Benefits and Drawbacks of Redundant Batch Requests

Henri Casanova
Dept. of Information and Computer Sciences, University of Hawai'i at Manoa

August 1, 2006

Abstract. Most parallel computing platforms are controlled by batch schedulers that place requests for computation in a queue until access to compute nodes is granted. Queue waiting times are notoriously hard to predict, making it difficult for users not only to estimate when their applications may start, but also to pick among multiple batch-scheduled platforms the one that will produce the shortest turnaround time. As a result, an increasing number of users resort to "redundant requests": several requests are simultaneously submitted to multiple batch schedulers on behalf of a single job; once one of these requests is granted access to compute nodes, the others are canceled. Using simulation as well as experiments with a production batch scheduler we evaluate the impact of redundant requests on (i) average job performance, (ii) schedule fairness, (iii) system load, and (iv) system predictability. We find that some of the popularly held beliefs about the harmfulness of redundant batch requests are unfounded. We also find that the two most critical issues with redundant requests are the additional load on current middleware infrastructures and unfairness towards users who do not use redundant requests. Using our experimental results we quantify both impacts in terms of the number of users who use redundant requests and of the amount of request redundancy these users employ.

Keywords: Job scheduling, Batch scheduling, Redundant requests

1. Introduction

Most parallel computing platforms are accessed via batch schedulers [5] to which users send requests specifying how many compute nodes they need for how long. Batch schedulers can be configured in various ways to implement ad-hoc resource management policies and may maintain multiple queues of pending requests. Most batch schedulers use "backfilling", which allows some requests to jump ahead in a queue to reduce queue fragmentation. Backfilling may happen when a request is submitted, canceled, or when a job runs for less time than initially requested (which is common). The above makes queue waiting time difficult to predict. Some batch schedulers can provide an estimate of queue waiting time based on the current state of the queue. Unfortunately, these estimates do not take backfilling into account, which makes them pessimistic. Conversely, they do not take future submissions of high priority requests into account either, which makes them

This work was supported by the NSF under Award 0546688.

optimistic. Although very recently developed forecasting methods for estimating lower or upper bounds on queue waiting time with certain levels of confidence are promising [1], most users today have at best a fuzzy notion of what queue waiting times to expect. At the same time many of these users have access to multiple batch-scheduled platforms that can be used for running their applications, possibly at different institutions. As a result, rather than picking one target platform based on a poor estimate of queue waiting time, if any, users can send a request to each platform; when one of these requests is granted access to compute nodes the others are canceled. This can be easily implemented by having the application send a callback to the user (or to the program that submitted the requests) when it starts executing.

The admittedly brute-force strategy described above, which we term "redundant requests", is gaining popularity because it obviates the need for difficult platform selection. However, there is a widespread but not verified notion that if "everybody were to use redundant requests" then "bad things would happen". In this paper we attempt to determine the effects of redundant requests. More specifically, we quantify the four following impacts of redundant batch requests:

1. **Impact on average job performance:** while redundant requests may intuitively lead to better load balancing across individual platforms, they may also disrupt the resource management policies implemented by batch schedulers and thereby decrease overall job performance. The question is: by how much do redundant requests improve or worsen average job performance?
2. **Impact on schedule fairness:** redundant requests give users who use them an advantage as they have the luxury to pick the shortest queue waiting times. The question is: how much of an advantage do redundant request provide and how penalized are users who do not employ them?
3. **Impact on system load:** redundant requests cause higher load on the batch schedulers, on the network, and on the middleware infrastructure used to access remote platforms. The question is: do redundant requests cause any of the system's component to become a bottleneck, and if so, which one?
4. **Impact on predictability:** submissions and cancellations of redundant requests cause churn in batch queues, which likely makes them less predictable. The question is: what is the decrease in queue waiting time prediction accuracy when redundant requests are used?

To answer the above questions we use simulations, real-world experiments, and analysis of results obtained by others. We find that several popularly held beliefs regarding the negative effects of redundant batch requests are unfounded. For instance, our experiments show that a batch scheduler, even when running on a mere 1GHz Pentium III processor, can most likely handle large amounts of request redundancy without becoming a bottleneck. In fact, we find that the two main issues with redundant requests are: (i) additional load on the middleware; and (ii) fairness towards users who do not use redundant requests.

This paper is organized as follows. Section 2 presents background on redundant requests and discusses related work. Sections 3, 4, 5, and 6 focus on the four questions above. Section 7 concludes the paper with a summary of our findings and with perspectives on future work.

2. Background and Related Work

Redundant requests can be sent to:

(i) individual batch queues on multiple platforms;
(ii) multiple batch queues of multiple platforms;
(iii) multiple batch queues of a single platform; or
(iv) a single batch queue of a single platform.

In (i), (ii), and (iii), the goal is to avoid selecting a batch queue a priori but instead to use the batch queue on which the shortest queue waiting time is experienced. When using multiple platforms, as in (i) and (ii), a difficulty may be the heterogeneity among these platforms. The computation times requested by each redundant request could be scaled to reflect platform heterogeneity and different numbers of compute nodes could be requested on different platforms. Sophisticated users could thus attempt to tailor their requests to achieve the best response times on each candidate platform. (Note that typical users are not sophisticated, request computation times that are gross overestimations of needed computation times [21], and may not even have a good understanding of the scaling properties of their applications). More importantly, the platform with the shortest queue waiting time could also be a platform with slow compute nodes, meaning that the shortest queue waiting time may not lead to the shortest turnaround time (i.e., queue waiting time plus execution time). Users then face a conundrum: should one wait possibly a long time for a faster platform? Another conundrum arises when using (iii) above. Different queues typically correspond to higher service unit costs. The question is then whether one should wait possibly a long time for a cheaper platform. Option (iv)

above can be useful for "moldable" jobs that can accommodate various numbers of compute nodes. Moldable jobs are common but requesting the optimal number of nodes is known to be difficult [18]. Typically, a larger number of nodes will lead to a longer queue waiting time and to a shorter execution time, while a smaller number of nodes will lead to a shorter queue waiting time and to a longer execution time. One approach is then to send redundant requests for different numbers of nodes. With this approach, one is faced again with a conundrum similar to the one for option (ii): should one wait possibly a long time for a larger number of nodes? Note that option (iv) can be combined with the other three.

There is no one-size-fits-all answer to these conundrums as the solution strongly depends both on the expected application execution times and on the system load, forcing users to use heuristics. These heuristics could be ad-hoc, could use queue waiting time statistics and/or forecasting [1], or could use real-time status information from the batch schedulers that gives a sense of the request's place in the queue (unfortunately many currently deployed schedulers do not provide such information). Finally, note that the number of redundant requests that can be used for (ii), (iii) and (iv) can be bounded by each batch scheduler. Indeed, batch schedulers can typically be configured so that each user can only have a limited number of pending requests in the batch queue(s). In this paper we study option (i), use a simple model for generating redundant requests in a heterogeneous environment, and leave options (ii), (iii), and (iv) for future work.

Previous works have explored the use of redundant requests. Most notably, Subramani et al. [19] and Sabin et al. [16] have studied them as a way to perform job scheduling in a grid platform. In their works, the redundant requests are not initiated by the users but by a metascheduler [7, 17, 15, 2] to potentially offload work to remote platforms. A metascheduler serves as a (centralized or distributed) resource broker and thus controls to some extent how a set of individual platforms are shared and used by a community of users. These works show that using redundant requests can lead to better overall performance, and more so in systems containing clusters with different numbers of nodes. Although related, our work studies redundant requests generated by users without the knowledge of the scheduler(s). This has two important implications. First, a metascheduler can choose remote clusters based on some global knowledge about the system (e.g., queue sizes) in order to let redundant requests "play nice" with each other. Note that as of today, no widely accepted metascheduler is deployed but users may resort to redundant requests directly. By contrast, we study user-driven redundant requests that may negatively disrupt the schedule at

remote clusters. Second, in our study we consider that only some users may be using redundant requests and thus obtain an unfair advantage over users who do not use redundant requests. By contrast, in [19, 16] all users benefit from the same benefits from redundant requests. We argue that in real systems today this is not the case, partly due to the lack of a metaschedulers, but also due to the fact that not all users are created equal: some may not have accounts on multiple platforms, some may not be sufficiently sophisticated to use redundant requests. Another difference between our work and these previous works is that we study the impact of redundant requests on the load on the batch scheduler, on the load on the middleware, and on the predictability of the system. Other relevant related work includes the "placeholder scheduling" technique [13], which allows for a late binding of the application to the resources allocated by a batch scheduler. It provides a simple way to implement redundant requests since a callback is sent to the user when the application is ready to execute. At that time the request submitter (the user or, more likely, a program) may cancel redundant requests.

3. Impact on Average Job Performance

In this section we investigate whether redundant requests negatively impact job scheduling in terms of average job performance. Before presenting our results we detail our experimental methodology and define our metric for job performance.

3.1. Experimental Methodology

3.1.1. *Simulation Model*

We use simulation because experiments on production systems would be prohibitive both in terms of time and money (i.e., service unit allocations on batch-scheduled platforms), because they would be limited to a specific configuration, and because they would hardly be repeatable. We have implemented a simulator using the SimGrid [9] toolkit which provides the needed abstractions and realistic models for the simulation of processes interacting over a network. We simulate a platform that consists of a number of *sites*, where each site holds a parallel platform, say, a *cluster*. Each cluster is managed by its *batch scheduler*. We also simulate a *stream of jobs* at each cluster. Each job requires some number of compute nodes for some duration, sends a request to the local cluster, and may send redundant requests to other clusters. We detail below the components of our simulation model.

Clusters and Batch Schedulers – We simulate a set of N clusters, $C_1, \ldots, C_N$. Cluster C_i contains n_i identical compute nodes. Different node speeds could be accounted for by scaling requested compute times and numbers of nodes (as discussed in Section 2), but this is not straightforward to model. Instead, we limit heterogeneity to the number of nodes in each cluster and to potentially different workloads at these clusters (i.e., more or fewer requests per second). Each cluster is managed by a batch-scheduler, which can use one of three job scheduling algorithms: EASY [10], Conservative Backfilling (CBF) [12], or First Come First Serve (FCFS). The EASY algorithm enables backfilling and is representative of algorithms running in deployed batch schedulers today. Although widely studied, the more complex CBF algorithm is, to the best of our knowledge, only implemented in the OAR batch scheduler [3]. This is also the case for FCFS, but it is a simple algorithm that is commonly used as a base-line comparator. We model each batch scheduler as managing a single queue and we do not consider request priorities.

Workload – Simulating a stream of jobs can be done either by using a workload model or by "replaying" traces collected from the logs of real-world batch schedulers. The results presented in this paper were obtained with the former approach. We use the model by Lublin et al. [11], which is the latest, most comprehensive, and most validated batch workload model in the literature. Accordingly, we model request arrival times using a Gamma distribution (corresponding to the so-called "peak hour" model). Note that [11] goes further by providing a "combined" model that uses two Gamma distributions: one to model job inter-arrival times during peak hours, and one to model the fraction of jobs that arrive during each of the 48 half-hour periods of the day, so as to reflect nocturnal and diurnal trends in the number of submissions. We conducted simulations with the combined model, but the results did not change our conclusions. For simplicity we only present results obtained with the peak hour model. We model the requested number of nodes with a two-stage log-uniform distribution biased towards powers of two. We model the requested compute times with a hyper-Gamma distribution whose p parameter depends on the requested number of nodes.

Unless specified otherwise, we instantiate the parameters of all the distributions using the "model" parameter values derived in [11], to which we refer the reader for all details. We conducted some simulations using real-world traces made available in the Parallel Workloads Archive [4] but, expectedly, did not observe significantly different results. We opted for using a workload model rather than using traces

for the experiments presented in this paper as it is straightforward to modify the model's parameters to study different scenarios.

3.1.2. *Assumptions*

To isolate the effects of redundant requests on scheduling we do not simulate any network traffic. This includes the cost of sending a request to a potentially remote cluster, which is arguably small. More importantly, we also ignore the overhead for sending application input data, if any, to a remote cluster. To use a remote cluster, a user must pre-stage input data on that platform (unless the application streams or downloads its input data directly). However, when using redundant requests, users usually do not pre-stage input data to all remote clusters but wait until nodes are allocated on a particular cluster. The typical approach is then to request extra computation time that will be used to upload application input data to the cluster. Therefore, the only direct impact of redundant requests on the specifics of the workload is that requested computation times may be higher than when there are no redundant requests. (Note that the workload model in [11], which we use in this work, was developed based on logs of requests that most likely were not redundant.) However, we performed experiments in which we increased the requested duration of redundant requests by 10% and 50% and observed no difference in our results. This showed that the results in this paper hold when users request more compute time to allow late binding of application data to remote platforms. Note that it is shown in [19] that the added cost of using redundant requests when proactively transferring application data to all candidate platforms does not impact the effectiveness of using redundant requests.

For the experiments in this section we ignore all overheads due to the network, the middleware, and the batch scheduler itself so as to isolate the effects of redundant request on job performance. We study these overheads in Section 5.

3.2. Performance Metric

We use a popular metric to assess the average job performance: the *average stretch* over all jobs in the system. The *stretch* of a job is the job's turnaround time, which is its execution time plus its queue waiting time, divided by the job's execution time. The stretch is often called "slowdown" because it measures by how much the job execution is slowed down when compared to execution on a dedicated platform. This metric has been used previously, both in practical works to evaluate the performance of batch schedulers and in theoretical works as objective functions, to be minimized, for job scheduling algorithms (which are

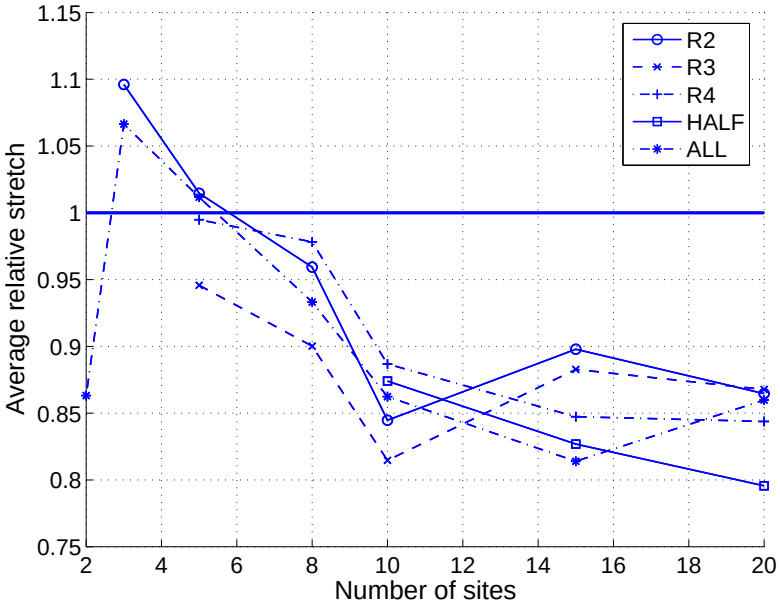

Figure 1. Average relative stretch for redundant request schemes relative to the scheme using no redundant requests, versus the number of clusters, averaged over 50 experiments.

ironically not used by batch schedulers). In this paper we often use the *average relative stretch*, that is the stretch relative to that when no redundant requests are used, averaged over all jobs. A value lower than 1 means that the use of redundant requests is beneficial, while a value higher than 1 means that the use of redundant requests is harmful.

We do not use the average turnaround time as a metric because it can be skewed by long jobs. Furthermore, the stretch makes it possible to easily compare results obtained with different workloads, i.e., different job durations. However, in our specific simulations, our results were essentially unchanged when analyzed with the turnaround time metric.

3.3. Simulation Results

We first simulate an environment that consists of N identical clusters, for $N = 2, 3, 4, 5, 8, 10, 15, 20$. Each cluster contains 128 compute nodes and is managed by a scheduler that uses the EASY algorithm. Each cluster receives a stream of jobs generated according to the model described in Section 3.1.1. We simulate 6 hours of job submissions (around 4,000 jobs given that the mean job interarrival time for the base model in [11] is roughly 5 seconds). We evaluate five redundant

request schemes: $R2$, $R3$, $R4$, $HALF$, ALL, in which a request is sent to 2, 3, 4, half, and all clusters, respectively. One request is always sent to the local cluster, and remote clusters are chosen randomly according to a uniform distribution. Other methods for choosing remote clusters are possible. For instance, inspired by the work in [19], one may select remote clusters based on batch queue lengths. However, it is not clear why users would go through the trouble of picking remote clusters as opposed to just blindly sending requests to all clusters on which they have accounts. Consequently, our method of random selection merely reflects the fact that different users have accounts on different clusters. We also report on results obtained with non-uniform distributions of accounts across clusters.

For now we assume that the same redundant request scheme is used by all jobs. For each experiment we generate N random job streams and simulate the six schemes above for these streams. Figure 1 shows the average relative stretch, averaged over 50 experiments. Over these 50 samples, we measure coefficients of variation ranging approximately from 50% to 5% when going from $N = 2$ clusters to $N = 20$ clusters. The same holds true for all experiments that follow. (We do not show error bars on the graphs to avoid clutter.) Values below 1 in Figure 1 mean that using redundant requests is beneficial on average.

One can see that in the worst case using redundant requests leads to an average relative stretch 10% higher than when not using them, on average. Using redundant requests becomes beneficial regardless of the redundant request scheme for $N > 5$. For $N \leq 5$, it seems that a few short jobs get penalized due to a few lost opportunities for backfilling, thereby increasing their stretch. Accordingly, when using the average turnaround time as a metric, using redundant requests is always beneficial even for $N \leq 5$. This phenomenon disappears, or at least becomes insignificant, as the number of cluster increases. Note that many high-performance computing users today have accounts on five different clusters or more, and that this number will increase as the number of deployed clusters continues to grow.

Redundant requests are beneficial because they allow for a better load-balancing of requests across clusters. Using redundant requests leads to better relative stretches in more than 95% of the experiments for $N = 20$, more than 90% for $N = 15$, and more than 85% for $N = 10$. When using redundant requests leads to a worse relative stretch, it is worse by at most 0.4%, 1.7%, and 2.1% for $N = 20$, $N = 15$, and $N = 10$, respectively. Conversely, when using redundant requests leads to a better relative stretch, the stretch is better on average by 15 to 25%.

Table I. Average relative stretch for three different job scheduling algorithms, and for conservative compute time estimates and exact compute time estimates, for the $HALF$ redundant request scheme, relative to the scheme using no redundant requests, averaged over 50 experiments, for $N = 10$ clusters.

Job Scheduling Algorithm	Average Relative Stretch	
	With Exact Estimates	With Conservative Estimates
EASY	0.88	0.83
CBF	0.90	0.83
FCFS	0.93	0.93

We examined some of the cases in which using no redundant requests is better than using redundant requests. These cases seem idiosyncratic and we could not derive any general reason why redundant requests may not lead to better results beyond particularly "unlucky" sequences of request submissions.

Other algorithms and compute time estimates – The results above were obtained for batch schedulers using the EASY algorithm and assuming that jobs request precisely the amount of compute time that they need. Table I shows results obtained for $N = 10$ clusters and for the $HALF$ redundant request scheme for the EASY, CBF, and FCFS algorithms, and for exact and conservative compute time estimates. We obtained similar results (i.e., all average relative metrics under 1) for other redundant request schemes and other values of $N > 5$. One can see that the average stretch and coefficient of variance of stretches, relative to when no redundant requests are used, are all below 1 no matter what scheduling algorithm is used (EASY, CBF, or FCFS), and whether jobs request exactly the compute time they need ("Exact Estimates") or more compute time than they actually need as is the case in practice ("Conservative Estimates"). We model requested compute times using the "ϕ model" proposed in [24], with $\phi = 0.10$, which leads to a uniformly distributed overestimation factor with mean 2.16. A more sophisticated model was recently proposed in [21], and although we may use it to repeat these experiments in future work, we do not expect the results to be significantly different.

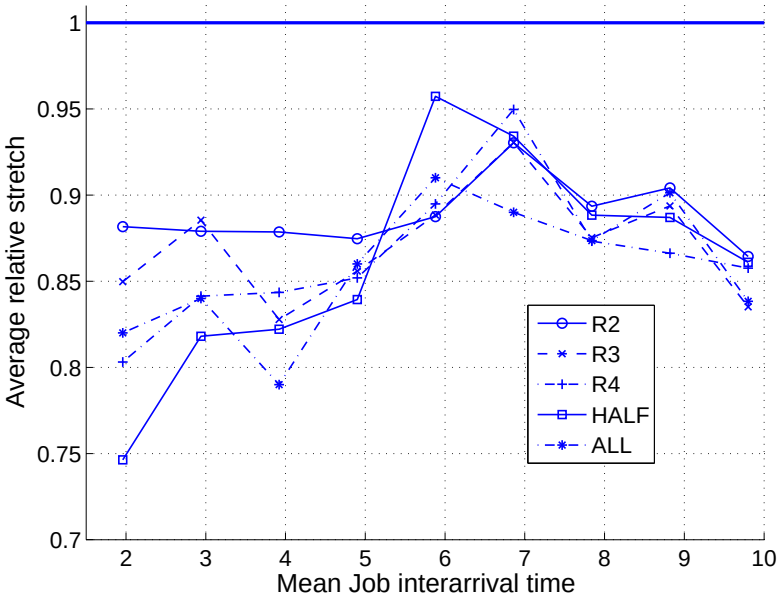

Figure 2. Average stretch for redundant request schemes relative to the scheme using no redundant requests, versus job interarrival times, averaged over 50 experiments, for $N = 10$ clusters.

Table II. Average relative stretches for non-uniformly distributed redundant requests, relative to the scheme using no redundant requests, averaged over 50 experiments, for $N = 10$ clusters.

Redundant Request Scheme	$R2$	$R3$	$R4$	$HALF$
Average Relative Stretch	0.94	0.95	0.88	0.89

Non-uniform redundant request distribution – We repeat our first experiment for $N = 10$ clusters, simulating redundant requests schemes that pick remote clusters at random but biased towards some clusters. We model the probability of picking remote cluster C_1 is twice as high as the probability of picking remote cluster C_2, which is twice as high as the probability of picking remote cluster C_3, and so on. This (arbitrary) distribution is heavily biased (half of the clusters are each picked with only probability 6.25%). Results are shown in Table II, averaged over 50 experiments. One can see that on average the use of redundant requests is beneficial for performance, even when the targets

of the redundant requests are not uniformly distributed. In fact, the results are similar to those obtained with a uniform distribution.

Job interarrival times – One may wonder whether using redundant requests is more or less harmful given different levels of workload. Figure 2 shows average relative stretch for all redundant request schemes in a 10-cluster platform, averaged over 50 experiments, for various job interarrival times (in seconds). The base model proposed in [11] produces mean interarrival times of 5.01 seconds, which is the mean of a Gamma distribution with parameters $\alpha = 10.23$ and $\beta = 0.49$ (the mean is equal to $\alpha \times \beta$). To simulate other workloads we vary the value of α from 4 to 20, leading to interarrival times between approximately 2 and 10 seconds. One can see on the figure that using redundant requests improves average relative stretch regardless of the job interarrival time.

Heterogeneity – So far, all our experiments were conducted on a homogeneous system with identical clusters and statistically identical job streams at all clusters. To evaluate whether redundant requests are harmful in a heterogeneous environment we conduct the following experiment. We simulate $N = 10$ clusters, where each cluster contains a number of nodes picked randomly among 16, 32, 64, 128, and 256. Each cluster receives a job stream with job interarrival times picked randomly between 2 and 20 seconds. Jobs arriving at a cluster request a number of nodes between 1 and the number of nodes available at that cluster. Therefore, jobs arriving at a large cluster may not be runnable on smaller clusters. Table III shows results for all redundant request schemes, relative to the scheme using no redundant requests, averaged over 50 experiments. One can see that using more redundant requests

Table III. Average stretch for heterogeneous platforms, relative to the scheme using no redundant requests, averaged over 50 experiments, for $N = 10$ clusters.

Redundant Request Scheme	Average Relative Stretch
$R2$	0.83
$R3$	0.74
$R4$	0.71
$HALF$	0.63
ALL	0.67

is even more beneficial (higher performance and better fairness) than in the homogeneous case across the board. This is because the load balancing due to redundant requests is more effective in a heterogeneous system. This result corroborates the findings in [19, 16].

3.4. Discussion

Virtually all simulation results presented so far indicate that the average job performance is improved when redundant requests are used. These results hold across ranges of job scheduling algorithms, job interarrival times, number of clusters, heterogeneity across clusters, distribution of redundant requests across clusters, and whether requested compute times are exact or conservative. Essentially, this is because redundant requests improve load balancing across clusters.

The reader may wonder whether race conditions are involved when using redundant requests. Indeed, two requests may start at approximately the same time and one of them may be canceled only after it is started. In our simulation we observed that such race conditions where rare (occurring for less than 0.005% of the jobs). However, for larger latencies of receiving notifications from a starting request and of canceling a request, race conditions could be slightly more frequent. One issue could then be wastage of CPU time. Unless latencies are enormous, wastage should not be significant and well worth the cost for the benefits of redundant requests, as will be seen in the next section. For instance, a 5 second delay when canceling a job that has just started on 64 processors would only waste 0.08 hours of CPU time, which is a minuscule fraction of typical user allocations. Another issue is whether the request submitter (user or program) should pay attention to such race conditions. Fortunately, a pending request or a running job are canceled in exactly the same way when using current batch scheduler interfaces. The only issue is that the request submitter must be prepared to ignore stale notifications from requests that are in the process of being canceled.

4. Impact on Schedule Fairness

We now investigate the question of schedule fairness, that is: do all jobs fare equally when redundant requests are used? The experimental methodology and assumptions are identical to those used in Section 3. We first present results obtained when all jobs use identical redundant requests schemes, and then turn to the case in which only a subset of the jobs use redundant requests.

Table IV. Coefficients of variation of stretches for three different job scheduling algorithms and for conservative compute time estimates and exact compute time estimates, for the $HALF$ redundant request scheme, relative to the scheme using no redundant requests, averaged over 50 experiments, for $N = 10$ clusters.

Job Scheduling Algorithm	Relative C.V. of Stretches	
	With Exact Estimates	With Conservative Estimates
EASY	0.83	0.83
CBF	0.86	0.83
FCFS	0.93	0.93

4.1. When All Jobs Use Redundant Requests

In this section we use the *coefficient of variation of stretches* (i.e., the standard deviation divided by the average, in percentage, over all jobs in the system), as a way to evaluate the fairness of a schedule. A lower coefficient of variation indicates a fairer schedule since on average all jobs achieve a stretch closer to the mean. Another popular metric that is related to the fairness of a schedule is the maximum stretch, and the conclusions from our results were not changed when using this metric.

We present results similar to those in Section 3.3, but from the perspective of fairness. Figure 3 shows the coefficient of variation of job stretches relative to that experienced when no redundant requests are used for all redundant request schemes, for $N = 2, 3, 4, 5, 8, 10, 15, 20$ identical clusters with statistically identical job streams, averaged over 50 experiments. One can see that in all cases using redundant requests improves the fairness of the schedule by approximately 10 to 25%. The maximum job stretch, which is often used as a measure of fairness, is improved even more by the use of redundant requests, from 10 to 60% on average (not shown on the figure). The improved fairness can be attributed to the fact that redundant requests lead to better load-balancing. Table IV shows results for the three job scheduling algorithms for either exact or conservative compute time estimates, and here again we see that fairness is improved by the use of redundant requests. Finally, Table V shows that the improvement in fairness holds for non-uniformly distributed redundant requests, as modeled in Section 3.3. These results also hold over job interarrival times from 2 to 10s as well as for heterogeneous platforms.

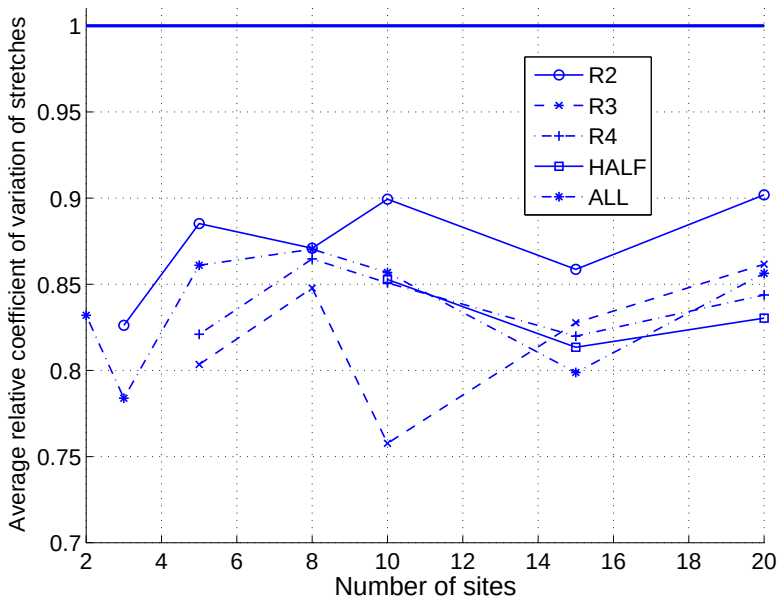

Figure 3. Coefficient of variation of stretches for redundant request schemes relative to the scheme using no redundant requests, versus the number of clusters, averaged over 50 experiments.

Table V. Coefficient of variation of stretches for non-uniformly distributed redundant requests, relative to the scheme using no redundant requests, averaged over 50 experiments, for $N = 10$ clusters.

Redundant Request Scheme	$R2$	$R3$	$R4$	$HALF$
Relative C.V. of Stretches	0.94	0.92	0.88	0.86

4.2. When Only Some Jobs Use Redundant Requests

In all results presented so far, we have assumed that all jobs benefit from the (same) opportunity to use redundant requests. This is most likely not the case in the real-world. Some users may have accounts only on one cluster. Some users may have accounts on all clusters. Some applications may need to run on a specific cluster due to the proximity of a large data set. Some users may not be sufficiently sophisticated to

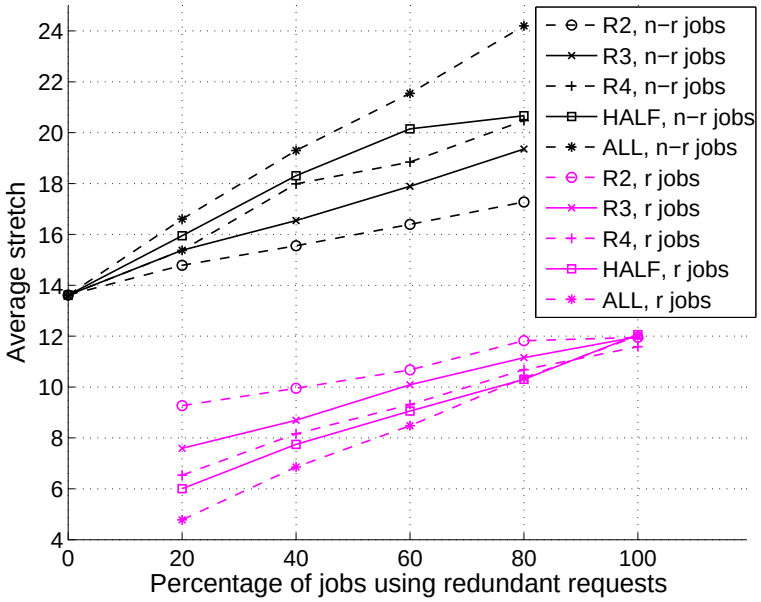

Figure 4. Average stretch for jobs using redundant requests ("r jobs") and jobs not using redundant requests ("n-r jobs"), for different redundant request schemes, versus the percentage of jobs using redundant requests, averaged over 50 experiments, for $N = 10$ clusters.

use redundant requests. An important question is then: how penalized are jobs not using redundant requests when compared to those using redundant requests?

To answer the above question we conduct experiments assuming that only p percent of the jobs use redundant requests (these jobs all use the same redundant request scheme). Figure 4 plots average (absolute) stretch versus p, for jobs using redundant requests ("r jobs") and jobs not using them ("n-r jobs"), for different redundant request schemes. These experiments are for $N = 10$ clusters, and each data point is averaged over 50 experiments. One can see several interesting features on this graph. The average stretch is better when $p = 100$ than when $p = 0$, which confirms the results of the previous experiments. One can see that when p increases, the average stretch increases, for both types of jobs. Therefore, jobs using redundant requests negatively impact the performance perceived by jobs not using redundant requests. This impact grows roughly linearly with p. Furthermore, this negative impact is larger for the schemes using more redundant requests per jobs. For instance, if 40% of jobs send requests to all N clusters, their stretches are reduced on average by approximately a factor 2, but the stretches

of the other jobs increase by over 40%. Note that jobs using redundant requests all benefit from using more redundant requests, and from fewer jobs using redundant requests.

It turns out that one can fit a planar model to the trends in Figure 4 reasonably well. Let p be the fraction of jobs using redundant requests, and r the fraction of the available clusters to which these jobs send their requests. For instance, if $p = .20$ and $r = .50$ then 20% of the jobs send requests to half the clusters, and 80% of the jobs send requests to only one cluster. (We assume that if $p = 0$ then $r = 0$.) We define the performance of jobs as the ratio of their average stretch over the averaged stretch achieved when no redundant requests are used and denote it by $RelStrechR(p, r)$ for jobs that use redundant requests and by $RelStretchNR(p, r)$ for jobs that do not. In both cases, lower values denote better performance. We have seen experimentally that $RelStretchR(p, r) \leq 1$ (i.e., redundant requests are always beneficial to the jobs that use them) and $RelStretchNR(p, r) \geq 1$ (i.e., jobs not using redundant requests suffer). By performing a planar fit for $RelStretchR(p, r)$ and $RelStretchNR(p, r)$ using our experimental data we obtain:

$$RelStretchR(p, r) \approx 0.467p - 0.175r + 0.500, \text{ and}$$
$$RelStretchNR(p, r) \approx 0.620p + 0.294r + 0.875.$$

These two fits have an average relative error of 5.9% and 4.1%, respectively. We can now easily see the trends that we observed in Figure 4: $RelStretchNR$ increases when either p or r increases (redundant requests are harmful to a user who does not use any), while $RelStretchR$ increases with p and decreases with r (it is best to be one of the few users who use redundant requests but to use as many of them as possible). We can also see that $RelStretchNR$ increases faster than $RelStretchNR$ when p increases.

4.3. Discussion

One may wonder about the values of p and r in real systems today. We are aware of several users employing redundant requests routinely, and we feel that the number of such users is likely to grow (perhaps in part due to reading this article), at least until the, for now, elusive establishment and deployment of metaschedulers. Nevertheless, getting a good handle of p and r values is a challenging proposition. This is in part because redundant requests are often difficult to detect. In fact, an important question is whether preventing their use to improve fairness is necessary. One problem here is that the notion of what value of

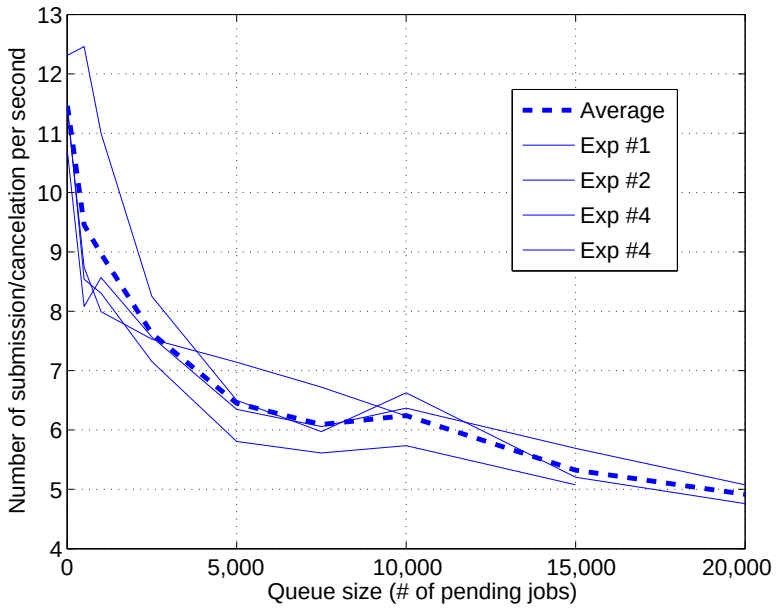

Figure 5. Measured throughput of the OpenPBS/Maui batch scheduler in number of request submissions/cancellations per second versus batch queue size in number of pending requests.

Loss is unacceptable is subjective. Some may even argue that unsophisticated users or users who have an account on a single machine being at a disadvantage is simply a fact of life. At any rate, without resolving philosophical questions regarding which levels of unfairness are acceptable and which are not, the results and derived empirical models in the previous section make it possible to quantify unfairness.

5. Impact on System Load

To isolate the effects of redundant requests on scheduling, our simulations so far have ignored all overheads involved in job submissions and job cancellations. In this section we study the impact of such overheads in terms of the batch scheduler, the network, and the middleware.

5.1. Batch Scheduler

Our experiments were conducted assuming that batch schedulers could make scheduling decisions instantaneously. But in reality redundant

requests require that batch schedulers perform more work due to more request submissions and cancellations.

To estimate the job submission/cancellation throughput of a batch scheduler we perform the following experiment. We use an installation of OpenPBS v2.3.16 using the Maui scheduler v3.2.6p.13, running on a 1GHz Pentium III, which is the front-end of a 16-node Linux cluster. The cluster runs a long job that monopolizes all compute nodes for the duration of the experiment, so that pending jobs are never executed. We submit a fixed number of random jobs to generate a given queue size (in number of pending jobs). We then saturate the batch scheduler with job submissions and job cancellations by running multiple processes that continuously submit new jobs using the `qsub` command and delete the job at the head of the queue using the `qdel` command. Deleting the job at the head of the queue causes the maximum amount of churn. Figure 5 shows the average number of job submissions/cancellations per second versus the queue size. Each solid curve corresponds to a 12-hour experiment. We do not distinguish these curves because experiments were identical. The thick dashed line shows the average over all experiments. The variations among experiments are in part due to non-deterministic load on the front-end node, which was not dedicated to our experiments but was only mostly quiescent. Note also that some curves do not show values for the higher queue sizes as experiments were interrupted due to the job scheduler process running out of memory, due to memory leaks in the version of OpenPBS deployed on the cluster.

One can see that as the queue size increases the throughput of the job scheduler decreases sharply at first and then slower, in a somewhat exponential manner. When the queue is empty the job scheduler can perform around 11 request submissions and 11 request cancellations per second. This number drops to about 5 when the queue contains 20,000 pending requests. In practice, the number of jobs in the queue of a batch scheduler is on the order of a few thousands. Using the workload model in [11], the queue of a batch scheduler grows by about 700 jobs per hour during so-called "peak" hours (independently of the size of the cluster). Therefore, after 12 hypothetical peak hours, the queue would contain under 10,000 jobs. The use of redundant requests should not increase queue sizes significantly since redundant requests are canceled upon the start of job execution (therefore, *in steady-state*, using redundant requests does not cause significantly more requests to be in the system). This simple observation is confirmed in simulation. For instance, when simulating $N = 10$ clusters for a 24-hour period, we found that the average maximum queue size across all clusters for the *ALL* redundant request scheme is larger than when no redundant requests are used by less than 2%.

Consider a system with N clusters, with mean job interarrival time of iat seconds at each cluster. If all jobs use r requests, then on average each cluster will receive r/iat requests per second and $(r-1)/iat$ request cancellations per second (assuming that the system is in steady state). Conservatively assuming that all queues contain 10,000 requests, the experimental results in Figure 5 indicate that the batch schedulers could support 6 submissions and 6 cancellations per second. Therefore, the batch schedulers operate within their achievable throughput if $r/iat \leq 6$. Assuming job interarrival times of 5 seconds, which corresponds to the peak hour rate of the model in [11], we obtain $r < 30$. This means that in a multi-cluster system, the load on batch schedulers due to redundant requests is tolerable as long as jobs do not use more than 30 redundant requests on average. This is well above the average level of request redundancy that we expect to see in real-world systems today.

The back-of-the-envelope calculation above is optimistic as it assumes that requests are uniformly distributed among clusters. But it is also pessimistic as it assumes that the job interarrival time is always 5 seconds and that all jobs use redundant requests. Furthermore, we measured the throughput of the batch scheduler on a 1GHz Pentium III, and architectures with higher performance would likely improve the throughput further. Our conclusion is that, at least in today's system, the use of redundant requests is unlikely to cause a prohibitive load increase on batch schedulers.

5.2. Network and Middleware

Redundant requests also increase the load on the network and the middleware layer used to access remote platforms. In the previous section we have determined that, with a queue size of 10,000, each batch scheduler can process on average 12 transactions per second. In this section we assess whether the network or the middleware infrastructure can support this throughput.

As discussed in Section 3.1.2, redundant requests cause no extra network load for transferring application data. Even if the network payload of a job submission or cancellation were on the order of hundreds of KBytes (for instance large SOAP [6] messages), most networks connecting a batch scheduler to the Internet can easily support tens of such interactions per second.

Beyond the network, redundant requests generate more load on the middleware system itself. Using a middleware layer to submit jobs to a Grid service, layered above a batch scheduler, entails many operations (service instantiations, marshalling and unmarshalling of SOAP

transactions, disk writes, etc.). Several authors have measured the performance and scalability of the service layer. For instance, the authors of [8] have measured the performance of various operations in service interactions. Their experiments, conducted on a dual-processor 2GHz Pentium 4 Xeon with 1GByte of RAM, show that the most time consuming operations are the serializations and de-serializations of input/output data. Many results are presented for a variety of SOAP implementations and of data structures, both for Linux and Windows. For instance, consider the results obtained when using the gSOAP [22] toolkit v2.7e to send "mesh interface elements", i.e., data structures containing two integers and a double precision number, to a service instance. The end-to-end performance results in [8] indicate that one can send an array of more than 30,000 such elements in less than 0.08 seconds. Assuming that all service interactions are serialized (which is probably pessimistic), the service infrastructure can therefore support more than 12 such interactions per second. This means that even if the interaction with a batch scheduler service to submit and cancel jobs entailed serialization and deserialization of 30,000 mesh interface elements, more than 450KB, the batch scheduler itself would still be the bottleneck, according to our results in Section 5.1. We argue that interacting with a batch scheduling would require orders of magnitude fewer serializations and deserializations.

These encouraging results must be put in perspective with the transaction throughput achieved by currently deployment and full-fledge middleware implementations such as the Globus®WS-GRAM [20]. Results obtained with GRAM running on a 2.16GHz AMD K7 processor are presented in [14] (Table 4). For the latest grid service version of GRAM (i.e., GT4 WS-GRAM), a throughput of slightly under 60 transactions per minutes can be sustained, or under one transaction per second. If a job cancellation causes roughly the same overhead as a job submission as far as GRAM is concerned, which is a reasonable assumption, then 0.5 job submissions and 0.5 job cancellations can be processed per second. Let us follow the same reasoning as in Section 5.1: assuming that the job interarrival time is $iat = 5$ seconds and that all jobs use r requests, then $r/iat \leq 0.5$ leading to $r < 3$. This indicates that the current version of GRAM would be the bottleneck for a system in which, for instance, all jobs use 3 or more redundant requests during peak job submission hours on production systems. (Recall that we found this number to be equal to 30 before the batch scheduler becomes the bottleneck.)

Table VI. Queue waiting time overestimation statistics for $N = 10$ clusters.

	0% jobs using redundant requests	40% jobs using redundant requests (*ALL*)	
		jobs not using redundant requests	jobs using redundant requests
Average	9.24	77.54	36.28
C.V.	204.98%	189.47%	205.26%

5.3. Discussion

In the light of the previous results we conclude that, in the presence of redundant requests, the middleware layer, or at least its current implementation, is the bottleneck of the system, tolerating under 3 redundant requests per job during peak hours of job submissions (i.e., one job every 5 seconds). This may inherently preclude the wide-spread use of redundant requests, although one can expect that current implementations as well as the hardware on which the GRAM service runs will improve with time, especially given the promising SOAP benchmarking results in works like [8].

6. Impact on Predictability

Although redundant requests are typically employed to obviate the need for platform selection, and thus for queue waiting time prediction, this does not mean that queue waiting time predictions become irrelevant for all users. Users always appreciate having some notion of when their job will start, whether they use redundant requests or not. Also, when submitting redundant requests to multiple clusters with different processor speeds, having estimates of queue waiting times would help in performing request selection (see the discussion in Section 3.1.2). Therefore, it is interesting to see whether redundant requests make queue waiting times more difficult to predict. One reason why they may is because they cause churn in batch queues. Therefore, users using redundant requests, although benefiting from shorter queue waiting times (as seen in the previous sections), may have a decreased ability to predict these queue waiting times. Furthermore, users not using redundant requests would then experience not only higher but also less predictable queue waiting times.

Batch schedulers can provide an estimate of queue waiting time upon request submission. This can be possible because the job scheduling algorithm assigns a "reservation" to a request as soon as it is submitted, as with the CBF algorithm. Otherwise, the queue waiting time can be estimated via a simulation of the batch queue. For instance, the `show_guess` command used in the S-Cubed portal (`http://www.sdsc.edu/PMaC/S-cubed/`) performs such a prediction for the Catalina batch scheduler [23]. At any rate, queue waiting time forecasting by the job scheduler can hardly be accurate. For instance, with the EASY job scheduling algorithm, request submissions can delay the execution of a previously submitted request. Perhaps more importantly, queue waiting time forecasts based on analysis or simulation of the queue are always conservative because requested computation times are themselves known to be very conservative [21]. Therefore, running jobs finish executing earlier than expected and pending jobs can start executing earlier as well. Generally, batch schedulers are complex software systems that can be configured in many different ways that affect the job scheduling process. For instance, jobs requesting particular numbers of compute nodes, submitted by particular users, or submitted to a different queue can be assigned higher priority. The arrivals of high-priority jobs, which are difficult to predict, can disrupt the schedule.

In order to investigate queue waiting time predictability, we first simulate $N = 10$ identical 128-node clusters with statistically identical job streams (conservative compute time estimates are used), and with no redundant requests in the system. We measure the average ratio of the predicted queue waiting time to the effective queue waiting time and the coefficient of variation of these ratios, across all jobs. Queue predictions were based on the reservations determined by the CBF job scheduling algorithm [12], which was used by all clusters. The left side of Table VI shows the results, averaged over 50 experiments. One can see that on average queue waiting times are over-predicted by a factor 9.24, with a coefficient of variation over 200%. The high amount of over-prediction is due in part to the fact that requested compute times are over-estimated by approximately a factor 2.16 on average in our simulations. The right side of the table shows the results for jobs using no redundant requests and for jobs using redundant requests, when 40% of the jobs use the *ALL* redundant request scheme. For jobs using redundant requests, the queue waiting time is predicted as the minimum predicted queue waiting time over all redundant requests. For both types of jobs, the average overestimation of queue waiting time is dramatically increased: about four times as much for jobs using redundant requests, and about eight times as much for jobs not using redundant requests. We have

observed similar results in other experiments, with higher increases in over-prediction for higher numbers of redundant requests per job and/or for higher fractions of jobs employing redundant requests. In all cases, jobs using no redundant requests are penalized.

6.1. Discussion

It is important to note that our results are for predictions based on the state of the queue and on requested compute times. As discussed above, this prediction method, although in use, is known not to be very effective due mainly to compute time overestimations and to backfilling. Nevertheless, one can note that there is no increase in the coefficients of variation of queue waiting time overestimations due to the use of redundant requests. This suggests that one could just scale down the predictions to achieve similar accuracy (or lack thereof) as when no redundant requests are used. Intuitively, this amounts to accounting for the steady-state queue churn added by redundant requests.

7. Conclusion

In this paper, we have explored whether sending redundant requests to batch-scheduled parallel computing platforms causes undesirable effects. In particular, we have focused on four impacts: (i) impact on average job performance, (ii) impact on schedule fairness, (iii) impact on system load, and (iv) impact on system predictability. Our results for each can be summarized as follows:

(i) When all jobs use redundant requests their average performance is improved, including cases in which redundant requests are sent to all available platforms. These results hold across ranges of job scheduling algorithms, job interarrival times, number of clusters, and whether requested compute times are exact or conservative.

(ii) Users who do not use redundant requests are penalized, and the more so when many other users use many redundant requests. One may wonder what fraction of users use redundant requests in real systems, and to what fractions of the available platforms they send these requests. This paper does not answer this question, but Section 4.3 presents an empirical model of performance benefits and performance penalties as a function of these two fractions.

(iii) Even large numbers of redundant requests (e.g., 30 requests per job) will likely not impose unacceptable load on the batch sched-

ulers or the network. We have determined that currently available middleware implementations running on currently available hardware would likely not scale to situations in which most users use redundant requests during peak hours of job submissions on production systems. One can however expect that middleware implementations (and the hardware on which they run) will improve with time, especially in the light of promising benchmark results.

(iv) The use of redundant requests reduces the accuracy of queue waiting time predictions based on the state of batch queues by close to one order of magnitude. This is expected as redundant requests increase queue churn. We must note that such queue waiting time prediction techniques are known not to be very accurate in the first place.

The question of whether and how to limit the number of redundant requests at a global system level remains open. Based on our results, some may feel that there are compelling reasons for curtailing redundant requests at least to some extent. Controlling the use of redundant requests in federated distributed platforms in which each platform has its own local scheduler is likely not straightforward and would require some software infrastructure (e.g., a metascheduler). At the moment, a typical "local" approach used today by some platform administrators consists of identifying users who use redundant requests (for instance because they cancel many requests) and of trying to discourage these users with methods ranging from an indignant e-mail to privilege removals. Note that although these administrators are typically concerned about the impact of redundant requests on the batch scheduler (e.g., increased load on the scheduler, increased queue sizes), our results show that such concerns are most likely unjustified.

We see opportunities for future work along three directions. First, a deeper investigation of the impact of redundant requests on queue waiting time predictions would be interesting. Section 6 mentions the possibility that simple predictions based on the state of the queue at job submission time could just be scaled to account for the extra queue churn cause by redundant requests. While determining whether this is indeed the case could be interesting, we feel that focusing on prediction techniques that are more accurate in the first place is more relevant. In particular, promising statistical techniques have been recently proposed in [1] and one may wonder how redundant requests affect such techniques. Ironically, better queue waiting time predictions may reduce the need for redundant requests since their primary goal is to remedy the problem of poor queue waiting time predictions in the first place.

An exploration of this intriguing interplay between redundant requests and queue predictions is bound to lead to interesting results.

A second direction for future work is the exploration of the effect of additional uses of redundant requests as described in Section 2: redundant requests sent to multiple batch queues at one or more clusters and redundant requests sent to a single batch queue for several configurations of a moldable parallel application. More redundant batch requests will exacerbate the negative impact of system load that we have identified in this paper. But more interesting is the question of whether the implications of our results regarding effects on average job performance and schedule fairness would be changed and if so, how.

Finally, a third direction for future work is the investigation of heuristics for request selection with multiple redundant requests in batch queues of a single system, multiple redundant requests in a single batch queue, and clusters with different node speeds. These situations lead to several conundrums for the user, as discussed in detail in Section 2. It would be interesting to study which heuristics are best to resolve these conundrums, whether these heuristics provide significant improvement over naïve approaches (e.g., pick the first request that makes it through any batch queue), and whether the use of queue predictions such as the ones in [1] would be a good basis on which such heuristics could be designed.

Acknowledgements

The author is grateful to Prof. Wolski's MAYHEM Lab in the Computer Science Dept. at the University of California Santa Barbara for providing the cluster and OpenPBS/Maui installation used for the experiments in Section 5. The author also wishes to thank the reviewers for their insightful comments and suggestions.

References

1. Brevik, J., Nurmi, D., Wolski, R.: 2006, Predicting Bounds on Queuing Delay for Batch-scheduled Parallel Machines. In: Proc. of the 11th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP), pp. 110–118.
2. Bucur, A., Epema, D.: 2003, The Performance of Processor Co-Allocation in Multicluster Systems. In: Proc. of the 3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid), pp. 302–309.
3. Capit, N., Da Costa, G., Georgiou, Y., Huard, G., Martin, C., Mounié, G., Neyron, P., RIchard, O.: 2005, A Batch Scheduler with High Level Compo-

nents. In: Proc. of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid), pp. 776–783.
4. Feitelson, D.: 2006, Parallel Workloads Archive. http://www.cs.huji.ac.il/labs/parallel/workload/.
5. Feitelson, D. G., Rudolph, L., Schwiegelshohn, U.: 2004, Parallel Job Scheduling — A Status Report. In: Proc. of the 10th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), Lecture Notes in Computer Science, Vol. 3277, pp. 1–16.
6. Gudgin, M., Hadley, M., Mendelsohn, U., Moreau, J.-J., Canon, S., Nielsen, H.: 2003, Simple Object Access Prototol 1.1. http://www.w3.org/TR/SOAP/.
7. Hamscher, V., Schwiegelshohn, U., Streit, A., Yahyapour, R.: 2000, Evaluation of Job-Scheduling Strategies for Grid Computing. In: Proc. of the 1st IEEE/ACM International Workshop on Grid Computing, Lecture Notes in Computer Science, Vol. 1971, pp. 191–202.
8. Head, M. R., Govindaraju, M., Slominski, A., Liu, P., Abu-Ghazaleh, N., van Engelen, R., Chiu, K., Lewis, M. J.: 2005, A Benchmark Suite for SOAP-based Communication in Grid Web Services. In: Proc. of the 2005 ACM/IEEE Conference on Supercomputing (SC), pp. 19–31.
9. Legrand, A., Marchal, L., Casanova, H.: 2003, Scheduling Distributed Applications: The SimGrid Simulation Framework. In: Proc. of the 3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid), pp. 138–145.
10. Lifka, D.: 1995, The ANL/IBM SP Scheduling System. In: Proc. of the 1st Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), Lecture Notes in Computer Science, Vol. 949, pp. 295–303.
11. Lublin, U., Feitelson, D.: 2003, The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs. Journal of Parallel and Distributed Computing (JPDC), **63(11)**, pp. 1105–1122.
12. Mu'alem, A., Feitelson, D. G.: 2001, Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. IEEE Transactions on Parallel and Distributed Computing (TPDS), **12**, pp. 529–543.
13. Pinchak, C., Lu, P., Goldenberg, M.: 2002, Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences. In: Proc. of the 8th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), Lecture Notes in Computer Science, Vol. 2537, pp. 85–105.
14. Raicu, I.: 2005, A Performance Study of the Globus Toolkit ®and Grid Services via DiPerf, an Automated Distributed Performance Testing Framework. Master's thesis, University of Chicago.
15. Ranganathan, K., Foster, I.: 2002, Decoupling Computation and Data Scheduling in Distributed Data-intensive Applications. In: Proc. of the 11th IEEE International Symposium for High Performance Distributed Computing (HPDC), pp. 352–358.
16. Sabin, G., Kettimuthu, R., Rajan, A., Sadayappan, P.: 2003, Scheduling of Parallel Jobs in a Heterogeneous Multi-Site Environment. In: Proc. of the 9th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), Lecture Notes in Computer Science, Vol. 2872, pp. 87–104.
17. Shan, H., Oliker, L., Biswas, R.: 2003, Job Superscheduler Architecture and Performance in Computational Grid Environments. In: Proc. of the 2003 ACM/IEEE Conference on Supercomputing (SC), pp. 44-58.
18. Srinivasan, S., Subramani, V., Kettimuthu, R., Holenarsipur, P., Sadayappan, P.: 2002, Effective Selection of Partition Sizes for Moldable Scheduling of

Parallel Jobs. In: Proc. of the 9th International Conference on High Performance Computing (HiPC), Lecture Notes in Computer Science, Vol. 2552, pp. 176–182.
19. Subramani, V., Kettimuthu, R., Srinivasan, S., Sadayappan, P.,: 2002, Distributed Job Scheduling on Computational Grids using Multiple Simultaneous Requests. In: Proc. of the High Performance and Distributed Conference (HPDC), pp. 359–366.
20. The Globus ®Alliance: 2006, WS GRAM: Developer's Guide. http://www.globus.org/toolkit/docs/4.0/execution/wsgram/.
21. Tsafrir, D., Etsion, Y., Feitelson, D. G.: 2005, Modeling User Runtime Estimates. In: Proc. of the 11th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), Lecture Notes in Computer Science, Vol. 3834. pp. 1–35.
22. van Engelen, R., Gallivan, K.: 2002, The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks. In: Proc. of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid), pp. 128–135.
23. Yoshimoto, K.: 2005, The Catalina Batch Scheduler. http://www.sdsc.edu/catalina/.
24. Zhang, Y., Franke, H., Moreira, J. E., Sivasubramaniam, A.: 2001, An Integrated Approach to PArallel Scheduling Using Gang-Scheduling, Backfilling, and Migration. In: Proc. of the 7th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), Lecture Notes on Computer Science, Vol. 2221. pp. 133–158.