

# Task Assignment with Unknown Duration

MOR HARCHOL-BALTER

*Carnegie Mellon University, Pittsburgh, Pennsylvania*

**Abstract.** We consider a distributed server system and ask which policy should be used for assigning jobs (tasks) to hosts. In our server, jobs are *not* preemptible. Also, the job's service demand is *not* known a priori. We are particularly concerned with the case where the workload is heavy-tailed, as is characteristic of many empirically measured computer workloads. We analyze several natural task assignment policies and propose a new one TAGS (Task Assignment based on Guessing Size). The TAGS algorithm is counterintuitive in many respects, including load *unbalancing*, *non-work-conserving*, and *fairness*. We find that under heavy-tailed workloads, TAGS can outperform all task assignment policies known to us by several orders of magnitude with respect to both mean response time and mean slowdown, provided the system load is not too high. We also introduce a new practical performance metric for distributed servers called *server expansion*. Under the server expansion metric, TAGS significantly outperforms all other task assignment policies, regardless of system load.

**Categories and Subject Descriptors:** C.1.4 [**Processor Architectures**]: Parallel Architectures—*distributed architectures*; C.4 [**Performance of Systems**]: *design studies*; D.4.8 [**Operating Systems**]: Performance—*modeling and prediction*; *queueing theory*

**General Terms:** Design, Performance

**Additional Key Words and Phrases:** Clusters, contrary behavior, distributed servers, fairness, heavy-tailed workloads, high variance, job scheduling, load balancing, load sharing, supercomputing, task assignment

## 1. Introduction

In recent years, distributed servers have become increasingly common because they allow for increased computing power while being cost-effective and easily scalable.

In a distributed server system, jobs (tasks) arrive and must each be dispatched to exactly one of several host machines for processing. We assume for simplicity that these host machines are identical and that there is no cost (time required) for dispatching jobs to hosts. The rule for assigning jobs to host machines is known as the *task assignment policy*. The choice of the task assignment policy

---

This article originally appeared as Technical Report CMU-CS-99-162, August 1999.

An extended abstract version appeared in *Proceedings of the ICDCS '00*.

Author's address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, e-mail: harchol@cs.cmu.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

© 2002 ACM 0004-5411/02/0300-0260 \$5.00

TABLE I. EXAMPLES OF DISTRIBUTED SERVERS DESCRIBED BY THE ARCHITECTURAL MODEL OF THIS PAPER. THE SCHEDULERS USED ARE LOAD-LEVELER, LSF, PBS, OR NQS. THESE SCHEDULERS TYPICALLY ONLY SUPPORT RUN-TO-COMPLETION (NONPREEMPTIVE) [PARSONS AND SEVCIK 1997]

Name	Location	No. Hosts	Host Machine
Xolas [Leiserson 1998b]	MIT Lab for Computer Science	8	8-processor Ultra HPC 5000 SMP
Pleiades [Leiserson 1998a]	MIT Lab for Computer Science	7	4-processor Alpha 21164 machine
J90 distributed server	NASA Ames Research Lab	4	8-processor Cray J90 machine
J90 distributed server [PSC Cray 1998]	Pittsburgh Supercomputing Center	2	8-processor Cray J90 machine
C90 distributed server [Supercomputing 1998]	NASA Ames Research Lab	2	16-processor Cray C90 machine

has a significant effect on the performance perceived by users. Designing a distributed server system often comes 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 that still remains open for many models.

In this article, we consider the particular model of a distributed server system in which jobs are *not preemptible*—that is, each job is *run-to-completion*. Jobs can be aborted, but then all work is lost and the job must be restarted from scratch. Our model is motivated by distributed servers for batch computing at Supercomputing Centers. For these distributed servers, each host machine is usually a multiprocessor machine (e.g., an 8-processor Cray J90) and each job submitted to the distributed server is a parallel job, intended to run on a single host. In such a setup, jobs are usually run-to-completion, rather than time-shared, for several reasons: First, the memory requirements of jobs tend to be huge, making it very expensive to swap out a job’s memory [Feitelson et al. 1997]. Thus, timesharing between jobs only makes sense if all the jobs being timeshared fit within the memory of the host, which is very unlikely. Also, many operating systems that enable timesharing for single-processor jobs do not facilitate preemption among several processors in a coordinated fashion [Parsons and Sevcik 1997]. Examples of distributed server systems that fit the above description are given in Table I.

Lastly, we assume that *no a priori information* is known about the job at the time when the job arrives. In particular, the *processing requirement* of the job is *not known*. We use the terms *processing requirement*, *CPU requirement*, *service demand*, and *size* interchangeably. Many studies have shown that even in cases where user estimates of their job processing requirements are available, those estimates are grossly inaccurate. For example, one study shows that for 38% of jobs, the actual processing requirement is only 4% of the user-predicted requirement, and for over 95% of jobs the actual processing requirement is under 10% of the user-predicted requirement [Feitelson and Jette 1997].

Figure 1 is one illustration of a distributed server. In this illustration, arriving jobs are immediately dispatched by the central dispatcher to one of the hosts and queue up at the host waiting for service, where they are served in first-come-first-served (FCFS) order. Observe however that our model in general does not preclude the possibility of having a central queue at the dispatcher where jobs might wait before being dispatched.

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 waiting time divided by its service requirement. All means are per-job

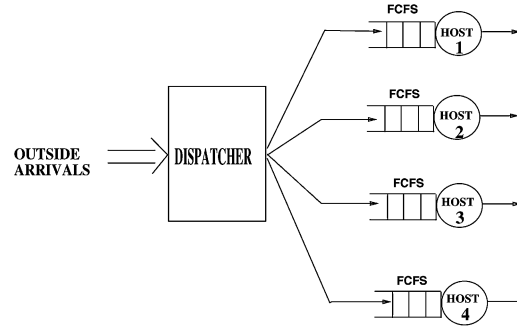


FIG. 1. Illustration of a distributed server.

averages. Mean slowdown is important because it is desirable that a job's delay be proportional to its processing requirement [Downey 1997; Rudolph and Smith 2000; Harchol-Balter and Downey 1997]. Users are likely to anticipate short delays for short jobs, and are likely to tolerate long delays for longer jobs. A secondary 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 should not be penalized—slowed down by a greater factor than are short jobs.<sup>1</sup>

Observe that for the architectural model we consider in this article, memory usage is not an issue with respect to scheduling. Recall that 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 assignment, jobs are assigned to hosts in a cyclical fashion with the  $i$ th 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 assignment, an incoming job is immediately dispatched to the host with the fewest *number* of jobs. This policy has the benefit of trying to equalize the instantaneous number of jobs at each host, rather than just the expected number of jobs.

Ideally, we would like to send a job to the host that has the least total outstanding *work* (work is the sum of the job processing requirements at the host) because that host would afford the job the shortest waiting time. However, we don't know a priori which host currently has the least work, since we don't know job processing requirements (sizes). Imagine for a moment that we did, however, know job sizes. Then we could imagine a Least-Work-Remaining policy that sends each job to the host with the currently least remaining work. It is in fact possible to

<sup>1</sup> For example, Processor-Sharing (which requires infinitely-many preemptions) is ultimately fair in that every job experiences the same expected slowdown.

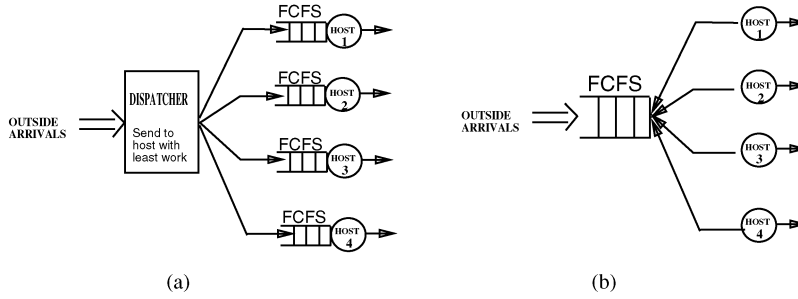


FIG. 2. The (a) Least-Work-Remaining policy and the (b) Central-Queue policy are equivalent.

achieve the performance of Least-Work-Remaining *without* knowing job sizes: Consider a different policy, called Central-Queue. 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 turns out that Central-Queue is equivalent to Least-Work-Remaining for any sequence of job requests (see Harchol-Balter et al. [1999] for a rigorous proof and Figure 2 for an illustration). Thus, since Central-Queue does not require a priori knowledge of job sizes, we can in fact achieve the performance of Least-Work-Remaining without requiring knowledge of the job sizes.

It may seem that Least-Work-Remaining is the best possible task assignment policy. In fact, previous literature suggests that it is the optimal policy if the job size distribution is *exponential* (see Section 2). This is not in conflict with our results.

But what if job size distribution is not exponential? We are motivated in this respect by the increasing evidence for high variability in job-size distributions, as seen in many measurements of computer workloads. In particular, measurements of many computer workloads have been shown to fit heavy-tailed distributions with very high variance, as described in Section 3—much higher variance than that of an exponential distribution. Is there a better policy than Least-Work-Remaining when the job-size variability is characteristic of empirical workloads? In evaluating various policies, we will be interested in understanding the influence of job size variability on the decision of which policy is best. For analytical tractability, we assume that the arrival process is Poisson—previous work indicates that the variability in the arrival process is much less critical to choosing a task assignment policy than is the variability in the job-size distribution [Schroeder and Harchol-Balter 2000].

In this article, we propose a new algorithm called TAGS—Task Assignment by Guessing Size—which is specifically designed for high variability workloads. TAGS works by associating a time limit with each host. A job is run at a host up to the designated time limit associated with the host. If the job has not completed at this point, it is killed and restarted from scratch at a new host. We prove analytically that when job sizes show the degree of variability characteristic of empirical (measured) workloads, the TAGS algorithm can outperform all the above-mentioned policies by *several orders of magnitude*. In fact, we show that the more heavy-tailed the job-size distribution, the greater the improvement of TAGS over the other policies.

The above improvements are contingent on the system load not being too high.<sup>2</sup> In the case where the system load is high, we show that all the policies perform so poorly that they become impractical, and TAGS is especially negatively affected. However, in practice, if the system load is too high to achieve reasonable performance, one adds new hosts to the server (without increasing the outside arrival rate), thus dropping the system load, until the system behaves as desired. We refer to the “number of new hosts that must be added” as the *server expansion* requirement. We show that TAGS outperforms all the previously mentioned policies with respect to the *server expansion* metric (i.e., given *any initial system load*, TAGS requires far fewer additional hosts to perform well).

We describe three flavors of TAGS. The first, TAGS-opt-slowdown, is designed to minimize mean slowdown. The second, TAGS-opt-waitingtime, is designed to minimize mean waiting time. Although very effective, these algorithms are not fair in their treatment of jobs. The third flavor, TAGS-opt-fairness, optimizes fairness. While managing to be fair, TAGS-opt-fairness still achieves mean slowdown and mean waiting time close to the other flavors of TAGS. The *point of this article* is not to promote the TAGS algorithm in particular, but rather to promote an appreciation for the unusual and counterintuitive ideas on which TAGS is based, namely: load *unbalancing*, *non-workconserving*, and *fairness*.

Section 2 elaborates on previous work. Section 3 provides the necessary background on measured job size distributions and heavy-tails. Section 4 describes the TAGS algorithm and all its flavors. Section 5 shows results of analysis for the case of two hosts, and Section 6 shows results of analysis for the multiple-host case. Section 7 explores the effect of less-variable job-size distributions. Lastly, we conclude in Section 8. Details on the analysis of TAGS are described in the Appendix.

## 2. Previous Work on Task Assignment

2.1. TASK ASSIGNMENT WITH NO PREEMPTION. The problem of task assignment in a model like ours (no preemption<sup>3</sup> and no a priori knowledge) has been extensively studied, but many basic questions remain open.

One subproblem which has been solved is that of task assignment under the *further restriction* that all jobs be immediately dispatched to a host upon arrival. Under this restricted model, Winston showed that when the job-size distribution is exponential and the arrival process is Poisson, then the Shortest-Queue task assignment policy is optimal [Winston 1977]. In this result, optimality is defined as maximizing the discounted number of jobs that complete by some fixed time  $t$ . Ephremides et al. [1980] showed that Shortest-Queue also minimizes the expected total time for the completion of all jobs arriving by some fixed time  $t$ , under an exponential job size distribution and arbitrary arrival process. Koole et al. [1999]

<sup>2</sup> For a distributed server, system load is defined as follows:

$$\text{System load} = \text{Outside arrival rate} \cdot \text{Mean job size} / \text{Number of hosts}.$$

For example, a system with 2 hosts and system load .5 has same outside arrival rate as a system with four hosts and system load .25. Observe that a four host system with system load  $\rho$  has twice the outside arrival rate of a two-host system with system load  $\rho$ .

<sup>3</sup> All the results here assume FCFS service order at each host machine.

showed that Shortest-Queue is optimal if the job-size distribution has Increasing Likelihood Ratio (ILR). The actual performance of the Shortest-Queue policy is not known exactly, but the mean response time is approximated by Nelson and Phillips [1989, 1993]. Whitt [1986] has shown that as the variability of the job size distribution grows, Shortest-Queue is no longer optimal. Whitt does not suggest which policy is optimal. Koole et al. [1999] later showed that Shortest-Queue is not even optimal for all job-size distributions with Increasing Failure Rate.

Under the model assumed in this article, but with *exponentially* distributed job sizes, several papers [Nelson and Phillips 1989, 1993] claim that the Central-Queue (or equivalently, Least-Work-Remaining) policy is optimal. Wolff [1989] suggests that Least-Work-Remaining is optimal because it maximizes the number of busy hosts, thereby maximizing the downward drift in the continuous-time Markov chain whose states are the number of jobs in the system.

Another model that has been considered is the case of no preemption but where the size of each job is *known* at the time of arrival of the job. Within this model, the SITA-E algorithm (see Harchol-Balter et al. [1999]) has been shown to outperform the Random, Round-Robin, Shortest-Queue, and Least-Work-Remaining algorithms by several orders of magnitude when the job-size distribution is heavy-tailed. In contrast to SITA-E, the TAGS algorithm does not require knowledge of job size. Nevertheless, for not-too-high system loads ( $<.5$ ), TAGS improves upon the performance of SITA-E by several orders of magnitude for heavy-tailed workloads.

**2.2. WHEN PREEMPTION IS ALLOWED.** Throughout this article, we maintain the assumption that jobs are *not* preemptible. That is, once a job starts running, it can not be stopped and recontinued where it left off. By contrast, there exists considerable work on the *very different* problem where jobs are preemptible and maybe even migrateable (see Harchol-Balter and Downey [1997] for many citations).

**2.3. TAGS-LIKE ALGORITHMS.** The idea of purposely unbalancing load has been suggested previously in Crovella et al. [1998a] and in Bestavros [1997], under different contexts from our article. In both these papers, it is assumed that job sizes are *known* a priori. In Crovella et al. [1998a], a distributed system with *preemptible* jobs is considered. It is shown that, in the preemptible model, mean waiting time is minimized by *balancing* load; however, mean slowdown is minimized by *unbalancing* load. In Bestavros [1997], real-time scheduling is considered where jobs have firm *deadlines*. In this context, the authors propose “load profiling,” which distributes load so that the probability of satisfying the utilization requirements of incoming jobs is maximized.

To the best of our knowledge, the TAGS idea of associating artificial “time-limits” with machines, killing jobs that exceed the time-limit on their machines, and restarting those jobs on hosts with higher time-limits, has not been considered before.

### 3. Heavy Tails

As described in Section 1, we are concerned with how the distribution of job sizes affects the decision of which task assignment policy to use.

Many application environments show a mixture of job sizes spanning many orders of magnitude. In such environments, there are typically many short jobs, and fewer

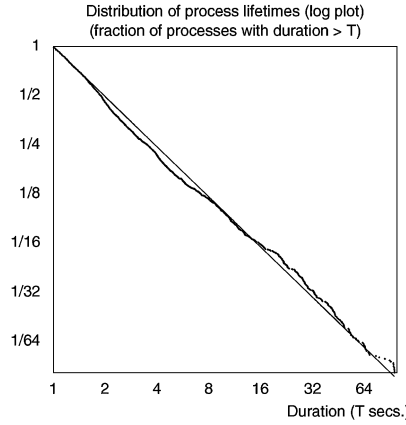


FIG. 3. Measured distribution of UNIX process CPU lifetimes, taken from Harchol-Balter and Downey [1997]. Data indicates fraction of jobs whose CPU service requirement exceeds  $T$  seconds, as a function of  $T$ .

long jobs. Much previous work has used the *exponential* distribution to capture this variability, as described in Section 2. However, recent measurements indicate that for many applications the exponential distribution is a poor model and that a *heavy-tailed* distribution is more accurate. In general, a heavy-tailed distribution is one for which

$$\Pr\{X > x\} \sim x^{-\alpha},$$

where  $0 < \alpha < 2$ . The simplest heavy-tailed distribution is the *Pareto* distribution, with probability mass function

$$f(x) = \alpha k^\alpha x^{-\alpha-1}, \quad \alpha, k > 0, \quad x \geq k,$$

and cumulative distribution function

$$F(x) = \Pr\{X \leq x\} = 1 - \left(\frac{k}{x}\right)^\alpha.$$

A set of job sizes following a heavy-tailed distribution has the following properties:

- (1) Decreasing failure rate: In particular, the longer a job has run, the longer it is expected to continue running.
- (2) Infinite variance (and if  $\alpha \leq 1$ , infinite mean).
- (3) The property that a tiny fraction ( $< 1\%$ ) of the very longest jobs comprise over half of the total load. We refer to this important property throughout the article as the *heavy-tailed property*.

The lower the parameter  $\alpha$ , the more variable the distribution, and the more pronounced is the heavy-tailed property, that is, the smaller the fraction of long jobs that comprise half the load.

As a concrete example, Figure 3 depicts graphically on a log-log plot the measured distribution of CPU requirements of over a million UNIX processes, taken

from Harchol-Balter and Downey [1997]. This distribution closely fits the curve

$$\Pr\{\text{Process CPU requirement} > T\} = \frac{1}{T}.$$

In Harchol-Balter and Downey [1997], it is shown that this distribution is present in a variety of computing environments, including instructional, research, and administrative environments.

In fact, heavy-tailed distributions appear to fit many recent measurements of computing systems. These include, for example:

- Unix process CPU requirements measured at Bellcore:  $1 \leq \alpha \leq 1.25$  [Leland and Ott 1986].
- Unix process CPU requirements, measured at UC Berkeley:  $\alpha \approx 1$  [Harchol-Balter and Downey 1997].
- Sizes of files transferred through the Web:  $1.1 \leq \alpha \leq 1.3$  [Crovella and Bestavros 1997; Crovella et al. 1998b].
- Sizes of files stored in Unix filesystems: [Irlam 1994].
- I/O times: [Peterson and Adams 1996].
- Sizes of FTP transfers in the Internet:  $.9 \leq \alpha \leq 1.1$  [Paxson and Floyd 1995].
- Pittsburgh Supercomputing Center (PSC) workloads for distributed servers consisting of Cray C90 and Cray J90 machines [Schroeder and Harchol-Balter 2000].<sup>4</sup>

In most of these cases, where estimates of  $\alpha$  were made,  $\alpha$  tends to be close to 1, which represents very high variability in job-service requirements.

In practice, there is some upper bound on the maximum size of a job, because files only have finite lengths. Throughout this article, we therefore model job sizes as being generated i.i.d. from a distribution that is heavy-tailed, but has an upper bound—a very high one. We refer to this distribution as a *Bounded Pareto*. It is characterized by three parameters:  $\alpha$ , the exponent of the power law;  $k$ , the shortest possible job; and  $p$ , the largest possible job. The probability density function for the Bounded Pareto  $B(k, p, \alpha)$  is defined as:

$$f(x) = \frac{\alpha k^\alpha}{1 - (k/p)^\alpha} x^{-\alpha-1} \quad k \leq x \leq p. \quad (1)$$

In this article, we vary the  $\alpha$ -parameter over the range 0 to 2 in order to observe the effect of *variability of the distribution*. To focus on the effect of changing variance, we keep the distributional mean fixed (at 3000) and the maximum value fixed (at  $p = 10^{10}$ ), which correspond to typical values taken from Crovella and Bestavros [1997]. In order to keep the mean constant, we adjust  $k$  slightly as  $\alpha$  changes ( $0 < k \leq 1500$ ).

Note that the Bounded Pareto distribution has all its moments finite. Thus, it is not a heavy-tailed distribution in the sense we have defined above. However, this

<sup>4</sup> While the distribution of job processing requirements at the PSC does not seem to exactly fit a Pareto distribution, these workloads do have a very strong heavy-tailed property and high variance. Specifically, our measurements showed that half the load is made up by only the biggest 1.3% of all jobs, and the squared coefficient of variation is 43.



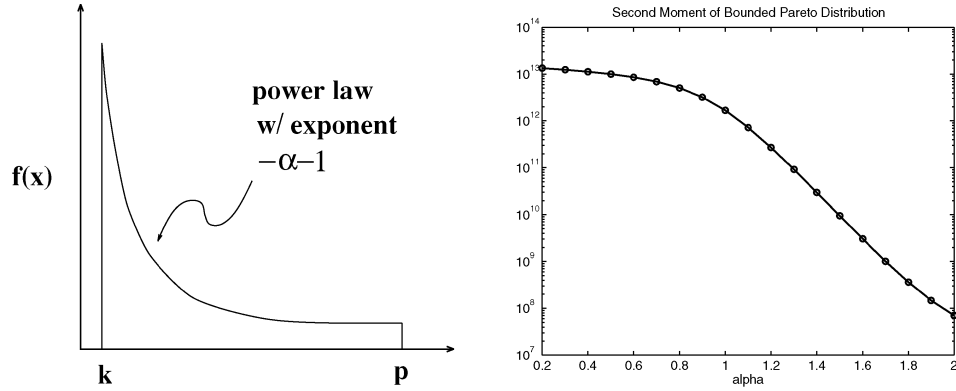


FIG. 4. Parameters of the Bounded Pareto Distribution (left); Second Moment of  $B(k, p = 10^{10}, \alpha)$  as a function of  $\alpha$ , when  $\mathbf{E}\{X\} = 3000$  (right).

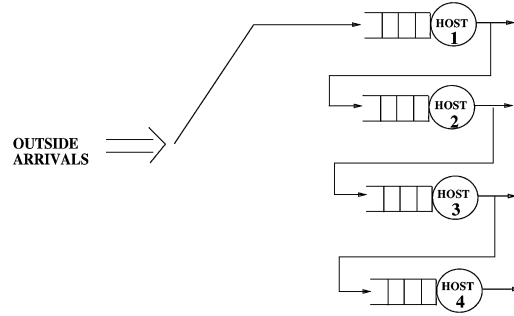


FIG. 5. Illustration of the flow of jobs in the TAGS algorithm.

distribution will still show very high variability if  $k \ll p$ . For example, Figure 4 (right) shows the second moment  $\mathbf{E}\{X^2\}$  of this distribution as a function of  $\alpha$  for  $p = 10^{10}$ , where  $k$  is chosen to keep  $\mathbf{E}\{X\}$  constant at 3000, ( $0 < k \leq 1500$ ). The figure shows that the second moment explodes exponentially as  $\alpha$  declines. Furthermore, the Bounded Pareto distribution also still exhibits the heavy-tailed property and (to some extent) the decreasing failure rate property of the unbounded Pareto distribution. We mention these properties because they are important in choosing the best task assignment policy.

#### 4. The TAGS Algorithm

This section describes the TAGS algorithm. Let  $h$  be the number of hosts in the distributed server. Think of the hosts as being numbered:  $1, 2, \dots, h$ . The  $i$ th host has a number  $s_i$  associated with it, where  $s_1 < s_2 < \dots < s_h$ .

TAGS works as shown in Figure 5: *All* incoming jobs are immediately dispatched to Host 1. There they are serviced in FCFS order. If they complete before using up  $s_1$  amount of CPU, they simply leave the system. However, if a job has used  $s_1$  amount of CPU at Host 1 and still has not completed, then it is killed (remember, jobs cannot be preempted). The job is then put at the end of the queue at Host 2,

where it must be restarted from scratch.<sup>5</sup> Each host services the jobs in its queue in FCFS order. If a job at host  $i$  uses up  $s_i$  amount of CPU and still has not completed it is killed and put at the end of the queue for Host  $i + 1$ . In this way, the TAGS algorithm “guesses the size” of each job, hence the name.

The TAGS algorithm may sound counterintuitive for a few reasons: First of all, there’s a sense that the higher-numbered hosts will be underutilized and the first host overcrowded since all incoming jobs are sent to Host 1. An even more vital concern is that the TAGS algorithm *wastes* a large amount of resources by killing jobs and then restarting them from scratch.<sup>6</sup> There is also the sense that the big jobs are especially penalized since they are the ones being restarted.

TAGS comes in three flavors; these only differ in how the  $s_i$ ’s are chosen. In TAGS-opt-slowdown, the  $s_i$ ’s are chosen so as to optimize mean slowdown. In TAGS-opt-waitingtime, the  $s_i$ ’s are chosen so as to optimize mean waiting time. As we’ll see, TAGS-opt-slowdown and TAGS-opt-waitingtime are not necessarily fair. In TAGS-opt-fairness, the  $s_i$ ’s are chosen so as to optimize fairness. Specifically, the jobs whose final destination is Host  $i$  experience the same expected slowdown under TAGS-opt-fairness as do the jobs whose final destination is Host  $j$ , for all  $i$  and  $j$ .

To use TAGS, one needs to compute the appropriate  $s_i$  cutoffs. The  $s_i$ ’s are a function of the distribution of job sizes (which in our case is defined by the parameters  $\alpha$ ,  $k$ , and  $p$ ) and the average outside arrival rate  $\lambda$ . These workload parameters can be determined by observing the system for a period of time. To determine the  $s_i$ ’s, given these workload parameters, we use *Mathematica*<sup>TM</sup> as described in the Appendix to solve for the optimal values of the  $s_i$ ’s, which minimize the performance formulas for mean slowdown, mean response time, etc.

TAGS may seem reminiscent of multilevel feedback queuing, but these are *not* related. In multilevel feedback queuing, there is only a single host with many *virtual* queues. The host is time-shared and jobs are preemptible. When a job uses some amount of service time, it is transferred (not killed and restarted) to a lower priority queue. Also, in multilevel feedback queuing, the jobs in that lower priority queue are *only* allowed to run when the higher priority queues are empty.

### 5. Analytic Results for the Case of Two Hosts

This section contains the results of our analysis of the TAGS task assignment policy and other policies. In order to clearly explain the effect of the TAGS algorithm, we limit the discussion in this section to the case of two hosts. In this case, we refer to the jobs whose final destination is Host 1 as the *short jobs* and the jobs whose final destination is Host 2 as the *big jobs*. Until Section 5.3, we always assume that the system load is 0.5 and there are two hosts. In Section 5.3, we consider other system loads, but still stick to the case of two hosts. Finally, in Section 6, we consider distributed servers with  $>2$  hosts.

We evaluate the Random, Least-Work-Remaining, and TAGS policies via analysis, all as a function of  $\alpha$ , where  $\alpha$  is the variance parameter for the Bounded Pareto

<sup>5</sup> Note, although the job is restarted, it is still the same job, of course. We must therefore be careful in our analysis *not* to assign it a new service requirement.

<sup>6</sup> My dad, Micha Harchol, would add that there’s also the psychological concern of what the angry user might do when he’s told his job has been killed to help the general good.

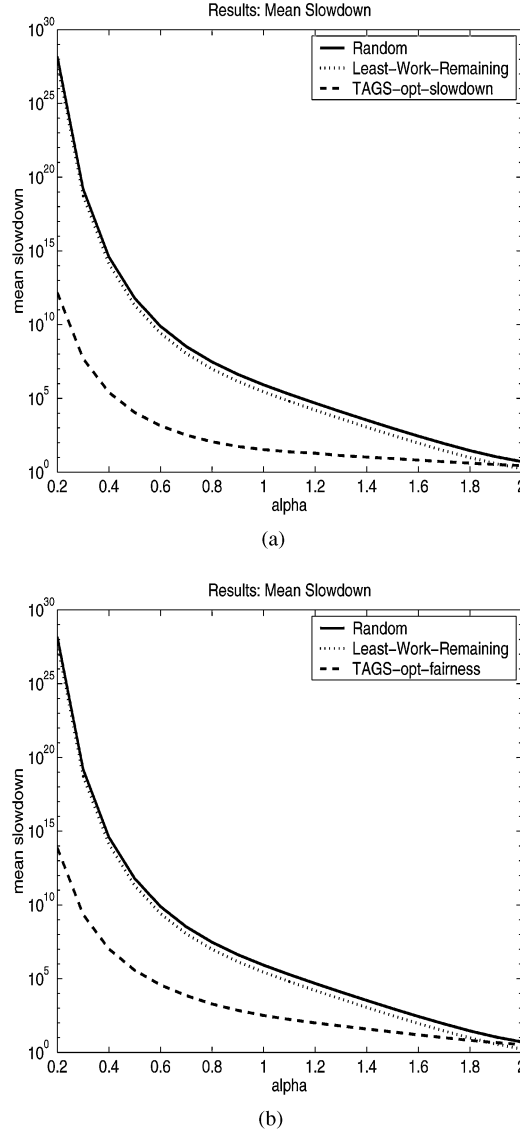


FIG. 6. Mean slowdown for distributed server with two hosts and system load .5 under (a) TAGS-opt-slowdown and (b) TAGS-opt-fairness as compared with the Least-Work-Remaining and Random task assignment policies.

job size distribution, and  $\alpha$  ranges between 0 and 2. Recall from Section 3 that the lower  $\alpha$  is, the higher the variance in the job-size distribution. Recall also that empirical measurements of job-size distributions often show  $\alpha \approx 1$ . Round-Robin (see Section 1) will not be evaluated directly because we showed in a previous paper [Harchol-Balter et al. 1999] that Random and Round-Robin have almost identical performance.

Figure 6(a) shows mean slowdown under TAGS-opt-slowdown as compared with the other policies. The y-axis is shown on a log scale. Observe that for very high  $\alpha$ , the performance of all the task assignment policies is

comparable and very good, however as  $\alpha$  decreases, the performance of all the policies degrades. The Least-Work-Remaining policy consistently outperforms Random by about an order of magnitude; however, TAGS-opt-slowdown offers several orders of magnitude further improvement: At  $\alpha = 1.5$ , TAGS-opt-slowdown outperforms Least-Work-Remaining by 2 orders of magnitude; at  $\alpha \approx 1$ , TAGS-opt-slowdown outperforms Least-Work-Remaining by over 4 orders of magnitude; at  $\alpha = .4$ , TAGS-opt-slowdown outperforms Least-Work-Remaining by over 9 orders of magnitude.

Figure 6(b) shows mean slowdown of TAGS-opt-fairness, as compared with the other policies. Surprisingly, the performance of TAGS-opt-fairness is not far from that of TAGS-opt-slowdown and yet TAGS-opt-fairness has the additional benefit of fairness.

Figure 7 is identical to Figure 6 except that in this case the performance metric is mean waiting time, rather than mean slowdown. Again the TAGS algorithm shows several orders of magnitude improvement over the other task assignment policies.

Why does the TAGS algorithm work so well? Intuitively, it seems that Least-Work-Remaining should be the best performer, since Least-Work-Remaining sends each job to where it will individually experience the lowest waiting time. The reason why TAGS works so well is two-fold: The first reason is *variance reduction* (Section 5.1) and the second reason is *load unbalancing* (Section 5.2).

**5.1. VARIANCE REDUCTION.** Variance reduction refers to reducing the variance of job sizes that share the same queue. Intuitively, variance reduction improves performance because it reduces the chance of a short job getting stuck behind a long job in the same queue. This is stated more formally in Theorem 1 below, which is derived from the Pollaczek–Khinchin formula.

**THEOREM 1.** *Given an  $M/G/1$  FCFS queue, where the arrival process has rate  $\lambda$ ,  $X$  denotes the service time distribution, and  $\rho$  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,*

$$\begin{aligned} \mathbf{E}\{W\} &= \frac{\lambda \mathbf{E}\{X^2\}}{2(1-\rho)} \quad \begin{array}{l} \text{(Pollaczek–Khinchin formula)} \\ \text{[Pollaczek 1930; Khinchi 1932]} \end{array} \\ \mathbf{E}\{S\} &= \mathbf{E}\left\{\frac{W}{X}\right\} = \mathbf{E}\{W\} \cdot \mathbf{E}\{X^{-1}\} \\ \mathbf{E}\{Q\} &= \lambda \mathbf{E}\{W\}. \end{aligned}$$

**PROOF.** The slowdown formula follows from the fact that  $W$  and  $X$  are independent for a FCFS queue, and the queue size follows from Little's formula.  $\square$

The above formulas apply to just a single FCFS queue, *not* a distributed server. Observe that every metric for the simple FCFS queue is dependent on  $\mathbf{E}\{X^2\}$ , the second moment of the service time. Recall that if the workload is heavy-tailed, the second moment of the service time explodes (Figure 4). We now discuss the effect of high variability in job sizes on a *distributed server* with  $h$  hosts under the various task assignment policies.

**5.1.1. Random Task Assignment.** This policy simply performs Bernoulli splitting on the input stream, with the result that each host becomes an independent

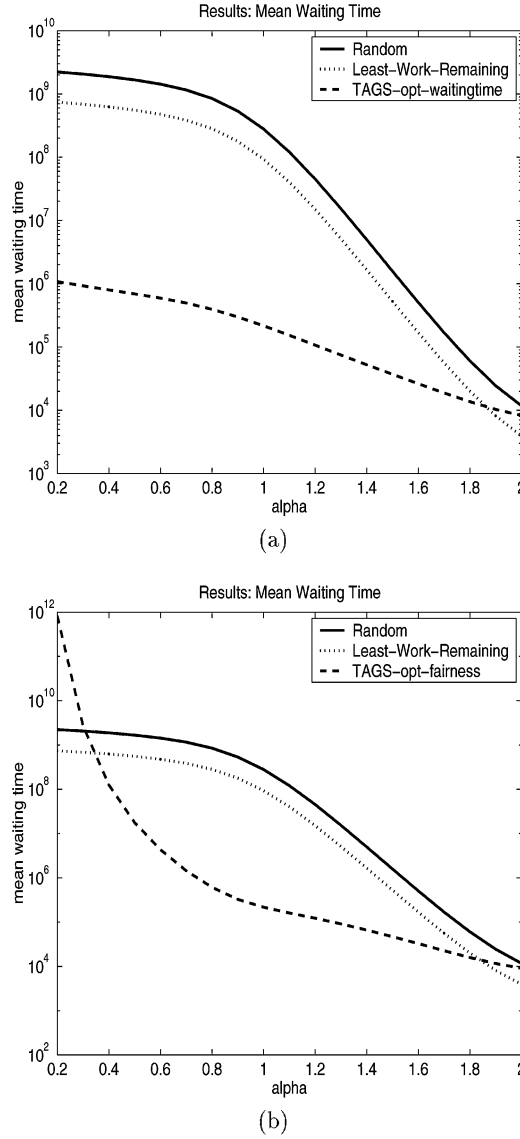


FIG. 7. Mean waiting time for distributed server with two hosts and system load .5 under (a) TAGS-opt-slowdown and (b) TAGS-opt-fairness as compared with the Least-Work-Remaining and Random task assignment policies.

$M/B(k, p, \alpha)/1$  queue. The load at the  $i$ th host,  $\rho_i$ , is equal to the system load,  $\rho$ . The arrival rate at the  $i$ th host is  $1/h$ -fraction of the total outside arrival rate. Theorem 1 applies directly, and all performance metrics are proportional to the second moment of  $B(k, p, \alpha)$ . Performance is generally poor because the second moment of the  $B(k, p, \alpha)$  is high.

**5.1.2. Round Robin.** This policy splits the incoming stream so each host sees an  $E_h/B(k, p, \alpha)/1$  queue, with utilization  $\rho_i = \rho$ , where  $E_h$  denotes an  $h$ -stage

Erlang distribution. This system has performance close to the Random policy since it still sees high variability in service times, which dominates performance.

5.1.3. *Least-Work-Remaining.* This policy is equivalent to Central-Queue, which is simply an M/G/h queue, for which there exist known approximations, [Sozaki and Ross 1978; Wolff 1989]:

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

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 mean slowdown, are all proportional to the second moment of the service time distribution, as was the case for the Random and Round-Robin policies. In fact, the performance metrics are all proportional to the squared coefficient of variation ( $C^2 = \mathbf{E}\{X^2\}/\mathbf{E}\{X\}^2$ ) of the service time distribution.

5.1.4. *TAGS.* The TAGS policy is the only policy that *reduces* the variance of job sizes at the individual hosts. Consider the jobs that queue at Host  $i$ : First, there are those jobs are destined for Host  $i$ . Their job-size distribution is  $B(s_{i-1}, s_i, \alpha)$  because the original job-size distribution is a Bounded Pareto. Then there are the jobs that are destined for hosts numbered greater than  $i$ . The service time of these jobs at Host  $i$  is capped at  $s_i$ . Thus, the second moment of the job-size distribution at Host  $i$  is lower than the second moment of the original  $B(k, p, \alpha)$  distribution (for all hosts except the highest numbered host, it turns out). The full analysis of the TAGS policy is presented in the Appendix. A sketch is given here: The initial difficulty is figuring out what to condition on, since jobs may visit multiple hosts. The solution is to partition the *jobs* based on their *final* host destination. Thus, the mean response time of the system is a linear combination of the mean response time of jobs whose final destination is Host  $i$ , where  $i = 1, \dots, h$ . The mean response time for a job whose final destination is Host  $i$  is the sum of the job's response times at Hosts 1 through  $i$ . The mean response time *at* Host  $i$  is computed via the M/G/1 formula. These computations are relatively straightforward except for one point that we have to approximate and that we explain now: For analytic convenience, we need to be able to assume that the jobs arriving at each host form a Poisson Process. This is, of course, true for Host 1. However, the arrivals at Host  $i$  are those departures from Host  $i - 1$  that exceed size  $s_{i-1}$ . They form a *less* bursty process than a Poisson Process since they are spaced apart by at least  $s_{i-1}$ . Since we make the assumption that the arrival process into Host  $i$  is a Poisson Process (that is more bursty than the actual process), our analysis if anything produces an upper bound on the response time and slowdown of TAGS. Finally, once the final expression for mean response time is derived, *Mathematica*<sup>TM</sup> is used to derive those cutoffs that minimize the expression.

5.2. *LOAD UNBALANCING.* The second reason why TAGS performs so well has to do with *load unbalancing*. Observe that all the other task assignment policies we described specifically try to *balance* load at the hosts. Random and Round-Robin balance the expected load at the hosts, while Least-Work-Remaining goes even further in trying to balance the instantaneous load at the hosts. In TAGS, we do the opposite.

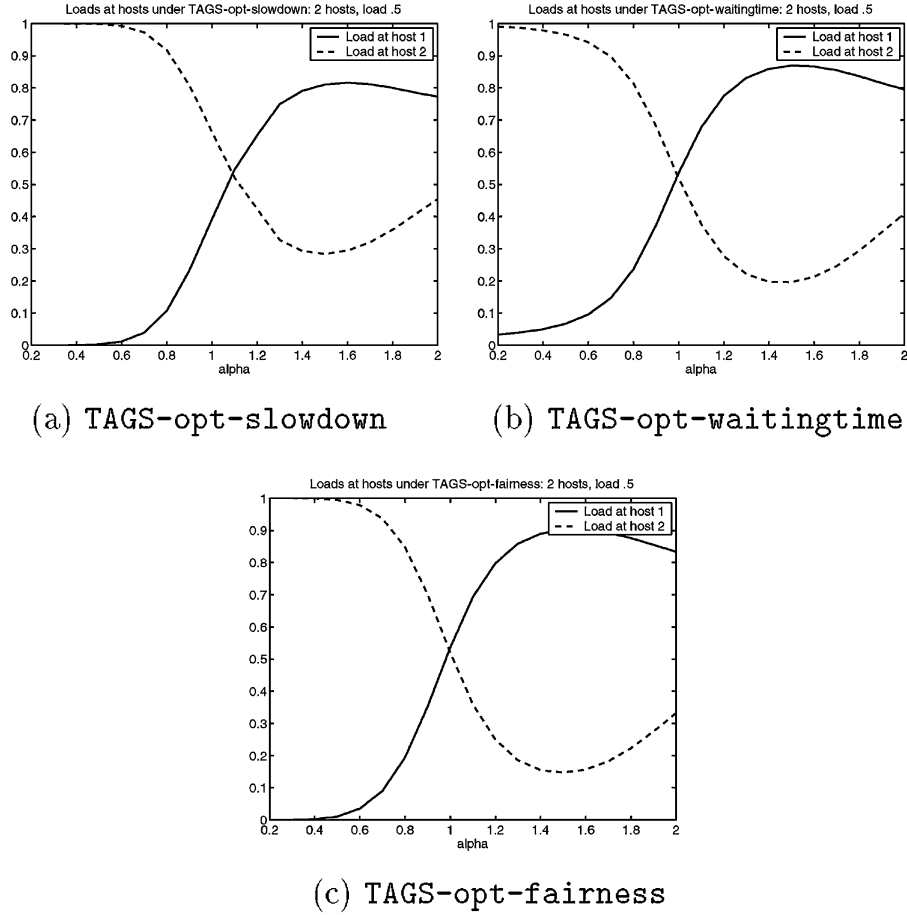


FIG. 8. Load at Host 1 as compared with Host 2 in a distributed server with two hosts and system load .5 under (a) TAGS-opt-slowdown, (b) TAGS-opt-waitingtime, and (c) TAGS-opt-fairness. Observe that for very low  $\alpha$ , Host 1 is run at load close to zero, and Host 2 is run at load close to 1, whereas for high  $\alpha$ , Host 1 is somewhat overloaded.

Figure 8 shows the load at Host 1 and at Host 2 for TAGS-opt-slowdown, TAGS-opt-waitingtime, and TAGS-opt-fairness as a function of  $\alpha$ . Observe that all three flavors of TAGS (purposely) severely underload Host 1 when  $\alpha$  is low but for higher  $\alpha$  actually overload Host 1 somewhat. In the middle range,  $\alpha \approx 1$ , the load is balanced in the two hosts.

We first explain why load unbalancing is desirable when optimizing overall mean slowdown of the system. We later explain what happens when optimizing fairness. To understand why it is desirable to operate at unbalanced loads, we need to go back to the heavy-tailed property. The heavy-tailed property says that when a distribution is very heavy-tailed (very low  $\alpha$ ), only a miniscule fraction of all jobs—the very longest ones—are needed to make up more than half the total load. As an example, for the case  $\alpha = .2$ , it turns out that the longest  $10^{-6}$  fraction of jobs alone are needed to make up half the load. In fact not many more jobs—just the longest  $10^{-4}$  fraction of all jobs—are needed to make up .99999 fraction of the load. This

suggests a load game that can be played: We choose the cutoff point ( $s_1$ ) such that *most* jobs ( $(1 - 10^{-4})$  fraction) have Host 1 as their final destination, and only a very *few* jobs (the longest  $10^{-4}$  fraction of all jobs) have Host 2 as their final destination. Because of the heavy-tailed property, the load at Host 2 will be extremely high (.99999) while the load at Host 1 will be very low (.00001). Since *most* jobs get to run at such reduced load, the overall mean slowdown is very low.

When the distribution is a little less heavy-tailed, for example,  $\alpha \approx 1$ , we can't play this load unbalancing game as well. Again, we would like to severely underload Host 1 and overload Host 2. Before, we were able to do this by sending only a very small fraction of all jobs ( $< 10^{-4}$  fraction) to Host 2. However, now that the distribution is not as heavy-tailed, a larger fraction of jobs must have Host 2 as its final destination to create high load at Host 2. But this, in turn, means that jobs with destination Host 2 count more in determining the overall mean slowdown of the system, which is bad since jobs with destination Host 2 experience larger slowdowns. Thus, we can only afford to go so far in overloading Host 2 before it turns against us.

When we get to  $\alpha > 1$ , it turns out that it actually pays to *overload* Host 1 a little. This seems counterintuitive, since Host 1 counts more in determining the overall mean slowdown of the system because most jobs have destination Host 1. However, the point is that now it is impossible to create the wonderful state where almost all jobs are on Host 1 and yet Host 1 is underloaded. The tail is just not heavy enough. No matter how we choose the cutoff, a significant portion of the jobs will have Host 2 as their destination. Thus, Host 2 will inevitably figure into the overall mean slowdown and so we need to keep the performance on Host 2 in check. To do this, it turns out we need to slightly underload Host 2, to make up for the fact that the job-size variability is so much greater on Host 2 than on Host 1.

The above has been an explanation for why load unbalancing is important with respect to optimizing the system mean slowdown. However, it is not at all clear why load unbalancing also optimizes fairness, as shown in Figure 8(c). Under TAGS-opt-fairness, 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 are treating the long jobs unfairly on three counts:

- (1) The short jobs run on Host 1 which has very low load (for low  $\alpha$ ).
- (2) The short jobs run on Host 1 which has very low  $\mathbf{E}\{X^2\}$ .
- (3) The short jobs don't have to be restarted from scratch and wait on a second line.

So how can it possibly be fair to help the short jobs so much? The answer is simply that the short jobs are short. Thus, they need low waiting times to keep their slowdown low. Long jobs, on the other hand, can afford a lot more waiting time. They are better able to amortize the punishment over their long lifetimes. It is important to mention, though, that this would not be the case for all distributions. It is because our job-size distribution for low  $\alpha$  is so heavy-tailed that the long jobs are truly *elephants* (way longer than the shorts) and thus can afford to suffer more.

**5.3. DIFFERENT LOADS.** Until now, we have studied only the model of a distributed server with two hosts and system load 0.5. In this section we consider the effect of system load on the performance of TAGS. We continue to



assume a two-host model. Figure 9 shows the performance of TAGS-opt-slowdown on a distributed server run at system load (a) 0.3, (b) 0.5, and (c) 0.7. In all three figures, TAGS-opt-slowdown improves upon the performance of Least-Work-Remaining and Random under the full range of  $\alpha$ ; however, the improvement of TAGS-opt-slowdown is much better when the system is more lightly loaded. In fact, all the policies improve as the system load is dropped; however, the improvement in TAGS is the most dramatic. In the case where the system load is 0.3, TAGS-opt-slowdown improves upon Least-Work-Remaining by over 4 orders of magnitude at  $\alpha = 1$ , by 7 orders of magnitude when  $\alpha = .6$  and by almost 20 orders of magnitude when  $\alpha = .2$ ! When the system load is 0.7, on the other hand, TAGS-opt-slowdown behaves comparably to Least-Work-Remaining for most  $\alpha$  and only improves upon Least-Work-Remaining in the narrower range of  $.6 < \alpha < 1.5$ . Notice however that, at  $\alpha \approx 1$ , the improvement of TAGS-opt-slowdown is still about 4 orders of magnitude.

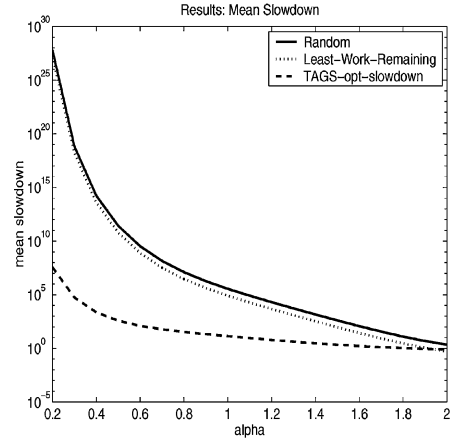
Why is the performance of TAGS so correlated with load? There are two reasons, both of which are explained by Figure 10, which shows the loads at the two hosts under TAGS-opt-slowdown in the case where the system load is (a) 0.3, (b) 0.5, and (c) 0.7.

The first reason for the ineffectiveness of TAGS under high loads is that the higher the load, the less able TAGS is to play the load-unbalancing game described in Section 5.2. For lower  $\alpha$ , TAGS reaps much of its benefit at the lower  $\alpha$  by moving all the load onto Host 2. When the system load is only 0.5, TAGS is easily able to pile all the load on Host 2 without exceeding load 1 at Host 2. However, when the system load is 0.7, the restriction that the load at Host 2 must not exceed 1 implies that Host 1 cannot be as underloaded as TAGS would like. This is seen by comparing Figure 10(b) and Figure 10(c) where in (c) the load on Host 1 is much higher for the lower  $\alpha$  than it is in (b).

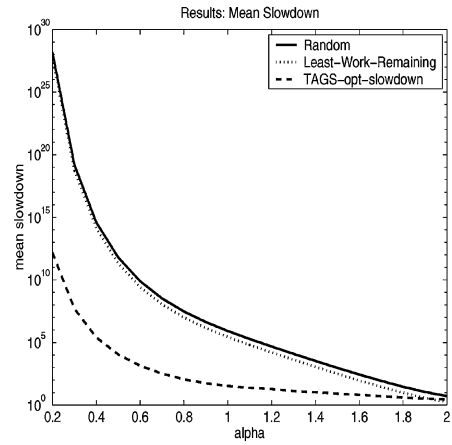
The second reason for the ineffectiveness of TAGS under high loads is due to what we call *excess*. Excess is the extra work created in TAGS by restarting jobs from scratch. In the two-host case, the excess is simply equal to  $\lambda \cdot p_2 \cdot s_1$ , where  $\lambda$  is the outside arrival rate,  $p_2$  is the fraction of jobs whose final destination is Host 2, and  $s_1$  is the cutoff differentiating short jobs from long jobs. An equivalent definition of excess is the difference between the actual sum of the loads on the hosts and  $h$  times the system load, where  $h$  is the number of hosts. The dotted line in Figures 10(a)–(c) shows the sum of the loads on the hosts.

Observe that for loads under 0.5, excess is not an issue. The reason is that for low  $\alpha$ , where we need to do the severe load unbalancing, excess is basically nonexistent for loads 0.5 and under, since  $p_2$  is so small (due to the heavy-tailed property) and since  $s_1$  could be forced down. For high  $\alpha$ , excess is present. However, all the task assignment policies already do well in the high  $\alpha$  region because of the low job size variability, so the excess is not much of a handicap.

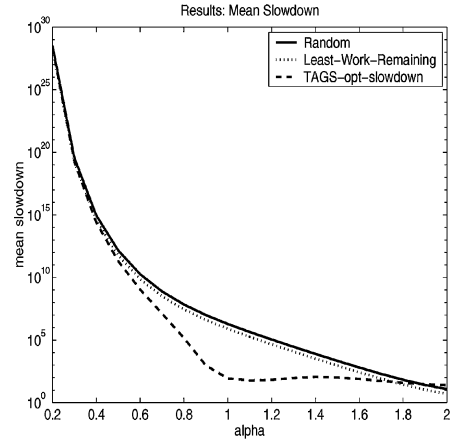
When system load exceeds 0.7, however, excess is much more of a problem, as is evidenced by the dotted line in Figure 10(c). One reason that the excess is worse is simply that overall excess increases with load because excess is proportional to  $\lambda$ , which is in turn proportional to load. The other reason that the excess is worse at higher loads has to do with  $s_1$ . In the low  $\alpha$  range, although  $p_2$  is still low (due to the heavy-tailed property),  $s_1$  cannot be forced low because the load at Host 2 is capped at 1. Thus, the excess for low  $\alpha$  is very high. In the high  $\alpha$  range, excess again is high because  $p_2$  is high.



(a) System load 0.3



(b) System load 0.5



(c) System load 0.7

FIG. 9. Mean slowdown under TAGS-opt-slowdown in a distributed server with two hosts with system load (a) 0.3, (b) 0.5, and (c) 0.7. In each figure, the mean slowdown under TAGS-opt-slowdown is compared with the performance of Random and Least-Work-Remaining. Observe that in all the figures TAGS outperforms the other policies under all  $\alpha$ . However, TAGS is most effective at lower system loads.

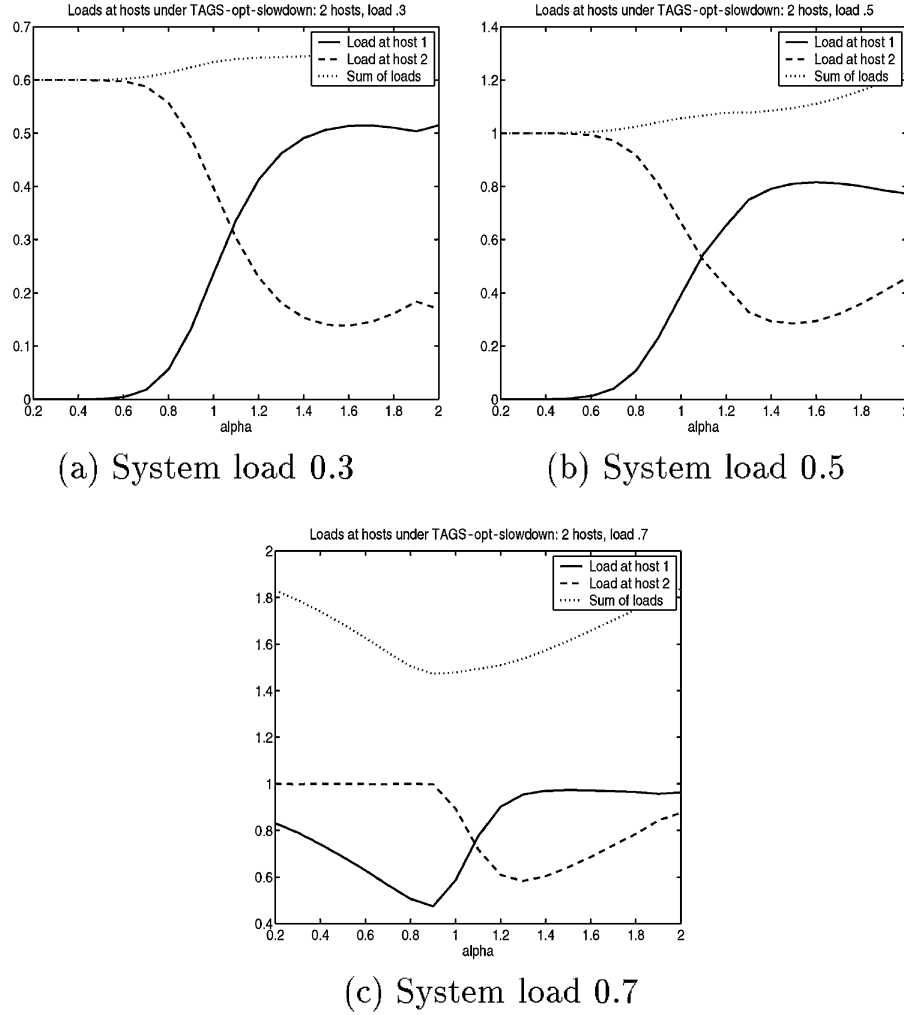


FIG. 10. Load at Host 1 and Host 2 under TAGS-opt-slowdown shown for a distributed server with two hosts and system load (a) 0.3 (b) 0.5 (c) 0.7. The dotted line shows the sum of the loads at the two hosts. If there were no excess, the dotted line would be at (a) 0.6 (b) 1.0 and (c) 1.4 in each of the graphs respectively. In figures (a) and (b), we see excess only at the higher  $\alpha$  range. In figure (c), we see excess in both the low  $\alpha$  and high  $\alpha$  range, but not around  $\alpha \approx 1$ .

Fortunately, observe that for higher loads excess is at its lowest point at  $\alpha \approx 1$ . In fact, it is barely existent in this region. Observe also that the  $\alpha \approx 1$  region is the region where balancing load is the optimal thing to do (with respect to minimizing mean slowdown), regardless of the system load. This “sweet spot” is fortunate because  $\alpha \approx 1$  is characteristic of many empirically measured computer workloads, see Section 3.

#### 6. Analytic Results for Case of More than Two Hosts

Until now, we have only considered distributed servers with two hosts. For two hosts, we saw that the performance of TAGS-opt-slowdown was amazingly good

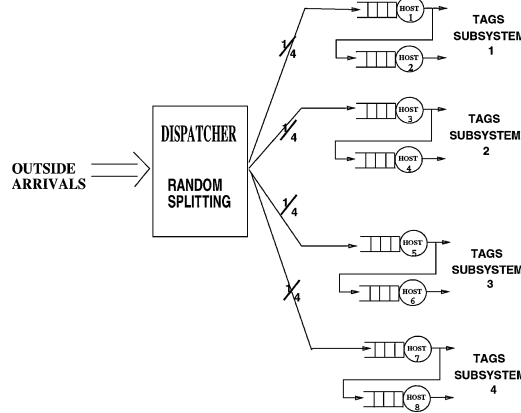


FIG. 11. Illustration of the claim that an  $h$  host system ( $h > 2$ ) with system load  $\rho$  can always be configured to produce performance at least as good as a two-host system with system load  $\rho$  (although the  $h$ -host system has much higher arrival rate).

if the system load was 0.5 or less, but not nearly as good for system load  $> 0.5$ . In this section, we consider the case of more than two hosts.<sup>7</sup>

One claim that can be made straight off is that an  $h$  host system ( $h > 2$ ) with system load  $\rho$  can always be configured to produce performance that is *at least as good* as the best performance of a two-host system with system load  $\rho$ . To see why, observe that we can use the  $h$  host system (assuming  $h$  is even) to simulate a two-host system as illustrated in Figure 11: Rename Hosts 1 and 2 as Subsystem 1. Rename Hosts 3 and 4 as Subsystem 2. Rename Hosts 5 and 6 as Subsystem 3, etc. Now split the traffic entering the  $h$  host system so that  $2/h$  fraction of the jobs go to each of the  $h/2$  subsystems. Now apply the best known task assignment policy to each subsystem independently—in our case we choose TAGS. Each subsystem will behave like a two-host system with load  $\rho$  running TAGS. Since each subsystem will have identical performance, the performance of the whole  $h$  host system will be equal to the performance of any one subsystem. (Observe that the above argument works for any task assignment policy).

However, the performance of a distributed server with  $h > 2$  hosts and system load  $\rho$  is often much superior to that of a distributed server with two hosts and system load  $\rho$ . Figure 12 shows the mean slowdown under TAGS-opt-slowdown for the case of a four-host distributed server with system load 0.3. Comparing these results to those for the two-host system with system load 0.3 (Figure 9(a)), we see that:

- (1) The performance of Random stayed the same, as it should.
- (2) The performance of Least-Work-Remaining improved by a couple orders of magnitude in the higher  $\alpha$  region, but less in the lower  $\alpha$  region. The

<sup>7</sup> The phrase “adding more hosts” can be ambiguous because it is not clear whether the arrival rate is increased as well. For example, given a system with two hosts and system load 0.7, we could increase the number of hosts to four hosts *without* changing the arrival rate, and the system load would drop to 0.35. On the other hand, we could increase the number of hosts to four hosts and increase the arrival rate appropriately (double it) so as to maintain a system load of 0.7. In our discussions below, we attempt to be clear as to which view we have in mind.

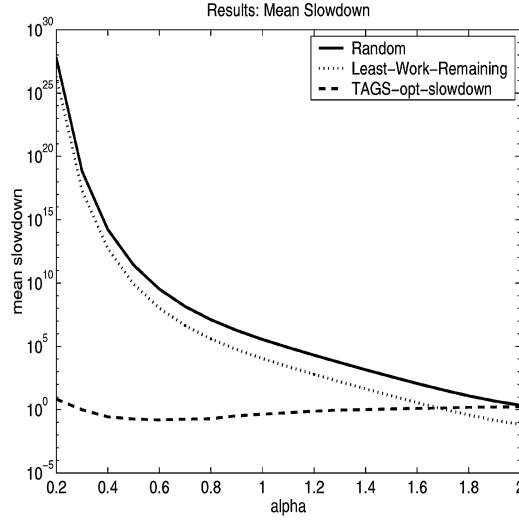


FIG. 12. Mean slowdown under TAGS-opt-slowdown compared with other policies in the case of a distributed server with four hosts and system load 0.3. The cutoffs for TAGS-opt-slowdown were optimized by hand. In many cases, it is possible to improve upon the results shown here by adjusting the cutoffs further, so the slight bend in the graph may not be meaningful. Observe that the mean slowdown of TAGS almost never exceeds 6.

Least-Work-Remaining policy is helped by increasing the number of hosts, although the system load stayed the same, because having more hosts increases the chances of one of them being free.

- (3) The performance of TAGS-opt-slowdown improved a lot. So much so, that the mean slowdown under TAGS-opt-slowdown is *never* over 6 and often under 1. At  $\alpha \approx 1$ , TAGS-opt-slowdown improves upon Least-Work-Remaining by 4–5 orders of magnitude. At  $\alpha = .6$ , the improvement increases to 8 or 9 orders of magnitude. At  $\alpha = .2$ , TAGS-opt-slowdown improves upon Least-Work-Remaining by over 25 orders of magnitude.

The enhanced performance of TAGS on more hosts may come from the fact that more hosts allow for greater flexibility in choosing the cutoffs. However, it is hard to say for sure because it is difficult to compute results for the case of more than two hosts. The cutoffs in the case of two hosts were all optimized by *Mathematica*<sup>TM</sup>, while in the case of four hosts, it was necessary to perform the optimizations by hand. For the case of system load 0.7 with four hosts, we ran into the same type of problems as we did for the two-host case with system load 0.7.

**6.1. THE SERVER EXPANSION PERFORMANCE METRIC.** There is one thing that seems very artificial about our current comparison of task assignment policies. No one would ever run a system with a mean of slowdown  $10^5$ . In practice, if a system was operating with mean slowdown of  $10^5$ , the number of hosts would be increased, without increasing the arrival rate, thus dropping the system load, until the system's performance improved to a reasonable mean slowdown, say 3. Consider the following example: Suppose we have a two-host system running

at system load .7 and with variability parameter  $\alpha = .6$ . For this system, the mean slowdown under TAGS-opt-slowdown is  $10^9$ , and no other policy that we know of does better. Suppose, however, we desire a system with mean slowdown under 3. So we double the number of hosts (without increasing the outside arrival rate). At four hosts, with system load 0.35, TAGS-opt-slowdown now has mean slowdown of around 1, whereas Least-Work-Remaining's slowdown has improved to  $10^8$ . It turns out we would have to increase number of hosts to 13 for the performance of Least-Work-Remaining to improve to the point of mean slowdown under 3. And for Random to reach that level it would require an additional  $10^9$  hosts.

The above example suggests a new practical performance metric for distributed servers, which we call the *server expansion metric*. The server expansion metric asks how many additional hosts must be added to the existing server (without increasing outside arrival rate) to bring mean slowdown down to a reasonable level (where we will arbitrarily define "reasonable" as slowdown of 3 or less). Figure 13 compares the performance of our policies according to the server expansion metric, given that we start with a two-host system with system load of 0.7. For TAGS-opt-slowdown, the server expansion is only 3 for  $\alpha = .2$  and no more than 2 for all the other  $\alpha$ . For Least-Work-Remaining, on the other hand, the server expansion ranges from 1 to 27, as  $\alpha$  decreases. Still Least-Work-Remaining is not so bad because at least its performance improves somewhat quickly as hosts are added and load is decreased, the reason being that both these effects increase the probability of a job finding an idle host. By contrast Random, shown in Figure 13(b), is exponentially worse than the others, requiring as many as  $10^5$  additional hosts when  $\alpha \approx 1$ . Although Random does benefit from increasing the number of hosts, the effect isn't nearly as strong as it is for TAGS and Least-Work-Remaining.

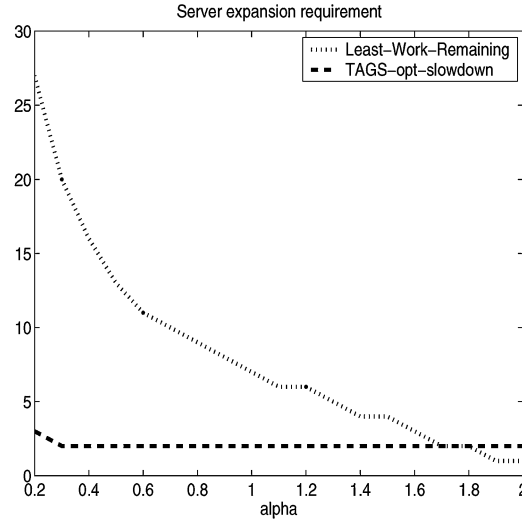
### 7. The Effect of the Range of Task Sizes

The purpose of this section is to investigate what happens when the *range* of job sizes is smaller than we have heretofore assumed, resulting in a smaller coefficient of variation in the job-size distribution.

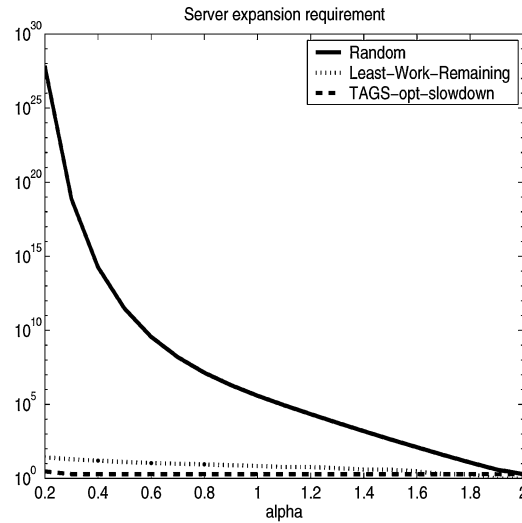
Until now, we have always assumed that the job sizes are distributed according to a Bounded Pareto distribution with upper bound  $p = 10^{10}$  and fixed mean 3000. This means, for example, that when  $\alpha \approx 1$ , we need to set the lower bound on job sizes to  $k = 167$ . However, this implies that the range of job sizes spans 8 orders of magnitude.

It is not clear that all applications have job sizes ranging 8 orders in magnitude. In this section, we rederive the performance of all the task assignment policies when the upper bound  $p$  is set to  $p = 10^7$ , while still holding the mean of the job-size distribution at 3000. This means, for example, that when  $\alpha \approx 1$  (as agrees with empirical data), we need to set the lower bound on job sizes to  $k = 287$ , which implies the range of job sizes spans just 5 orders of magnitude. Figure 14 shows the second moment of the Bounded Pareto job size distribution as a function of  $\alpha$  when  $p = 10^7$ . Comparing this figure to Figure 4, we see that the job-size variability is far lower when  $p = 10^7$ .

Lower variance in the job-size distribution suggests that the improvement of TAGS over the other assignment policies will not be as dramatic as in the higher variability



(a) Non-log scale



(b) Log scale

FIG. 13. Server expansion requirement for each of the task assignment policies, given that we start with a two-host system with system load of 0.7. (a) Shows just Least-Work-Remaining and TAGS-opt-slowdown on a non-log scale (b) Shows Least-Work-Remaining, TAGS-opt-slowdown, and Random on a log scale.

setting (when  $p = 10^{10}$ ). This is, in fact, the case. What is interesting, however, is that, even in this lower variability setting, the improvement of TAGS over the other policies is still impressive, as shown in Figure 15. Figure 15 shows the mean slowdown of TAGS-opt-slowdown as compared with Random and Least-Work-Left for the case of two hosts with system load 0.5. Observe that for  $\alpha \approx 1$ , TAGS improves upon the other policies by over 2 orders of magnitude. As  $\alpha$  drops, the

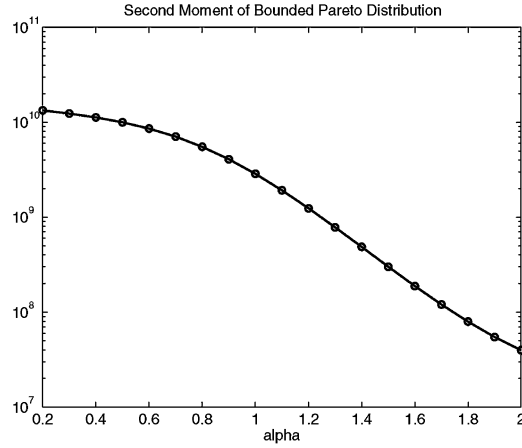


FIG. 14. Second moment of  $B(k, p, \alpha)$  distribution, where now the upper bound,  $p$ , is set at  $p = 10^7$ , rather than  $10^{10}$ . The mean is held fixed at 3000 as  $\alpha$  is varied. Observe that the coefficient of variation now ranges from 2 (when  $\alpha = 2$ ) to 33 (when  $\alpha = .2$ ).

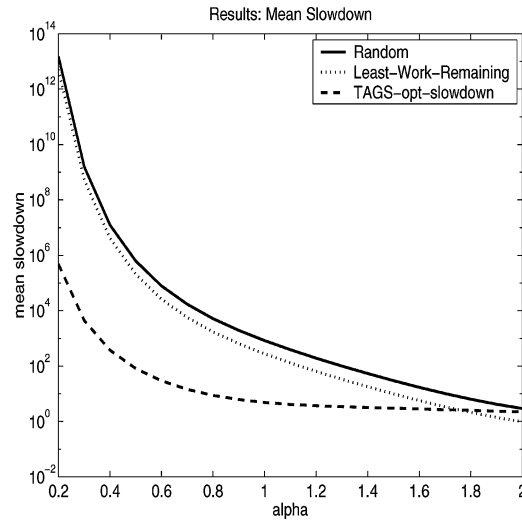


FIG. 15. Mean slowdown under TAGS-opt-slowdown in a distributed server with two hosts with system load 0.5, as compared with Random and Least-Work-Remaining. In this set of results, the job size distribution is  $B(k, p, \alpha)$ , where  $p = 10^7$ .

improvement increases. This figure should be contrasted with Figure 6(a), which shows the same scenario where  $p = 10^{10}$ .

## 8. Conclusion

This article is interesting not only because it proposes a powerful new task assignment policy, but more so because it challenges some natural intuitions which we have come to adopt over time as common knowledge.



Traditionally, the area of task assignment, load balancing and load sharing has consisted of heuristics that seek to balance the load among the multiple hosts. TAGS, on the other hand, specifically seeks to unbalance the load, and sometimes severely unbalance the load. Traditionally, the idea of killing a job and restarting it from scratch on a different machine is viewed with skepticism, but possibly tolerable if the new host is idle. TAGS, on the other hand, kills jobs and then restarts them from scratch at a target host which is typically operating at extremely high load, much higher load than the original source host. Furthermore, TAGS proposes restarting the same job multiple times. Traditionally optimal performance and fairness are viewed as conflicting goals. In TAGS, fairness and optimality are surprisingly close.

It is interesting to consider further implications of these results, outside the scope of task assignment. Consider, for example, the question of scheduling CPU-bound jobs on a single CPU, where jobs are not preemptible and no a priori knowledge is given about the jobs. At first, it seems that FCFS scheduling is the only option. However, in the face of high job-size variability, FCFS may not be wise. This article suggests that killing and restarting jobs may be worth investigating as an alternative, if the load on the CPU is low enough to tolerate the extra work created.

This work may also have implications in the area of network flow routing. A very interesting recent paper by Shaikh et al. [1999] takes a first step in this direction. The article discusses routing of IP flows (which also have heavy-tailed size distributions) and recommends routing long flows differently from short flows.

## Appendix

This section provides the analysis of the TAGS policy. Throughout this discussion it will be necessary to refer to Table II to understand the notation.

We start with some properties of the original distribution of job sizes  $B(k, p, \alpha)$ :

$$f(x) = \frac{\alpha k^\alpha}{1 - (k/p)^\alpha} x^{-\alpha-1} \quad k \leq x \leq p$$

$$\mathbf{E}\{X^j\} = \int_k^p f(x) \cdot x^j dx = \begin{cases} \frac{\alpha k^\alpha (k^{j-\alpha} - p^{j-\alpha})}{(\alpha-j)(1-(k/p)^\alpha)} & \text{if } \alpha \neq j \\ \frac{k}{1-(k/p)} \cdot (\ln p - \ln k) & \text{if } \alpha = j = 1 \end{cases}$$

$$\lambda = \frac{1}{\mathbf{E}\{X\}} \cdot h \cdot \rho.$$

Let  $p_i$  denote the fraction of jobs whose final destination is Host  $i$  and  $p_i^{\text{visit}}$  denote the fraction of jobs which ever visit Host  $i$ .

$$p_i = \int_{s_{i-1}}^{s_i} f(x) dx = \frac{k^\alpha}{1 - (k/p)^\alpha} (s_{i-1}^{-\alpha} - s_i^{-\alpha})$$

$$p_i^{\text{visit}} = \sum_{j=i}^h p_j$$

TABLE II. NOTATION FOR ANALYSIS OF TAGS

$h$	Number of hosts
$B(k, p, \alpha)$	Job size distribution
$p$	Upper bound on job size distribution
$k$	Lower bound on job size distribution
$f(x)$	Probability density function for $B(k, p, \alpha)$ .
$\alpha$	Heavy-tailed parameter
$s_0, s_1, \dots, s_h$	Job size cutoffs
$s_i$	Upper bound on job size seen by Host $i$
$\lambda$	Outside arrival rate into system
$\rho$	System load
$\rho_i^{visit}$	Load at Host $i$
$p_i$	Fraction of jobs whose final destination is Host $i$ , i.e., whose size is between $s_{i-1}$ and $s_i$ .
$p_i^{visit}$	Fraction of jobs which spend time at Host $i$
$\lambda_i^{visit}$	Arrival rate into Host $i$
$\mathbf{E}\{X\}$	Mean job size under $B(k, p, \alpha)$ distribution
$\mathbf{E}\{X^j\}$	$j$ th moment of job size distribution $B(k, p, \alpha)$
$\mathbf{E}\{X_i\}$	Expected size of jobs whose final destination is Host $i$ .
$\mathbf{E}\{X_i^{visit}\}$	Expected size of jobs which spend time at Host $i$
$\mathbf{E}\{X_i^2\}$	Second moment of size of jobs whose final destination is Host $i$ .
$\mathbf{E}\{X_i^j\}$	$j$ th moment of size of jobs whose final destination is Host $i$ .
$\mathbf{E}\{X_i^{2(visit)}\}$	Second moment of size of jobs which spend time at Host $i$
$\mathbf{E}\{X_i^{j(visit)}\}$	$j$ th moment of size of jobs which spend time at Host $i$
$\mathbf{E}\{1/X_i\}$	Expected 1/size of jobs whose final destination is Host $i$
$\mathbf{E}\{W_i^{visit}\}$	Expected waiting time at Host $i$
$\mathbf{E}\{W_i\}$	Total expected waiting time for jobs with final destination Host $i$
$\mathbf{E}\{S_i\}$	Expected slowdown for jobs with final destination Host $i$
$\mathbf{E}\{W\}$	Expected waiting time for jobs under TAGS
$\mathbf{E}\{S\}$	Expected slowdown for jobs under TAGS
$Excess$	Total excess work being done

Now consider those jobs whose *final destination* is Host  $i$ . Observe that since the original distribution is Bounded Pareto  $B(k, p, \alpha)$ , then the distribution of jobs whose final destination is Host  $i$  is also a Bounded Pareto  $B(s_{i-1}, s_i, \alpha)$ . This makes it easy to compute  $\mathbf{E}\{X_i^j\}$ , the  $j$ th moment of the distribution of jobs whose final destination is Host  $i$ :

$$\mathbf{E}\{X_i^j\} = \int_{s_{i-1}}^{s_i} x^j \frac{f(x)}{p_i} dx = \begin{cases} \frac{\alpha s_{i-1}^\alpha (s_{i-1}^{j-\alpha} - s_i^{j-\alpha})}{(\alpha-j)(1-(s_{i-1}/s_i)^\alpha)} & \text{if } \alpha \neq j \\ \frac{s_{i-1}s_i}{s_i - s_{i-1}} (\ln s_i - \ln s_{i-1}) & \text{if } \alpha = j = 1 \\ \frac{\alpha s_{i-1}^\alpha}{(1-(s_{i-1}/s_i)^\alpha)} \cdot (\ln s_i - \ln s_{i-1}) & \text{if } \alpha = j = 2. \end{cases}$$

Now consider all jobs which *visit* Host  $i$ . These include the  $p_i$  fraction of all jobs which have Host  $i$  as their final destination. However these also include the  $p_i^{visit} - p_i$  fraction of all jobs which have Host  $j$  as their final destination, where  $j > i$ . Those jobs which have Host  $j$ ,  $j > i$ , as their final destination will only

have a service requirement of  $s_i$  at Host  $i$ . Thus, it follows that:

$$\begin{aligned}\mathbf{E}\{X_i^{visit}\} &= \frac{p_i}{p_i^{visit}} \cdot \mathbf{E}\{X_i\} + \frac{p_i^{visit} - p_i}{p_i^{visit}} \cdot s_i \\ \mathbf{E}\{X_i^{2(visit)}\} &= \frac{p_i}{p_i^{visit}} \cdot \mathbf{E}\{X_i^2\} + \frac{p_i^{visit} - p_i}{p_i^{visit}} \cdot s_i^2 \\ \lambda_i^{visit} &= \lambda \cdot p_i^{visit} \\ \rho_i^{visit} &= \lambda_i^{visit} \cdot \mathbf{E}\{X_i^{visit}\} \\ \mathbf{E}\{1/X_i^j\} &= \mathbf{E}\{X_i^{-j}\}\end{aligned}$$

There are two equivalent ways of defining excess. We show both below and check them against each other in our computations.

$$\begin{aligned}\text{true-sum-of-loads} &= \sum_{i=1}^h \rho_i^{visit} \\ \text{desired-sum-of-loads} &= h \cdot \rho \\ \text{Excess}_a &= \text{true-sum-of-loads} - \text{desired-sum-of-loads} \\ \text{Excess}_b &= \sum_{i=2}^h \lambda_i^{visit} \cdot s_{i-1} \\ \text{Excess} &= \text{Excess}_a = \text{Excess}_b\end{aligned}$$

Computing mean waiting time and mean slowdown follows from Theorem 1, except for one approximation, as explained earlier in the text: we will assume that the arrival process into each host is a Poisson Process. Observe that in computing mean slowdown, we have to be careful about which jobs we're averaging over. The calculation works out most easily if we condition on the final destination of the job, as shown below.

$$\begin{aligned}\mathbf{E}\{W_i^{visit}\} &= \lambda_i^{visit} \cdot \mathbf{E}\{X_i^{2(visit)}\} / (2(1 - \rho_i^{visit})) \\ \mathbf{E}\{W_i\} &= \sum_{j=1}^i \mathbf{E}\{W_j^{visit}\} \\ \mathbf{E}\{W\} &= \sum_{i=1}^h \mathbf{E}\{W_i\} \cdot p_i \\ \mathbf{E}\{S_i\} &= \mathbf{E}\{W_i\} \cdot \mathbf{E}\{1/X_i\} \\ \mathbf{E}\{S\} &= \sum_{i=1}^h \mathbf{E}\{S_i\} \cdot p_i\end{aligned}$$

All the formulas above assume knowledge of the cutoff points  $s_0, s_1, \dots, s_h$ . To determine these cutoff points, we feed all of the above formulas into *Mathematica*<sup>TM</sup>, leaving the  $s_i$ 's as undetermined variables. We then solve for

the optimal setting of the  $s_i$ 's which minimizes the mean slowdown, mean waiting time, or fairness, as desired, subject to conditions that the load at each host stays below 1.

## REFERENCES

- BESTAVROS, A. 1997. Load profiling: A methodology for scheduling real-time tasks in a distributed system. In *Proceedings of ICDCS '97* (May).
- CROVELLA, M. E., AND BESTAVROS, A. 1997. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Trans. Netw.* 5, 6 (Dec.), 835–846.
- CROVELLA, M. E., HARCHOL-BALTER, M., AND MURTA, C. 1998a. Task assignment in a distributed system: Improving performance by unbalancing load. In *Proceeding of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems Poster Session*. ACM, New York.
- CROVELLA, M. E., TAQQU, M. S., AND BESTAVROS, A. 1998b. Heavy-tailed probability distributions in the World Wide Web. In *A Practical Guide to Heavy Tails*. Chapman & Hall, New York, chap. 1, pp. 1–23.
- DOWNEY, A. B. 1997. A parallel workload model and its implications for processor allocation. In *Proceedings of High Performance Distributed Computing* (Aug.). pp. 112–123.
- EPHREMIDES, A., VARAIYA, P., AND WALRAND, J. 1980. A simple dynamic routing problem. *IEEE Trans. Automat. Cont.* AC-25, 4, 690–693.
- FEITELSON, D., AND JETTE, M. A. 1997. Improved utilization and responsiveness with gang scheduling. In *Proceedings of IPPS/SPDP '97 Workshop* (Apr.). Lecture Notes in Computer Science, vol. 1291. Springer-Verlag, New York, pp. 238–261.
- FEITELSON, D., RUDOLPH, L., SCHWIEGELSHOHN, U., SEVCIK, K., AND WONG, P. 1997. Theory and practice in parallel job scheduling. In *Proceedings of IPPS/SPDP '97 Workshop* (Apr.). Lecture Notes in Computer Science, vol. 1291. Springer-Verlag, New York, pp. 1–34.
- HARCHOL-BALTER, M., CROVELLA, M., AND MURTA, C. 1999. On choosing a task assignment policy for a distributed server system. *J. Paral. Distr. Comput.* 59, 204–228.
- HARCHOL-BALTER, M., AND DOWNEY, A. 1997. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.* 15, 3.
- IRLAM, G. 1994. Unix file size survey—1993. Available at <http://www.base.com/gordon/ufs93.html>.
- KHINCHIN, A. Y. 1932. Mathematical theory of stationary queues. *Mat. Sbornik* 39, 73–84.
- KOOLE, G., SPARAGGIS, P., AND TOWSLEY, D. 1999. Minimizing response times and queue lengths in systems of parallel queues. *J. Appl. Prob.* 36, 1185–1193.
- LEISERSON, C. 1998a. The Pleiades alpha cluster at M.I.T. Documentation at: <http://bonanza.lcs.mit.edu>.
- LEISERSON, C. 1998b. The Xolas supercomputing project at M.I.T. Documentation available at: <http://xolas.lcs.mit.edu>.
- LELAND, W. E., AND OTT, T. J. 1986. Load-balancing heuristics and process behavior. In *Proceedings of Performance and ACM SIGMETRICS*. ACM, New York, pp. 54–69.
- NELSON, R. D., AND PHILIPS, T. K. 1989. An approximation to the response time for shortest queue routing. *Perf. Eval. Rev.* 7, 1, 181–189.
- NELSON, R. D., AND PHILIPS, T. K. 1993. An approximation for the mean response time for shortest queue routing with general interarrival and service times. *Perf. Eval.* 17, 123–139.
- PARSONS, E. W., AND SEVCIK, K. C. 1997. Implementing multiprocessor scheduling disciplines. In *Proceedings of IPPS/SPDP '97 Workshop* (Apr.). Lecture Notes in Computer Science, vol. 1459. Springer-Verlag, New York, pp. 166–182.
- PAXSON, V., AND FLOYD, S. 1995. Wide-area traffic: The failure of Poisson modeling. *IEEE/ACM Trans. Netw.* (June), 226–244.
- PETERSON, D. L., AND ADAMS, D. R. 1996. Fractal patterns in DASD I/O traffic. In *CMG Proc.*
- POLLACZEK, F. 1930. Über eine aufgabe der wahrscheinlichkeitstheorie. *I-II Math. Zeitschrift.* 32, 64–100.
- THE PSC'S CRAY J90's. 1998. <http://www.psc.edu/machine/cray/j90/j90.html>.
- Supercomputing at the NAS facility. 1998. <http://www.nas.nasa.gov/Technology/Supercomputing/>.
- RUDOLPH, L., AND SMITH, P. H. 2000. Valuation of ultra-scale computing systems. In *Proceedings of the 6th Workshop on Job Scheduling Strategies for Parallel Processing* (Cancun, Mexico, May).

- Lecture Notes in Computer Science, vol. 1911. Springer-Verlag, New York, <http://www.cs.huji.ac.il/~feit/parsched/parsched00.html>.
- SCHROEDER, B., AND HARCHOL-BALTER, M. 2000. Evaluation of task assignment policies for super-computing servers: The case for load unbalancing and fairness. In *Proceedings of the 9th IEEE Symposium on High Performance Distributed Computing* (Aug.) IEEE Computer Society Press, Los Alamitos, Calif.
- SHAIKH, A., REXFORD, J., AND SHIN, K. G. 1999. Load-sensitive routing of long-lived ip flows. In *Proceedings of SIGCOMM* (Sept.). ACM, New York.
- SOZAKI, S., AND ROSS, R. 1978. Approximations in finite capacity multiserver queues with poisson arrivals. *J. Appl. Prob.* 13, 826–834.
- WHITT, W. 1986. Deciding which queue to join: Some counterexamples. *Oper. Res.* 34, 1 (Jan.). 226–244.
- WINSTON, W. 1977. Optimality of the shortest line discipline. *J. Appl. Prob.* 14, 181–189.
- WOLFF, R. W. 1989. *Stochastic Modeling and the Theory of Queues*. Prentice-Hall, Englewood Cliffs, N.J.

RECEIVED MARCH 2000; REVISED JANUARY 2001; ACCEPTED JANUARY 2002