ADUS: Adaptive Resource Allocation in Cluster Systems under Heavy-Tailed and Bursty Workloads

Zhen Li, Jianzhe Tai, Jiahui Chen, Ningfang Mi
Department of Electrical and Computer Engineering
Northeastern University, Boston, MA 02115
{zli, jtai, jchen, ningfang}@ece.neu.edu

***Abstract*—A large-scaled cluster system has been employed in various areas by offering pools of fundamental resources. How to effectively allocate the shared resources in a cluster system is a critical but challenging issue, which has been extensively studied in the past few years. Despite the fact that classic load balancing policies, such as Random, Join Shortest Queue and size-based polices, are widely implemented in actual systems due to their simplicity and efficiency, the performance benefits of these policies diminish when workloads are highly variable and heavily dependent. In this paper, we propose a new load balancing policy named ADUS, which attempts to partition jobs according to their sizes and to further rank the servers based on their loads. By dispatching jobs of similar size to the servers with the same ranking, ADUS can adaptively balance user traffic and system load in the system and thus achieve significant performance benefits. Extensive simulations show the effectiveness and the robustness of ADUS under many different environments.**

I. INTRODUCTION

Present day, large-scale cluster computing environments are being employed in an increasing number of application areas. Examples of these systems include High Performance Computing (HPC), enterprise information systems, data centers, cloud computing and cluster servers, which provide the pools of fundamental resources. In particular, a cloud platform, as a new and hot infrastructure, provides a shared "cloud" of resources as an unified hosting pool, which requires a central mechanism for resource provisioning and resource allocation based on multiple remote clients' demands [1], [2]. Figure 1 shows an example of a classic cluster system, which consists of a hierarchy of server nodes and a front-end dispatcher which is responsible for distributing the incoming jobs among these server nodes. In our paper, we focus on the load balancing design for the dispatcher, which is featured as a redirect buffer without a central waiting queue while each server node has its own queue for waiting jobs and serves these jobs under the first-come first-serve (FCFS) queuing discipline.

A lot of previous studies have been focusing on developing load balancing policies for a large-scale cluster computing system over the past decades [3], [4], [5], [6], [7]. Examples of these policies include Join Shortest Queue (JSQ) and the size-based ADAPTLOAD. When there is no a priori knowledge of job sizes and the job sizes are exponentially distributed, JSQ has been proven to be optimal [8]. However, prior research has shown that the job service time distribution is critical for the performance of load balancing policies. [9] evaluated how JSQ performs under various workloads by measuring the mean response times and demonstrated that the performance of JSQ clearly varies with the characteristics of different service time distributions. For example, the optimality of JSQ quickly disappears when job service times are highly variable and heavy-tailed [10], [11].

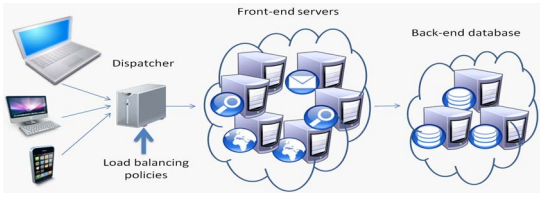

Fig. 1. Example of a classic cluster system.

Recently, size-based policies were proposed to balance the load in the system, only using the knowledge of the incoming job sizes. The literature in [6], [11], [12] have shown that such size-based policies are optimal if one aims to achieve the minimum job response times and job slowdowns. The ADAPTLOAD policy being a representative example of size-based policies, has been developed to improve average job response time and average job slowdown by on-the-fly building the histogram of job sizes and distributing jobs of similar sizes to the same server [12]. Nonetheless, [4] demonstrated that such size-based solutions are not adequate if the job service times are temporally dependent.

Based on different assumptions, other classic load balancing policies are developed under cluster and cloud computing environments. The Min-Min and the Max-Min algorithms that focus on the problem of scheduling a bag of tasks, assume that the execution times of all jobs are known [13]. Meta-schedulers, like Condor-G [14], rely on accurate queuing time prediction in scheduling jobs on computing grids. However, accurate predictions become more challenging in the current virtualized and multi-tiered environments. Recently, techniques of advance-reservation and job preemption were presented to allocate jobs in cluster or grid systems [15]. Yet, extra system overheads and job queue disruptions could decrease the overall system performance.

In this paper, we propose a new load balancing policy ADUS, which adaptively distributes work among all servers by taking account of both *user traffic* and *system load*. In detail, ADUS ranks all servers based on their present system

978-1-4577-2053-6/12/$31.00 ©2012 IEEE

loads and updates job size boundaries on-the-fly, allowing the dispatcher to direct jobs of similar sizes to the servers which have the same ranking. We expect that our new policy can improve the overall system performance by inheriting the effectiveness of both JSQ and ADAPTLOAD and meanwhile overcoming the limitations of these two policies.

Trace-driven simulations are used to evaluate the performance of ADUS. Extensive experimental results indicate that ADUS significantly improves the system performance - job response times and job slowdowns, under heavy-tailed and temporal dependent workloads. We also use the case study of a real TCP trace to further validate the benefits of ADUS in actual systems, where ADUS again achieves a clear improvement of up to one order of magnitude. The reminder of this paper presents our results in detail.

II. MOTIVATION

In this section, we first give an overview of three existing load balancing policies which are widely used in cluster systems. By analyzing these policies under highly variable and/or heavily dependent workloads, we then illustrate their limitations in terms of performance matrices, which motivate the design of a new efficient policy proposed in this paper.

Here, We consider the following three load balancing policies.

- *Random*: it randomly determines the server that will be allocated for each incoming job.
- *Join Shortest Queue*(JSQ)[16]: the newly arrived job is assigned to the server with the least number of jobs in the queue.
- A size-based policy, e.g., ADAPTLOAD [12]: it builds the histogram of job sizes and partitions the work into equal areas. Each server will then dedicate to processing jobs with similar sizes. The boundaries of each size interval are adjusted dynamically according to the past history. ADAPTLOAD has been proven to achieve high performance by reducing the number of small jobs from waiting behind large ones.

Despite the fact that such classic load balancing policies are widely used because of their simplicity (e.g., Random) and their efficiency (e.g., JSQ and ADAPTLOAD), we found that these policies exhibit performance limitations under the workloads with high variability and strong temporal dependence. Here, we exemplify such performance limitations by evaluating the JSQ and ADAPTLOAD policies.

Limitations of JSQ*:* Since JSQ always sends a job to the shortest (or least loaded) queue, there is a high probability such that small jobs are stuck behind large ones, which may consequently cause performance degradation when the distribution of job sizes is highly variable. To verify this limitation, we measure the fraction of small jobs that are stuck in the queue due to large ones, i.e., the ratio of the number of small jobs which wait behind large jobs to the total number of small jobs. Here, we use $T = \mu^{-1}(1 + 2CV)$ to classify small and large jobs, where μ^{-1} is the mean of job sizes and CV is the coefficient of variation of job sizes.

In this experiment, the job interarrival times are exponentially distributed with mean rate $\lambda = 2$ while the job sizes are drawn from a 2-state Markov-Modulated Poisson Process (MMPP) with mean rate $\mu = 1$ and squared coefficient of variation $SCV = 20$. We also consider 4 homogeneous server nodes in the system. By scaling the mean processing speed of each server node (i.e., $\mu_p = 1$), we obtain 50% utilization per server node. The sample space in this experiment is 1 million.

Table I shows the measured results under the three load balancing policies. We first observe that among the three policies, Random as expected obtains the largest fraction of small jobs which have to wait behind large ones in the queue. Observe also that although JSQ is well known for its optimal performance, there are about 23% of small jobs being stuck in the queue due to large ones, which thus degrades the performance of small jobs and further deteriorates the overall system performance. By dispatching jobs based on their sizes, the size-based policy ADAPTLOAD successfully avoids the majority of small jobs (i.e., about 95%) waiting behind large ones in the queue.

TABLE I
FRACTION OF SMALL JOBS THAT ARE STUCK BEHIND LARGE ONES WHEN THE JOB SIZES HAVE HIGH VARIABILITY.

Policy	Random	JSQ	ADAPTLOAD
Ratio	42.53%	22.93%	4.78%

Limitations of ADAPTLOAD*:* We investigate the performance of ADAPTLOAD under the workload with high variance and strong temporal dependence in job sizes by measuring the queuing times of small and large jobs separately. Table II shows the measured results, where we use the same experimental setting as shown in the above but introduce the temporal dependence into job sizes. We observe that the effectiveness of ADAPTLOAD dramatically deteriorates such that its performance becomes comparable to the worst one (i.e., Random). This essentially motivates that simply directing jobs with similar size to the same server is not sufficient under temporally dependent workloads. We interpret that because of the temporal dependence in job sizes, it becomes highly possible that the jobs with similar sizes are clustered together and arrive the system in bursts. If the policy simply assigns such jobs to the same server, then that particular server will experience a sudden load spike during a short period, causing long waiting times for those jobs and thus degrading the overall system performance.

TABLE II
OVERALL AND INDIVIDUAL QUEUING TIMES FOR SMALL AND LARGE JOBS WHEN THE JOB SIZES ARE HIGHLY VARIABLE AND STRONGLY DEPENDENT.

Policy	Overall	Small	Large
Random	22.17	21.72	34.46
JSQ	10.75	10.45	25.62
ADAPTLOAD	14.66	13.98	34.59

In summary, both JSQ and ADAPTLOAD have the limita-

tions under highly variable and/or strongly dependent workloads, which thus cause negative impacts on system performance. Motivated by this problem, we present a new load balancing policy which attempts to address the limitations of both JSQ and ADAPTLOAD and thus achieve visible performance improvement.

III. NEW LOAD BALANCING POLICY: ADUS

Now, we turn to present our new load balancing policy ADUS which adaptively distributes work among all servers by taking account of both *user traffic* and *system load*, aiming to inherit the effectiveness of JSQ and ADAPTLOAD and meanwhile overcome the limitations of these two policies as shown in the previous section. The main idea of ADUS is to on-the-fly rank all the servers according to their system loads and dynamically tune the job size boundaries based on the current user traffic loads. ADUS then directs the jobs whose sizes locate in the same size boundary to the corresponding servers which are in the same ranking. Based on the observation that the majority of jobs in a heavy-tailed workload is small, ADUS always gives small jobs high priority by sending them to the highly ranked (i.e., less loaded) servers.

Recall that ADAPTLOAD [12] evenly balances the load across the entire cluster by determining boundaries of jobs sizes for each server. ADUS adopts such boundaries where the histogram of job sizes is built and partitioned into N equal areas for N servers in the system. Then, the i^{th} server is responsible for the work locating in the i^{th} area, which allows the decreasing variation in job sizes (or job service times) on each server. Consequently, the proportion of small jobs that wait behind long ones is reduced as well and the user traffic is thus well balanced among all servers.

However, as we discussed in Section II, simply separating requests according to their sizes is not sufficient under temporally dependent workloads. Therefore, ADUS periodically ranks all servers based on their present system loads and keeps sending the incoming jobs of similar sizes to a server with the same ranking instead of the same server which might be overloaded by the previous arrived jobs. Given N servers $\{S_1, ..., S_N\}$ and N boundaries of jobs sizes $[0, b_1), [b_1, b_2), ..., [b_{n-1}, \infty)$. For every window of C jobs, ADUS sorts all the servers in a non-decreasing order of their loads (e.g., queue length) and then gets a priority list $S^{'}$. Then, the i^{th} server $S_i^{'}$ in the priority list will be assigned to the incoming jobs in the next window which have the sizes within the i^{th} boundary $[b_{i-1}, b_i)$. It follows that the first server $S_1^{'}$ with the least load in the priority list will then serve small yet a large number of jobs while the last server $S_N^{'}$ that became heavy loaded due to serving small ones during the last window then starts to serve large but few jobs. As a result, ADUS further successfully balances the system load among all N servers and thus significantly diminishes the proportion of similar sized jobs (especially small ones) being queued on the same server during a short period.

Figure 2 gives the high level description of our new policy ADUS. We remark that the window size C indicates how often

Algorithm: ADUS
begin
1. initialization
  a. priority list: $S^{'} = \{S_1^{'}, ..., S_N^{'}\}$;
  b. size boundaries: $[0, b_1), [b_1, b_2), ..., [b_{n-1}, \infty)$;
2. upon the completion of every C jobs
  a. sort all N servers in a non-decreasing order of system loads and update the priority list $S^{'}$;
  b. update the size boundaries such that the work is equally divided into N areas;
3. **for** each arriving job
  a. **if** its job size $\in [b_{i-1}, b_i)$
    then direct this job to server $S_i^{'}$;
end

Fig. 2. The high level description of ADUS.

ADUS re-ranks all servers. A large window size updates the rankings of servers in a lower frequency, which introduces less computational overhead but might provide slower reactions to frequently changed workloads. In contrast, a small window size can quickly adapt to workload changes, improving the system performance, but needs a higher computational cost in the meanwhile. Thus, we conclude that selecting an appropriate window size is critical to ADUS's performance as well as its computational overheads. In our experiments, we set C to different values under various workloads and find that the best performance is obtained when window size C is equal to 100. We remark that the selection of C depends on the workload changes. If the workload changes frequently, then a smaller window will achieve a better performance. How to dynamically adjust C will be studied in our future work.

IV. PERFORMANCE EVALUATION OF ADUS

In this section, we use trace-driven simulations to evaluate the performance improvement of ADUS in a homogeneous clustered system with N (FCFS) servers. Various load balancing algorithms are then executed by a load dispatcher which is responsible for distributing the arrivals to one of N servers. For all simulations, the specifications of a job include job arrival times and job sizes, which are generated based on the specified distributions or real traces. Throughout all experiments, the N servers have the same processing rates $\mu_p = 1$.

We first produce an experiment where the job inter-arrival times are exponentially distributed with mean $\lambda^{-1} = 0.5$ while the job sizes (i.e., the service process) are drawn from a MMPP(2) with the mean equal to $\mu^{-1} = 1$, squared coefficient of variation equal to $SCV = 20$, and autocorrelation function (ACF) at lag 1 equal to 0.40. Therefore, high variability and temporal dependence are injected into the workload, i.e., the service process. Also, we consider $N = 4$ homogeneous server nodes in the cluster such that the average utilization levels at each server node are $\rho = 50\%$. The sample space in all simulations is 1 million jobs. Simulations stop only after all the jobs have been used.

Table III shows the system performance under four policies including Random, JSQ, ADAPTLOAD and ADUS. Here, we measure the mean job response times "Resp$_{avg}$" (i.e., the summation of waiting times in the queue and service times)

TABLE III
OVERALL RESPONSE TIMES (RESP) AND SLOWDOWNS (SLW) OF FOUR POLICIES WHEN $\lambda = 2, \mu = 1, \mu_p = 1, SCV = 20, ACF = 0.40$, AND $\rho = 50\%$.

Policy	Resp_{avg}	SLW_{avg} $(*10^3)$
Random	23.17	175.54
JSQ	11.75 (49%)	82.31 (53%)
ADAPTLOAD	15.66 (32%)	110.68 (37%)
ADUS	9.63 (58%)	59.42 (66%)

and the mean job slowdown "SLW_{avg}"(i.e., job response times normalized by job service times). The relative improvement with respect to Random is also given in parenthesis, see Table III.

We observe that compared to Random, both ADAPTLOAD and JSQ improve the overall system performance, while ADUS outperforms among all the policies. We also measure the performance metrics shown in Tables I and II under ADUS and find that only 3.99% of small jobs being stuck behind large ones and the queueing times of small and large jobs are 7.74 and 32.62, respectively. This further shows that ADUS achieves the best performance for small jobs. We interpret the significant performance improvement as an outcome of *always* assigning small jobs to the least loaded servers (i.e., the 1^{st} rank). Furthermore, the performance of large jobs is improved as well, although not as significant as small ones.

We further investigate the tail distribution of response time and slowdown under the four policies. Figures 3 and 4 plot the complementary cumulative distribution function (CCDF) of overall and individual response times and slowdowns, respectively. As there are 4 server nodes, we here classify job sizes into 4 categories: small, medium small, medium large and large. Each of them represents 25% of the whole workload. We observe that under ADUS, the majority (about 99.4%) of jobs experience faster response times and almost all jobs have smallest slowdowns. Figure 4 further shows that ADUS benefits small jobs by avoiding them waiting after large ones, see plots (a) and (b). On the other hand, due to its unfairness to large ones by sending them to high loaded servers, ADUS degrades the performance of medium and large jobs, see plot (c) and (d). Fortunately, the proportion of large jobs is quite small. Such performance penalty does not significantly affect the overall system performance.

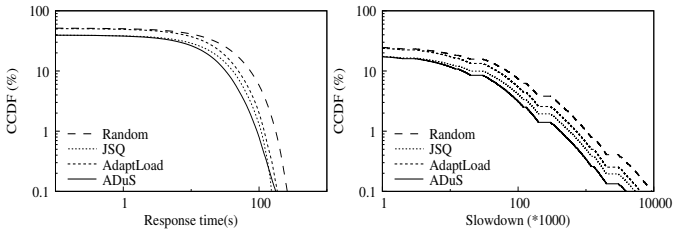

Fig. 3. CCDFs of overall response times and slowdowns.

A. Impacts of Variability and Temporal Dependence

As coefficient of variation and temporal dependence are two important characteristics in the job size distribution, we further analyze their impacts on policy performance. Figure 5 and Figure 6 show the corresponding experimental results under the workloads with different coefficient of variations and different temporal dependence profiles, respectively.

We first conduct experiments with various SCVs (e.g.,10, 20, and 40) of the job sizes but keep the same mean. As shown in Figure 5, both ADUS and JSQ obtain 30% and 40% performance improvement over Random when the job sizes are not highly variable. However, as the variability in job sizes increases (e.g., $SCV = 40$), the workload contains few but extremely large jobs, which unfortunately diminishes the effectiveness of JSQ. It becomes highly likely that under JSQ small jobs may have to wait behind several extremely large ones, resulting in long response time as well as long slowdown. While ADUS can avoid this situation by assigning jobs with similar size to the same ranked server and thus achieves better performance improvement. Additionally, the tail distributions of response times in these experiments are qualitatively the same as we show in Figure 3, which are then omitted here in order to save the space.

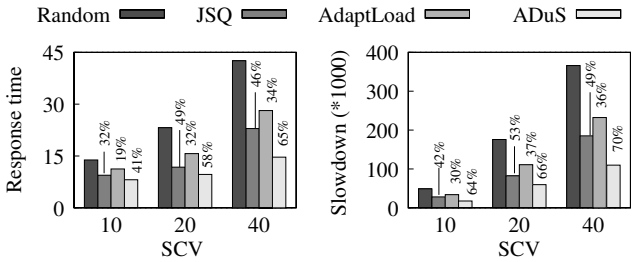

Fig. 5. Response times and slowdowns under different SCVs when $\lambda = 2, \mu = 1, \mu_p = 1, \rho = 50\%$,and $ACF = 0.40$.

In order to investigate the sensitivity of ADUS to temporal dependence, we conduct experiments with three different temporal dependence profiles, but the same mean and the same SCV of the job sizes. Figure 6 shows the experimental results as a function of temporal dependence profiles (e.g., weak, medium and strong). When the temporal dependence is weak, all the three policies JSQ, ADAPTLOAD and ADUS perform better than Random. However, the strong temporal dependence in job sizes dramatically degrades the overall system performance and deteriorates the effectiveness of JSQ and ADAPTLOAD, under which the system experiences similar performance as under Random. In contrast, ADUS still achieves a clear performance improvement, which indicates that ADUS is more robust to temporally dependent workloads than the other policies.

B. Case Study: Real TCP Trace

In this section, we validate the effectiveness and the robustness of ADUS using actual measured data. The real trace, named LBL-TCP-3, were collected by Lawrence Berkeley Laboratory over a two-hour period in January 1994, which consists of 1.8 million wide-area TCP packets. Each packet has a timestamp and the number of data bytes. We here use the number of data bytes as the input for job sizes, which

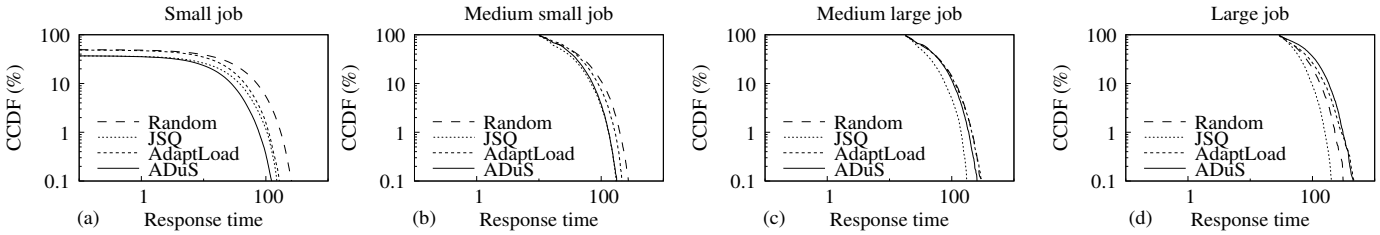

Fig. 4. CCDFs of response times for small, medium small, medium large, and large jobs.

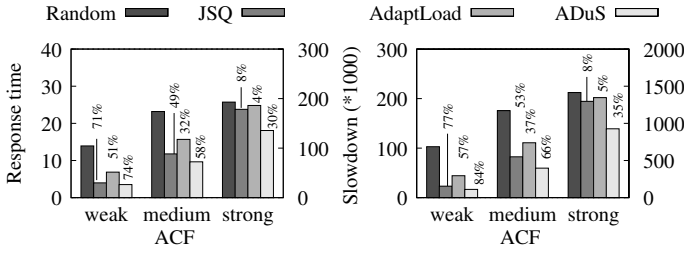

Fig. 6. Response times and slowdowns under different ACFs with $\lambda = 2, \mu = 1, \mu_p = 1, SCV = 20$ and $\rho = 50\%$. Here, the strong ACF case uses the right side y-axis labels in both two plots.

are highly variable with $SCV = 2$ and temporally dependent with ACF at lag 1 equal to 0.44. We also scale the packet interarrival times in order to emulate a heavy loaded system with 75% utilization. Our experimental setup assumes a cluster of remote servers in the communication network.

Table IV summarizes the performance measurement (e.g., TCP packet round trip times and TCP packet slowdown) under the four load balancing policies. Here, the server processing rate μ_p is equal to 1, mean interarrival time is scaled to $\lambda^{-1} = 67$, and the mean package size is $\mu^{-1} = 200.87$. Consistent to the previous experiments, ADuS achieves a clear improvement. Furthermore, the slowdown under ADuS is significantly improved by one order of magnitude compared to the Random policy.

TABLE IV
REAL TRACE PERFORMANCE OF LBL-TCP-3.

Policy	Random	JSQ	AdaptLoad	ADuS
Resp	0.2964	0.252 (15%)	0.228 (23%)	0.193 (35%)
SLW	269.53	225.50 (16%)	39.16 (85%)	32.93 (88%)

In summary, the extensive experimentation carried out in this section shows that ADuS effectively improves the overall system performance and thus becomes the best choice for load balancing in a cluster system. Sensitivity analysis to job size variation and job size temporal dependence further demonstrates the visible gains of ADuS. Finally, the case study of a real TCP trace further validates the effectiveness and the robustness of ADuS.

V. CONCLUSION

In this paper, we evaluate the performance of the classic load balancing policies including JSQ and size-based approach for a homogeneous cluster system under highly variable and strongly temporal dependent workloads. We demonstrated that these policies are now ineffective when workloads become heavy-tailed and bursty. We thus proposed a new load balancing policy ADuS, which distributes the work in the system by taking account of both user traffic and system load. Using trace-driven simulations, we showed that ADuS inherits the effectiveness of JSQ and size-based policies and meanwhile overcomes their limitations, which results in significant performance benefits. We also showed that ADuS can quickly adapt to the workload changes by monitoring user traffic and system loads, repeatedly ranking the servers and partitioning the work in an on-line fashion.

REFERENCES

[1] P. Chaganti, "Cloud computing with amazon web services," *IBM Technical Library*, pp. 1–10, 2008.

[2] J. Varia, "Building grepptheweb in the cloud," in *Amazon Elastic Compute Cloud Articles and Tutorials*, 2008.

[3] Y. M. Teo and R. Ayani, "Comparison of load balancing strategies on cluster-based web servers," *Trans. Soc. for Modeling and Simulation*, vol. 77, no. 5-6, pp. 185–195, Nov. 2001.

[4] Q. Zhang, N. Mi, A. Riska, and E. Smirni, "Performance-guided load (un)balancing under autocorrelated flows," *IEEE Transs on Parallel and Distributed Systems*, vol. 19, no. 2, pp. 652–665, 2008.

[5] N. Mi, Q. Zhang, A. Riska, and E. Smirni, "Load balancing for performance differentiation in dual-priority clustered servers," *the 3rd International Conference on the Quantitative Evaluation of Systems (QEST'06)*, pp. 385–395, 2006.

[6] H. Feng, M. Visra, and D. Rubenstein, "Optimal state-free, size-aware dispatching for heterogeneous m/g/-type systems," *Performance Evaluation J.*, vol. 62, no. 1-4, pp. 475–492, Nov. 2005.

[7] M. Harchol-Balter and A. Downey, "Exploiting process lifetime distributions for dynamic load balancing," *ACM Trans. Computer Systems*, vol. 15, no. 3, pp. 253–285, Aug. 1997.

[8] W. Winston, "Optimality of the shortest line discipline," *Journal of applied probability*, vol. 14, pp. 181–189, 1977.

[9] R. Nelson and T. Philips, "An approximation for the mean response time for shortest queue routing with general interarrival and service times," *Performance Evaluation*, vol. 17, pp. 123–139, 1998.

[10] W. Whitt, "Deciding which queue to join: Some counterexamples," *Operations Research*, vol. 34, no. 1, pp. 226–244, Jan. 1986.

[11] M. Harchol-Balter, M. Crovella, and C. Murta, "On choosing a task assignment policy for a distributed server system," *J. Parallel and Distributed Computing*, vol. 59, no. 2, pp. 204–228, Nov. 1999.

[12] Q. Zhang, A. Riska, W. Sun, E. Smirni, and G. Ciardo, "Workload-aware load balancing for clustered web servers," *IEEE Trans. Parallel and Distributed Systems*, vol. 16, no. 3, pp. 219–233, Mar. 2005.

[13] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," vol. 24(2), 1977.

[14] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-G: A computation management agent for multi-institutional grids," *Cluster Computing*, vol. 5, pp. 237–246, 2002.

[15] W. Smith, I. Foster, and V. Taylor, "Scheduling with advanced reservations," *Proc. of the IPDPS Conference*, pp. 127 – 132, 2000.

[16] L.Kleinrock, *Queueing Systems Volume 1: Theroy*, Wiley, 1975.

[17] M. F. Neuts, *Structured Stochastic Matrices of M/G/1 Type and Their Applications*. New York: Marcel Dekker, 1989.