Scheduling Task Parallel Applications for Rapid Application Turnaround on Enterprise Desktop Grids

Derrick Kondo[1], Andrew A. Chien[2], Henri Casanova[3]
[1]*Laboratoire de Recherche en Informatique/INRIA Futurs*
[2]*Dept. of Computer Science and Engineering, University of California, San Diego*
[3] *Dept. of Information and Computer Sciences, University of Hawai'i at Manoa*

Abstract.
Desktop grids are popular platforms for high throughput applications, but due to their inherent resource volatility it is difficult to exploit them for applications that require rapid turnaround. Efficient desktop grid execution of short-lived applications is an attractive proposition and we claim that it is achievable via intelligent resource selection. We propose three general techniques for resource selection: resource prioritization, resource exclusion, and task duplication. We use these techniques to instantiate several scheduling heuristics. We evaluate these heuristics through trace-driven simulations of four representative desktop grid configurations. We find that ranking desktop resources according to their clock rates, without taking into account their availability history, is surprisingly effective in practice. Our main result is that a heuristic that uses the appropriate combination of resource prioritization, resource exclusion, and task replication can achieve performance within a factor of 1.7 of optimal in practice.

Keywords: Desktop Grids, Network of Workstations, Resource Management, Scheduling

1. Introduction

Cycle stealing systems that harness the idle cycles of desktop PCs and workstations date back to the PARC Worm [26] and have shown widespread success with popular projects such as SETI@home [48, 45], the GIMPS [25], Folding@Home [46], FightAidsAtHome [20], Computing Against Cancer [10], and others sustaining the throughput of over one million CPU's and 10's of Teraflops/seconds [38]. These successes have inspired similar efforts in the enterprise as a way to maximize return on investment for desktop resources by harnessing their cycles to service large computations. As a result, numerous academic projects have explored developing a global computing infrastructure for the internet [39, 43, 2, 11, 9, 5, 19] and local-area networks [32, 4, 24]. In addition, commercial products have also emerged [18, 52, 49, 40]. We term such computing infrastructures *desktop grids*, and while these systems can be used in a variety of environments, in this paper we focus on the enterprise setting (e.g., within an institution).

A challenge for the efficient utilization of desktop grids for compute-intensive applications is that of resource management. While these two issues have been thoroughly studied in the areas of parallel and Grid computing, resource failures are typically considered as rare events. By contrast, desktop grid resources are inherently volatile. Due to the lack of resource management and application scheduling techniques that account for resource volatility, the traditional use of desktop grids has focused on high-throughput applications that consist of *large numbers* (i.e., orders of magnitude larger than the number of available resources) of *independent* tasks. The performance metric used in this scenario is the asymptotic task completion rate, that is the number of tasks that are completed per time unit when the application execution is in steady-state.

In this paper we consider parallel applications that consist of independent, identical tasks, but we study these applications for numbers of tasks that are relatively small, i.e., comparable to the number of available resources. Numerous interactions with industrial companies by one of the authors suggest that desktop grids within the enterprise are often underutilized. Also, applications in a company's workload often require relatively rapid turnaround (for example, within a day's time). As a result, applications often consist of a moderate number of individual tasks. Consequently, a scenario in which the number

This material is based upon work supported by the National Science Foundation under Grant ACI-0305390.

of tasks is of an order of magnitude comparable to the number of resources is not uncommon. As such, rather than asymptotic work rate, the relevant performance metric is the application execution time, or *makespan*.

This work focuses on minimizing the makespan of a single application, rather than trying to optimize the performance of multiple, competing applications, or jobs. The design of so-called "job scheduling" strategies that focus on aggregate job performance and fairness among jobs that belong to different users is outside the scope of this paper as we solely focus on application scheduling strategies. However, our heuristics provide key elements for designing effective job scheduling strategies (e.g., for doing appropriate space-sharing among jobs, for selecting which resources are used for which job, for determining task replication levels for each job).

Minimizing the makespan of a parallel application that consist of independent tasks is the objective of numerous research projects in parallel computing (see [8] for a survey). Here we address the specific challenges posed by resource selection for volatile resources. In particular, we design various resource selection heuristics to support rapid application turnaround. We evaluate these heuristics via simulations driven by traces gathered from three real desktop grid platforms. Also, we evaluate the heuristics on three other representative desktop grid configurations. To measure the performance of our heuristics, we compare the resulting makespans to the optimal, which we can compute using a fictitious scheduler that has full knowledge of the traces.

Our heuristics are based on three scheduling techniques, namely *resource prioritization*, *resource exclusion*, and *task replication*. We find that the popular first-come-first-serve (FCFS) scheduling approach, which one can view as random resource prioritization, performs poorly in most desktop grid configurations. Instead, simply prioritizing resources according to their clock rates and ignoring their historical availability turns out to be surprisingly effective. We find also that excluding resources based on a fixed threshold, that is ignoring resources with a clock rate below from value, works well on platforms with "left-heavy" clock rate distributions. We develop an adaptive heuristic that uses a prediction of the application's makespan to exclude slow resources. This heuristic outperforms simple resource exclusion based on clock rate on multi-cluster and more homogeneous grids. Finally, we develop and evaluate three task replication techniques, which can be combined with our prioritization and exclusion heuristics. The performance of the best resulting method (clock rate prioritization, adaptive resource exclusion based on makespan prediction, reactive task replication based on time-outs) is less than a factor of 1.7 away from the optimal, and significantly better than FCFS, the default scheduling strategy in current desktop grid systems.

The remainder of this paper is organized as follows. In Section 2, we define the scheduling problem and outline our approach. In Section 3, we describe our experimental methodology: simulation model, trace data, and performance metrics. In Section 4, we evaluate resource prioritization heuristics. In Section 5, we develop heuristics that exclude resources using different criteria. In Section 6, we augment our resource exclusion heuristics to use replication. Finally, in sections 8 and 9, we discuss related work, summarize our contributions, and give future research directions.

2. Scheduling Short-lived Applications on Desktop Grids

2.1. Problem Definition

We consider the problem of scheduling an application that consists of T independent, identical tasks onto a desktop grid. The desktop grid comprises N hosts that can execute application tasks. These hosts are individually owned and can only be used for running application tasks when up and when their CPU is not used by their owners, making host and CPU availability dynamic. The hosts are managed by a master, which we will call the "server", in the following way. The server holds the input data necessary for each of the T application tasks. When a host becomes available for executing an application task it sends a notification to the server. The server maintains a queue of available hosts, the "ready queue", and may choose to send a task to one of them at any given time. When a host is executing an application

task and its CPU becomes unavailable (e.g., when the owner uses the mouse or the keyboard, when the owner starts a CPU-intensive application), the task is suspended and can be resumed on the same host at a later time. When a host executing an application becomes unavailable (e.g., due to a shutdown), the application task fails and must be restarted from scratch on another host. (We do not consider checkpointing in this paper.) While an application task is running on a host, the host sends a "heartbeat" to the server every minute; in the worst case it takes 1 minute before a server determines that a task has been terminated. These assumptions are representative of real-world desktop grid infrastructures (XtremWeb [19], Entropia [18], BOINC [21]).

Given the above platform model, we consider the problem of scheduling the T tasks onto the N hosts such that the time in between the scheduling of the first task and the completion of the last task, i.e., the application's *makespan*, is minimized. In the case when $T >> N$, the scheduling problem is almost equivalent to maximizing the steady-state performance of the application (i.e., the number of tasks completed per time unit in steady-state), as the start-up and the wind-down phases of application executions are negligible versus the steady-state phase [30]. In such a situation, a simple first-come first-server (**FCFS**) strategy in which application tasks are assigned to hosts in a greedy fashion is close to being optimal. This is the strategy used by most existing desktop grid systems.

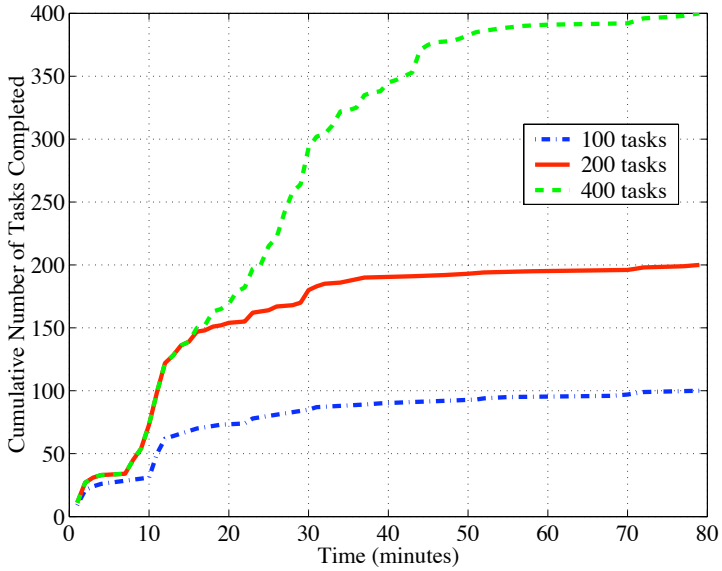

Figure 1. Cumulative task completion vs. time.

In this paper we focus on "short-lived" applications, by which we mean that T is of the same order of magnitude as N. In this case FCFS is clearly suboptimal, as seen in Figure 1. This figure plots the cumulative number of completed tasks (or cumulative throughput) throughout time observed for the FCFS strategy. These simulation results are obtained for $T = 100, 200, 400$ and $N = 190$, where each task would execute in 15 minutes on a dedicated 1.5GHz processor, and the hosts in the platforms are modeled after those in a real-world desktop grid (see Section 3 for a detailed description of our simulation methodology). In each of the three curves there is an initial hump as the system reaches steady state, after which throughput increases roughly linearly. The cumulative throughput then reaches a plateau, which accounts for an increasingly large fraction of application makespan as T decreases. For the application with $T = 100$, 90% of the tasks are completed in about 39 minutes, but the application does not finish until 79 minutes have passed, which is almost identical to the makespan for the case $T = 400$. The plateau is partly due to task failures near the end of the execution, which forces these tasks to be restarted on a new host late in the computation. The other cause for the plateau is the well-known syndrome of waiting for the slowest hosts to complete their tasks. We can see that as T gets large when compared to N the plateau becomes less significant, thus justifying the use of a FCFS strategy.

However, for smaller values of T, it is clear that some method of *resource selection* could improve the performance of short-lived applications significantly. In the next section we outline various resource selection approaches, which we evaluate and contrast in the rest of this paper.

2.2. Proposed Approaches

We consider four general resource selection approaches:

Resource Prioritization – One way to do resource selection is to sort hosts in the ready queue according to some criteria (e.g., by clock rate, by the number of cycles delivered in the past) and to assign tasks to the "best" hosts first. Such prioritization has no effect when the number of tasks left to execute is greater than the number of hosts in the ready queue. However, when there are fewer tasks to execute than ready hosts, typically at the end of application execution, prioritization is a simple way of avoiding picking the "bad" hosts.

Resource Exclusion Using a Fixed Threshold – A simple way to select resources is to excluded some hosts and never use them to run application tasks. Filtering can be based on a simple criterion, such as hosts with clock rates below some threshold. Often, the distribution of resource clock rates is so skewed [25, 53] that the slowest hosts significantly impede application completion, and so excluding them can potentially remove this bottleneck.

Resource Exclusion via Makespan Prediction – A more sophisticated resource exclusion strategy consists in removing hosts that would not complete a task, if assigned to them, before some expected application completion time. In other words, it may be possible to obtain an estimate of when the application could reasonably complete, and not use any host that would push the application execution beyond this estimate. The advantage of this method compared to blindly excluding resources with a fixed threshold is that it should not be as sensitive to the distribution of clock rates.

Task Replication – Task failures near the end of the application, and unpredictably slow hosts can cause major delays in application execution. This problem can be remedied by means of replicating tasks on multiple hosts, either to reduce the probability of task failure or to schedule the application on a faster host. This method has the drawback of wasting CPU cycles, which could be a problem if the desktop grid is to be used by more than one application.

We propose several instantiations of the above approaches and compare them in simulation. In the next section we detail our simulation methodology.

3. Experimental Methodology

We use simulation for studying resource selection on desktop grids as direct experimentation does not allow controlled, repeatable experiments. Our approach is to use simulations driven by traces that were collected from three real desktop grid platforms. In this section, we describe our trace-gathering method, trace data sets, simulation model, and performance metrics.

3.1. Availability Measurement Method and Data Sets

We gather traces by submitting measurement tasks to a desktop grid system that are perceived and executed as real tasks. These tasks perform computation and periodically write their computation rates to file. This method requires that no other desktop grid application be running, and allows us to measure exactly the compute power that a real, compute-bound application would be able to exploit.

During each measurement period, we keep the desktop grid system fully loaded with requests for our CPU-bound, fixed-time length tasks, most of which were around 10 minutes in length. The desktop grid worker running on each host ensures that these tasks do not interfere with the desktop user and that the tasks are suspended/terminated as necessary; the resource owners were unaware of our measurement activities. Each task of fixed time length consists of an infinite loop that performs a mix of integer and floating point operations. A dedicated 1.5GHz Pentium processor can perform 110.7 million such

operations per second. Every 10 seconds, a task evaluates how much work it has been able to achieve in the last 10 seconds, and writes this measurement to a file. These files are retrieved by the desktop grid system and are then assembled to construct a time series of CPU availability in terms of the number of operations that were available to the desktop grid application within every 10 second interval.

With this procedure we were able to measure two kinds of availability: (i) **host availability**, a binary value that indicates whether a host is reachable and the desktop grid software is up, which corresponds to the definition of availability in [7, 1, 6, 14, 44]; and (ii) **CPU availability**, a percentage value that quantifies the fraction of the CPU that can be exploited by a desktop grid application, which corresponds to the definition in [3, 12, 47, 15, 54]. When a host becomes unavailable (e.g., during a shutdown of the O/S), no new task can be started, and any currently executing task fails. When a CPU becomes unavailable (that is with $< X\%$ CPU availability, where X is defined by the desktop grid software) but its host is still available (e.g., when local processes use the CPU, or there is keyboard/mouse activity from the resource owner), then a running task is suspended and can be resumed when the CPU becomes available again. Host unavailability implies CPU unavailability. We call the interval of time in between the time when a host becomes available until the time of a host failures an *availability interval*.

This active but non-intrusive measurement methodology made it possible to observe CPU availability just as it would be experienced by a compute-intensive desktop grid application. As a result, our method provides more detailed information than just measuring host availability [7, 1, 6, 14, 44]. Moreover, our traces are not susceptible to OS idiosyncrasies, and can directly measure the effect of task failures (caused by mouse or keyboard activity, for example). In contrast, lightweight CPU availability or load sensing techniques [15, 17, 54, 36] are vulnerable to artifacts of the OS, and inferring task failures from traces obtained by passive sensors would be difficult. A number of interesting features of our data are reported in [29], but in this paper, we use our availability traces solely to drive simulations.

Using the previously described method, we collected data sets from two desktop grids. The first desktop grid consisted of desktop PC's at the San Diego Super Computer Center (SDSC) and ran the commercial Entropia [13] desktop grid software. We refer to the data collected from the SDSC environment as the *SDSC trace*. During a cumulative 1 month period in the last quarter of 2003, we conducted availability measurements with a deployment of Entropia DCGrid™ [18] at the San Diego Supercomputer Center (SDSC) over about 200 hosts. Their clock rates ranged from 179MHz up to 3.0GHz, with an average of 1.19GHz. Figure 2a shows the cumulative distribution function (CDF) of clock rates. The curve is not continuous as for instance no host has a clock rate between 1GHz and 1.45GHz. The curve is also skewed as for instance over 30% of the hosts have clock rates between 797MHz and 863MHz, which represents under 3.5% of the clock rate range.

The second data set was collected using the XtremWeb desktop grid software continuously over about a 1 month period (1/5/05 - 1/30/05) on a total of about 100 hosts at the University of Paris-Sud. More specifically, XtremWeb was deployed on a cluster (denoted by LRI) with a total of 40 hosts, and a classroom (denoted by DEUG) with 40 hosts respectively.

Compared to the clock distribution of hosts in the SDSC platform, the hosts in the DEUG and LRI platforms have relatively homogeneous clock rates (see Figure 2a). A large mode in the clock rate distribution for the DEUG platform occurs at 2.4GHz, which is also the median; almost 70% of the hosts have clock rates of 2.4GHz. The clock rates in the DEUG platform range from 1.6GHz to 2.8GHz. In the LRI platform, a mode in the clock rate distribution occurs at about 2GHz, which is also the median; about 65% of the hosts in this platform have clock rates at that speed. The range of clock rates is 1.5GHz to 2GHz.

We also obtained an older data set first reported in [3], which used a different measurement method. The traces were collected using a daemon that logged CPU and keyboard/mouse activity every 2 seconds over a 46-day period (2/15/94 - 3/31/94) on 85 hosts. The hosts were used by graduate students the EE/CS department at UC Berkeley. We use the largest continuously measured period between 2/28/94 and 3/13/94. The traces were post-processed to reflect the availability of the hosts for a desktop grid application using the following desktop grid settings. A host was considered available for task execution if the CPU average over the past minute was less than 5%, and there had been no keyboard/mouse activity during that time. A recruitment period of 1 minute was used, i.e., a busy host was considered

available 1 minute after the activity subsided. Task suspension was disabled; if a task had been running, it would immediately fail with the first indication of user activity.

The clock rates of hosts in the UCB platform were all identical, but of extremely slow speeds. In order to make the traces usable in our simulations experiments, we transform clock rates of the hosts to a clock rate of 1.5GHz (see Figure 2a), which is a modest and reasonable value relative to the clock rates found in the other platforms, and close to the clock rate of host in the LRI platform.

The reason the UCB data set is usable for desktop grid characterization is because the measurement method took into account the primary factors affecting CPU availability, namely both keyboard/mouse activity and CPU availability. As mentioned previously, this method of determining CPU availability may not be as accurate as our application-level method of submitting real tasks to the desktop grid system. However, given that the desktop grid settings are relatively strict (e.g., a host is considered busy if 5% of the CPU is used), we believe that the result of post-processing is most likely accurate. The one weakness of this data set is that it more than 10 years old, and host usage patterns might have changed during that time. We use this data set to show that in fact many characteristics of desktop grids have remained constant over the years. Note that, to the best of our knowledge, the UCB trace is the only previously existing data set that tracked user mouse/keyboard activity, which is why it is usable for our desktop grid simulations.

3.2. Simulated Platforms

Using the aforementioned trace data sets, we instantiate the platform model described in Section 2.1 to model the SDSC, DEUG, LRI and UCB platforms. In particular, we instantiated each platform model with about 200 hosts per desktop grid with the availability defined by the corresponding traces. SDSC was the only platform with about 200 hosts, and the remaining platform had significantly fewer hosts. So to compensate, we aggregated host traces on different days until there was approximately 200 host traces per platform. After aggregation, there were at least seven full days of traces to be used to drive simulations.

As shown in a number of studies [1, 36] including our own, hosts during weekday business hours often exhibit higher and more variable load than during off-peak hours on weekday nights and weekends. As such, all simulations were performed using traces captured during business hours which varied depending on the platform. (9AM-6PM for SDSC, 6AM-6PM for DEUG, all day for LRI, and 10AM-5PM for UCB.) We believe our heuristics would perform relatively the same during off-peak hours when host performance is more predictable, although the performance difference might be lessened.

Given the diversity of desktop configurations, we compare the performance of our heuristics on two other configurations representative of Internet and multi-cluster desktop grids. Because we do not have access to these types of desktop grids, we were unable to gather traces for these types of platforms. Nevertheless, many desktop grid projects [25, 33] publicly report the clock rates of participating hosts. So instead of using real traces, we transform the clock rates of hosts in the SDSC grid to reflect the distribution of clock rates found in a particular platform, and transform the CPU availability trace corresponding to each host accordingly.

For example, Internet desktop grids that utilize machines both in the enterprise and home settings usually have many more slow hosts than fast hosts, and so the host speed distribution is left heavy. We used host CPU statistics collected from the GIMPS Internet-wide project [25] to determine the distribution of clock rates, which ranged from 25MHz to 3.4GHz. Other projects such as Folding@home and FightAids@home show similar distributions [53].

Much work [22, 19, 41] in desktop grids has focused on using resources found in multiple labs. Recently, [33] reports the use of XtremWeb [19] at a student lab in LRI with nine 1.8GHz machines, and a Condor cluster in WISC with fifty 600MHz machines and seventy-three 900MHz machines. We use the configuration specified in that paper to model the multi-cluster scenario. We plot the cumulative clock rate distribution functions for our additional two platform scenarios in Figure 2b. For each of these distributions, we ran simulations using the SDSC desktop grid traces but transforming host clock speeds accordingly.

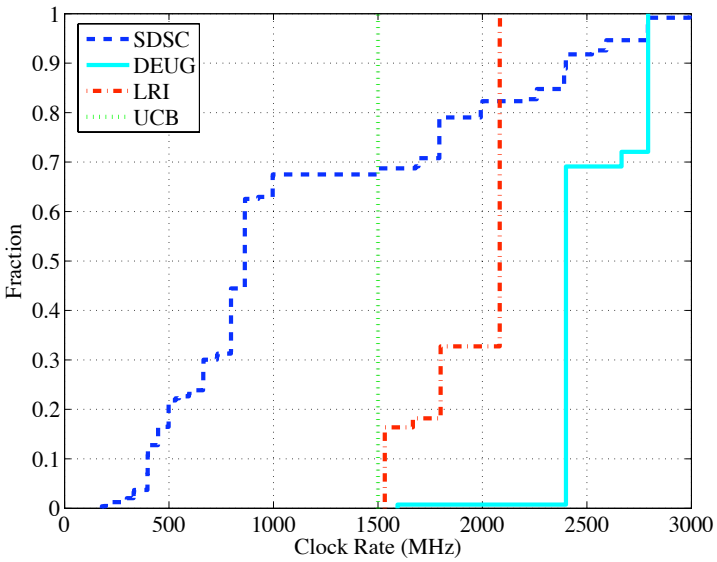

Figure 2a. Cumulative clock rate distributions from real and simulated platform.

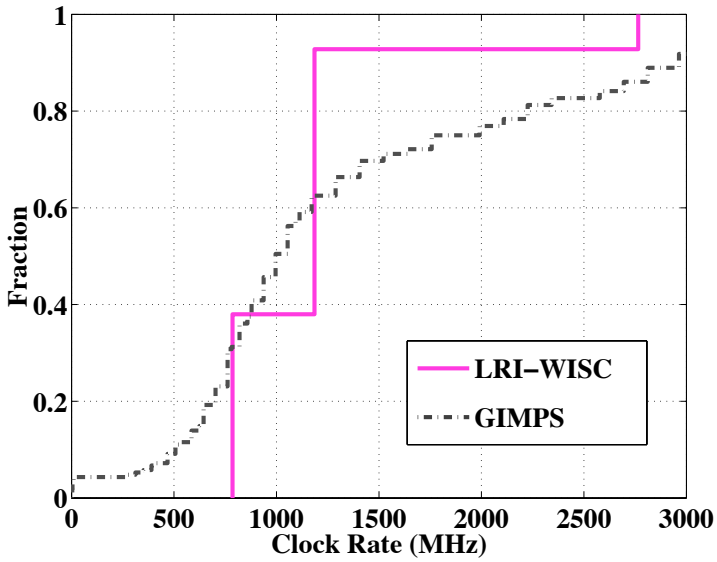

Figure 2b. Cumulative clock rate distributions from real and simulated platform.

Our justification for these clock rate transformations is that the availability interval size is independent of host clock rate as found in [27]. So for a desktop grid with a similar user base as SDSC (i.e., researchers, administrative assistants that work from 9AM-5PM), we expect that hosts will have similar availability interval lengths, regardless of CPU speed.

In summary, Table I shows the type of platforms on which we evaluate each of our heuristics. As clock rates and host volatility are the primary sources of poor application performance, we explore real and hypothetical platforms with a wide range of clock rates and volatility levels that are representative of real desktop grid systems. In particular, we examine all the cases where a platform has medium volatility and a wide range of clock rates, and a low range of clock rates but high volatility.

Table I. Qualitative platform descriptions.

Platform	Range of Clock Rates	Volatility
SDSC	high	med
DEUG	low	med
LRI	low	low
UCB	low	high
GIMPS	very high	med
LRI-WISC	bimodal	med

3.3. Simulated Applications

We simulate applications that vary in both the number and size of their tasks. Applications have 100, 200, or 400 independent tasks (which is roughly half, the same, and twice the number of hosts in our desktop grid, respectively) for reasons discussed in Section 2.1. We experimented with tasks that would exhibit 5, 15, and 35 minutes of execution time on a dedicated 1.5GHz host. Each of these task sizes has a corresponding failure rate when scheduled on the set of resources during business hours. Previously, in [29], we determined the failure rate of a task given its size using random incidence over the entire trace period. That is, in the collected traces, we chose many thousands of random points to start the execution of a task and noted whether the task would run to completion or would meet a host failure. Task failure rate increases linearly with task size from a minimum of 6.33% for a 5 minute task to a maximum of 22% for a 35 minute task. A maximum task size of 35 minutes was chosen so that a significant number of applications could complete when scheduled during the business hours of a single weekday.

For each experiment (i.e., for a particular number of tasks, and a task size), we simulated all our competing scheduling strategies for applications starting at different times during business hours. We ran each experiment for over one-hundred such starting times to obtain statistically significant results, and in true desktop grid fashion, we executed our simulations on a compute platform running the XtremWeb desktop grid software [19].

3.4. Performance metrics and analysis method

While application makespan is a good metric to compare results achieved with different scheduling heuristics, we wish to compare it to the execution time that could be achieved by an algorithm that has full knowledge of future host availabilities. This "prescient" algorithm works as follows. First, it determines the soonest time that each host would complete a task, by looking at the future availability traces and scheduling the task as soon as the host is available. Then, it selects the host that completes the task the soonest, and it repeats this process until all tasks have been completed. This greedy algorithm results in an optimal schedule, which is easy to see intuitively but which we nevertheless proved formally in [28]. We compare the performance of our heuristics using the ratio of the makespan for a particular heuristic to the optimal makespan that is achieved by this algorithm.

To determine the cause of poor performing heuristic, we visually inspect the execution trace of a subset of all applications scheduled by the heuristic. However, analyzing the performance using visual inspection of all application execution traces is not possible due to the high number, i.e., thousands, of applications executed in simulation. So to supplement our visual analysis of application execution traces, we develop a simple automated approach for determining the causes of poor performance.

As discussed in Section 2.1, delays in task completion near the end of application execution can result in a plateau of the task completion rate and thus, poor performance; we refer to the tasks completed inordinately late during application execution as *laggers*. After visual inspection of numerous application execution traces, we hypothesize that the causes of these laggers are hosts with relatively low clock rates, and task failures that occur near the end of application execution. We confirm this hypothesis in

the following manner. First, we use an automated method to find laggers in application executions that have been scheduled by a FCFS scheduler. Then, we automatically classify the cause of each lagger to be either slow host clock rate or task failure. We find that a high percentage of the laggers ($>70\%$) are caused by either low host clock rates or task failures, thus giving strong evidence for our hypothesis. After confirming the hypothesis, we use this automated method to determine the impact of slow host clock rates and task failures on application execution when other scheduling heuristics are used. We describe the automated method in detail below.

To determine the cause of poor performing applications automatically, we mine the simulation logs, determining the number of laggers and classifying each lagger by a particular cause, i.e., low clock rate or task failure. First, we classify completed tasks as laggers by determining the interquartile range (IQR) of task completion. The IQR is defined as the range between the lower quartile (25th percentile) and upper quartile (75th percentile) of task completion times, excluding task executions that fail to complete. Then, we multiply the IQR by a certain factor (which we term *IQR factor*) and add the result to the upper quartile to give a lagger threshold. If a task is completed after the threshold, it is classified as a lagger. In particular, assuming an IQR factor F, if there exists a lagger after the threshold, then a lower bound on the task completion rate slowdown between the interquartile and the last quartile of tasks is given by $1/2F$. So laggers signify a dramatic decrease in task completion rate.

Figure 3 shows the cumulative throughput for an application with 400 tasks as execution progresses. In the figure, the first and third quartiles of task completions are labeled, showing the IQR. Using an IQR factor of 1, the figure also shows where the lagger threshold is with respect to the third quartile. The tasks that finish execution after the threshold are considered laggers.

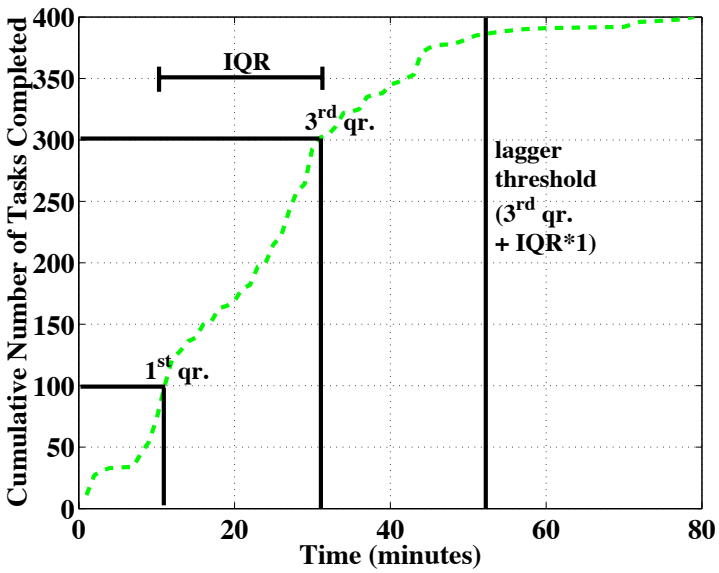

Figure 3. Laggers for an application with 400 tasks.

Our rationale to use the IQR to define the lagger threshold is that the task completion rate during the IQR is a close approximation to the optimal. This is because application execution enters steady state during the IQR as each available host is assigned a task. If $T >= N$, the task completion rate during the IQR is guaranteed to be optimal and follows trivially from our proof of the optimal scheduling algorithm discussed in [28].

An alternative is to use find the standard deviation of application makespans, and then to define a lagger threshold according to some factor of the standard deviation. However, by the very nature of the laggers, they tend to be relatively extreme outliers in terms of task completion times, and the standard deviation could be too sensitive to these outliers; an extreme lagger could cause the standard deviation to be quite high and using a lagger threshold based on the standard deviation could then result in many false

negatives. Our approach of using quantiles is less affected by these extreme laggers. Another alternative method is to classify the last $X\%$ of completed tasks in an application as laggers. However, in the case of an optimal application execution, this method would classify the last $X\%$ of tasks as laggers, and yield relatively many false positives.

One question related to our lagger analysis is how to choose a suitable IQR factor F. Clearly, an extremely low IQR factor would yield several false positives, and an extremely high IQR factor would miss most of the true laggers, limiting our analysis to an insignificant number of laggers. To determine the set of possible IQR factors to use, we conducted a simple sensitivity analysis of the number of laggers determined by the IQR factor. The lowest possible IQR factor is about .5, since the steady state and maximum task completion rate usually occurs during the IQR. We found that the maximum IQR factor was about 1.5; for IQR factors greater than 1.5, there were near-zero laggers for all the application. Within the range of IQR factors of .5 and 1.5, we found that the number of laggers decreases only gradually as the factor is increased. We choose an intermediate IQR factor of 1 and found that using values of .5 and 1.5 do not significantly change the distribution of laggers.

After discovering the set of laggers for each application, we determine the cause for each lagger as follows. To determine if the cause is a task failure near the end of application completion, we look at all tasks completed after the 75^{th} percentile; if the task fails after that point, we conclude that failure is at least one cause of the lagger. To determine if the cause is a slow clock rate, we use the clock rate of the slowest host used in the corresponding optimal application execution. That is, for an application that begins execution at a particular time, we compute the optimal schedule and find the slowest host used in that execution. The clock rate of the slowest host is then used to classify whether a lagger was assigned to a slow host or not. The advantage of using this method of classification is that it actually confirms that a faster host could have been used in the application execution. Another advantage is that this method determines a clock rate threshold for each application execution instance. So, if only relatively fast hosts are used in the optimal application schedule, the resulting clock rate threshold that we use to determine whether a lagger was assigned to a slow host will also be high.

When comparing our heuristics we do not consider the number of laggers as we found weak correlation (with correlation coefficient usually under .22) between the number laggers and makespan for the FCFS scheduling method when a relatively low IQR factor of .5 is used. (By using a low IQR factor of .5 we ensure that all laggers are counted.) The weak correlation is cause by the application waiting for the completion of a task scheduled on an extremely slow host when the rest of the tasks have already been completed (as shown for the application with 100 tasks in Figure 1). In this case, there is only one lagger but its effect on application makespan is dramatic.

Another reason not to consider the number of laggers is the weak correlation between the mean application makespan and mean number of laggers across the set of scheduling heuristics; that is, a heuristic that results in a lower mean makespan could in fact have more laggers than a different heuristic that results in a higher mean makespan. The reason for this is that the IQR is relative to the total makespan of the application; as the mean makespan for a particular heuristic decreases, the IQR itself decreases, which in turn lowers the threshold defined by the 75^{th} quantile, the IQR, and the IQR factor F. A lower threshold could raise the chance that a host will complete a task after that threshold, since the clock rate distribution and CPU availability of the hosts in the platform remains fixed. Thus, to supplement our lagger analysis, we also show the absolute measure of length of times intervals delimited by the the first, second, third, and fourth quartiles of task completion times. While we could have used an absolute IQR for all heuristics (for example, the IQR resulting from the FCFS scheduling method), the IQR for FCFS can be much higher (as much as three times higher, although in general the IQR's tend to be similar) than the IQR's for the other heuristics, and this could result in several false negatives. Instead, we define a relative IQR for each heuristic.

4. Resource Prioritization

We examine three methods for resource prioritization using different levels of information about the hosts, from virtually no information to historical statistics derived from our traces for each host, and we evaluate each method using trace-driven simulation first on the SDSC grid, which contains volatile hosts that exhibit a wide range of clock rates. We also report the results of heuristics run on the other platforms when applicable and interesting.

For the **PRI-CR** method, hosts in the server's ready queue are prioritized by their clock rates. Clock rates are correlated to a number of host characteristics such as probability of task completion and the number of operations availability per availability interval. Thus, ordering hosts by their clock rates may ensure that a task is sent to the "best" host currently available for task execution.

Similarly to PRI-CR, **PRI-CR-WAIT** sorts hosts by clock rates, but the scheduler waits for a fixed period of 10 minutes before assigning tasks to hosts. The rationale is that collecting a pool of ready hosts before making task assignments can improve host selection if, for example, the scheduler waits until a "good" host appears in the resource pool. The scheduler stops waiting if the ratio of ready hosts to tasks is above some threshold. A threshold ratio of 10 to 1 was used in all our experiments. We experimented with other values for the fixed waiting period and the above ratio, but obtained similar results.

The method **PRI-HISTORY** uses a history of a host's past performance to predict its future performance. Specifically, for each host, the scheduler calculates the expected operations per availability interval (that is how many operations can be executed in between two host failures) using the previous weekday's trace. This value is used to determine in which of two priority queues a host is placed, as follows. If the expected number of operations per intervals is greater than or equal to the number of operations of an application task, then the host is placed in the higher of two priority queues. Otherwise, the request is put in the low priority queue. Within each queue, the hosts are prioritized according to the expected operations per interval divided by expected operations per second; this way, hosts in each queue are prioritized according to their speed. Intuitively, the higher priority queue contains those hosts that are expected to be able to complete a task given their past history of availability. The lower priority queue lists those hosts whose availability intervals tend to be too short for task completion. The scheduler picks hosts in the lower priority queue only if the higher priority queue is empty.

Figure 4 shows the average makespan of these three algorithms and of the FCFS strategy normalized to the mean optimal execution time (represented by the horizontal bold black dotted line) for applications with 100, 200, and 400 tasks. Recall that these averages are obtained for over one-hundred distinct experiments. The general trend is that the larger the number of tasks in the application the closer the achieved makespans are to the optimal, which is expected since for larger number of tasks resource selection is not as critical to performance and a greedy method approaches the optimal one. In Figure 4, we also see that PRI-CR has considerably better performance than FCFS for applications with 100 tasks. Since the number of tasks is less than the number of available hosts, the slowest hosts are guaranteed to be excluded from the computation, whereas FCFS might have used some of these slow hosts.

PRI-CR-WAIT performs poorly for small 5 minutes tasks and improves thereafter, but never surpasses PRI-CR. The initial waiting period of 10 minutes is costly for the 100 task / 5 minute application, which takes about 6 minutes to complete in the optimal case. As the task size increases (along with application execution time), the penalty incurred by waiting for client requests is lessened, but since most hosts are already in the request queue when the application is first submitted, the PRI-CR-WAIT performs almost identically to PRI-CR and is no better. Figure 7 provides additional insights as to why PRI-CR-WAIT is largely ineffectual. This figure shows the number of available hosts and the number of tasks that are yet to be scheduled throughout time for a typical execution. Initially, there are about 150 hosts available and 400 tasks to execute, and this immediately drops to 0 hosts and about 250 tasks as each available host gets assigned a task. One can see that it is usually the case that either there are far more tasks to schedule than ready hosts or far more ready hosts than tasks to schedule. In the former scenario, PRI-CR-WAIT performs exactly as PRI-CR. In the latter case, waiting does not give the algorithm more choice in selecting resources.

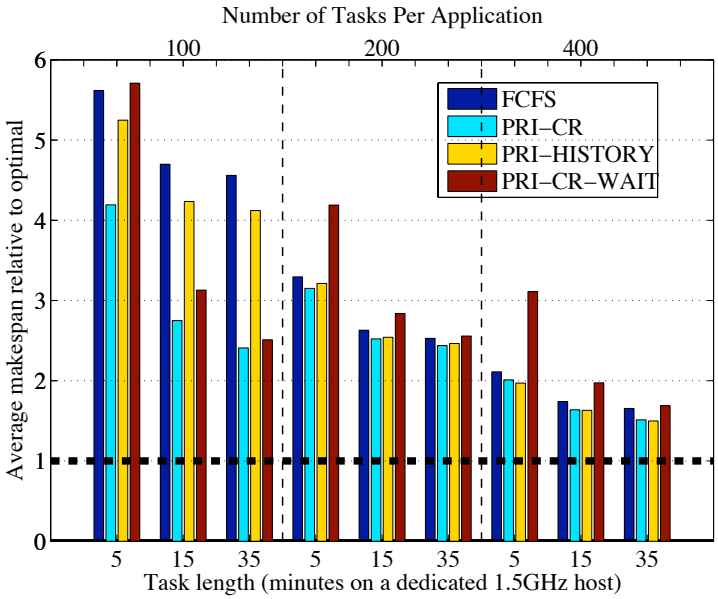

Figure 4. Performance of resource prioritization heuristics on the SDSC grid.

Surprisingly, PRI-HISTORY performs poorly compared to PRI-CR, which uses static instead of dynamic information. We found that the availability interval size, both in terms of time and in terms of operations, was not stationary across weekdays. Therefore, the expected operations per second is a poor predictor of performance for certain hosts. We determined the per host prediction error from one day to the next as follows. For each host we calculated the mean number of operations per interval on a given weekday during business hours. We then took the absolute value of the difference between a host's mean on one particular day and the next. In Figure 6, we show the complementary cumulative distribution function of prediction error of the expected time per interval for each host. That is, the figure plots the fraction of prediction errors greater than some length of time. We can see that 80% of the predictions errors are 50 minutes in length or more. On average, the mean prediction error is 109 minutes in length and the is median error is 122 minutes. Given that many applications are less than an hour in length, the high prediction error is problematic.

Moreover, in Figure 5, we show the complementary CDF of prediction error of the expected operations per interval for each host. That is, the figure plots the fraction of prediction errors greater than some quantity of operations delivered per interval. We find that 80% of the prediction errors are equivalent to 40 minutes or more on a dedicated 1.5GHz host. On average, the mean prediction error is 99 minutes in length and the median error is 85 minutes. Again, the high prediction error is significant given that many applications are less than an hour in length (and since PRI-HISTORY will tend to use hosts with high expected operations per interval). The authors of [55, 17] also found that the using the host's mean performance over long durations does not reflect the dynamism of CPU availability, and thus is a poor predictor.

We also compared the prediction error of the compute rate per host estimated using the expected operations and time length per interval. Since hosts are usually completely idle [29], the rate itself was predicted correctly. So we attribute the poor performance of PRI-HISTORY to the poor operations per interval predictions, which cause hosts to be put in the wrong priority queues.

In summary, we see that although PRI-CR outperforms FCFS consistently, resource prioritization still leads to performance that is far from the optimal (by more than a factor of 4 for 100 5-minute tasks). Looking at the schedules in detail, we noticed that performance was severely limited due to the use of slow hosts. We address this issue through heuristics described in the next section.

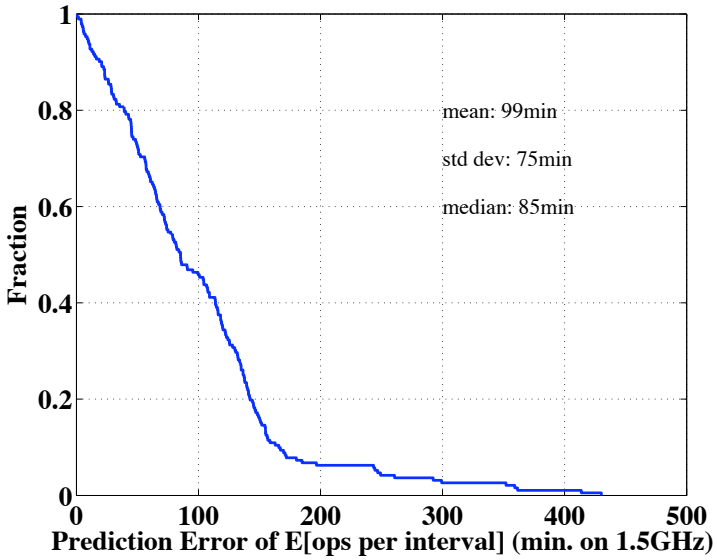

Figure 5. Complementary CDF of Prediction Error When Using Expected Operations Per Interval.

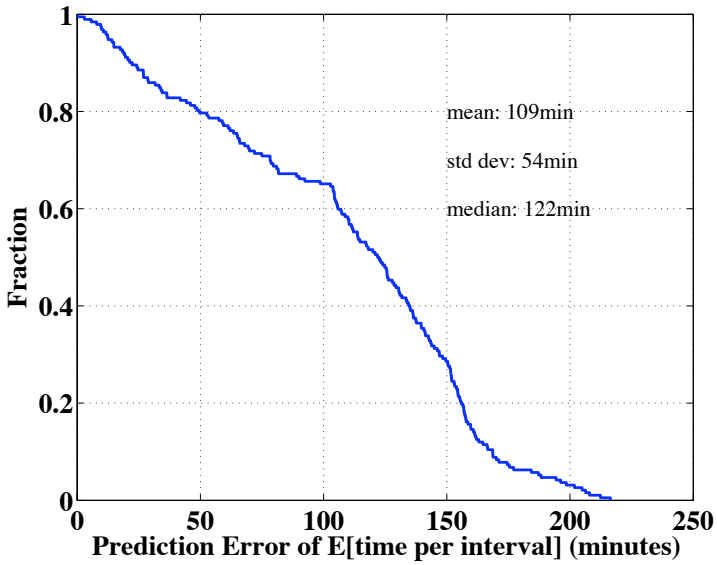

Figure 6. Complementary CDF of Prediction Error When Using Expected Time Per Interval.

5. Resource Exclusion

To prevent slower hosts from delaying application completion, we developed several heuristics that exclude hosts from the computation using a variety of criteria. These criteria only use host clock rates as we have seen that past availability is not a good predictor of future availability.

5.1. Excluding Resources By Clock Rate

Our first group of heuristics excludes hosts whose clock rates are lower than the mean clock rate over all hosts (1.2GHz for the SDSC platform) minus some factor of the standard deviation of clock rates (730MHz for the SDSC platform). The heuristics **EXCL-S1.5**, **EXCL-S1**, **EXCL-S.5**, and **EXCL-S.25** exclude those hosts that are 1.5, 1, .5, and .25 standard deviations below the mean clock rate. We can see in Figure 8 that excluding hosts 1 or .5 standard deviations below the mean can bring substantial benefits

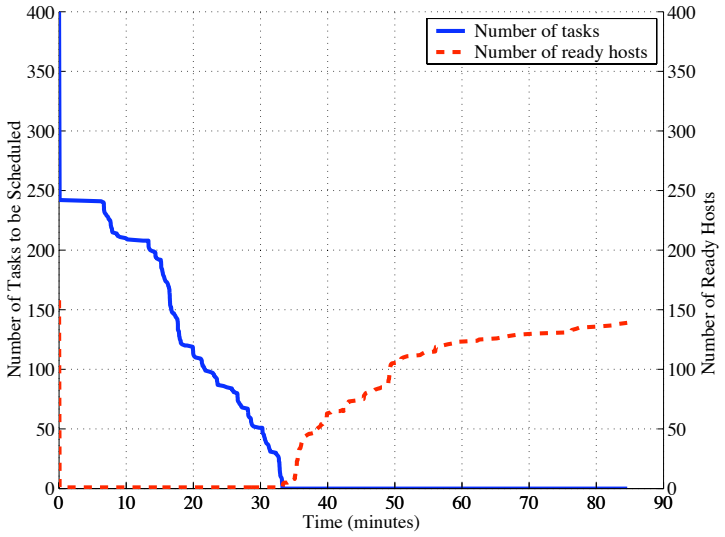

Figure 7. Number of tasks to be scheduled (left y-axis) and hosts available (right y-axis).

relative to FCFS and PRI-CR. Usually, EXCL-S.25 excludes so many hosts that it not only removes the "slow" hosts but also excludes the useful ones; the exception is the application with 100 tasks, which is equal to roughly half the number of hosts and so excluding those hosts with speeds 25% below the mean will leave slightly more than half of the hosts and thus filtering in this case does not hurt performance. EXCL-S1.5 excludes too few hosts, and the remaining "slow" hosts hurt the application makespan.

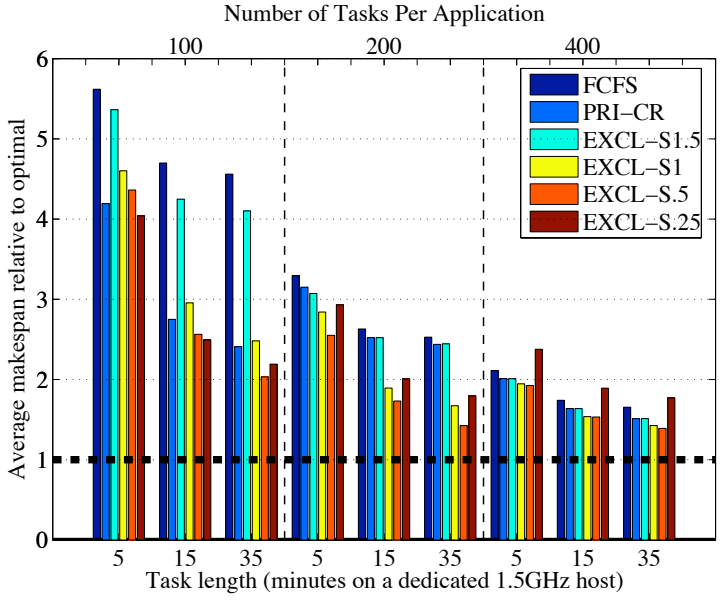

Figure 8. Performance of heuristics using fixed thresholds on SDSC grid

For the SDSC platform, EXCL-S.5 has the particular threshold that yields the best performance; on average, EXCL-S.5 performs 8%, 30%, and 6% better than PRI-CR for applications with 100, 200, and 400 tasks respectively. However, it may be that useful hosts are excluded when they should not be, and that the .5 threshold is not appropriate for different clock rate distributions of hosts in the desktop grid. In the next section, we propose strategies that use a sophisticated makespan prediction as a way to filter hosts, and evaluate it and compare it to EXCL-S.5 for different desktop grid configurations.

5.2. USING MAKESPAN PREDICTIONS

To avoid the pitfalls of using a fixed threshold, such as a particular clock rate 50% of the standard deviation below the mean in the case of EXCL-S.5, we develop a heuristic where the scheduler uses more sensitive criteria for eliminating hosts. Specifically, the heuristic predicts the application's makespan, and then excludes those resources that cannot complete a task by the predicted completion time. Our rationale is that the definition of a "slow" host should vary with the application size (or number of tasks to be completed during runtime), instead of the distribution of clock rates. That is, large applications with many tasks relative to the number of hosts should use most of the hosts as long as they do not delay application completion, whereas small applications with fewer tasks than hosts should use only a small subset of hosts that can complete a task by the application's projected makespan.

To predict the makespan, we compute the average operations completed per second for each host taking into account host load and availability using the traces and then computing the average over all hosts (call this average r). If N is the number of hosts in the desktop grid, we assume a platform with N hosts of speed r, and then estimate the makespan for the entire application with T tasks of size s in operations as $\mu = \lceil T/N \rceil (s/r)$. The rationale behind this prediction method is that the optimal schedule will never encounter task failures. So host unavailability and CPU speed are the two main factors influencing application execution time, and these factors are accounted for by r. In addition, we account for the granularity at which tasks can be completed with $\lceil T/N \rceil$.

To assess the quality of our predictor μ, we compared the optimal execution time with the predicted time for tasks 5, 15, and 35 minutes in size and applications with 100, 200, and 400 tasks. The average error over 1,400 experiments is 7.0% with a maximum of 10%. The satisfactory accuracy of the prediction can be explained by the fact that the total computational power of the grid remains relatively constant, although the individual resources may have availability intervals of unpredictable lengths. To show this, we computed the number of operations delivered during weekday business hours in 5 minute increments, aggregated over all hosts. We found that the coefficient of variation of the operations available per 5 minute interval was 13%. This relatively low variation in aggregate computational power makes a reasonable prediction of μ possible.

The heuristic **EXCL-PRED** uses the makespan prediction, and also adaptively changes the prediction as application execution progresses. In particular, the heuristic starts off with a makespan computed with μ, and then after every N tasks are completed, it recomputes the projected makespan. We choose to recompute the prediction after N tasks are completed for the following reasons. On one extreme, a static prediction computed only once in the beginning is prone to errors due to resource availability variations. At the other extreme, recomputing the prediction every second would not be beneficial since it would create a moving target and slide the prediction back (until a factor of N tasks are completed).

If the application is near completion and the predicted completion time is too early, then there is a risk that almost all hosts get excluded. So, if there are still tasks remaining at time $\theta - .95 * meanops$, where θ is the predicted application completion time and $meanops$ is the mean clock rate over all hosts, the EXCL-PRED heuristic reverts to PRI-CR at that time. This ensures that EXCL-PRED switches to PRI-CR when it is clear that most hosts will not complete a task by the predicted completion time. Note that waiteing until time θ (versus $\theta - .95 * meanops$) before switching to PRI-CR can result in poor resource utilization as most hosts are available and excluded by time θ, as seen in some of our simulations. Therefore, waiting until time θ before making task assignments via PRI-CR would cause most hosts to sit needlessly idle.

5.2.1. *Evaluation on Different Desktop Grids*

We tested and evaluated our heuristics in simulation on all the desktop grid platforms described in Section 5.2.1. We focus our discussion here on the platforms on which we found remarkable results, namely the SDSC, GIMPS, and LRI-WISC platforms. In particular, since all of our heuristics use only clock rate information for resource selection or exclusion, the heuristics executed on platforms that contained hosts with relatively similar clock rates usually produced similar results (for the DEUG and LRI platforms, with the exception of the UCB platform).

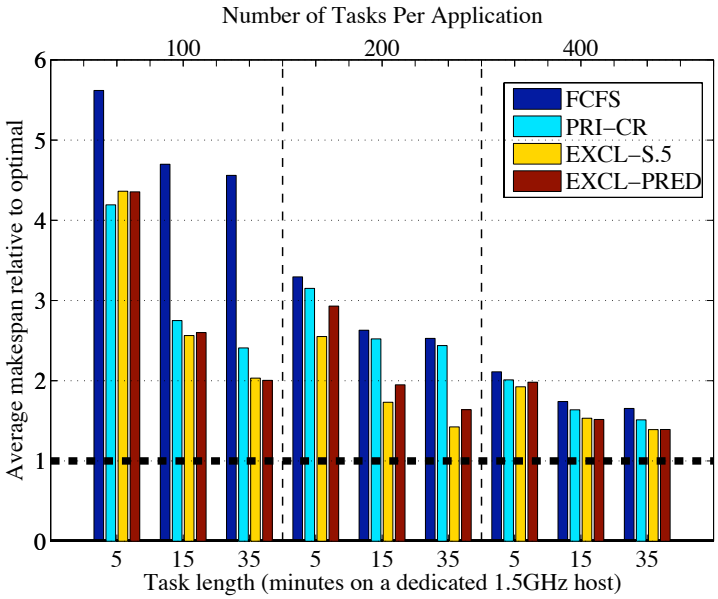

Figure 9. Heuristic performance with makespan predictor on the SDSC grid

Figure 9 shows that on the SDSC grid PRI-CR performs nearly as well as EXCL-PRED or EXCL-S.5 for applications with 100 or 400 tasks, but performs more than 23% worse than EXCL-S.5 for applications with 200 tasks. The performance of PRI-CR depends greatly on number of tasks in the application and on whether this number causes the slow hosts to be excluded from the computation. For the application with 100 tasks, the slow hosts get excluded and so PRI-CR does relatively well, as shown by the results of our lagger analysis. However, for the application with 200 tasks, PRI-CR assigns tasks to slow hosts, which then impede application completion. For the application with 400 tasks, there are enough tasks such that most hosts are kept busy with computation while the slow tasks complete.

In contrast to PRI-CR, the exclusion heuristics perform relatively well for all application sizes. Figure 9 shows that EXCL-PRED usually performs as well as EXCL-S.5 on the machines at SDSC, but there is no clear advantage for using EXCL-PRED; for the particular distribution of clock rates in the SDSC desktop grid, EXCL-S.5 appears to have the particular threshold that yields the best performance. Of all the heuristics, EXCL-S.5 eliminates the highest percentage of laggers caused by slow hosts; the reduction in the percent of laggers caused by slow hosts is as high as $\sim$60%. EXCL-PRED has slightly more laggers caused by slow hosts than EXCL-S.5 as it is less aggressive in excluding hosts. Consequently, EXCL-PRED performs 13% more poorly than EXCL-S.5 for the application with two-hundred 15-minute tasks. We have found after close inspection of our traces and the laggers that this is because of a handful relatively slow hosts that finish execution past the projected makespan and/or task failures on these slow hosts occurring near the end of the application. For the application with 400 tasks, the delay is hidden as there are enough tasks to keep other hosts busy until the slow hosts can finish task execution. For the application with 100 tasks, the relatively slow and unstable hosts get filtered out as there are fewer tasks than hosts and the heuristic prioritizes resources by clock rate.

Using the same reasoning for the SDSC platform, we can explain why EXCL-S.5 outperforms EXCL-PRED for the GIMPS desktop grid (see Figure 10), which like the SDSC grid has a left heavy distribution of resource clock rates. On the GIMPS resources, applications scheduled with FCFS or PRI-CR often cannot finish during the weekday business hours period, i.e., have application completion times greater than 8 hours, because of the use of the extremely slow resources. So, slow hosts especially in Internet desktop grids having a left-heavy distribution of clock rates are detrimental to the performance of both FCFS and PRI-CR.

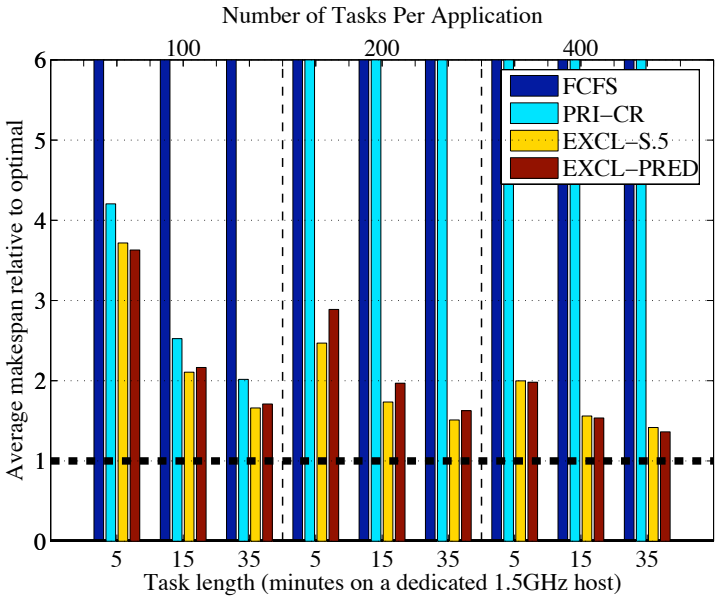

Figure 10. Heuristic performance with makespan predictor on the GIMPS grid

Although EXCL-S.5 performs the best for the SDSC and GIMPS desktop grids, the threshold used by EXCL-S.5 is inadequate for different desktop grid platforms, and the filtering criteria and adaptiveness of EXCL-PRED is advantageous in the other scenarios. In particular, EXCL-PRED either performs the same as or outperforms EXCL-S.5 for the multi-cluster LRI-WISC platform. For the application with 400 tasks (see Figure 11), EXCL-PRED outperforms EXCL-S.5 in the case of the LRI-WISC by 17%. EXCL-S.5 in the LRI-WISC desktop grid excludes all 600MHz hosts, which contribute significantly to the platform's overall computing power. In general, the longer the steady state phase of the application, the better EXCL-PRED performs with respect to EXCL-S.5, since EXCL-S.5 excludes useful resources some of which are utilized by EXCL-PRED. This explains why EXCL-PRED performs better than EXCL-S.5 for applications with more tasks and larger task sizes. While PRI-CR does as well as EXCL-PRED in these experiments, we saw that PRI-CR was not as effective on other platforms, especially those with a left heavy distribution of clock rates.

In conclusion, using a makespan prediction can prevent unnecessary exclusion of useful resources. However, this method is sometimes too conservative in the elimination of hosts, especially for shorter applications. Although this is disappointing, we find that the makespan predictor is useful and advantageous for task replication heuristics, which we discuss in the following section.

6. Task Replication

In the previous sections, we proposed several heuristics that differ by the way in which a host is selected when scheduling a task. However, even if the best resource selection method is used, performance degradation due to task failures is still possible. In this section, we augment the resource selection and exclusion heuristics described previously to use task replication techniques for dealing with failures.

We define task replication as the assignment of multiple *task instances* of a particular *task* to a set of hosts; a *task* is the application's unit of work and a *task instance* is the corresponding application executable and data inputs to be assigned to a host. We refer to the first task instance created as the *original* and the replicated task instances as *replicas*. By assigning multiple task instances to hosts, the probability of all tasks failing can be reduced. Also, replication can be a means of adapting to dynamic host arrivals (as most desktop grid systems do not support process migration); for example, in the case

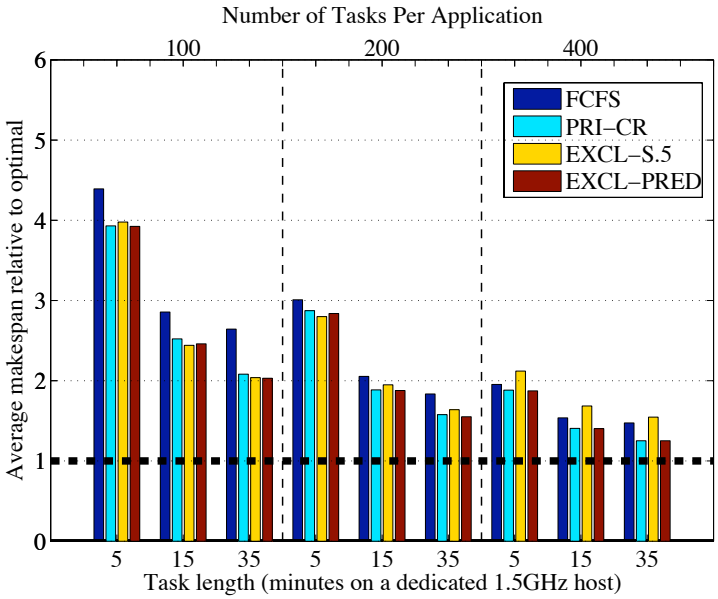

Figure 11. Heuristic performance with makespan predictor on the LRI-WISC grid

where a task has been assigned to a relatively slow host but a fast host arrives shortly thereafter, a task can be replicated on the fast host (as opposed to migrated) to accelerate task completion.

In addition to using makespan relative to optimal as the performance metric, we use *waste*, which is the percent of tasks replicated (including those that fail), to quantify the expense of wasting CPU cycles. A replication heuristic that has high waste would be problematic if the entire desktop grid is loaded and multiple applications are competing for resources. (Note that the reason we did not consider the heuristics with replication in the earlier heuristics is that replication is not always an option when there is high resource contention among multiple applications in the system.)

We investigate three approaches for task replication, namely *proactive*, *reactive*, and *hybrid* methods. All of our heuristics only replicate when there are more hosts than tasks. Applications that have a number of tasks larger than the number of hosts will often have a steady-state phase, and replicating during this steady state phase will usually not improve makespan, and only delay task completion. With proactive replication, multiple instances of each task are created initially and assigned as hosts become available. Proactive replication techniques are aggressive in the sense that replication is done before a delay in application completion time has occurred. A heuristic, which we call **EXCL-PRED-DUP**, uses EXCL-PRED but replicates each task whenever the number of ready hosts is greater than the number of tasks to schedule.

In contrast, a heuristic using reactive replication will replicate a task only when the task's completion has been delayed and its execution is delaying application completion; in this sense, the heuristic are reactive. **EXCL-PRED-TO** is based on EXCL-PRED and uses a timeout for each task to determine when replication should occur. That is, whenever a task is scheduled, a timeout occurs if the task has not been completed by the predicted makespan. Upon timeout, the task is replicated.

Finally, we develop a heuristic **REP-PROB** that uses a hybrid approach for replicating tasks that either have a high risk of delaying application completion or are currently delaying completion; as such, the heuristic uses both proactive and reactive replication techniques. REP-PROB is more sophisticated than EXCL-PRED-DUP and EXCL-PRED-TO in that it uses the probability of task completion on the previous day to predict the probability of task completion on the following day. Using these predicted probabilities, the heuristic replicates tasks until the predicted probabilities of task completion are below some threshold. We describe the heuristic in detail below.

The **REP-PROB** heuristic uses the history of host availability to make informed decisions regarding replication. Specifically, the heuristic prioritizes each host according to its predicted probability of completing a task by the projected application completion time. We use random incidence within the previous day's host traces to determine the predicted probability of task completion. The projected application completion time is determined using the same makespan predictor as EXCL-PRED described in Section 5.2.

In addition, REP-PROB prioritizes each task according to its probability of completion by the predicated makespan given the set of hosts it has been assigned to; the task with the lowest probability of completion is replicated on the host with the highest probability. Regarding how many task instances to create, the heuristic could create a single replica as in the EXCL-PRED-DUP heuristic. But if the two task instances were both scheduled on slow and unreliable hosts, then the probability of task completion would remain low and the task would require more replicas. Instead, REP-PROB uses the probability of completion to estimate how many task replicas to create in order to ensure the probability of task completion is greater than some threshold.

To understand the probabilistic model, it is beneficial to review a simple deterministic finite automata (DFA) of task execution (see Figure 12). Note that the concept of availability used in the figure refers to availability of the host with respect to the task execution. First, the task begins execution (state 1). If the host fails before the task can complete, the task fails (state 2), and we must wait until the host becomes available again before beginning task execution again (state 1). If the host is available long enough for the task complete, task completes (state 3).

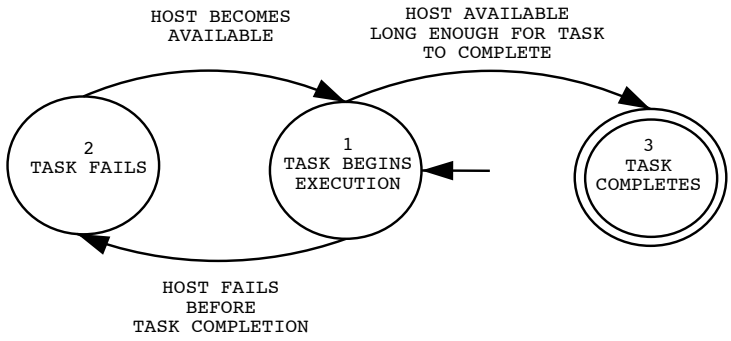

Figure 12. Finite automata for task execution.

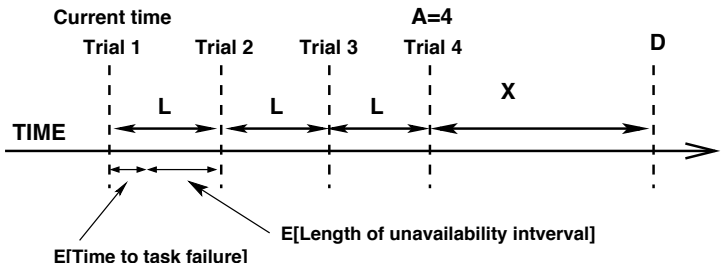

Figure 13. Timeline of task completion.

With this model, one can use a geometric distribution to model the probability that a task completes after a certain number of attempts. By using a geometric distribution, we assume that each attempt to complete a task instance on some host is independent of other attempts on the same host. In particular, the probability of task completion can be computed using the following parameters:

- $H_1, H_2, ..., H_N$: the set of N heterogeneous and volatile hosts.
- $W_1, W_2, ..., W_T$: the set of T tasks of an application to be scheduled.
- C_i: completion time of *task* W_i.
- $c_{i,j}$: completion of time of *task instance* j of task W_i.
- r_i: number of instances of task W_i.
- D: the desired completion time of the application, as determined by our makespan predictor, for example.
- X: execution time of a task on a particular host.
- A: starting from the current time, the number of attempts, i.e., trials, possible on a particular host to complete a task by time D,
- L: the length of time for each failed attempt on a particular host
- p: the probability of task completion (as computed by random incidence) for a particular host.

The parameters $X, A, L,$ and p are all defined for a particular host H_m (and should be written as $X_{H_m}, A_{H_m}, L_{H_m}$, and p_{H_m}, respectively), but for brevity we omit the subscripts in our discussion below.

If a task is to complete by time D when executed on a particular host H_m, the last attempt to complete a task must occur by time $D - X$. Thus, the number of attempts A for task completion is given by $\lfloor (D - X)/L \rfloor + 1$, where L is the time required for each failed attempt (see Figure 13). In the DFA in Figure 12, L is the time the task had been executing before failure just before entering state 2 from state 1 plus the time before the host becomes available again, i.e., the length of the unavailability interval, incurred when going from state 2 back to state 1. Ideally, L would be modeled by the probability distribution of the task's time to failure and the length of unavailability intervals. However, constructing such a joint probability distribution is difficult as using only a day's worth of historical data results in a very sparse probability distribution over multiple dimensions. So, as a simplification, we calculate L using the expected time to task failure (which we can compute with random incidence) plus the expected length of unavailability for a particular host (which we can derive from the traces). Then, the probability that a task instance j of task W_i completes by time D can be estimated by:

$$P(c_{i,j} \leq D) = \sum_{a=1}^{A} (1-p)^{a-1} p, \tag{1}$$

which sums the probability that the task completes in the a^{th} attempt.

Assuming that availability is independent among hosts (in [27], we give evidence that availability is independent among hosts on certain platforms), the probability that a particular task W_i completes by time D is estimated by:

$$\begin{aligned} P(C_i \leq D) &= P(min_j(c_{i,j}) \leq D) && (2)\\ &= 1 - P(min_j(c_{i,j}) > D) && (3)\\ &= 1 - P(c_{i,1} > D)P(c_{i,2} > D)... && (4)\\ &\quad P(c_{i,j} > D) \text{ where } 1 \leq j \leq r_i && (5) \end{aligned}$$

Then, the probability that the application completes in time D can be estimated by:

$$P(max_i(C_i) \leq D) = P(C_1 \leq D)P(C_2 \leq D)... \quad P(C_T \leq D) \tag{6}$$

So using the probability of completion for each host and desired completion time, we can determine the amount of replication needed to achieve some minimum probability threshold. Clearly, at any particular time during application execution, there may not be enough hosts to replicate on in order to achieve the threshold. The heuristic REP-PROB makes the best effort by replicating the task with lowest probability of completion on the host with the highest, until there are no remaining hosts left; if a task has no instances assigned, it is given the highest task priority to ensure that an instance of each task is assigned before replicating.

While Equation 6 can be used to estimate the probability of application completion in theory, in practice it is almost impossible to achieve given the high amount of replication and number of hosts required. This can be shown by a simple back of the envelope calculation to determine the number of instances per task required to achieve some probability bound. That is, assume our application consists of 100 tasks to be scheduled on the SDSC grid, and that our desired probability of application completion $P(max_i(C_i) \leq D)$ is 80%. Achieving this threshold requires that each task is completed with probability $P(C_i \leq D) = e^{ln(.8)/100}$, assuming that each task is completed with equal probability. If a task instance fails with probability 20% (a realistic number as shown in [29]), it would require four task instances for each task, totaling 400 task instances for the application with 100 tasks. Since there are only $\sim$200 hosts in the SDSC platform, computing all task instances at once is not possible for even a relatively small application. Furthermore, waste of 300% is extremely high and would reduce the effective system throughput considerably. We confirmed these conclusions in simulation for a range of application sizes (100, 200, 400 tasks) and task sizes (5, 15, 35 minutes on a dedicated 1.5GHz host); each task is replicated so often that the application rarely completes by the predicated makespan. So instead of trying to achieve a probability threshold *per application*, REP-PROB makes the best effort to achieve a probability threshold *per task* using Equation 5.

A procedural outline of the REP-PROB heuristic is given below:

1. Predict the application completion time D using the makespan predictor.
2. Prioritize tasks according to the probability of task completion by time D estimated by Equation 5. Unassigned tasks have the highest priority. Tasks that have timed out have the second highest priority.
3. Prioritize hosts according to the probability of completing a task by time D.
4. While there are tasks remaining in the queue:
   a) Assign an instance of the task with the lowest probability of completion to the host with the highest probability.
   b) Assign a timeout D to that task. If the task has not been completed by time D, the task will be given the second highest possible priority (corresponding to "timed-out" tasks).
   c) Recompute the task's probability of completion.
   d) Remove the task from the queue if its probability of completion is above 80% (We tested a range of thresholds from 50-90% and found that a threshold of 80% is the most adequate in terms of improving application performance.

We hypothesize that REP-PROB should outperform EXCL-PRED-TO. REP-PROB takes into account both host clock rate *and host volatility* when deciding which task to replicate and which host to replicate on. As such, REP-PROB aggressively replicates tasks that have a low probability of completion as soon as the original task instance is assigned; this in turn reduces the chance that tasks scheduled on volatile hosts will delay application completion. In contrast, EXCL-PRED-TO only replicates a task if it has not been completed by the predicted makespan, and the replica is assigned to a host based on its clock rate (disregarding host's volatility). As such, tasks initially assigned to a volatile host will not have replicas scheduled until late during the application execution, and this may result in a delays in

application completion. Also, replicas may be assigned to volatile hosts (although the hosts may have relatively fast clock rates).

6.1. Results and Discussion

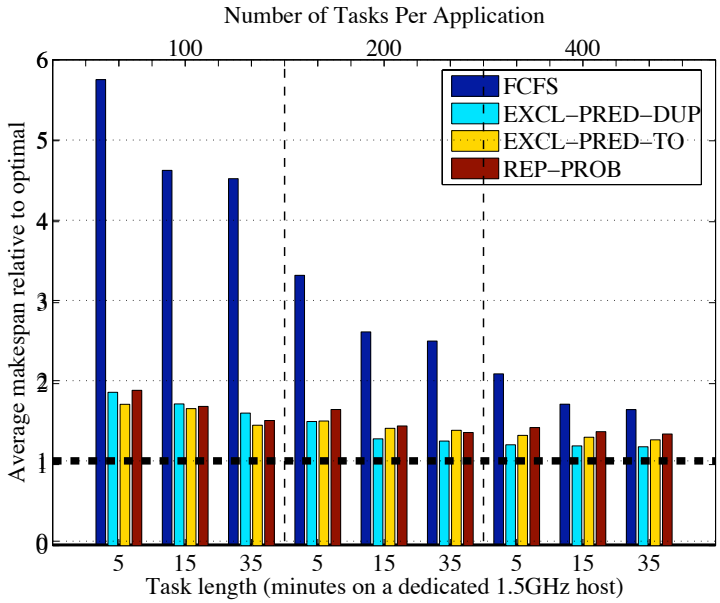

Figure 14. Performance of REP-PROB on SDSC grid.

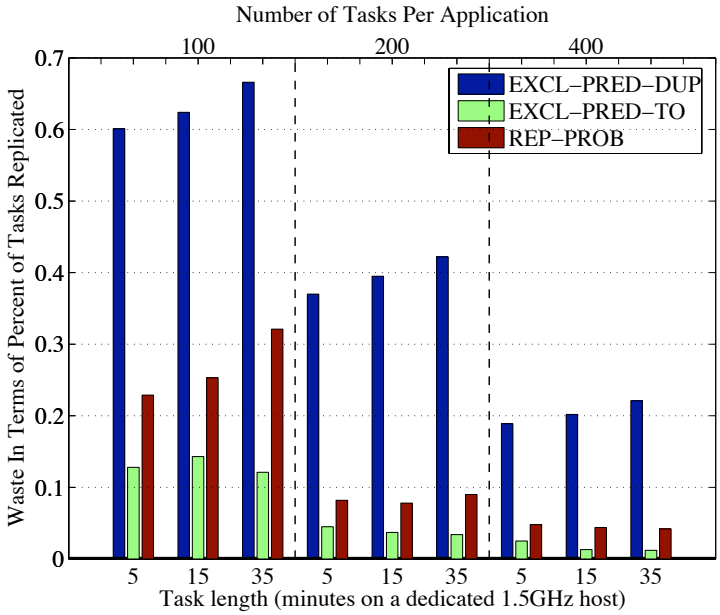

Figure 15. Waste of REP-PROB on SDSC grid.

Figure 14 shows the results for the SDSC platform for each application size. In all cases on the SDSC platform, EXCL-PRED-TO achieves the lowest waste of all the heuristics while obtaining similar performance gains. We attribute the efficiency of EXCL-PRED-TO to the makespan predictor, which forces the scheduler to wait as long as possible (without significantly delaying application execution)

Table II. Mean performance difference and waste difference between EXCL-PRED-DUP and EXCL-PRED-TO.

	Platform			
Metric	*SDSC*	*DEUG*	*LRI*	*UCB*
Makespan	+0.06%	-8.7%	-10.8%	-17.5%
Waste	+86.2%	+39%	+71.4%	+26.1%

Table III. Mean performance and waste difference between EXCL-PRED-TO and REP-PROB.

	Platform			
Metric	*SDSC*	*DEUG*	*LRI*	*UCB*
Makespan	-4%	-11.1%	+1.7%	+13.1%
Waste	-140%	-37.5%	+3.3%	-19.2%

before replicating a task. In addition to EXCL-PRED-TO, we investigated another reactive replication heuristic that used timeouts based on clock rates instead of a makespan predictor and found that EXCL-PRED-TO achieved similar performance but 65% less waste; again, we attribute the good performance of EXCL-PRED-TO to its makespan predictor. Overall, the results show that reactive replication can achieve high performance gains at relatively low resource waste.

Furthermore, Table II summarizes the mean makespan difference and mean difference of waste of EXCL-PRED-DUP and EXCL-PRED-TO on all platforms. A positive percentage means that EXCL-PRED-TO did better than EXCL-PRED-DUP. In terms of mean makespan, EXCL-PRED-TO performs about 8.7% and 10.8% worse than EXCL-PRED-DUP on the DEUG and LRI platforms, respectively. Although EXCL-PRED-TO performs slightly worse on these platforms, it is much less wasteful (on average 39% and 71.4% less wasteful on the DEUG and LRI platforms). This is partly because EXCL-PRED-TO can adjust to the volatility of the platforms. Because EXCL-PRED-TO does not replicate until a task delays execution, which is unlikely in the LRI scenario, there is much less waste with EXCL-PRED-TO than EXCL-PRED-DUP. In the opposite scenario, where the platform is volatile, EXCL-PRED-TO will replicate tasks more often. Nevertheless, we find that EXCL-PRED-TO performs worse (17.5% on average) than EXCL-PRED-DUP because tasks have a relatively high chance of failing on the UCB platform. When EXCL-PRED-TO replicates a task instance that has timed out, there is a relatively high probability that the replica itself will fail, and so the benefits of using EXCL-PRED-TO are less on the relatively volatile UCB platform compared to the other platforms. In contrast to EXCL-PRED-TO, EXCL-PRED-DUP replicates tasks immediately as soon as the original task instance is assigned to a host (versus waiting until the predicted application completion time) so there is a smaller chance that both task instances will fail and delay application completion. At the same time, the waste of EXCL-PRED-DUP is significantly hither than EXCL-PRED-TO, by 26.1% on average.

We also investigated variations of EXCL-PRED-DUP which increased the number of replicas created for each task. In general, we found that after a task is replicated once, replicating more often does not improve performance. The lack of performance improvement is partly due to the fact that replicating a task once dramatically decreases the probability of failure since there are many fast hosts available near the end of application execution. Thereafter, creating more replicas will not significantly reduce the probability of failure. Moreover, if too many replicas are created, then a large fraction of the hosts will be doing redundant work preventing useful work from being done, thus degrading performance.

Table III shows the performance of REP-PROB relative to EXCL-PRED-TO for all four platforms. A positive value in the table means that REP-PROB performed that much better than EXCL-PRED-TO. Surprisingly, REP-PROB does not perform better than EXCL-PRED-TO in the SDSC and DEUG

platforms. In the one platform where REP-PROB does perform significantly better than EXCL-PRED-TO, the performance difference on average is only 13%. The performance of EXCL-PRED-TO is similar to REP-PROB for several reasons. First, there is strong correlation between the probability of completion by a particular time and clock rate as shown in [27]. Since EXCL-PRED-TO replicates tasks on the hosts with the highest clock rates, these hosts will tend to have the highest probability of completion by the projected completion time. Second, a large fraction of the hosts in each platform are relatively stable. For example, for a fifteen minute task, the fraction of hosts with failure rates less than 20% for the SDSC, DEUG, LRI, and UCB platforms are $\sim$60%, 75%, 100%, and 50% respectively. So for the small fraction of tasks that timeout, the replica will most likely be scheduled on a relatively stable host (especially since EXCL-PRED-TO will choose the host with the fastest clock rate, which is correlated with the probability of task completion) and the resulting probability of the task will be dramatically lowered. For example, if the timed-out task has 50% chance of failure and a replica is then scheduled on a host with a 20% chance of completion, then the probability that the task will fail is a mere 10%. Third, the (un)availability of one host with respect to another can be correlated in some platforms and so the probability of task completion computed is only a lower bound. The fact that availability of hosts in the DEUG is correlated as shown in [27] may be one reason why EXCL-PRED-TO outperforms REP-PROB by about $\sim$11% on that particular platform.

Moreover, REP-PROB wastes significantly more resources (as much as 140% more than EXCL-PRED-TO) without much gain in performance (see Table III). REP-PROB naturally replicates more than EXCL-PRED-TO when the heuristic replicates tasks with low probabilities of completion. One reason that this does not result in significant performance improvement could be because of mispredictions in the probability of task completion. Although a significant fraction of predictions may be within 25% of the actual value [27], any misprediction that leaves a task assigned to a volatile host unreplicated could be costly for the application. Also, our assumption that the series of attempts to complete a task instance on a particular host are independent may not be valid; by observing our traces, we found that a short availability interval is often followed by a another short availability interval.

Nevertheless, REP-PROB does perform slightly better than EXCL-PRED on the UCB platform. Because all the hosts in the UCB platform have the same clock rates and EXCL-PRED-TO prioritizes hosts only by clock rates, EXCL-PRED-TO cannot distinguish a stable host from a volatile one. REP-PROB on the other hand will prioritize the hosts by their predicted probability of completion, and have an advantage in this case. But, the performance improvement is limited again because a large fraction of the hosts in the UCB platform are relatively stable.

6.2. Evaluating the benefits of REP-PROB

The "achilles heel" of EXCL-PRED-TO is the fact that it sorts only by clock rates, and one can certainly construct pathological cases that make EXCL-PRED-TO perform more poorly than REP-PROB. For example, one could imagine the scenario where half the hosts are extremely volatile while the other half are extremely stable but have slightly lower clock rates than the volatile hosts. In this case, EXCL-PRED-TO will tend to schedule tasks to hosts with faster (albeit only slightly faster) clock rates, which are also the most volatile; as a result, the tasks will tend to fail and delay application completion. REP-PROB on the other hand will take into account host volatility and schedule tasks to stable hosts.

To investigate this issue, we construct a new platform, half of which consists of volatile hosts from the UCB platform. The clock rates of the UCB hosts are transformed to follow a normal distribution with mean 1500MHz and standard deviation of 250MHz. The other half of the new platform consists of stable hosts from the LRI cluster, which is relatively homogeneous in terms of host clock rates. We then create a set of platforms where we transform the clock rates of hosts from the LRI platform. Clearly, if the clock rates of the stable LRI hosts are relatively very low, then it will be better to schedule tasks to the volatile UCB hosts, and EXCL-PRED-TO will perform better then REP-PROB. If the clock rates of the stable LRI hosts are higher than UCB hosts, then it will be better to schedule tasks to the stable and fast LRI hosts, and again EXCL-PRED-TO will outperform REP-PROB. However, when clock rates of

the LRI hosts are "slightly" less than the clock rates of the UCB hosts, then REP-PROB has a chance of outperforming EXCL-PRED-TO.

Specifically, we transform the clock rates of LRI hosts by -33%, -15%, -6%, +6%, +15%, and +33% relative to the mean clock rate of UCB hosts (1500MHz), and we refer to the resulting platforms as UCB-LRI-n33, UCB-LRI-n15, UCB-LRI-n06, UCB-LRI-p06, UCB-LRI-p15, and UCB-LRI-p33, respectively. Then we run both EXCL-PRED-TO and REP-PROB on each platform and determine how each heuristic performs. We observe that REP-PROB performs better by $\sim$13% or less than EXCL-PRED-TO for only a limited set of the platforms with the range of -15% and -6% relative to the mean UCB clock rate (see Figure 16).

This limited improvement in a relatively small set of hypothetical scenarios indicates that REP-PROB will rarely outperform EXCL-PRED-TO, and any performance difference is slight. Moreover, in practice on our real platforms, we find that REP-PROB performs at most 13.1% better than EXCL-PRED-TO while causing 40% more waste on average across all platforms. So, in general, we believe that EXCL-PRED-TO will usually outperform or perform as well as REP-PROB, with the possibility of performing slightly worse.

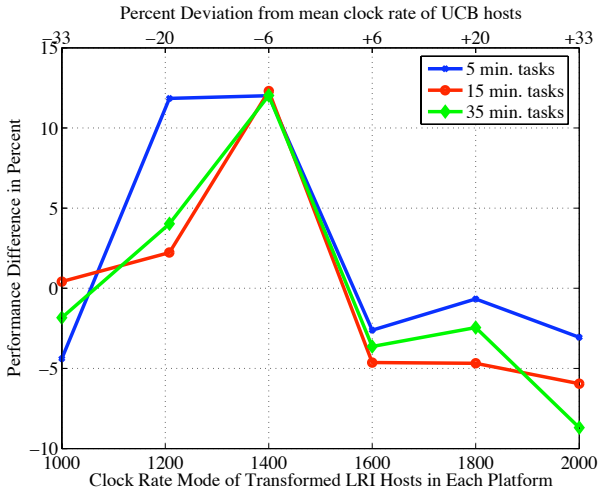

Figure 16. Performance difference between EXCL-PRED-TO and transformed UCB-LRI platforms

7. Scheduler Prototype

In this section, we describe our implementation of the best performing heuristic EXCL-PRED-TO, and show that the scheduling model we use is feasible in a real system. Our implementation of EXCL-PRED-TO is integrated with the open source XtremWeb desktop grid software [19].

7.1. Overview of the XtremWeb Scheduling System

The architecture of the XtremWeb system consists of a dispatcher, scheduler, client, and multiple workers. After an application is submitted, the dispatcher periodically selects a subset of tasks from a task pool and distributes them to a scheduler (or set of schedulers). The scheduler is then responsible for the completion of tasks. Workers make a request for work to the scheduler typically using Java RMI, although other methods of communication through SSL and TCP-UDP are supported. The default scheduler in XtremWeb schedules tasks to hosts in a FCFS fashion. Upon completion, the worker will return the result to the scheduler, which stores the result on the server's disk and records the task completion in the results database.

7.2. EXCL-PRED-TO Heuristic Design and Implementation

We replaced the FCFS scheduler in XtremWeb with our EXCL-PRED-TO scheduler. This involved a number of changes to the XtremWeb system which we describe below. In the process, we made several additions and changes to the scheduler to implement the timeout mechanisms, and describe here a few interesting details. One potential hazard of replication is that the replicas could delay original task instances from being executed. For example, suppose instances of a particular task are replicated a high number of times and then placed in a work queue. Then suppose task instances of a different task are placed in the work queue after instances of the first task. Since task instances are assigned in the order that they are placed in the work queue, the second task could "starve" as the workers are kept busy executing replicas of the first task.

To reduce the chance of task starvation, our scheduler uses a two-level work queue; we refer to the higher level queue as the *primary* queue and the lower level queue as the *secondary* queue. When an application is submitted by the client, an instance of each task is placed in the primary queue. For the EXCL-PRED-TO heuristic, a timeout is associated with each original task instance when it is scheduled on a host. When this time out expires, a task replica is placed in the secondary queue. When doing task assignment, the scheduler will first schedule tasks in the primary queue before those in the secondary queue in an effort to ensure that at least one instance of each task will always be scheduled before any replicas.

To keep the number of replicas from growing too rapidly, only original task instances are allowed to time out. Also, when the original task instance fails, a new corresponding instance is placed in the primary queue; however, if a replica fails, nothing more is done.

The task instance priority queues are implemented as fixed-sized lists within the XtremWeb scheduler, and the lists act as a buffer between the database of tasks and workers requesting task instances. Periodically, the primary priority queue is filled with original task instances instantiated from tasks in the database. A thread periodically checks the state of all the task instances in each priority queue. Given a particular state, this thread causes the appropriate action to be taken. For example, to implement the timeout mechanism, we set a timeout for each task instance in the primary work queue. Periodically, the thread checks the state of each tasks and if a task has timed out, it places a replica in the secondary queue.

8. Related Work

8.1. Resource selection

Since the emergence of grid platforms, resource selection on heterogeneous, shared, and dynamic systems has been the focus of intense investigation. However, desktop grids compared with traditional grid systems incorporating mainly a set of clusters and/or MPP's are much more heterogeneous and volatile, as quantified in [29]. Consequently, the platforms models used in typical grid scheduling research are inadequate for desktop grids. Availability models based on host or CPU availability, such as those described in [15, 34, 54] do not accurately reflect the availability of the resource as perceived by a desktop grid application. So, the scheduling heuristics designed and evaluated with these models may not be applicable to desktop grids.

For instance, the work in [16] describes a system for scheduling soft real-time tasks using statistical predictors of host load. The system presents to a user confidence intervals for the running time of a task. These confidence intervals are formed using time series analysis of historical information about host load. However, the work assumes a homogeneous environment and disregards task failures caused by user activity (as the system does not specifically target desktop grid environments). As such, the effectiveness of the system on desktop grids is questionable.

The work in [47], which studies the problem of scheduling tasks on a computational grid for the purpose of online tomography. The application consists of multiple independent tasks that must be

scheduled in quasi-real-time on a shared network of workstations and/or MPP's. To this end, the authors formalize the scheduling problem as a constrained optimization problem. The scheduling heuristics then construct a plan given the constraints of the user (e.g., requirement of feedback within a certain time) and the characteristics of the application (e.g., data input size). Although the scheduling model considers the fact that host can be loaded, the model does not consider task failures. Given the high task failure rates in desktop grid systems, the same heuristics executed desktop grids could suffer from poor performance.

The work described in [35] is the most related to ours. The author investigates the problem of scheduling multiple independent compute-bound applications that have soft-deadline constraints on the Condor desktop grid system. Each "application" in this study consists of a single task. The issue addressed in the paper is how to prioritize multiple applications having soft deadlines so that the highest number of deadlines can be met. The author uses two approaches. One approach is to schedule the application with the closest deadline first. Another approach is to determine whether the task will complete by the deadline using a history of host availability from the previous day, and then to randomly choose a task that is predicted to complete by the deadline. The author finds that a combined approach of scheduling the task that is expected to complete with the closest deadline is the best method. Although the platform model in that study considers shared and volatile hosts, the platform model assumes that the hosts have identical clock rates and that the platform supports checkpointing. Therefore, the study did not determine impact of relatively slow hosts or task failures on execution for a set of tasks; likewise, the author did not study the effect of resource prioritization (e.g., according to clock rates) or resource exclusion.

Currently, many desktop systems exist, but none have schedulers that target rapid application turnaround as most are geared toward high throughput applications only. XtremWeb uses a FCFS to schedule tasks to resources [19]. In Entropia [13, 37], the scheduler maintains several priority queues and allows applications to specify constraints on resources used (such as CPU speed), and as such, would be able to support many of the heuristics described previously. However, the policies by which to achieve rapid application turnaround has been unclear. BOINC [21] and Condor [32] also lack schedulers for short-lived jobs.

8.2. TASK REPLICATION

The authors in [23] use a probabilistic model similar to the one we described in Section 6 to analyze various replication issues. The platform model used in the study is similar to ours in that the resources are shared, task preemption is disallowed, and checkpointing is not supported. The application models were also similar; one model was based on tightly-coupled applications, while the other was based on loosely-coupled application, which consisted of task parallel components before each barrier synchronization. The authors then assume that the probability of task completion follows a geometric distribution.

Despite the similarities in platform and application models, there were a number of important differences between that study and our own. First, the results were based on a discrete time model, where the unit of time is the length of the task l. That is, if a task that began execution at time t fails, the task is started only after time $t + l$. This assumption is made to ensure each "trial" is evenly spaced so that computing the time to task completion is simplified. However, their assumption is problematic because it places an unrealistic constraint on the time required to restart task execution. In particular, in the case of a task failure, their model assumes that the expected time to failure plus the expected period of unavailability) must equal the task length l and is thus entirely dependent on the task length, which is a rare and improbable occurrence; The second difference between that study and our own is that their platform model assumes a homogeneous environment, and so their study does not consider the effect of using hosts of different speeds when replicating.

The work in [31] examines analytically the costs of executing task parallel applications in desktop grid environments. The model assumes that after a machine is unavailable for some fixed number of time units, at least one unit of work can be completed. Thus, the estimates for execution time are lower bounds. We believe the assumption is too restrictive, especially since the size of an availability intervals can be correlated in time [36]; that is, a short availability interval (which would likely cause task failure) will most likely be followed by another short availability interval.

Note that many works have employed task replication [51, 42, 50] to detect errors and ensuring result correctness. Although many of these types of security methods have been deployed by current systems, most are ad-hoc and none are fail-proof. For example, SETI@home recomputes tasks that have indicated a positive signal has been found on a dedicated machine to prevent false positives. Another example is the work described in [42] where the author develops methods to give probabilistic guarantees on result correctness, using a credibility metric for each worker.

8.3. CHECKPOINTING

Task checkpointing is another means of dealing with task failures since the task state can be stored periodically either on the local disk or on a remote checkpointing server; in the event that a failure occurs, the application can be restarted from the last checkpoint. In combination with checkpointing, process migration can be used to deal with CPU unavailability or when a "better" host becomes available by moving the process to another machine. However, remote checkpointing or process migration may be difficult in Internet environments, as the application can often consume hundreds of megabytes of memory and bandwidth. We will investigate checkpointing in desktop grids in future work.

9. Conclusions and Future Work

We summarize the findings of this work with a set of guidelines useful for desktop grid users and developers. We discuss these guidelines in terms of resource prioritization, exclusion, and task replication. In terms of resource prioritization, using static clock rate information alone resulted in the best improvement in application performance, despite (or perhaps because of) the complexity of the factors behind host volatility. This is partly due to the fact that host clock rates are correlated with several metrics of host performance, such as task completion rates and task completion time. Surprisingly, using more dynamic information, such as historical information about host availability, is not as beneficial because availability can be unpredictable as it tends to vary dramatically from one day to the next.

However, the performance benefits of prioritization alone can be hampered significantly by hosts with relatively slow clock rates. To remedy this, resource exclusion can be used to prevent these slow hosts from impeding application completion. We found that using a fixed threshold by which to exclude hosts according to clock rates was effective in environments with a large range of host clock rates. However, using a fixed threshold could actually exclude useful hosts, especially in environments whose hosts had a relatively smaller range of clock rates. Alternatively, we investigated a heuristic that excludes hosts using a makespan prediction, and found that this outperformed the fixed-threshold heuristics in computing environments with a relatively smaller range of clock rates. Moreover, this makespan predictor proved to be vital for the efficiency of replication heuristics.

Finally, to cope further with unpredictable task failures and host load, we designed heuristics that use active, reactive, or hybrid replication techniques. We found the most effective heuristic used reactive replication via timeouts based on our makespan predictor, which achieved performance with a factor of 1.7 of optimal. The reason timeouts are so effective is that platforms often have a large portion of relatively stable hosts and volatility is negatively correlated with clock rates; thus, when the reactive replication heuristic prioritizes tasks and hosts according to clock rates, the probability of failure is reduced dramatically after replicating a task once. Proactive replication could result is slightly better performance than reactive methods, but at the cost of wasting a high percentage of resources. Unfortunately, our hybrid replication heuristic based on a probabilistic model of task completion did not perform any better than the best reactive heuristic. We believe that this is due to effectiveness of using timeouts via a makespan predictor, and also the complexity of resource behavior, which makes it not amenable to probabilistic models.

While this study focuses on the scheduling of a single application, a desktop grid application does not always have the luxury of using the entire platform exclusively. It would be useful to investigate the scenario where multiple applications are competing for the same set of resources. Given that the

costs of using desktop resources for users that submit applications can be quite low, applications are often very large; at the same time there may be users that require rapid application turnaround. How to balance system throughput and response time while ensuring some notion of fairness among users is an interesting research direction. Toward this end, we hope that the scheduling guidelines found in this study will be a helpful stepping stone for future desktop grid research.

Acknowledgements

We thank Gilles Fedak for his invaluable assistance with the XtremWeb system used to conduct our simulation experiments. Also, we gratefully acknowledge Kyung Ryu and Amin Vahdat for providing the UCB trace data set and documentation.

References

1. Acharya, A., G. Edjlali, and J. Saltz: 1997, 'The Utility of Exploiting Idle Workstations for Parallel Computation'. In: *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. pp. 225–234.
2. Alexandrov, A. D., M. Ibel, K. E. Schauser, and C. Scheiman: 1997, 'SuperWeb: Towards a Global Web-Based Parallel Computing Infrastructure'. In: *Proc. of the 11th IEEE International Parallel Processing Symposium (IPPS)*.
3. Arpaci, R., A. Dusseau, A. Vahdat, L. Liu, T. Anderson, and D. Patterson: 1995, 'The Interaction of Parallel and Sequential Workloads on a Network of Workstations'. In: *Proceedings of SIGMETRICS'95*. pp. 267–278.
4. Barak, A., S. Guday, and W. R.: 1993, *The MOSIX Distributed Operating System, Load Balancing for UNIX*, Vol. 672 of *Lecture Notes in Computer Science*. Springer-Verlag.
5. Baratloo, A., M. Karaul, Z. Kedem, and P. Wyckoff: 1996, 'Charlotte: Metacomputing on the Web'. In: *Proc. of the 9th International Conference on Parallel and Distributed Computing Systems (PDCS-96)*.
6. Bhagwan, R., S. Savage, and G. Voelker: 2003, 'Understanding Availability'. In: *In Proceedings of IPTPS'03*.
7. Bolosky, W., J. Douceur, D. Ely, and M. Theimer: 2000, 'Feasibility of a Serverless Distributed file System Deployed on an Existing Set of Desktop PCs'. In: *Proceedings of SIGMETRICS*.
8. Braun, T., H. Siegel, and N. Beck: 2001, 'A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems'. *Journal of Parallel and Distributed Computing* **61**, 810–837.
9. Camiel, N., S. London, N. Nisan, and O. Regev: 1997, 'The PopCorn Project: Distributed Computation over the Internet in Java'. In: *Proc. of the 6th International World Wide Web Conference*.
10. CANCER, 'The Compute Against Cancer project'. `http://www.computeagainstcancer.org/`.
11. Cappello, P., B. Christiansen, M. Ionescu, M. Neary, K. Schauser, and D. Wu: 1997, 'Javelin: Internet-Based Parallel Computing Using Java'. In: *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.
12. Casanova, H., A. Legrand, D. Zagorodnov, and F. Berman: 2000, 'Heuristics for Scheduling Parameter Sweep Applications in Grid Environments'. In: *Proceedings of the 9th Heterogeneous Computing Workshop (HCW'00)*. pp. 349–363.
13. Chien, A., B. Calder, S. Elbert, and K. Bhatia: 2003, 'Entropia: Architecture and Performance of an Enterprise Desktop Grid System'. *Journal of Parallel and Distributed Computing* **63**, 597–610.
14. Chu, J., K. Labonte, and B. Levine: 2003, 'Availability and locality measurements of peer-to-peer file systems'. In: *Proceedings of ITCom: Scalability and Traffic Control in IP Networks*.
15. Dinda, P.: 1999, 'The Statistical Properties of Host Load'. *Scientific Programming* **7**(3–4).
16. Dinda, P.: 2002a, 'A Prediction-Based Real-Time Scheduling Advisor'. In: *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)*.
17. Dinda, P.: 2002b, 'Online Prediction of the Running Time of Tasks'. *Cluster Computing* **5**(3), 225–236.
18. entropia, 'Entropia, Inc.'. `http://www.entropia.com`.
19. Fedak, G., C. Germain, V. N'eri, and F. Cappello: 2001, 'XtremWeb: A Generic Global Computing System'. In: *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'01)*.
20. FIGHTAIDS, 'The Fight Aids At Home project'. `http://www.fightaidsathome.org/`.
21. for Network Computing, T. B. O. I. `http://boinc.berkeley.edu/`.
22. Frey, J., T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke: 2002, 'Condor-G: A Computation Management Agent for Multi-Institutional Grids'. *Cluster Computing* **5**(3), 237–246.
23. Ghare, G. and L. Leutenegger: 2004, 'Improving Speedup and Response Times by Replicating Parallel Programs on a SNOW'. In: *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*.

24. Ghormley, D., D. Petrou, S. Rodrigues, A. Vahdat, and T. Anderson: 1998, 'GLUnix: a Global Layer Unix for a Network of Workstations'. *Software-Practice and Experience* **28**(9).
25. GIMPS, 'The Great Internet Mersene Prime Search (GIMPS)'. `http://www.mersenne.org/`.
26. Hupp, S.: 1982, 'The "Worm" Programs – Early Experience with Distributed Computation'. *Communications of the ACM* **3**(25).
27. Kondo, D.: 2005, 'Scheduling Task Parallel Applications on Enterprise Desktop Grids'. Ph.D. thesis.
28. Kondo, D. and H. Casanova: 2004, 'Computing the Optimal Makespan for Jobs with Identical and Independent Tasks Scheduled on Volatile Hosts'. Technical Report CS2004-0796, Dept. of Computer Science and Engineering, University of California at San Diego.
29. Kondo, D., M. Taufer, C. Brooks, H. Casanova, and A. Chien: 2004, 'Characterizing and Evaluating Desktop Grids: An Empirical Study'. In: *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'04)*.
30. Kreaseck, B., L. Carter, H. Casanova, and J. Ferrante: 2003, 'Autonomous Protocols for Bandwidth-Centric Scheduling of Independent-task Applications'. In: *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*.
31. Leutenegger, S. and X. Sun: 1993, 'Distributed Computing Feasibility in a Non-Dedicated Homogeneous Distributed System'. In: *Proc. of SC'93, Portland, Oregon*.
32. Litzkow, M., M. Livny, and M. Mutka: 1988, 'Condor - A Hunter of Idle Workstations'. In: *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS)*.
33. Lodygensky, O., G. Fedak, V. Neri, F. Cappello, D. Thain, and M. Livny: 2003, 'XtremWeb and Condor: Sharing Resources Between Internet Connected Condor Pool'. In: *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'03) Workshop on Global Computing on Personal Devices*.
34. Long, D., A. Muir, and R. Golding: 1995, 'A Longitudinal Survey of Internet Host Reliability'. In: *14th Symposium on Reliable Distributed Systems*. pp. 2–9.
35. Mutka, M.: 1994, 'Considering Deadline Constraints When Allocating the Shared Capacity of Private Workstations.'. *Int. Journal in Computer Simulation* **4**(1), 41–63.
36. Mutka, M. and M. Livny: 1991, 'The available capacity of a privately owned workstation environment '. *Performance Evaluation* **4**(12).
37. Nabrzyski, J., J. Schopf, and J. Weglarz (eds.): 2003, *Grid Resource Management*, Chapt. 26. Kluwer Press.
38. Oram, A. (ed.): 2001, *Peer-To-Peer: Harnessing the Power of Disruptive Technologies*. Sebastopol, CA, USA: O'Reilly & Associates, Inc.
39. Pedroso, J., L. Silva, and J. Silva: 1997, 'Web-based metacomputing with JET'. In: *Proc. of the ACM PPoPP Workshop on Java for Science and Engineering Computation*.
40. Platform, 'Platform Computing Inc.'. `http://www.platform.com/`.
41. Pruyne, J. and M. Livny: 1996, 'A Worldwide Flock of Condors : Load Sharing among Workstation Clusters '. *Journal on Future Generations of Computer Systems* **12**.
42. Sarmenta, L.: 2001, 'Sabotage-tolerance mechanisms for volunteer computing systems'. In: *Proceedings of IEEE International Symposium on Cluster Computing and the Grid*.
43. Sarmenta, L. and S. Hirano: 1999, 'Bayanihan: Building and Studying Web-Based Volunteer Computing Systems Using Java'. *Future Generation Computer Systems* **15**(5-6), 675–686.
44. Saroiu, S., P. Gummadi, and S. Gribble: 2002, 'A measurement study of peer-to-peer file sharing systems'. In: *Proceedinsg of MMCN*.
45. SETI@home, 'The SETI@home project'. `http://setiathome.ssl.berkeley.edu/`.
46. Shirts, M. and V. Pande: 2000, 'Screen Savers of the World, Unite!'. *Science* **290**, 1903–1904.
47. Smallen, S., H. Casanova, and F. Berman: 2001, 'Tunable On-line Parallel Tomography'. In: *Proceedings of SuperComputing'01, Denver, Colorado*.
48. Sullivan, W. T., D. Werthimer, S. Bowyer, J. Cobb, G. Gedye, and D. Anderson: 1997, 'A new major SETI project based on Project Serendip data and 100,000 personal computers'. In: *Proc. of the Fifth Intl. Conf. on Bioastronomy*.
49. Synapse, 'DataSynapse Inc.'. `http://www.datasynapse.com/`.
50. Taufer, M., C. An, A. Kerstens, and C. L. B. III: 2005a, 'Predictor@Home: A "Protein Structure Prediction Supercomputer" Based on Public-Resource Computing.'. In: *IPDPS*.
51. Taufer, M., D. Anderson, P. Cicotti, and C. L. B. III: 2005b, 'Homogeneous Redundancy: a Technique to Ensure Integrity of Molecular Simulation Results Using Public Computing.'. In: *IPDPS*.
52. UD, 'United Devices Inc.'. `http://www.ud.com/`.
53. Vijay Pande: 2004. Private communication.
54. Wolski, R., N. Spring, and J. Hayes: 1999, 'Predicting the CPU Availability of Time-shared Unix Systems'. In: *Peoceedings of 8th IEEE High Performance Distributed Computing Conference (HPDC8)*.
55. Wyckoff, P., T. Johnson, and K. Jeong: 1998, 'Finding Idle Periods on Networks of Workstations'. Technical Report CS761, Dept. of Computer Science, New York University.