An Adaptive Priority Tuning System for Optimized Local CPU Scheduling using BOINC Clients

Adel B. Mnaouer
College of Computer Engineering and Information Technology
Dar Al-Uloom University
Email: adelm@dau.edu.sa

Colin Ragoonath
Dept. of Information and Communication Technology
University of Trinidad and Tobago
Email: colinfagan@gmail.com

Abstract. Volunteer Computing (VC) is a Distributed Computing model which utilizes idle CPU cycles from computing resources donated by volunteers who are connected through the Internet to form a very large-scale, loosely coupled High Performance Computing environment. Distributed Volunteer Computing environments such as the BOINC framework is concerned mainly with the efficient scheduling of the available resources to the applications which require them. The BOINC framework thus contains a number of scheduling policies/algorithms both on the server-side and on the client which work together to maximize the available resources and to provide a degree of QoS in an environment which is highly volatile. This paper focuses on the BOINC client and introduces an adaptive priority tuning client side middleware application which improves the execution times of Work Units (WUs) while maintaining an acceptable Maximum Response Time (MRT) for the end user. We have conducted extensive experimentation of the proposed system and the results show clear speedup of BOINC applications using our optimized middleware as opposed to running using the original BOINC client.

1. Introduction
Internet-based volunteer computing (VC) environments are distributed client-server computing frameworks which seek to provide a means where participants (volunteers) wishing to contribute the idle computing resources of their computers can be facilitated. BOINC (Berkeley Open Infrastructure for Network Computing) [1], [6] is one such distributed computing environment and is used by many research projects in molecular biology, medicine, physics, chemistry, mathematics, astronomy [4] and many other fields of study requiring enormous amounts of computing resources to solve large problems using parallel processing techniques.

BOINC is the classic client-server distributed environment, with each of these components containing a set of scheduling policies/algorithms which enable the system to function in a cohesive manner despite the many drawbacks of operating in such a large, distributed, open environment. In these distributed heterogeneous parallel computing systems, computing resources vary with different
levels of availability and reliability, also projects tend to vary in terms of job length, sensitivity to errors in the results and in result verification requirements (BOINC uses replication of results for comparison before final acceptance) [8].

Most of the previous research into optimizing the BOINC system is focused on job scheduling within the BOINC framework and neglects external factors which may contribute positively to the speedup of jobs within the system. BOINC runs on top of an underlying Operating System and there are advantages which could be gained by improving the interaction between BOINC and the underlying system. Currently BOINC jobs are run at the lowest priority from the perspective of the Operating System (OS), and even if BOINC applications are run in the highest priority within the BOINC framework, at the OS level those jobs are at the lowest priority.

It is understood that the creators of the BOINC system never intended for BOINC processes to actively compete for computing resources with the local processes, but to use the idle resources available to it. But this concept leaves the BOINC processes at a disadvantage especially when the user's preferences are set to lower values for resource usage. The current BOINC client is static (See ‘figure 1’) and does not adapt to the constantly changing nature of the system.

If a user sets usage for the BOINC system to 50% CPU time the BOINC system would constantly request 50% CPU time even if the CPU is underutilized and BOINC is in danger of missing the deadline set by the project. This is not a very efficient way of utilizing the donated computing resources and our Adaptive Priority Tuning system seeks to address this issue resulting in a more dynamic client (See ‘figure 2’) which improves the Quality of Service (QoS) to the projects that use this platform.

This paper is organized as follows: Section 2 describes work which relates to the work done in this paper. Section 3 presents a short overview of how the BOINC client works. The Local Optimization Middleware is described in Section 4. In Section 5 we describe the experiment design used to test the Optimization Middleware. Section 6 is where our results are presented and discussed. In Section 7 we conclude this paper.

2. Related Works
BOINC server-side scheduling have been studied by a number of researchers in [9] the authors use a method based on a distributed, evolutionary algorithm capable of searching a large space of scheduling policies looking for policies that outperform manually-designed scheduling policies. In [5] the authors increase throughput for BOINC projects by intelligently coordinating schedules for volunteers. They achieve this, by developing a constraint optimization (COP) mapping that pursues high throughput while trying to adhere to the volunteer’s preferences.
Research into the operation of the BOINC client is focused mainly on its work fetch policy and its CPU scheduling policy, in [7] the authors evaluate a combination of CPU Scheduling policies, Work Fetch policies, Work Send policies and Job Completion Estimation policies to determine the performance of the local client with a new set of metrics i.e. waste due to deadline misses, idleness, share violation and monotony. In [3] the authors describes the issues involved in local scheduling for volunteer computing, and they present policies that have proven to work well in the real world. They also pointed out that client and server scheduling are not well integrated but plan to remedy this by having the client send information about queued and in-progress work, including completion time estimates. The server will then use this information to do a deadline-scheduling based simulation to decide what jobs, if any, can safely be sent.

3. How the BOINC client works
The BOINC architecture consists of the BOINC client and the BOINC server components. The BOINC client is responsible for running the application for the particular project/s to which the host is attached. These applications are linked with a runtime system whose functions include process control, checkpoint control and graphics [2]. The client performs CPU scheduling for BOINC jobs (performed on top of the Operating System's scheduler, these jobs are run at the lowest (Linux) nice value (-19)). This point is very important to this research since our algorithm seeks to dynamically adjust this value to the ever changing environment which exists within a host system.

The BOINC client is also responsible for all network communication between the host system and the project server. The client communicates with the project server via HTTP. The requests are sent as an XML document and includes a description of the host hardware and availability, a list of completed jobs, and a request for a certain amount (expressed in CPU time) of additional work. The reply message includes a list of new jobs (each described by an XML element which lists the application, input and out files, including a set of data servers from which each file can be downloaded) [3].

Volunteers install the BOINC client software onto their system and attach to the projects of their interest using the client. Once installed on the system the client transparently contacts the project server to request work to process (Work Units WU). The BOINC client is responsible for a number of important tasks these include:

- Network communication with scheduling and data servers.
- Executes and monitors applications and enforces user preferences.
- Implements a Local Scheduling Policy to achieve the following goals:
To maximize resource usage (i.e. keep all processors busy).
- To satisfy result deadlines.
- To respect the participant's resource share allocation among projects.
- To maintain a minimal "variety" among projects.

- Handles request to provide graphics and provide heartbeat functionality.
- Implements a scheduling policy based on a dynamic "resource debt" to each project [4].

Some hosts have intermittent physical network connections (for example, portable computers or those with modem connections). Such computers may connect only every few days. During a period of network connection, BOINC attempts to download enough work to keep the computer busy until the next connection [3].

The BOINC client local scheduling
The client is responsible for implementing two related scheduling policies:

Work Fetch Policy: this policy is responsible for deciding when to ask a project for more work, which of the projects to ask and how much work to ask for [3].

CPU Scheduling Policy: this policy deals with deciding which of the currently runnable jobs should be run. And for the jobs which are preempted, which of these should be kept in memory [3].

As mentioned earlier both of these policies are enforced within the BOINC client but whatever decision is made e.g. if it is decided that a job X should be run at the highest priority by the client that decision does not translate into job X actually being run at a higher priority by the OS. Our Adaptive Priority Tuning mechanism seeks to address this issue in a reasonable manner while maintaining a Maximum Response Time (MRT) which is acceptable to the end user.

4. The Optimization Middleware Algorithm
In this section we describe the Optimization Algorithm with the aid of pseudo code, for simplicity we have decomposed the algorithm into four sections each of which is responsible for particular tasks being completed. Our algorithm is a simple algorithm which seeks to maintain an acceptable MRT for the user while providing quality processor time to the BOINC applications.

BOINC was designed with a number of entry points into the core-client and the algorithm takes advantage of this fact to make the client more dynamic and to adjust to the changes in its environment.

On startup the algorithm determines this is the first time it is running (since the core-client can run without our middleware present) if it’s the first run a copy of the global_prefs_override.xml file is made. This copy would be used as a reference (since it contains the user's original preferences) for the middleware in its decision making process since the original would be constantly changing as processing proceeds and the original settings of the user, would be lost. Both the original (now dynamic and is required by BOINC) and the copy (static and is required by the middleware) would then be read into memory upon further iterations of the algorithm (see ‘Table 1’ below).

During the beginning of the optimize() function the middleware determines if any BOINC applications are currently running. If no BOINC applications are running the middleware exits, however if there are BOINC applications currently running the middleware determines some key user preference settings and system state such as whether the BOINC client should continue processing if the system is currently running on batteries. If the system is running on batteries then the middleware updates the global_prefs_override.xml file and then forces the BOINC client to reread the file by calling its --read_global_prefs_override() function so that the new settings can take effect, which is to set the percent CPU usage to one quarter of what the user settings contain. The middleware also renice the nice values under which the running BOINC applications are being executed to the lowest setting (in Linux 19), these settings are meant to extend the battery life of the computer while still getting some work done. This process is completed in the doUpdate() function ‘Table1’ above, the middleware then sleeps for 4 seconds which is important to meet the MRT criteria.
Table 1: Algorithm 1 – Optimized Middleware onStartup() function

```python
on startup()
if This is first run then
    Make a copy of the global prefs override.xml file
    Read the original and the copy of global_prefs_override.xml files and store in variables
    SET the firstrun indicator variable to 1 {indicates firstrun has occurred}
else
    Read original and copy of global_prefs_override.xml files and store in variables
end if
```

If the computer is not running on batteries then the optimize() function gathers system information such as the current CPU idle time, the user preference settings for CPU usage and whether the BOINC

Table 2: Algorithm 2 – Optimized Middleware optimize() function

```python
if BOINC not running then
    Quit Program
end if
else BOINC is running
while true do
    if computer is on battery AND user preference is set to run on battery AND to run while user is active then
        UPDATE values in a new global_prefs_override.xml file
        SET the nice value variable to the lowest nice level(19)
        DO an update() {see below for details}
        sleep for 4 seconds {4 seconds is chosen to meet the MRT criteria}
    end if
    if computer is not on battery AND user preference is set to run while user is active then
        get CPUidletime(), user preferences and the status {indicates settings of the last iteration of the algorithm}
        then
            SELECT appropriate optimization logic – {Algorithm 3}
            SET values in a new global_prefs_override.xml file
            SET the nice value variable based on – {Algorithm 3}
            SET the new status variable value {this indicates what was previously done}
            DO an update() {see below for details}
            sleep for 4 seconds {4 seconds is chosen to meet the MRT criteria}
        end if
    end while
```
client should run while the user is active, and the status (which indicates what was done before) see ‘Table 3’ for details. Once these pieces of information have been gathered the optimize() algorithm then selects an appropriate optimization logic, see ‘Table 3’ for further details.

**Table 3: Algorithm 3 - Optimized Middleware example of Optimization Logic**

```plaintext
if user preference for CPU usage is \( \geq X \) AND idle time of CPU is \( \leq Y \) AND status == Z
\{Y chosen after several observations and experiments to determine an optimized value to release resources to other programs\}
then
SET the cpu usage limit to random value within a specified range for these values
SET new nice value
UPDATE status variable
end if
```

In the optimization logic algorithm ‘Table 3’:
- **X** is the threshold value for CPU usage set by the end user and can have a value between 1% and 100%.
- **Y** is the threshold value for the current idle time of the CPU and its value ranges as follows:
  - Less than 17% CPU idle time
  - Between greater than 18% and less than and equal to 40% CPU idle time
  - Greater than 40% CPU idle time
- **Z** indicates what was done before by the system, this is important since this value is used to maintain the MRT for the end user. It’s can be either:
  - 0: indicates the middleware used is least aggressive algorithm for the previous iteration taking into account the amount of free resources available.
  - 1: indicates the middleware used a medium aggressive algorithm for the previous iteration again taking into account the amount of free resources available.
  - 2: indicates the middleware used the most aggressive algorithm for the previous iteration taking into account the available resources.

The optimization logic algorithm compares the user preference for CPU usage to a predetermined value (we decided to set these values after several experimental observations and selected the performance we determined to be optimum for our experiments), it also looks at the current idle time of the CPU and what was done during the last iteration (status). Based on these values **X, Y, and Z** the middleware sets the CPU usage limit variable to a random value which is generated by the system but falls within a specified range for the user preferred CPU usage and current idle time of the CPU for these values. It then writes this new value for the CPU usage to the global_prefs_override.xml file for the BOINC client to be updated with this new value during the call to the doUpdate() function, it also determines a new value for the nice local variable and the status local variable according to the algorithm logic. The optimize() algorithm then calls the doUpdate() function.

The doUpdate() function is an update process which simply makes a call to the core-client’s --read_global_prefs_override() function, this forces the core-client to reread the global_prefs_override.xml file which now contains the updated information (see doUpdate() Function ‘Table 4’). It then sets the nice value to the new value for each running BOINC application as determined by the optimization logic algorithm.
‘Figure 3’ shows the State diagram of the Optimization Middleware describing the decisions the middleware makes to ensure optimum performance of the BOINC client taking into consideration the user’s preference and the MRT constraint.

Table 4: Algorithm 4 - Optimized Middleware doUpdate() function

| query BOINC client to get the following parameters: |
| estimated CPU time remaining, current CPU time, fraction done |
| then |
| use these parameters to determine the following: |
| percent CPU time to completion [see equation 2] |
| days to deadline (deadline date – today’s date) |
| for each BOINC application |
| if (days to deadline < 2 and fraction done < 0.8 and percent to complete < estimated CPU time remaining) |
| set the nice value to the highest (-19) |
| else |
| call BOINC client’s read global_prefs_override() function |
| to reread the global_prefs_override.xml file |
| then |
| for each BOINC application not running in high priority do |
| SET the nice value to the new value as per the algorithm’s choice |
| end for |
| end doUpdate |

5. Our Experimental Design

We conducted a series of simulation experiments to test the optimization middleware and compared the results with simulation results using the plain BOINC client. In order to test the middleware we needed to process the same combination or Work Units repeatedly recording the Wall Time spent processing each WU for each test.

Table 5: Hardware Specification

| Model                                | Specification                                      |
|--------------------------------------|---------------------------------------------------|
| 2 Dell Precision Workstations        | O.S. Ununtu 9.10                                 |
|                                      | CPU: Intel Xeon Dual Core 3.00 GHz.               |
|                                      | RAM: 2048 G.B.                                    |
|                                      | Hard Disk: 160 G.B.                               |
| 2 IBM 6223MCI Workstations           | O.S. Ubuntu 9.10                                 |
|                                      | CPU: Intel Xeon HT 3.89 GHz.                      |
|                                      | Hyper Thread Technology Enabled                   |
|                                      | RAM: 2048 M.B.                                    |
|                                      | Hard Disk: 160 G.B.                               |
Scripts were developed to intermittently consume system resources so as to imitate the effects of a user on the system. Also we decided to test the middleware using hardware with different numbers of CPUs in order to study the effects of the multi-core architectures which are available today.

Tests were designed to be run for the user’s preference settings for CPU times at intervals of [40%, 60%, 80% and 90%].

**Table 6: Experiment Configuration**

| Station | Architecture                  | WU Description        |
|---------|-------------------------------|-----------------------|
| WKS 1   | IBM Single Core with HT       | Einstein@home         |
| WKS 2   | IBM Single Core with HT       | Einstein@home         |
|         |                               | Edges@home            |
| WKS 3   | Dual Core Dell Precision      | Einstein@home         |
|         |                               | Milkyway@home         |
| WKS 4   | Dual Core Dell Precision      | Einstein@home         |
|         |                               | Milkyway@home         |
|         |                               | Rosetta@home          |
|         |                               | Charmm@home           |

**Figure 3: BOINC Optimized Middleware diagram**
A minimum of two repetitions were done for each interval level for both the optimization middleware and without the optimization middleware (plain BOINC client). The results were then normalized by finding the average for each run at the particular interval for the specific architecture (i.e. dual core or single core with Hyper Threading HT). The ‘Table 5’ details the model of workstations used for this experiment along with the hardware specifications and Operating System.

In ‘Table 6’ a brief description of the system architecture along with the project name for the WU selected for processing is given.

NOTE: The repeated project names for the different workstations (WKS) does not indicate different WUs from the projects, but means the same WUs are used on the different machines.

To determine the speedup gained at each setting the following equation is used:

\[
\text{ETP} = \text{execution time of BOINC plain client} \\
\text{ETO} = \text{execution time of BOINC client with optimization middleware} \\
\text{speedup} = \frac{\text{ETP}}{\text{ETO}}
\]  

Experiments were also conducted to test the performance of the optimization middleware when a project was likely to miss its deadline. The setup for these simulations was the same as for the previous simulations except the workstation clock was adjusted so that the BOINC client would calculate an application would miss its deadline. The selected applications for this simulation were Einstein@home (the original one used for all experiments) and a new WU for the Charmm@home application (this WU would be run in normal mode within the BOINC client). The Einstein@home application was the application which the BOINC client would identify as likely to miss its deadline during the experiment, this would cause the BOINC client to run this application in “Highest Priority”. It was decided to run this experiment using the Dual Core system (WKS 3) replacing the Milkyway@home application with the Charmm@home WU.

This section describes how the optimization middleware identify WUs which are likely to miss their deadlines. The optimization middleware queries the BOINC client to gather useful information which is then used to determine which WU the BOINC client would most likely run at the Highest Priority.

The following parameters are used in the algorithm:
- estimated CPU time remaining
- current CPU time
- fraction done (see ‘Table 4’ for usage)

Using the estimated CPU time remaining and the current CPU time, the middleware calculates the percentage of time needed to complete the WU \(\text{percenttocomplete}\) using the following formula:

\[
\text{percenttocomplete} = 0.2 \times (\text{rtime} + \text{ctime})
\]  

Where : \text{rtime} represents the estimated CPU time remaining and \text{ctime} denotes the current CPU time

6. Experimental Results and Analysis

The results of the experiments are now presented and analyzed to demonstrate how well the optimization middleware performed when compared to the plain BOINC client. The ‘figure 4’ illustrate the speedup for the Einstein@home WU, the graph clearly show that at the lower CPU usage
settings the largest speedup is realized, also the graph shows the optimization middleware caused the Wall Processing Time to be almost constant at all the CPU usage settings. This occurred because the middleware made the BOINC client utilize all available resources even though the user settings were at a lower value.

![Figure 4: Experiment 1 - One WU One CPU](image)

In ‘Table 7’ the speedup gained at each tested CPU usage level is listed. A speedup value $> 1$ indicates a realized speedup whereas a value $\leq 1$ indicates no realized speedup.

| % CPU Usage | Speedup |
|-------------|---------|
| 40          | 2.30    |
| 60          | 1.45    |
| 80          | 1.12    |
| 90          | 1.01    |

According to ‘Table 7’ as the CPU Usage setting is increased the effects of using the middleware decreases and at 90% there is almost no difference when using the middleware or the plain BOINC client.
The ‘figure 5’ displays the results for two WUs, Einstein@home (the same WU used in ‘figure 4’) and Edges@home. In this case the Einstein@home WU requires the larger amount of CPU time while the Edges@home WU requires less CPU processing time which were calculated by their respective project server. Here the optimization middleware caused the Wall Processing Times for both applications to be maintained for all the CPU Usage settings while the plain BOINC Client performs poorly at the lower CPU Usage settings and increases as the CPU Usage settings are increased.

The results in ‘Table 8’ shows similar speedup values when two WUs are processed on a single CPU system, however the length of the WU in CPU time requirements have an effect on the ability of

| Percent CPU Usage | Speedup Einstein | Speedup Edges |
|-------------------|------------------|---------------|
| 40                | 2.09             | 1.83          |
| 60                | 1.37             | 1.22          |
| 80                | 1.17             | 1.09          |
| 90                | 1.04             | 0.97          |

the middleware to improve Wall Processing Time when high CPU Usage settings are used. In this case the results show at 80% CPU Usage for the shorter Edges application the speedup realized is insignificant (1.09) and there is no significant speedup beyond that setting.

The ‘figure 6’ shows the comparison of the total times for processing both WUs (Einstein and Edges) using the optimization middleware and the plain BOINC client. Again the curve of the graph indicates the same characteristics exist where speedup values at the lower CPU Usage settings are greater, than for higher CPU Usage settings. The ‘Table 9’ contains the actual speedup values for the total times comparison in ‘figure 6’. According to ‘Table 9’, as the CPU Usage setting is increased the effects of
using the middleware decreases and at 90% there is almost no difference when using the middleware or the plain BOINC client. This result is similar to the results which was obtained in the earlier experiments see ‘Table 7’ and ‘Table 8’ for confirmation of this trend.

Table 9: Speedup for Two WUs One CPU (Total Times)

| % CPU Usage | Speedup |
|-------------|---------|
| 40          | 2.30    |
| 60          | 1.45    |
| 80          | 1.12    |
| 90          | 1.01    |

The graph in ‘figure 3’ displays the comparison of the Wall Processing times for two WUs (Einstein and Milkyway) but these results are for the system with two CPUs. The graph continues to show the same trend when using the optimization middleware and the plain BOINC client.
In this experiment the curve of the graph again indicates the same characteristics which were displayed in the previous simulation results. The speedup values at the lower CPU Usage settings are greater than for higher CPU Usage settings by the user. In ‘Table 10’ the actual speedup values for the two WUs again show that the speedup gained decreases as the user settings for percent CPU usage is increased.

### Table 10: Speedup for Two WUs Two CPUs

| Percent CPU Usage | Einstein | Milkyway |
|-------------------|----------|----------|
| 40                | 1.89     | 2.08     |
| 60                | 1.31     | 1.44     |
| 80                | 1.08     | 1.16     |
| 90                | 0.97     | 1.05     |

An interesting observation in both ‘Table 8’ and ‘Table 10’ is that in both cases the shorter of the two jobs in cputime requirements (Edges@home ‘Table 8’ and Einstein@home ‘Table 10’) showed no significant speedup at the 80% settings. This may suggest that if a system has WUs of fairly differing lengths, the WUs with the shorter duration would benefit less at the higher CPU usage settings when compared to the larger WUs when using the optimization middleware.

### Figure 8: Experiment 4 - Two CPUs Four WUs

In this experiment ‘figure 8’ the curve of the graph again indicates the same characteristics which were displayed in all previous simulation results, this consistency was expected. Again the speedup values at the lower CPU Usage settings are greater than when the CPU Usage settings are increased. In ‘Table 11’ the actual speedup values for the four WUs using two CPUs confirm the consistency of the middleware to speedup the processing of the WUs at the lower user settings for percent CPU usage.

### Table 11: Speedup for Four WUs Two CPUs

| Percent CPU Usage | Speedup |      |      |      |
|-------------------|---------|------|------|------|
|                   | Einstein| Milkyway| Rosetta| Charmm |
| 40                | 1.89    | 2.08  |      |      |
| 60                | 1.31    | 1.44  |      |      |
| 80                | 1.08    | 1.16  |      |      |
| 90                | 0.97    | 1.05  |      |      |
All the previous analysis was based on the optimization middleware running while the BOINC client was processing applications that were well within their deadline period. This meant that the WUs were processed at normal priority within the BOINC framework and were supposed to run at the lowest nice value (priority) by the OS, if the optimization middleware was not used. The next simulation result was produced while the BOINC client was running a WU in “Highest Priority” (a job likely to miss its deadline) and another job in normal priority within the framework. The result of this simulation would be analyzed next.

![Figure 9: Experiment 5 - Two WUs Two CPUs (High Priority)](image)

Before analyzing ‘figure 9’ it must be noted that the EINSTEIN Plain (Non HP) data is taken from ‘figure 7’ which was the experiment use to determine how the middleware performed using Two WUs on a Two CPU system. This was done in order to compare the performance of the processing of this WU (EINSTEIN) when it was run at normal priority and highest priority within the BOINC framework without the middleware running. Looking at the graph both of these curves are almost identical except for a slight variation at the 60% value which can be attributed the fact that these two curves were produced at two different times during different experiments. But the way both curves closely follow the same path indicates that the plain BOINC client has no influence on increasing the speed at which a WU is processed even if it r

|   | 40 | 60 | 80 | 90 |
|---|----|----|----|----|
| 1.41 | 1.11 | 1.09 | 1.03 |
| 1.53 | 1.15 | 1.05 | 1.03 |
| 1.25 | 1.13 | 1.06 | 1.00 |
| 1.25 | 1.12 | 1.12 | 1.06 |

The optimization middleware addresses these issues and produces results similar to the previous simulations. Looking at the EINSTEIN Opt. (HP) and the EINSTEIN Plain (HP) (remember this WU was run at the highest priority within the BOINC client) the curve of the graphs indicates similar speedups as in the previous experiments when using the middleware. The CHARMM WU was run at
normal priority within the client so it was expected to have the same curve as in the previous simulations which indicate a speedup was realized for that WU.

**Table 12:** Speedup for Two WUs Two CPUs (High Priority)

| Percent CPU Usage | Speedup  |
|-------------------|---------|
|                   | Einstein | Charmm |
| 40                | 1.97    | 1.84   |
| 60                | 1.29    | 1.24   |
| 80                | 1.20    | 1.04   |
| 90                | 1.00    | 0.99   |

The ‘Table 11’ contains the data for experiment 5 and as expected it shows a steady decrease in the speedup realized at each increased CPU Usage setting. It also confirms that the shorter WU benefits less in terms of speedup when using the optimization middleware as the data once again shows the shorter WU (CHARMM) had no significant speedup from the 80% settings.

The experiments clearly indicate the effectiveness of the optimization middleware to decrease the time taken by volunteer systems to process their allocated WUs. This behaviour would mean that the optimized middleware can improve the Quality of Service provided by the BOINC framework to the research projects that use the BOINC platform for their high performance processing needs.

The optimization middleware was stable throughout the experiments and this is indicated by the results which were produced by the different experiments. The middleware was able to improve the performance of jobs which were not in any danger of missing their deadlines and it was equally successful in improving the processing of jobs which were in danger of missing their deadlines.

### 7. Conclusions and Recommendations

The adaptive priority tuning middleware system for optimizing local CPU scheduling using the BOINC client has shown that it is possible to fully utilize the unused processing resource of a volunteer system even if the user has low resource settings for CPU usage. This middleware uses the “User Preferences” settings and the underlying operating system state to dynamically adjust the BOINC client properties so that the client can adjust to the constantly changing nature of its running environment. This concept provides a new approach to enhancing the local execution of BOINC applications to provide a meaningful speedup of jobs while maintaining an acceptable response time to the end-user especially at lower CPU Usage settings.

Results have shown a larger speedup is realized at the lower CPU Usage settings and this speedup gradually decline as the CPU Usage settings are increased eventually resulting in no realizable speedup at settings > 90%. This suggests that setting the CPU Usage at > 90% in the normal BOINC client would result in optimal performance for applications. It also suggests that using the optimization middleware for lower CPU Usage settings < 90% would result in better performance for applications.

The optimization middleware monitors the operating system and is able to adjust certain parameters within the BOINC client which causes the client to change its resource requests to the Operating System. The middleware also is able to adjust the *nice* value of the running BOINC applications which (in Linux 2.6) causes the Operating System to adjust the weighted values of the BOINC applications resulting in larger timeslices for those applications. These features result in a more dynamic BOINC client which is able to adjust to the changing nature of the environment in which it runs. This would benefit the various projects which use the BOINC framework by reducing the Wall Time taken to process the WUs, they would thus be able to reduce the deadline time for their WUs. Another benefit
of this research would be in providing a degree of QoS by the BOINC framework to the projects since the speedup would mean results can be returned within the expected time with a reduction in missed deadlines. Further research needs to be done to determine why shorter WUs does not seem to benefit from using the optimization middleware when the CPU usage setting is above 80%.

It is recommended that the middleware be maintained as a separate piece of software and not be integrated within the BOINC client since there are security issues to consider. Also the middleware was designed to be a separate tool from the actual BOINC client so that the end user has the final decision on whether they want to use it or not. This was on purpose since it was felt that users should have the choice to decide whether they wanted to use the middleware or not, also this decision does not violate the spirit of BOINC, which is to only use the unused CPU time on the system.

References
[1] Anderson D 2004 BOINC: A System for Public Resource Computing. GRID, 4-10
[2] Anderson D, Christensen C and Allen B 2006 Designing a Runtime System for Volunteer Computing. SC, 126
[3] Anderson D and McLeod J 2007 Local Scheduling for Volunteer Computing. IDPS, 1-8
[4] Anderson D and Reed K 2009 Celebrating Diversity in Volunteer Computing. HICSS, 1-8
[5] Atlas J, Estrada T, Decker K and Taufer M 2009 Balancing Scientist Needs and Volunteer Preferences in Volunteer Computing using Constraint Optimization. CCGRID, 331-38
[6] http://boinc.berkeley.edu/ 2010
[7] Anderson D, Kondo D and McLeod J 2007 Performance Evaluation of Scheduling Policies for Volunteer Computing. eScience, 415-22
[8] Estrada T, Anderson D and Tauer M 2009 Performance Prediction and Analysis of BOINC Projects: An Empirical Study with EmBOINC. J. Grid Comput. 7(4), 537-54
[9] Estrada T, Fuentes O and Tauer M 2008 A Distributed Evolutionary Method to Design Scheduling Policies for Volunteer Computing. Conf. Computing Frontiers, 313-22