# Performance Evaluation of Scheduling Policies for Volunteer Computing

Derrick Kondo[1] David P. Anderson[2] John McLeod VII[3]
[1]INRIA, France [2]University of California at Berkeley, U.S.A. [3]Sybase Inc., U.S.A.

## Abstract

*BOINC, a middleware system for volunteer computing, allows hosts to be attached to multiple projects. Each host periodically requests jobs from project servers and executes the jobs. This process involves three interrelated policies: 1) of the runnable jobs on a host, which to execute? 2) when and from what project should a host request more work? 3) what jobs should a server send in response to a given request? 4) How to estimate the remaining runtime of a job? In this paper, we consider several alternatives for each of these policies. Using simulation, we study various combinations of policies, comparing them on the basis of several performance metrics and over a range of parameters such as job length variability, deadline slack, and number of attached projects.*

## 1 Introduction

Volunteer computing is a form of distributed computing in which the general public volunteers processing and storage resources to computing projects. Early volunteer computing projects include the Great Internet Mersenne Prime Search [12] and Distributed.net [1]. BOINC (Berkeley Open Infrastructure for Network Computing) is a middleware system for volunteer computing [6, 3]. BOINC projects are independent; each has its own server, applications, and jobs. Volunteers participate by running BOINC client software on their computers (hosts). Volunteers can attach each host to any set of projects. There are currently about 40 BOINC-based projects and about 400,000 volunteer computers performing an average of over 500 TeraFLOPS.

BOINC software consists of client and server components. The BOINC client runs projects' applications, and performs CPU scheduling (implemented on top of the local operating system's scheduler). All network communication in BOINC is initiated by the client; it periodically requests jobs from the servers of projects to which it is attached. Some hosts have intermittent physical network connections (for example, portable computers or those with modem connections). Such computers may connect only every few days, and BOINC attempts to download enough work to keep the computer busy until the next connection.

This process of getting and executing jobs involves four interrelated policies:

1. CPU scheduling: of the currently runnable jobs, which to run?
2. Work fetch: when to ask a project for more work, which project to ask, and how much work to ask for?
3. Work send: when a project receives a work request, which jobs should it send?
4. Completion time estimation: how to estimate the remaining CPU time of a job?

These policies have a large impact on the performance of BOINC-based projects. Because studying these policies in situ is difficult, we have developed a simulator that models a BOINC client and a set of projects. We have used this simulator to compare several combinations of scheduling policies, over a range of parameters such as job-length variability, deadline slack, and number of attached projects. With the simulator, our main goals are to show the effectiveness of scheduling policies in scenarios representative of current project workloads, and also hypothetical extreme scenarios that reflect the scheduler's ability to deal with new future job workloads.

The remainder of the paper is organized as follows. In Section 2, we describe various scheduling policies in detail. Then in Section 3, we detail the simulator and its input parameters. In Section 4, we describe our metrics for evaluating the different aspects of each scheduling policy. In Sections 5 and 6, we present the results of our evaluation and deployment. In Section 7, we compare and contrast our work with other related work. Finally, in Section 8, we summarize the main conclusions of this paper.

## 2 Scheduling Policy Details

The scheduling policies have several inputs. First, there are the host's hardware characteristics, such as number of processors and benchmarks. The client tracks various usage characteristics, such as the active fraction (the fraction of time BOINC is running and is allowed to do computation), and the statistics of its network connectivity.

Second, there are user preferences. These include:

- A resource share for each project.
- Limits on processor usage: whether to compute while the computer is in use, the maximum number of CPUs to use, and the maximum fraction of CPU time to use (to allow users to reduce CPU heat).
- *ConnectionInterval*: the minimum time between periods of network activity. This lets users provide a "hint"

about how often they connect, and it lets modem users tell BOINC how often they want it to automatically connect.
- *SchedulingInterval*: the "time slice" of the BOINC client CPU scheduler (the default is one hour).
- *WorkBufMinDays*: how much work to buffer.
- *WorkBufAdditional*: how much work to buffer beyond WorkBufMinDays.

Finally, each job (that is, unit of work) has a number of project-specified parameters, including estimates of its number of floating-point operations, and a deadline by which it should be reported. Most BOINC projects use replicated computing, in which two or more instances of each job are processed on different hosts. If a host doesn't return a job by its deadline, the user is unlikely to receive credit for the job. We will study two or three variants of each policy, as described below.

### 2.1 CPU Scheduling Policies

**CS1**: round-robin time-slicing between projects, weighted according to their resource share.

**CS2**: do a simulation of weighted round-robin (RR) scheduling, identifying jobs that will miss their deadlines. Schedule such jobs earliest deadline first (EDF). If there are remaining CPUs, schedule other jobs using weighted round-robin.

### 2.2 Work fetch policies

**WF1**: keep enough work queued to last for WorkBufMinDays+WorkBufAdditional, and divide this queue between projects based on resource share, with each project always having at least one job running or queued.

**WF2**: maintain the "long-term debt" of work owed to each project. Use simulation of weighted round-robin to estimate its CPU shortfall. Fetch work from the project for which debt - shortfall is greatest. Avoid fetching work repeatedly from projects with tight deadlines. The details of this policy are given in [14].

### 2.3 Work Send Policies

**WS1**: given a request for X seconds of work, send a set of jobs whose estimated run times (based on FLOPS estimates, benchmarks, etc.) are at least X.

**WS2**: the request message includes a list of all jobs in progress on the host, including their deadlines and completion times. For each "candidate" job J, do an EDF simulation of the current workload with J added. Send J only if it meets its deadline, all jobs that currently meet their deadlines continue to do so, and all jobs that miss their deadlines don't miss them by more.

### 2.4 Job Completion Estimation Policies

**JC1**: BOINC applications report their fraction done periodically during execution. The reliability of an estimate based on fraction done presumably increases as the job progresses. Hence, for jobs in progress BOINC uses the estimate $FA + (1 - F)B$ where $F$ is the fraction done, $A$ is the estimate based on elapsed CPU time and fraction done, and $B$ is the estimate based on benchmarks, floating-point count, user preferences, and CPU efficiency (the average ratio of CPU time to wall time for this project).

**JC2**: maintain a per-project duration correction factor (**DCF**), an estimate of the ratio of actual CPU time to originally estimated CPU time. This is calculated in a conservative way; increases are reflected immediately, but decreases are exponentially smoothed.

An "overall" policy is a choice of each sub-policy (for example, CS2_WF2_WS2_JC2).

## 3 Scheduling Scenarios

We evaluate these scheduling policies using a simulator of the BOINC client. The simulator simulates the CPU scheduling logic and work-fetch policies of the client exactly down to the source code itself. This was made possible by refactoring the BOINC client source code such that the scheduler code is cleanly separated from the code required for networking and memory access. As such, the simulator links to the real BOINC scheduling code with little modification. In fact, only the code for network accesses (i.e., RPC's to communicate with the project and data servers) are substituted by simulation stubs. Thus, the scheduling logic and almost all the source code of the simulated client are identical to the real one.

The simulator allows specification of the following parameters:

- For a host:
  - Number of CPUs and CPU speed.
  - Fraction of time available
  - Length of availability intervals as an exponential distribution with parameter $\lambda_{avail}$
  - ConnectionInterval
- For a client:
  - CPU scheduling interval
  - WorkBufMinDays, WorkBufAdditional
- For each project:
  - Resource share
  - Latency bound (i.e., a job deadline)
  - Job FLOPs estimate

| Parameter | Value |
|---|---|
| duration | 100 days |
| delta | 60 seconds |

**Table 1. Simulator Parameters.**

| Parameter | Value |
|---|---|
| resource share | 100 |
| latency bound | 9 days |
| job FPOPS estimate | 1.3e13 (3.67 hours on a dedicated 3GHz host) |
| job FPOPS mean | 1.3e13 (3.67 hours) |
| FPOPS std dev | 0 |

**Table 2. Project Parameters.**

– Job FLOPs actual distribution (normal)

Each combination of parameters is called a scenario.

When conducting the simulation experiments, we use the values shown in Tables 1,2,3, and 4 as the base input parameters to the simulator. In our simulation experiments, we vary one or a small subset of these parameter settings as we keep the other parameter values constant. This is useful for identifying the (extreme) situations where a particular scheduling policy performs poorly.

The values shown in these tables are chosen to be as realistic as possible using data collected from real projects, as the accuracy of the simulation depends heavily on these values and their proportions relative to one another. For the host and project parameters, we use the median values found from real BOINC projects as described in [7, 2]. For the client parameters, we use the default values in the real client.

| Parameter | Value |
|---|---|
| number of cpus | 2 |
| dedicated cpu speed in FPOPS | 1e9 |
| fraction of time available | 0.8 |
| $\lambda_{avail}$ | 1,000 |
| ConnectionInterval | 0 |

**Table 3. Host Parameters.**

| Parameter | Value |
|---|---|
| CPU scheduling period | 60 minutes |
| WorkBufMinDays | 0.1 days (2.4 hours) ) |
| WorkBufAdditional | 0.25 days (6 hours) |

**Table 4. Client Parameters.**

## 4 Performance metrics

We use the following metrics in evaluating scheduling policies:

1. **Idleness**: of the total available CPU time, the fraction that is unused because there are no runnable jobs. This is in [0,1], and is ideally zero.
2. **Waste**: of the CPU time that is used, the fraction used by jobs that miss their deadline.
3. **Share violation**: a measure of how closely the user's resource shares are respected, where 0 is best and 1 is worst.
4. **Monotony**: an inverse measure of how often the host switches between projects: 0 is as often as possible (given the scheduling interval and other factors) and 1 means there is no switching. This metric represents the fact that if a volunteer has devoted equal resource share to two projects, and sees his computer running only one of them for a long period, he will be dissatisfied.

## 5 Results and Discussion

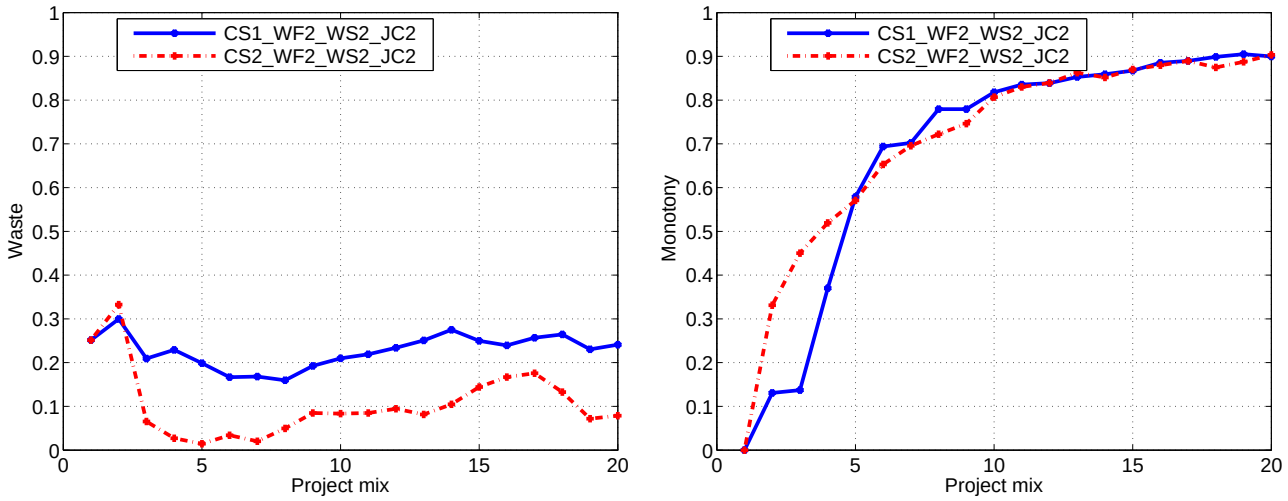
### 5.1 CPU scheduling policies

In this section, we compare a policy that only uses round-robin time-slicing (**CS1_WF2_WS2_JC2**) with a policy that will potentially schedule (some) tasks in EDF fashion if their deadlines are at risk of being missed (**CS2_WF2_WS2_JC2**).

To simulate a real workload of projects, we must ensure that there are a mix of projects with a variety of job sizes and deadlines. So, we define the term *mix* as follows. A workload has mix $M$ if it contains $M$ projects, and the client is registered with each project $P_i$ where $1 \leq i \leq M$, and each project $P_i$ has the following characteristics:

1. mean FPOPS per job = $i \times$ FPOPS base
2. latency bound = $i \times$ latency bound base
3. resource share = $i \times$ resource share base

where the base values are those values defined in Tables 1, 3, and 4.




(a) Waste

(b) Project Monotony

**Figure 1. CPU Scheduling**

We vary $M$ from 1 to 20 for each simulation workload. The results are shown in Figure 1. We observe that CS2_WF2_WS2_JC2 often outperforms CS1_WF2_WS2_JC2 by more than 10% in terms of waste for when the project mix is greater than 3. The improved performance of CS2_WF2_WS2_JC2 can be explained by the switch to EDF mode when job deadlines are in jeopardy.

In terms of idleness, share violation, and project monotony, the performance of CS2_WF2_WS2_JC2 and CS1_WF2_WS2_JC2 are roughly equivalent. Share violation is low, usually below 0.10, and increases only slightly with the number projects and mix . With respect to idleness, both policies have remarkably low (near zero) idleness for any number of projects and mix . By contrast, project monotony tends to increase greatly with the project mix. We find that relatively long jobs can often cause monotony, particularly when the job is at risk of missing a deadline and the scheduler switches to EDF mode to help ensure its timely completion. Indeed, an increase in monotony with job length is inevitable but acceptable, given the more important goal of meeting job deadlines and to minimize waste. Thus, in almost all cases, the policy CS2_WF2_WS2_JC2 combining EDF with RR outperforms or performs as well as CS1_WF2_WS2_JC2.

As a side effect of these experiments, we determine the scalability of the BOINC client with respect to the number of projects. (One could ask how a user could register with 20 projects. This scenario could very well arise as the number of projects is rapidly increasing, and the idea of project "mutual funds" has been proposed, where projects are group together by certain characteristics [for example, non-profit, computational biology], and users register for funds instead of only projects.)

We find that even when the number of projects is high at 20, waste, idleness, and share violation remain relatively constant throughout the range between 1-20. The exception is with project monotony which increases rapidly but is acceptable. Thus, we conclude the scheduler scales with the number of projects. For the remainder of this subsection, we investigate further the CS2_WF2_WS2_JC2 policy with respect to deadline slack and buffer size.

### 5.1.1 Deadline slack

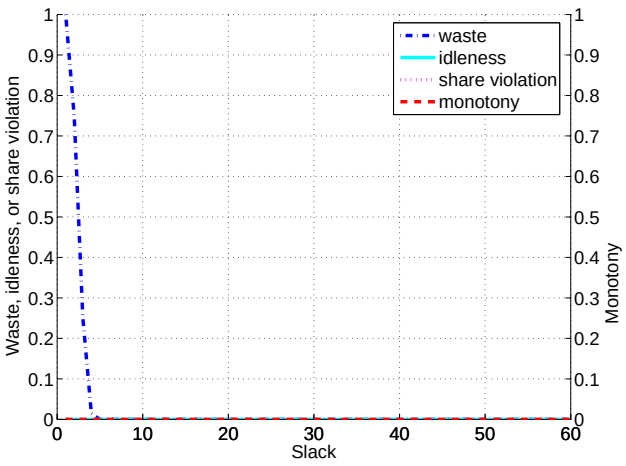



**Figure 2. Slack**

In this section, we investigate the sensitivity of

CS2_WF2_WS2_JC2 to the latency bound relative to the mean job execution time on a dedicated machine. We define the **slack** to be the ratio of the latency bound to the mean FPOPS estimate multiplied by the inverse of the host speed.

We then run simulations testing the scheduler with single project and a range of slack values from 1 to 60. A slack of 1 means that the latency bound is roughly equivalent to execution time. A slack of 60 means that the latency bound is 60 times greater than the mean execution time on a dedicated machine. We select this maximum value because it in fact represents the median slack among all existing BOINC projects [7]. That is, the execution of jobs in many projects is on the order of hours, where as the latency bound is in terms of days.

Figure 2 shows the result of the simulation runs testing slack. Obviously, the share violation and monotony are 0 since we only have a single project. The idleness is 0 because of the low connection interval and so the scheduler does a perfect job of requesting work. However, it tends to fetch more than it can compute. We find that for a slack of 1, waste is 1, and quickly drops to 0 when slack reaches 5. The waste corresponding to slack values of 2, 3, 4, and 5 are 0.75, 0.24, 0.02, and 0 respectively. Thereafter, for slack values greater than or equal to 5, waste remains at 0. After careful inspection of our simulation experiments, we find that the unavailability of the host (set to be 20% of the time) is one main factor that causes the high waste value when the slack is near 1. That is, if a host is unavailable 20% of the time, then a 3.67-hour job on a dedicated machine will take about 4.58 hours on average and likely surpass the 3.67 hour latency bound.

Another cause of the waste is the total work buffer size, which is set at a default of 0.35 ($= 0.25 + 0.1$) days. When the work buffer size is specified, the client ensures that it retrieves at least that amount of work with each request. Setting a value that is too high can result in retrieving too much work that cannot be completed in time by the deadline. We investigate this issue further the following section.

#### 5.1.2 Buffer size

To support our hypothesis that the buffer size is a major factor that contributes to waste, we ran simulations with different buffer sizes. In particular, we determined how waste varies as function of the buffer size and slack. We varied slack from 1 to 5, which corresponded to the range of slack values in Figure 2 that resulted positive waste (except when slack was 5, where the waste was 0). We varied the buffer size between about 1 to 8.4 hours of work. The maximum value in that range corresponds to the default buffer size used by the client, i.e. 0.35 days, and used for the experiments with slack in Section 5.1.1.

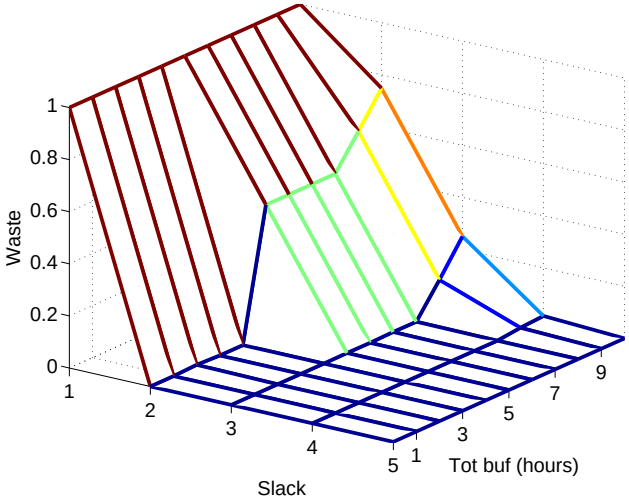



**Figure 3. Waste as a function of minimum buffer size and slack**

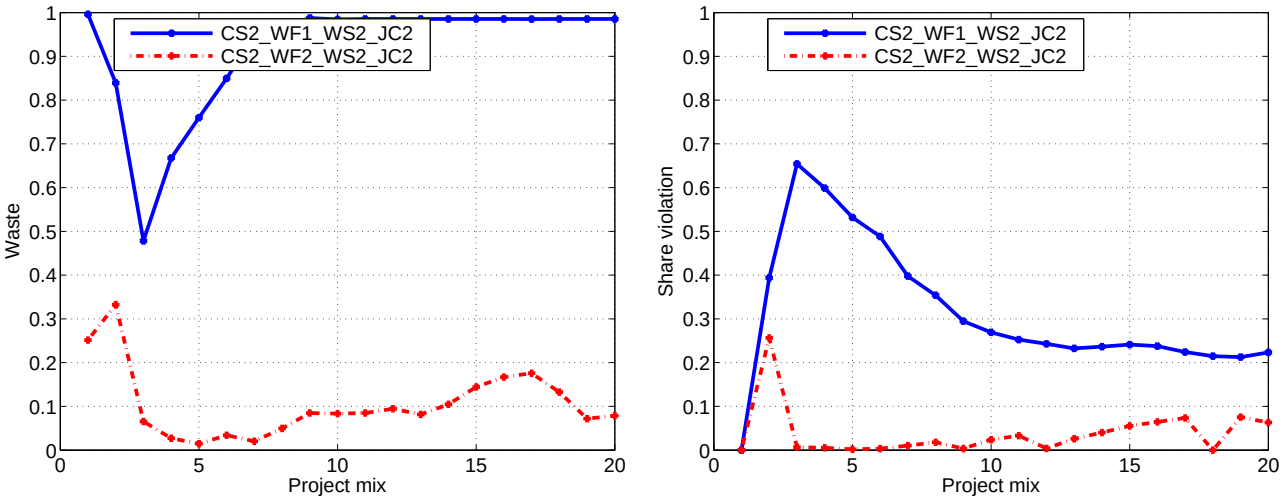
Figure 3 shows the result of our simulation runs. The differences in color of each line segment correspond to a different value of waste. When the buffer size is at its maximum value, the resulting waste as a function of slack is identical to the waste seen in Figure 2. As the buffer size is lowered, the waste (and number of deadlines missed) dramatically decreases. For the case where slack is 2, waste is constant for buffer sizes between 5 to 7 hours. The reason this occurs is simply because of job sizes are about 3.67 hours in length, and so incremental increases in the buffer size may not change the number of jobs downloaded from the server.

While decreasing the buffer size can also decrease waste, the exception is when the slack is 1; waste remains at 1 regardless of the buffer size. This is due to the host unavailability and its effect on job execution time. It has a lesser effect when slack reaches 2 since the host is unavailable only 20% of the time, and corresponds to a slowdown less than 2.

We conclude that decreasing the work buffer size can result in tremendous reduction in waste, but the benefits are limited by the availability of the client. However, one cannot set the work buffer size arbitrarily low since it could result in frequent flooding of the server with work requests from clients. We will investigate this interesting issue in future work.

### 5.2 Work Fetch Policies

In this section, we compare two ways of determining which project to ask for more work, and how much work to ask for. When forming a work request, the first policy




(a) Waste (b) Share violation

**Figure 4. Work fetch**

(**CS2_WF1_WS2_JC2**) tries to ensure that the work queue is filled by asking for work based on the resource shares so that at least one job is running or queued. The second policy (**CS2_WF2_WS2_JC2**) determines how much "long-term debt" is owed to each project, and fetches work from the project with the largest difference between the debt and shortfall. (Details of each policy can be found in [14]).)

We find that waste for CS2_WF1_WS2_JC2 tends to increase dramatically with the project mix (see Figure 4(a)), and that the share violation is relatively high (see Figure 4(b)) as jobs for projects with tight deadlines are downloaded, and preempt other less urgent jobs, causing them to miss their deadlines. By contrast, CS2_WF2_WS2_JC2 improves waste by as much as 90% and share violation by as much as 50% because the policy avoids fetching work from projects with tight deadlines. (Idleness and monotony for the two policies were almost identical.)

### 5.3 Work Send Policies

In this section we compare two policies for sending work from the server. The first policy (**CS2_WF2_WS1_JC2**) simply sends the amount of work requested by the client. The second policy (**CS2_WF2_WS2_JC2**) checks via server-side online EDF simulation if the newly requested jobs would meet their deadlines, if all jobs in progress on the client would continue to meet their deadlines, and if all jobs in progress on the client that miss their deadlines don't miss them by more.

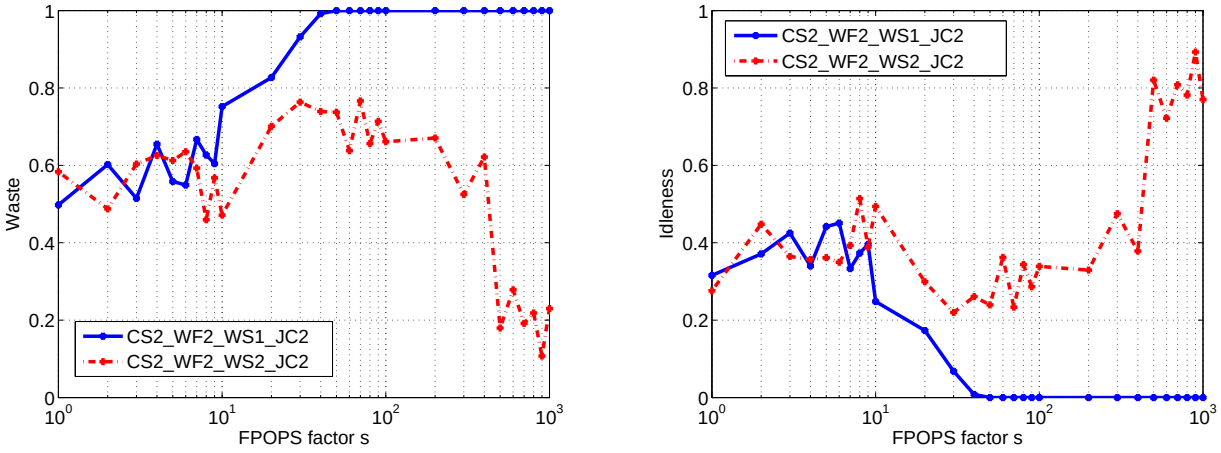
We found that the second work send policy CS2_WF2_WS2_JC2 is advantageous only when there are relatively many jobs on the client (having a large work queue buffer), and if the amount of work per job has high variance. This would create the "ideal" situation where new jobs could disrupt jobs in-progress. So to show the effectiveness of work send policies, we increased the connection interval and work buffer size to 2 days, varied the standard deviation of the FPOPS per job by a factor of $s$ relative to the mean job size, and created two projects with equal resources shares and job sizes that differed by a factor of 10.

In Figure 5, we vary the standard deviation of the job size by a factor $s$ between 1 to 1000. We find that CS2_WF2_WS2_JC2 often outperforms CS2_WF2_WS1_JC2 by 20% or more in terms of waste, at the cost of idleness, which is often higher by 20% or more. The main reason for this result is that CS2_WF2_WS2_JC2 is naturally more conservative in sendings job to a client, which reduces deadline misses but in turn causes more frequent idleness.

### 5.4 Job Completion Estimation Policies

To create a project and when forming jobs, the application developer must provide an estimate of the amount of work in FPOPS per job. This estimate in turn is used by the client scheduler along with the job deadline to form work requests and to determine which job to schedule next. However, user estimates are often notoriously off from the actual amount of work. This has been evident in both volunteer computing systems such as BOINC and also with the usage of traditional MPP's [13]. As such, the scheduler uses a duration correction factor to offset any error in estimation based on the history of executed jobs.

In this section, we measure the impact of using DCF for a range of estimation errors by comparing




(a) Waste (b) Idleness

**Figure 5. Work send**

**CS2_WF2_WS2_JC1** with **CS2_WF2_WS2_JC2**.

Let $f$ be the *error factor* such that the user estimated work per job is given by the actual work divided by $f$. When $f = 1$, the user estimate matches the actual precisely. When $f > 1$, then the user estimate under-estimates the amount of real work.

We observe that initially when the estimate matches the actual, then both policies with DCF (when it is adjusted to offset any error) and without DCF (when it is always set to 1) have identical performance in terms of waste (see Figure 6). For higher error factors, the policy with DCF greatly outperforms the policy that omits it. The waste for DCF remains remarkably constant regardless of the error factor while the waste for the non-DCF policy shoots up quickly. In fact, the waste reaches nearly 1 when the work estimate is 4 times less than the actual. With respect to the speed of convergence to the correct DCF, we found that the DCF converges immediately to the correct value after the initial work fetch, after inspecting our simulation traces. (For conciseness, note that we only show the figure for waste; all other values for idleness, share violation, and project monotony were zero.)

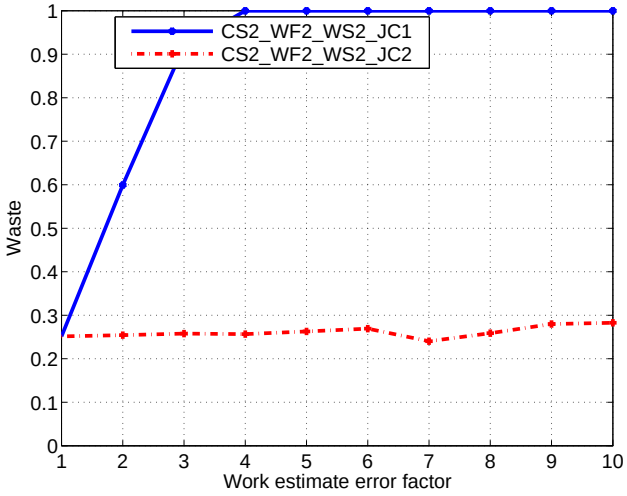



**Figure 6. Job Completion**

## 6 Deployment

The client scheduler has been implemented in C++ and deployed across about one million hosts over the Internet [6, 3]. One of the primary metrics for performance in the real-world settings is feedback from users about the scheduling policies. Users have a number of ways of directly or indirectly monitoring the client scheduling, including the ability to log the activities of the client scheduler with a simple debug flag, or the amount of credit granted to them. (In general, projects will only grant credit to work given before deadlines.) Since the introduction of policies corresponding to CS2_WF2_WS2_JC2, the number of complaints sent to the BOINC developers has been reduced to about one per month. These complaints are due to mainly ideological differences in how the BOINC scheduler should work, and not relevant specific to the performance metrics as defined in Section 4.

## 7 Related Work

Two of this paper's authors recently described in-depth the same scheduling policies in [14]. However, there was little performance evaluation in that study. Outside of that

work, computational grids [10, 11] are the closest body of research related to volunteer computing, and scheduling on volatile and heterogeneous has been investigated intensely in the area of grid research [4, 5, 8, 9]. However, there are a number of reasons these strategies are inapplicable to volunteer computing. First, the scheduling algorithms and policies often assume that work can be pushed to the resources themselves, which in volunteer computing is precluded by firewalls and NAT's. The implication is that a schedule determined by a policy for grid environments cannot be successfully deployed as the resources are unreachable from the server directly. Second, volunteer computing systems are several factors (and often an order of magnitude) more volatile and heterogeneous than grid systems, and thus often require a new set of strategies for resource management and scheduling. Finally, to the best of our knowledge, the work presented here is the first instance of where scheduling for volunteer computing is done locally on the resources themselves, and where the scheduler is evaluated with a new set of metrics, namely waste due to deadline misses, idleness, share violation, and monotony.

## 8 Conclusion

The following are the main conclusions from our simulation results and analysis:

1. *CPU scheduling:* EDF simulation often results in a 10% performance improvement by detecting and avoiding potential deadline misses early on.
2. *Work fetch:* Fetching work based on long-term debt and CPU shortfall is better than fetching work proportionally according to resource shares. The improvement using the former policy improves waste by as much as 90% and share violation by as much as 50% by avoiding work fetch from projects with tight deadlines.
3. *Work send:* Server-side simulation to determine the impact of new jobs on an existing client workload is critical when job sizes have high variability (a standard deviation greater than 10 times the mean amount of work per job), when clients have a large work buffer, and when clients connect relatively infrequently.
4. *Job completion time estimation:* When the user estimate work per job is off by a factor of 2 or greater, maintaining a duration correction factor to compensate for this difference is essential. Otherwise, waste often reaches nearly 1.

## References


[1] Distributed.net. `www.distributed.net`.

[2] XtremLab. `http://xtremlab.lri.fr`.

[3] D. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, 2004.

[4] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003.

[5] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-Level Scheduling on Distributed Heterogeneous Networks. In *Proc. of Supercomputing'96, Pittsburgh*, 1996.

[6] The berkeley open infrastructure for network computing. `http://boinc.berkeley.edu/`.

[7] Catalog of boinc projects. `http://boinc-wiki.ath.cx/index.php?title=Catalog_of_BOINC_Powered_Proje%cts`.

[8] 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.

[9] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of SuperComputing 2000 (SC'00)*, Nov. 2000.

[10] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 2001. to appear.

[11] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., San Francisco, USA, 1999.

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

[13] C.B. Lee, Y. Schartzman, J.Hardy, and A.Snavely. Are user runtime estimates inherently inaccurate? In *10th Workshop on Job Scheduling Strategies for Parallel Processing*, 2004.

[14] J. McLeod VII and D. P. Anderson. Local Scheduling for Volunteer Computing. In *Workshop on Large-Scale and Volatile Desktop Grids*, March 2007.