Computing the Optimal Makespan for Jobs with Identical and Independent Tasks Scheduled on Volatile Hosts

Derrick Kondo[1] Henri Casanova[1,2]

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

1 Introduction

Desktop grids harvest the cycles of idle workstations to support large-scale computation. As such, these systems are cost-effective platforms for compute-intensive applications. Ongoing Internet-wide projects such as SETI@home and FOLDING@home use hundreds of thousands of PC's for high-throughput applications. In addition to these public projects, several companies such as Entropia and United Devices [3, 10, 9] have developed software for desktop grids within an enterprise.

The resources used by these projects are volatile; failures due to prolonged user activity (keyboard/mouse) or to a machine being shutoff are common. The default scheduling approach in most desktop grids systems [4, 5] is First Come First Serve (FCFS), by which the next application task is assigned to the next available host. For high-throughput jobs, that is ones in which the number of tasks is orders of magnitude larger than the number of resources, this method achieves close to optimal performance. However when the number of tasks in a job is comparable to the number of hosts and the goal is to minimize job *makespan*, that is the time between job submission and job completion, the FCFS approach is inadequate for two reasons [6]. First, FCFS might unnecessarily schedule tasks on slower resources near the end of job completion, thus delaying completion while other, faster hosts remain idle. Second, task failures near the end of job execution force the task to be restarted from scratch (assuming no checkpointing).

In [6], we developed several heuristics for scheduling relatively small jobs on an enterprise desktop grid, and evaluated their performance using simulations driven by traces of an actual desktop grid platform. To evaluate our heuristics, we wish to compare them to an optimal scheduler. In practice, an optimal schedule can generally not be achieved as resource availabilities are dynamic and unpredictable. However, in trace-based simulation it is possible to devise an "omniscient" scheduling algorithm that has full knowledge of future resource availabilities, and that can use this knowledge to compute the optimal schedule. In [6], we used a greedy algorithm to determine the optimal schedule, and thus the optimal makespan. While in that work we had only strong presumptions that the algorithm was optimal, in this report we give a formal proof of optimality.

Our approach to prove optimality is as follows. After defining the problem formally in Section 2, we first show optimality within a single availability interval on a single host in Section 3. Then, in Section 4, we show optimality for multiple availability intervals separated by failures on a single host. In Section 5, we show the optimality for multiple availability intervals across multiple hosts. Finally, in Section 6, we consider a variation of the problem that allows for task checkpointing.

2 Scheduling Problem

2.1 Host and CPU Availability Traces

One of the difficult aspects of application scheduling on desktop grids is the dynamic resource availability. Our approach was to collect availability traces on a real platform, and use these traces to drive simulation of scheduling algorithms. We obtained one set of traces by submitting an infinite stream of tasks to the Entropia desktop grid system [3] at the San Diego Supercomputer Center (SDSC). Each (short) task constantly performed a mix of floating point and integer operations, and periodically (every 10 seconds) logged the number of operations completed to file. After the tasks had completed, the output files were assembled to produce a single continuous trace per host.

The main advantage of this approach is that our tasks utilize and perceive the desktop grid exactly as a real application would. Our traces account for task failures (due to prolonged keyboard/mouse activity, host shutdown), and result in a detailed log of fluctuating CPU availability that is immune to OS idiosyncrasies. Other methods that use lightweight sensing techniques do not adequately detect task failures [12, 2, 11], and trace studies such as those in [1] contain only coarse grain information.

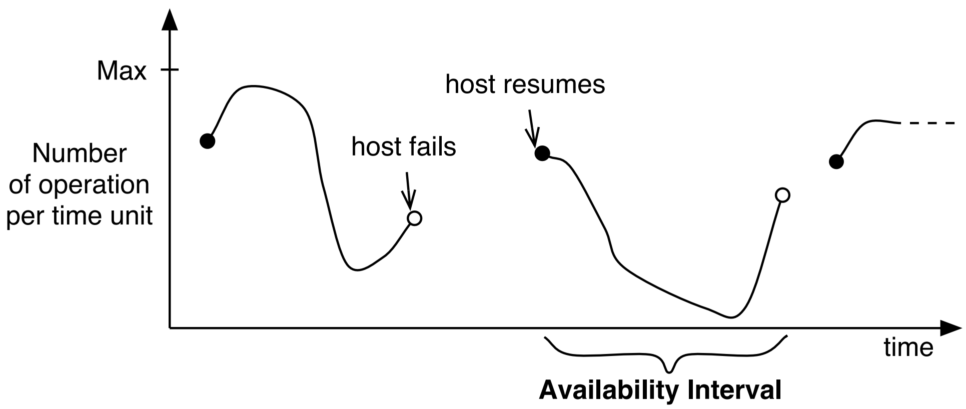

Figure 1: Depiction of a host availability trace.

Figure 1 depicts the structure of one of our availability traces. A trace consists of time-stamped values of the number of operations delivered since the last time-stamp (every 10 seconds for our traces), as seen on the y-axis of the figure. The maximum number of operations, `Max`, is achieved when 100% of the CPU of the host is used by the desktop grid application. A number of operations of zero means that the desktop grid application is suspended and may be resumed at a later date. Our traces also contain events when the host "fails" (which may be due to the host shutting down, to the desktop grid application being terminated due to user activity, or to the host actually failing), and when the host "resumes". We call the interval of time between a resume and a failure an *availability interval*. Each trace thus consists of a sequence of availability intervals separated by gaps during which the host is unusable for the desktop grid application. To complete successfully, an application task must start and finish within a single availability interval (but may be suspended during that interval).

We denote by $f(t)$ the instantaneous number of operations delivered at instant t. Although Figure 1 depicts an availability trace as continuous, it is in fact a discrete step functions. We denote by $\int_a^b f(t)\,dt$ the number of operations that would be delivered by the host to the desktop grid application between time a and time b, provided that the interval $[a, b]$ is fully contained within an availability interval.

2.2 Problem Statement

Consider a job that consists of T tasks, and consider N hosts, with variable CPU availability described by traces as in the previous section, and with possibly different maximum amount of operations delivered per time unit. All the tasks are of identical computational cost and independent of one another. We denote by S the task size in number of operations, and h is the overhead in seconds for scheduling each task, which is incurred before computation can begin (we have observed this overhead in practice, as seen in [7]. Finally, we denote by $f_m(t)$ the instantaneous number of operations delivered at time t on some host m, which is fully known given a trace (see the previous section). Recall that although in practice function f_m would not be known, in this report we focus on developing an optimal schedule that could be achieved by an omniscient algorithm that has foreknowledge of future host availability for all hosts. The scheduling problem is to assign the tasks to the hosts so that the job's makespan is minimized.

The optimal algorithm we proposed in [6] consisted of starting a task on each host as soon as possible (during an availability interval), and assigning tasks to a host as soon as the previous task on that host completes. If a task fails due to the end of the availability interval being reached before all necessary operations have been delivered to the task, then the task is started at the beginning of the next availability interval. In this greedy fashion, the availability intervals of all hosts can be "filled" with an infinite number of tasks, with tasks packed together as tightly as possible. These tasks are sorted by increasing completion time, and we pick the first T tasks. The assignment of these T tasks to hosts corresponds to the optimal schedule. While this schedule is intuitively optimal, one may wonder whether inserting some delays before or in between tasks could not be beneficial in order to match the overhead periods of length h with periods during which host CPUS exhibit low, or in fact 0%, availability. In the following sections we give a formal description of the algorithm and formal proofs of its optimality first for a single availability interval on a single hosts, then for multiple availability intervals on a single hosts, and finally for multiple availability intervals on multiple hosts, which is the general case. While, a posteriori, the proof is straightforward, the algorithm's optimality is still worth proving formally since it is used heavily in [6].

3 Single Availability Interval On A Single Host

We first focus on optimally scheduling tasks within a single availability interval. We formalize the algorithm and then prove its optimality.

3.1 Scheduling Algorithm

Let us first define a helper function, INTG, that takes 4 arguments as input: a number of operations, $numop$, an overhead in seconds, $overhead$, a task start time, a, an upper bound on task finish time, b, and a CPU availability function, f, defined as in Section 2.1, that corresponds to a single availability interval. INTG returns the time at which a task of size $numop$, started at time a on a host whose CPU availability is described by function f, incurring overhead $overhead$ before it can actually start computing, would complete if it would complete before time b, or -1 otherwise. It is assumed that time a lies inside the single availability interval defined by function f. In other words, function INTG returns, if it exists, a time $t < b$ such that $\int_{a+h}^{t} f(x)\, dx = numop$, or -1 if such a t does not exist. We show an implementation of INTG in pseudo-code in Figure 2. Note that the pseudo-code shows a discrete implementation; the loop increments the value of local variable t by one, assuming the step size of the trace's step function is 1 second. Intuitively, function INTG will be used by our scheduling algorithm to see whether a task can actually "fit" inside an availability interval when started somewhere in that interval.

The greedy algorithm OPTINTV given in Figure 3 computes the schedule described informally in Section 3.1. It takes the following parameters:

T: the number of tasks to be scheduled,
h: the overhead for starting a task,
S: the task size in number of operations,
f: the CPU availability function, as defined in Section 2.1,
$[a, b]$: the absolute start and end times of the host availability interval we consider,
A: an array that stores the time at which each task begins, to be filled in,
B: an array that stores the time at which each task completes, to be filled in,

and returns the number of tasks that could not be scheduled in the availability interval, out of the T tasks. From the pseudo-code it is easy to see that OPTINTV schedules tasks from the very beginning of the availability interval, and a

Algorithm 3.1: INTG($numop, overhead, a, b, f$)

```
sum ← 0
for t ← a + h to b
   sum ← sum + f(t)
   if sum = S
      return (t)
return (−1)
```

Figure 2: INTG: helper function for the scheduling algorithm.

task is scheduled immediately after the previous task completes. The duration of each task execution is computed via a call to the INTG helper function.

Algorithm 3.2: OPTINTV(T, h, S, f, a, b, A, B)

```
t ← a
B[1] ← a
for i ← 2 to T + 1
   t ← INTG(S, h, t, b, f)
   if t ≥ 0
      // The time at which task i − 1 completes is the time at which task i is scheduled
      B[i − 1] ← t
      if i < T + 1
         A[i] ← t
   else return (T − (i − 2))
end for
return (0)
```

Figure 3: Scheduling algorithm over a single availability interval.

3.2 Proof of Optimality

Let $[t_1, t_2, t_3, ..., t_T]$ denote the times at which each task begins execution in the schedule computed by the OPTINTV algorithm. Note that t_1 is just the beginning of the availability interval. Let $[e_1, e_2, e_3, ..., e_T]$ be the task execution times without counting the overhead, such that $t_i = t_{i-1} + h + e_{i-1}$, for $2 \leq i < T$.

Consider another schedule obtained by an algorithm, which we call OPTDELAY, that does not start each task as early as possible. In other words, the algorithm adds a time delay, $w_i \geq 0$, before starting task i, for $1 \leq i \leq T$. Let $[t'_1, t'_2, t'_3, ..., t'_T]$ be the times at which tasks start execution and $[e'_1, e'_2, e'_3, ..., e'_T]$ be the task execution times without counting the overhead, in the OPTDELAY schedule. We prove that the OPTDELAY schedule is never better than the OPTINTV schedule, which then guarantees that the OPTINTV schedule is optimal within an availability interval. Let the proposition $P(k)$ be that OPTINTV schedules k tasks optimally. We prove $P(k)$ by induction.

Base case – Let us assume that $P(1)$ does not hold, meaning that $t_1 + h + e_1 > t'_1 + w_1 + h + e'_1$. This situation is depicted in Figure 4. For convenience, let c_1 and c'_1 denote the completion times of the task under the OPTINTV and the OPTDELAY schedules, respectively, so that our assumption is $c_1 > c'_1$.

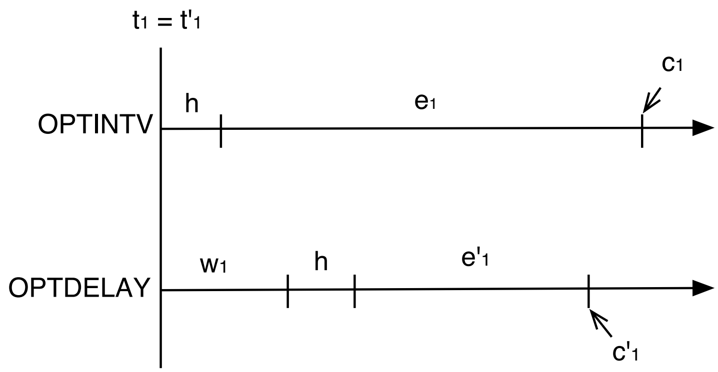

Figure 4: An example of task execution for OPTINTV (higher) and OPTDELAY (lower) at the beginning of the job. Both jobs arrive at the same time. In the case of OPTINTV, the first task is scheduled immediately and an overhead of h is incurred. In the case of OPTDELAY, the scheduler waits of a period of w_1 before scheduling the task.

For the OPTINTV schedule we can write that:

$$S = \int_{t_1+h}^{t_1+h+e_1} f(t)\, dt,$$

which just means that, during its execution, the task consumes exactly the number of operations needed. We can write that:

$$\int_{t_1+h}^{t_1+h+e_1} f(t)\, dt = \int_{t_1+h}^{t_1+h+w_1} f(t)\, dt + \int_{t_1+h+w_1}^{c'_1} f(t)\, dt + \int_{c'_1}^{c_1} f(t)\, dt.$$

First, we note that the second integral in the right-hand side of the above equation is equal to S, since it corresponds to the full computation of the task in the OPTDELAY schedule. Second, we note that the third integral is strictly positive. Indeed, if it were equal to zero, then the number of operations delivered by the host to the task during the $[c'_1, c_1]$ interval would be zero, meaning that no useful computation would be performed on that interval in the OPTINTV schedule. Therefore, in the OPTINTV schedule, the completion time c_1 would in fact be lower or equal to c'_1, which does not agree with our hypothesis. As a result, we have:

$$S = \int_{t_1+h}^{t_1+h+e_1} f(t)\, dt \, > \, S,$$

which is a contradiction. We conclude that $P(1)$ holds.

Inductive case – Let us assume that $P(j)$ holds, and let us prove $P(j+1)$. The execution timeline for both the OPTINTV and OPTDELAY schedule is depicted on Figure 5 with t_{j+1} lower or equal to t'_{j+1} due to $P(j)$. As for the base case, c_{j+1} and c'_{j+1} denote the completion times of task $j+1$ under both schedules.

Suppose $c_{j+1} > c'_{j+1}$. For the OPTINTV schedule, we can write that:

$$S = \int_{t_{j+1}+h}^{c_{j+1}} f(t)\, dt,$$

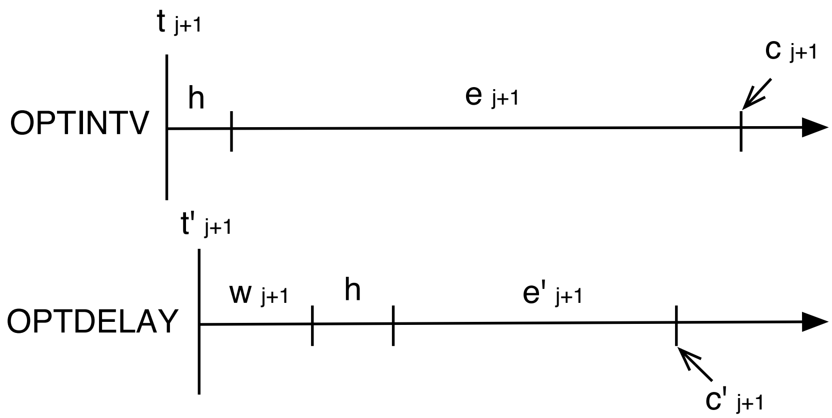

Figure 5: An example of task execution for OPTINTV (higher) and OPTDELAY (lower) in the middle of the job.

which just means that S operations are delivered to the application task during its execution. We can split the above integral as follows:

$$\int_{t_{j+1}+h}^{c_{j+1}} f(t)\,dt = \int_{t_{j+1}+h}^{t'_{j+1}+w_{j+1}+h} f(t)\,dt + \int_{t'_{j+1}+w_{j+1}+h}^{c'_j} f(t)\,dt + \int_{c'_j}^{c_j} f(t)\,dt.$$

The first integral is valid because $w_{j+1} \geq 0$ and $t_{j+1} \leq t'_{j+1}$ (due to property $P(j)$). Following the same argument as in the base case, the last integral in the right-hand side of the above equation is strictly positive (otherwise $c_{j+1} \leq c_j$). The second integral is equal to S, since this is the number of operations delivered to task $j+1$ during its execution in the OPTDELAY schedule. We then obtain that

$$S = \int_{t_{j+1}+h}^{c_{j+1}} > S,$$

which is a contradiction. Therefore $c_{j+1} \leq c'_{j+1}$, and property $P(j+1)$ holds, which completes our proof by induction.

4 Multiple Availability Intervals On A Single Host

In this section, we consider scheduling tasks during multiple intervals of availability, whose start and stop times are denoted by $[a_i, b_i]$. Without loss of generality, we can ignore all availability intervals during which a single task cannot complete, i.e. for which $\int_{a_i}^{b_i} f(t)\,dt < S$. We also consider an infinite number of availability intervals for the host, or at least a number large enough to accommodate all T tasks. The scheduling algorithm OPTMINTV, seen in Figure 6, takes the the following parameters:

T: the number of tasks to be scheduled,
h: the overhead for starting a task,
S: the task size in number of operations,
f: the CPU availability function, as defined in Section 2.1,
C: an array that stores start times of all the tasks, to be filled in,
D: an array that stores completion times of all the tasks, to be filled in.

Let $P(i)$ be the property that OPTMINTV schedules k tasks optimally. We prove that $P(k)$ is true by induction for $k \geq 1$.

Base Case – P(1) is true because OPTMINTV(1,...) is equivalent to running OPTINTV(1,..) for the first availability interval, which we know leads to an optimal schedule from Section 3.

Algorithm 4.1: OPTMINTV(T, h, S, f, C, D)

$numleft \leftarrow T$
$i \leftarrow 1$
while $numleft \neq 0$
  $numleft \leftarrow$ OPTINTV$(numleft, h, S, f, a_i, b_i, A, B)$
  $C \leftarrow$ CONCAT(C, A)
  $D \leftarrow$ CONCAT(D, B)
  $i \leftarrow i + 1$
end while

Figure 6: Scheduling algorithm over multiple availability intervals.

$P(2)$ **–** When there are two tasks to schedule, either both tasks can be scheduled in the first availability interval, or if both tasks cannot finish in the first availability interval, one task must be scheduled in the first interval and the other scheduled in the second. In the former case, OPTMINTV(2,...) is equivalent to OPTINTV(2,...) for the first interval, and the result is optimal as proved in the previous section. In the latter case, a task is scheduled at the beginning of the first and at the beginning of the second interval. This results in the optimal schedule since the earliest the second task can execute (and finish) is at the beginning of the second interval.

Inductive Case – Let us assume that $P(j)$ is true. Then OPTMINTV(j,...) gives the optimal schedule for the first j tasks. If OPTMINTV must schedule another task, OPTMINTV will either place it in the in the same interval as the j^{th} task, or if that is not possible, it will place it at the beginning the following interval. The former case results in an optimal schedule since OPTINTV will schedule the $j+1^{th}$ task optimally in that interval, and the resulting makespan for all $j+1^{th}$ tasks will also be optimal. The latter case results in an optimal schedule since we know that OPTINTV(1,..) is optimal on the second availability interval. Therefore, $P(j+1)$ is true.

It follows that $P(k)$ is true for all $k \geq 1$, and thus OPTMINTV computes an optimal schedule.

5 Multiple Availability Intervals On Multiple Hosts

The algorithm for scheduling tasks across multiple availability intervals over multiple hosts, OPTIMAL, seen in Figure 7, takes the following parameters:

T: the number of tasks to be scheduled;

h: the overhead for starting a task;

S: the task size in number of operations;

v: the arrival time of the job;

E: a $N \times T$ matrix that stores the start times of the T tasks scheduled on each of the N hosts; each row of E is computed by a call to OPTMINTV;

F: a $N \times T$ matrix that stores the completion times of the T tasks scheduled on each of the N hosts; each row of F is computed by a call to OPTMINTV;

and returns the total makespan. We assume that the functions describing CPU availabilities for each host, f_m for $m = 1, ..., N$, are known. OPTIMAL uses a local variable, I, which is a $1 \times N$ array that stores the index of the last completed task for each host. Finally, we define the $argmin$ operator in the classical way for a series, say $\{x_i\}_{i=1,..,n}$, by $x_{argmin(x)} = \min_{i=1,..,n} x_i$.

Since OPTMINTV(k,..) schedules each task optimally, OPTIMAL(N,k,..) selects the k tasks that complete the soonest, resulting in the optimal schedule.

Algorithm 5.1: OPTIMAL(N, T, h, S, v, E, F)

// schedule T tasks on each host and determine each task's completion time

for $i \leftarrow 1$ **to** N
  OPTMINTV(T, h, S, f_i, C, D)
  $E[i] \leftarrow C$
  $F[i] \leftarrow D$
// select the T tasks that completed the soonest

$j \leftarrow T$
while $j > 0$
  $i \leftarrow argmin_{i\epsilon 1,\ldots,N}(F[I[i]])$
  $I[i] \leftarrow I[i] + 1$
  $j \leftarrow j - 1$
return $(M[I[i]] - v)$;

Figure 7: Scheduling algorithm over multiple availability intervals over multiple hosts

6 Optimal Makespan with Checkpointing Enabled

In the section, we consider the scenario where a desktop grid system, such as Condor [8], is able to support task checkpointing and restart. We assume that when a task encounters a failure, it is always restarted on the machine where it began execution, i.e., we do not consider process migration. Since the greedy algorithm that accounts for checkpointing and its proof of optimality are similar to those described in the previous sections, we give only a high-level description of the new optimal scheduling algorithm and proof sketch of its optimality. For our discussion of checkpointing, we define the following new parameters:

p: the overhead in terms of time for checkpointing a task

r: the overhead in terms of time for restarting a task from its checkpoint.

s: the number of operations to be completed before a checkpoint is performed.

We make the following changes to OPTINTV to account for checkpointing. Because checkpointing is enabled, we can view intervals of failures as periods of 0% CPU availability. So, the a host's trace can be treated as single continuous availability interval, on which we can use OPTINTV to schedule tasks. After a task is scheduled and it begins execution, a checkpointing overhead of p is incurred after every s operations are completed. If during execution a task encounters a failure, whatever progress made since the last checkpoint is lost, and an overhead of r is incurred to restart the task from the last checkpoint.

We can reduce the problem of scheduling tasks with checkpointing enabled to the problem of scheduling task without checkpointing as follows. Consider a single task k scheduled within an availability interval, i.e., the task's execution does not encounter any failures. When k is scheduled, it first incurs an overhead of h. Then, after every s operations are complete, an overhead of p is incurred due to checkpointing. So, one can treat the task k as $\lceil S/s \rceil$ subtasks, $\lfloor S/s \rfloor$ of which are of size s and $\lceil S/s \rceil - \lfloor S/s \rfloor$ of which are of size $S - s * \lfloor S/s \rfloor$. The first subtask is scheduled with an overhead of h; thereafter, each subtask is "scheduled" with an overhead of p. If no failures are encountered during task execution, then OPTINTV achieves the optimal schedule by an argument similar to the one used in Section 3, and the same is true if we used OPTVINTV to schedule multiple tasks (treated as batches of subtasks) in the same availability interval.

If a failure is encountered during task execution, then whatever progress made since the last checkpointing is lost, an overhead of r is incurred immediately after the failure, and the task is restarted from the last checkpoint. By the same argument used to prove OPTMINTV in Section 4, OPTINTV would have scheduled subtasks in the previous availability interval optimally, and so starting a subtask at the beginning of the next availability interval (and then incurring an overhead of r before execution) gives an optimal schedule. Finally, we can replace OPTMINTV with

OPTINTV in OPTIMAL since failures are viewed as 0% CPU availability, and the resulting algorithm achieves the optimal schedule over all hosts.

7 Conclusion

We have shown that a greedy algorithm that has full knowledge of future host and CPU availabilities achieves the optimal makespan when scheduling a job with identical and independent tasks on a volatile desktop grid. To the best of our knowledge, previous work has not dealt with the case where CPU availability fluctuates between 0 and 100% and at the same time taken into account host heterogeneity and failures. Note also that although are algorithm achieves the optimal makespan, it does not necessarily achieve optimal execution time, since delaying a task might allow it to encounter periods of higher CPU availability. An interesting extension of this work is to consider the multiple job scenario where minimizing execution time (versus makespan) could be beneficial to system performance.

References

[1] 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.

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

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

[4] G. Fedak, C. Germain, V. Néri, 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.

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

[6] D. Kondo, A. Chien, and H. Casanova. Resource Management for Rapid Application Turnaround on Enterprise Desktop Grids. In *to appear in Proceedings of SuperComputing '04, Pittsburgh, Pennsylvania*, September 2004.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5–6):757–768, 1999.