Available online at www.sciencedirect.com
ScienceDirect
Performance Evaluation 65 (2008) 606–622
PERFORMANCE EVALUATION
An International Journal
www.elsevier.com/locate/peva

Effective load balancing for cluster-based servers employing job preemption☆

Victoria Ungureanu[a], Benjamin Melamed[b,*], Michael Katehakis[c]

[a] DIMACS Center, Rutgers University, 96 Frelinghuysen Road, Piscataway, NJ 08854, United States
[b] Department of MSIS, Rutgers University, 94 Rockafeller Road, Piscataway, NJ 08854, United States
[c] Department of MSIS, Rutgers University, 180 University Ave., Newark, NJ 07102, United States

Received 1 September 2005; received in revised form 29 May 2007; accepted 13 January 2008
Available online 25 January 2008

Abstract

A cluster-based server consists of a front-end dispatcher and multiple back-end servers. The dispatcher receives incoming jobs, and then decides how to assign them to back-end servers, which in turn serve the jobs according to some discipline. Cluster-based servers have been widely deployed, as they combine good performance with low costs.

Several assignment policies have been proposed for cluster-based servers, most of which aim to balance the load among back-end servers. There are two main strategies for load balancing: The first aims to balance the *amount of workload* at back-end servers, while the second aims to balance the *number of jobs* assigned to back-end servers. Examples of policies using these strategies are `Dynamic` and `LC` (Least Connected), respectively.

In this paper we propose a policy, called `LC*`, which combines the two aforementioned strategies. The paper shows experimentally that when preemption is admitted (i.e., when jobs execute concurrently on back-end servers), `LC*` substantially outperforms *both* `Dynamic` and `LC` in terms of response-time metrics. This improved performance is achieved by using only information readily available to the dispatcher, rendering `LC*` a practical policy to implement. Finally, we study a refinement, called `ALC*` (Adaptive `LC*`), which further improves on the response-time performance of `LC*` by adapting its actions to incoming traffic rates.
Published by Elsevier B.V.

Keywords: Cluster-based servers; Back-end server architecture; Job preemption; Simulation

1. Introduction

Web servers are becoming increasingly important as the Internet assumes an ever more central role in the telecommunications infrastructure. Applications that handle heavy loads commonly use a *cluster-based* architecture for Web servers because it combines low costs with good performance. A cluster-based server consists of a front-end dispatcher and several back-end servers (see Fig. 1). The dispatcher receives incoming jobs and then decides how to

☆ A preliminary version of this paper appeared in the *Proceedings of the IEEE International Symposium on Network Computing and Applications (IEEE NCA04)*.

* Corresponding author. Tel.: +1 732 445 3128; fax: +1 732 445 6329.
E-mail addresses: ungurean@dimacs.rutgers.edu (V. Ungureanu), melamed@rbs.rutgers.edu (B. Melamed), mnk@andromeda.rutgers.edu (M. Katehakis).

0166-5316/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.peva.2008.01.001

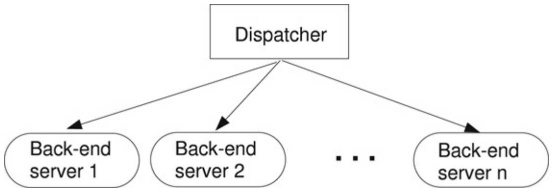


Fig. 1. Architecture of a cluster-based server.

assign them to back-end servers, which in turn process the jobs according to some discipline. The dispatcher is also responsible for passing incoming data pertaining to a job from a client to a back-end server. Accordingly, for each job in progress at a back-end server there is an open connection between the dispatcher and that server.

Several dispatcher assignment policies have been proposed for this type of architecture (see, e.g., [3,16,18,21]). Generally, these policies attempt to balance the load among back-end servers according to two main strategies: balancing the *amount of workload* at back-end servers, and balancing the *number of jobs* being processed by back-end servers.

Policies balancing the amount of workload. Well-known policies that fall into this category include `Dynamic` [20] and `Size-Range` [10,5,17]. Under `Dynamic`, the dispatcher assigns an incoming job to the back-end server with the smallest *amount* of residual workload, i.e., the sum of service demands of all jobs in the server queue plus the residual workload of the jobs currently being served. This policy is optimal from the standpoint of an individual task [10], provided that the following assumptions hold: (a) jobs arrive at the dispatcher according to a Poisson process; (b) job sizes follow an exponential distribution; and (c) jobs are served FCFS (first-come first-served) at each server queue. However, the optimality of `Dynamic` comes with several caveats. First, `Dynamic` requires the dispatcher to know the status of all back-end servers at all times. This type of information is difficult, if at all possible, to acquire, and consequently, `Dynamic` has not been used by commercial products [15,19]. Second, there is considerable evidence that the sizes of files traveling on the Internet do not follow an exponential-type distribution. Rather, file sizes appear to follow power-law (heavy-tailed) distributions [2,7,9] of the form $\mathbb{P}[X > x] \sim \frac{c}{x^{\alpha}}$, where X is the random file size, $c > 0$, and $1 \le \alpha \le 2$. For power-law job-size distributions, a relatively small fraction of jobs accounts for a relatively large fraction of the overall load.

In contrast with `Dynamic`, `Size-Range` policies were designed for heavy-tailed distributions. Examples of this type of policy include `SITA-E` [10], `EquiLoad` [5], and `AdaptLoad` [17]. These policies are motivated by the observation that when "small" jobs are stuck behind "large" jobs, then response-time performance suffers degradation. Such situations can be avoided if any back-end server is assigned only jobs of similar sizes. Specifically, if a cluster consists of n back-end servers, then n size ranges, $[s_0, s_1),\ [s_1, s_2), \ldots, [s_{n-1}, s_n)$, are determined so that each range contains approximately the same amount of workload. Accordingly, when a request in the range $[s_j, s_{j+1})$ is received, the dispatcher assigns it to back-end server j. It was shown experimentally that when the sizes of requested files follow a heavy-tailed distribution, then `Size-Range` performs similarly to or better than `Dynamic` [17].

Policies balancing the number of jobs. A well-known policy that falls into this category is `Least-Connected (LC)`. Under `LC`, the dispatcher assigns a job to the back-end server currently processing the smallest number of jobs, i.e., the one having the least number of open connections with the dispatcher. Implementing this policy requires the dispatcher to obtain little information; moreover, this information is easy to acquire.

On the performance of assignment policies. Numerous performance studies compare assignment policies experimentally. To the best of our knowledge, all of these studies show that balancing the amount of workload is a better strategy than balancing the number of jobs. Indeed, these results reinforce the intuition that using more information leads to better assignments, and hence, to better performance. However, these studies preclude *job preemption*, i.e., they assume that jobs execute to completion, in FCFS order.

In contrast, this paper studies the performance of the aforementioned policies, with and without job preemption at back-end servers. The incorporation of preemption is motivated by the fact that in practice, most, if not all, back-end servers use preemption to process jobs concurrently (see Section 2.1 for further details). This paper shows empirically that preemption affects dramatically the performance of the policies considered: the performance of policies balancing the number of jobs improves, while the performance of policies balancing the workload amount worsens. More specifically, we found the following:

- When preemption is precluded, Size-Range outperforms LC. However, when preemption is allowed, LC outperforms Size-Range substantially.
- When preemption is precluded, Dynamic outperforms LC as expected. However, when preemption is allowed, LC outperforms Dynamic by a factor of two. Moreover, LC with preemption still outperforms Dynamic without preemption by a substantial margin. This suggests that deploying LC with preemptive back-end servers is not only a practical choice, but also yields better performance.

The results of the aforementioned study suggest that if jobs can be preempted, then balancing the number of jobs at back-end servers is a better strategy than balancing the amount of workload there. On the face of it, this contention appears to be counter-intuitive, because the latter strategy uses more information than the former. However, our results lend support to the contention that this apparently more detailed information is not as relevant as intuition would lead us to believe. We will elaborate on this point in Section 2.

LC—a policy combining the two load balancing strategies.* An interesting question in this context is whether *combining the two strategies* would further improve the performance of a server cluster employing preemption. In this vein, we propose a new policy, called LC*, which aims to balance the number of jobs at back-end servers in a manner that avoids creating large disparities in the amount of workload at them. In a nutshell, LC* operates as follows: The dispatcher uses a threshold parameter to classify incoming jobs into *small* and *large*, and then small jobs are assigned to the least connected back-end server, while large jobs are assigned to a back-end server, not currently processing a large job. In particular, if all back-end server-queues contain a large job, then the assignment of an incoming large job is deferred until a back-end server completes its large job.

The proposed LC* policy does not achieve perfect balancing of the number of jobs nor the amount of workload. However, we argue heuristically that LC* tends to give rise to only relatively minor disparities in these metrics across back-end servers. First, there are no large disparities in the amount of workload, because a back-end server queue may contain at most one large job at any given time. Second, there are only minor disparities in the number of jobs: at any given time, the maximal possible difference in the number of jobs assigned to any two back-end server is 2. To see that, note that under LC, the maximal possible difference is 1. However, because, LC* does not require a large job to be assigned to the least connected server, the maximal possible difference may increase, but the increase is limited to 2, since only one long job may be processed by a back-end server at any given time.

To gauge the response-time performance of the LC* policy, we exercised it on empirical data traces measured at Internet sites serving the 1998 World Cup. We mention that Arlitt and Jin [2] show that job sizes from these traces do indeed follow a power-law distribution with $\alpha = 1.37$. In particular, for the trace considered in this paper, files with sizes greater than 30 kB make up less than 3% of the files requested, but account for over 50% of the overall workload. We show that when files in excess of 30 kB are classified as large, LC* outperforms substantially the other policies considered (LC, Dynamic and Size-Range). Thus, the study demonstrates that a careful assignment of a small number of jobs can have a profound impact on overall response-time performance. It is worth pointing out that the observed improvement in performance is achieved by using only information readily available to the dispatcher. Consequently, LC* is a practical policy with regard to implementation.

The rest of the paper is organized as follows. Section 2 discusses the effects of preemption on the performance of the aforementioned policies by presenting a performance study based on World Cup data traces. Section 3 presents in detail the LC* policy and illustrates its performance. Section 4 presents an adaptive version of LC*, called ALC*, and discusses its performance using various data traces. Finally, Section 5 concludes the paper.

2. The impact of preemption on the performance of assignment policies

This section presents a simulation study driven by real-life data traces, which shows that when back-end servers admit preemption (i.e., jobs execute concurrently), the strategy of balancing the *number of jobs* at back-end servers outperforms the strategy of balancing the *amount of workload*.

The study compares Dynamic, Size-Range and LC. To compare the performance of assignment policies, we simulated a cluster of four back-end servers, driven by a World Cup trace to be described below. The experiments were subject to the following assumptions: communication times between the dispatcher and back-end servers and the overhead incurred by the dispatcher to select (job, server) pairs are negligible, and all requests are static and served from cache. We will relax these assumptions in Section 4, which studies the effect of the dispatcher overhead, cache misses and dynamic requests on the performance of these policies.

2.1. Back-end server scheduling policies

The study compares the performance of assignment policies under two scenarios:

- Back-end servers employ the non-preemptive scheduling discipline FCFS. Under this policy, a back-end server processes to completion jobs assigned to it in the order of their arrival. We mention that this policy is rarely, if at all, used in practice. (Possibly the only realistic scenario where it can be assumed that jobs are executed in FCFS order is when back-end servers are multi-threaded and use non-preemptive scheduling for threads [6]).
- Back-end servers employ the preemptive scheduling discipline Round-Robin. Under the Round-Robin discipline, tasks assigned to a back-end server are processed in turn for an equal pre-defined time interval, or quota. When a task finishes its quota, it is suspended and the next task is allowed to execute (for the same pre-defined quota).

It is worth mentioning that there are other preemptive scheduling policies, most notably SRPT (Shortest Remaining Processing Time). Under SRPT, a back-end server selects for execution the task with the shortest remaining processing time. It was shown experimentally in [11] that SRPT has excellent performance. Moreover, it was shown in [8] that multi-layered Round-Robin at the dispatcher followed by SRPT at back-end servers is optimal under certain assumptions.[1] However, SRPT is rarely used outside of specialized environments because it requires accurate estimates of the runtime of all processes waiting to execute. In contrast, Round-Robin is very simple to implement and commonly used in practice.

2.2. Simulation data

Our study used trace data from Internet sites serving the 1998 World Cup. The data used are available from the Internet Traffic Archive (see [2] and http://ita.ee.lbl.gov/html/traces.html). This repository provides detailed information about the 1.3 billion requests received by World Cup sites over 92 days—from April 26, 1998, to July 26, 1998.

We mention again that Arlitt and Jin [2] have shown that job sizes from these traces follow a heavy-tailed distribution. To further highlight the power-law distribution of job sizes, we point out the following aspects of this trace:

- Approximately 75% of the files requested have sizes less than 2 kB, which account for less than 12% of the transferred data.
- Files with sizes greater than 30 kB, which make up less than 3% of the files requested, account for over 50% of the transferred data. Even more striking, files in excess of 100 kB, which make up less than 0.04% of all files requested, account for 7% of the transferred data.

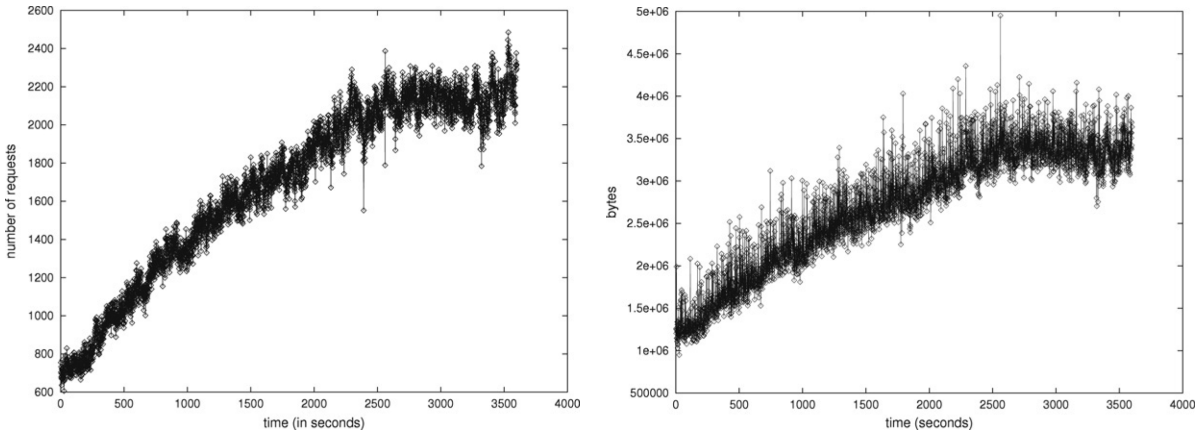
From this repository, we selected a trace covering 1 hour from the June 26 data, containing approximately 6 million requests. The relevant statistics of this trace are described next. Fig. 2(a) depicts the number of requests received by the World Cup cluster in successive one-second intervals, while Fig. 2(b) plots the number of bytes requested from the same cluster in successive one-second intervals.

The selection of the particular data trace was motivated by the fact that it exhibits arrival-rate fluctuations corresponding to light, medium and heavy loadings in this temporal order, as evidenced by Fig. 2. More specifically, the trace allows us to study policy performance under various loading conditions, as follows:

- *Light loading.* In the time interval [0, 1200], the arrival rate is relatively low (below 1200 requests/s), and the resultant utilization coefficient is also low ($\approx$40%).
- *Medium loading.* In the time interval (1200, 2400], the arrival rate is between 1200 and 1800 requests/s, and the resultant utilization coefficient is intermediate.
- *Heavy loading.* In the time interval (2400, 3600], the arrival rate often exceeds 2000 requests/s, and the corresponding utilization coefficient is high ($\approx$75%).

From each trace record, we extracted only the request arrival time and the name and size of the requested file. We mention that the recorded time stamps are in integer seconds, with arrivals on the order of several hundreds of requests per second. Consequently, we have distributed request arrivals uniformly over each second. Since no service

[1] Jobs arrive according to a Poisson process, and their processing times follow a discrete distribution with finite support.



(a) Number of request arrivals per second.

(b) Total bytes requested per second.

Fig. 2. Empirical request time series from a World Cup trace.

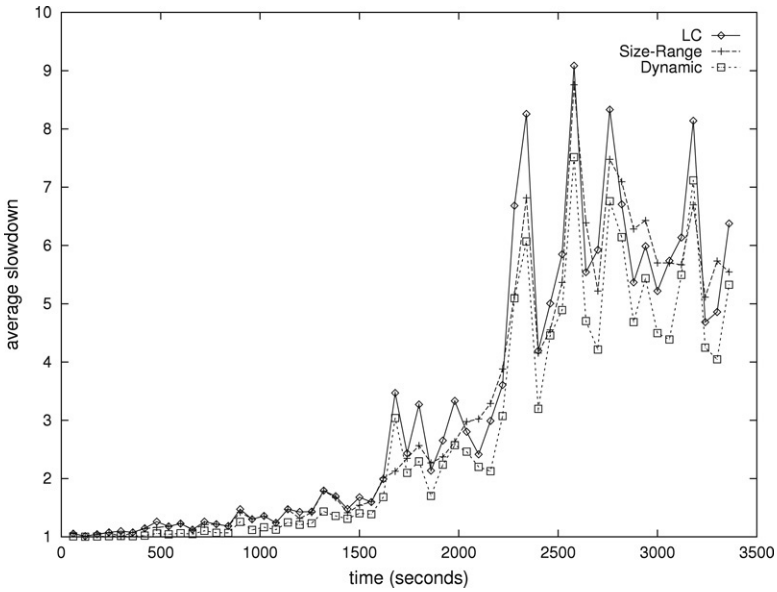


Fig. 3. Successive average slowdowns of `Dynamic`, `LC` and `Size-Range` on a cluster with four back-end servers, when jobs are not preempted.

time information was recorded, the simulation estimates the service time as the sum of the (constant) time to establish and close a connection and the (variable, size-dependent) time required to retrieve and transfer a file. The justification for this estimation method may be found in [15,17,19].

2.3. Simulation experiments

The performance metric used to compare assignment policies is *slowdown*, defined as the ratio of a job's response time (the time interval from the moment a job arrives at the dispatcher and up until it ends processing at the corresponding back-end server) and its service time.

2.3.1. Job processing without preemption

Fig. 3 displays average slowdowns in successive one-second intervals for the three assignment policies considered, under the assumption that jobs are not preempted (i.e., back-end servers use the FCFS scheduling discipline). The figure shows that `Dynamic` outperforms `LC` over all time intervals, and therefore, under all simulated loading regimes.

Fig. 4. Server queues under (a) `Dynamic` and `Size-Range`, and (b) under `LC`.

`Size-Range` performs very well under low and medium loadings: it outperforms `LC` in all intervals, and it even outperforms `Dynamic` in several time intervals. However, its performance degrades as the load increases: for high loading, `Size-Range` still outperforms `LC` most of the time, but there are a few time intervals where `LC` performs slightly better.

These observations can be explained by the fact that if preemption is precluded, then performance degrades considerably when small jobs are stuck after a large job, as small jobs cannot start before the large one finishes. This situation is best avoided by `Dynamic` and `Size-Range`, which attempt to balance the amount of workload at back-end servers. In contrast, this situation occurs more often under `LC`, which balances the number of jobs.

To gain insight into this explanation, consider the scenario, where a sequence of jobs, J1, ..., J100, arrives back-to-back at a cluster consisting of two back-end servers, S1 and S2. Assume that J1 requires a service time of 100 seconds, while all others require a service time of 1 second.

Under `Dynamic` and `Size-Range`, S1 is assigned the large job, and S2 is assigned all the small jobs (see Fig. 4(a)). This way, the amount of workload is evenly distributed among the two back-end servers. Then, the large job will be completed in 100 seconds, and the small jobs will be completed in 1, 2, ..., 99 seconds, respectively. It follows that the slowdown of the large job is 1, while the slowdowns of the small jobs are 1, 2, ..., 99. Consequently, the average slowdown is approximately 50.

Under `LC` one of the back-end servers is assigned the large job and 49 small ones, while the other is assigned 50 small jobs (see Fig. 4(b)). This way both back-end servers are assigned the same number of jobs. Then, the response times at the first back-end server are 100, 101, ..., 149 seconds, and those at the second server are 1, 2, ..., 50 seconds. Therefore the slowdowns incurred by the jobs assigned at the first back-end server are 1, 101, 102, ..., 149, while the slowdowns incurred by the jobs assigned to the second server are 1,2, ..., 50. Consequently, the average slowdown in this case is approximately 75.

2.3.2. Job processing with preemption

The corresponding experiments with preemption simulate back-end servers that schedule jobs using the Round-Robin scheduling discipline. Fig. 5 displays average slowdowns in successive 60-second intervals. Interestingly, under these assumptions, `LC` outperforms both `Dynamic` and `Size-Range` over *all* time intervals. Moreover, the relative advantage of `LC` over `Dynamic` and `Size-Range` increases as the loading increases. Under light loading, their performance is quite close, but under heavy loading, `LC` outperforms `Dynamic` by as much as a factor of 3.

These observations can be explained by the fact that performance is primarily affected here by the number of jobs contending for resources. To gain insight into this explanation, consider the previous scenario, where the sequence of jobs, J1, ..., J100, arrives back-to-back at a cluster consisting of two back-end servers, S1 and S2 (see Fig. 4), and assume that processes are scheduled for execution in Round-Robin manner, with a quota of 0.1 second. Under `Dynamic` and `Size-Range`, the average slowdown incurred is approximately 95 seconds. To see why, note that the response times (in seconds) for the small jobs are 90, 90.1, 90.2, ..., 100, respectively, and therefore, their slowdowns are 90, 90.1, 90.2, ..., 100, respectively. By contrast, the large job incurs a slowdown of only 1, since it is served in 100 seconds.

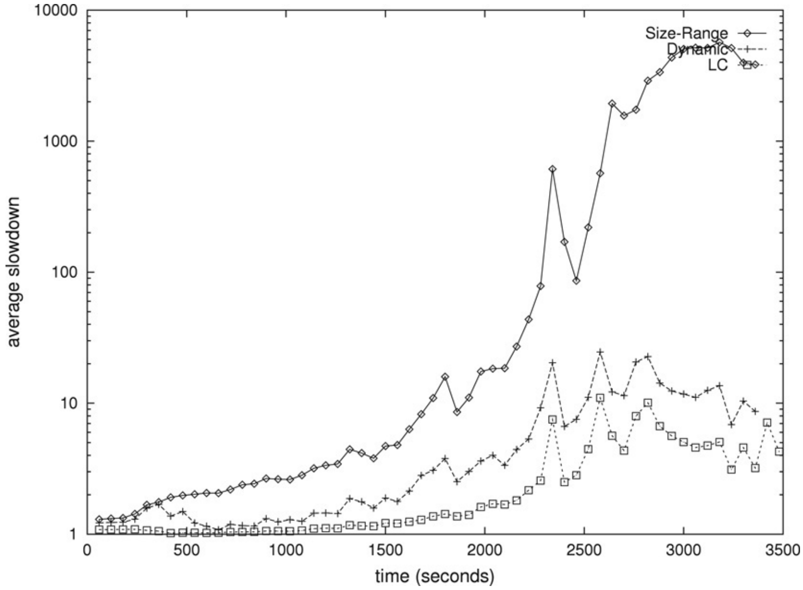


Fig. 5. Successive average slowdowns for `Dynamic`, `LC` and `Size-Range` on a cluster with four back-end servers, when jobs are preempted (note the logarithmic scale on the vertical axis).

Under `LC`, the average slowdown is approximately 47 seconds. To see why, note that the response times (in seconds) of small jobs assigned to S2 are just 45, 45.1, 45.2, ..., 50. Small jobs assigned to S1 have similar response times, while the response time of the large job is 150 seconds, and therefore, its slowdown is 1.5. Consequently, the average slowdown incurred (≈47) is better by a factor of 2 as compared to `Dynamic` and `Size-Range`. Intuitively, the improvement is due to the fact that the average response time is dominated by small jobs, and `LC` arranges to double the throughput of small jobs as compared to the other two policies.

This example leads us to contend that the larger the variability in the number of jobs across server queues, the poorer the performance of `Dynamic` and `Size-Range`. This contention is supported by our simulation experiments: `Dynamic` performs far worst than `LC` under heavy loading (where the length of back-end server queues and their variability tend to be large) than under light loading (where the length of server queues and their variability tend to be small).

However, this example fails to explain why `Size-Range` performs so poorly in comparison to `Dynamic`, especially under very high loading. The intuitive explanation is that, by design, `Size-Range` induces variability in server queues, while `Dynamic` does not. To see why, recall that `Size-Range` clusters jobs of similar sizes and assigns them to dedicated back-end servers. For high loadings, the back-end server that serves the largest jobs will have relatively few jobs in its queue, while the back-end server that serves the smallest ones will have relatively many jobs. In contrast, `Dynamic` simply attempts to balance the number of jobs without segregating them in different servers by size. Consequently, server queues tend to be less disparate in size.

To motivate this contention consider again that a sequence of jobs, J1, ..., J100, arrives back-to-back at a cluster consisting of two back-end servers, S1 and S2. However, their sizes are slightly modified, namely, assume now that J1 and J2 require a service time of 50 seconds, while all others require a service time of just 1 second. `Size-Range` assigns the two large jobs to one server, and the small jobs to the other, thus creating a large disparity in server queue sizes. In contrast, `Dynamic` assigns to both servers a large job and 49 small ones, thus balancing both the queue sizes and the amount of workload there. It is worth mentioning that the same assignment is produced by `LC`.

We conclude this section by comparing the average performance of the `Dynamic` and `LC` policies with and without preemption (see Table 1). We make three observations based on the table data. First, `Dynamic` yields better performance without preemption than with preemption. Second, `LC`'s behavior is opposite, namely, it yields better performance with preemption than without preemption. Finally, `LC` with preemption outperforms `Dynamic` without preemption.

Table 1
Comparison of statistics for Dynamic and LC

Policy	Average slowdown	
	No job preemption	Job preemption
Dynamic	4.09	6.84
LC	4.7	3.53

3. The LC* policy

The experiments and the examples presented in the previous section support the contention that if jobs are executed concurrently, then aiming to balance the number of jobs assigned to back-end servers is more important than balancing the amount of workload there. We now proceed to demonstrate experimentally that response-time performance may be further improved if *balancing the number of jobs is combined with balancing the amount of workload*.

To support this claim, consider the following motivating example. Suppose that two large jobs and two small jobs arrive back-to-back at a cluster consisting of two back-end servers. Then, under LC, there are two possible assignments of these jobs. The first assigns a large job and a small job to each back-end server, while the second assigns the two large jobs to one back-end server and the two small ones to the other. Note that the first assignment balances the amount of workload, while the second does not. To gauge how this disparity affects response-time performance, assume that the service times of large and small jobs are 100 seconds and 1 second, respectively. Assume further that processes are scheduled for execution in Round-Robin manner, with a quota of 0.1 second. Thus, the response time of each large job under the first assignment is ≈100 seconds, while under the second assignment, it is ≈200 seconds. This outcome is due to the large disparity of workload across back-end servers under the second assignment.

This example suggests that balancing both the number of jobs and the amount of workload may lead to improved performance. It further suggests that assigning multiple large jobs to the same back-end server exacts a large response-time penalty, and therefore, such assignments should be avoided. These observations motivate the hybrid LC* policy, defined as follows:

1. The dispatcher classifies incoming jobs into large or small relative to a cutoff parameter C.
2. The dispatcher assigns small jobs to the least connected back-end server.
3. The dispatcher assigns a large job to a back-end server whose queue does not already contain a large job. If there is no such back-end server, the dispatcher defers the assignment until a back-end server completes its large job.

We draw the reader's attention to the following points. First, the classification into large and small jobs is based on the *size of requested documents*. Although *job service time* is pro forma a more relevant classification metric, we nevertheless chose *job size*, because the dispatcher has this type of information readily available. Furthermore, it has been shown that the time to serve a request is well approximated by the size of the requested file [11]. The actual selection of the cutoff parameter is made in such a way that requests for files whose sizes are bigger than C represent only a small percent of incoming requests, but a large fraction of the overall workload (the functional form of a power-law distribution facilitates that choice). For example, the trace-driven World Cup simulation used 30K as the value of C.

Secondly, the dispatcher can estimate job sizes only for *static* requests (dynamic files are created on the fly by the server in response to a request). Consequently, LC* implicitly treats dynamic requests as small jobs. Even though this classification may be inaccurate for certain dynamic requests, we argue that for web servers that receive mostly static requests (such as news and e-commerce servers), the errors incurred do not substantially affect the performance of LC*. This point will be discussed in greater detail in Section 4.3.

Finally, LC* is practical to implement in that the extra information required is readily available to the dispatcher, and the processing of this information is quite fast.

We next proceed to demonstrate experimentally, via a case study, that LC* outperforms both Dynamic and LC. The study simulates a cluster of four back-end servers that process the jobs recorded by the trace of Section 2.2. The simulation sets the value of the cutoff parameter to 30K. Recall that for the trace considered, files with sizes greater than 30 kB (which make up less than 3% of the files requested) account for over 50% of the overall workload.

Fig. 6 displays average slowdowns of LC and LC* in successive 60-second intervals. Table 2 displays slowdown averages under various loading regimes (light, medium and heavy), as well as over the entire time horizon.

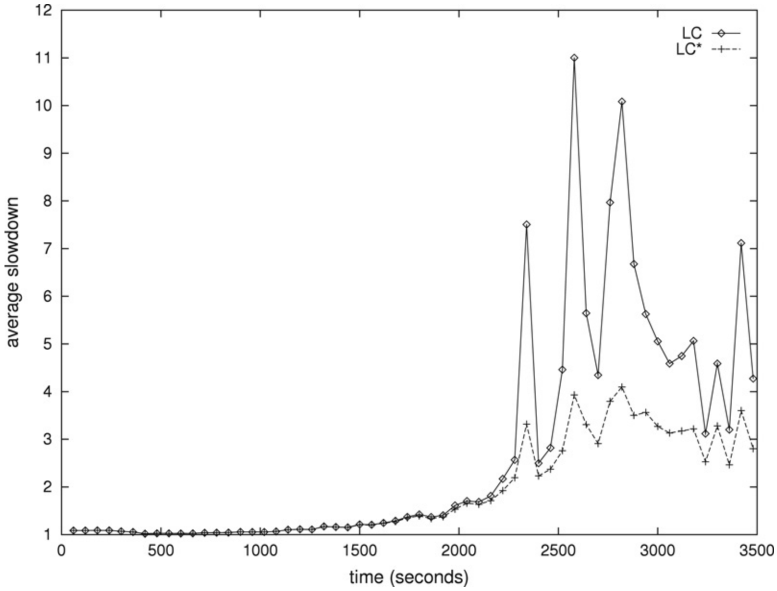


Fig. 6. Successive average slowdowns for LC and LC* on a cluster with four back-end servers.

Table 2
Comparative statistics for LC and LC*

Policy	Average slowdown			
	Light loading	Medium loading	Heavy loading	Overall
LC	1.06	1.90	6.17	3.53
LC*	1.06	1.59	3.22	2.16

We next proceed to compare the average slowdowns of LC to those of LC*:

- *Light loading.* The average slowdowns of both policies in the lightly-loaded interval [0, 1200] is very close to 1, meaning that jobs are served with almost no delay. These statistics are due to the fact that when the arrival rate is low, a job is often assigned to an idle back-end server.
- *Medium loading.* As the arrival rate and subsequent utilization increase in the interval [1200, 2400], LC* slightly outperforms LC. This is due to the fact that LC* has only few occasions to make better assignments than LC.
- *Heavy loading.* As the loadings become heavy in the interval [2400, 3600], LC* outperforms LC by a factor of approximately 2. This behavior is due to the fact that under heavy loading, LC* has many occasions to make better assignments than LC.

We conclude this section by noting that LC* outperforms Dynamic with and without preemption by a factor of 3 and 2, respectively (see Tables 1 and 2).

4. ALC*—an adaptive version of the LC* policy

The previous experiments made the simplifying assumption that the overhead incurred by the dispatcher to select (job, server) pairs are negligible. However, this assumption ignores the fact that LC* does incur some overhead beyond LC, because the dispatcher has to determine the size of a requested file before assigning the request to a back-end server. Consequently, we need to gauge the trade-off between the additional overhead incurred by LC* and its posited improved performance.

The results of the previous experiment show that LC* outperforms LC dramatically under heavy loading, while under light and medium loadings, their performance is quite similar. This behavior is explained by the fact that when the arrival rate and utilization are low, back-end server queues rarely contain more than one job. Therefore the chance of two large jobs being assigned to the same back-end server – and, consequently, the chance of unbalancing the

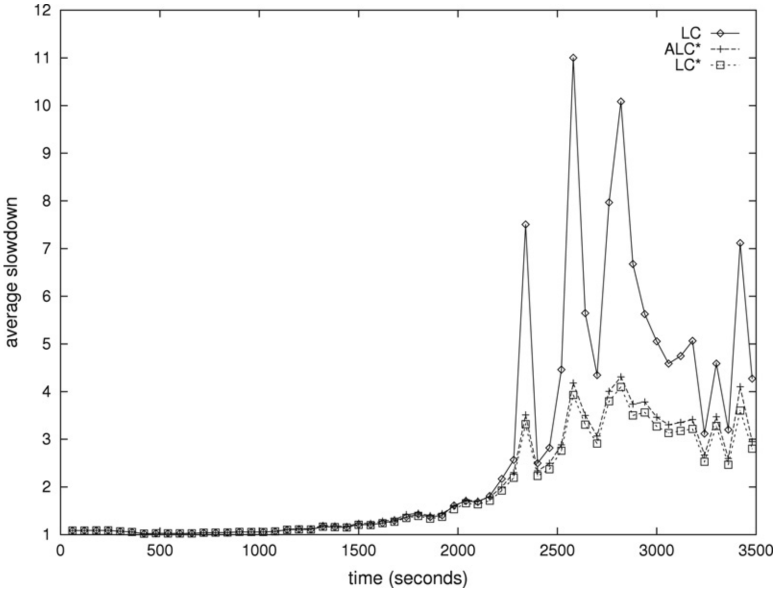


Fig. 7. Successive average slowdowns for LC, ALC* and LC* on a cluster with four back-end servers.

amount of workload across back-end servers – is very small. In contrast, under heavy loading, back-end servers often process several jobs simultaneously, and the chances that LC assigns two or more large jobs to the same back-end server increases.

This observation suggests that the dispatcher should employ an adaptive policy: it should use LC under light loadings, and switch to LC* under heavy loadings. Thus, under such an adaptive policy, the overhead attendant to LC* is incurred only when it gives rise to improved performance. There is a direct correspondence between loading and M, the minimum queue size of back-end servers: under light loading, M tends to be 0, while under medium or heavy loadings, M tends to exceed 1.

The adaptive version of LC*, which we call ALC*, is defined as follows:

1. If M is 0, then the dispatcher uses the LC policy, i.e. it assigns an incoming request to a back-end server with the smallest number of jobs.
2. Otherwise, if M exceeds 0, then the dispatcher assigns an incoming request using the LC* policy.

To gauge the performance of ALC* we exercised it on the same World Cup trace, and compared the results with those obtained for LC* and LC (see Section 3). The experiment used the same parameter value for ALC* as for LC*, namely, a file was classified as large, if it exceeded 30K. When M exceeded 0, the simulation added an overhead of 19 μs to the processing time of each request (this value was obtained experimentally).

The results of the simulation are displayed in Fig. 7. In the time interval [0, 2000], M is almost always 0, and the performances of ALC* and LC are identical as expected. In the time interval (2400, 3600], M almost always exceeds 0 and ACL* outperforms LC by almost a factor of 2 (the average slowdown of ALC* is 3.43, while the average slowdown of LC is 6.28). Note that the job classification overhead incurred by ALC* affects its performance only minimally (the average slowdown increases from 3.22 to 3.43). Thus, it pays to spend the extra overhead time of searching file sizes in order to balance the amount of workload at back-end servers.

The remainder of this section studies the effect of various factors on ALC* performance, as well as its scalability.

4.1. The effect of dispatcher overhead on ALC performance*

This section starts with a brief review of common cluster architectures and analyzes the resultant dispatcher overhead. It then proceeds to analyze the impact of dispatcher overhead on ALC* performance.

From the point of view of how requests are assigned to back-end servers one distinguishes between two types of architectures:

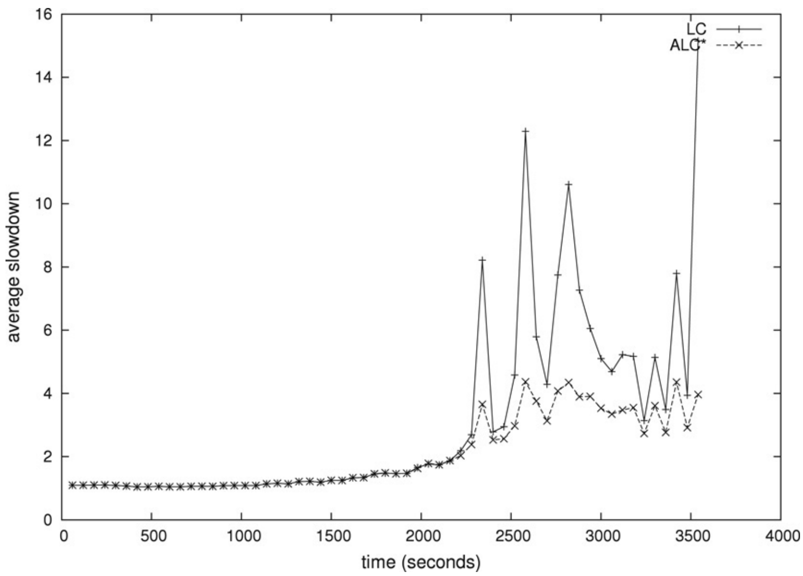


Fig. 8. Successive average slowdowns for LC and ALC* on a two-way cluster employing delayed binding.

- Architectures that perform delayed binding (content-aware binding), where the dispatcher examines client requests before assigning them to back-end servers. In particular, this type of architecture is employed by layer-7 Web switches.
- Architectures that perform immediate binding (content-blind routing), in which the dispatcher assigns jobs to back-end servers immediately, without examining their content. In particular, this type of architecture is used by layer-4 Web switches.

From the point of view of how responses are returned to clients, one distinguishes between two architectures:

- One-way architectures, where back-end servers respond directly to clients.
- Two-way architectures, where back-end servers return responses to the dispatcher, which in turn sends them back to clients.

From this brief presentation[2] we conclude that dispatcher overhead is indeed negligible when the cluster employs immediate binding. However, dispatcher overhead is not negligible in two-way clusters employing delayed binding.

To study the effect of dispatcher overhead on the performance of assignment policies, we ran experiments simulating two-way clusters employing delayed binding. In these experiments, the dispatcher overhead was estimated by the formula: $c_1 + c_2 * file\_size$, where the first term of the sum denotes the (constant) time to perform delayed binding, and the second term denotes the (variable, size-dependent) time required to transmit a client request via the dispatcher. The numerical values used by the simulation for the constants c_1 and c_2 are, respectively, 10 μs and 5 μs/kB. These values have been derived from the experimental results presented in [22]. As in the previous experiment, when ALC* behaves like LC*, it incurs an additional overhead of 19 μs, which estimates the time it takes to determine whether a requested file is large or small.

Fig. 8 depicts the average slowdowns of LC and ALC* when the dispatcher overhead is computed as described above. As expected, the slowdown performance degrades: the average slowdowns of LC and ALC* under heavy loading are 6.53 and 3.56, respectively. However, the performance is not heavily impacted by the dispatcher overhead: recall that in the previous experiment, where dispatcher overhead was ignored, the average slowdowns of LC and ALC* in the corresponding regime were 6.28 and 3.56, respectively (see Fig. 7).

It is worth mentioning that the LC policy can be supported by an immediate binding architecture, whereas ALC* can be supported only by a delayed binding architecture (which incurs higher overhead than immediate binding architectures). This experiment shows that ALC* achieves better performance even when compared with LC

[2] See [4] for more details regarding cluster architectures.

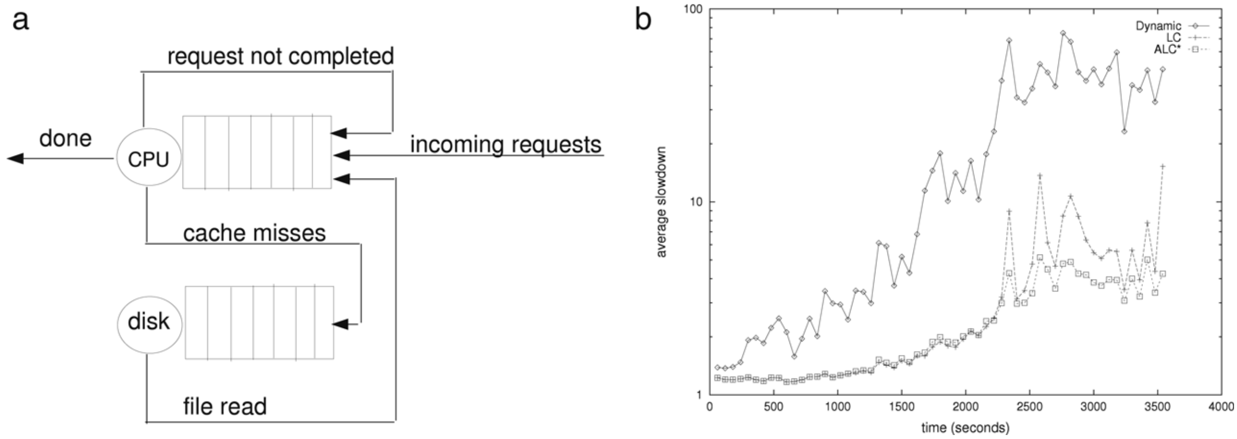


Fig. 9. (a) Back-end server processing when cache misses occur. (b) Successive average slowdowns of Dynamic, LC and ALC* on a cluster with four back-end servers, for a cache hit ratio of 99% (note the logarithmic scale on the vertical axis).

implemented on immediate binding architectures. More specifically, the average slowdown of ALC* in periods of heavy loadings is 3.56. In contrast, when the dispatcher overhead is ignored, the average slowdown incurred by LC in the corresponding regime is 6.28.

4.2. The effect of cache misses on ALC* performance

All the experiments performed so far assumed that all requests are served from cache. This assumption is motivated by the fact that cache misses occur relatively infrequently. For example, Arlitt and Jin [2] studied the World Cup trace and observed that 89% of the total requested data referred to only 10% of the files. Still, it is worthwhile to study the impact of cache misses on assignment policies. To this end, this section compares the effect of cache misses on the performance of Dynamic, LC and ALC*. (The performance of Size-Range is not depicted because it is substantially worse than the performance of all other policies.)

The simulation experiments used the LRU cache replacement policy, where files with sizes in excess of 500 kB are never cached. The model for back-end server processing used by the study assumes that the CPU and disk times for distinct requests can overlap, as shown in Fig. 9(a). (For more details regarding the underlining model, see [15].) The simulation used the following values prescribed in [15]: disk latency was set at 28 ms (two seeks plus a rotational latency), and disk transfer was set at 410 μs per 4 kB. An additional 14 ms (a seek plus rotational latency) is incurred for every 44 kB of files larger than 44 kB.

Fig. 9(b) depicts the average slowdowns of Dynamic, LC and ALC* when the cache hit ratio is 99% (i.e., 1% of the requests are served from disk). Dynamic still has the worst performance, even though it assumes known service times, and therefore, the dispatcher knows whether a request is served from cache or from disk. Note, however, that this information is not assumed known in LC and ALC*. Even in the presence of cache misses, ALC* still outperforms LC significantly under heavy loading. To wit, the average slowdown of ALC* is 4.01, as compared to an average slowdown of 6.99 for LC. Note that this performance advantage of ALC* over LC is slightly less than the case when cache misses were precluded (see Fig. 7).

4.3. The effect of dynamic requests on ALC* performance

The experiments performed so far assumed that all requests are static, i.e., requests are for already-created documents. The assumption is motivated by the fact that most Web servers receive overwhelmingly more static requests than dynamic ones [1,11,14]. A case in point is the World Cup trace, where only 0.02% of requests were dynamic. Still, it is important to study the effect of dynamic requests on the performance of assignment policies. This section shows that when the percentage of dynamic requests is small, then ACL* outperforms all other policies considered here. However, when the percentage of dynamic requests increases, the gap in performance between ALC* and LC closes.

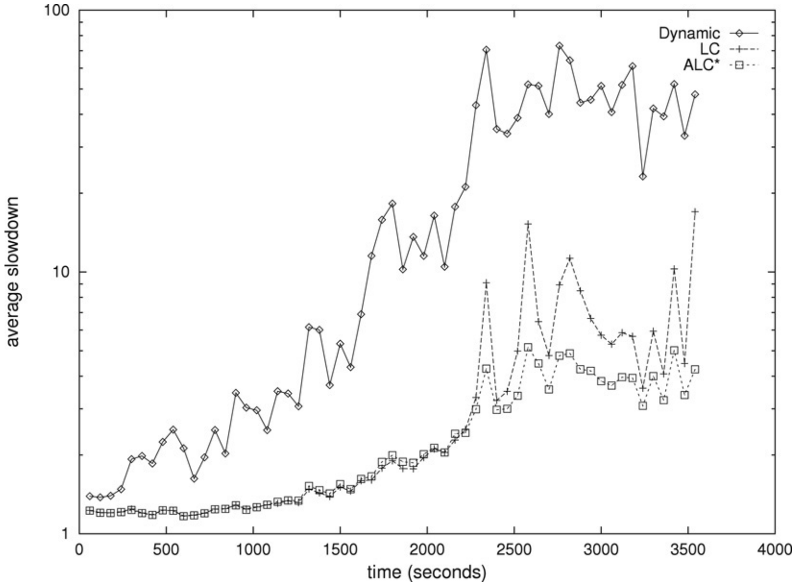


Fig. 10. Successive average slowdowns for Dynamic, LC and ALC* on a cluster with four back-end servers, when 0.02% of requests are dynamic and the cache hit ratio is 99% (note the logarithmic scale on the vertical axis).

Fig. 10 displays the average slowdowns of Dynamic, LC and ALC* when the cache hit ratio is 99% and 0.02% of requests are treated as dynamic. Here, the service time of a dynamic request is larger by a factor of 10 than the service time of a static request of the same size (served from cache). This factor value was estimated empirically in [19].

The results show that Dynamic again exhibits the worst performance, even though this policy assumes precisely known service times for all requests. And again, ALC* outperforms LC significantly under heavy loading. To wit, the average slowdown of ALC* is 4.07, while that of LC is 7.52. Note that the performance of ALC* is influenced by the presence of dynamic requests to a lesser degree than LC. Specifically, ALC* experiences a 1.5% increase in average slowdown, while LC experiences a 7% increase.

The rather negligible effect of dynamic requests on ALC* performance can be explained by analyzing the nature of disparities created by such requests. First, since the dispatcher does not know the size of dynamic requests (which are created on the fly by back-end servers), it follows that large disparities in the amount of workload assigned to back-end servers can occur when a large static job and a dynamic job are assigned to the same back-end server. However, such disparities do not affect performance considerably, because the corresponding jobs rarely compete for resources (a large file is likely not cached, so the corresponding job is disk-intensive, while a dynamic request is CPU-intensive).

Second, disparities may occur when two or more dynamic requests are assigned to the same back-end server. However, if the percentage of dynamic requests is small, then this situation occurs only rarely.

Next, we consider the case where dynamic requests are a substantial part of the workload. Accordingly, Fig. 11 displays the average slowdowns of Dynamic, LC and ALC* when 10% of requests are treated as dynamic. Note that in this experiment the cluster has eight back-end servers; this is because the substantial increase in dynamic requests leads to a substantial increase in the cluster workload, which basically brings a four back-end cluster to a standstill (average slowdowns well in excess of 5000).

The results show that the performances of LC and ALC* are essentially indistinguishable. Specifically, the average slowdown over the entire time horizon of ALC* and LC is 2.589 and 2.588, respectively. This result can be explained by the fact that dynamic requests, which now constitute the bulk of large requests, are treated similarly under ALC* and LC. By definition, the sizes of dynamic requests are not known a priori, and, therefore, ALC* treats them as small jobs, and thus cannot make better assignments than LC. In effect, ALC* makes better assignments just for large static requests, which is barely enough to compensate for the overhead incurred. In conclusion, this last experiment supports the contention that ALC* should be used only for clusters that receive overwhelmingly more static requests than dynamic ones.

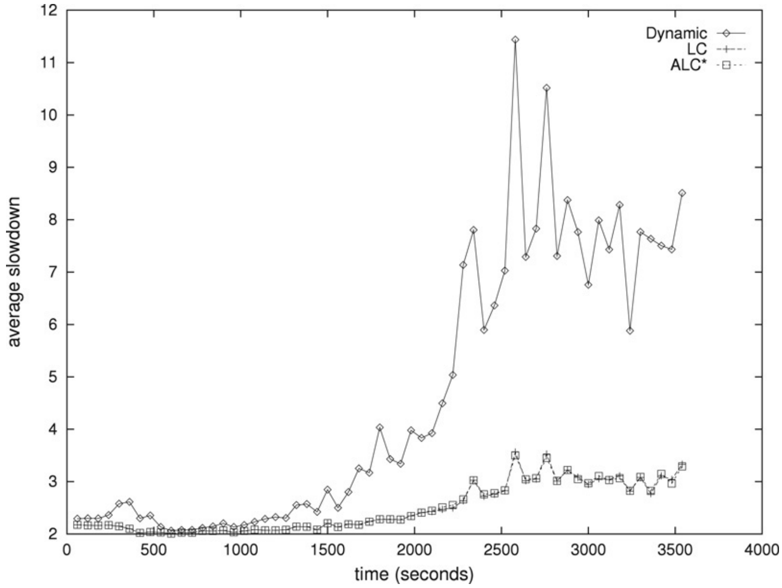


Fig. 11. Successive average slowdowns for Dynamic, LC and ALC* on a cluster of eight back-end servers, when 10% of requests are dynamic and the cache hit ratio is 99%.

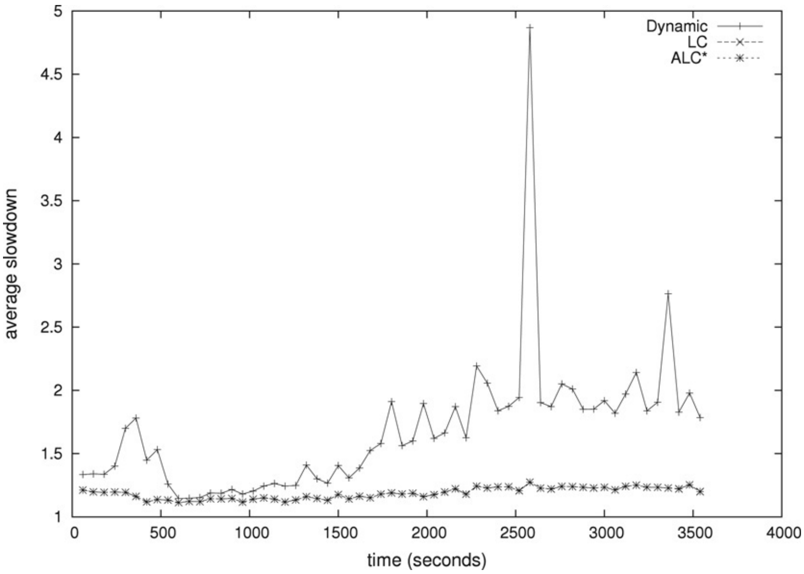
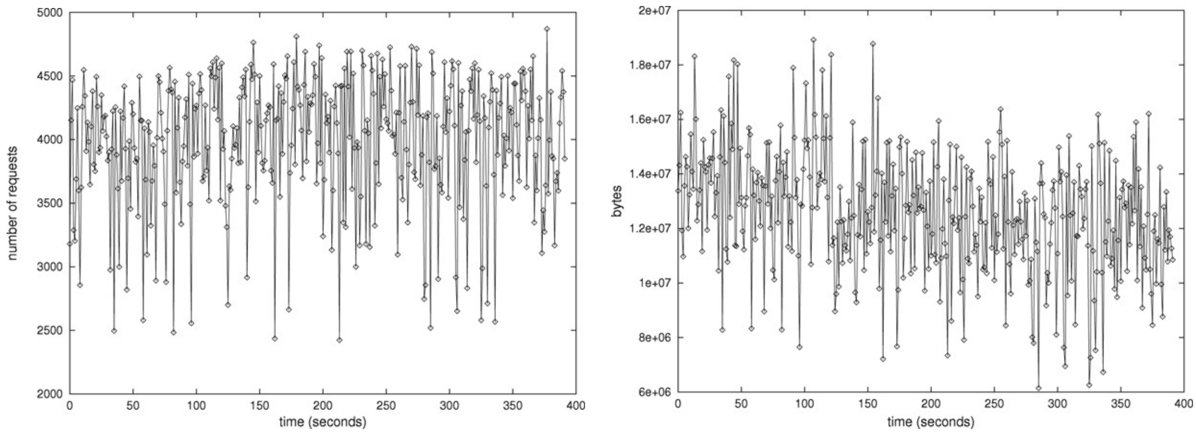


Fig. 12. Successive average slowdowns for LC, ALC* and LC* on a cluster of six back-end servers, when 10% of requests are dynamic and the cache hit ratio is 99%.

*4.4. Scalability of ALC**

The study presented in this section compares the performance of Dynamic, LC and ALC* as the cluster size grows larger. The first experiment, depicted in Fig. 12, considered a cluster with six back-end servers and used the same World Cup trace. As can be seen, the performances of LC and ALC* are practically indistinguishable: the average slowdown over the entire time horizon is 1.19 for both of them. The explanation is that the loading on individual back-end servers decreases as their number increases. In effect, under LC and ALC*, there is often an idle back-end server, so that M is 0 most of the time. Consequently, ALC* largely behaves like LC.



(a) Number of request arrivals per second.

(b) Total bytes requested per second.

Fig. 13. Empirical time series of the synthetic trace.

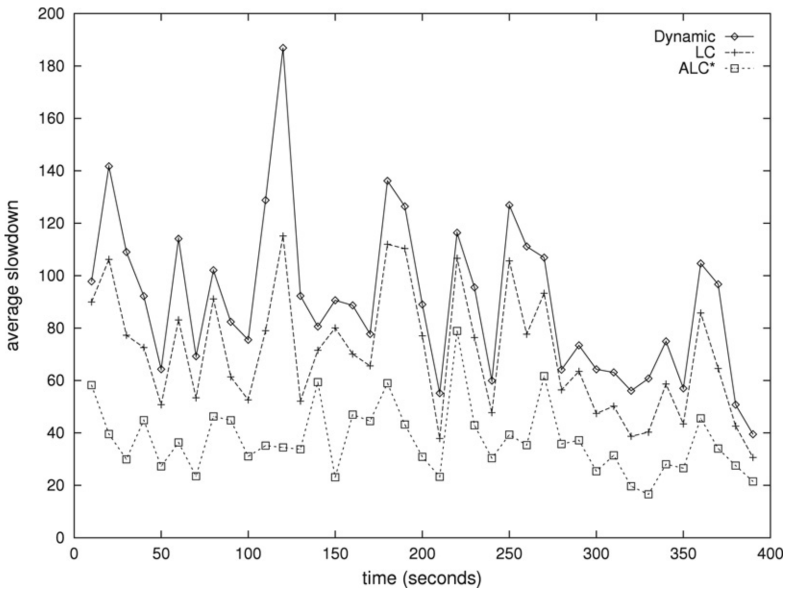


Fig. 14. Successive average slowdowns of Dynamic, LC and ALC* in a cluster of eight back-end servers (note the logarithmic scale on the vertical axis).

Since studying the scalability of ALC* using the World Cup trace proved inconclusive because the loading was too low, we generated a power-law synthetic trace which could stress a bigger cluster. The synthetic trace was generated using a traffic generator, called Geist, designed at Intel [12,13]. The relevant statistics of this trace are described next.

Simulation data. The aforementioned trace consists of over 4.6 million requests received over a time interval of approximately 400 seconds. This trace offered the cluster both medium and high loadings, with utilization exceeding 50%. Fig. 13(a) depicts the number of requests received by the cluster in successive one-second intervals, while Fig. 13(b) plots the number of bytes due to file requests in successive one-second intervals.

Simulation experiments. Fig. 14 displays the average slowdowns of Dynamic, LC and ALC* in successive 10-second intervals for a cluster of eight back-end servers. Observe that ALC* outperforms both Dynamic and LC in all time intervals. Moreover, the performance advantage of ALC* over Dynamic and LC in a cluster of eight back-end servers is comparable to that in a cluster of four back-end servers. This suggests that ALC* scales well.

5. Conclusion

Several conclusions can be drawn from the studies presented in this paper. First, the architecture of back-end servers affects profoundly the slowdown performance of assignment policies. Specifically, the paper shows experimentally that the slowdown performance of `Dynamic` and `Size-Range` is far better when preemption is precluded than when it is admitted.

The second conclusion is that if jobs can be preempted, then balancing the number of jobs at back-end servers is a better strategy than balancing the amount of workload there. In particular, the paper shows experimentally that `LC` (which uses the former strategy) performs far better than `Dynamic` and `Size-Range` (which use the latter strategy).

Finally, the study supports the contention that combining the two balancing strategies yields slowdown performance superior to that of each strategy alone. Specifically, the paper proposes two new policies, `LC*` and `ALC*`, which improve over `LC`, `Dynamic` and `Size-Range`, especially under heavy loading regimes. A notable feature of the proposed policies is that they have only modest informational and computational requirements, and this renders them practical candidates for real-life implementation.

References

[1] M. Arlitt, R. Friedrich, T. Jin, Workload characterization of a web proxy in a cable modem environment, ACM Performance Evaluation Review 27 (2) (1999) 25–36.
[2] M. Arlitt, T. Jin, Workload characterization of the 1998 world cup web site, IEEE Network 14 (3) (2000) 30–37. Extended version: Tech Report HPL-1999-35R1, Hewlett-Packard Laboratories, September 1999.
[3] P. Bruckner, Scheduling Algorithms, third ed., Springer-Verlag, 2001.
[4] V. Cardellini, E. Casalicchio, M. Colajanni, P.S. Yu, The state of the art in locally distributed Web-server systems, ACM Computing Surveys 34 (2) (2002) 263–311.
[5] G. Ciardo, A. Riska, E. Smirni, EquiLoad: A load balancing policy for clustered web servers, Performance Evaluation 46 (2–3) (2001) 101–124.
[6] G. Coulouris, J. Dollimore, T. Kindberg, Distributed Systems — Concepts and Design, third ed., Addison-Wesley, 2001.
[7] M.E. Crovella, M.S. Taqqu, A. Bestavros, Heavy-tailed probability distributions in the world wide web, in: A Practical Guide To Heavy Tails, Chapman Hall, New York, 1998, pp. 3–26.
[8] D.G. Down, Rong Wu, Multi-layered round robin routing for parallel servers, Queueing Systems: Theory and Applications 53 (4) (2006) 177–188.
[9] M. Faloutsos, P. Faloutsos, C. Faloutsos, On power-law relationships of the internet topology, in: Proceedings of ACM SIGCOMM ’99, 1999 pp. 251–262.
[10] M. Harchol-Balter, M.E. Crovella, C.D. Murta, On choosing a task assignment policy for a distributed server system, in: Proceedings of Performance Tools ’98, in: Lecture Notes in Computer Science, vol. 1468, 1998, pp. 231–242.
[11] M. Harchol-Balter, B. Schroeder, N. Bansal, M. Agrawal, Size-based scheduling to improve web performance, ACM Transactions on Computer Systems 21 (2) (2003).
[12] K. Kant, V. Tewari, R. Iyer, Geist: A generator of e-commerce and internet server traffic, in: Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, November 2001, pp. 49–56.
[13] K. Kant, V. Tewari, R. Iye, Geist: A web traffic generation tool source, in: Proceedings of the 12th International Conference on Computer Performance Evaluation, Modeling Techniques and Tools, in: Lecture Notes in Computer Science, vol. 2324, 2002, pp. 227–232.
[14] B. Krishnamurthy, J. Rexford, Web Protocols and Practice : HTTP 1.1, Networking Protocols, Caching, and Traffic Measurement, Addison-Wesley, 2001.
[15] V.S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, E. Nahum, Locality-aware request distribution in cluster-based network servers, in: Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-VIII, 1998.
[16] M. Pinedo, Scheduling: Theory, Algorithms, and Systems, Prentice Hall, 2002.
[17] A. Riska, W. Sun, E. Smirni, G. Ciardo, AdaptLoad: Effective balancing in clustered web servers under transient load conditions, in: 22nd International Conference on Distributed Computing Systems, ICDCS’02, 2002.
[18] S.M. Ross, Probability Models for Computer Science, Academic Press, 2002.
[19] Y.M. Teo, R. Ayani, Comparison of load balancing strategies on cluster-based web servers, Simulation 77 (5–6) (2001) 185–195.
[20] W. Winston, Optimality of the shortest line discipline, Journal of Applied Probability 14 (1977) 181–189.
[21] V. Ungureanu, B. Melamed, M. Katehakis, P.G. Bradford, Deferred assignment scheduling in cluster-based servers, Cluster Computing 9 (1) (2006).
[22] L. Zhao, Y. Luo, L. Bhuyan, R. Iyer, A network processor-based, content-aware switch, IEEE Micro 26 (3) (2006) 72–84.

Victoria Ungureanu is a senior analyst at Standard & Poor's. She earned a Ph.D. in Computer Science from Rutgers University. This paper has been written while she was a visiting researcher in the DIMACS at Rutgers University.

Benjamin Melamed is a Professor II at the Rutgers Business School - Newark and New Brunswick, Department of management Science and Information Systems. Melamed earned a B.Sc. in Mathematics and Statistics from Tel Aviv University in 1972, and M.S. and Ph.D. in Computer Science from the University of Michigan in 1973 and 1976, respectively. Melamed was awarded an AT&T Fellow in 1988 and an IEEE Fellow in 1994.

Michael N. Katehakis is a Professor at the Rutgers Business School - Newark and New Brunswick, Department of management Science and Information Systems. He earned a Diploma in Mathematics from the University of Athens in 1974, an M.A. in Statistics from the University of South Florida in 1978, and a Ph.D. in Operations Research from Columbia University in 1980. Katehakis is a recipient of the 1992 Wolfowitz Prize (with Govindarajulu Z.).