros. Gismo: a generator of internet streaming media objects and workloads. *SIGMETRICS Perform. Eval. Rev.*, 29(3):2–10, 2001.

[12] C. B. Lee et al. Are user runtime estimates inherently inaccurate? In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*. Springer Verlag, 2004. Lecture Notes in Computer Science.

[13] W. Leland and T. J. Ott. Load-balancing heuristics and process behavior. In *ACM SIGMETRICS Joint Conference Computer Performance Modelling, Measurement and Evaluation*, pages 54–69, 1986. Raleigh, NC.

[14] J. F. Meyer. On evaluating the performability of degradable computing systems. *IEEE Transactions on Computers*, C-29(8):720–731, Aug. 1980.

[15] A. Mu'alem and D. Feitelson. Utilization, predictability, workloads, and user run time estimates in scheduling the ibm sp2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6), June 2001.

[16] Parallel Workload Archive. The hebrew university of jerusalem, school of computer science and engineering. `www.cs.huji.ac.il/labs/parallel/workload`, 2005.

[17] J. M. Schopf and F. Berman. Stochastic scheduling. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, 1999.

[18] R. M. Smith, K. S. Trivedi, and A. V. Ramesh. Performability analysis: Measures, an algorithm, and a case study. *IEEE Transactions on Computers*, C-37(4):406–417, Apr. 1988.

[19] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *18th ACM Symposium on Operating Systems Principles, SOSP'01*, 2001.