# The Effect of Heavy-Tailed Job Size Distributions on Computer System Design.

Mor Harchol-Balter*
Laboratory for Computer Science
MIT, NE43-340
Cambridge, MA 02139
harchol@theory.lcs.mit.edu

**Abstract**

Heavy-tailed distributions, a.k.a. power-law distributions, have been observed in many natural phenomena ranging from physical phenomena to sociological phenomena. Recently heavy-tailed distributions have been discovered in computer systems. In particular the sizes (service demands) of computing jobs have been found to exhibit a heavy-tailed (power-law) distribution. Most previous analytic work in the area of computer system design has assumed that job sizes (service demands) are exponentially distributed. Many of the policies, algorithms, and general rules-of-thumb which are currently used in computer systems originated from analyses which assumed an exponentially-distributed workload.

In this paper we argue that we need to reevaluate existing computer system algorithms in light of the discovery of heavy-tailed workloads. We argue that an algorithm which is optimal under an exponentially distributed workload may be very far from optimal when the workload is heavy-tailed. We demonstrate this point via three common problems in the design of computer systems:

1. Choosing a migration policy in a network of workstations.
2. Choosing a task assignment policy for a distributed server.
3. Scheduling HTTP requests within a Web server.

For each problem above, we show that the answer is highly dependent on the job size distribution. We show how to do analysis under heavy-tailed job size distributions. We then show that our analysis leads us to policies whose performance improves greatly over commonly-used solutions, in some cases by orders of magnitude.

This paper is a compilation of a sequence of papers which the author co-wrote: [9, 6, 7, 8, 1]. Far more detail is contained in the original papers.

*Supported by the NSF Postdoctoral Fellowship in the Mathematical Sciences.

# 1 Introduction

Heavy-tailed distributions (also known as power-law distributions) have been observed in many natural phenomena including both physical and sociological phenomena. A commonly cited example is the distribution of hurricane damage: Most hurricanes cause very little damage; a small percentage of hurricanes cause a whole lot of damage; and the density function which describes this statement has the form of a power-law, [5]. Not surprisingly, the distribution of fire damage and earthquake damage are also heavy-tailed, [5]. As another exmaple, the overall distribution of national wealth is also heavy-tailed.

Recently heavy-tailed distributions have been discovered in computer systems. In particular the sizes (service demands) of computing jobs have been found to exhibit a heavy-tailed (power-law) distribution. For example, if we consider the CPU requirement of jobs, it turns out that most jobs have low CPU requirements; a few jobs have very high CPU requirements; and the density function has the form of a power-law.

Most previous analytic work in the area of computer system design has assumed that job sizes (service demands) are exponentially distributed. This assumption may have been made, in part, for analytical tractability. Many of the policies, algorithms, and general rules-of-thumb which are currently used in computer systems originated from analyses which assumed an exponentially-distributed workload.

The point of this paper is that existing policies and intuitions with respect to computer system design may be severely suboptimal when applied to heavy-tailed workloads. To correct this, first heavy-tailed distributions need to be understood at an intuitive level so that we can form new intuitions about which policies are effective and which aren't. Next, existing policies and new policies must be evaluated using new analysis which can incorporate heavy-tailed distributions.

In Section 2 we describe measurements of computing systems in which heavy-tailed distributions were observed, and we define what we mean by a heavy-tailed distribution. We then characterize the important properties of heavy-tailed distributions with respect to computer system design.

Sections 3, 4, and 5 are each a case study of a common system design problem. For each problem above, we solve the problem first in the context of an exponentially-distributed job size distribution and then in the context of a heavy-tailed job size distribution, obtaining very different answers.

This paper is based on a sequence of papers which the author co-wrote: [9, 6, 7, 8, 1]. This paper only provides brief summaries of the insights learned in the original papers. Throughout the reader is referred to the original papers for more detail.

# 2 Measurements of Heavy-tailed job size distributions in computer systems

Figure 1 depicts graphically on a log-log plot the measured distribution of CPU requirements of over a million UNIX processes, taken from paper [9]. Note, the figure shows only jobs which require at least 1 second of CPU. This distribution closely fits the curve

$$\Pr\{\text{Process CPU requirement} > x\} = 1/x.$$

In [9] it is shown that this distribution for CPU requirements is present in a variety of computing environments, including instructional, research, and administrative environments.

Figure 1 says that most jobs are small (require little CPU) and a few jobs are large (require a lot of CPU). This fact is also the case for an exponential distribution. However the curve shown in Figure 1 is far from exponential. To demonstrate this point, Figure 2 shows the best possible fit of an exponential distribution to the measured data.

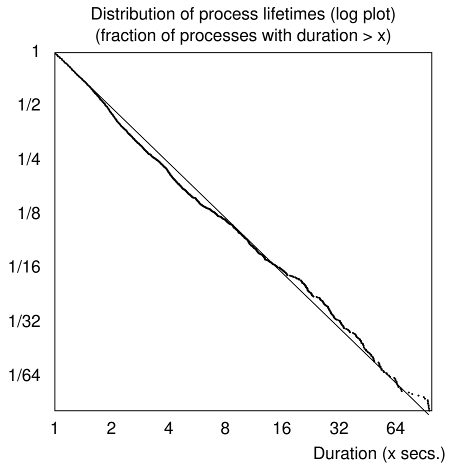



Figure 1: Measured distribution of UNIX process CPU lifetimes, taken from [HD97]. Data indicates fraction of jobs whose CPU service demands exceed $x$ seconds, as a function of $x$. The bumpy line indicates the measured data. The straight line is the curve-fit.

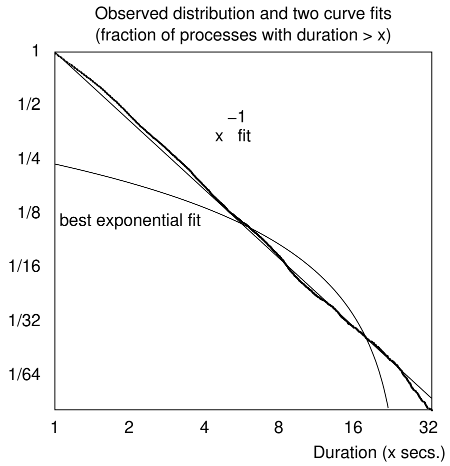



Figure 2: Curve shows best possible fit of an exponential distribution to the measured data.

The distribution shown in the above measurements is one example of a heavy-tailed distribution. In general a heavy-tailed distribution is one for which

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

where $0 < \alpha < 2$. (In the above measurements, $\alpha = 1$.) 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

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

We have found that when thinking about computer system design problems, there are a few *key characteristic properties* of heavy-tailed distributions. We list these here:

1. **Decreasing failure rate**: In particular, the longer a task has run, the longer it is expected to continue running. In fact, in the $1/x$ distribution shown in Figure 1, a job of CPU age $x$ (i.e. a job which has thus far used $x$ seconds of CPU) has probability half of using another $x$ seconds of CPU. In other words, the median remaining lifetime of a job is its current age. Contrast this with the exponential distribution which is memoryless. The exponential distribution and the heavy-tailed distribution have a similar "ski-slope" shape when drawn on non-logarithmic scales, however the heavy-tailed distribution drops off at a slower and slower rate as we move along the x-axis, whereas the exponential distribution drops off at a constant rate throughout.

2. **Infinite variance** (and if $\alpha \leq 1$, infinite mean). In reality, of course, any finite set of measurements or finite trace has finite variance. The point is that the variance is very high when the workload is heavy-tailed.

3. The property that a very small fraction ($< 1\%$) of the very largest jobs make up a large fraction (half) of the workload. We will refer to this key property throughout the paper as the **heavy-tailed property**. Contrast this with the exponential distribution where the largest 1% of the very largest jobs make up about 5% of the total workload. To understand the heavy-tailed property, consider the example of the distribution of our national wealth: most people have very little money; a few people have a lot of money; and the 1% richest people together have more money than all the other people combined.

The lower the parameter $\alpha$, the more variable the distribution, and the more pronounced is the heavy-tailed property, *i.e.* the smaller the faction of large tasks that comprise half the load.

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: $1 \leq \alpha \leq 1.25$ [12].
- Unix process CPU requirements, measured at UC Berkeley: $\alpha \approx 1$ [9].
- Sizes of files stored at Web sites and sizes of files accessed by Web requests: $1.1 \leq \alpha \leq 1.3$ [2, 4].
- Sizes of files stored in Unix filesystems: [11].
- 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 $\alpha$ were made, $1 \leq \alpha \leq 2$. In fact, typically $\alpha$ tends to be close to 1, which represents very high variability in task service requirements.

# 3 Case study 1: Choosing a migration policy in a network of workstations

The discussion in this section is taken entirely from the following paper: [9]. We present here a small subset of the material presented in that paper.

Consider the following problem: We have a Network of Workstations (NOW) with processes currently running on some of the workstations. We are interested in running some automated CPU load balancing policy, whereby processes are migrated from heavily-loaded hosts, across the network, to more lightly-loaded hosts where they can receive a greater share of the CPU. There are two types of load balancing possible: *remote execution* and *active process migration*. In remote execution (a.k.a. placement), a process may only be migrated if it has not started running yet. Active process migration, on the other hand, is the migration of active (already running) processes. Both types of migration have cost, but cost of remote execution is very small whereas the cost of active process migration can be quite large since an active process has accumulated memory which must be moved along with the process. Active process migration can in fact be very expensive because the cost of migrating the process is charged not only to the migrant but also to the source and target hosts. We are interested in answering following two policy questions:

1. Is active process migration really necessary to achieve good load balancing, or is remote execution enough?

2. If we do opt for active process migration, what is a good active migration policy? That is, which active processes does it pay to migrate?

In answering the above questions we will assume a somewhat simplified model in which processes consist only of CPU and memory (i.e. no interactive jobs). All hosts are timesharing hosts.

Our discussion in this section will center on how the *distribution of process sizes* effects the answer to this question. A process' *size* (or *lifetime*) is defined to be its total CPU requirement. Likewise a process' *age* is defined to be the amount of CPU it has used so far. We say a process is *old* if it has used a lot of CPU so far. Note that process lifetimes are not known a priori.

Returning to question (1) above, suppose the distribution of process lifetimes was exponential. Then processes of all ages would have the same expected remaining lifetime, so there would not be benefit to migrating active processes which have much higher migration costs. However if the distribution of UNIX process lifetimes is heavy-tailed as our measurements show in Figure 1, older processes live longer, i.e., the older a job is the more likely it is to use another second of CPU. Thus it may pay to migrate old active jobs because although their migration cost is very high, their remaining CPU requirement of the job may be even higher.

To answer the question of whether active process migration is in fact necessary, we performed a trace-driven simulation. We simulated a network of 6 hosts. Process start times, durations, etc. were taken from real machine traces of UNIX processes. We performed 8 independent 1-hour experiments. Each experiment involved 15,000–30,000 processes. The overall system utilization in the experiments ranged from .27 – .54. The complete details of the trace-driven simulation experiment setup are described in [9].

The purpose of the trace-driven simulation experiment was to compare two load-balancing strategies, remote execution versus active process migration, as described below:

In choosing a remote execution strategy, it is important to observe that remote execution, while cheap, is still not free. According to our measurements, [9], the size of most processes (i.e. their total CPU requirement) is smaller than the cost of remote execution. This is well-known and thus the standard remote execution strategy used is practical systems is based on name-lists. The name-list is a list of names of jobs which on average tend to have large size. Examples are `cc, gcc`. The name-list is created by looking at past history. A newborn process at a busy host is only chosen to be remotely executed if its name appears on the name-list.

In our trace-driven simulation experiment, we chose to use the standard name-list based remote execution strategy, except that in constructing the name-list, we allowed our remote execution policy to see all the traces ahead of time and make up the best possible name-list for that trace. The details are described in [9]. The purpose of doing this was to give the remote execution strategy every possible advantage.

Deriving an active migration policy for the trace-driven simulation experiment was not so obvious.

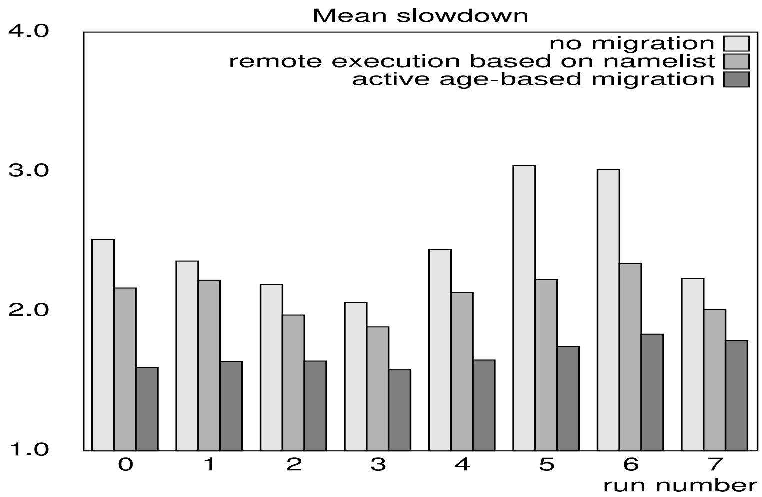



Figure 3: Trace-driven simulation results for 8 different runs. Mean slowdown is shown under no load balancing, under remote execution only, and under only migration of active processes.

The job size distribution of UNIX processes suggests that it may pay to migrate older jobs since they live longer, but how old is enough? In particular, the particular $1/x$ job size distribution from Figure 1 suggests that the remaining CPU requirement of *every* job is infinity. In [9], we show how to use the $1/x$ job size distribution to derive a migration policy for active processes, which we show has some interesting properties. For now we simply state the policy: An active process is migrated if:

$$\text{age of process} > \frac{\text{migration cost of process}}{\#\text{source} - \#\text{target} - 1},$$

where #source (#target) represents the number of jobs at the source (respectively target) host. Intuitively, this policy says that if a process' migration cost is high, the process needs to be old enough to afford the migration cost. However, if the difference between the loads at the source and target hosts is really great, then the process doesn't need to be quite as old to make the migration still worthwhile.

The trace-driven simulation experiment was performed first using no load balancing, then using only the remote execution policy, and finally using only the active migration policy. The results for the 8 runs are shown in Figure 3. Figure 3 shows that the remote execution policy improved the mean slowdown improved by about 20% over no load balancing, but the active migration policy created a much more dramatic effect –about 50% improvement over no load balancing. ( The *slowdown* of a process is its wall-time divided by its size. For example, a process whose CPU requirement was 3 seconds, but had a wall-time (a.k.a. response time) of 12 seconds, experienced a slowdown factor of 4.) In [9] we see that under other metrics (like mean response time, variance in slowdown, and number of severely slowed processes) improvements were even more dramatic in favor of active process migration.

In the experiment just described the average active migration cost of processes (memory/bandwidth) was 2.3 seconds (this cost was charged to both the migrant process and to the source host). The cost of remote execution was .3 seconds. In this experiment active process migration was more effective than remote execution. This begs the question: Suppose the mean active migration cost is much higher (say bandwidth is very low), does active migration then become less effective than remote execution? Figure 4 answers this question. Here we scaled the mean cost of active migration while holding the cost of remote execution constant at .3 seconds. The figure shows that even when the cost of active process migration is as high as 20 seconds, active process migration is still more effective in reducing mean slowdown than remote execution.

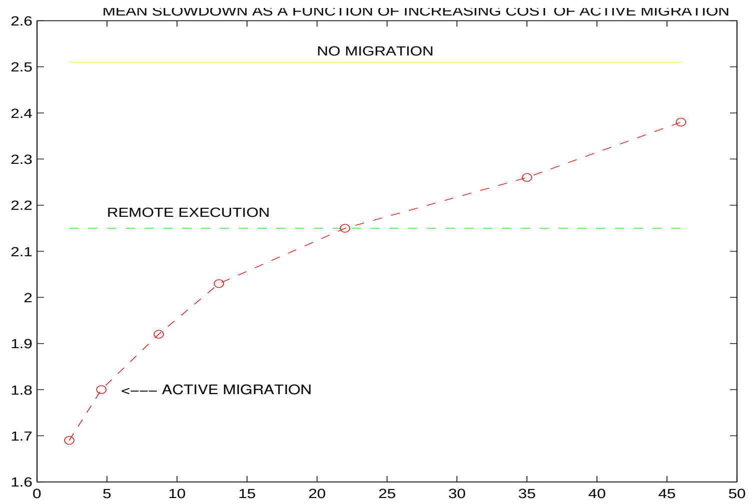



Figure 4: This figure shows the effect of increasing the cost of active process migration while holding constant the cost of remote execution. The x-axis shows the cost of active process migration. The y-axis shows mean slowdown. The figure shows that even when the cost of active process migration is as high as 20 seconds, active process migration is still more effective in reducing mean slowdown than remote execution.

Why? Why is migration of active processes more effective in load balancing than remote execution, especially given our highly-optimized remote execution policy?

The answer lies in understanding the job size distribution. First of all, because of the decreasing failure rate property of the job size distribution, active process migration was better able to detect long jobs than was remote execution. In fact, the mean lifetime of a migrant under the remote execution policy was $1.5-2.1$ seconds, whereas the mean lifetime of a migrant under active process migration was $4.5-5.7$ seconds. This simply says that the best indicator of how long a process is going to live, is how long it has lived already. But why does it matter if remote execution misses the opportunity to migrate a few big jobs? To see this we need to go back to the heavy-tailed property of the $1/x$ distribution. The active migration policy only migrated 4% of all jobs, but that 4% accounted for 55% of the total CPU. Thus by detecting the few very largest jobs and only migrating those, the active process migration policy was able to affect a large percentage of the total load.

To summarize, in this case study, the important properties of the process size distribution were the decreasing failure rate property and the heavy-tailed property. The reader is encouraged to seek the original paper [9] for a more complete understanding of these results.

# 4 Case study 2: Choosing a Task Assignment Policy for a distributed server system

The material in this section is taken entirely from the following papers: [7] and [6].

## 4.1 Introduction

To build high-capacity server systems, developers are increasingly turning to distributed designs because of their scalability and cost-effectiveness. Examples of this trend include distributed Web servers,

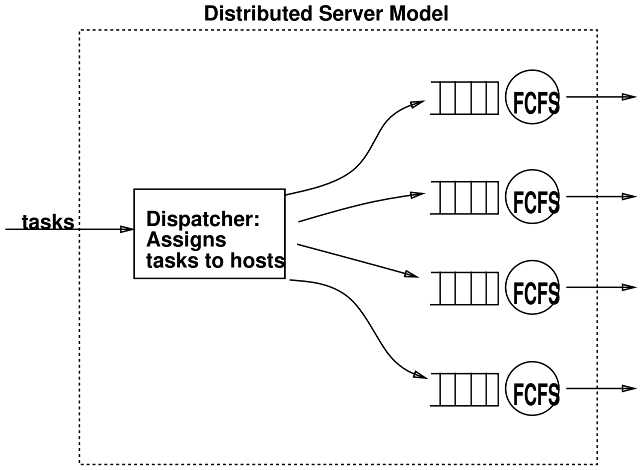



Figure 5: Distributed Server Model

distributed database servers, and high performance computing clusters. In such a system, requests for service 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*.

We will limit our discussion to the particular model of a distributed server system in which each incoming task is immediately assigned to a host machine, and each host machine processes its assigned tasks in first-come-first-served (FCFS) order, as shown in Figure 5. We also assume that the task's service demand is known in advance. Our motivation for considering this model is that it is an abstraction of some existing distributed servers, described in Section 4.2.

We consider four task assignment policies commonly proposed for such distributed server systems and ask which has the best mean response time and mean slowdown[2]. The four task assignment policies are:

**Random** : an incoming task is sent to host $i$ with probability $1/h$.

**Round-Robin** : tasks are assigned to hosts in cyclical fashion with the $i$th task being assigned to host $i \bmod h$.

**Size-Based** : Each host serves tasks whose service demand falls in a designated range.

**Dynamic** : Each incoming task is assigned to the host with the smallest amount of outstanding work, which is the sum of the sizes of the tasks in the host's queue plus the work remaining on that task currently being served.

Our goal is to understand the influence of the job size distribution on the question of which task assignment policy is best. In particular we are interested in heavy-tailed job size distributions and in exponential job size distributions.

## 4.2 Model and Problem Formulation

We are concerned with the following specific model of a distributed server. The server is composed of $h$ hosts, each with equal processing power. Tasks arrive to the system according to a Poisson process with rate $\lambda$. When a task arrives to the system, it is inspected by a dispatcher facility which assigns

[2]All means are per-task averages.

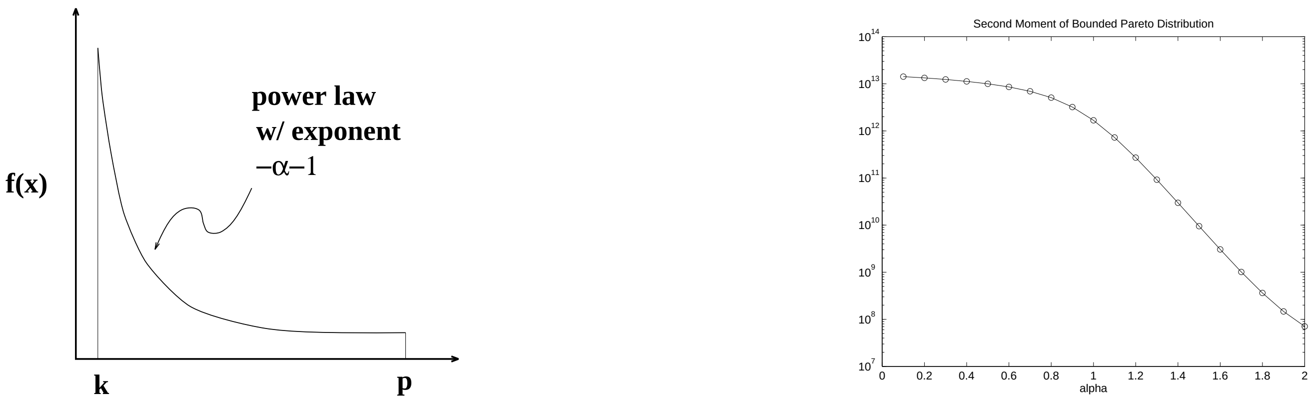



Figure 6: Parameters of the Bounded Pareto Distribution (left); Second Moment of $B(k, 10^{10}, \alpha)$ as a function of $\alpha$, when $\mathbf{E}\{X\} = 3000$ (right).

it to one of the hosts for service. We assume the dispatcher facility knows the size of the task. The tasks assigned to each host are served in FCFS order, and tasks are not preemptible. We assume that processing power is the only resource used by tasks.

The above model for a distributed server was initially inspired by the `xolas` batch distributed computing facility at MIT's Laboratory for Computer Science. `Xolas` consists of 4 identical multiprocessor hosts. Users specify an upper bound on their job's processing demand. If the job exceeds that demand, it is killed. The `xolas` facility has a dispatcher front end which assigns each job to one of the hosts for service. The user is then given an upper bound on the time her job will have to wait in the queue, based on the sum of the sizes of the jobs in that queue. The jobs queued at each host are each run to completion in FCFS order.

We assume that task sizes show some maximum (but large) value. As a result, we model task sizes using a distribution that follows a power law, but has an upper bound. We refer to this distribution as a *Bounded Pareto.* It is characterized by three parameters: $\alpha$, 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:

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

Throughout this section we model task sizes using a $B(k, p, \alpha)$ distribution, and vary $\alpha$ 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 $p = 10^{10}$). In order to keep the mean constant, we adjust $k$ slightly as $\alpha$ changes ($0 < k \leq 1500$). The above parameters are summarized in Table 1.

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 6 (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.

| | |
|---|---|
| Number of hosts | $h = 8$. |
| System load | $\rho = .8$. |
| Mean service time | $\mathbf{E}\{X\} = 3000$ time units |
| Task arrival process | Poisson with rate $\lambda = \rho \cdot 1/\mathbf{E}\{X\} \cdot h = .0021$ tasks/unit time |
| Maximum task service time | $p = 10^{10}$ time units |
| $\alpha$ parameter | $0 < \alpha \leq 2$ |
| Minimum task service time | chosen so that mean task service time stays constant as $\alpha$ varies ($0 < k \leq 1500$) |

Table 1: Parameters used in evaluating task assignment policies

### 4.3 A New Size-Based Task Assignment Policy: SITA-E

Before delving into simulation and analytic results, we need to specify a few more parameters of the size-based policy.

In size-based task assignment, a size range is associated with each host and a task is sent to the appropriate host based on its size. In practice the size ranges associated with the hosts are often chosen somewhat arbitrarily. There might be a 15-minute queue for tasks of size between 0 and 15 minutes, a 3-hour queue for tasks of size between 15 minutes and 3 hours, a 6-hour queue, a 12-hour queue and an 18-hour queue, for example. (This example is used in practice at the Cornell Theory Center IBM SP2 job scheduler [10].)

We choose a more formal algorithm for size-based task assignment, which we refer to as SITA-E — Size Interval Task Assignment with Equal Load. The idea is simple: define the size range associated with each host such that the total work (load) directed to each host is the same. The motivation for doing this is that balancing the load minimizes mean waiting time.

The mechanism for achieving balanced expected load at the hosts is to use the *task size distribution* to define the cutoff points (defining the ranges) so that the expected work directed to each host is the same. The task size distribution is easy to obtain by maintaining a histogram (in the dispatcher unit) of all task sizes witnessed over a period of time.

More precisely, let $F(x) = \Pr\{X \leq x\}$ denote the cumulative distribution function of task sizes with finite mean $M$. Let $k$ denote the smallest task size, $p$ (possibly equal to infinity) denote the largest task size, and $h$ be the number of hosts. Then we determine "cutoff points" $x_i$, $i = 0 \ldots h$ where $k = x_0 < x_1 < x_2 < \ldots < x_{h-1} < x_h = p$, such that

$$\int_{x_0=k}^{x_1} x \cdot dF(x) = \int_{x_1}^{x_2} x \cdot dF(x) = \cdots = \int_{x_{h-1}}^{x_h=p} x \cdot dF(x) = \frac{M}{h} = \frac{\int_k^p x \cdot dF(x)}{h}$$

and assign to the $i$th host all tasks ranging in size from $x_{i-1}$ to $x_i$.

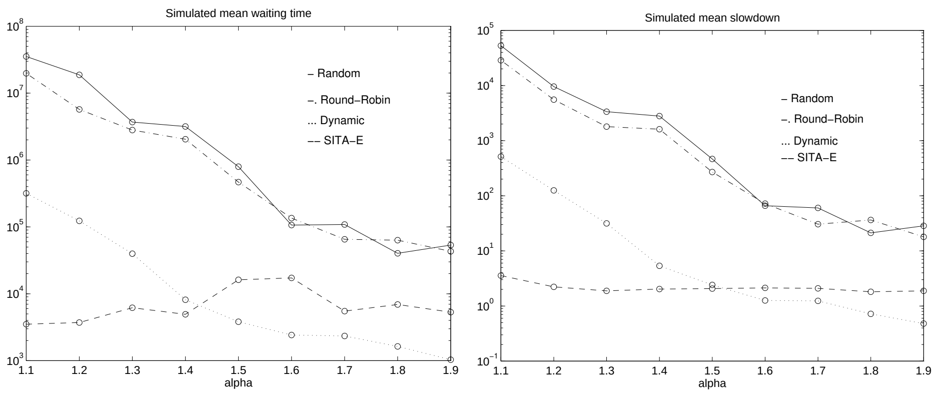
SITA-E as defined can be applied to *any* task size distribution with finite mean. In the remainder of this case study we will always assume the task size distribution is the Bounded Pareto distribution, $B(k, p, \alpha)$.

### 4.4 Simulation Results

In this section we compare the Random, Round-Robin, SITA-E, and Dynamic policies via simulation. Simulation parameters are as shown in Table 1.

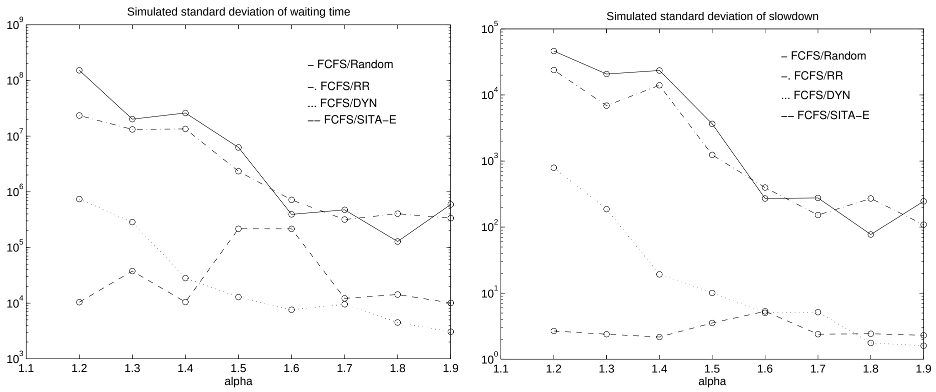
Simulating a server system with heavy-tailed, highly variable service times is difficult because the system approaches steady state very slowly and usually from below [3]. This occurs because the running average of task sizes is typically at the outset well below the true mean; the true mean isn't achieved until enough large tasks arrive. The consequence for a system like our own is that simulation outputs appear more optimistic than they would in steady-state. To make our simulation measurements less sensitive to the startup transient, we run our simulation for $4 \times 10^5$ arrivals and then capture data from the next single arrival to the system only. Each data point shown in our plots is the average of 400 independent runs, each of which started from an empty system.

We consider $\alpha$ values in the range 1.1 (high variability) to 1.9 (lower variability). As described in Section 2, $\alpha$ values in the range 1.0 to 1.3 tend to be common in empirical measurements of computing systems.




(a) (b)

Figure 7: Mean Waiting Time (a) and Mean Slowdown (b) under Simulation of Four Task Assignment Policies as a Function of $\alpha$.




(a) (b)

Figure 8: Standard Deviation of Waiting Time (a) and Standard Deviation of Slowdown (b) under Simulation of Four Task Assignment Policies as a Function of $\alpha$.

Figure 7 and 8 show the performance of the system for all four policies, as a function of $\alpha$ (note the logarithmic scale on the $y$ axis). Figure 7(a) shows mean waiting time and 7(b) shows mean slowdown. Below we simply summarize these results; in the next section, we will use analysis to explain these results.

First of all, observe that the performance of the system under the Random and Round Robin policies is similar, and that both cases perform much more poorly than the other two (SITA-E and Dynamic). As $\alpha$ declines, both of the performance metrics under the Random and Round-Robin policies explode approximately exponentially. This gives an indication of the severe impacts that heavy-tailed workloads can have in systems with naive task assignment policies.

The Dynamic policy shows the benefits of instantaneous load balancing. Dynamic is on the order of 100 times better for both metrics when compared to Random and Round Robin. For large $\alpha$, this means that Dynamic performs quite well—with mean slowdown less than 1. However as the variability in task size increases (as $\alpha \to 1$), Dynamic is unable to maintain good performance. It too suffers from roughly exponential explosion in performance metrics as $\alpha$ declines.

In contrast, the behavior of SITA-E is quite different from that of the other three. Over the entire range of $\alpha$ values studied, the performance of the system under SITA-E is relatively unchanged, with mean slowdown always between 2 and 3. This is the most striking aspect of our data: in a range of $\alpha$ in which performance metrics for Random, Round Robin, and Dynamic all explode, SITA-E's performance remains remarkably insensitive to increase in task size variability.

As a result we find that when task size is less variable, Dynamic task assignment exhibits better performance (an order of magnitude better than the SITA-E); but when task sizes show the variability that is more characteristic of empirical measurements ($\alpha \approx 1.1$), SITA-E's performance can be two orders of magnitude (100 times) better than that of Dynamic.

In [7] we simulate a range of loads ($\rho$) and show that as load increases, SITA-E becomes preferable to Dynamic over a larger range of $\alpha$.

The remarkable consistency of system performance under the SITA-E policy across the range of $\alpha$ from 1.1 to 1.9 is difficult to understand using the tools of simulation alone. For that reason the next section develops analysis of SITA-E and the other policies, and uses that analysis to explain SITA-E's performance.

### 4.5 Analysis of Task Assignment Policies

To understand the differences between the performance of the four task assignment policies, we provide a full analysis of the Round-Robin, Random, and SITA-E policies, and an approximation of the Dynamic policy.

In the analysis below we will repeatedly make use of the Pollaczek-Kinchin formula below which analyzes the M/G/1 FCFS queue:

$$\begin{aligned} \mathbf{E}\{\text{Waiting Time}\} &= \lambda \mathbf{E}\left\{X^2\right\}/2(1-\rho) \qquad \text{[Pollaczek-Kinchin formula]} \\ \mathbf{E}\{\text{Slowdown}\} &= \mathbf{E}\{W/X\} = \mathbf{E}\{W\} \cdot \mathbf{E}\left\{X^{-1}\right\} \end{aligned}$$

where $\lambda$ denotes the rate of the arrival process, $X$ denotes the service time distribution, and $\rho$ denotes the utilization ($\rho = \lambda \mathbf{E}\{X\}$). The slowdown formulas follow from the fact that $W$ and $X$ are independent for a FCFS queue.

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 6.

*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 $i$th host, is equal to the system load, that is, $\rho_i = \rho$. So the Pollaczek-Kinchin formula 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 case since it still sees high variability in service times, which dominates performance.

*SITA-E.* The SITA-E policy also performs Bernoulli splitting on the arrival stream (which follows from our assumption that task sizes are independent). By the definition of SITA-E, $\rho_i = \rho$. However the task sizes at each queue are determined by the particular values of the interval cutoffs, $\{x_i\}, i = 0, ..., h$. In fact, host $i$ sees a $M/B(x_{i-1}, x_i, \alpha)/1$ queue. The reason for this is that partitioning the Bounded Pareto distribution into contiguous regions and renormalizing each of the resulting regions to unit probability yields a new set of Bounded Pareto distributions. In [7] we show how to calculate the set of $x_i$s for the $B(k, p, \alpha)$ distribution, and we present the resulting formulas that provide full analysis of the system under the SITA-E policy for all the performance metrics.

*Dynamic.* The Dynamic policy is not analytically tractable, which is why we performed the simulation study. However, in [7] we prove that a distributed system of the type in this paper with $h$ hosts which performs Dynamic task assignment is actually equivalent to an M/G/h queue. Fortunately, there exist known approximations for the performance metrics of the M/G/h queue [15]:

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

where $X$ denotes the service time distribution and $Q$ denotes the number in queue. 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.

Using the above analysis we can compute the performance of the above task assignment policies over a range of $\alpha$ values. Figure 9 shows the analytically-derived mean waiting time and mean slowdown of the system under each policy over the whole range of $\alpha$. Figure 10 again shows these analytically-derived metrics, but only over the range of $1 \leq \alpha \leq 2$, which is the range of $\alpha$ corresponding to most empirical measurements of process lifetimes and file sizes (see Section 2). (Note that, because of slow simulation convergence as described at the beginning of Section 4.4, simulation values are generally lower than analytic predictions; however all simulation trends agree with analysis).

First observe that the performance of the Random and Dynamic policies in both these figures grows worse as $\alpha$ decreases, where the performance curves follow the same shape as the second moment of the Bounded Pareto distribution, shown in Figure 6. This is expected since the performance of Random and Dynamic is directly proportional to the second moment of the service time distribution. By contrast, looking at Figure 10 we see that in the range $1 < \alpha < 2$, the mean waiting time and especially mean slowdown under the SITA-E policy is remarkably constant, with mean slowdowns around 3, whereas Random and Dynamic explode in this range. The insensitivity of SITA-E's performance to $\alpha$ in this range is the most striking property of our simulations and analysis.

Why does SITA-E perform so well in a region of task size variability wherein a Dynamic policy explodes? A careful analysis of the performance of SITA-E at each queue of the system (see [7]) leads us to the following answers:

1. By limiting the range of task sizes at each host, SITA-E greatly reduces the variance of the task size distribution witnessed by the lowered-numbered hosts, thereby improving performance at these hosts. In fact the performance at most hosts is superior to that of an M/M/1 queue with utilization $\rho$.

2. When load is balanced, the majority of tasks are assigned to the low-numbered hosts, which are the hosts with the best performance. This is intensified by the heavy-tailed property which implies that *very* few tasks are assigned to high numbered hosts.

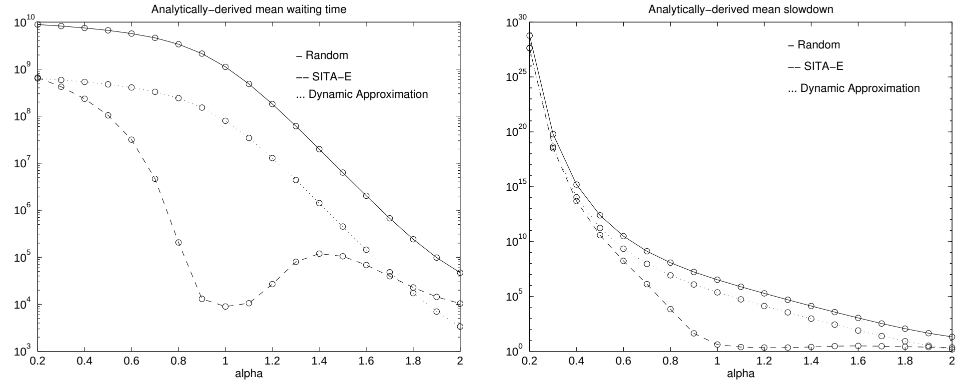



Figure 9: Analysis of mean waiting time and mean slowdown over whole range of $\alpha$, $0 < \alpha \leq 2$.

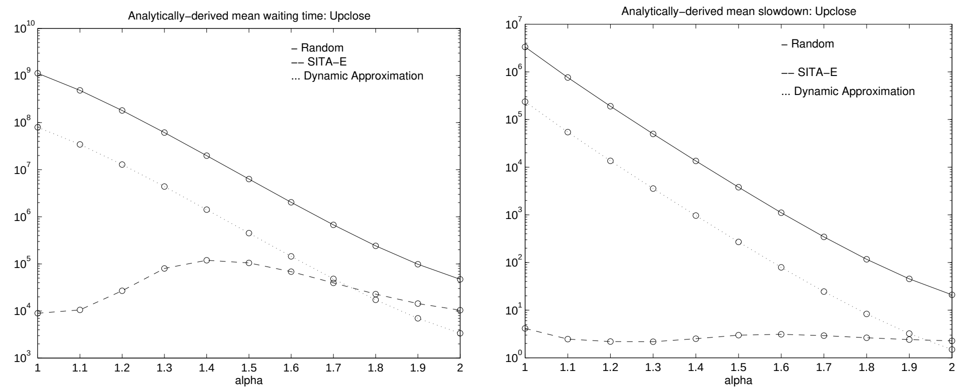



Figure 10: Analysis of mean waiting time and mean slowdown over empirically relevant range of $\alpha$, $1 \leq \alpha \leq 2$.

3. Furthermore, mean slowdown is improved because small tasks observe proportionately lower waiting times.

For the case of $\alpha \leq 1$, shown in Figure 9, even under the SITA-E policy, system performance eventually deteriorates badly. The reason is that as overall variability in task sizes increases, eventually even host 1 will witness high variability. Further analysis [7] indicates that adding hosts can extend the range over which SITA-E shows good performance. For example, when the number of hosts is 32, SITA-E's performance does not deteriorate until $\alpha \leq .8$.

### 4.6 Conclusion

We have studied how the task size distribution influences which task assignment policy is best in a distributed system. We considered four policies: Random, Round-Robin, SITA-E (a size-based policy), and Dynamic (sending the task to the host with the least remaining work).

We have found that the best choice of task assignment policy depends critically on the variability of task size distribution. When the task sizes are not highly variable, for example in the case of an exponential distribution, the Dynamic policy is preferable – by as much as an order of magnitude in performance. However, when task sizes are heavy-tailed with $\alpha \approx 1$ as is common in empirical measurements (see Section 2), SITA-E is best – by as much as two orders of magnitude in performance.

## 5 Case study 3: Scheduling in a Web server

The discussion in this section is taken entirely from the papers: [8, 1]. We present here a brief overview of the material presented in those papers.

Consider a Web server servicing multiple connections (jobs). Traditionally, the Web server will time-slice between the connections. Each resource (CPU and I/O) is "shared fairly" between the open connections. However, it may be possible to achieve better mean response time (mean over all jobs) by using a different scheduling policy – one which favors small jobs.

Heretofore, Web servers have not obtained or utilized information about the size of the job (it's resource requirement). However the size of a job is in fact easy information to gauge for *static* requests (requests which simply GET a file), since the "size" of a job is then proportional to the size of the file being retrieved.

Given that the size of a job is in fact obtainable, we propose changing the scheduling policy in Web servers to give preference to small jobs (connections) over big ones. More specifically, [8, 1] propose giving preference to jobs whose *remaining size* (remaining service demands) are smallest.

The motivation for this type of scheduling is the well-known scheduling theorem that states that for *single* resource scheduling, always running the job with the shortest remaining processing time (SRPT) will minimize mean response time. SRPT minimizes mean response time because it allows small jobs to get out fast. However, observe that this scheduling theorem applies to single resource scheduling. By contrast, a Web server has multiple resources and requires that multiple jobs be present in the system at once in order to achieve good throughput. In the context of a Web server it is not even obvious what SRPT scheduling means.

In [1] we first define what is meant by giving preference to jobs whose remaining size is smallest in the context of a Web server. We then build a Web server and show that this type of SRPT-like scheduling policy in fact leads to substantial performance improvements for Web servers with respect to mean response time.

We will not attempt to regurgitate all the issues brought up in [8, 1] since these are long detailed papers. Instead we will concentrate on just one particular issue: *starvation*. If the smaller jobs are getting preference, does that not imply that the bigger jobs will be starved? In [8, 1] we show that this does not in fact happen. In fact, long connections pay very little penalty. This surprising result can be understood by looking at the *job size distribution*, which is of interest to the current paper:

Consider a job in the 99th percentile of the job size distribution (i.e. a very large job). It turns out that such a job has lower expected slowdown when the scheduling policy is SRPT-like than under a fair scheduling (Processor-Sharing) type of policy. To see this, recall from Section 2 that the sizes of requests arriving at a Web server have been shown to have a heavy-tailed distribution. Now consider a job $j$ in the 99th percentile of the job size distribution. By the heavy-tailed property (see Section 2), more than half the total workload is contained in jobs of size greater than $j$. Thus job $j$ is preempted by less than half the total workload, which in turn implies (see [8]) that $j$'s expected response time is actually better under SRPT-like scheduling than under a Processor-Sharing type of scheduling where job $j$ would have to share the resource with the total workload. By contrast, in the case of an exponential distribution only 5% of the total workload is contained in jobs of size greater than $j$. Thus under an exponential workload, job $j$ would be held up by over 95% of the workload and would in fact have significantly worse performance under an SRPT-like scheduling policy than under a processor-sharing-like scheduling policy. Thus for an exponential workload, SRPT-like scheduling is not a good idea.

The above argument holds as well for a job in the 99.5th percentile. The bottom line is that when the job size distribution is heavy-tailed, starvation under SRPT-like scheduling is provably not a problem for 99.5% of the jobs, and thus isn't observable in practice.

In conclusion, on the question of choosing the best scheduling policy, understanding the job size distribution is critical to determining the best solution.

## 6 Conclusion

In this paper we have discussed three case studies in the area of computer system design. In each case study we have shown that the job size distribution has a great effect on the problem solution. In particular a heavy-tailed job size distribution leads to very different answers from an exponential job size distribution. Furthermore, the optimal solution under the assumption of an exponential job size distribution is far from optimal (sometime by orders of magnitude) when the workload is heavy-tailed.

This is unfortunate for two reasons: First of all, solving these problems analytically is often far easier under an exponential job size distribution. Second, in each of these case studies, our natural intuition tends to lead us to the solution which is best under an exponential job size distribution, rather than to the solution which is best under a heavy-tailed job size distribution. With respect to the former problem, we have shown that (at least with respect to these three case studies) analysis is sometimes also possible under heavy-tailed job size distributions. To combat the latter problem, we have characterized heavy-tailed distributions in terms of three properties which are especially important to think about when solving problems in system design: 1) decreasing failure rate, 2) greater variance than imaginable, and 3) the heavy-tailed property – a miniscule fraction of the biggest jobs account for more than half the total workload.

It is our belief that increasingly computer workload measurements will show the presence of heavy-tails. This will force us to reevaluate age-old truisms in the area of system design and develop new ones.

## References


[1] M. E. Crovella, B. Frangioso, and M. Harchol-Balter. Connection scheduling in Web servers. Technical Report BUCS-TR-99-003, BU Computer Science Department, March 1999.

[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 and Lester Lipsky. Long-lasting transient conditions in simulations with heavy-tailed workloads. In *Proceedings of the 1997 Winter Simulation Conference*, pages 1005–1012, 1997.

[4] Mark E. Crovella, Murad S. Taqqu, and Azer Bestavros. Heavy-tailed probability distributions in the World Wide Web. In Robert J. Adler, Raisa E. Feldman, and Murad S. Taqqu, editors, *A Practical Guide To Heavy Tails*, chapter 1, pages 3–26. Chapman & Hall, New York, 1998.

[5] John Doyle. Highly optimized tolerance: A mechanism for power laws in designed systems. From talk slides presented at MIT, March 1999.

[6] M. Harchol-Balter, M. E. Crovella, and C. D. Murta. On choosing a task assignment policy for a distributed server system. *Proceedings of Performance Tools '98. Lecture Notes in Computer Science*, 1469:231–242, 1998.

[7] M. Harchol-Balter, M. E. Crovella, and C. D. Murta. On choosing a task assignment policy for a distributed server system. Technical Report MIT-LCS-TR-757, MIT Laboratory for Computer Science, 1998.

[8] M. Harchol-Balter, M. E. Crovella, and S. Park. The case for SRPT scheduling in Web servers. Technical Report MIT-LCS-TR-767, MIT Lab for Computer Science, October 1998.

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

[10] Steven Hotovy, David Schneider, and Timothy O'Donnell. Analysis of the early workload on the cornell theory center ibm sp2. Technical Report 96TR234, Cornell Theory Center, January 1996.

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

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

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