Cluster Computing 7, 151–161, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

Evaluation of Task Assignment Policies for Supercomputing Servers: The Case for Load Unbalancing and Fairness

BIANCA SCHROEDER * and MOR HARCHOL-BALTER
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA

Abstract. While the MPP is still the most common architecture in supercomputer centers today, a simpler and cheaper machine configuration is appearing at many supercomputing sites. This alternative setup may be described simply as a *collection of multiprocessors* or a *distributed server system*. This collection of multiprocessors is fed by a single common stream of jobs, where each job is dispatched to exactly *one* of the multiprocessor machines for processing.

The biggest question which arises in such distributed server systems is what is a good rule for assigning jobs to host machines: i.e. what is a good *task assignment policy*. Many task assignment policies have been proposed, but not systematically evaluated under supercomputing workloads.

In this paper we start by comparing existing task assignment policies using a trace-driven simulation under supercomputing workloads. We validate our experiments by providing analytical proofs of the performance of each of these policies. These proofs also help provide much intuition. We find that while the performance of supercomputing servers varies widely with the task assignment policy, none of the above task assignment policies perform as well as we would like.

We observe that all policies proposed thus far aim to balance load among the hosts. We propose a policy which purposely *unbalances* load among the hosts, yet, counter-to-intuition, is also *fair* in that it achieves the same expected slowdown for all jobs – thus no jobs are biased against. We evaluate this policy again using both trace-driven simulation and analysis. We find that the performance of the load unbalancing policy is significantly better than the best of those policies which balance load.

Keywords: load balancing, task scheduling, performance evaluation, fairness

1. Introduction

This paper considers an increasingly popular machine configuration in supercomputer centers today, and addresses how best to schedule jobs within such a configuration.

1.1. Architectural model

The setup may be described simply as a *collection of multiprocessors* or a *distributed server system*. This collection of multiprocessors is fed by a single common stream of batch jobs, where each job is dispatched to exactly *one* of the multiprocessor machines for processing. Observe that we specifically do *not* use the word "cluster" because the word "cluster" in supercomputing today includes the situation where a single job might span more than one of these multiprocessors.

Figure 1 shows a very typical example of a distributed server system consisting of a dispatcher unit and 4 identical host machines. Each host machine consists of 8 processors and one shared memory. In practice the "dispatcher unit" may not exist and the clients themselves may decide which host machine they want to run their job on. Jobs which have been dispatched to a particular host are run on the host in FCFS (first-come-first-served) order. Typically, in the case of batch jobs, *exactly one* job at a time occupies each host machine (the job is designed to run on 8 processors), although it is sometimes possible to run a very small number of jobs simultaneously on a single host machine, if the total memory of the jobs fits within the host machine's memory space. The jobs are each *run-to-completion* (i.e., no preemption, no time-sharing). We will assume the above model throughout this paper, see section 2.2.

Run-to-completion is the common mode of operation in supercomputing environments for several reasons. First, the memory requirements of jobs tend to be huge, making it very expensive to swap out a job's memory [8]. Thus timesharing between jobs only makes sense if all the jobs being time-shared fit within the memory of the host, which is very un-

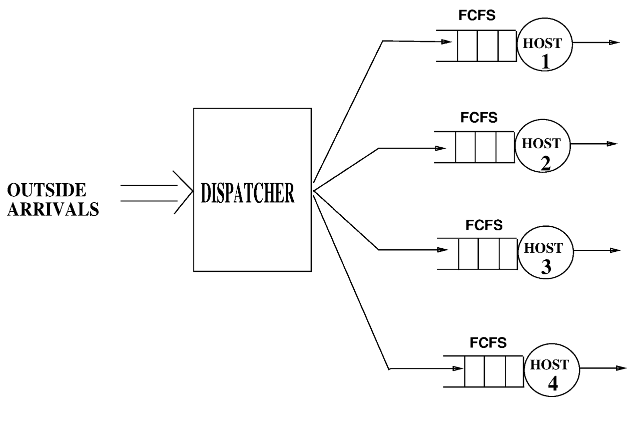

Figure 1. Illustration of a distributed server with 4 host machines, each of which is a multiprocessor.

* Correspoding author.
E-mail: bianca@cs.cmu.edu

likely. Also, many operating systems that enable timesharing for single-processor jobs, do not facilitate preemption among several processors in a coordinated fashion.

While the distributed server configuration described above is less flexible than an MPP, system administrators we spoke with at supercomputing centers favor distributed servers for their ease of administration, ease of scheduling, scalability, and price [3]. Also, the system administrators felt that distributed servers achieve better utilization of resources and make users happier since they are better able to predict when their job will get to run.

Examples of distributed server systems that fit the above description are the Xolas distributed server at the MIT Lab for Computer Science (LCS), which consists of eight 8-processor Ultra HPC 5000 SMPs [14], the Pleiades Alpha Cluster also at LCS, which consists of seven 4-processor Alpha 21164 machines [13], the Cray J90 distributed server at NASA Ames Research Lab, which consists of four 8-processor Cray J90 machines, the Cray J90 distributed server at the Pittsburgh Supercomputing Center (PSC), which consists of two 8-processor Cray J90 machines [20], and the Cray C90 distributed server at NASA Ames Research Lab, which consists of two 16-processor Cray C90 machines [19].

1.2. The task assignment problem

The main question in distributed servers such as those described above is "What is a good task assignment policy". A *task assignment policy* is a rule for assigning jobs (tasks) to host machines. Designing a distributed server system often boils down to choosing the "best" task assignment policy for the given model and user requirements. The question of which task assignment policy is "best" is an age-old question which still remains open for many models.

Our main performance goal, in choosing a task assignment policy, is to minimize *mean response time* and more importantly *mean slowdown*. A job's slowdown is its response time divided by its service requirement. (Response time denotes the time from when the job arrives at the system until the job completes service. Service requirement is just the CPU requirement – in our case this is the response time minus the queuing time.) All means are per-job averages. Mean slowdown is important because it is desirable that a job's response time be proportional to its processing requirement [1,6,12]. Users are likely to anticipate short delays for short jobs, and are likely to tolerate long delays for longer jobs. For lack of space, we have chosen to only show mean slowdown in the graphs in this paper, although we will also comment on mean response time. A second performance goal is *variance in slowdown*. The lower the variance, the more predictable the slowdown. A third performance goal is *fairness*. We adopt the following definition of fairness: All jobs, long or short, should experience the same expected slowdown. In particular, long jobs shouldn't be penalized – slowed down by a greater factor than are short jobs.[1]

Observe that for the architectural model we consider in this paper, memory usage is not an issue with respect to scheduling. Recall that in the above described distributed server system, hosts are identical and each job has exclusive access to a host machine and its memory. Thus a job's memory requirement is not a factor in scheduling. However CPU usage is very much an issue in scheduling.

Consider some task assignment policies commonly proposed for distributed server systems: In the `Random` task assignment policy, an incoming job is sent to Host i with probability $1/h$, where h is the number of hosts. This policy equalizes the expected number of jobs at each host. In `Round-Robin` task assignment, jobs are assigned to hosts in a cyclical fashion with the ith job being assigned to Host $i \bmod h$. This policy also equalizes the expected number of jobs at each host, and has slightly less variability in interarrival times than does `Random`. In `Shortest-Queue` task assignment, an incoming job is immediately dispatched to the host with the fewest number of jobs. This policy tries to equalize the instantaneous number of jobs at each host, rather than just the expected number of jobs. The `Least-Work-Left` policy sends each job to the host with the currently least remaining work. Observe that `Least-Work-Left` comes closest to obtaining instantaneous load balance. The `Central-Queue` policy holds all jobs at the dispatcher in a FCFS queue, and only when a host is free does the host request the next job. It has been proven (see section 1.3) that the `Least-Work-Left` policy is equivalent to the `Central-Queue` policy. Lastly, the `SITA-E` policy, suggested in [11], does duration-based assignment, where "short" jobs are assigned to Host 1, "medium-length" jobs are assigned to Host 2, "long" jobs to Host 3, etc., where the duration cutoffs are chosen so as to equalize load (SITA-E stands for Size Interval Task Assignment with Equal Load). This policy requires knowing the *approximate duration* of a job. All the above policies aim to balance the load among the server hosts.

What task assignment policy is generally used in practice? This is a difficult question to answer. Having studied Web pages and spoken to several system administrators, we conclude that task assignment policies vary widely, are not well understood, and often rely on adhoc parameters. The Web pages are very vague on this issue and are often contradicted by users of these systems [4]. The schedulers used are Load-Leveler, LSF, PBS, or NQS. These schedulers typically only support run-to-completion (no preemption) [15].

In many distributed servers, task assignment is done by the user (rather than a dispatcher). Typically, the user implements a `Least-Work-Left` policy as follows: with each submitted job, an estimated runtime is also submitted. A user then can compute the "work left" at a host by summing the running time estimates of the jobs queued at the hosts. Other distributed servers use more of a `SITA-E` policy, where different host machines have different duration limitations: up to 2 hours, up to 4 hours, up to 8 hours, or unlimited. In yet other distributed server systems, the scheduling policies are closer to `Round-Robin`.

1.3. Relevant previous work

The problem of task assignment in a model like ours has been studied extensively, but many basic questions remain open. See [11] for a long history of this problem. Much of the previous literature has only dealt with exponentially-distributed job service requirements. Under this model, it has been shown that the `Least-Work-Left` policy is the best. A recent paper, [11], has analyzed the above policies assuming the job service requirements are *i.i.d.* distributed according to a heavy-tailed Pareto distribution. Under that assumption `SITA-E` was shown to be the best of the above policies, by far. Several papers make the point that the distribution of the job service requirement has a huge impact on the relative performance of scheduling policies [6,8,11]. No paper we know of has compared the above task assignment policies on supercomputing trace data (real job service requirements).

The idea of purposely unbalancing load has been suggested previously in [2,5,10] under very different contexts from our paper. In [5] a distributed system with *preemptible* tasks is considered. It is shown that in the preemptible model, mean response time is minimized by *balancing* load, however mean slowdown is minimized by *unbalancing* load. In [2], *real-time scheduling* is considered where jobs have firm *deadlines*. In this context, the authors propose "load profiling", which "distributes load in such a way that the probability of satisfying the utilization requirements of incoming jobs is maximized". Paper [10] also considers the problem discussed in this paper, however in a different context. In [10] a task assignment policy for the case of *unknown job duration* is proposed. This algorithm also uses the idea of load unbalancing. The work in [10] is limited to analysis only, where the job runtimes are drawn from a Pareto distribution with particular parameters. While the Pareto distribution has been found to be empirically realistic, it still may not capture every aspect of job runtimes. In our work along with trace-driven simulation we also considered analysis, based on Pareto distributions. We found that analysis is a good predictor when the number of host machines was small, but a worse predictor under a large number of hosts.

1.4. Paper contributions

In this paper we propose to do two things: First, we will compare all of the task assignment policies listed in section 1.2 in a trace-driven simulation environment using job traces from supercomputing servers which fit the above model. In simulation we are able to study both mean and variance metrics. We also use analysis to validate some of the simulation results, and to provide a lot of intuition. We find that there are big differences between the performance of the task assignment policies. In the majority of this paper we concentrate on the case of 2 host machines. We find that `Random` and `Least-Work-Left` differ by a factor of 2–10 (depending on load) with respect to mean slowdown, and by a factor of 30 with respect to variance in slowdown. `Random` and `SITA-E` differ by a factor of 6–10 with respect to mean slowdown and by several orders of magnitude with respect to variance in slowdown. In the latter part of the paper we consider what happens when the number of hosts is large. We find that in the case of a very huge number of hosts the results reverse themselves and `Least-Work-Left` outperforms `SITA-E`. However, regardless of which policy is better, none of the above task assignment policies perform as well as we would like.

This leads us to the question of whether we are looking in the right search space for task assignment policies. We observe that all policies proposed thus far aim to balance load among the hosts. We propose a new policy which purposely *unbalances* load among the hosts. Counter-to-intuition, we show that this policy is also *fair* in that it achieves the same expected slowdown for all jobs – thus no jobs are biased against. We show surprisingly that the optimal degree of load unbalancing seems remarkably similar across many different workloads. We derive a rule of thumb for the appropriate degree of unbalancing. We evaluate our load unbalancing policy again using both trace-driven simulation and analysis. The performance of the load unbalancing policy improves upon the best of those policies which balance load by more than an order of magnitude with respect to mean slowdown and variance in slowdown.

We feel that the above results are dramatic enough that they should affect the direction we take in developing task assignment policies. We elaborate on this in the conclusion.

2. Experimental setup

This section describes the setup of our simulator and the trace data.

2.1. Collection of job traces

The first step in setting up our simulation was collecting trace data. In collecting job data, we sought data from systems which most closely matched the architectural model in our paper. We obtained traces from the PSC for the J90 and the C90 machines. Recall from section 1.1 that these machines are commonly configured into distributed server systems. Jobs on these machines are run-to-completion (no stopping/preempting). The jobs on these machines were submitted under the category of "batch" jobs.

The figures throughout this paper will be based on the C90 trace data. All the results for the J90 trace data are virtually identical and are provided in appendix B. For the purpose of comparison, we also consider a trace of jobs which comes from a 512-node IBM-SP2 at Cornell Theory Center (CTC). The CTC trace was obtained from Feitelson's Parallel Workloads Archive [7]. Although this trace did *not* come from the distributed server configuration, it is interesting in the context of this work since it reflects a common practice in supercomputing centers: unlike the J90 and C90 jobs, the jobs in the CTC trace had an upper bound on the run-time, since users are told jobs will be killed after 12 hours. We were surprised to find that although this upper bound leads to a considerably

Table 1
Characteristics of the trace data.

System	Duration	Number of jobs	Mean service requirement (sec)	Min (sec)	Max (sec)	Squared coefficient of variation
PSC C90	January–December 1997	54962	4562.6	1	2222749	43.16
PSC J90	January–December 1997	3582	9448.6	4	618532	10.02
CTC IBM-SP2	July 1996–May 1997	5729	2903.6	1	43138	5.42

lower variance in the service requirements, the comparative performance of the task assignment policies under the CTC trace was very similar to those for the J90 and C90 traces. All the CTC trace results[2] are shown in appendix C. Characteristics of all the jobs used in this paper are given in table 1.

2.2. Simulation setup

Our trace-driven simulation setup is very close to that of section 1.1. We simulate a distributed server for batch jobs with h host machines. *Throughout most of this paper we assume* $h = 2$. Jobs are dispatched immediately upon arrival to one of the host machines according to the task assignment policy. Jobs have *exclusive access* to host machines, and jobs are run-to-completion.

While the job *service requirements* are taken from a trace, we generate the *arrival times* according to a Poisson arrival process. The reason is that we were particularly interested in studying the performance of our task assignment policies for all ranges of system load. Using the original arrival times for these experiments makes extreme scaling of the interarrival times from the trace necessary. The same problem arises if the original distributed server has not the same number of hosts that we are simulating. Nevertheless, we redo all our experiments with scaled interarrival times in section 6.

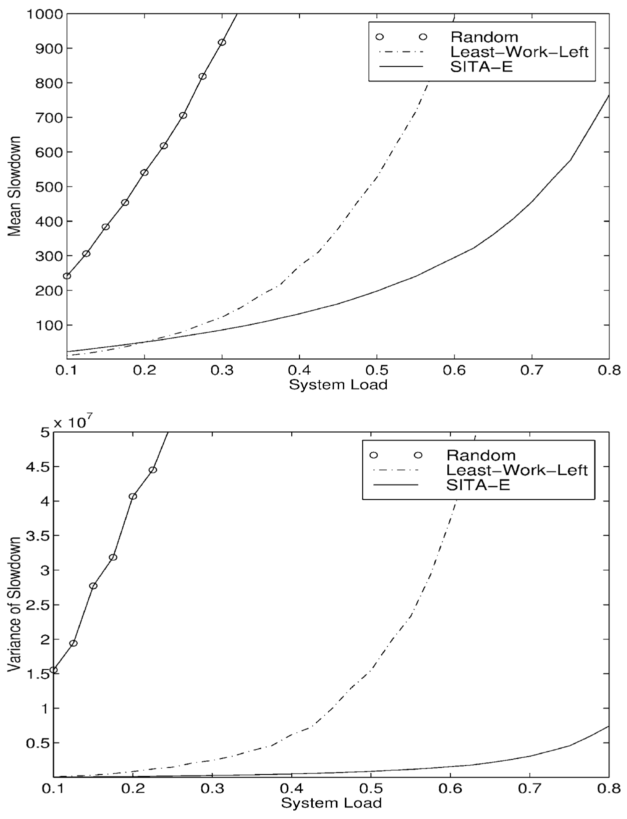

Figure 2. Experimental comparison of task assignment policies which balance load for a system with 2 hosts in terms of: (top) mean slowdown and (bottom) variance in slowdown.

3. Evaluation of policies which balance load

This section describes the result of our simulation of task assignment policies which aim to balance load.

3.1. The load balancing task assignment policies

The task assignment policies we evaluate are `Random`, `Least-Work-Left`, and `SITA-E`, as described in section 1.2. In [11] it was shown that the `Least-Work-Left` policy is equivalent to the `Central-Queue` policy for any sequence of job requests. Thus it suffices to only evaluate the former. We also evaluated the other policies mentioned, e.g., `Round-Robin`, but their performance is not notable and we omitted it to avoid cluttering the graphs.

3.2. Results from simulation

All the results in this section are trace-driven simulation results based on the C90 job data. The results for the J90 job data and other workloads are very similar and are shown in the appendix B. The plots only show system load up to 0.8 (because otherwise they become unreadable), however the discussion below spans all system loads under 1.

Figure 2(top) compares the performance of the policies which balance load (`Random`, `Least-Work-Left`, and `SITA-E`) in terms of their mean slowdown over a range of system loads. These results assume a 2-host distributed server system. Figure 2(bottom) makes the same comparison in terms of variance in slowdown. Observe that the slowdown under `Random` is higher than acceptable in any real system even for low loads and explodes for higher system loads. The slowdown under `Random` exceeds that of `SITA-E` by a factor of 10. The slowdowns under `SITA-E` and `Least-Work-Left` are quite similar for low loads, but for medium and high loads `SITA-E` outperforms `Least-Work-Left` by a factor of 3–4. The difference with respect to variance in slowdown is even more dramatic: `Least-Work-Left` improves upon the variance under `Random` by up to a factor

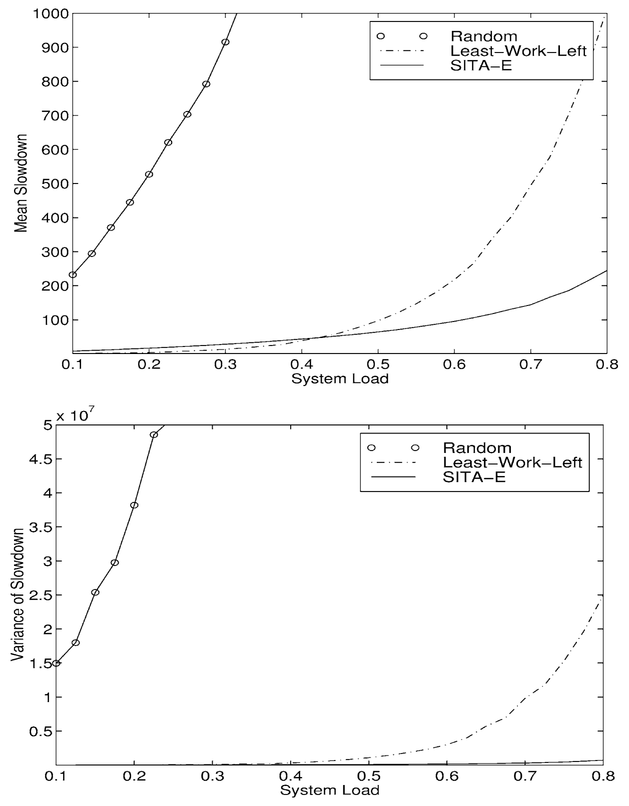

Figure 3. Experimental comparison of task assignment policies which balance load for a system with 4 hosts.

of 10 and SITA-E in turn improves upon the variance under Least-Work-Left by up to a factor of 10.

The same comparisons with respect to mean response time (not shown here) are very similar. For system loads greater than 0.5, SITA-E outperforms Least-Work-Left by factors of 2–3, and Random is by far the worst policy. The difference with respect to variance in response time is not quite as dramatic as for variance in slowdown.

Figure 3(top) again compares the performance of policies which balance load, except this time for a distributed server system with 4 hosts. Figure 3(bottom) makes the same comparison in terms of variance in slowdown. This figure shows that the slowdown and the variance in slowdown under both Least-Work-Left and SITA-E improves significantly when switching from 2 hosts to 4 hosts. The results for Random are the same as in the 2 host system. For low loads Least-Work-Left leads to lower slowdowns than SITA-E, but for system load 0.5 SITA-E improves upon Least-Work-Left by a factor of 2, and for high loads, SITA-E improves upon Least-Work-Left by a factor of 4. More dramatic are the results for the variance in slowdown: SITA-E's variance in slowdown is 25 times lower than that of Least-Work-Left.

3.3. *Results from analysis*

We also evaluated all of the above policies via analysis, based on the supercomputing workloads. Via analysis we were only able to evaluate the *mean* performance metrics. The results are shown in appendix A, figure 8. These are in very close agreement with the simulation results.

The analysis is beneficial because it explains *why* SITA-E is the best task assignment policy. For lack of space, we omit most of the analysis, providing the reader only with the resulting intuition.

The analysis of each task assignment policy makes use of the analysis of a single $M/G/1/$ FCFS queue, which is given in theorem 1 (Pollaczek–Kinchin) below:

Theorem 1. Given an $M/G/1$ FCFS queue, where the arrival process has rate λ, X denotes the service time distribution, and ρ denotes the utilization ($\rho = \lambda\mathbf{E}\{X\}$). Let W be a job's waiting time in queue, S be its slowdown, and Q be the queue length on its arrival. Then,

$$\begin{aligned}\mathbf{E}\{W\} &= \frac{\lambda\mathbf{E}\{X^2\}}{2(1-\rho)} \quad \text{(Pollaczek–Kinchin)},\\ \mathbf{E}\{S\} &= \mathbf{E}\{W/X\} = \mathbf{E}\{W\}\cdot\mathbf{E}\{X^{-1}\},\\ \mathbf{E}\{Q\} &= \lambda\mathbf{E}\{W\}.\end{aligned}$$

The above formula applies to just a single FCFS queue, *not* a distributed server. The formula says that *all* performance metrics for the FCFS queue are dependent on the *variance* of the distribution of job service demands (this variance term is captured by the $\mathbf{E}\{X^2\}$ term above). Intuitively, reducing the variance in the distribution of job processing requirements is important for improving performance because it reduces the chance of a short job getting stuck behind a long job. For our job service demand distribution, the variance is very high ($C^2 = 43$). Thus it will turn out that a key element in the performance of task assignment policies is how well they are able to reduce this variance.

We now discuss the effect of high variability in job service times on a distributed server system under the various task assignment policies.

Random Assignment The Random policy simply performs Bernoulli splitting on the input stream. The result is that each host becomes an independent $M/G/1$ queue, with the same (very high) variance in job service demands as was present in the original stream of jobs. Thus performance is very bad.

Round Robin The Round Robin policy splits the incoming stream so each host sees an $E_h/G/1$ queue, where h is the number of hosts. This system has performance close to the Random policy since it still sees high variability in service times, which dominates performance.

Least-Work-Left The Least-Work-Left policy is equivalent to an $M/G/h$ queue, for which there exist known approximations, [17,21]:

$$\mathbf{E}\{Q_{M/G/h}\} \approx \mathbf{E}\{Q_{M/M/h}\}\cdot\frac{\mathbf{E}\{X^2\}}{\mathbf{E}\{X\}^2},$$

where X denotes the service time distribution, and Q denotes queue length. What's important to observe here is that the mean queue length, and therefore the mean waiting time and slowdown, are all still proportional to the second moment of the service time distribution, as was the case for Random and Round-Robin. The Least-Work-Left policy does however improve performance for another reason: This policy is optimal with respect to sending jobs to idle host machines when they exist.

The SITA-E policy is the *only* policy which reduces the variance of job service times at the individual hosts. The reason is that Host 1 only sees small jobs and Host 2 only sees large jobs. For our data, $\mathbf{E}\{X^2_{\text{host 1}}\} = 4.5 \cdot 10^7$ and $\mathbf{E}\{X^2_{\text{host 2}}\} = 6.5 \cdot 10^{10}$ and $\mathbf{E}\{X^2\} = 9.2 \cdot 10^8$. Thus we've reduced the variance of the job service time distribution at Host 1 a lot, and increased that at Host 2. The point though is that 98.7% of jobs go to Host 1 under SITA-E and only 1.3% of jobs go to Host 2 under SITA-E. Thus SITA-E is a great improvement over the other policies with respect to mean slowdown, mean response time, and variance of slowdown and response time.

This discussion, so far, explains why SITA-E is so good: it reduces the variance of job service times. However, there are other effects, that are not so evident from the equations, but which become clear in trace-driven simulation. One is that Least-Work-Left gets much better when we increase the number of hosts. The reason is that it gets more likely that an arriving job can be assigned to a free host. Another one is, that the above analysis assumes that the interarrival times are independent identically distributed. One can imagine that if there are dependencies and many jobs with similar runtimes arrive simultaneously, the performance of SITA-E becomes worse.

4. Unbalancing load fairly

The previous policies all aimed to balance load. The Least-Work-Left policy in fact aimed to balance *instantaneous* load. However it is not clear *why* this is the best thing to do. We have no *proof* that load balancing minimizes mean slowdown or mean response time. In fact, a close look at the analysis shows that load *unbalancing* is desirable. In this section we show that load unbalancing is not only preferable with respect to all our performance metrics, but it is also desirable with respect to fairness. Recall, we adopt the following definition of fairness: All jobs, long or short, should experience the same expected slowdown. In particular, long jobs shouldn't be penalized – slowed down by a greater factor than are short jobs. Our goal in this section is to develop a fair task assignment policy with performance superior to that of all the other policies.

4.1. Definition of SITA-U-opt and SITA-U-fair

In searching for policies which don't balance load, we start with SITA-E, since in the previous section we saw that SITA-E was superior to all the other policies because of its variance-reduction properties. We define two new policies:

- SITA-U-opt: Size Interval Task Assignment with Unbalanced Load, where the service-requirement cutoff is chosen so as to minimize mean slowdown.
- SITA-U-fair: Size Interval Task Assignment with Unbalanced Load, where the service-requirement cutoff is chosen so as to maximize fairness.

In SITA-U-fair, the mean slowdown of *short* jobs is equal to the mean slowdown of *long* jobs.

The cutoff defining "long" and "short" for these policies was determined both analytically and experimentally using half of the trace data. The algorithms were then evaluated on the other half of the data.

Note that for a given cutoff we can compute the load and $\mathbf{E}\{X^2\}$ at each host from the trace data. Theorem 1 then allows us to determine the expected slowdown and response time for each host and hence also the overall slowdown and response time. Furthermore, the search space for the optimal cutoff is limited by the fact that neither host machine is allowed to exceed a load of 1. We can therefore find the optimal analytical cutoff by doing a search on the set of all feasible cutoffs. The experimental cutoffs are derived in the same way only that for a given cutoff we used simulation instead of analysis to find the corresponding slowdowns and response times. Both methods yielded about the same result.

4.2. Simulation results for SITA-U-opt and SITA-U-fair

Figure 4 compares SITA-E, the best of the load balancing task assignment policies, with SITA-U-opt and SITA-U-fair.

What is most interesting about the above figures is that SITA-U-fair is only a slight bit worse than SITA-U-opt. Both SITA-U-fair and SITA-U-opt improve greatly upon the performance of SITA-E, both with respect to mean slowdown and especially with respect to variance in slowdown. In the range of load 0.5–0.8, the improvement of SITA-U-fair over SITA-E ranges from 4 to 10 with respect to mean slowdown, and from 10 to 100 with respect to variance in slowdown.

4.3. Analysis of SITA-U-opt and SITA-U-fair

Figure 9 in appendix A shows the analytic comparison of mean slowdown for SITA-E, SITA-U-opt, and SITA-U-fair. These are in very close agreement with the simulation results.

Figure 5 shows the fraction of the total load which goes to Host 1 under SITA-U-opt and under SITA-U-fair. Observe that under SITA-E this fraction would always be 0.5.

Observe that for both SITA-U-opt and for SITA-U-fair we are underloading Host 1. Secondly observe that SITA-U-opt is not far from SITA-U-fair. In this section we explain these phenomena.

The reason why it is desirable to operate at unbalanced loads is mostly due to the *heavy-tailed* nature of our workload. In our job service time distribution, half the total load is

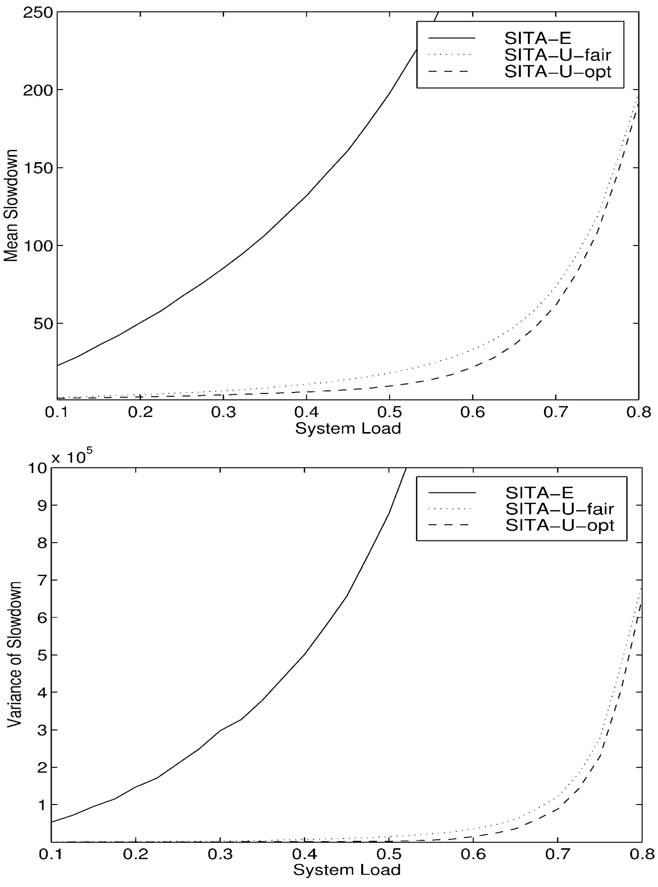

Figure 4. Experimental comparison of mean slowdown and variance of slowdown on SITA-E versus SITA-U-fair and SITA-U-opt as a function of system load.

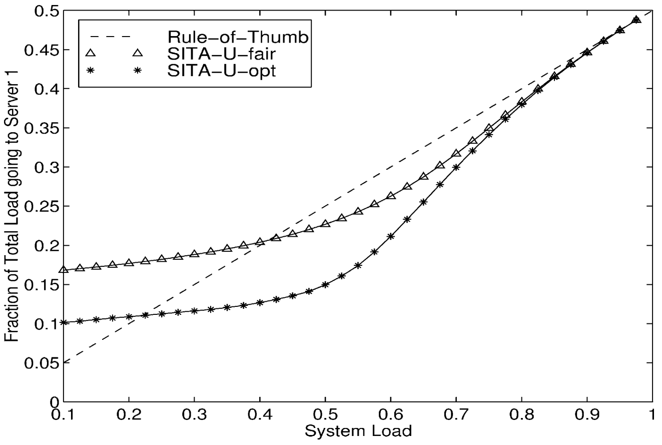

Figure 5. Fraction of the total load which goes to Host 1 under SITA-U-opt and SITA-U-fair and our rule of thumb.

made up by only the biggest 1.3% of all the jobs. This says that in SITA-E 98.7% of jobs go to Host 1 and only 1.3% of jobs go to Host 2. If we can reduce the load on Host 1 a little bit, by sending fewer jobs to Host 1, it will still be the case that most of the jobs go to Host 1, yet they are all running under a *reduced load.*

So load unbalancing optimizes mean slowdown, however it is not at all clear why load unbalancing also optimizes fairness. Under SITA-U-fair, the mean slowdown experienced by the short jobs is *equal* to the mean slowdown experienced by the long jobs. However it seems in fact that we're treating the long jobs unfairly because long jobs run on a host with extra load and extra variability in job durations.

So how can it possibly be fair to help short jobs so much? The answer is simply that the short jobs are short. Thus they need low response times to keep their slowdown low. Long jobs can afford a lot more waiting time, because they are better able to amortize the punishment over their long lifetimes. Note that this hold for all distributions. It is because our job service requirement distribution is so heavy-tailed that the long jobs are truly elephants (way longer than the shorts) and thus can afford more suffering.

4.4. A rule of thumb for load unbalancing

If load unbalancing is helpful, as seems to be the case, is there a rule of thumb for how much we should unbalance load?

Figure 5 gives a rough rule of thumb which says simply that if the system load is ρ, then the fraction of the load which is assigned to Host 1 should be $\rho/2$. For example, when the system load is 0.5, only $1/4$ of the total load should go to Host 1 and $3/4$ of the total load should go to Host 2. Contrast this with SITA-E which says that we should always send half the total load to each host.

We redid the simulations using out our rule-of-thumb cutoffs, rather than the optimal cutoffs, and the results were within 10%. We also tested out the rule-of-thumb when using the J90 data and when using the CTC data, and results were similar as well. Figures 11 and 13 in appendices B and C show the rule-of-thumb fit for the J90 data and the CTC data, respectively.

5. Systems with more than 2 machines

The results so far have all been based on systems with 2 or 4 hosts. Extending the SITA-policies to more hosts in the obvious way by using $h-1$ cutoffs for a system with h hosts has the disadvantage that it requires more precise runtime estimates. Also the search space for the optimal and fair cutoffs becomes much larger making the search computationally expensive. For this reason we decided to modify the SITA-policies slightly for experiments with larger machine numbers allowing each policy to use only the 2-host cutoff that has been derived for it previously. We divide the hosts into two groups, one for short jobs and one for long jobs. Each of the SITA-policies uses its 2-host cutoff to decide which jobs are short and which long and schedules the jobs within each group by Least-Work-Left. Figure 6 shows the slowdowns as a function of the number of machines for experiments at a system load of 0.7.[3]

Observe that the modified SITA-E is better than Least-Work-Left for small numbers of machines. However, for

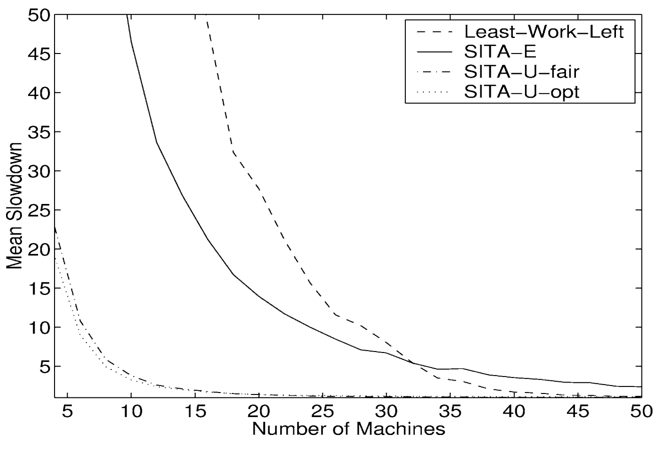

Figure 6. Results for systems with more than 4 machines and a system load of 0.7.

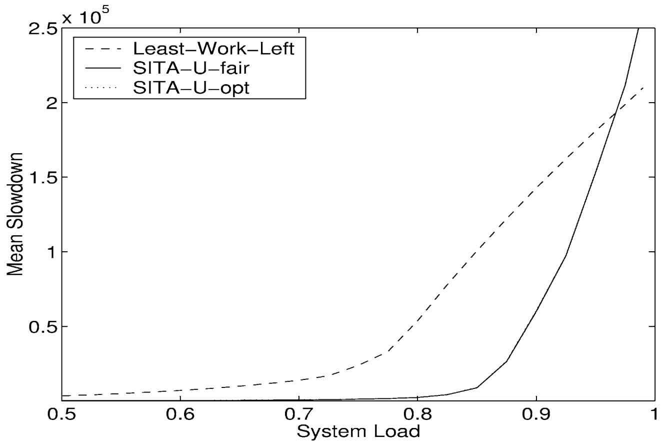

Figure 7. Results for scaled arrival times.

systems with a large number of machines Least-Work-Left outperforms SITA-E. The reason is that for systems with a large number of machines there is a high probability that one of the machines is idle or only lightly loaded. This is the case even for high system loads. Least-Work-Left can make efficient use of this idleness since it takes the current state of the machines into account when making its scheduling decisions. Nevertheless, for most numbers of machines Least-Work-Left still performs significantly worse than the modified versions of the two load unbalancing strategies (SITA-U-fair and SITA-U-opt). Only when the number of machines is larger than 70 the slowdown under all policies becomes comparable.

6. Non-Poisson arrivals

Throughout this paper we have repeatedly made the point that SITA-type policies (particularly SITA-U) can greatly improve over Least-Work-Left policies because they reduce the variability in the job sizes that a host machine sees. In this section we again compare SITA-U versus Least-Work-Left, but this time we remove the assumption of a Poisson-arrival-process. Instead we use the interarrival times from the traces, scaled to create the appropriate load. Setting up the arrival process this way will lead to a burstier arrival process. Since this makes the problem analytically intractable we use the analytical cutoffs derived under the Poisson assumption. We also determined the cutoffs via experiments and found that they agree with the analytical cutoffs derived under the Poisson assumption.

Figure 7 shows the results for Least-Work-Left and the two strategies that unbalance load (SITA-U-opt and SITA-U-fair).[4] The SITA-U-policies again significantly improve over Least-Work-Left for the range of loads that is most interesting for real systems (between 0.6 and 0.9). However, Least-Work-Left wins for very high loads (greater than 0.95).

To explain these results let us look at the two parameters that mainly affect the response times and hence also the slowdowns. The first one is the variability in the job size distribution. Its effect on the response time has been discussed extensively in sections 3 and 4. The second parameter, which we haven't discussed yet, is the variability in the interarrival times. A burstier arrival process will lead to higher response times. The SITA-type policies work well because they are the only ones that address the first issue: they reduce the variability in the job size distribution and thereby reduce response times. However they don't make any attempts to reduce the variability in the interarrival times. This is not a problem under a Poisson-arrival process, which exhibits a relatively low variability in the interarrival times. It is also not an issue under a bursty arrival process as long as the load is low. However, the variability in the interarrival times becomes the dominating factor under a bursty arrival process if the load is very high. This is why for high loads Least-Work-Left does better than the SITA-type policies: it is the only policy that reduces the variability in the arrival process.

7. Limitations and future work

In our simulations we assumed that the users correctly classify their jobs as short or long with respect to the given cutoff. It has been shown for several real systems including the CTC system that user runtime estimates can be rather poor. However, these systems typically maintain 15 or more different classes of jobs based on runtimes or require the user to give an absolute estimate of the runtime. In contrast we require the user only to estimate whether his job is short or long with respect to one cutoff. Furthermore, sending small jobs by mistake to the wrong machine will hurt only the performance of these small jobs (since their size is small compared to that of the other jobs on that machine). On the other hand a job will greatly benefit from being classified correctly giving users a strong incentive to care about their estimates.

Another way of obtaining runtime estimates, which does not depend on the willingness or ability of users to classify their jobs correctly, is to use techniques from machine learning to predict the runtime of a job. Recent work (see for example [9,16]) shows that in an MPP setting it is possible to predict runtimes based on historical information of previous similar runs. We are currently investigating whether similar techniques can be used to predict runtimes for the distributed server system we are concerned with.

8. Conclusions and implications

The contributions of this paper are detailed in section 1.4, so we omit the usual summary and instead discuss some further implications of this work. There are a few interesting points raised by this work:

- Task assignment policies differ widely in their performance (by an order of magnitude or more)! The implication is that we should take the policy determination more seriously, rather than using whatever Load Leveler gives us as a default.
- The "best" task assignment policy depends on characteristics of the distribution of job processing requirements. Thus workload characterization is important. Although our model is nothing like an MPP, the intuitions we learned here with respect to workloads may help simplify the complex problem of scheduling in this more complicated architecture as well.
- What appear to just be "parameters" of the task assignment policy (e.g., duration cutoffs) can have a greater effect on performance than anything else. Counter-to-intuition, a slight imbalance of the load can yield huge performance wins.
- Most task assignment policies don't perform as well as we would like, within the limitations of our architectural model. As Feitelson et al. [8] point out, to get good performance what we really need to do is favor short jobs in our scheduling (e.g., Shortest-Job-First). However, as Downey and Subhlok et al. point out, biasing may lead to starvation of certain jobs, and undesirable behavior by users [6,18]. What's nice about our `SITA-U-fair` policy is that it both gives extra benefit to short jobs (by allowing them to run on an underloaded host), while at the same time guaranteeing that the expected slowdown for short and long jobs is equal (fairness) – so that starvation is not an issue and users are not motivated to try to "beat the system". We feel that these are desirable goals for future task assignment policies.

Acknowledgements

We would like to thank Chad Vizino and Mike Levine at the PSC for providing us with access to logs from their Cray J90's and their Cray C90 and helping us to interpret these logs. Thanks also to Dror Feitelson for making public the CTC traces. Many thanks to the system administrators who spoke with us about task assignment in their systems, in particular Scott Blomquist and Chris Hill. Many others contributed to this paper via useful discussions: James Ho, Charles Leiserson, Larry Rudolph, Mark Squillante, Ken Stanley.

Notes

1. For example, Processor-Sharing (which requires infinitely-many preemptions) is ultimately fair in that every job experiences the same expected slowdown.
2. To make the workload more suitable for our model we used only those CTC jobs that require 8 processors, although using all jobs does lead to similar results.
3. `Random` was again by far the worst policy and is for better readability not included in the plot.
4. For high loads the results for `SITA-U-opt` and `SITA-U-fair` are virtually identical and therefore the two curves lie on top of each other.

Appendix A. Analytical results for distributed server running under C90 data

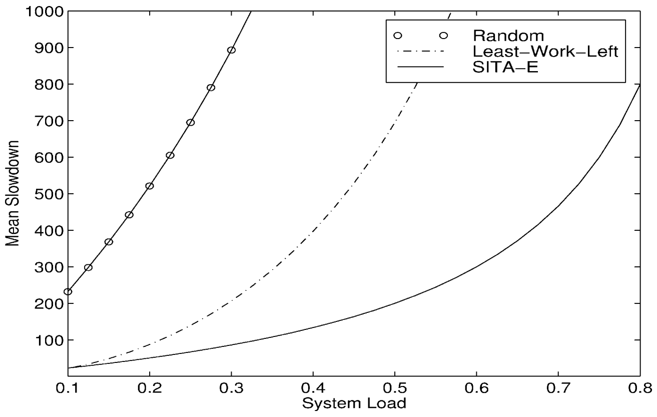

Figure 8. Analytical comparison of mean slowdown on task assignment policies which balance load, as a function of system load.

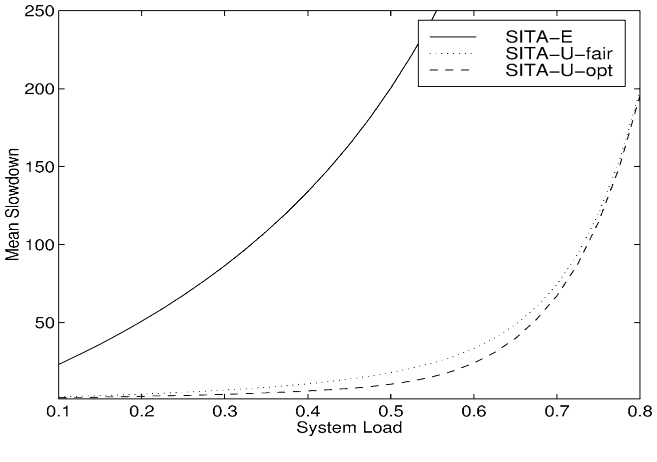

Figure 9. Analytical comparison of mean slowdown for `SITA-E` and `SITA-U-opt` and `SITA-U-fair`, as a function of system load.

Appendix B. Simulation results for distributed server under J90 data

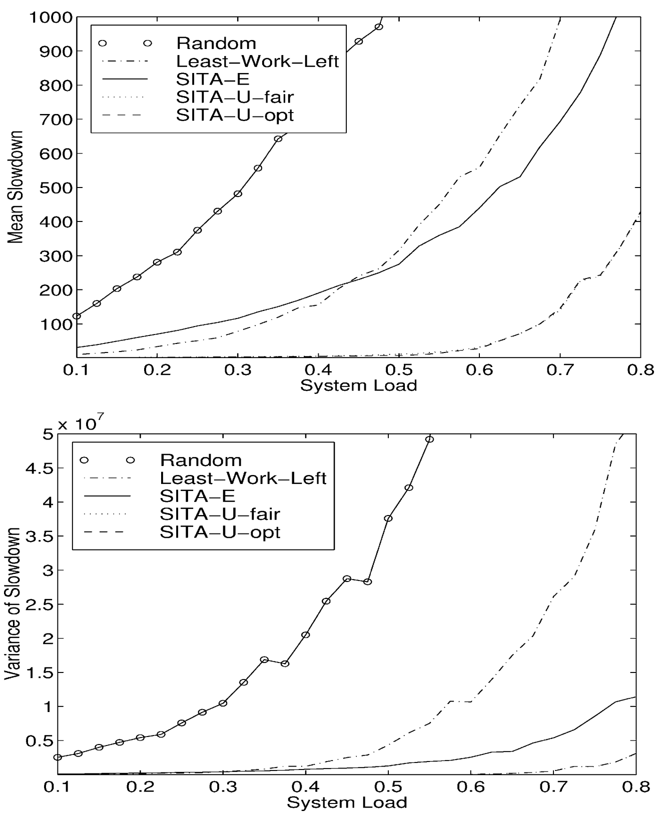

Figure 10. Experimental comparison of mean slowdown and variance of slowdown on all task assignment policies.

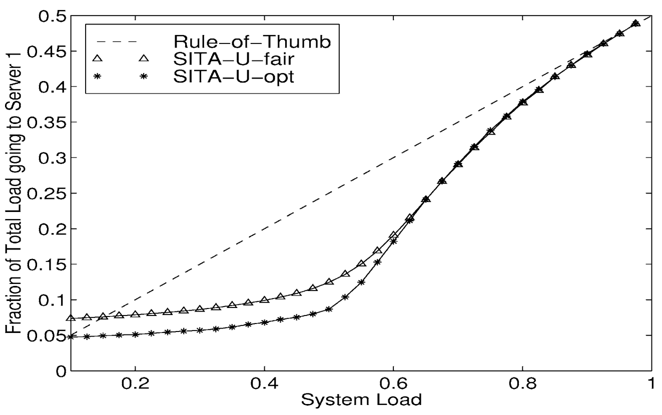

Figure 11. Fraction of the total load which goes to Host 1 under `SITA-U-opt` and `SITA-U-fair` and our rule of thumb.

Appendix C. Simulation results for distributed server under CTC data

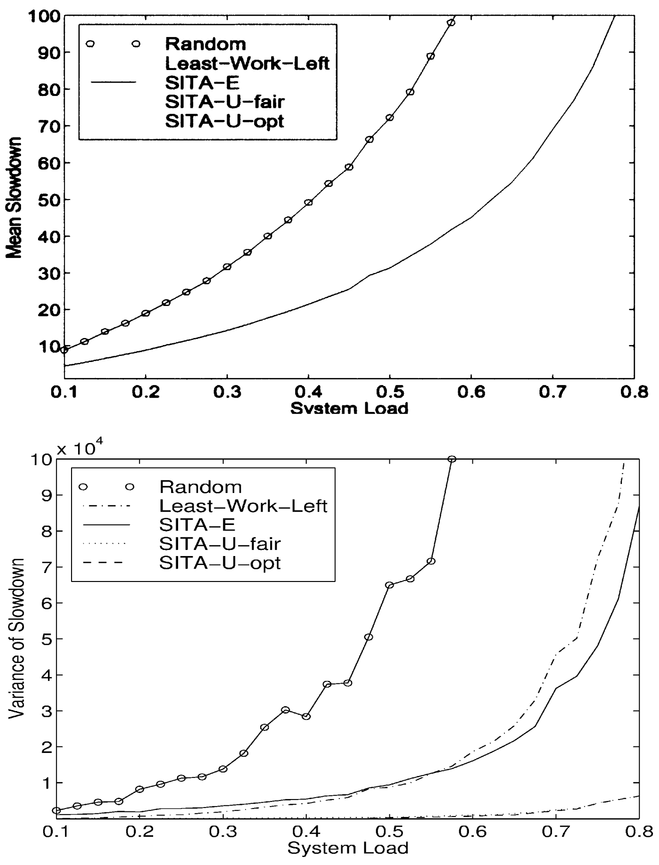

Figure 12. Experimental comparison of mean slowdown and variance of slowdown on all task assignment policies.

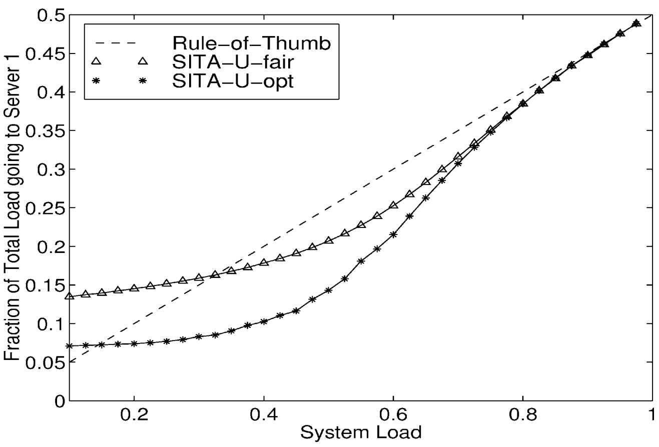

Figure 13. Fraction of the total load which goes to Host 1 under `SITA-U-opt` and `SITA-U-fair` and our rule of thumb.

References

[1] Baily, Foster, Hoang, Jette, Klingner, Kramer, Macaluso, Messina, Nielsen, Reed, Rudolph, Smith, Tomkins, Towns and Vildibill, Valuation of ultra-scale computing systems, White Paper (1999).

[2] A. Bestavros, Load profiling: A methodology for scheduling real-time tasks in a distributed system, in: *Proceedings of ICDCS '97* (May 1997).

[3] S. Blomquist and C. Hill, Personal communication (2000).

[4] S. Blomquist, C. Hill, J. Ho, C. Leiserson, L. Rudolph, M. Squillante and K. Stanley, Personal communication (2000).

[5] M.E. Crovella, M. Harchol-Balter and C. Murta, Task assignment in a distributed system: Improving performance by unbalancing load, in: *Sigmetrics '98 Poster Session* (1998).

[6] A.B. Downey, A parallel workload model and its implications for processor allocation, in: *Proceedings of High Performance Distributed Computing* (August 1997) pp. 112–123.

[7] D. Feitelson, The parallel workload archive, http://www.cs.huji.ac.il/labs/parallel/workload/ (1998).

[8] D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik and P. Wong, Theory and practice in parallel job scheduling, in: *Proceedings of IPPS/SPDP '97 Workshop*, Lecture Notes in Computer Science, Vol. 1291 (April 1997) pp. 1–34.

[9] R. Gibbons, A historical application profiler for use by parallel schedulers, in: *Proceedings of IPPS/SPDP '97 Workshop*, Lecture Notes in Computer Science, Vol. 1291 (April 1997) pp. 58–77.

[10] M. Harchol-Balter, Task assignment with unknown duration, in: *Proceedings of ICDCS* (2000), to appear.

[11] M. Harchol-Balter, M. Crovella and C. Murta, On choosing a task assignment policy for a distributed server system, IEEE Journal of Parallel and Distributed Computing 59 (1999) 204–228.

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

[13] C. Leiserson, The Pleiades alpha cluster at M.I.T., Documentation at: //http://bonanza.lcs.mit.edu/ (1998).

[14] C. Leiserson, The Xolas supercomputing project at M.I.T., Documentation available at: http://xolas.lcs.mit.edu (1998).

[15] E.W. Parsons and K.C. Sevcik, Implementing multiprocessor scheduling disciplines, in: *Proceedings of IPPS/SPDP '97 Workshop*, Lecture Notes in Computer Science, Vol. 1459 (April 1997) pp. 166–182.

[16] W. Smith, V. Taylor and I. Foster, Using runtime predictions to estimate queue wait times and improve scheduler performance, in *Proceedings of IPPS/SPDP '99 Workshop*, Lecture Notes in Computer Science, Vol. 1659 (April 1999) pp. 202–219.

[17] S. Sozaki and R. Ross, Approximations in finite capacity multiserver queues with poisson arrivals, Journal of Applied Probability 13 (1978) 826–834.

[18] J. Subhlok, T. Gross and T. Suzuoka, Impacts of job mix on optimizations for space sharing schedulers, in: *Proceedings of Supercomputing* (1996).

[19] Supercomputing at the NAS facility, http://www.nas.nasa.gov/Technology/Supercomputing/ (1998).

[20] The PSC's Cray J90's, http://www.psc.edu/machines/cray/j90/j90.html (1998).

[21] R.W. Wolff, *Stochastic Modeling and the Theory of Queues* (Prentice Hall, 1989).

Bianca Schroeder received her Master's degree in computer science from University of Saarland, Germany, in 1999. She is currently pursuing a Ph.D. in computer science at Carnegie Mellon University. Her research interests include performance analysis of computer networks and database systems and web traffic analysis.
E-mail: bianca@cs.cmu.edu

Mor Harchol-Balter received a Ph.D. in computer science from the University of California at Berkeley in 1996. From 1996 to 1999, Mor was funded by the NSF Postdoctoral Fellowship in the mathematical sciences at M.I.T. In the Fall of 1999, she joined Carnegie Mellon University as an Assistant Professor, and in 2001 received the McCandless chair. Mor is also a recipient of the NSF CAREER award. Mor is heavily involved in the ACM SIGMETRICS research community. Mor's work spans both analysis and implementation and emphasizes integrating measured workload distributions into the problem solution. Her research involves deriving often counter-intuitive theorems in the area of scheduling theory and queuing theory and applying these theorems towards building servers with improved performance.
E-mail: harchol@cs.cmu.edu