Resource Management for Rapid Application Turnaround on Enterprise Desktop Grids

Derrick Kondo[1] Andrew A. Chien[1] Henri Casanova[1,2]

[1]Dept. of Computer Science and Engineering [2] San Diego Supercomputer Center
University of California, San Diego

Abstract

Desktop grids are popular platforms for high throughput applications, but due 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 achieves performance within a factor of 1.7 of optimal.

1 Introduction

Cycle stealing systems that harness the idle cycles of desktop PCs and workstations date back to the PARC Worm [25] and have shown widespread success with popular projects such as SETI@home [43, 40], the GIMPS [24], Folding@Home [41], FightAidsAtHome [20], Computing Against Cancer [10], and others sustaining the throughput of over one million CPU's and 10's of Teraflops/seconds [33]. 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 [35, 38, 2, 11, 9, 5, 19] and local-area networks [28, 4, 23]. In addition, commercial products have also emerged [18, 45, 44, 36]. 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 resource management and application scheduling issues have been thoroughly studied in the areas of parallel and Grid computing, most of this work considers resource failures 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, nearly identical tasks that vary in size, and 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, so that a scenario in which the number of resources is of an order of magnitude comparable to the number of tasks is not uncommon. As such, rather asymptotic work rate, application execution time is an appropriate performance metric. This corresponds to a different usage of desktop grids, in which users wish

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

SC2004, Pittsburgh, Pennsylvania
0-7695-2153-3/04 $20.00 (c)2004 IEEE

to execute relatively small and/or short-lived applications quickly, rather than achieve high throughput over a long period of time.

Note that our work focuses on minimizing the overall execution elapsed time, or *makespan*, of a single parallel application, rather than trying to optimize the performance of multiple, competing jobs. While the design of so-called "job scheduling" strategies that promote average job performance and fairness among jobs that belong to different users is part of our larger goal in the context of desktop grids, it is outside the scope of this paper as we solely focus on application scheduling strategies. Note however the heuristics we develop to schedule a single application 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 deciding the task duplication level for each job).

Minimizing the makespan of a parallel application is the objective of numerous research projects in parallel computing; 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 based on traces of a real desktop grid running the Entropia commercial software at the San Diego Supercomputer Center (SDSC) [27]. Also, we evaluate the heuristics on three other representative desktop grid configurations. More specifically, by transforming the Entropia desktop traces, we model a homogeneous cluster, a multi-cluster grid, and a typical Internet-wide desktop grid. To measure the performance of our heuristics, we compare the resulting makespans to the optimal, which we can compute using an omniscient 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 simple prioritization techniques perform poorly in most desktop grid configurations. We also find that exclusion based on a fixed threshold works well on platforms where the distribution of clock rates is left heavy, in our case for the SDSC and Internet-wide desktop grids. We develop an adaptive heuristic that uses a prediction of the application's makespan to exclude slow resources; using historical host availability information, the makespan prediction is accurate to within 10% on average. Moreover, the heuristic based on makespan prediction usually outperforms simple resource exclusion techniques on multi-cluster and homogeneous grids. We then modify the heuristic to use replication techniques, and the performance of the best resulting method is less than a factor of 1.7 away from the optimal schedule, and significantly better than first-come-first-serve (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 methods, in particular, our simulation model, source of our trace data, and performance metrics. In Section 4, we evaluate resource prioritization heuristics. In Section 5, we develop heuristics that filter resources using different criteria. In Section 6, we augment our resource exclusion heuristics to use replication. Finally, in sections 7 and 8, 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. Host and CPU availability is assumed to be 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 "heart-beat" 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])l

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 tasks 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. 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. In fact, this is the strategy used by most existing desktop grid systems.

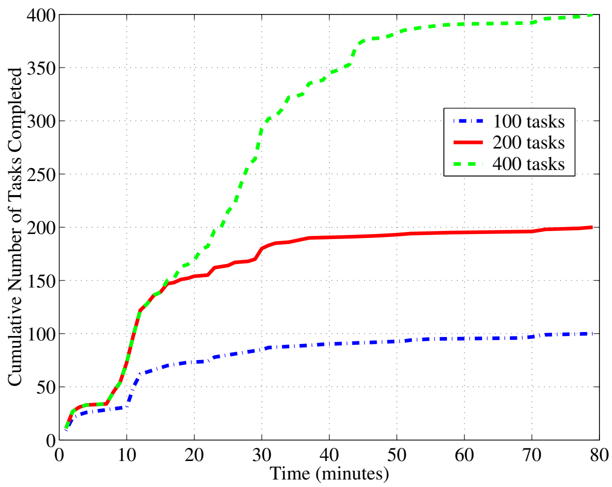

Figure 1: Cumulative task completion vs. time.

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 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 different 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 [24, 46] 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 and thus repeatable experiments. However, our approach is to use simulations driven by traces that were collected from a real desktop grid platform. These traces are time-series of CPU availability measurements (from 0% to 100%, including failure data) obtained over a 1-month time period on a 220-host desktop grid running the Entropia software infrastructure. Using these traces we can simulate that same desktop grid, as well as 3 other representative grid configurations with different host clock rate probability distributions. Our simulations implement the platform model described in Section 2.1. We simulate the execution of task-parallel applications with different numbers of tasks and different task durations. For each simulated execution we compute the makespan achieved when using different resource selection heuristics. Our performance metric is the makespan relative to the optimal makespan. We provide relevant details on the above in the following four sections.

3.1 Availability Traces

We have conducted availability measurements over a 1 month period with a deployment of Entropia DCGrid™ [18] at the San Diego Supercomputer Center (SDSC) over about 200 hosts. We continuously submitted short tasks to the Entropia system so that its work queue was never empty and each host was sent a task as soon as it became available. Note that these tasks, because managed by the Entropia virtual machine, did not interfere with the work of the resource owners who were in fact unaware of our measurement activities. Each compute-bound task performed a mix of floating point and integer operations and periodically (every 10 seconds) logged the computation rates to a file. A dedicated 1.5GHz Pentium processor can perform 37.5 million such operations per second. Note that the Entropia virtual machine makes it possible for a task to use only a fraction of a resource's CPU.

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 [8, 1, 7, 14, 39]; 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, 42, 15, 47]. 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 <1% CPU availability) but its host is still available (e.g., when local processes use more than 99% of 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 two 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 [8, 1, 7, 14, 39]. 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, 47, 31] 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 this data are reported in [27], but in this paper, we use our availability traces solely to drive simulations. Let us note that our traces revealed the overhead incurred by the Entropia system for initiating a task on a host (on the order of 40 seconds), which we take into account in our simulations.

3.2 Simulated Platform

We implemented the platform model described in Section 2.1, with 220 hosts with the availability given by the traces described in the previous section. All simulations were performed using traces captured during business hours from 9AM to 5PM. As shown in a number of studies [1, 31], hosts during weekdays business hours often exhibit higher user load, which in turn results in a more challenging scheduling problem.

Given the diversity of desktop configurations, we compare the performance of our heuristics on three other configurations representative of Internet, single cluster, and multi-cluster desktop grids. Internet desktop grids that utilize machines both in the enterprise and home settings usually have many more slow hosts than fast hosts, and so its host speed distribution is left heavy. We used host CPU statistics collected by the GIMPS Internet-wide project 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 [46].

For smaller projects [3, 19], desktop grids are often limited to machines within a student lab, for example, where the distribution of CPU speeds is relatively homogeneous. To model this scenario, we used the log of clock rates of machines at SDSC, which yielded a narrow normal distribution with a mean speed of 3.2GHz and standard de-

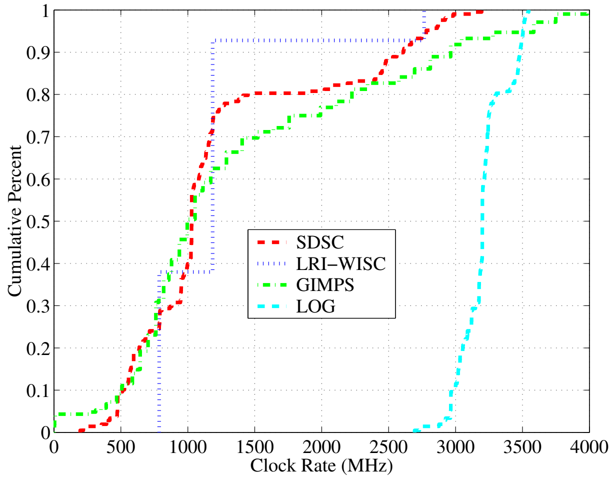

Figure 2: Cumulative clock rate distributions from real systems used for our simulations.

viation of 170MHz. Much work [22, 19, 37] in desktop grids has focused on using resources found in multiple labs. Recently, [29] 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 four platform scenarios in Figure 2. For each of these distributions, we ran simulations using the same traces described in Section 3.1 but transforming host clock speeds accordingly.

3.3 Simulated Applications

We simulated applications that varied in both the number and size of tasks. Applications have 100, 200, or 400 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 [27], 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, an XtremWeb [19] platform was used to run our simulations.

3.4 Performance metrics

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 oracle that has full knowledge of future host availabilities. Our oracle 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 prove formally in [26]. 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 our oracle.

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 on the SDSC grid using trace-driven simulation. For the **PRI-CR** method, hosts in the server's ready queue are prioritized by their clock rates. 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. 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. In this way, hosts in each queue are prioritized according to their speed.

Figure 3 shows the average makespan of these three algorithms and of the FCFS strategy normalized to the mean optimal execution time (labeled **OPTIMAL**) 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 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 3, 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 / 5min 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 4 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 do, 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.

Perhaps unexpectedly, PRI-HISTORY achieves poor performance. We found that the availability interval size, both in terms of time and in terms of operations, was not stationary across weekdays. We determined the error from

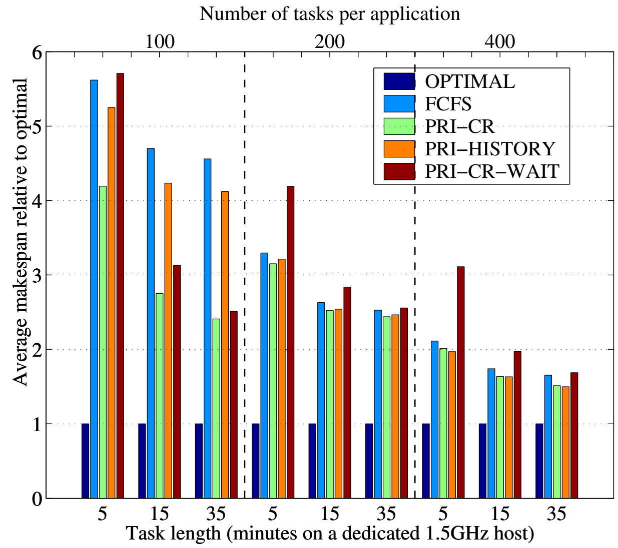

Figure 3: Performance of resource prioritization heuristics on the SDSC grid.

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 next. Then, we took the mean over all hosts and over all pairs of days and found that the average prediction error when using the previous day as a predictor was 110 minutes on a dedicated 1.5GHz host. (The host speed is chosen arbitrarily to give a human readable number instead of the number of operations.) The same process was done for predicting availability intervals in terms of time from one day to the next, and we found that the average error was 108 minutes. Given that most tasks in our experiments are less then 2 hours in length, these prediction errors show that using the expected value for operations per interval or time per interval is a poor predictor, which is also supported by the findings in [48].

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 using the slowest hosts significantly limited performance, and we address this issue through heuristics described in the next section.

5 Resource Exclusion

To prevent slower hosts from delaying application completion, we developed several heuristics that excluded hosts from the computation using a variety of criteria. All

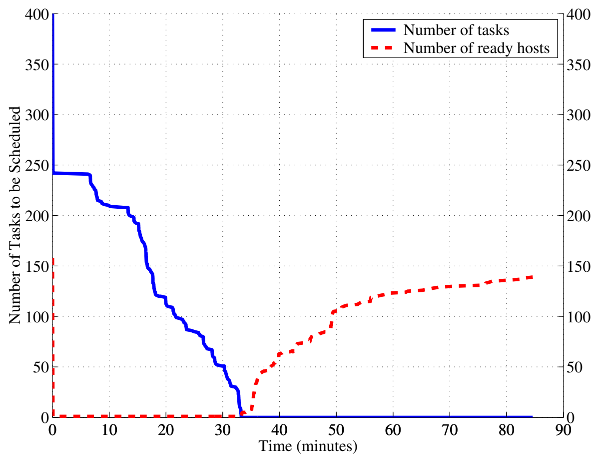

Figure 4: Number of tasks to be scheduled (left y-axis) and hosts available (right y-axis).

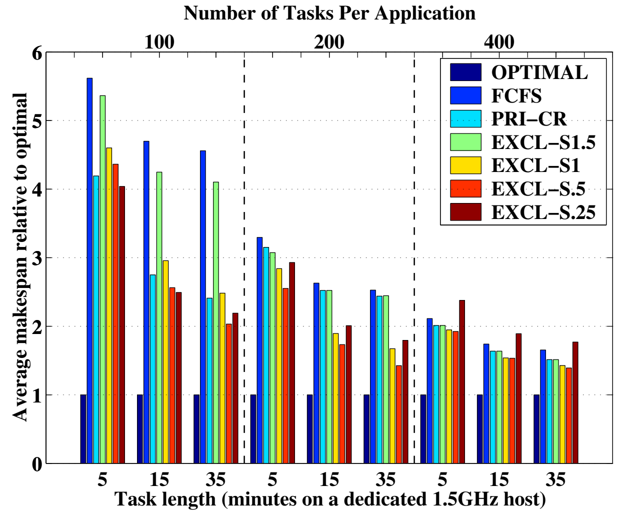

Figure 5: Performance of heuristics using thresholds on SDSC grid

these heuristics use only host clock rates to obtain lower bounds on task completion time (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 5 that excluding hosts 1 or .5 standard deviations below the mean can bring substantial benefits 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.

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 PRI-S.5, we develop a heuristic where the scheduler makes a prediction of application makespan, and then excludes those resources that cannot complete a task by the projected completion time. To predict the makespan, we compute the average operations completed per second for each host from our traces and then compute the average over all hosts (call this average r). If N is the number of hosts in the desktop grid, we then assume we have N hosts of speed r. If T is the number of tasks and s is the size of the task in operations, the optimal execution time of the entire application is estimated with $wr = \lceil T/N \rceil (s/r)$. 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 by which tasks can be completed with $\lceil T/N \rceil$.

To assess the quality of our predictor wr, 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 computation power of the grid remains relatively constant, although the individual resources may have unpredictable availability intervals as discussed previously in Section 4. 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 the accurate predictions of wr 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 wr, 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 $pred - .95 * meanops$, where $pred$ 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 if the heuristic waited until time $pred$ (versus $pred - .95 * meanops$) before switching to PRI-CR, it would result in poor resource utilization as seen in some of our early simulations, since most hosts are available and excluded by time $pred$. Therefore, waiting until time $pred$ before making task assignments via PRI-CR would cause most hosts to sit needlessly idle.

5.2.1 Evaluation on Different Desktop Grids

Figure 6 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. EXCL-PRED performs more poorly than EXCL-S.5 for the application with two-hundred 15-minute tasks. We have found after close inspection of our traces that this is because of a handful unpredictably 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

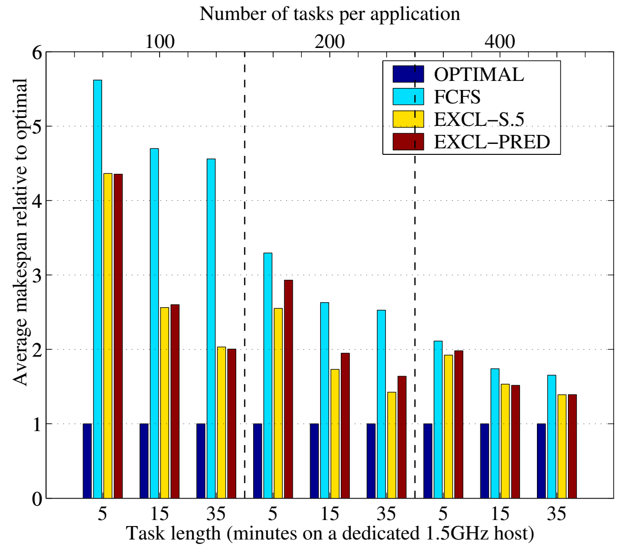

Figure 6: Heuristic performance on the SDSC grid

enough tasks to keep other hosts busy until the slow hosts can finish task execution. For the application with 100 tasks, the unpredictably slow and unstable hosts get filtered out automatically as there are fewer tasks than hosts and the heuristic prioritizes resources by clock rate. The same reasoning can explain why EXCL-S.5 outperforms EXCL-PRED for the GIMPS desktop grid(see Figure 7), which like the SDSC grid has a left heavy distribution of resource clock rates. On the GIMPS resources, applications scheduled with FCFS invariably 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 .

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 case and homogeneous cluster. EXCL-PRED outperforms EXCL-S.5 in the case of the LRI and log distributions by 17% and 12% respectively for the application with 400 tasks (see Figures 8 and 9). EXCL-S.5 in the LRI desktop grid excludes all 600MHz hosts, which contribute significantly to the platform's overall computing power. In the relatively homogeneous desktop grid, EXCL-S.5 unnecessarily filters about 10% of the total computing power when in fact these resources are running at speeds close to the mean and so would not significantly delay application completion. Although the hosts excluded by EXCL-S.5 are relatively slow, their absolute speeds are still close to the faster hosts and thus

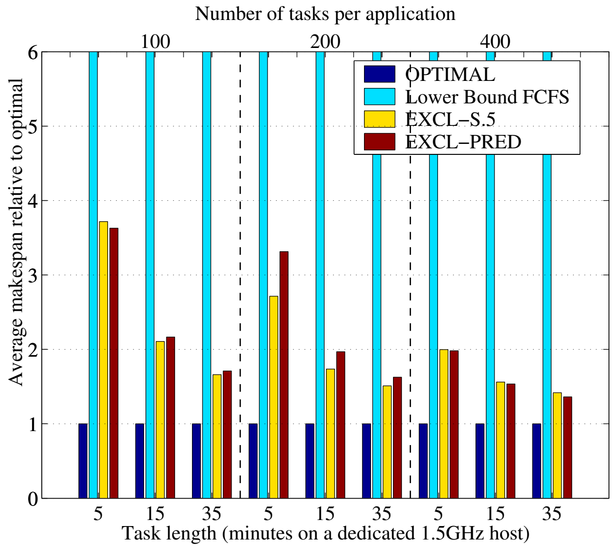

Figure 7: Heuristic performance the GIMPS grid

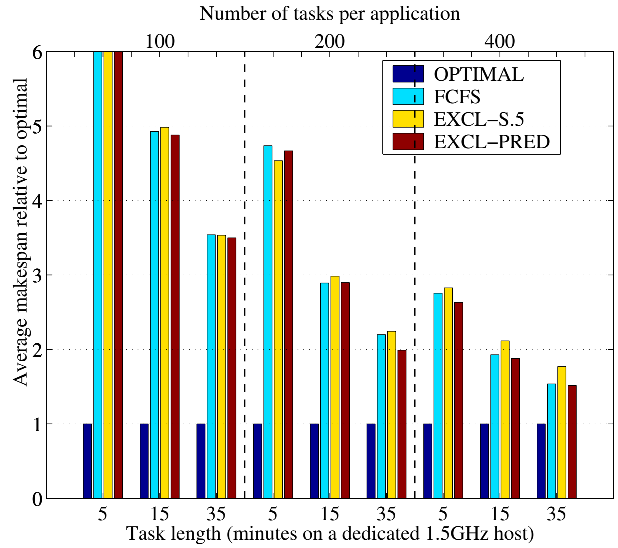

Figure 9: Heuristic performance on the homogeneous grid

contribute significantly to progress in application execution. 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, as seen in Figures 8 and 9.

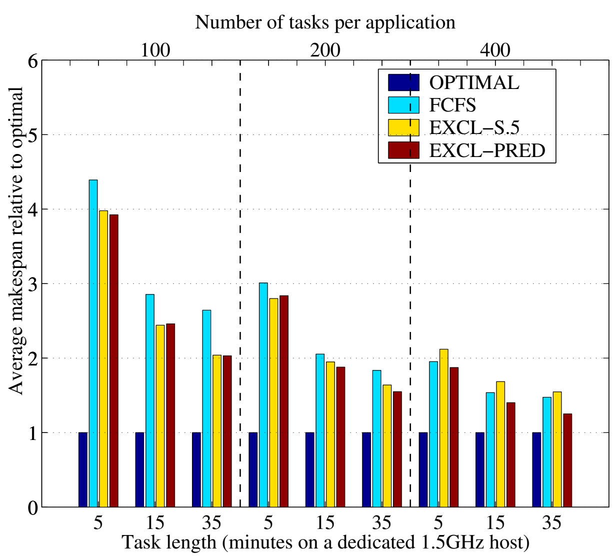

Figure 8: Heuristic performance on the LRI-WISC grid

6 Task Replication

An approach to reduce application completion delays caused by failures and loaded hosts is to extend EXCL-PRED to use task replication. One method, which we call **EXCL-PRED-DUP**, uses EXCL-PRED but replicates each task when the number of ready hosts is greater than the number of tasks to schedule. Replicating anytime sooner could cause a host to do redundant work when there are more unscheduled tasks than hosts, and thereby cause a delay in application completion. We refer to the original task being replicated as a *primary* task, and the replicas are called *duplicates*. Primary tasks are always scheduled before duplicates, which reduces the chance that a high number of duplicates ahead in the work queue prevent a primary task from being scheduled. The duplicates themselves are sorted in increasing order by the clock rate of the host that the primary task was first assigned to so that replicas of tasks that were originally assigned to slower hosts are scheduled earlier.

In addition to failures near the end of application execution, delays can be caused by hosts that run unexpectedly slow; since EXCL-PRED filters resources by clock rate, the heuristic is susceptible to such slow hosts. (Nevertheless, most hosts, when available, have completely unloaded CPU's most of the time [27], which was our justification for using clock rate as a predictor of execution time in EXCL-PRED.) To deal with such delays, another heuristic **EXCL-PRED-TO** based on EXCL-PRED 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, and the primary and duplicate tasks are prioritized similarly to EXCL-PRED-DUP.

Figure 10 shows that the runtime improvement due to EXCL-PRED-TO is most dramatic for smaller applications, where much of the execution time is spent near the end as unpredictably slow hosts complete tasks and failed tasks are successfully finished. As shown from our simulation logs, timeouts can effectively reschedule tasks that are assigned to unpredictably slow and unstable hosts. EXCL-PRED-TO does 13% better than EXCL-S.5 in almost all cases except the application with two-hundred, 35-minute tasks; for that particular application size, EXCL-PRED-TO does equally well. EXCL-PRED-TO usually performs similarly to EXCL-S.5 because replicating only those tasks at the end of the application does not effectively deal with straggling or unstable hosts that began execution earlier.

EXCL-PRED-DUP shows little improvement over EXCL-PRED because the failures that occur near the end of the application are due to relatively slow hosts, many of which began task execution before the point at which there are more ready hosts than tasks. EXCL-PRED-DUP does too much replication too late in the application's lifespan. In contrast to EXCL-PRED-DUP, the EXCL-PRED-TO heuristic is able to deal with these failures in addition to unpredictably slow hosts by means of timeouts, and as such, performs remarkably well as all execution times are within a factor of 1.7 or less with respect to the optimal. For the desktop configurations other than SDSC, the degree of improvement for the heuristic is similar.

Surprisingly, the gain in performance due to replication in EXCL-PRED-TO comes at little expense of resource utilization. That is, the improved performance is obtained is obtained through relatively little replication. The bar labeled as 'Waste' in Figure 11 shows that the percent of replicated tasks (relative to the total and including replicated tasks that fail to complete) for EXCL-PRED-TO is less than about 12% for each application size. The low amount of replication required is due to the fact that only the relatively few tasks uncompleted near the end of the application need to be replicated. Moreover, by the end of the application, the number of available hosts compared to the number of tasks is quite high and so, the chance of selecting a relatively fast host is high.

Figure 11 also compare the performance of EXCL-PRED-DUP with EXCL-PRED-TO where EXCL-PRED-DUP uses the same number of replicated tasks as EXCL-PRED-TO (denoted by 'EXCL-PRED-DUP*'). For example, for an application consisting of one-hundred 5-minute tasks, EXCL-PRED-TO replicates 12 tasks, which corresponds to a 12% waste. We then modified EXCL-PRED-DUP to replicate each task a fixed number of times so that in the end, 12 tasks are also replicated, and so the percentage of waste is equivalent to that of EXCL-PRED-TO.

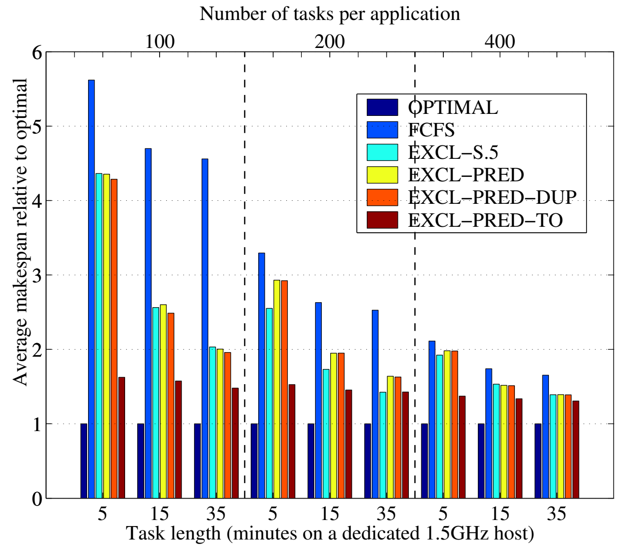

Figure 10: Heuristic performance using replication on SDSC grid

The corresponding bars in Figure 11 shows that for the same level of replication, EXCL-PRED-TO is far more effective is reducing application makespan.

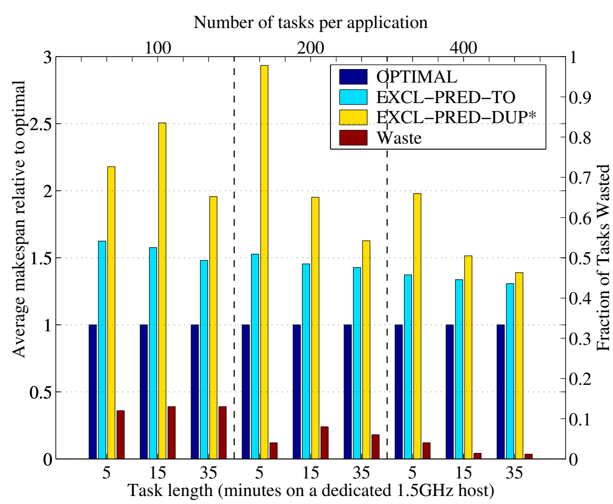

Figure 11: Performance of replication algorithm on SDSC grid for a particular level of replication

Of all the heuristics evaluated, EXCL-PRED-TO performs closest to the optimal without significantly negatively affecting resource utilization. We obtained confidence intervals for application completion time when using EXCL-PRED-TO by scheduling more than 1000 applications for a given size through trace driven simulation, and then used the empirical CDF to determine confidence intervals for application makespans. For example, the lower 95% confidence intervals for an application with four-hundred tasks are [24, 36], [70, 92], and [161, 205] minutes for 5-minute, 15-minute, 60-minute tasks respectively, where the mean application completion times are 30, 81, and 182 minutes. With these confidence intervals, a user could get a reasonable estimate of when her application would complete.

7 Related Work

Although many desktop systems exist, none have schedulers that promote 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, 32], 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 mechanisms describe previously. However, the method by which to achieve rapid application turnaround has been unclear. BOINC [21] and Condor [28] also lack schedulers for short-lived jobs.

Much scheduling research has been done with the prediction of host load for resource selection [16, 6]. However, as discussed in Section 3.1, these studies do not take into account task failures (caused by a user reclaiming her machine during task execution), which can significantly delay application completion. (For example, the failure rate for a 35 minute task during business hours is 22%.) Moreover, the studies that are based on host load traces are susceptible to OS idiosyncrasies.

Batch resource management systems such as the Maui Scheduler [30] and PBS [34] assume a relatively dedicated and stable computing environment, and are inadequate for scheduling applications on desktop grids because they lack extensive mechanisms to deal with task failures. As such, scheduling features that are normally available in these batch systems (such as backfilling, advance reservation, or mechanisms for fairness) are not supported on desktop grids.

8 Conclusion

We have developed resource selection heuristics that can achieve good performance for short-lived, task-parallel applications on desktop grids. The heuristics used three techniques, namely resource prioritization, resource exclusion, and task replication, and were evaluated using trace-driven simulation of four grid configurations.

We found that simple prioritization of resources was usually ineffective, and that utilizing all hosts in desktop grid can prove detrimental to application completion time. Consequently, we investigated methods for excluding hosts by using a fixed threshold, or an adaptive threshold based on a prediction of application makespan. Although using a fixed threshold to excluded certain hosts is beneficial for desktop grids with a left-heavy distribution of clock rates, the adaptive makespan heuristic performs as well or better for other configurations, such as multi-cluster or homogeneous desktop grids. Then, we adapted the makespan prediction heuristic to use replication as a means to deal with task failures and unreliable hosts that often delayed application completion. With little waste caused by replicated tasks, the new heuristic brings application completion to within a factor of 1.7 of the optimal for all application sizes in our experiments.

Surprisingly, using minimal information, i.e., clock rates, about the hosts, our heuristics were able to improve application makespan drastically. Given that both Internet and Enterprise desktop grids [21, 19, 18] collect and store clock rate information, the scheduling heuristics could easily be implemented and integrated with current systems. We plan to implement these heuristics in the XtremWeb software [19].

For future work, we will evaluate our heuristics for a system such as Condor [28] or MOSIX [4] that enables checkpointing and migration of tasks. Checkpointing and process migration are two methods that deal with host volatility, and we will investigate how these methods complement resource prioritization, resource exclusion, and task replication.

We will also design scheduling heuristics for the scenario where multiple applications are submitted over time. With the understanding how our heuristics affect the execution of a single application, the results can be used as the basis for supporting a multi-application online workload, consisting of both short-lived and high throughput applications. In addition to application makespan, metrics for system performance and fairness will have to be considered.

Acknowledgments

The authors wish to thanks Gilles Fedak for his invaluable assistance with the XtremWeb system used to conduct our simulation experiments.

References

[1] A. Acharya, G. Edjlali, and J. Saltz. 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*, pages 225–234, 1997.

[2] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C.J. Scheiman. SuperWeb: Towards a Global Web-Based Parallel Computing Infrastructure. In *Proc. of the 11th IEEE International Parallel Processing Symposium (IPPS)*, April 1997.

[3] R.H. Arpaci, A.C. Dusseau, A.M. Vahdat, L.T. Liu, T.E. Anderson, and D.A." Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of SIGMETRICS'95*, pages 267–278, May 1995.

[4] A. Barak, S. Guday, and Wheeler R. *The MOSIX Distributed Operating System, Load Balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[5] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proc. of the 9th International Conference on Parallel and Distributed Computing Systems (PDCS-96)*, 1996.

[6] A. Bestavros. Load Profiling In Distributed Real-Time Systems. In *The 17th International Conference on Distributed Computer Systems*, May 1997.

[7] R. Bhagwan, S. Savage, and G. Voelker. Understanding Availability. In *In Proceedings of IPTPS'03*, 2003.

[8] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a Serverless Distributed file System Deployed on an Existing Set of Desktop PCs. In *Proceedings of SIGMETRICS*, 2000.

[9] N. Camiel, S. London, N. Nisan, and O. Regev. The PopCorn Project: Distributed Computation over the Internet in Java. In *Proc. of the 6th International World Wide Web Conference*, April 1997.

[10] The Compute Against Cancer project. `http://www.computeagainstcancer.org/`.

[11] P. Cappello, B. Christiansen, M. Ionescu, M. Neary, K. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1997.

[12] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW'00)*, pages 349–363, May 2000.

[13] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and Performance of an Enterprise Desktop Grid System. *Journal of Parallel and Distributed Computing*, 63:597–610, 2003.

[14] J. Chu, K. Labonte, and B. Levine. Availability and locality measurements of peer-to-peer file systems. In *Proceedings of ITCom: Scalability and Traffic Control in IP Networks*, July 2003.

[15] P. Dinda. The Statistical Properties of Host Load. *Scientific Programming*, 7(3–4), 1999.

[16] P. Dinda. A Prediction-Based Real-Time Scheduling Advisor. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)*, April 2002.

[17] P. Dinda. Online Prediction of the Running Time of Tasks. *Cluster Computing*, 5(3):225–236, July 2002.

[18] Entropia, Inc. `http://www.entropia.com`.

[19] G. Fedak, C. Germain, V. N'eri, and F. Cappello. XtremWeb: A Generic Global Computing System. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'01)*, May 2001.

[20] The Fight Aids At Home project. `http://www.fightaidsathome.org/`.

[21] The Berkeley Open Infrastructure for Network Computing. `http://boinc.berkeley.edu/`.

[22] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[23] D. Ghormley, D. Petrou, S. Rodrigues, A. Vahdat, and T. Anderson. GLUnix: a Global Layer Unix for a Network of Workstations. *Software-Practice and Experience*, 28(9), July 1998.

[24] The great internet mersene prime search (gimps). `http://www.mersenne.org/`.

[25] S.A. Hupp. The "Worm" Programs – Early Experience with Distributed Computation. *Communications of the ACM*, 3(25), 1982.

[26] D. Kondo and H. Casanova. 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, July 2004.

[27] D. Kondo, M. Taufer, , C. Brooks, H. Casanova, and A. Chien. Characterizing and Evaluating Desktop Grids: An Empirical Study. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'04)*, April 2004.

[28] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS)*, 1988.

[29] O. Lodygensky, G. Fedak, V. Neri, F. Cappello, D. Thain, and M. Livny. 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*, May 2003.

[30] Maui Scheduler. `http://www.supercluster.org/maui`.

[31] M. Mutka and M. Livny. The available capacity of a privately owned workstation environment . *Performance Evaluation*, 4(12), July 1991.

[32] J. Nabrzyski, J. Schopf, and J. Weglarz, editors. *Grid Resource Management*, chapter 26. Kluwer Press, 2003.

[33] Andy Oram, editor. *Peer-To-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[34] The Portable Batch System Webpage. `http://www.openpbs.com`.

[35] J. Pedroso, L.M. Silva, and J.G. Silva. Web-based metacomputing with JET. In *Proc. of the ACM PPoPP Workshop on Java for Science and Engineering Computation*, June 1997.

[36] Platform Computing Inc. `http://www.platform.com/`.

[37] J. Pruyne and M. Livny. A Worldwide Flock of Condors : Load Sharing among Workstation Clusters . *Journal on Future Generations of Computer Systems*, 12, 1996.

[38] L. Sarmenta and S. Hirano. Bayanihan: Building and Studying Web-Based Volunteer Computing Systems Using Java. *Future Generation Computer Systems*, 15(5-6):675–686, 1999.

[39] S. Saroiu, P.K. Gummadi, and S.D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedinsg of MMCN*, January 2002.

[40] The seti@home project. `http://setiathome.ssl.berkeley.edu/`.

[41] M.R. Shirts and V.S. Pande. Screen Savers of the World, Unite! *Science*, 290:1903–1904, 2000.

[42] S. Smallen, H. Casanova, and F. Berman. Tunable On-line Parallel Tomography. In *Proceedings of SuperComputing'01, Denver, Colorado*, Nov. 2001.

[43] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, G. Gedye, and D. Anderson. A new major SETI project based on Project Serendip data and 100,000 personal computers. In *Proc. of the Fifth Intl. Conf. on Bioastronomy*, 1997.

[44] DataSynapse Inc. `http://www.datasynapse.com/`.

[45] United Devices Inc. `http://www.ud.com/`.

[46] Vijay Pande. Private communication, 2004.

[47] R. Wolski, N. Spring, and J. Hayes. Predicting the CPU Availability of Time-shared Unix Systems. In *Peoceedings of 8th IEEE High Performance Distributed Computing Conference (HPDC8)*, August 1999.

[48] P. Wyckoff, T. Johnson, and K. Jeong. Finding Idle Periods on Networks of Workstations. Technical Report CS761, Dept. of Computer Science, New York University, March 1998.