A Delay Scheduling Strategy for Dynamic Load Balancing

Schedulers

Chi-Chung Hui & Samuel T. Chanson

Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong

Abstract

This paper investigates the issue of practical load balancing in the highly dynamic environment of local area networks (LANs). Most existing dynamic load balancing techniques dispatch the jobs immediately upon arrival irrespective of the overall loading of the LAN. This may lead to system saturation and thrashing. Moreover, these schemes focus on steady state system throughput without considering the behavior in the transient periods. As a result, they may not deliver satisfactory performance in practice where jobs may arrive in batches and system loading may fluctuate widely. To tackle the above problems, the delay scheduling strategy is proposed to dynamically delay the execution of jobs after the system is fully utilized. The proposed strategy is general and can be augmented on most existing algorithms. Experimental results showed that the proposed strategy adapts well to load fluctuation, minimizes system loading while producing shorter schedules and improved job fairness compared to some popular schedulers.

1 Introduction

In the past, local area networks (LANs) were mainly used to facilitate communication (e.g., electronic mail and file transfer) and sharing of resources such as printers and file servers. As processors continue to get cheaper and faster, the aggregate computing power of the workstations in a modern LAN can easily exceed that of a supercomputer. The network speeds of some LANs have exceeded 100 megabits per second, thereby reducing the penalty of interprocessor communication. As a result, it is both logical and feasible to dispatch tasks from heavily loaded computers to lightly loaded ones to improve system performance [2, 14, 21].
Many techniques have been proposed to allocate jobs dynamically in a LAN or multiprocessor system to improve performance. Existing dynamic load balancing algorithms can be broadly grouped into the least loaded approach, the threshold based approach and the bidding approach. The objective of the least loaded approach is to allocate jobs to the least loaded computers in the system. Under the threshold based approach, load balancing is performed only when the load levels of some computers have exceeded a predefined threshold value. The jobs in the bidding approach compete for CPU power through a bidding mechanism. In all cases, jobs are allocated to computers based on limited information obtained by the schedulers dynamically.

Unfortunately, most of these algorithms ignore practical issues such as the difference in workstation speed, sudden batch arrivals and user behavior which are especially important in the LAN environment, and so they may not deliver satisfactory performance in practice. Moreover, most algorithms have been evaluated by simulation using the steady state mean response time as the primary performance metric, and some algorithms assume the workstations are identical. Not much work has been done on keeping the load balanced in the transient periods so that the scheduling of batches of jobs (e.g., by shell scripts) may greatly upset the state of global balance and cause poor performance. Another simplification which is commonly adopted is that jobs are dispatched immediately upon arrival irrespective of the overall system loading. As a result, sudden arrival of a large number of jobs may saturate the entire system. An important difference between a LAN environment and a multiprocessor system is that there is usually a user associated with each workstation. It is necessary to limit the number of remote jobs in a workstation with an active user to minimize user dissatisfaction. These issues are important in the contemporary LAN environment in which workstations have different capabilities and are highly autonomous.

The delay scheduling strategy is proposed to address the above issues. The basic idea is simple and intuitive—the execution of jobs are dynamically delayed when the system is fully utilized to maximize the level of load balance in the system. By delaying the scheduling of jobs when the system load is high, the executing and ultimately the delayed jobs receive better service by eliminating over-competition. This technique is general and can be incorporated into existing scheduling algorithms. We have implemented the delay scheduling strategy on the well known centralized least load scheduler, conducted experiments on a network of heterogeneous SPARC stations, and compared its performance with other standard schedulers as well as Utopia/LSF\(^1\) [28] which is a load sharing software for heterogeneous LANs. The results indicated that the proposed algorithms produced better schedules, ranging from 7.8% to 22.2% reduction in batch completion time compared to

\(^1\)LSF (Load Sharing Facility) [1] is the commercial version of Utopia.
those of Utopia/LSF. The proposed algorithm also adapted well to load fluctuation and improved job fairness.

The rest of the paper is organized as follow. Section 2 gives a brief discussion of the related work. The principle of the delay strategy is described in Section 3 and illustrated with an example. The experimental environment and results are presented in Sections 4 and 5 respectively. Section 6 describes an enhanced adaptive algorithm and its performance. Finally, Section 7 concludes the paper.

2 Load Balancing Algorithms

The delay scheduling strategy can be incorporated into most existing load balancing algorithms. Some of them are reviewed in this section.

One of the most intuitive scheduling algorithms is to execute the jobs on the least loaded or idle workstations [4, 7, 17, 18, 24, 25]. To reduce the overhead of disseminating load information among the workstations, algorithms that only allow immediate neighbors to communicate have been proposed. By requiring the hosts to negotiate before migration can take place, a lightly loaded computer will not be overloaded by multiple busy computers [5]. In the contracting with neighborhood approach, the number of migrations that a job can take is restricted to ensure stability [13]. Several algorithms that eliminate load updating messages altogether have been proposed, in which a computer estimates the loading of the other computers based on the pattern of remote execution observed in the communication network [23].

Another major load balancing technique is the threshold based algorithm, where a workstation triggers load balancing actions if its load level exceeds a certain threshold. Based on who starts the load balancing process, the algorithm can be classified as sender-initiated, receiver-initiated or symmetrically-initiated. In the sender-initiated approach, the highly loaded computers dispatch their jobs to computers with lighter loads [6, 9, 10, 15, 22]. On the other hand, the lightly loaded computers in the receiver-initiated approach request the busy computers for jobs [15, 22]. The symmetrically-initiated approach employs a combination of these two schemes [15]. Under the single-queue multiple-server model where a computer does not process any job until the queue length exceeds the computer’s threshold value, an algorithm that computes near-optimal threshold values to minimize mean response time has been proposed [19]. It has been found that the threshold values are not constants and are dependent on the overall system utilization. In general, when the system utilization is low, most incoming jobs are sent to the faster computers, and the slower computers do not process jobs even if there are jobs waiting in the queue.
The bidding approach views the computers as resources and the jobs as consumers. The jobs compete for the resources by issuing bids [8, 26]. The extended stochastic learning automaton is another technique where each computer maintains probability vectors for locating the underloaded computers, and the vectors are updated by a learning process [16, 20]. Utopia/LSF adopt a matching algorithm [28]. Load information is classified into general resource requirements (e.g., CPU and memory) and restrictive resource requirements (e.g., architecture). The workstations are grouped into small clusters and a centralized algorithm is used within each cluster to select the fastest one from the qualified computers [27].

To augment the above techniques for supporting the delay strategy, we can simply add a queue to the scheduler. The incoming jobs are appended to the queue, and are dispatched only when the system load (or the load of the target workstation) is acceptably low (e.g., below some thresholds). In the following sections we discuss the delay strategy in details, and show how to set the workstation thresholds adaptively.

3 The Delay Strategy

3.1 Basic Principle

We differentiate between two kinds of jobs in the LAN environment: the local jobs and the schedulable jobs. The local jobs are those that depend on the information of the workstation where the job is generated (e.g., hostname and uname) or are I/O-bound jobs (e.g., vi, sed and awk) which are best served on the local workstations. On the other hand, the schedulable jobs are mostly CPU-bound jobs (e.g., long running simulation jobs) that can benefit from remote executions. It is assumed that the local jobs are executed on the local workstations immediately upon arrival, and the schedulable jobs are handled by the scheduler, and may be executed locally or a remote workstation either immediately or delayed. Unless stated otherwise, the term job refers to a schedulable job rather than a local job.

In a LAN environment $W$ where each workstation $w_i \in W$ is running at speed $s_i$, let $f_i$ be the scheduling function reflecting the desirability of executing a new job on $w_i$. The larger $f_i$ is, the more desirable to send a job to $w_i$. A job is not submitted to $w_i$ if $f_i < 0$ (e.g., when $w_i$ is overloaded). In a busy system where $f_{\text{max}} = \max_{w_i \in W} \{f_i\} < 0$, it is better to wait for some running jobs to complete rather than to dispatch additional jobs to overload an already saturated system. Accordingly, a job should be submitted to $w_{\text{max}}$ when $f_{\text{max}} \geq 0$.

Intuitively, the best load balancing status is achieved when all workstations are at the point of full utilization (but not saturated) and the workload assigned to each workstation is proportional to
its capacity. Allocating more jobs to a fully utilized system may cause imbalance without improving the overall throughput. Consider the example in Figure 1 where 10 jobs are to be allocated to 2 workstations. Here, 4 jobs have been scheduled, with 2 long jobs in \( w_1 \) and 2 short jobs in \( w_2 \). If long jobs take a hundred times longer to complete than short jobs, and if the job type (i.e., short or long) is known in advance, it is logical to dispatch the 2 pending long jobs in the queue to \( w_2 \) to balance the distribution of the long jobs. In this way, the two workstations will have a better chance of finishing the jobs in roughly the same time. Unfortunately, the job types in a LAN environment are usually not known in advance. In practice, the only information available to the scheduler is that 2 jobs are running in each workstation and 6 jobs are pending. If any pending long job is sent to \( w_1 \), the execution time of the whole batch will be lengthened.

A reasonable scheme is to set \( f_i < 0 \) as long as \( w_i \) is fully utilized. This scheme does not affect system throughput and tends to even out the workload distribution in the LAN. Returning to the example, if we assume a workstation is fully utilized when there are 2 jobs running on it, then \( f_1 < 0 \) and \( f_2 < 0 \) and the scheduler will not dispatch the next job until a running job is completed. Since all the short jobs are in \( w_2 \), it is more likely that a job in \( w_2 \) would be completed first. Therefore, the 2 pending long jobs have higher chance of being allocated to \( w_2 \). In general, this delay scheduling strategy prevents the workstations from being overloaded and tends to distribute the long jobs to the workstations more evenly. A sample implementation is presented in the next section, and it is shown in Section 5 that the proposed algorithm minimizes the elapsed time of the

![Figure 1: Motivation of the delay scheduling strategy.](image-url)
job groups as well as improves the responsiveness of individual jobs.

3.2 System Issues and Sample Implementation

In order to demonstrate the effectiveness of the delay scheduling strategy, we have implemented DELAY (Figure 2), a centralized least load scheduler augmented with the delay strategy which works on heterogeneous workstations. The algorithm takes the speeds \((s_i)\) and bounding factors \((b_i)\) of the workstations as the input parameters (the meanings of \(b_i\), \(c_i\), \(j_i\) and \(s'_i\) are described below). First, DELAY initializes the pending queue \(Q_{\text{pending}}\) and other state variables (steps 1 to 3). It then processes the incoming events in the main loop (steps 4 to 9). There are three possible events: (i) update of load information (load_update), (ii) arrival of a new job (new_job), and (iii) completion of a running job (job_done). \(Q_{\text{pending}}\) is a FIFO queue in which a new job enters at the end (step 6). The scheduler assigns the first job in \(Q_{\text{pending}}\) to the fastest workstation only when the workstation satisfies the scheduling criteria (step 8).

Most scheduling algorithms submit a job to the selected workstation without further checking if it has become congested after selection and before sending out the job. Furthermore, the algorithms continue to dispatch jobs even when the entire system is saturated. This may decrease execution efficiency by introducing resource contention (especially in main memory), resulting in excessive

**Figure 2:** Formal description of DELAY.

DELA Y \((B, S)\)

- \(B = \{b_1, \ldots, b_n\}\): Bounding factors of the workstations.
- \(S = \{s_1, \ldots, s_n\}\): Speeds of the workstations.

1. Let \(Q_{\text{pending}} = \phi\).
2. Obtain the (initial) run queue lengths \(q_i\) for all \(i\).
3. Let \(c_i = 0, j_i = \max\{q_i, c_i\}\), \(s'_i = \frac{s_i}{j_i + 1}\) and \(f_i = s'_i - \frac{1}{b_i}\) for all \(i\).
4. Extract the first incoming message \((t, x, q)\) from the message queue.
5. If \(t = \text{load\_update}\) then
   - (a) Let \(q_x = q, j_x = \max\{q_x, c_x\}\), \(s'_x = \frac{s_x}{j_x + 1}\) and \(f_x = s'_x - \frac{1}{b_x}\).
6. If \(t = \text{new\_job}\) then
   - (a) Append \(x\) at the end of \(Q_{\text{pending}}\).
7. If \(t = \text{job\_done}\) then
   - (a) Let \(c_x = c_x - 1, j_x = \max\{q_x, c_x\}\), \(s'_x = \frac{s_x}{j_x + 1}\) and \(f_x = s'_x - \frac{1}{b_x}\).
8. While \(Q_{\text{pending}} \neq \phi\) and \((W_{\text{best}} = \max_i\{s_i | f_i \geq 0\}) \neq \phi\) do
   - (a) Extract the first job, \(j\), from \(Q_{\text{pending}}\).
   - (b) Execute job \(j\) at \(w_x \in W_{\text{best}}\).
   - (c) Let \(c_x = c_x - 1, j_x = \max\{q_x, c_x\}\), \(s'_x = \frac{s_x}{j_x + 1}\) and \(f_x = s'_x - \frac{1}{b_x}\).
swapping and context switches. Moreover, heavy resource contention may result in thrashing which could prevent the workstations from doing useful work.

The delay scheduling strategy rectifies the above problems by postponing the dispatch of new jobs when the system loading is too high. Each workstation \( w_i \) is associated with a bounding factor \( b_i \) which limits the number of schedulable jobs in \( w_i \). If there are \( j_i \) jobs running in \( w_i \) and a new job is submitted to \( w_i \), the effective speed of \( w_i \) in processing the new job can be approximated by

\[
s'_i = \frac{s_i}{f_i + 1},
\]

The scheduling function \( f_i \) is defined as

\[
f_i = s'_i - \frac{1}{b_i}.
\]

These variables are initialized in step 3 and updated in steps 5a, 7a and 8c. According to the definition of \( f_i \), a job can be dispatched to \( w_i \) only if \( s'_i \geq \frac{1}{b_i} \), and a job is assigned to the fastest workstation \( w_{\text{max}} \) when \( f_{\text{max}} \geq 0 \) (step 8). If there exists more than one candidate workstation (kept in the set \( W_{\text{best}} \) in step 8), one of them is selected randomly (step 8b). On the other hand, if there is no workstation satisfying the condition, the incoming jobs are held in \( Q_{\text{pending}} \) and are dispatched only if some running jobs leave the system (step 7).

DELAY also improves on the existing algorithms in several ways. A factor which often leads to inaccurate job scheduling is the lag time in disseminating load information. In order to reduce the overhead in collecting load information, most load monitors query and export load information only once every few seconds. It is possible that several jobs submitted to the scheduler in between load updates are scheduled to the same workstation. Utopia/LSF tries to avoid this problem by updating the load information in the scheduler as soon as a scheduling decision is made [27, 28]. However, experimental evaluation in Section 5 showed that Utopia/LSF is not adaptive enough to handle very high workload.

Previous studies suggested that the loading of a workstation \( w_i \) is best described by the run queue length \( q_i \), which represents the number of processes in the run queue of \( w_i \) [16]. This metric is widely adopted in many dynamic load balancing algorithms [3, 11, 16, 27]. However, our results showed that the run queue length is not the best indicator in the LAN environment. Contemporary workstations often consist of a number of co-processors in addition to the CPU processor. These may include the I/O processor, the DMA processor and the network processor which handle different types of operations. An I/O-bound job frequently activates the I/O processor, and is often blocked
without staying in the run queue. As a result, some processes which are actually running may not be in the run queue, making \( q_i \) inadequate as an indicator of workstation loading.

We address this problem by maintaining the number of active jobs on \( w_i \) (denoted by \( j_i \)) as

\[
j_i = \max\{q_i, c_i\}
\]

where \( c_i \) is the number of jobs currently allocated by the scheduler on \( w_i \). \( c_i \) is greater than \( q_i \) if there is no local job and some jobs are being served by \( w_i \)'s co-processors. On the other hand, \( q_i \) may be greater than \( c_i \) when the console user of \( w_i \) submits local jobs which bypass the scheduler.

We claim that \( j_i \) is a better load index than \( q_i \) in the LAN environment where local jobs are best executed locally. This index is conservative since the local jobs can always be executed on the local workstation while the remote jobs must satisfy the scheduling rule before they can be executed. The bounding factor \( b_i \) controls the maximum number of remote jobs that can run on \( w_i \). In the extreme case, \( b_i \) can be set to zero to exclude remote jobs. The variables \( q_i, c_i \) and \( j_i \) are set in steps 3, 5a, 7a, 8c and used in step 8.

For the purpose of comparison, the following algorithms in addition to DELAY have been implemented and tested:

- **Immediate scheduling algorithm**: This is a centralized algorithm in which the submitted jobs are immediately dispatched to the fastest workstation. By defining \( j_i = q_i \), we can study how system performance is affected by the delay in updating load information.

- **Random scheduling algorithm**: This is a distributed dynamic scheduler. Each job is scheduled to run on a randomly selected workstation based on its relative speed. Specifically, the probability of scheduling a job to \( w_i \) is equal to \( \frac{s_i}{\sum_j s_j} \).

- **Utopia/LSF scheduling algorithm**: Utopia/LSF has a sophisticated scheduler for load balancing in large systems [28]. In our experiments, both Utopia and LSF v2.2 were installed and treated as black boxes which provided scheduling decisions only. Since their performance were measured to be very similar, only one set of results was presented for Utopia/LSF.

## 4 Experimental Environment

### 4.1 The BALANCE Testbed

The experiments are conducted under the *BALANCE testbed* [12]. BALANCE is a flexible load balancing system designed to support a wide range of software, including parallel applications, dis-
tributed applications and system schedulers. It provides efficient remote execution and interprocess communication facilities, and is not tied to a particular scheduling algorithm. For example, \texttt{baltcs}
sh (an extension of the GNU tcsh) supports the command \texttt{balschedule X} that instructs \texttt{baltcs}
sh to consult the scheduler named \texttt{X} to select workstations for executing subsequent commands. The users may write their own schedulers and plug them into \texttt{baltcs}
sh dynamically. In our experiments, the load monitors and DELAY were implemented as generic servers to provide load information and scheduling decision respectively.

4.2 Workload Characteristics

The main objective of the experiment was to investigate the ability of the delay scheduling strategy to balance the workload under heavy load which fluctuates widely rather than to compare the performance of the different scheduling algorithms. Since the jobs are not delayed when the system loading is light, heavy workloads which sometime result in full system utilization were chosen. Each workload is called a batch and is created by a load generator. The batch is divided into groups, where the group size and the group interarrival time are uniformly distributed in the range $[1 : 6]$ and $[9s : 28s]$ respectively. Each job is characterized by the following resource requirements:

- \textit{Processing demand (CPU)}: One unit is equivalent to 100 iterations of a floating point multiplication followed by a floating point division and an assignment statement.

- \textit{Disk access (I/O)}: For each unit of I/O, the job writes 1 MB of data to the disk and then reads it back. As I/O operations are either performed on the local disk or the shared file system, we define the parameter \texttt{IO-L} as the probability that the I/O operations are executed locally.

- \textit{Interprocessor communication (IPC)}: This parameter describes the amount of interprocessor communication performed by the job. A job spawns a slave process on a randomly selected workstation. Then for each unit of IPC, it sends 1MB of data to the slave and receives an acknowledge from it.

- \textit{Memory operation (MEMORY)}: The job allocates and frees 100 KB of memory space for each unit of memory operation.

- \textit{Number of phases (P)}: The above operations are evenly divided into P phases which are executed by a job in an interleaved manner.

The scheduling policies were tested under three types of workloads: (i) \textit{CPU-bound workload}, (ii) \textit{I/O-bound workload}, and (iii) \textit{Mix workload}. Each workload (batch) consists of 100 jobs with
characteristics summarized in Table 1. The Mix workload consists of four types of jobs which are

<table>
<thead>
<tr>
<th>Type</th>
<th>%</th>
<th>CPU</th>
<th>I/O</th>
<th>IO-L</th>
<th>IPC</th>
<th>MEMORY</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-bound</td>
<td>100%</td>
<td>[1000 : 2000000]</td>
<td>10</td>
<td>100%</td>
<td>10</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>I/O-bound</td>
<td>100%</td>
<td>1000</td>
<td>[100 : 3500]</td>
<td>60%</td>
<td>10</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>Mix(CPU)</td>
<td>40%</td>
<td>[1000 : 2000000]</td>
<td>10</td>
<td>100%</td>
<td>10</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>(I/O)</td>
<td>40%</td>
<td>1000</td>
<td>[100 : 3500]</td>
<td>60%</td>
<td>10</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>(IPC)</td>
<td>10%</td>
<td>[1000 : 500000]</td>
<td>[100 : 1000]</td>
<td>50%</td>
<td>[100 : 1000]</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>(MEM)</td>
<td>10%</td>
<td>[1000 : 500000]</td>
<td>[100 : 1000]</td>
<td>50%</td>
<td>10</td>
<td>[5000, 20000]</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 1: Definitions of the workloads.

CPU-bound jobs (40%), I/O-bound jobs (40%), IPC-bound jobs (10%) and MEMORY-bound jobs (10%). There is no local job in the workloads.

4.3 Architecture of the Evaluation System

The experiments were conducted in a controlled environment consisting of seven Sun SPARC stations connected by a LAN (see Table 2). The software architecture comprised of a set of communicating processes shown in Figure 3. The Driver gets the jobs from the Load-generator. When a job arrives, the Driver sends a request to the Scheduler which computes the decision according to the specific scheduling algorithm and returns the decision to the Driver. Accordingly, the Driver spawns a Worker at the designated workstation which may in turn spawn a Slave to perform interprocessor communication.

For the immediate and delay scheduling algorithms, a Load-monitor runs on every workstation and computes the load every second. The load information is sent to the Scheduler only if there is a change. The random scheduling algorithm computes the scheduling decision without using any

<table>
<thead>
<tr>
<th>Machine Name</th>
<th>Type of CPU</th>
<th>Relative Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPX.1</td>
<td>40 MHz Sun 4/50</td>
<td>1.0</td>
</tr>
<tr>
<td>SPARC5.1</td>
<td>70 MHz microSPARC II</td>
<td>2.4</td>
</tr>
<tr>
<td>SPARC5.2</td>
<td>70 MHz microSPARC II</td>
<td>2.4</td>
</tr>
<tr>
<td>SPARC10.1</td>
<td>SuperSPARC Model 10/30</td>
<td>2.9</td>
</tr>
<tr>
<td>SPARC10.2</td>
<td>SuperSPARC Model 10/30</td>
<td>2.9</td>
</tr>
<tr>
<td>SPARC10.3</td>
<td>SuperSPARC Model 10/40</td>
<td>3.2</td>
</tr>
<tr>
<td>SPARC20.1</td>
<td>Model 20/50 SuperSPARC</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>18.8</td>
</tr>
</tbody>
</table>

Table 2: Configuration of the workstations.
load information, and so no Load-monitor is needed. For the Utopia/LSF scheduling algorithm, the Scheduler is implemented as a dummy process which consults Utopia/LSF for scheduling decisions.

4.4 Performance Metrics

The schedulers make no assumption on the nature of the jobs. Six metrics were selected to compare the scheduling performance:

- **Batch completion time** \((B)\): This is defined as the elapsed time of executing the entire workload. This metric can be viewed as a measure of system throughput, and reflects whether the jobs are allocated to the workstations proportional to their processing speeds. This is because an evenly distributed workload implies that the workstations will complete their jobs at about the same time. In general, the smaller the value of \(B\), the more even is the workload distribution and the higher the throughput.

- **Overall speedup** \((S)\): This is defined as

\[
S = \frac{B_1}{B}
\]

where \(B\) is the batch completion time and \(B_1\) is the sum of the job completion times when all the jobs are executed one at a time on the slowest workstation (i.e., IPX.1 with speed equal to 1).

- **Mean response time** \((\overline{T})\): This is defined as the average completion time of the jobs in each batch. This metric is used in most of the previous studies as an indication of scheduling
performance.

- **Mean job completion time ratio ($\mathcal{J}$):** The job completion time ratio $J$ is defined for each job by the formula
  
  $$J = \frac{E_2}{E_1}$$

  where $E_1$ is the shortest possible elapsed time when the job is executed on the fastest idle machine (in this case it is SPARC20.1), and $E_2$ is the elapsed time when the job is executed in competition with the other jobs. The metric $\mathcal{J}$ is then defined as the average value of $J$ over all the jobs. Compared to $\mathcal{T}$, $\mathcal{J}$ eliminates the effect of job size on individual job performance. The smaller the value of $\mathcal{J}$, the better the performance.

- **Standard deviation of $J$ ($sd(J)$):** This is defined as the standard deviation of $J$ over all jobs. This metric provides an indication of whether the jobs receive equal treatment. The smaller the value of $sd(J)$, the fairer the scheduling algorithm.

- **Average waiting period ($\overline{W}$):** The waiting period $W$ is defined as the percentage of time the jobs spent in the pending queue relative to its execution time. If $W = 200\%$, the job spends 2$x$ time units waiting to be scheduled and $x$ time units in actual execution. The metric $\overline{W}$ is defined as the mean of $W$ over all jobs.

$B$ and $S$ represent the system throughput of the jobs as a batch, while $\mathcal{T}$ and $\mathcal{J}$ reflect average performance of individual jobs. A scheduling algorithm may produce small $\mathcal{T}$ and $\mathcal{J}$ but large $B$ and $S$ if the workload is not distributed fairly evenly to the workstations. The fairness of the schedule and the overhead of the delay scheduling algorithm are summarized in $sd(J)$ and $\overline{W}$ respectively.

5 Experimental Results and Discussions

The experimental results are summarized in Table 3. It can be seen that the performance of the algorithms varies greatly. DELAY has the best overall performance. On the other hand, the random scheduling algorithm performs poorly since it does not make use of current load information.

5.1 The Effect of Lag Time in Load Updates

An important observation is that the delay in updating load information is a critical factor on scheduling performance. Although the load information was updated every second, the immediate scheduling algorithm allocated most jobs in the same group to the same workstation. Although Utopia/LSF alleviates this problem by load pre-adjustment (refer to Section 3.2 for details), it also
<table>
<thead>
<tr>
<th>Type</th>
<th>B</th>
<th>S</th>
<th>T</th>
<th>J</th>
<th>sd(J)</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU-bound workload</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td>936.51</td>
<td>14.23</td>
<td>117.24</td>
<td>3.61</td>
<td>2.35</td>
<td>0.76</td>
</tr>
<tr>
<td>Immediate</td>
<td>830.52</td>
<td>15.77</td>
<td>86.10</td>
<td>3.25</td>
<td>1.44</td>
<td>0.56</td>
</tr>
<tr>
<td>Utopia/LSF</td>
<td>797.83</td>
<td>16.47</td>
<td>81.15</td>
<td>2.87</td>
<td>1.25</td>
<td>0.68</td>
</tr>
<tr>
<td>DELAY(1)</td>
<td>739.10</td>
<td>17.75</td>
<td>68.55</td>
<td>2.51</td>
<td>1.34</td>
<td>7.36</td>
</tr>
<tr>
<td>DELAY(2)</td>
<td>757.04</td>
<td>17.30</td>
<td>69.29</td>
<td>2.23</td>
<td>0.76</td>
<td>0.73</td>
</tr>
<tr>
<td>DELAY(3)</td>
<td>768.81</td>
<td>17.07</td>
<td>68.67</td>
<td>2.22</td>
<td>0.75</td>
<td>0.77</td>
</tr>
<tr>
<td>DELAY(4)</td>
<td>760.48</td>
<td>17.26</td>
<td>67.94</td>
<td>2.21</td>
<td>0.73</td>
<td>0.74</td>
</tr>
<tr>
<td>DELAY(5)</td>
<td>759.15</td>
<td>17.27</td>
<td>69.77</td>
<td>2.26</td>
<td>0.76</td>
<td>0.73</td>
</tr>
<tr>
<td>DELAY(∞)</td>
<td>760.35</td>
<td>17.22</td>
<td>70.48</td>
<td>2.27</td>
<td>0.76</td>
<td>0.76</td>
</tr>
<tr>
<td><strong>I/O-bound workload</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td>1101.89</td>
<td>13.89</td>
<td>103.59</td>
<td>2.54</td>
<td>2.03</td>
<td>0.46</td>
</tr>
<tr>
<td>Immediate</td>
<td>1166.69</td>
<td>13.15</td>
<td>143.83</td>
<td>3.81</td>
<td>2.56</td>
<td>0.25</td>
</tr>
<tr>
<td>Utopia/LSF</td>
<td>1073.23</td>
<td>14.38</td>
<td>98.32</td>
<td>2.54</td>
<td>1.78</td>
<td>0.47</td>
</tr>
<tr>
<td>DELAY(1)</td>
<td>878.49</td>
<td>17.42</td>
<td>65.67</td>
<td>1.83</td>
<td>0.89</td>
<td>2.45</td>
</tr>
<tr>
<td>DELAY(2)</td>
<td>890.50</td>
<td>17.16</td>
<td>68.44</td>
<td>1.82</td>
<td>0.95</td>
<td>0.43</td>
</tr>
<tr>
<td>DELAY(3)</td>
<td>904.27</td>
<td>16.95</td>
<td>65.52</td>
<td>1.74</td>
<td>0.89</td>
<td>0.45</td>
</tr>
<tr>
<td>DELAY(4)</td>
<td>900.42</td>
<td>17.04</td>
<td>66.34</td>
<td>1.78</td>
<td>0.92</td>
<td>0.43</td>
</tr>
<tr>
<td>DELAY(5)</td>
<td>909.16</td>
<td>16.85</td>
<td>68.82</td>
<td>1.83</td>
<td>0.95</td>
<td>0.42</td>
</tr>
<tr>
<td>DELAY(∞)</td>
<td>936.47</td>
<td>16.32</td>
<td>75.92</td>
<td>1.98</td>
<td>1.13</td>
<td>0.44</td>
</tr>
<tr>
<td><strong>Mix workload</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td>983.47</td>
<td>22.78</td>
<td>190.47</td>
<td>4.03</td>
<td>2.80</td>
<td>0.13</td>
</tr>
<tr>
<td>Immediate</td>
<td>1060.50</td>
<td>21.34</td>
<td>181.01</td>
<td>4.15</td>
<td>2.73</td>
<td>0.10</td>
</tr>
<tr>
<td>Utopia/LSF</td>
<td>942.25</td>
<td>23.75</td>
<td>167.88</td>
<td>3.79</td>
<td>2.15</td>
<td>0.11</td>
</tr>
<tr>
<td>DELAY(1)</td>
<td>848.57</td>
<td>26.23</td>
<td>161.28</td>
<td>4.20</td>
<td>3.04</td>
<td>3.436</td>
</tr>
<tr>
<td>DELAY(2)</td>
<td>844.25</td>
<td>26.39</td>
<td>151.73</td>
<td>3.23</td>
<td>1.64</td>
<td>1.54</td>
</tr>
<tr>
<td>DELAY(3)</td>
<td>850.23</td>
<td>26.30</td>
<td>155.65</td>
<td>3.27</td>
<td>1.63</td>
<td>0.13</td>
</tr>
<tr>
<td>DELAY(4)</td>
<td>844.63</td>
<td>26.44</td>
<td>155.41</td>
<td>3.23</td>
<td>1.64</td>
<td>0.13</td>
</tr>
<tr>
<td>DELAY(5)</td>
<td>861.26</td>
<td>25.83</td>
<td>151.37</td>
<td>3.19</td>
<td>1.61</td>
<td>0.12</td>
</tr>
<tr>
<td>DELAY(∞)</td>
<td>846.28</td>
<td>26.41</td>
<td>152.71</td>
<td>3.19</td>
<td>1.59</td>
<td>0.13</td>
</tr>
</tbody>
</table>

Table 3: Performance of the dynamic scheduling algorithms. The parameter associated with the delay scheduling algorithm corresponds to the bounding factor $b$. 
suffered from overallocation. Figure 4 shows the Gantt chart of an Utopia/LSF schedule where each horizontal line presents the execution profile of a job. As illustrated in Figure 5, each line consists of three checkpoints X, Y and Z which represent the job submission time, the execution start time and the job finish time respectively. In the Utopia/LSF scheme, since a job starts execution as soon as it is submitted, X and Y occur at the same time. According to Figure 4, many jobs in the same groups were scheduled to the same workstation. For example, six jobs were dispatched to SPARC10.2 and SPARC10.3 at time instants 456s and 250s respectively. On the other hand, DELAY made use of the number of active jobs (i.e., \( j_i \)) on selecting the fastest workstation. Since DELAY maintained the number of scheduled jobs (i.e., \( c_i \)) inside the scheduler, it could accurately estimate the workstation loadings even though there is non-negligible lag time for the up-to-date load indices to arrive. As a result, even DELAY(\( \infty \)) (i.e., with the delay strategy disabled) produced
significant improvement over the other scheduling algorithms.

5.2 Performance of the Delay Scheduling Strategy

The delay scheduling algorithm has the best batch completion time and speedup for all three types of workloads. It reduces the completion times of the job groups by 7.8% (for CPU-bound workload), 22.2% (for I/O-bound workload) and 11.6% (for Mix workload) compared to those of Utopia/LSF. Under the CPU-bound and I/O-bound workloads, the delay scheduling algorithm with \( b = 1 \) for all workstations achieved speedups of 17.75 and 17.42 for CPU-bound and I/O-bound workloads respectively, which is very close to the optimal speedup of 18.8, the total computing power of the 7 workstations (see Table 2). By putting an upper limit on the number of schedulable jobs on each workstation, the delay scheduling algorithm effectively avoids excessive contention and produces better batch completion time. Moreover, the delay scheduling algorithm reduces the probability of making bad choices of workstations for large jobs by postponing the scheduling decision until it is clear a machine with sufficient resource is available. As a result, the workload allocated to each workstation is more even compared to the other scheduling algorithms.

In addition, since the delay scheduling algorithm limits the number of jobs in each workstation, it is possible to impose an upper bound on the workstation loading. Therefore, local jobs can be guaranteed some fraction (or all) of the capacity of the local workstation. For the other scheduling algorithms this bound does not exist. For example, it can be determined from Figure 4 that the maximum run queue lengths for SPARC10.2 and SPARC10.3 in the Utopia/LSF schedule were 11 and 16 respectively. Under the delay scheduling algorithm it is easy to trade off console responsiveness with system scheduling quality by adjusting the bounding factors of the workstations.

In the absence of information on job characteristics, it is impossible to predict the future loadings of the workstations with any accuracy. Since the whole batch completes only when all the jobs are done, a wrong allocation would adversely affect the batch completion time. For example, it can be seen from Figure 4 that Utopia/LSF scheduled several long jobs to the same workstation (e.g., SPARC10.2) while some workstations were starving from insufficient jobs (e.g., IPX.1). This problem did not occur in the case of delay scheduling, where a typical schedule is shown in Figure 6. When all the workstations are fully utilized, an incoming job is not scheduled until a job has departed. This keeps the system relatively balanced, and the long jobs are more evenly distributed across the workstations.
5.3 The Effect of Multiple Processors in a Workstation

In the case of Mix workloads, superlinear speedups are obtained for all algorithms. This is because jobs with different resource requirements were served by different processors in a workstation simultaneously (e.g., CPU-bound and I/O-bound jobs were served by the CPU and I/O processors). The same reason explains the relatively poor performance of the immediate and Utopia/LSF scheduling algorithms for I/O-bound and Mix workloads. The jobs in these two types of workloads are mainly I/O-bound, MEMORY-bound and IPC-bound which do not use the CPU processor often. Therefore, for much of the time they are utilizing system resources without being in the run queue. Consequently, the immediate and Utopia/LSF scheduling algorithms, which assume that the loadings of the workstations are best described by the run queue length, fail to balance the workload effectively. The delay scheduling algorithm, on the other hand, performs much better by using the total number of active jobs on each workstation.

5.4 Performance of Individual Jobs

It is worth noting that the mean response time $T$ is not directly proportional to the batch completion time $B$, although a schedule with small $T$ usually has small $B$. For example, although the mean response time of the immediate scheduling algorithm (181.01s) under the Mix workload
is smaller than that of the random scheduling algorithm (190.47s), its batch completion time is longer (1060.50s vs. 983.47s). This implies that a scheduling algorithm that minimizes $T$ may not necessarily maximize the batch throughput. The loads at all the workstations must be balanced as a whole in order to minimize $B$.

It is interesting to note that the average job response time ratio $J$ of the delay scheduling algorithm was also the best among the algorithms tested (except for the case of DELAY(1) under the Mix workload which will be discussed later). Moreover, the jobs also received fairer treatment (i.e., $sd(J)$ was the smallest). This is not intuitive since the delay scheduling algorithm postpones the execution of some jobs, and so one would expect a larger variation on $J$. The explanation is that a poor schedule (i.e., one that produces large $B$ and small $S$) has larger job response times in the overloaded workstations. From the experimental result we know that if the bounding factor $b$ is set to be sufficiently large (in our case, it was 2), most of the jobs are not delayed so that the job wait time in DELAY is negligible. This implies that a suitable level of delay can improve the overall throughput without sacrificing responsiveness.

5.5 Optimal Delay Factor

If the bounding factor $b$ is not set properly, the responsiveness of the individual jobs could suffer. Take the example of DELAY(1) in the Mix workload. It can be seen from Table 3 that the metrics $J$, $sd(J)$ and $W$ were relatively large although the batch completion time and the speedup of the overall workload were among the best and were relatively insensitive to $b$. As seen in Figure 7, most of the jobs were delayed. On average, the jobs spent an additional 34.36% of the execution time in waiting for execution (i.e., $W$). The main reason is that the resource requirement of the Mix workload (best $J = 3.19$) is higher than that of the CPU-bound workload (best $J = 2.21$) and the I/O-bound workload (best $J = 1.74$). As a result, a large number of jobs were held back by the scheduler for a long time. In order to obtain good response times, $b$ should be set such that jobs are delayed just enough to balance the system loading and no more. It is clear that the optimal value of $b$ depends on the resource requirements of the workload and is not a constant.

6 Adaptive Delay Strategy

In order to improve the adaptability of DELAY on workload variation, an improved algorithm DELAY$*$ is proposed. The main objective of the adaptive algorithm is to tune the bounding factor $b_i$ to improve response times as the load changes while maintaining the overall balance of the system. As shown in Figure 8, DELAY$*$ is different from DELAY in several aspects. First of all, the user
no longer need to specify the initial values of $b_i$. They are initialized to 1 in step 3 and updated in steps 9 and 10 depending on the length of the pending queue (i.e., $|Q_{pending}|$). If the queue grows and exceeds the aggregate workstation speeds (i.e., $\sum_i s_i$), the bounding factor is increased by 1 in step 9. In this way, the pending jobs are distributed to the workstations proportional to their speeds. This allows the bounding factor to be adjusted to the most suitable value without introducing imbalance in the system. On the other hand, in step 10 the bounding factor is checked to see if it has grown too large relative to the workload. If the current speed of $w_i$ (i.e., $\frac{b}{b_i-1}$) is greater than or equal to $\frac{1}{b_i-1}$ for all $i$, the incoming jobs can be dispatched immediately even if the bounding factor is reduced by 1. Accordingly, $b_i$ is decremented by 1 for all $i$.

DELAY* was evaluated using the same experimental setup as described in Section 4, and the results are shown in Table 4. The results of DELAY(1) and Utopia/LSF are included for comparison purposes. It can be seen that for CPU-bound and I/O-bound workloads, DELAY* did not adjust the bounding factor and obtained similar performance as DELAY(1) which indicates that the overhead of DELAY* is similar to that of DELAY(1). For the Mix workload where the resource requirements were much higher so that serious delay occurred for DELAY(1), DELAY* significantly improved the responsiveness and fairness of the jobs and outperformed Utopia/LSF. Moreover, the average waiting period $\bar{W}$ was reduced from 34.36% to 15.55%. By inspecting a

Figure 7: Gantt chart of the DELAY(1) schedule (Mix workload).
DELAY*(S)

\[ S = \{s_1, \ldots, s_n\} \]: Speeds of the workstations.

1. Let \( Q_{\text{pending}} = \emptyset \).
2. Obtain the (initial) run queue lengths \( q_i \) for all \( i \).
3. Let \( b_i = 1, c_i = 0, j_i = \max\{q_i, c_i\}, s'_i = \frac{s_i}{j_i + 1} \) and \( f_i = s'_i - \frac{1}{b_i} \) for all \( i \).
4. Extract the first incoming message \((t, x, q)\) from the message queue.
5. If \( t = \text{“load update”} \) then
   (a) Let \( q_x = q, j_x = \max\{q_x, c_x\}, s'_x = \frac{s_x}{j_x + 1} \) and \( f_x = s'_x - \frac{1}{b_x} \).
6. If \( t = \text{“new job”} \) then
   (a) Append \( x \) at the end of \( Q_{\text{pending}} \).
7. If \( t = \text{“job done”} \) then
   (a) Let \( c_x = c_x - 1, j_x = \max\{q_x, c_x\}, s'_x = \frac{s_x}{j_x + 1} \) and \( f_x = s'_x - \frac{1}{b_x} \).
8. While \( Q_{\text{pending}} \neq \emptyset \) and \( W_{\text{best}} = \max_i \{s_i|f_i \geq 0\} \neq \emptyset \) do
   (a) Extract the first job, \( j \), from \( Q_{\text{pending}} \).
   (b) Execute job \( j \) at \( w_x \in W_{\text{best}} \).
   (c) Let \( c_x = c_x - 1, j_x = \max\{q_x, c_x\}, s'_x = \frac{s_x}{j_x + 1} \) and \( f_x = s'_x - \frac{1}{b_x} \).
9. If \( |Q_{\text{pending}}| \geq \sum_i s_i \) then
   (a) Let \( b_i = b_i + 1 \) for all \( i \).
   (b) Goto step 8.
10. If \( b_i > 1 \) and \( \frac{s_i}{j_i} - \frac{1}{b_i - 1} \geq 0 \) for all \( i \) then
    (a) Let \( b_i = b_i - 1 \) for all \( i \).

Figure 8: Formal description of DELAY*.

typical DELAY* schedule (Figure 9), it is seen that the bounding factor was increased to 2 at time instant 3888s. The 15.55% average waiting period was mainly contributed by the jobs which were delayed during the transient period (i.e., when \( |Q_{\text{pending}}| \) was increasing). After the bounding factor stabilized, the waiting times of the subsequent jobs dropped and were negligible. In general, DELAY* will deliver similar performance as that of DELAY set with the optimal bounding factor, and the overhead due to dynamic adjustment of the bounding factor is negligible.

To confirm the result, we performed additional experiments on a larger system with heavier workload. The computing system consisted of 20 SPARCstations, in which 10 of them were SPARCstation IPXs (IPX.1 to IPX.10) with a relative speed of 1, and the other 10 workstations were SPARCstation 10s (SPARC10.1 to SPARC10.10) with a speed index of 2.9. Two workloads were used, namely BigMix and BigDenseMix. There were 200 jobs for each workload, and the group size of both workloads were set to [1 : 6]. The group interarrival times of the BigMix workload and BigDenseMix workload were uniformly distributed in the range [10s, 20s] and [5s, 10s] respectively. Although the total workload requirements of BigMix and BigDenseMix were basically
identical, the jobs in BigDenseMix arrived much sooner than those in BigMix. It was designed to test whether the load balancing algorithms could handle sudden arrival of heavy workload. Since the I/O requirements of both BigMix and BigDenseMix were very high, the ratios of CPU-bound, I/O-bound, IPC-bound and MEMORY-bound jobs were scaled down to 7:1:1:1 in order to avoid thrashing the file servers. The resultant batch completion times are shown in Table 5. It can be seen that DELAY* adapts very well to heavy workloads as well as large fluctuation in the workloads. The adaptive power of DELAY* is clearly seen in the similar batch completion times of BigMix and BigDenseMix under DELAY* as their resource requirements are identical. On the other hand, under Utopia/LSF, immediate and random algorithms the slowdown on the batch completion times of BigDenseMix compared to BigMix was significant, being equal to 78.28%, 11.73% and 31.42% respectively. This shows the three algorithms are not able to handle very high level of job arrival density. As shown in Figure 10, Utopia/LSF allocated too many jobs (including some long jobs) towards the end of the batch in BigDenseMix to the slow workstations IPX.2, IPX.3, IPX.4 and IPX.5. These wrong decisions resulted in doubling the batch completion time compared to the case of BigMix. DELAY* did not suffer from this problem. As shown in Figure 11, DELAY* queued up the jobs to slow down the job dispatch rate when the system loading was high. The bounding

<table>
<thead>
<tr>
<th>Type</th>
<th>B</th>
<th>S</th>
<th>T</th>
<th>J</th>
<th>sd(J)</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-bound workload</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELAY*</td>
<td>743.47</td>
<td>17.67</td>
<td>69.19</td>
<td>2.59</td>
<td>1.49</td>
<td>8.48</td>
</tr>
<tr>
<td>DELAY(1)</td>
<td>739.10</td>
<td>17.75</td>
<td>68.55</td>
<td>2.51</td>
<td>1.34</td>
<td>7.36</td>
</tr>
<tr>
<td>Utopia/LSF</td>
<td>797.83</td>
<td>16.47</td>
<td>81.15</td>
<td>2.87</td>
<td>1.25</td>
<td>0.68</td>
</tr>
<tr>
<td>I/O-bound workload</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELAY*</td>
<td>860.73</td>
<td>17.75</td>
<td>60.98</td>
<td>1.71</td>
<td>0.88</td>
<td>2.09</td>
</tr>
<tr>
<td>DELAY(1)</td>
<td>878.49</td>
<td>17.42</td>
<td>65.67</td>
<td>1.83</td>
<td>0.89</td>
<td>2.45</td>
</tr>
<tr>
<td>Utopia/LSF</td>
<td>1073.23</td>
<td>14.38</td>
<td>98.32</td>
<td>2.54</td>
<td>1.78</td>
<td>0.47</td>
</tr>
<tr>
<td>Mix workload</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELAY*</td>
<td>841.44</td>
<td>26.53</td>
<td>158.86</td>
<td>3.64</td>
<td>1.92</td>
<td>15.55</td>
</tr>
<tr>
<td>DELAY(1)</td>
<td>848.57</td>
<td>26.23</td>
<td>161.28</td>
<td>4.20</td>
<td>3.04</td>
<td>34.36</td>
</tr>
<tr>
<td>Utopia/LSF</td>
<td>942.25</td>
<td>23.75</td>
<td>167.88</td>
<td>3.79</td>
<td>2.15</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Table 4: Performance of the adaptive dynamic scheduling algorithms.

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>DELAY*</th>
<th>Utopia/LSF</th>
<th>Immediate</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigMix</td>
<td>1119.39</td>
<td>1273.51</td>
<td>1367.57</td>
<td>1610.41</td>
</tr>
<tr>
<td>BigDenseMix</td>
<td>1062.42</td>
<td>2270.35</td>
<td>1528.05</td>
<td>2116.48</td>
</tr>
</tbody>
</table>

Table 5: Batch completion time (in second) of the scheduling algorithms in a large system.
factor was increased twice from 1 to 3 in the first 500s so that jobs were scheduled in a balanced manner. As a result, the pending jobs were fairly evenly distributed to the 20 workstations which minimized the chance of allocating the long jobs to heavily loaded workstations.

7 Conclusions

The principle of the delay scheduling strategy and a prototype implementation have been presented in this paper. By delaying the jobs in a controlled manner after the system is fully utilized, the workstations are prevented from being overloaded and the workload is distributed more evenly across the workstations in the LAN. This scheme is general and can be incorporated into existing schedulers. A simple load balancing algorithm augmented with the delay strategy was implemented and its performance evaluated. The results indicated that this scheduling algorithm produced better schedules which reduce load imbalance and excessive resource contention compared to other popular schedulers which do not use the delay strategy. By dynamically adjusting the bounding factor, the resultant load balancing algorithms are able to adapt better to fluctuations in system loading and produce better performance in terms of throughput, response time and fairness.
Figure 10: Gantt chart of the Utopia/LSF schedule (BigDenseMix workload).

References


Figure 11: Gantt chart of the DELAY* schedule (BigDenseMix workload).


