The authors propose two strategies: One takes network delay into account to avoid errors in scheduling jobs; the other also delays job execution when the system is fully used. These strategies are general and can augment existing algorithms.

The work speeds of some LANs have exceeded 100 Mbits per second, thereby reducing the penalty of interprocessor communication. So, dispatching tasks from heavily loaded computers to lightly loaded ones to improve system performance is both logical and feasible.1

Researchers have proposed many techniques to allocate jobs dynamically in a LAN or multiprocessor system to improve performance. For simplicity, most algorithms ignore practical issues such as the difference in workstation speeds, sudden batch arrivals, and user behaviors. These schedulers might not perform satisfactorily in practice. Moreover, most algorithms have been evaluated by simulations using the steady-state mean response time as the primary performance metric. Not much research has been done on keeping the load balanced in the transient periods, so the scheduling of batches of jobs (for example, by shell scripts) might greatly upset the state of global balance and cause poor performance. Another common simplification is to dispatch jobs immediately upon arrival, irrespective of the overall system loading. Therefore, the sudden arrival of a large number of jobs might saturate the system. An important difference between a LAN environment and a multiprocessor system is that a LAN usually has a user associated with each workstation. These issues are important in contemporary LAN environments, in which workstations have different capabilities and are highly autonomous.

We’ve developed techniques that remedy these simplistic, inadequate system assumptions. These techniques enable schedulers to adapt to more situations (for example, batch arrivals) and to improve job fairness. (Job fairness means that all jobs receive equal treatment if desired.) We also propose a strategy to delay job execution dynamically when the system is fully utilized. In this way, both the executing and delayed jobs receive better service because overcompetition is eliminated.

These techniques and strategy are general and can be incorporated into existing schedulers. We’ve applied them on a well-known scheduler and compared its performance to other standard schedulers and to the Utopia2 and LSF3 load-sharing software packages. (Load Sharing Facility is the commercial version of Utopia.) The results indicate that the proposed strategy adapts well to load fluctuation, improves job fairness, and produces shorter schedules (from 7.8% to 22.2% shorter) than Utopia and LSF do.
New techniques to improve scheduling on LANs

We differentiate between two kinds of jobs in the LAN environment: local and schedulable. Local jobs either depend on the information of the workstation that generates them (for example, hostname and uname) or are highly interactive (for example, v1, sed and awk). They are best served by local workstations. On the other hand, schedulable jobs are mostly CPU-bound (for example, long-running simulation jobs). We assume that local jobs execute on local workstations immediately upon arrival. Conversely, schedulable jobs are handled by the scheduler and might execute locally or on a remote workstation either immediately or after a delay. Moreover, we assume that job characteristics are not known in advance, that a job is scheduled only once, and that no migration is allowed after a job has started execution. For the rest of the article, job will refer to a schedulable rather than local job, unless we state otherwise.

Taking delay into account

The computing system is a LAN, $W$, which consists of compatible workstations running at different speeds. The speed of a workstation $w_i \in W$ is $s_i$, and $f_i$ is the scheduling function reflecting the desirability of executing a new job on $w_i$. The larger $f_i$ is, the more desirable sending a job to $w_i$. A job is not submitted to $w_i$ if $f_i < 0$ (for example, when $w_i$ is overloaded). In a busy system where $w_{\text{max}}$ waiting for some running jobs to complete is better than dispatching additional jobs to overload an already saturated system. Accordingly, a job should be submitted to $w_{\text{max}}$ only when $f_{\text{max}} \geq 0$.

Intuitively, the best load-balancing status occurs when all workstations are at the point of full utilization (but are not saturated) and each workstation’s workload is proportional to its capacity. Allocating more jobs to a fully utilized system might cause imbalance without improving the overall throughput. Consider the example in Figure 1, where 10 jobs are to be allocated to two workstations. Here, four jobs have been scheduled, with two long jobs in $w_1$ and two short jobs in $w_2$. If long jobs take 100 times longer to complete than short jobs, and if the job type (that is, short or long) is known in advance, it is logical to dispatch the two 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 two jobs are running in each workstation and six jobs are pending. Sending any pending long job to $w_1$ will lengthen the whole batch’s execution time.

The basic strategy

A reasonable scheme is to set $f_i < 0$ as long as $w_i$ is fully utilized. This scheme does not affect system throughput but tends to even out the workload distribution in the LAN. In our example, if we assume a workstation is fully utilized when two jobs are 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. Because all the short jobs are in $w_2$, a job in $w_2$ will more likely be completed first. Therefore, the two pending long jobs have a higher chance of being allocated to $w_2$. In general, this delay strategy reduces the chances of overloading the workstations by setting appropriate bounding factors, and tends to distribute long jobs to the workstations more evenly. As we demonstrate later, adding the delay strategy to a scheduling algorithm reduces the elapsed time of the job groups and improves the responsiveness of individual jobs.

A sample implementation

Figure 2 shows DELAY, a conventional centralized least-load scheduler augmented with the delay strategy. (For more information on least-load and other types of schedulers, see the sidebar.) The algorithm’s input parameters are the speeds ($s_i$) and bounding factors ($b_i$) of the workstations. It obtains the run-queue length $q_i$ (the number of processes in the workstation’s run queue) from each workstation $w_i$, keeps track of the number of jobs $c_i$ allocated to $w_i$, and calculates the number of active jobs $j_i$, which it uses to compute the effective speed $s_i$ of the workstation $w_i$. (We explain why we use $f_i$ and $s_i$ in the next section.)

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). Three events are possible: the update of load information (load_update), the arrival of a new job (new_job), or the 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 checking if the workstation 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 might decrease execution efficiency by introducing resource contention (especially in main memory), causing excessive swapping and context switches. Moreover, heavy resource contention might
cause thrashing, which could prevent the workstations from doing useful work.

**DELAY** fixes these problems by postponing the dispatch of new jobs when the system loading is too high. A workstation’s bounding factor \((b_i)\) limits the number of schedulable jobs in that workstation. The bounding factor is generally set when \(b_i\) jobs are scheduled to \(w_i\), the workstation is fully utilized. However, if response time is important to \(w_i\) (for example, if it’s a user desktop), \(b_i\) should be set to a lower value or even zero.

If \(j\) jobs are running on \(w_i\) and a new job is submitted to \(w_i\), we can approximate the effective speed of \(w_i\) in processing the new job by

\[
\frac{s_j}{f_i + 1}.
\]

The scheduling function is

\[
f_i = s_j - \frac{1}{b_i},
\]

where \(s_j\) is the speed perceived by a job when it is scheduled to \(w_i\) and \(f_i\) is non-negative when the number of scheduled jobs is less than or equal to \(b_i\). 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

\[
s_j \geq \frac{1}{b_i},
\]

and a job is assigned to the fastest workstation \(w_{max}\) when \(f_{max} \geq 0\) (Step 8). If more than one candidate workstation exists (kept in the set \(W_{best}\) in Step 8), **DELAY** randomly selects one of them (Step 8b). If no workstation satisfies the condition, the incoming jobs are held in \(Q_{pending}\) and are dispatched only if some running jobs leave the system (Step 7).

**JOBS SCHEDULING**

Previous studies have suggested that the run-queue length best describes a workstation’s loading, and many dynamic load-balancing algorithms have adopted this metric. However, our research has shown that the run-queue length is inadequate for LAN environments. Workstations often have several coprocessors in addition to the CPU. These can include the I/O processor, the DMA (direct memory access) 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. So, some processes that are actually running might not be in the run queue.

Another factor that often leads to inaccurate job scheduling is the lag time in disseminating load information. To reduce the overhead in collecting load information, most load monitors query and export load information only once every few seconds. Several jobs submitted to the scheduler between load updates might be scheduled to the same workstation. Utopia and LSF try to avoid this problem by updating the load information in the scheduler as soon as a scheduling decision is made. However, as we discuss later, experimental evaluation showed that they are not adaptive enough to handle a very high workload.

**DELAY** eliminates these problems by maintaining the number of active jobs on \(w_i\) (denoted by \(j\) as \(j = \max\{q_j, c_j\}\). \(c_j\) (the number of jobs currently allocated to \(w_i\) by the scheduler) is greater than \(q_j\), if no jobs are local and some jobs are being served by \(w_i\)’s coprocessors. Conversely, \(q_j\) might be greater than \(c_j\) when the console user of \(w_i\) submits local jobs (which bypass the scheduler). We claim that the composite index \(j\) gives a higher priority to the local jobs, which can always execute on the local workstation. On the other hand, remote jobs must satisfy the scheduling rule before they can execute. As we previously mentioned, 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.

---

**Figure 2.** A formal description of DELAY, a conventional centralized least-load scheduler augmented with the delay strategy.
Dynamic load-balancing algorithms

Generally, dynamic load-balancing algorithms take either the least-loaded approach, the threshold-based approach, or the bidding approach. The least-loaded approach attempts to allocate jobs to the least-loaded computers in the system.\textsuperscript{1,2} To reduce the overhead of disseminating load information among the workstations, some least-load algorithms let only immediate neighbors communicate.\textsuperscript{3} Utopia and LSF (Load Sharing Facility, the commercial version of Utopia) use a matching algorithm\textsuperscript{4} to filter out the inadequate workstations. The matching criteria include general resource requirements (for example, CPU and memory) and restrictive resource requirements (for example, architecture). Utopia and LSF group the workstations into small clusters, each with a least-loaded algorithm to select the fastest computers that satisfy the resource requirements. Several algorithms eliminate load-updating messages altogether by observing the network to estimate the loading of the other computers.\textsuperscript{5}

In the threshold-based approach, a workstation triggers load-balancing actions if its load level exceeds a certain threshold.\textsuperscript{6} Threshold-based algorithms take one of three approaches. In the sender-initiated approach, the highly loaded computers dispatch their jobs to computers with lighter loads. In the receiver-initiated approach, the lightly loaded computers request jobs from the busy computers. The symmetrically initiated approach combines these two schemes.\textsuperscript{6}

The bidding approach views the computers as resources and the jobs as consumers. For example, the Spawn system simulates an open financial market, using a form of priority for currency.\textsuperscript{7} Application managers bid for CPU time; only the winners can execute jobs on workstations. However, because the bidding process takes nonnegligible time, it might not work well in the LAN environment, which consists of both short and long jobs with unknown characteristics.

References


Testing the techniques

The testbed for our experiments was Balance, a flexible load-balancing system that supports parallel applications and system schedulers.\textsuperscript{5} It also supports efficient remote execution and interprocess communication and is not tied to a particular scheduling algorithm. For example, baltcsh (an extension of the GNU tcsh) supports the command balsched x, which instructs baltcsh to consult the scheduler named x to select workstations for executing subsequent commands. Users can write their own schedulers and plug them into baltcsh dynamically. In our experiments, we implemented the load monitors and the DELAY algorithm as generic servers to provide load information and scheduling decisions, respectively.

We tested several implementations of DELAY, which we lump under the name DELAY(b), where b is the bounding factor for all workstations. DELAY(\infty) takes into account the delay factor and alleviates the incorrect network assumptions mentioned earlier, but it does not employ the delay strategy. The other implementations of DELAY employed that strategy, with bounding factors ranging from one to five.

In addition to DELAY, we implemented and tested these scheduling algorithms:

- **Immediate**: This centralized algorithm immediately dispatches the submitted jobs to the workstation with the highest effective speed. By defining \( j_s = q_s \), we can study how the delay in updating load information affects system performance.
- **Random**: This distributed dynamic scheduler schedules each job to run on a randomly selected workstation based on that workstation’s relative speed. Specifically, the probability of scheduling a job to \( w \) is equal to \( j_s/\sum_j j_s \).
- **Utopia/LSF**: Utopia and LSF have a sophisticated scheduler for load balancing in large systems.\textsuperscript{2} In our experiments, we installed both Utopia and LSF v2.2 and treated them as black boxes that provided scheduling decisions only. Because they performed similarly, we present only one set of results for them.

**Workload characteristics**

The experiments’ main objective was to investigate DELAY’s ability to balance heavy workloads that fluctuate widely, rather than to compare the performance of the different scheduling algorithms. Because the jobs are not delayed when system loading is light, we chose heavy workloads that sometime result in full system utilization. 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 ranges [1–6] and [9–28] seconds. Each job has these types of resource requirements:

- **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.
Disk access (I/O): For each I/O unit, the job writes 1 Mbyte of data to the disk and then reads it back. Because I/O operations are performed on either the local disk or the shared file system, we define the parameter IO-L as the probability that I/O operations execute locally.

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 IPC unit, it sends 1 Mbyte of data to the slave and receives an acknowledgment from it.

Memory operation (Memory): For each memory-operation unit, the job allocates and frees 100 Kbytes of memory.

Number of phases (P): The above operations are evenly divided into P phases that a job executes in an interleaved manner.

We tested the scheduling algorithms under three types of workloads: a CPU-bound workload, an I/O-bound workload, and a mix workload. Each workload (batch) consists of 100 jobs; Table 1 summarizes the workloads’ characteristics. The mix workload consists of 40% CPU-bound jobs, 40% I/O-bound jobs, 10% IPC-bound jobs, and 10% Memory-bound jobs. The workloads contain no local jobs.

**The Evaluation-System Architecture**

We conducted the experiments in a controlled environment consisting of seven workstations connected by a LAN (see Table 2). The software architecture comprised the set of communicating processes shown in Figure 3. The driver receives 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 might in turn spawn a slave to perform interprocessor communication.

**Performance Metrics**

The schedulers make no assumption on the nature of the jobs. To compare their performance, we selected six metrics:

- **The batch completion time (B)** is the elapsed time for executing the entire workload. This metric measures system throughput and reflects whether the jobs are allocated to the workstations proportionally 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 B is, the more even the workload distribution and the higher the throughput.
- The overall speedup (S) is defined as $S = B_1/B$, where $B_1$ is the sum of the job completion times when all the jobs execute one at a time on the slowest workstation (that is, IPX.1).
- The mean response time (T) is the average completion time of the jobs in each batch. Previous studies used this metric as an indication of scheduling performance.
- The mean job completion time ratio ($\bar{f}$) is the average of $f$, the job completion time, for each job.
time ratio, over all the jobs. For each job, \( J = E_t/E_i \), where \( E_t \) is the shortest possible elapsed time when the job executes on the fastest idle machine (in this case, the Sparc20.1), and \( E_i \) is the elapsed time when the job executes in competition with the other jobs. Compared to \( T \), \( J \) eliminates the effect of job size on individual job performance. The smaller \( J \) is, the better the performance.

• **Standard deviation of \( J (sd(J)) \):** This is the standard deviation of \( J \) over all jobs. This metric indicates whether the jobs receive equal treatment. The smaller \( sd(J) \) is, the fairer the scheduling algorithm.

• **Average waiting period \( \bar{W} \):** This is the mean of \( W \), the waiting period, over all the jobs. \( W \) is the percentage of time a job spends in the pending queue relative to its execution time. If \( W = 200\% \), the job spends 2x time units waiting to be scheduled and \( x \) time units in actual execution.

\( B \) and \( S \) represent the system throughput of the jobs as a batch, while \( T \) and \( J \) reflect the average performance of individual jobs. A scheduling algorithm might produce small \( T \) and \( J \) but large \( B \) and \( S \) if the workload is not distributed fairly evenly to the workstations. \( sd(J) \) and \( \bar{W} \) summarize the schedule’s fairness and \( \text{DELAY}^{(\infty)} \)’s overhead, respectively.

**RESULTS AND DISCUSSION**

Table 3 summarizes the experimental results. In general, Random’s throughput \( (B, S, \text{ and } T) \) is the worst because Random does not use current load information. It outperforms Immediate only on I/O-bound and mix workloads because Immediate makes wrong scheduling decisions in these cases (see the next section). \( \text{DELAY}(b) \) performs the best overall. \( \text{DELAY}^{(\infty)} \) produces a 4.8% (CPU-bound), 12.8% (I/O-bound), and 10.2% (mix) improvement over Utopia/LSF; adding the delay strategy produces an additional 2.8%, 6.2%, and 0.2% improvement over \( \text{DELAY}^{(\infty)} \) in batch completion time. So, both enhancements improve the scheduling quality.

**The effect of lag time in load updates**

The delay in updating load information is a critical factor in scheduling performance; taking this factor into account greatly improves scheduling performance. Although the load information was updated every second, Immediate allocated most jobs in the same group to the same workstation. Utopia/LSF tries to alleviate this problem by load preadjustment. However, it suffered from overallocation in many situations.

Figure 4 shows a Gantt chart of a Utopia/LSF schedule where each horizontal line presents a job’s execution profile. Each line consists of three points: the first is the job submission time, the second is the execution start time, and the third is the job finish time. With Utopia/LSF, because a job starts execution as soon as it is submitted, the first two points occur at the same time. According to Figure 4a, many jobs in the

---

Table 2. Configuration of the workstations.

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>CPU TYPE</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>SuperSparc Model 20/50</td>
<td>4.0</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>18.8</td>
</tr>
</tbody>
</table>

---

Table 3. Scheduling algorithm performance. \( B \) is the batch completion time; \( S \) is the overall speedup; \( T \) is the mean response time; \( J \) is the mean job completion time; \( sd(J) \) is the standard deviation of the job completion time ratio; and \( W \) is the average waiting period. \( B, T, J \), and \( W \) are measured in seconds.

<table>
<thead>
<tr>
<th>WORKLOAD/SCHEDULER</th>
<th>CPU-bound</th>
<th>I/O-bound</th>
<th>Mix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( B )</td>
<td>( S )</td>
<td>( T )</td>
</tr>
<tr>
<td>Random</td>
<td>937</td>
<td>14.23</td>
<td>117.2</td>
</tr>
<tr>
<td>Immediate</td>
<td>831</td>
<td>15.77</td>
<td>86.1</td>
</tr>
<tr>
<td>Utopia/LSF</td>
<td>798</td>
<td>16.47</td>
<td>81.2</td>
</tr>
<tr>
<td>( \text{DELAY}(1) )</td>
<td>739</td>
<td>17.75</td>
<td>68.6</td>
</tr>
<tr>
<td>( \text{DELAY}(2) )</td>
<td>757</td>
<td>17.30</td>
<td>69.3</td>
</tr>
<tr>
<td>( \text{DELAY}(3) )</td>
<td>789</td>
<td>17.07</td>
<td>68.7</td>
</tr>
<tr>
<td>( \text{DELAY}(4) )</td>
<td>760</td>
<td>17.26</td>
<td>67.9</td>
</tr>
<tr>
<td>( \text{DELAY}(5) )</td>
<td>759</td>
<td>17.27</td>
<td>69.8</td>
</tr>
<tr>
<td>( \text{DELAY}^{(\infty)} )</td>
<td>760</td>
<td>17.22</td>
<td>70.5</td>
</tr>
<tr>
<td>Random</td>
<td>1,102</td>
<td>13.89</td>
<td>103.6</td>
</tr>
<tr>
<td>Immediate</td>
<td>1,167</td>
<td>13.15</td>
<td>143.8</td>
</tr>
<tr>
<td>Utopia/LSF</td>
<td>1,073</td>
<td>14.38</td>
<td>98.3</td>
</tr>
<tr>
<td>( \text{DELAY}(1) )</td>
<td>878</td>
<td>17.42</td>
<td>65.7</td>
</tr>
<tr>
<td>( \text{DELAY}(2) )</td>
<td>891</td>
<td>17.16</td>
<td>68.4</td>
</tr>
<tr>
<td>( \text{DELAY}(3) )</td>
<td>904</td>
<td>16.95</td>
<td>65.5</td>
</tr>
<tr>
<td>( \text{DELAY}(4) )</td>
<td>900</td>
<td>17.04</td>
<td>66.3</td>
</tr>
<tr>
<td>( \text{DELAY}(5) )</td>
<td>909</td>
<td>16.85</td>
<td>68.8</td>
</tr>
<tr>
<td>( \text{DELAY}^{(\infty)} )</td>
<td>936</td>
<td>16.32</td>
<td>75.9</td>
</tr>
</tbody>
</table>

---
same groups were scheduled to the same workstation. For example, six jobs were dispatched to Sparc10.2 and Sparc10.3 at 456 and 250 seconds, respectively.

In contrast, DELAY used the number of active jobs ($j_i$) when selecting the fastest workstation. DELAY maintained the number of scheduled jobs ($c_i$) at the scheduler by incrementing it each time a job is dispatched to workstation $i$. So, it did not underestimate the workstation loadings even though there was non-negligible lag time for the current load indices to arrive from the workstations. At most, it overestimated the loading by one for a second. At high loads, slight underestimation is not an issue; at light loads, all the schedulers perform similarly anyway. So, even DELAY($\infty$) (that is, with the delay strategy disabled) produced significant improvement over the other scheduling algorithms.

**Batch completion time and speedup**

The delay strategy effectively balances the load in the workstations and keeps the system from being saturated. DELAY 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 the CPU-bound workload), 22.2% (for the I/O-bound workload), and 11.6% (for the mix workload), compared to Utopia/LSF. For all workstations, DELAY(1) achieved speedups of 17.75 for CPU-bound workloads and 17.42 for I/O-bound workloads. This is very close to the optimal speedup of 18.8, the total computing power of the seven workstations (see Table 2). By putting an upper limit on the number of schedulable jobs on each workstation, DELAY effectively avoids excessive contention and produces better batch completion times. Moreover, it reduces the probability of making bad choices of workstations for large jobs by postponing the scheduling decision until a machine with sufficient resources is available.

In addition, because DELAY limits the number of jobs in each workstation, imposing an upper bound on the workstation loading is possible. Therefore, local jobs can be guaranteed some fraction (or all) of the local workstation’s capacity. For the other scheduling algorithms, this bound does not exist. For example, we can determine from Figure 4a that the maximum run-queue lengths for Sparc10.2 and Sparc10.3 in the Utopia/LSF schedule were 11 and 16. With DELAY, we can easily trade off console responsiveness with system scheduling quality by adjusting the bounding factors of the workstations.

Because the whole batch completes only when all the jobs are done, a wrong allocation would adversely affect the batch completion time. For example, Figure 4a shows that Utopia/LSF scheduled several long jobs to the same workstation (for example, Sparc10.2) while some workstations were starving from insufficient jobs (for example, IPX.1). DELAY did not have this problem (Figure 4b shows a typical schedule). When all the workstations are fully utilized, it does not schedule an incoming job until a job has departed. This keeps the system relatively balanced and distributes the long jobs more evenly across the workstations.

**The effect of multiple processors in a workstation**

For mix workloads, all algorithms achieved superlinear speedups. This is because jobs with different resource requirements were served by different processors in a workstation simultaneously (for example, the CPU and I/O processors served the CPU-bound and I/O-bound...
This also explains the relatively poor performance of Immediate and Utopia/LSF for I/O-bound and mix workloads. These two types of workloads involve mainly I/O-bound, Memory-bound, and IPC-bound jobs, which do not use the CPU often. Therefore, for much of the time these jobs use system resources without being in the run queue. Consequently, Immediate and Utopia/LSF, which assume that the run-queue length best describes the loadings of the workstations, fail to balance the workload effectively. DELAY, on the other hand, performs much better by using the total number of active jobs on each workstation.

The performance of individual jobs

The mean response time $\bar{T}$ \[\text{is not directly proportional to the batch completion time} \] $B$, although a schedule with small $T$ usually has small $B$. For example, although the mean response time of Immediate (181.01 s) under the mix workload is smaller than that of Random (190.47 s), its batch completion time is longer (1060.50 s versus 983.47 s). This implies that a scheduling algorithm that minimizes $T$ might not necessarily maximize the batch throughput. The loads at all the workstations must be balanced as a whole to minimize $B$.

Interestingly, DELAY’s mean job completion time ratio $J$ was also the best among the algorithms tested (except for the case of DELAY(1) under the mix workload, which we’ll discuss later). Moreover, the jobs also received fair treatment (that is, $s_i(j)$ was the smallest). This is not intuitive; DELAY postpones the execution of some jobs, so we would expect a larger variation on $J$. The explanation is that a poor schedule (that is, one that produces large $B$ and small $S$) has larger job response times in the overloaded workstations. From the experiment’s result, we know that if the bounding factor is sufficiently large (in our case, 2), most of the jobs are not delayed, so the job wait time in DELAY is negligible. This implies that a suitable level of delay can improve the overall throughput without sacrificing responsiveness.

An adaptive delay strategy

To improve DELAY’s adaptability to workload variations, we propose an improved, adaptive algorithm, DELAY*. Its main objective is to tune the bounding factor $b_i$ to improve response times as the load changes, while maintaining the system’s overall balance.

As Figure 5 shows, DELAY* is different from DELAY in several aspects. The user no longer needs to specify the initial values of $b_i$. DELAY* initializes them to 1 in Step 3 and updates them in Steps 9 and 10, depending on the pending queue’s length (that is, $|Q_{\text{pending}}|$). If the queue grows and exceeds the aggregate workstation speeds (that is, $\sum s_i$), the bounding factor increases by 1 in Step 9. In this way, DELAY* distributes the pending jobs to the workstations proportionally to their speeds. This allows the bounding factor to be adjusted to the most suitable value without introducing imbalance in the system. In contrast, Step 10 checks the bounding factor to see if it has grown too large relative to the workload. If the current speed of $s_i$ (that is, $s_i/j_i$) is greater than or equal to $1/(b_i-1)$ 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 $i$.

We evaluated DELAY* using the exper-

---

**Figure 5. A formal description of DELAY*, an adaptive version of DELAY.**
Figure 6. A Gantt chart of the Delay* schedule (mix workload).

Table 4. The performance of the adaptive dynamic scheduling algorithms. $B$ is the batch completion time; $S$ is the overall speedup; $T$ is the mean response time; $J$ is the mean job completion time; $SD(J)$ is the standard deviation of the job completion time; and $W$ is the average waiting period. $B$, $T$, $J$, and $W$ are measured in seconds.

<table>
<thead>
<tr>
<th>WORKLOAD/SCHEDULER</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</strong></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><strong>I/O-bound</strong></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>1,073.23</td>
<td>14.38</td>
<td>98.32</td>
<td>2.54</td>
<td>1.78</td>
<td>0.47</td>
</tr>
<tr>
<td><strong>Mix</strong></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>

BigDenseMix. Each workload had 200 jobs, and the group sizes of both workloads were uniformly distributed in the range $[1–6]$. The group interarrival times of the BigMix workload and BigDenseMix workload were uniformly distributed in the ranges $[10 s–20 s]$ and $[5 s–10 s]$. Although BigMix and BigDenseMix had basically identical total workload requirements, the jobs in BigDenseMix arrived much sooner than those in BigMix. We did this to test whether the load-balancing algorithms could handle the sudden arrival of a heavy workload. Because the I/O requirements of both BigMix and BigDenseMix were very high, we scaled down the ratios of CPU-bound, I/O-bound, IPC-bound, and Memory-bound jobs to 7:1:1:1 to avoid thrashing the file servers. Further increasing the resource requirements of BigMix and BigDenseMix would cause overall system saturation.

Table 5 lists the resultant batch completion times, which show that Delay* adapts very well to a heavy workload as well as to a large fluctuation in the workload. The adaptive power of Delay* is evident in the similar batch completion times of BigMix and BigDenseMix (which have the same resource requirements). On the other hand, under Utopia/LSF, Immediate, and Random, batch completion times for BigDenseMix were significantly larger (78.3%, 11.7%, and 31.4%, respectively) than for BigMix. This shows the three algorithms cannot handle a very high level of job-arrival density. As Figure 7a shows, Utopia/LSF allocated too many jobs (including some long jobs) toward the end of the batch in BigDenseMix to the slow workstations IPX.2, IPX.3, IPX.4, and IPX.5. These wrong decisions doubled the batch completion time compared to that of BigMix.

Delay* did not suffer from this problem. As Figure 7b shows, Delay* queued up the jobs to slow down the job dispatch rate when the system loading was high. It increased the bounding factor twice (from 1 to 3) in the first 500 s, so that jobs were scheduled in a balanced manner. So, the pending jobs were fairly evenly distributed to the 20 workstations, which
minimized the chance of allocating the long jobs to heavily loaded workstations.

**WE´VE SHOWN THAT BY** applying the delay strategy to existing scheduling algorithms, the schedulers adapt better to fluctuations in system loading and produce better performance in terms of throughput, response time, and fairness. The proposed strategy makes minimal assumptions on the scheduler, is easy to implement, and has negligible operational overhead. It can be implemented on top of most existing commercial schedulers. A distributed version of the delay strategy would benefit the next-generation schedulers that work in multidomain environments involving many computers.

**ACKNOWLEDGMENTS**

A grant from the Research Grant Council of Hong Kong partially supported this work.

**References**


**Chi-Chung Hui** is a member of the technical staff at Sun Microsystems. His research interests include software design for Internet systems and load-balancing and scheduling algorithms in multidomain systems. He received his PhD and MPhil in computer science from the Hong Kong University of Science and Technology and his BSc in computer science from the Chinese University of Hong Kong. Contact him at the Dept. of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong; cchui@cs.ust.hk; http://www.cs.ust.hk/~cchui.

**Samuel T. Chanson** is a professor in, and the associate head of, the Computer Science Department at the Hong Kong University of Science & Technology. His research interests include communication protocols, multimedia communication, distributed systems, and Internet technologies. He was a general cochair of the 1998 IEEE International Conference on Distributed Computing Systems and will be the conference cochair of IFIP FORTE/PSTV in 1999. He serves on the editorial boards of IEEE/ACM Transactions on Networking, New Generation Computing, and the Journal of Computing and Information. He received his PhD in electrical engineering and computer sciences from the University of California, Berkeley. Contact him at the Dept. of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong; chanson@cs.ust.hk; http://www.cs.ust.hk/~chanson.

**Table 5. The batch completion time (in seconds) of the scheduling algorithms in a large system.**

<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>1,119.39</td>
<td>1,273.51</td>
<td>1,367.57</td>
<td>1,610.41</td>
</tr>
<tr>
<td>BigDenseMix</td>
<td>1,062.42</td>
<td>2,270.35</td>
<td>1,528.05</td>
<td>2,116.48</td>
</tr>
</tbody>
</table>

**Figure 7. Gantt charts of schedules for the BigDenseMix workload: (a) Utopia/LSF; (b) DELAY*.**