Task Assignment with Unknown Duration

Mor Harchol-Balter

August 1999

CMU-CS-99-162

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We consider a distributed server system and ask which policy should be used for assigning tasks to hosts. In our server, tasks are *not* preemptible. Also, the task'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 *un*balancing, *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 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.

Keywords: Task assignment, load sharing, load balancing, scheduling, heavy-tailed workloads, high variance, distributed servers, fairness, contrary behavior.

1 Introduction

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

In a distributed server system, requests for service (tasks) arrive and must be assigned to one of the host machines for processing. The rule for assigning tasks to host machines is known as the *task assignment policy.* The choice of the task assignment policy 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 which still remains open for many models.

In this paper we consider the particular model of a distributed server system in which tasks are *not preemptible* – i.e. we are concerned with applications where context switches are too costly. For example, one such application is batch computing environments where the hosts themselves are parallel processors and the tasks are parallel. Context switching between tasks involves reloading all the processors and memory to return them to the state before the context switch. Because context switching is so expensive in this environment, tasks are always simply run to completion. Note, the fact that context switches are too expensive does not preclude the possibility of killing a job and restarting it from scratch.

We assume furthermore that *no a priori information* is known about the task at the time when the task arrives. In particular, the service demand of the task is not known. We assume all hosts are identical and there is no cost (time required) for assigning tasks to hosts. Figure 1 is one illustration of a distributed server. In this illustration, arriving tasks 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 tasks might wait before being dispatched. It also does not preclude the possibility of an alternative scheduling discipline at the hosts, so long as that scheduling discipline does not require preempting tasks and does not rely on a priori knowledge about tasks.

Our main performance goal, in choosing a task assignment policy, is to minimize *mean waiting time* and more importantly *mean slowdown.* A task's slowdown is its waiting time divided by its service demand. All means are per-task averages. We consider mean slowdown to be more important than mean waiting time because it is desirable that a task's delay be proportional to its size. That is, in a system in which task sizes are highly variable, users are likely to anticipate short delays for short tasks, and are likely to tolerate long delays for longer tasks. Later in the paper we introduce a new performance metric, called *server expansion* which is related to mean slowdown. A secondary performance goal is fairness. We adopt the standard definition of fairness that says all tasks, large or small, should experience the same expected slowdown. In particular, large tasks shouldn't be penalized – slowed down by a greater factor than are small tasks.[1]

Consider some task assignment policies commonly proposed for distributed server systems: In the `Random` task assignment policy, an incoming task is sent to Host i with probability $1/h$, where h is the number of hosts. This policy equalizes the expected number of tasks at each host. In `Round-Robin` task assignment, tasks are assigned to hosts in a cyclical fashion with the ith task being assigned to Host $i \bmod h$. This policy also equalizes the expected number of tasks at each host, and has slightly less variability in interarrival times than does `Random`. In `Shortest-Queue` task assignment, an incoming task is immediately dispatched to the host with the fewest number of tasks. This policy has the benefit of trying to equalize the instantaneous number of tasks at each host, rather than just the expected number of tasks. All the above policies have the property that the tasks arriving at each host are serviced in FCFS order.

The literature tells us that `Shortest-Queue` is in fact the best task assignment policy in a model where the following conditions are met: (1) there is no a priori knowledge about tasks, (2) tasks are not preemptible, (3) each host services tasks in a FCFS order, (4) incoming tasks are immediately dispatched to a host, and (5) the task size distribution is Exponential (see Section 2).

If one removes restriction (4), it is possible to do even better. What we'd really like to do is send a task to the host which has the least total outstanding work (work is the sum of the task sizes at the host) because that host would afford the task the smallest waiting time. However, we don't know a priori which host currently has the least work, since we don't know task sizes. It turns out this is actually easy to get around: we simply hold all tasks at the dispatcher in a FCFS queue, and only when a host is free does it request the next task. It is easy to prove that this holding method is exactly equivalent to immediately dispatching arriving tasks to the host with least outstanding work (see [6] for a proof and Figure 2 for an illustration). We will refer to this policy as `Least-Work-Remaining` since it has the effect of sending each task to the host with the currently least remaining work. Observe that `Least-Work-Remaining` comes closest to obtaining instantaneous load balance.

It may seem that `Least-Work-Remaining` is the best possible task assignment policy. Previous literature shows that `Least-Work-Remaining` outperforms all of the above previously-discussed policies under very general conditions (see Section 2). Previous literature also suggests that `Least-Work-Remaining`

[1] For example, Processor-Sharing (which requires infinitely-many preemptions) is ultimately fair in that every task experiences the same expected slowdown.

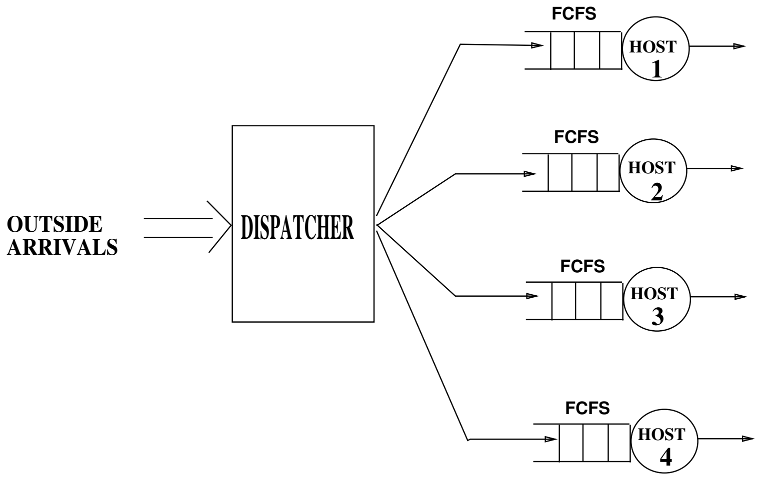


Figure 1: *Illustration of a distributed server.*

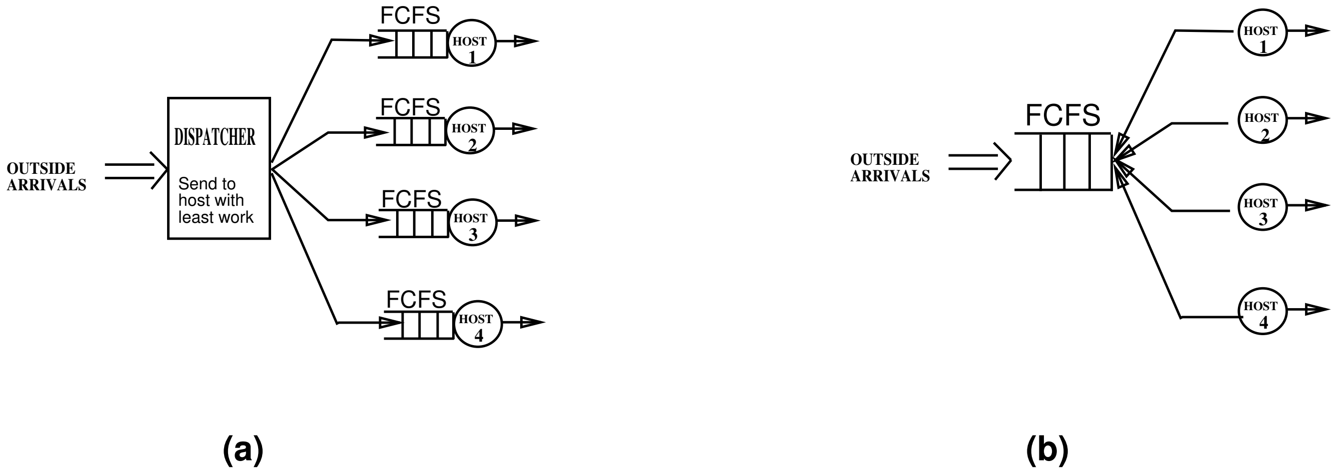


Figure 2: *Two equivalent ways of implementing the* `Least-Work-Remaining` *task assignment policy. (a) Shows incoming tasks immediately being dispatched to the host with the least remaining work, but this requires knowing a priori the sizes of the tasks at the hosts. (b) Shows incoming tasks pooled at a FCFS queue at the dispatcher. There are no queues at the individual hosts. Only when a host is free does it request the next task. This implementation does not require a priori knowledge of the task sizes, yet achieves the same effect as (a).*

may be the optimal (best possible) task assignment policy in the case where the task size distribution is *Exponential* (see Section 2 for a detailed statement of the previous literature).

But what if task size distribution is not Exponential? We are motivated in this respect by the increasing evidence for high variability in task 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 task assignment policy than `Least-Work-Remaining` when the task size variability is characteristic of empirical workloads? In evaluating various task assignment policies, we will be interested in understanding the influence of task size variability on the decision of which task assignment policy is best. For analytical tractability, we will assume that the arrival process is Poisson – our simulations indicate that the variability in the arrival process is much less critical to choosing a task assignment policy than is the variability in the task size distribution.

In this paper we propose a new algorithm called `TAGS` – Task Assignment by Guessing Size which is specifically designed for high variability workloads. We will prove analytically that when task sizes show the degree of variability characteristic of empirical (measured) workloads, the `TAGS` algorithm can outperform all the above mentioned algorithms by several orders of magnitude. In fact, we will show that the more heavy-tailed the task size distribution, the greater the improvement of `TAGS` over the other task assignment algorithms.

The above improvements are contingent on the system load not being too high. [2] In the case where the system load is high, we show that all the task assignment policies have such poor performance that they become impractical, and `TAGS` is especially negatively affected. 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 which must be added" above as the *server expansion* requirement. We will show that `TAGS` outperforms all the previously-mentioned task assignment policies with respect to the *server expansion* metric (i.e., given any initial load, `TAGS` requires far fewer additional hosts to perform well).

We will describe three flavors of `TAGS`. The first, called `TAGS-opt-meanslowdown` is designed to minimize mean slowdown. The second, called `TAGS-opt-meanwaitingtime`

[2] For a distributed server, system load is defined as follows:

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

For example, a system with 2 hosts and system load .5 has same outside arrival rate as a system with 4 hosts and system load .25. Observe that a 4 host system with system load ρ has twice the outside arrival rate of a 2 host system with system load ρ.

is designed to minimize mean waiting time. Although very effective, these algorithms are not fair in their treatment of tasks. The third flavor, called `TAGS-opt-fairness`, is designed to optimize fairness. While managing to be fair, `TAGS-opt-fairness` still achieves mean slowdown and mean waiting time close to the other flavors of `TAGS`.

Section 2 elaborates in more detail on previous work in this area. Section 3 provides the necessary background on measured task 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 2 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 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 restriction that all tasks be immediately dispatched to a host upon arrival and each host services its tasks in FCFS order. Under this restricted model, it has been shown that when the task size distribution is exponential and the arrival process is Poisson, then the `Shortest-Queue` task assignment policy is optimal, Winston [19]. In this result, optimality is defined as maximizing the discounted number of tasks which complete by some fixed time t. Ephremides, Varaiya, and Walrand [5] showed that the `Shortest-Queue` task assignment policy also minimizes the expected total time for the completion of all tasks arriving by some fixed time t, under an exponential task size distribution and arbitrary arrival process. The actual performance of the `Shortest-Queue` policy is not known exactly, but the mean response time is approximated by Nelson and Phillips [11], [12]. Whitt has shown that as the variability of the task size distribution grows, the `Shortest-Queue` policy is no longer optimal [18]. Whitt does not suggest which policy is optimal.

The scenario has also been considered, under the same restricted model described in the above paragraph, but where the ages (time in service) of the tasks currently serving are known, so that it is possible to compute an arriving task's expected delay at each queue. In this scenario, Weber [17] considers the `Shortest-Expected-Delay` rule which sends each task to the host with the least expected work (note the similarity to the `Least-Work-Remaining` policy). Weber shows that this rule is optimal for task size distributions with increasing failure rate (including Exponential). Whitt [18] shows that there exist task size distributions for which this rule is not optimal.

Wolff, [20] has proven that `Least-Work-Remaining` is the best possible task assignment policy out of all policies which do not make use of task size. This result holds for any distribution of task sizes and for any arrival process.

Another model which has been considered is the case of no preemption but where the size of each task is known at the time of arrival of the task. Within this model, the `SITA-E` algorithm (see [7]) has been shown to outperform the `Random`, `Round-Robin`, `Shortest-Queue`, and `Least-Work-Remaining` algorithms by several orders of magnitude when the task size distribution is heavy-tailed. In contrast to `SITA-E`, the `TAGS` algorithm does not require knowledge of task 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 and other generalizations

Throughout this paper we maintain the assumption that tasks are *not* preemptible. That is, once a task 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 tasks are preemptible (see [8] for many citations).

Other generalizations of the task assignment problem include the scenario where the hosts are heterogeneous or there are multiple resources under contention.

The idea of purposely unbalancing load has been suggested previously in [3] and in [1], under different contexts from our paper. In both these papers, it is assumed that task sizes are *known* a priori. In [3] a distributed system with preemptible tasks 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 [1], real-time scheduling is considered where tasks 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 tasks is maximized."

3 Heavy Tails

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

Many application environments show a mixture of task sizes spanning many orders of magnitude. In such environments there are typically many small tasks, and fewer large tasks. 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

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

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

1. Decreasing failure rate: In particular, the longer a task 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 very small fraction ($< 1\%$) of the very largest tasks make up a large fraction (half) of the load. We will refer to this important property throughout the paper as the *heavy-tailed property.*

The lower the parameter α, the more variable the distribution, and the more pronounced is the heavy-tailed property, *i.e.* the smaller the fraction of large tasks 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 paper [8]. This distribution closely fits the curve

$$\Pr\{\text{Process Lifetime } > T\} = 1/T.$$

In [8] 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: $1 \leq \alpha \leq 1.25$ [10].
- Unix process CPU requirements, measured at UC Berkeley: $\alpha \approx 1$ [8].
- Sizes of files transferred through the Web: $1.1 \leq \alpha \leq 1.3$ [2, 4].
- Sizes of files stored in Unix filesystems: [9].
- I/O times: [14].
- Sizes of FTP transfers in the Internet: $.9 \leq \alpha \leq 1.1$ [13].

In most of these cases where estimates of α were made, α tends to be close to 1, which represents very high variability in task service requirements.

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

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

In this paper, we will vary the α-parameter over the range 0 to 2 in order to observe the effect of changing 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 [2]. In order to keep the mean constant, we adjust k slightly as α 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 distribution will still show very high variability if $k \ll p$. For example, Figure 4 (right) shows the second moment $\mathbf{E}\left\{X^2\right\}$ of this distribution as a function of α for $p = 10^{10}$, where k is chosen to keep $\mathbf{E}\left\{X\right\}$ constant at 3000, ($0 < k \leq 1500$). The figure shows that the second moment explodes exponentially as α 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 determining our choice of the best task assignment policy.

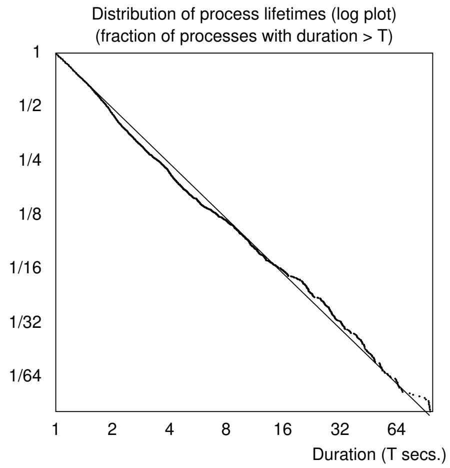


Figure 3: *Measured distribution of UNIX process CPU lifetimes, taken from [HD97]. Data indicates fraction of jobs whose CPU service demands exceed T seconds, as a function of T.*

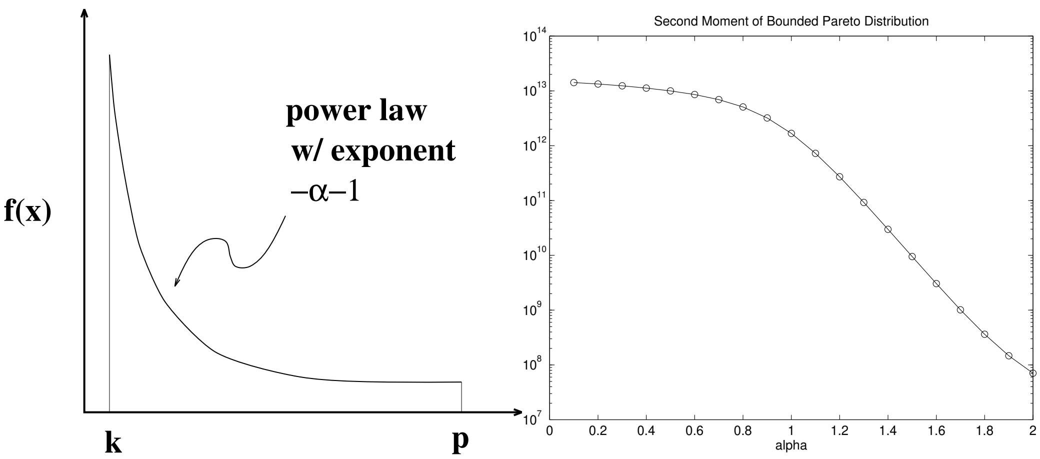


Figure 4: *Parameters of the Bounded Pareto Distribution (left); Second Moment of $B(k, 10^{10}, \alpha)$ as a function of α, when $\mathbf{E}\{X\} = 3000$ (right).*

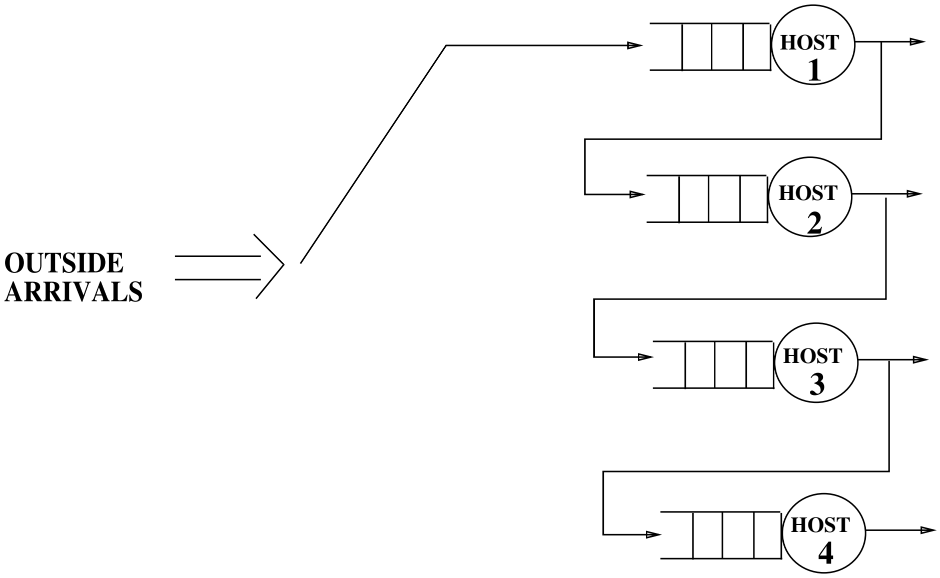


Figure 5: *Illustration of the flow of tasks in the* `TAGS` *algorithm.*

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: $1, 2, \ldots, h$. The ith host has a number s_i associated with it, where $s_1 < s_2 < \ldots < s_h$.

`TAGS` works as shown in Figure 5: *All* incoming tasks 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 task has used s_1 amount of CPU at Host 1 and still hasn't completed, then it is killed (remember tasks cannot be preempted because that is too expensive in our model). The task is then put at the end of the queue at Host 2, where it is restarted from scratch[3]. Each host services the tasks in its queue in FCFS order. If a task at host i uses up s_i amount of CPU and still hasn't 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 task, 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

[3]Note, although the task is restarted, it is still the same task, of course. We are therefore careful in our analysis *not* to assign it a new service requirement.

first host overcrowded since all incoming tasks are sent to Host 1. An even more vital concern is that the `TAGS` algorithm *wastes* a large amount of resources by killing tasks and then restarting them from scratch.[4] There's also the sense that the big tasks are especially penalized since they're the ones being restarted.

`TAGS` comes in 3 flavors; these only differ in how the s_i's are chosen. In `TAGS-opt-meanslowdown`, the s_i's are chosen so as to optimize mean slowdown. In `TAGS-opt-meanwaitingtime`, the s_i's are chosen so as to optimize mean waiting time. As we'll see, `TAGS-opt-meanslowdown` and `TAGS-opt-meanwaitingtime` are not necessarily fair. In `TAGS-opt-fairness` the s_i's are chosen so as to optimize fairness. Specifically, the tasks whose final destination is Host i experience the same expected slowdown under `TAGS-opt-fairness` as do the tasks whose final destination is Host j, for all i and j.

`TAGS` may seem reminiscent of multi-level feedback queueing, but they are *not* related. In multi-level feedback queueing there is only a single host with many *virtual* queues. The host is time-shared and tasks are preemptible. When a task uses some amount of service time it is transferred (not killed and restarted) to a lower priority queue. Also, in multi-level feedback queueing, the tasks in that lower priority queue are *only* allowed to run when there are no tasks in any of the higher priority queues.

5 Analysis and Results and For the Case of 2 Hosts

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

We evaluate several task assignment policies, all as a function of α, where α is the variance-parameter for the Bounded Pareto task size distribution, and α ranges between 0 and 2. Recall from Section 3 that the lower α is, the higher the variance in the task size distribution. Recall also that empirical measurements of task size distributions often show $\alpha \approx 1$.

[4] 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 task's been killed to help the general good.

We will evaluate the Random, Least-Work-Remaining, and TAGS policies. The Round-Robin policy (see Section 1) will not be evaluated directly because we showed in a previous paper [7] that Random and Round-Robin have almost identical performance. As we'll explain in Section 5.1, our analysis of Least-Work-Remaining is only an approximation, however we have confidence in this approximation because our extensive simulation in paper [7] showed it to be quite accurate in this setting. As we'll discuss in Section 5.1, our analysis of TAGS is also an approximation, though to a lesser degree.

Figure 6(a) below shows mean slowdown under TAGS-opt-slowdown as compared with the other task assignment policies. The y-axis is shown on a log scale. Observe that for very high α, the performance of all the task assignment policies is comparable and very good, however as α decreases, the performance of all the policies degrades. The Least-Work-Remaining policy consistently outperforms the Random policy by about an order of magnitude, however the TAGS-opt-slowdown policy offers several orders of magnitude further improvement: At $\alpha = 1.5$, the TAGS-opt-slowdown policy outperforms the Least-Work-Remaining policy by 2 orders of magnitude; at $\alpha \approx 1$, the TAGS-opt-slowdown policy outperforms the Least-Work-Remaining policy by over 4 orders of magnitude; at $\alpha = .4$ the the TAGS-opt-slowdown policy outperforms the Least-Work-Remaining policy by over 9 orders of magnitude, and this increases to 15 orders of magnitude for $\alpha = .2$!

Figures 6(b) and (c) show mean slowdown of TAGS-opt-waitingtime and TAGS-opt-fairness, respectively, as compared with the other task assignment policies. Since TAGS-opt-waitingtime is optimized for mean waiting time, rather than mean slowdown, it is understandable that its performance improvements with respect to mean slowdown are not as dramatic as those of TAGS-opt-slowdown. However, what's interesting is that the performance of TAGS-opt-fairness is very close to 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, especially TAGS-opt-waitingtime, 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 task to where it will individually experience the lowest waiting time. The reason why TAGS works so well is 2-fold: The first part is *variance reduction* (Section 5.1) and the second part is *load unbalancing* (Section 5.2).

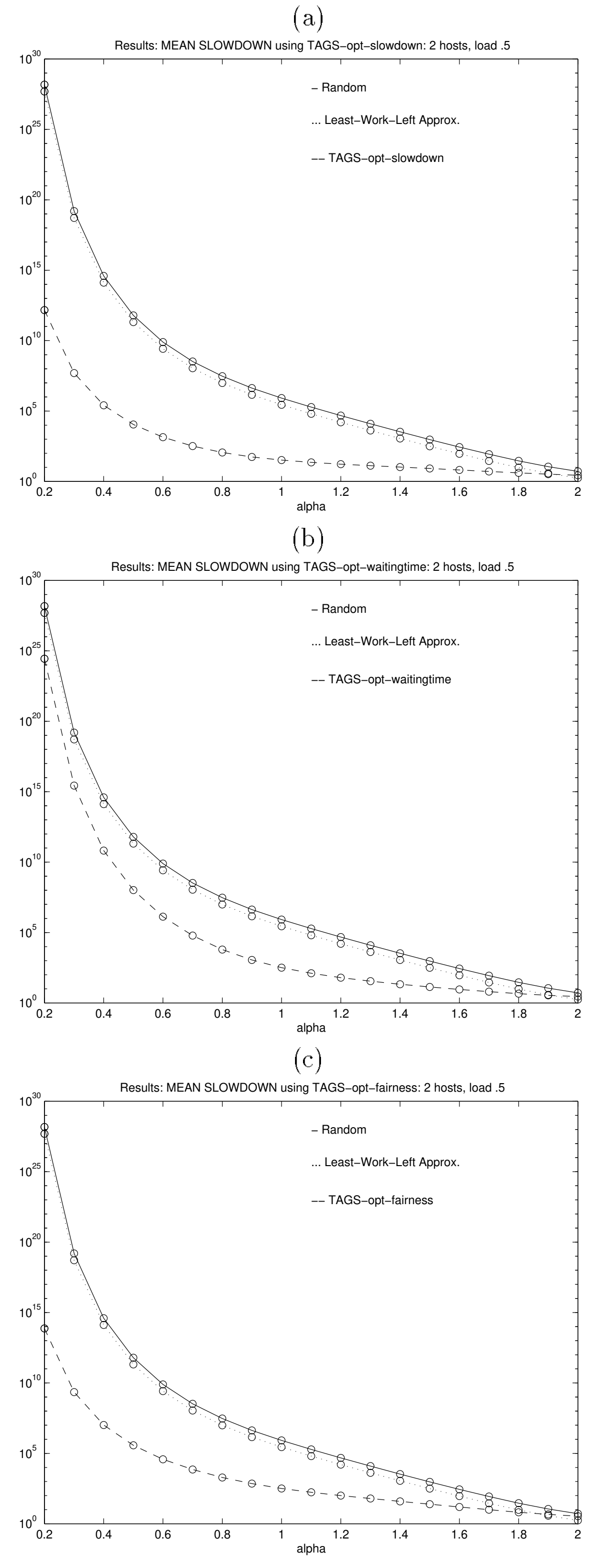
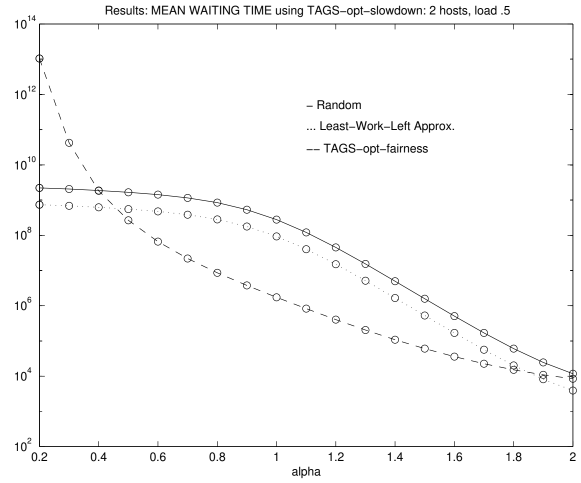
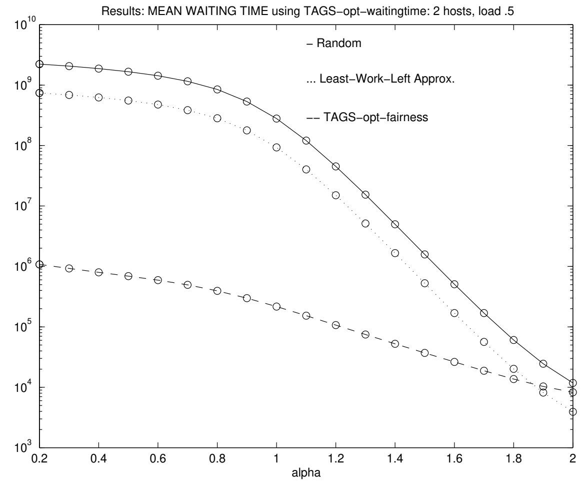


Figure 6: *Mean slowdown for distributed server with 2 hosts and system load .5 under (a)* `TAGS-opt-slowdown`*, (b)* `TAGS-opt-waitingtime`*, and (c)* `TAGS-opt-fairness` *as compared with the* `Least-Work-Remaining` *and* `Random` *task assignment policies.*

(a)



(b)



(c)

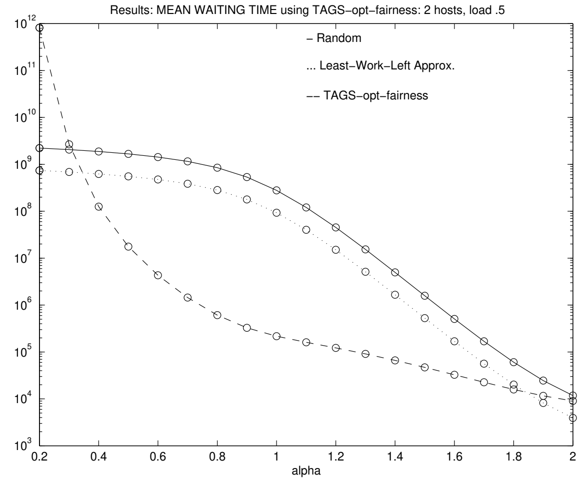


Figure 7: *Mean waiting time for distributed server with 2 hosts and system load .5 under (a)* `TAGS-opt-slowdown`*, (b)* `TAGS-opt-waitingtime`*, and (c)* `TAGS-opt-fairness` *as compared with the* `Least-Work-Remaining` *and* `Random` *task assignment policies.*

5.1 Variance Reduction

Variance reduction refers to reducing the variance of task sizes that share the same queue. Intuitively, variance reduction is important for improving performance because it reduces the chance of a small task getting stuck behind a big task in the same queue. This is stated more formally in Theorem 1 below, which is derived from the Pollaczek-Kinchin formula.

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 task's waiting time in queue, S be its slowdown, and Q be the queue length on its arrival. Then,*

$$\begin{aligned} \mathbf{E}\{W\} &= \frac{\lambda \mathbf{E}\{X^2\}}{2(1-\rho)} \qquad \textit{[Pollaczek-Kinchin formula]} \\ \mathbf{E}\{S\} &= \mathbf{E}\{W/X\} = \mathbf{E}\{W\} \cdot \mathbf{E}\{X^{-1}\} \\ \mathbf{E}\{Q\} &= \lambda \mathbf{E}\{W\} \end{aligned}$$

Proof: The slowdown formulas follow from the fact that W and X are independent for a FCFS queue, and the queue size follows from Little's formula. ∎

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, as shown in Figure 4.

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

Random Task Assignment The `Random` policy simply performs Bernoulli splitting on the input stream, with the result that each host becomes an independent $M/B(k,p,\alpha)/1$ queue. The load at the ith host, ρ_i, is equal to the system load, ρ. The arrival rate at the ith 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.

Round Robin The `Round Robin` policy splits the incoming stream so each host sees an $E_h/B(k,p,\alpha)/1$ queue, with utilization $\rho_i = \rho$. This system has performance close to the `Random` policy since it still sees high variability in service times, which dominates performance.

Least-Work-Remaining The `Least-Work-Remaining` policy is equivalent to an M/G/h queue, for which there exist known approximations, [16],[21]:

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

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` task assignment policies. In fact, the performance metrics are all proportional to the squared coefficient of variation ($C^2 = \frac{\mathbf{E}\{X^2\}}{\mathbf{E}\{X\}^2}$) of the service time distribution.

TAGS The `TAGS` policy is the only one which reduces the variance of task sizes at the individual hosts. Let p_i be the fraction of tasks whose final destination is Host i. Consider the tasks which queue at Host i: First there are those tasks which are destined for Host i. Their task size distribution is $B(s_{i-1}, s_i, \alpha)$ because the original task size distribution is a Bounded Pareto. Then there are the tasks which are destined for hosts numbered greater than i. These tasks are all capped at size s_i. Thus the second moment of the task 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* and is relatively straightforward except for one point which we have to fudge and which we explain now: For analytic convenience, we need to be able to assume that the tasks 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 $i-1$ which 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}. Throughout our analysis of `TAGS`, we make the assumption that the arrival process into Host i is a Poisson Process.

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.

Figure 8 shows the load at Host 1 and the load at Host 2 for `TAGS-opt-slowdown`, `TAGS-opt-waitingtime`, and `TAGS-opt-fairness` as a function of α. Observe that all 3 flavors of `TAGS` (purposely) severely underload Host 1 when α is low but for higher α 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 will 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 α), only a miniscule fraction of all tasks – the very largest ones – are needed to make up more than half the total load. As an example, for the case $\alpha = .2$, it turns out that less than 10^{-6} fraction of all tasks are needed to make up half the load. In fact not many more tasks, still less than 10^{-4} fraction of all tasks, 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* tasks ($(1 - 10^{-4})$ fraction) have Host 1 as their final destination, and only a very *few* tasks (the largest 10^{-4} fraction of all tasks) 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 tasks get to run at such reduced load, the overall mean slowdown is very low.

When the distribution is a little less heavy-tailed, e.g., $\alpha \approx 1$, we can't play this load unbalancing game as well. Again, we would like to severely underload Host 1 and send .999999 fraction of the load to go to Host 2. Before we were able to do this by making only a very small fraction of all tasks ($< 10^{-4}$ fraction) go to Host 2. However now that the distribution is not as heavy-tailed, a larger fraction of tasks must have Host 2 as its final destination to create very high load at Host 2. But this in turn means that tasks with destination Host 2 count more in determining the overall mean slowdown of the system, which is bad since tasks 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 get to $\alpha > 1$, it turns out that it actually pays to *overload* Host 1 a little. This seems counter-intuitive, since Host 1 counts more in determining the overall mean slowdown of the system because the fraction of tasks with destination Host 1 is greater. However, the point is that now it is impossible to create the wonderful state where almost all tasks 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 tasks 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 task 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. Under `TAGS-opt-fairness`,

(a)

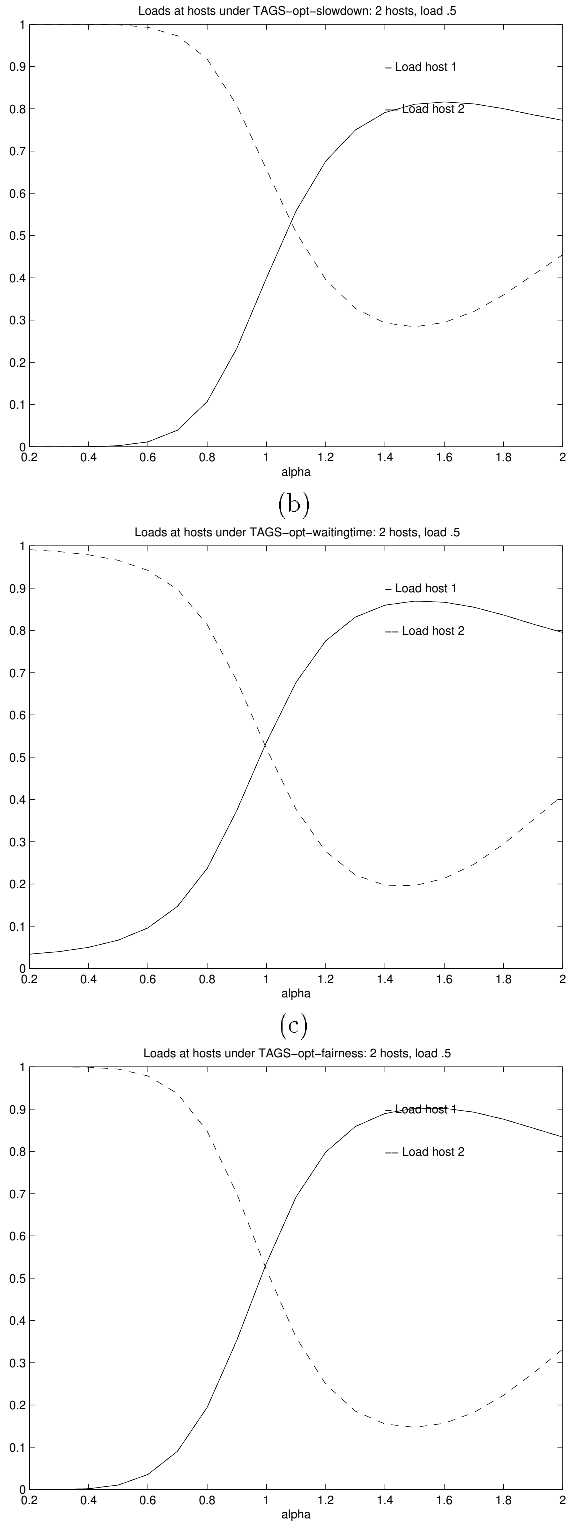
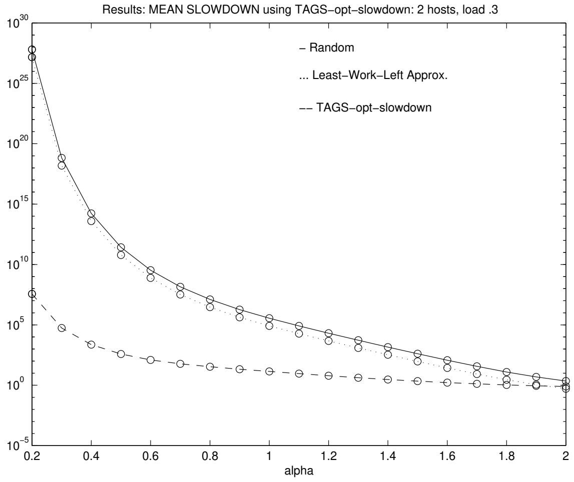
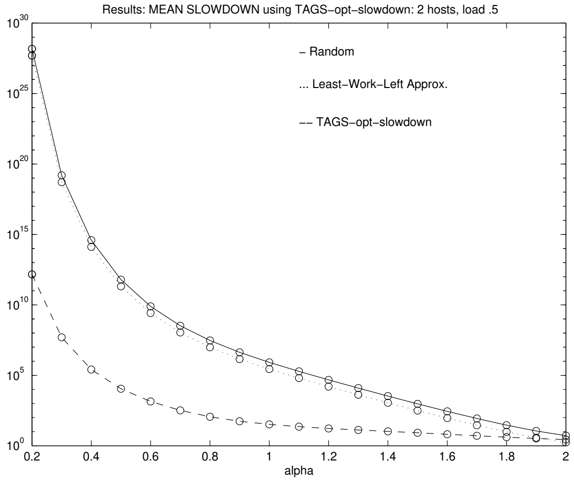


Figure 8: *Load at Host 1 as compared with Host 2 in a distributed server with 2 hosts and system load .5 under (a)* `TAGS-opt-slowdown`*, (b)* `TAGS-opt-waitingtime`*, and (c)* `TAGS-opt-fairness`*. Observe that for very low α, Host 1 is run at load close to zero, and Host 2 is run at load close to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

(a) System load 0.3



(b) System load 0.5



(c) System load 0.7

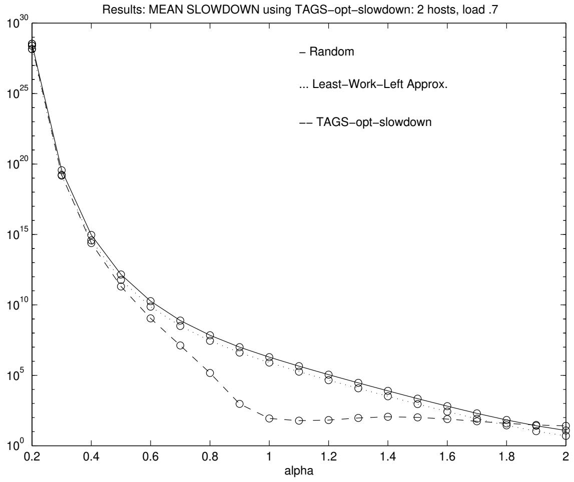


Figure 9: *Mean slowdown under* `TAG$9opt-slowdown` *in a distributed server with 2 hosts with system load (a)* 0.3, *(b)* 0.5, *and (c)* 0.7. *In each figure the mean slowdown under* `TAGS-opt-slowdown` *is compared with the performance of* `Random` *and* `Least-Work-Remaining`. *Observe that in all the figures* `TAGS` *outperforms the other task assignment policies under all* α. *However* `TAGS` *is most effective at lower system loads.*

the mean slowdown experienced by the small tasks is *equal* to the mean slowdown experienced by the big tasks. However it seems in fact that we're treating the big tasks unfairly on 3 counts:

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

So how can it possibly be fair to help the small tasks so much? The answer is simply that the small tasks are small. Thus they need low waiting times to keep their slowdown low. Big tasks 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 task size distribution for low α is so heavy-tailed that the big tasks are truly elephants (way bigger than the smalls) and thus can afford to suffer more.[5]

5.3 Different Loads

Until now we have studied only the model of a distributed server with two hosts and system load .5. In this section we consider the effect of system load on the performance of `TAGS`. We continue to assume a 2 host model. Figure 9 shows the performance of `TAGS-opt-slowdown` on a distributed server with 2 hosts 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 α, however the improvement of `TAGS-opt-slowdown` is much better when the system is more lightly loaded. In fact, all the task assignment 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 6 or 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

[5] It may interest the reader to understand the degree of unfairness exhibited by `TAGS-opt-slowdown` and `TAGS-opt-waitingtime`. For `TAGS-opt-slowdown`, our analysis shows that the expected slowdown of the big tasks always exceeds that of the small tasks and the ratio increases exponentially as α drops, so that at $\alpha = 2$, $\mathbf{E}\left\{Slowdown(bigs)\right\} \approx 2 \cdot \mathbf{E}\left\{Slowdown(smalls)\right\}$, and at $\alpha = .2$, $\mathbf{E}\left\{Slowdown(bigs)\right\} \approx 10^4 \cdot \mathbf{E}\left\{Slowdown(smalls)\right\}$. In contrast, for `TAGS-opt-waitingtime`, the expected slowdown of the big tasks is approximately equal to that of the small tasks until α drops below 1, at which point the expected slowdown of the big tasks drops way below that of the small tasks, the ratio of bigs to smalls decreasing superexponentially as α drops.

hand, `TAGS-opt-slowdown` behaves comparably to `Least-Work-Remaining` for most α 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 2 reasons, both of which can be understood by looking at Figure 10 which shows the loads at the 2 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 α, `TAGS` reaps much of its benefit at the lower α 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 becomes a bottleneck for `TAGS` since it means that Host 1 can not 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 α than it is in (b).

The second reason for the ineffectiveness of `TAGS` under high loads has to do with what we call *excess*. Excess is the extra work created in `TAGS` by tasks being killed and restarted. In the 2-host case, the excess is simply equal to $\lambda \cdot p_2 \cdot s_1$, where λ is the outside arrival rate, p_2 is the fraction of tasks whose final destination is Host 2, and s_1 is the cutoff differentiating small tasks from big tasks. 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. Notice that the dotted line in Figure 10(a)(b)(c) shows the sum of the loads on the hosts.

Until now we've only considered the distributed servers with 2 hosts and system load 0.5. For this scenario, excess has not been a problem. The reason is that for low α, where we need to do the severe load unbalancing, excess is basically non-existent 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 α, excess is present. However all the task assignment policies already do well in the high α region because of the low task size variability, so the excess is not much of a handicap.

When we look at the case of system load 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 λ 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 α 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 α is very high. To make matters worse, some of this excess must be heaped on Host 1. In the high α range, excess again is high because p_2 is high.

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 Multiple Hosts

Until now we have only considered distributed servers with 2 hosts. For the case of 2 hosts, we saw that the performance of `TAGS-opt-slowdown` was amazingly good 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 2 hosts.

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 2 hosts and system load 0.7, we could increase the number of hosts to 4 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 4 hosts and increase the arrival rate appropriately (double it) so as to maintain a system load of 0.7. In our discussions below we will attempt to be clear as to which view we have in mind.

One claim that can be made straight off is that an h host system ($h > 2$) with system load ρ can always be configured to produce performance which is at least as good as that of a 2 host system with system load ρ. To see why, observe that we can use the h host system (assuming h is even) to simulate a 2 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 $2/h$th of the tasks go to each of the $h/2$ Subsystems. Now apply your favorite task assignment policy to each Subsystem independently – in our case we choose `TAGS`. Each Subsystem will behave like a 2 host system with load ρ 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 cute argument works for any task assignment policy).

Figure 12 shows the mean slowdown under `TAGS-opt-slowdown` for the case of a 4 host distributed server with system load 0.3. Comparing these results to those for the 2 host system with system load 0.3 (Figure 9(a)), we see that:

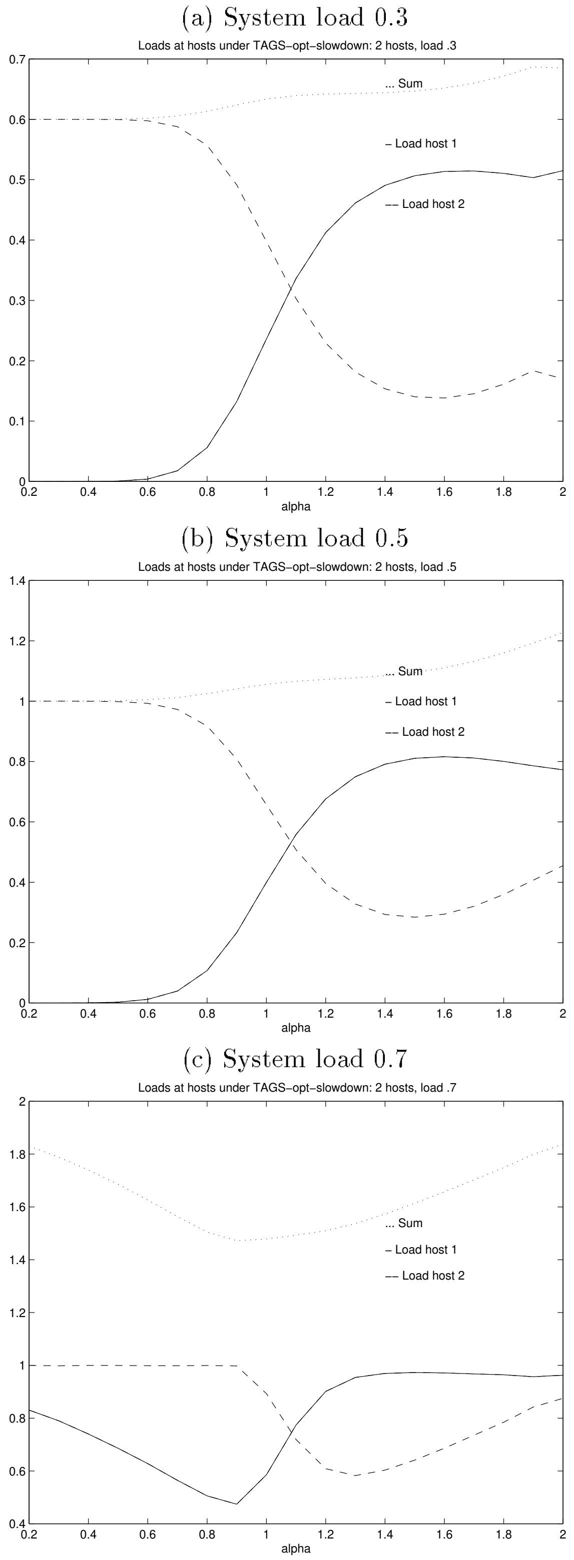


Figure 10: *Load at Host 1 and Host 2 under* `TAGS-opt-slowdown` *shown for a distributed server with 2 hosts and system load (a)* 0.3 *(b)* 0.5 *(c)* 0.7*. The dotted line shows the sum of the loads at the 2 hosts. If there were no excess, the dotted line would be at (a)* 0.6 *(b)* 1.0 *and (c)* 1.4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

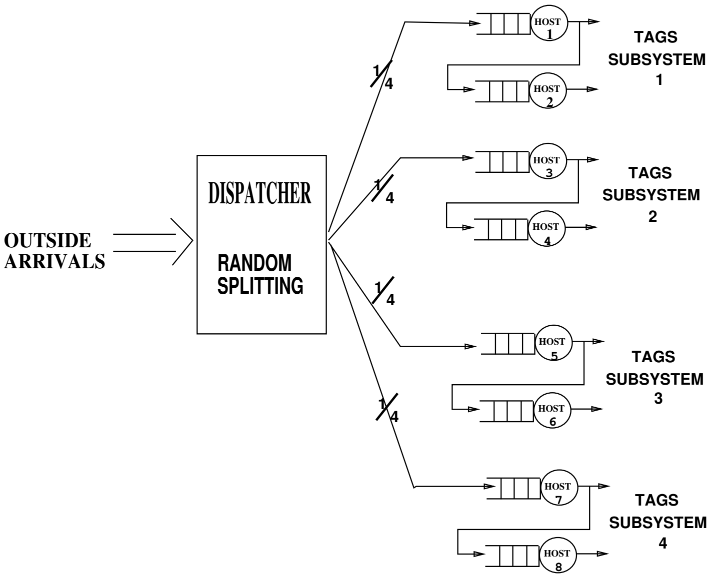


Figure 11: *Illustration of the claim that an h host system ($h > 2$) with system load ρ can always be configured to produce performance at least as good as a 2 host system with system load ρ (although the h host system has much higher arrival rate).*

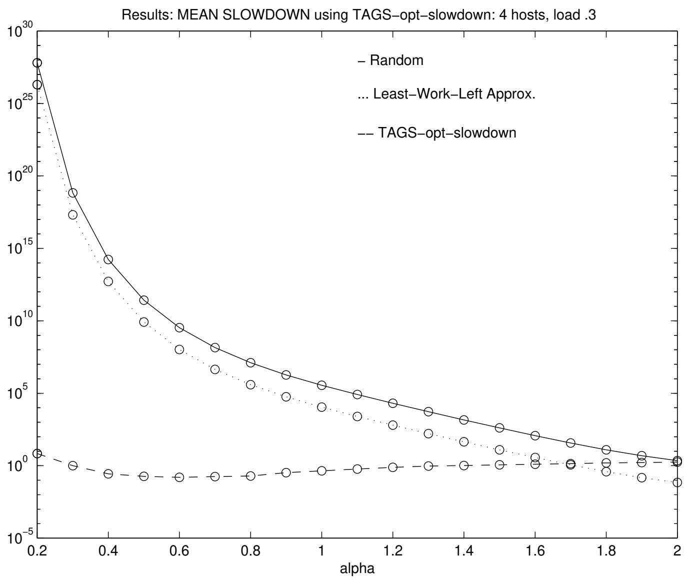


Figure 12: *Mean slowdown under* `TAGS-opt-slowdown` *compared with other task assignment policies in the case of a distributed server with 4 hosts and system load* 0.3*. The cutoffs for* `TAGS-opt-slowdown` *were optimized by hand. In many cases it is possible to improve upon the results shown here 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 1.*

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 α region, but less in the lower α region. The `Least-Work-Remaining` task assignment 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 almost always under 1. At $\alpha \approx 1$, `TAGS-opt-slowdown` improves upon `Least-Work-Remaining` by 4-5 orders of magnitude. At $\alpha = .6$, `TAGS-opt-slowdown` improves upon `Least-Work-Remaining` by 8-9 orders of magnitude. At $\alpha = .2$, `TAGS-opt-slowdown` improves upon `Least-Work-Remaining` by over 25 orders of magnitude!

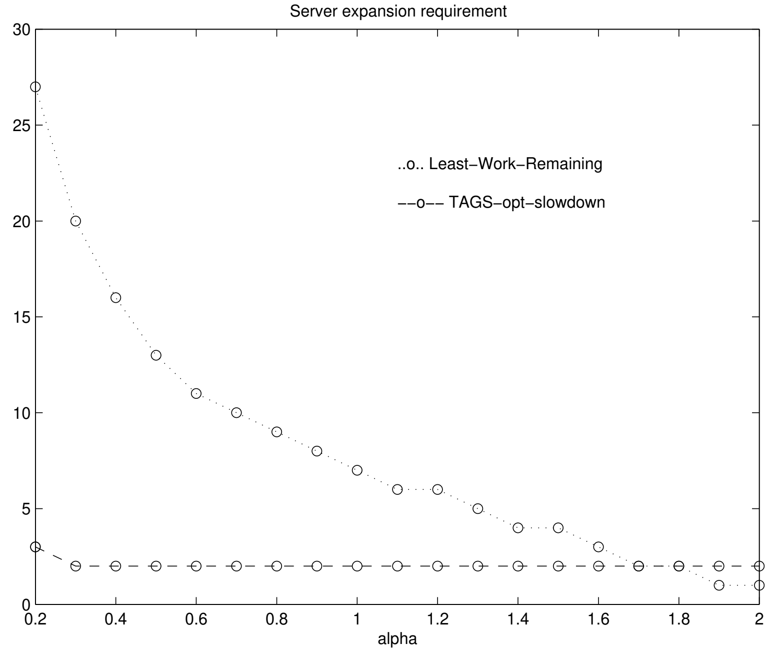
The enhanced performance of `TAGS` on more hosts may come from the fact that more hosts allow for greater flexibility in choosing the cutoffs. However it is hard to say for sure because it is difficult to compute results for the case of more than 2 hosts. The cutoffs in the case of 2 hosts were all optimized by Mathematica, while in the case of 4 hosts it was necessary to perform the optimizations by hand (and for all we know, it may be possible to do even better). For the case of system load 0.7 with 4 hosts we ran into the same type of problems as we did for the 2 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 be willing to run a system whose expected mean slowdown was 10^{10}. In practice, if a system was operating with mean slowdown of 10^{10}, 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, like 3 or less. Consider the following example: Suppose we have a 2-host system running at system load .7 and with variability parameter $\alpha = .6$. For this system the mean slowdown under `TAGS-opt-slowdown` is on the order of 10^9, and no other task assignment policy that we know of does better. Suppose however we desire a system with mean slowdown of 3 or less. So we double the number of hosts (without increasing the outside arrival rate). At 4 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 around 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 of 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'll arbitrarily define "reasonable" as 3 or less). Figure 13 compares the performance of our task assignment policies according to the server expansion metric, given that we start with a 2 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 α. For `Least-Work-Remaining`, on the other hand, the number of hosts we need to add ranges from 1 to 27, as α 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 task 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

(a) Non-log scale



(b) Log scale

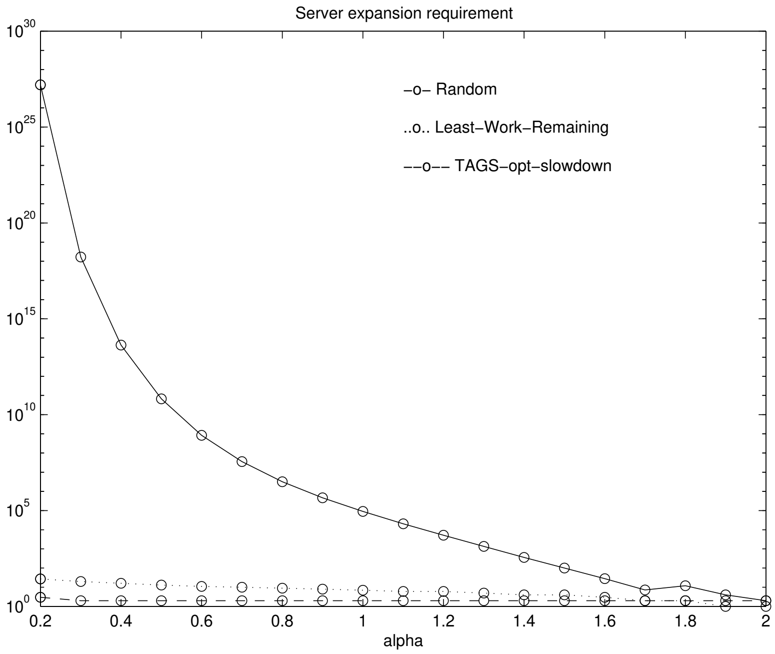


Figure 13: *Server expansion requirement for each of the task assignment policies, given that we start with a 2 host system with system load of* 0.7*. (a) Shows just* `Least-Work-Remaining` *and* `TAGS-opt-slowdown` *on a non-log scale (b) Shows* `Least-Work-Remaining`, `TAGS-opt-slowdown`, *and* `Random` *on a log scale.*

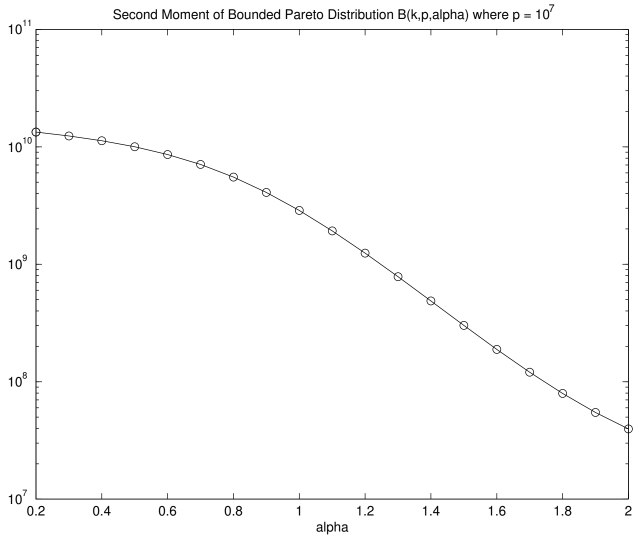


Figure 14: *Second moment of $B(k, p, \alpha)$ distribution, where now the upper bound, p, is set at $p = 10^7$, rather than 10^10. The mean is held fixed at 3000 as α is varied. Observe that the coefficient of variation now ranges from 2, when $\alpha = 2$ to 33, when $\alpha = .2$.*

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 task sizes (difference between the biggest and smallest possible task sizes) is smaller than we have heretofore assumed, resulting in a smaller coefficient of variation in the task size distribution.

Until now we have always assumed that the task 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 $\alpha \approx 1$ (as agrees with empirical data), we need to set the lower bound on task sizes to $k = 167$. However this implies that the range of task sizes spans 8 orders of magnitude!

It is not clear that most applications have task 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 task size distribution at 3000. This means, for example, that when $\alpha \approx 1$

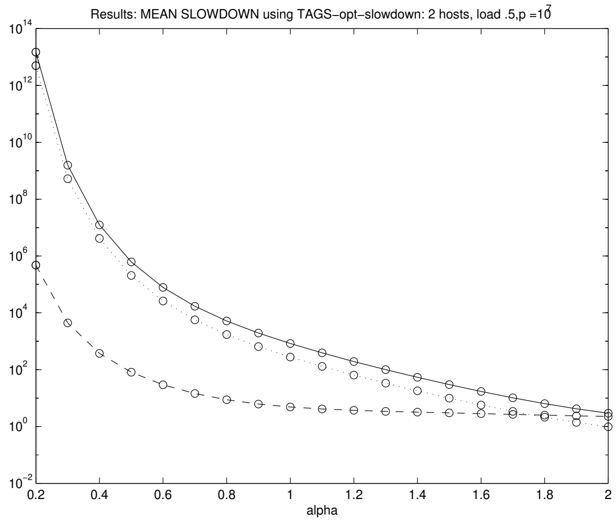


Figure 15: *Mean slowdown under* `TAGS-opt-slowdown` *in a distributed server with 2 hosts with system load* 0.5*, as compared with the performance of* `Random` *and* `Least-Work-Remaining`*. In this set of results the task size distribution is* $B(k, p, \alpha)$*, where* $p = 10^7$*.*

(as agrees with empirical data), we need to set the lower bound on task sizes to $k = 287$, which implies the range of task sizes spans just 5 orders of magnitude.

Figure 14 shows the second moment of the Bounded Pareto task size distribution as a function of α when $p = 10^7$. Comparing this figure to Figure 4, we see that the task size variability is far lower when $p = 10^7$ and therfore so is the coefficient of variation.

Lower variance in the task size distribution suggests that the improvement of `TAGS` over the other task assignment policies will not be as dramatic as in the higher variability 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 task assignment 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 task assignment policies by over 2 orders of magnitude. As α drops, the improvement increases. This figure should be contrasted with Figure 9(b), which shows the same scenario where $p = 10^{10}$.

8 Conclusion and Future Work

This paper 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 which 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 task and restarting 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 tasks and then restarts them 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 task multiple times.

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 tasks on a single CPU, where tasks are not preemptible and no a priori knowledge is given about the tasks. At first it seems that FCFS scheduling is the only option. However in the fact of high task size variability, FCFS may not be wise. This paper suggests that killing and restarting tasks may be worth investigating as an alternative, if the load on the CPU is low enough to tolerate the extra work created.

Task assignment also has applications outside of the context of a distributed server system described in this paper. A very interesting recent paper by Shaikh, Rexford, and Shin [15] discusses routing of IP flows (which also have heavy-tailed size distributions) and recommends routing long flows differently from short flows.

References

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

[2] Mark E. Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1997.

[3] Mark E. Crovella, Mor Harchol-Balter, and Cristina Murta. 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*, 1998.

[4] Mark E. Crovella, Murad S. Taqqu, and Azer Bestavros. Heavy-tailed probability distributions in the world wide web. In *A Practical Guide To Heavy Tails*, chapter 1, pages 1–23. Chapman & Hall, New York, 1998.

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

[6] Mor Harchol-Balter, Mark Crovella, and Cristina Murta. Task assignment in a distributed server. *To appear in IEEE Journal of Parallel and Distributed Computing, scheduled for late 1999.*

[7] Mor Harchol-Balter, Mark Crovella, and Cristina Murta. Task assignment in a distributed server. In *10th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, Lecture Notes in Computer Science, No. 1469.*, pages 13–24, September 1998.

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

[9] Gordon Irlam. Unix file size survey - 1993. Available at `http://www.base.com/gordoni/ufs93.html`, September 1994.

[10] W. E. Leland and T. J. Ott. Load-balancing heuristics and process behavior. In *Proceedings of Performance and ACM Sigmetrics*, pages 54–69, 1986.

[11] Randolph D. Nelson and Thomas K. Philips. An approximation to the response time for shortest queue routing. *Performance Evaluation Review*, 7(1):181–189, 1989.

[12] Randolph D. Nelson and Thomas K. Philips. An approximation for the mean response time for shortest queue routing with general interarrival and service times. *Performance Evaluation*, 17:123–139, 1993.

[13] Vern Paxson and Sally Floyd. Wide-area traffic: The failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, pages 226–244, June 1995.

[14] David L. Peterson and David B. Adams. Fractal patterns in DASD I/O traffic. In *CMG Proceedings*, December 1996.

[15] Anees Shaikh, Jennifer Rexford, and Kang G. Shin. Load-sensitive routing of long-lived ip flows. In *Proceedings of SIGCOMM*, September 1999.

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

[17] R. W. Weber. On the optimal assignment of customers to parallel servers. *Journal of Applied Probability*, 15:406–413, 1978.

[18] Ward Whitt. Deciding which queue to join: Some counterexamples. *Operations Research*, 34(1):226–244, January 1986.

[19] W. Winston. Optimality of the shortest line discipline. *Journal of Applied Probability*, 14:181–189, 1977.

[20] Ronald W. Wolff. An upper bound for multi-channel queues. *Journal of Applied Probability*, 14:884–888, 1977.

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

9 Appendix

This section contains the formulas we used in evaluating the `TAGS` task assignment policy. We will use the notation defined in Table 1.

The following observation will be helpful in understanding the first batch of formulas below: Observe that the original tasks, all of which enter Host 1, are have sizes *i.i.d.* from $B(k,p,\alpha)$. However, once a task is moved to Host 2, we know that its size exceeds s_1. Conditional on this knowledge, we can assume that the tasks *entering* Host 2 have sizes *i.i.d.* from $B(s_1,p,\alpha)$. Likewise the tasks *entering* Host j have sizes *i.i.d.* from $B(s_{j-1},p,\alpha)$. Observe also that the tasks whose *final* destination is Host j have sizes *i.i.d.* from $B(s_{j-1},j,\alpha)$.

The formulas below assume knowledge of the cutoff points $s_0, s_1, \ldots, s_h$. These are determined using mathematica to optimize either mean slowdown, mean waiting time or fairness, as desired.

$$
\begin{aligned}
f(x) &= \frac{\alpha k^\alpha}{1-(k/p)^\alpha} x^{-\alpha-1} \quad k \le x \le p \\
\mathbf{E}\left\{X^j\right\} &= \int_k^p f(x)\cdot x^j dx = \begin{cases} \frac{\alpha k^\alpha \left(k^{j-\alpha}-p^{j-\alpha}\right)}{(\alpha-j)(1-(k/p)^\alpha)} \text{ if } \alpha \neq j \\ \\ \frac{k}{1-(k/p)} \cdot (\ln p - \ln k) \text{ if } \alpha = j = 1 \end{cases} \\
\lambda &= \frac{1}{\mathbf{E}\left\{X\right\}} \cdot h \cdot \rho \\
p_i &= \int_{s_{i-1}}^{s_i} f(x)dx = \frac{k^\alpha}{1-(k/p)^\alpha}\left(s_{i-1}^{-\alpha} - s_i^{-\alpha}\right) \\
hostp_i &= \sum_{j=i}^{h} p_i \\
\mathbf{E}\left\{X_i^j\right\} &= \int_{s_{i-1}}^{s_i} x^j \frac{f(x)}{p_i} dx = \begin{cases} \frac{\alpha s_{i-1}^\alpha \left(s_{i-1}^{j-\alpha}-s_i^{j-\alpha}\right)}{(\alpha-j)\left(1-(s_{i-1}/s_i)^\alpha\right)} \text{ if } \alpha \neq j \\ \\ \frac{s_{i-1}s_i}{s_i-s_{i-1}}(\ln s_i - \ln s_{i-1}) \text{ if } \alpha = j = 1 \\ \\ \frac{\alpha s_{i-1}^\alpha}{\left(1-\left(\frac{s_{i-1}}{s_i}\right)^\alpha\right)} \cdot (\ln s_i - \ln s_{i-1}) \text{ if } \alpha = j = 2 \end{cases} \\
\mathbf{E}\left\{hostX_i\right\} &= \frac{p_i}{hostp_i} \cdot \mathbf{E}\left\{X_i\right\} + \frac{hostp_i - p_i}{hostp_i} \cdot s_i \\
\mathbf{E}\left\{hostX_i^2\right\} &= \frac{p_i}{hostp_i} \cdot \mathbf{E}\left\{X_i^2\right\} + \frac{hostp_i - p_i}{hostp_i} \cdot s_i^2 \\
host\lambda_i &= \lambda \cdot hostp_i
\end{aligned}
$$

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

Table 1: *Notation for analysis of* `TAGS`

$$
\begin{aligned}
host\rho_i &= host\lambda_i \cdot \mathbf{E}\left\{hostX_i\right\} \\
\mathbf{E}\left\{1/X_i^j\right\} &= \mathbf{E}\left\{X_i^{-j}\right\}
\end{aligned}
$$

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

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

Computing mean waiting time and mean slowdown follows from Theorem 1, except for one fudge, 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.

$$
\begin{aligned}
\mathbf{E}\left\{hostW_i\right\} &= host\lambda_i \cdot \mathbf{E}\left\{hostX_i^2\right\}/(2(1-host\rho_i)) \\
\mathbf{E}\left\{W_i\right\} &= \sum_{j=1}^{i} \mathbf{E}\left\{hostW_j\right\} \\
\mathbf{E}\left\{W\right\} &= \sum_{i=1}^{h} \mathbf{E}\left\{W_i\right\} \cdot p_i \\
\mathbf{E}\left\{S_i\right\} &= \mathbf{E}\left\{W_i\right\} \cdot \mathbf{E}\left\{1/X_i\right\} \\
\mathbf{E}\left\{S\right\} &= \sum_{i=1}^{h} \mathbf{E}\left\{S_i\right\} \cdot p_i
\end{aligned}
$$