Appeared in the Proceedings of the ACM Sigmetrics Conference, May, 1993, pp. 86-97.

The Process-Flow Model: Examining I/O Performance from the System's Point of View

Gregory R. Ganger, Yale N. Patt
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor 48109-2122
ganger@eecs.umich.edu

Abstract

Input/output subsystem performance is currently receiving considerable research attention. Significant effort has been focused on reducing average I/O response times and increasing throughput for a given workload. This work has resulted in tremendous advances in I/O subsystem performance. It is unclear, however, how these improvements will be reflected in overall system performance. The central problem lies in the fact that the current method of study tends to treat all I/O requests as equally important. We introduce a three class taxonomy of I/O requests based on their effects on system performance. We denote the three classes *time-critical*, *time-limited*, and *time-noncritical*. A system-level, trace-driven simulation model has been developed for the purpose of studying disk scheduling algorithms. By incorporating knowledge of I/O classes, algorithms tuned for system performance rather than I/O subsystem performance may be developed. Traditional I/O subsystem simulators would rate such algorithms unfavorably because they produce suboptimal subsystem performance. By studying the I/O subsystem via global, system-level simulation, one can more easily identify changes that will improve overall system performance.

0

1 Introduction

Input/output subsystem performance is currently receiving considerable research attention. Significant effort has been focused on reducing average response times and increasing throughput for a given workload (whether synthetically generated or traced from real-world environments). The results of this work are tremendous advances in I/O subsystem performance. Unfortunately, in many cases it is not clear how these improvements will be reflected in overall system performance. The central problem lies in the fact that the current method of study tends to treat all I/O requests as equally important. We believe, however, that there are at least three distinct classes of I/O requests in terms of their effect on system performance. We denote these three classes of I/O requests as *time-critical*, *time-limited* and *time-noncritical*.

A request is time-critical if the process (the term we will use to describe any instruction stream) which generated it, either implicitly or explicitly, must halt execution until the request has completed. Time-limited requests are those which must be completed within some amount of time or they will become time-critical. Lastly, time-noncritical signifies a request which does not require any process to wait for its completion, yet must be completed to maintain the accuracy of the non-volatile storage and to free any resources (e.g., memory) that are held on its behalf.

We denote the two most important negative performance consequences of input/output as *false idle time* and *false computation time*. False idle time is that time during which a processor executes the idle loop because all active processes are blocked waiting for I/O requests to complete. This is differentiated from regular idle time which is due to a lack of available work in the system. False computation time denotes time that is wasted handling a process that has blocked waiting for an I/O request to complete. This includes the time required to disable the process, context switch to a new process and, upon completion, re-enable the process.

To reduce the negative impacts of I/O on system performance, time-critical requests should be completed as quickly as possible, time-limited requests should be completed within their time limits, and time-noncritical requests should be completed in a timely manner with deference to the first two I/O classes. Of course, achieving these goals requires that the I/O subsystem have full knowledge of the appropriate classification for each incoming request.

Taking a "systems" approach to I/O research may require a methodology beyond conventional trace-driven simulation of the I/O subsystem and graphs of response time and throughput for streams of I/O requests. These component metrics fail to capture the effect of I/O subsystem modifications on overall system performance. To understand the performance interaction between the I/O subsystem and the rest of the system, a more complete method of study is required. Toward this end, the conventional subsystem simulation model is supplemented with a system-level model and the I/O traces are replaced by traces of system activity (including I/O activity). We call our simulation environment the *process-flow model.*

The process-flow model reproduces the flow of work in the system and its interaction with the I/O subsystem. Rather than focusing simply on the stream of I/O requests emanating from the system, the process-flow model focuses on the processes which reside in the system at any point in time. Each process is represented by a stream of I/O requests and other important events which may change the state of the process (or another process). In UNIX ^{TM} such events would include fork, exit, sleep, wakeup, kernel entry, and kernel exit. The time between each event is computation (think) time. External interrupts, such as clock interrupts, are also included in the simulation. I/O requests are sent to an I/O subsystem simulator which generates interrupts in the process-flow model at the appropriate times. By simulating the way in which processes interact with their I/O requests, this model provides for recognizing the class to which each I/O request belongs. In addition, changes in I/O performance can be measured in terms of their effect on overall system performance.

Our contribution is not the development of a new model of system activity, but rather the use of this model in examining the performance interaction between the I/O subsystem and the rest of the system. We have instrumented the operating system to produce traces of system activity for input to our process-flow simulator as well as for validation purposes. Our initial experiments center on alternative disk scheduling strategies using class information about pending I/O requests. We find that recognizing and scheduling requests based on class membership improves system performance even though I/O performance as measured by component metrics is degraded. This observation indicates the potential importance of system-level models like the process-flow model in focusing the study of I/O performance on those aspects which will most improve overall system performance.

The remainder of the paper is organized as follows. Section 2 acknowledges previous work related to studying the I/O subsystem in terms of system performance and identifying classes of I/O requests. Section 3 expands the description of our I/O request taxonomy. Section 4 describes our process-flow model. Section 5 discusses the remainder of our experimental apparatus. Section 6 investigates disk scheduling algorithms which use knowledge of I/O request classes to improve system performance. Section 7 presents our conclusions and some areas for future work.

2 Previous Work

As with many other topics in computer design, one finds discussions of system-level modeling in the archives of IBM. [Seam69] and [Chiu78], for example, describe such modeling efforts used mainly for examining various system configurations and evaluating proposed changes. Our process-flow model revisits this methodology and extends it. [Haig90] describes a system performance measurement technique which is very similar to our tracing methodology. However, the end purpose for this technique is to measure system performance under various workloads rather than as input to a simulator to study various system design options. [Rich92] describes a set of tools under development which are intended to allow for studying I/O performance in much the same fashion as ours. The major difference is that they are basing their tools on instruction-level traces. While certainly the ideal case (i.e., simulating the entire activity of the system is more accurate than abstracting part of it away), we do not believe that it is practical at this time. The enormous simulation times and instruction trace storage requirements, as well as the need for instruction level traces of operating system functionality, make this approach both time- and cost-prohibitive. Finally, [Mill91] makes use of a simple system-level model to study the effects of read-ahead and write buffering on supercomputer applications.

Although we have found no previous work which specifically attempts to classify I/O requests as is done in this paper, previous researchers have noted differences between various I/O requests. Many have recognized that synchronous (time-critical) file system writes are undesirable [Oust90] [McVo91]. [Ruem93] recognizes the difference between synchronous (time-critical) and asynchronous (time-limited and time-noncritical) disk requests. They captured this information (as flagged by the file system) in their extensive traces of disk behavior. They found that 50-75% of disk requests are synchronous, partially due to the write-through meta-data cache on the systems traced. It has also been noted that bursts of delayed (time-noncritical) writes caused by periodic update policies can seriously degrade performance. [Cars92] and [Cars92a] argue that disk cache performance should be measured in terms of its effect on read performance. They study different update policies with analytical models using read performance as their metric. They suggest that a disk scheduling algorithm which gives preference to reads may provide significant improvements. While not describing and distinguishing between the classes of I/O requests as we have done, they do make a solid distinction between two types of I/O requests based on process interaction.

3 I/O Request Taxonomy

3.1 Three Classes

We separate I/O requests into three classes — time-critical, time-limited and time-noncritical. This taxonomy is based upon the interaction between the I/O request and the process which generated it (explicitly or implicitly). An I/O request is time-critical if the process which generated it must stop execution until the I/O is complete. Examples of time-critical I/O requests include page faults, synchronous file system writes and database block reads. Time-limited requests are those that will become time-critical if not completed within some time limit. Beneficial file system read-aheads are an example of this. The third class of requests is time-noncritical. These are the requests which no process waits for but which must be completed to maintain a stable copy of the data and to free the resources (e.g., memory) held on their behalf. Examples of time-noncritical requests are delayed file system writes and database writes for which the log has already been written.

Both time-critical requests and time-noncritical requests can be viewed as special cases of time-limited requests. Time-critical requests are time-limited requests with infinitely short time lmiits, whereas time-noncritical requests are time-limited requests with an infinite time limit.

3.2 Performance Impact

Storage accesses interfere with system performance in several ways. Some of these, such as increased system bus and memory bank contention, depend simply on the quantity and timing of the accesses and are essentially independent of our taxonomy. Others, such false computation time and false idle time, are highly dependent on the class of the I/O request. We designate false computation time as that computation required to stop a process that must wait for an I/O request. This time includes removing the process from the runnable queue, context-switching to another process, and re-enabling the original process when the I/O completes. We designate false idle time as the time that the CPU executes the idle loop because all processes are waiting for I/O requests to complete rather than because there is no work in the system at all.

Time-critical accesses, by definition, cause the processes which initiate them to block and wait for their completion. In addition to false computation time, false idle time is accumulated if there are no other processes which can be executed when the current process blocks. This is certainly the largest concern, as it completely wastes the CPU for some period of time (independent of the processor speed) rather than a set number of cycles. To reduce false idle time, time-critical requests must be expedited.

Time-limited requests are similar to time-critical requests in their effect on performance. The major difference is that they are characterized by a time window in which they must complete. If completed within this window they cause no false idle time and no false computation time. Therefore, it is critical to complete these requests within their time limits. Time-limited requests are often speculative in nature, such as read-aheads. If such prefetched blocks are not accessed, performance may be nagetively affected. Sending any request to the I/O subsystem and handling the subsequent interrupt(s) requires CPU time, which in this case can be considered false computation time. Any useless requests will not directly cause any process to stop, but they may interfere with the completion of time-critical and useful time-limited requests. Also, the cache of disk blocks will be polluted with useless blocks, which may lower its hit rate causing additional I/O requests to be generated.

Time-noncritical requests impact performance indirectly. They can interfere with the completion of time-limited and time-critical requests, causing added false idle time and false computation time. Time-noncritical requests are often write requests which must be completed both to bring the on-disk copy of the data up-to-date and to free the system resources (e.g., memory) that are held on their behalf. Delays in completing these requests can reduce the effectiveness of the in-memory disk cache. Because dirty blocks cannot be replaced until their contents have been written to the next lower level in the hierarchy, either other (potentially useful) blocks must be replaced or the process which needs the new block must wait for the write of the dirty block to be completed. The first option can reduce the hit rate of the cache causing additional I/O requests. The second option has the effect of converting the time-noncritical request into a time-critical request. Time-noncritical requests can also be given time limits if the guarantees offered by the system require that written data must reach stable storage within a given amount of time. These requests are not really time-limited because a process will not block if they don't complete within the guaranteed time, but the I/O subsystem must be designed to match such guarantees. Further, these time limits are usually sufficiently large to offer a good deal of latitude in scheduling.

3.3 Workloads

For purposes of clarification, we will describe several possible workloads in terms of our I/O taxonomy.

Consider a workload which consists solely of time-noncritical requests. This would be the case if all reads are handled by the in-memory disk cache and the only requests to the I/O subsystem are updates for dirty cache blocks and background activity such as disk reorganization. Given such a workload, changes to the I/O subsystem could alter system performance in two ways. First, if the time required to compute time-noncritical requests were increased, a less effective cache would result. Second, if the subsystem throughput was insufficient to handle the flow of time-noncritical requests, the I/O subsystem would become a true bottleneck to overall system performance. As an aside, we note that (ignoring these two possible performance problems) this is precisely the type of system which is modeled by trace-driven simulation of the I/O subsystem. That is, changes to the completion times of I/O requests do not effect the generation of subsequent requests.

Another possible workload could consist exclusively of time-critical requests. This might occur if all writes were synchronous and there was no read-ahead. Such a system would tend to be limited by the I/O subsystem. Certainly false computation time could be a problem and, if there were not enough work to cover the time during which processes are blocked, false idle time would be evident. If the computation time between I/O requests is assumed to be zero, this represents the type of system modeled by I/O subsystem simulators assuming a constant number of outstanding requests.

Far more likely than either of these cases is a mixture of time-critical, time-limited and time-noncritical requests. In such a workload, false computation time will accumulate as time-critical requests are serviced. This is also true for time-limited requests which fail to complete within their time limits. False idle time will fill the time during which all active processes are blocked waiting for I/O requests. To reduce false idle time, it is important to complete time-critical requests as quickly as possible. Further, time-limited requests must be completed within their time limits to avoid both false computation time and false idle time. Finally, time-noncritical requests should be completed in a timely manner with deference to the goals related to time-limited and time-critical requests. If there is sufficient work for the CPU, it is not clear whether it is more important to handle time-limited requests or to handle time-critical requests. Another open question is when (if ever) delaying time-noncritical requests will have the negative indirect impacts on performance described above. These are items for future research.

In addition to using class information to tune the I/O subsystem for system performance, this information can be used to improve the storage management software which utilizes the I/O subsystem. The improvements should center on reducing the number of I/O requests (obviously), but also on reducing the criticality associated with each request. For example, time-critical requests should be made time-limited or time-noncritical. Also, the time limits associated with time-limited requests should be maximized.

3.4 Class Identification

Dynamically identifying the class to which an I/O request belongs should not require changes to user applications. Some changes to the system code which makes calls to the device driver may be required. We assume that the interface between these two components is clean, as with the DDI/DKI [DDI90] in UNIX ^{TM}. The key then is to get the class information past the interface. The various pieces of system code which call the device driver to initiate the requests can generally infer the proper I/O classification based on their higher-level function. Examples of such code include the file system, virtual memory manager and application processes (via system calls). If the class for a request is not known at the time it is initiated, the identification must be done through a new interface to the device driver.

Event	Description
Fork	Process creation
Exit	Process completion
Sleep	Disable current process
Wakeup	Enable blocked process
System call	Kernel entry
Trap	Kernel entry (if user)
Return	Kernel exit
Dispatch	Make dispatch decision
Switch	Context switch
Request	I/O request generation
Access	I/O access initiation
Interrupt	Interrupt system
Int. end	Interrupt complete

Table 1: List of major events in process-flow model.

Identifying the time limits associated with time-limited requests is more difficult. For example, the time-limit for a beneficial file system read-ahead will depend upon how much computation must be done before the data is required and whether the process in question gets context switched out and must wait for another (probably higher-priority) process. Therefore, the time limit can only be estimated. This estimate should be as high as possible to allow latitude in scheduling decisions, but at the same time as low as necessary to avoid the penalties associated with overshooting the actual time limit. Examining various methods of estimating the time limit for time-limited requests and the penalties associated with each is an item for future research.

4 Process-Flow Model

The process-flow model is a high-level, trace-driven simulator of a computer system. It models the processor(s), the processes, the dispatcher, the interrupt controller, and the I/O subsystem (including the device drivers). It is driven by traces of key system-level events (see table 1). For validation purposes, the model is designed specifically to model NCR's System 3000 level 3, 4, and 5 computer systems running under a version of UNIX ^{TM}. However, it has been built in a highly modular, parameterized fashion to allow the study of a wider range of design choices.

Because of the high-level nature of the events which drive the model, some of the hardware-specific behavior is not modeled. For example, we do not model the cold-cache effect caused by context switches. Further, we do not model system bus contention or memory bank contention caused by multiple processors and I/O requests. While these effects will reduce the accuracy of our results to some degree, we believe that the abstraction is not unreasonable. A detailed study of how this abstraction affects our simulation results and how these issues might be incorporated is an item for future research.

5 Experimental Apparatus

5.1 Trace Collection

To validate and experiment with the process-flow model, we have used traces of real system activity. SVR4 MP UNIX ^{TM}, a production multiprocessor version of System V Release 4 UNIX developed at NCR Columbia, has been instrumented to collect such traces. The information is gathered in real-time and stored in kernel memory. These traces allow us to accurately recreate the system activity present during the traced period. For each event, we capture a timestamp, the event type and any additional information required to fully understand the event (such as the new process id for a fork or the interrupt vector for an interrupt). The resolution of the timestamps is less than 840 nanoseconds. Our instrumentation was designed to be as unobtrusive as possible, increasing the dynamic instruction count by less than 0.1% in the worst case. Also, to prevent unnecessary paging due to the use of main memory as a trace buffer, the trace machine was equipped with additional main memory. The experimental system was an NCR 3433, a 33MHz Intel 80486 machine equipped with 48 MB of main memory (44 MB for system use and 4 MB for the trace buffer) and 535 MB of secondary storage (a Maxtor LXT-535SY hard drive).

5.2 Traced Workloads

While many traces were gathered for use in validating the simulator, the data presented in section 6 are for only one of these traces. We chose a file compression trace for the experiments due to its significant I/O requirements. The trace was captured while compressing two large (30-40 MB) files simultaneously. Two processes (one to compress each file) were started at approximately the same time. The completion times presented in section 6 represent the times when the last of the two processes ends execution.

To understand the behavior of the I/O stream in the trace, one must understand the nature of the sources of the I/O requests. The compression processes sequentially read a large file and sequentially write out a large file. The first read request is time-critical because the block is not in the cache. The file system, which is based on the Berkeley Fast File System [McKu84], also generates a time-limited read-ahead request. Each time the previous read-ahead block is accessed, a time-limited read request is generated for the next block. So, essentially all of the reads are time-limited. The writes, however, are delayed by the file system so that the processes do not have to wait for them to actually be sent to disk. The dirty blocks are sent to the I/O subsystem at some later point in time by a system-resident background process awakened once per second. It checks the blocks in one-sixtieth of the cache, marks all dirty blocks and initiates write requests for them. This algorithm represents a significant reduction in write burstiness (as studied in [Cars92]), but does not completely alleviate this phenomenon. With the exception of one time-critical write caused by each file creation, all of the writes are time-noncritical.

5.3 Simulator Validation

As mentioned above, the tracing mechanism gathers enough information to allow for both execution and validation of the process-flow model. The process-flow model was validated by using an I/O subsystem model which uses the additional I/O access information to recreate the exact behavior of the traced I/O subsystem. Utilizing this model, our process-flow simulator produced an identical ordering of events, with timestamps

Processors	1
Clock Interrupt Frequency	≈10 ms
Avg. Clock Intr. Handler	92 μs
Avg. I/O Completion Intr.	380 μs
Avg. Context Switch	62 μs
Disks	1
Data Surfaces	11
Spindle RPM	3600
Avg. Seek	12.0 ms
Read Buffer Size	64 KB
Write Buffer Size	64 KB
Zones	8
Blocks Per Track	46-71

Table 2: System Parameters.

very close to those traced (less than 0.05 percent deviation). The I/O subsystem simulator was validated by comparing the time required to complete requests in the simulator with the actual response times traced. Because we simulated only the disk (rather than the disk, the busses, the controller, etc...), the variation was more substantial. After allowing for average interconnection overheads, we found that our I/O subsystem simulation was within 10% of the actual system for the I/O streams validated.

6 Disk Scheduling Experiment

6.1 Priority Scheduling

Table 2 lists important parameters used for the experiments presented in this section. They match the observed values of the system from which the traces were gathered.

There have been many studies of alternative disk scheduling algorithms [Teor72] [Geis87] [Selt90] [Jaco91]. Most of them use request sorting to reduce seek times while incorporating some type of fairness algorithm. Some of the more recent work has added rotational position to the equation, targeting a reduction in total access time [Selt90] [Jaco91]. In all these cases, however, the work is limited by the fact that all requests are treated equally. As described in section 3, we have separated I/O requests into three distinct classes. From a short-term viewpoint, time-critical I/O requests are clearly more important to system performance than time-noncritical requests. On the other hand, if time-noncritical requests are consistently starved for disk time, the system may run out of memory resources, changing the status of these requests to time-critical.

Three disk scheduling algorithms are compared in this section. The first algorithm is the standard elevator sort, also known as the modified SCAN or LOOK algorithm. The second algorithm, Priority #1, maintains two separate request queues. Time-critical and time-limited requests are placed in the high-priority queue and scheduled by shortest seek. Requests to the same cylinder are sorted in ascending order to take advantage of the disk buffer. Combining time-critical and time-limited requests into a single queue does not exploit the difference between the two but result in a major performance benefit by separating them from time-noncritical requests. When a disk completes its current access, the next request from its high-priority queue is sent to the disk. Time-noncritical requests are placed in the second queue and scheduled by the modified SCAN algorithm. If the high-priority queue is empty, the next request from the low-priority queue is scheduled. When the higher-priority requests interrupt the modified SCAN of the low-priority queue, the state of the SCAN is retained. That is, when the disk completes all requests from the high-priority queue, it returns to its previous place in the modified SCAN of the low-priority queue. The third sorting algorithm, Priority #2, differs from Priority #1 in that it restarts the low-priority queue SCAN at the nearest cylinder which has a pending request rather than returning to the previous SCAN point.

Table 3 presents performance data for three different scheduling algorithms. The priority-based algorithms provide 13-14% improvement in system performance by distinguishing between the various classes of I/O requests. We measure system performance as the time required to complete the application processes in the system. This means that the queue may contain an increased number of pending time-noncritical requests at the completion of the application. If the system continues to be extremely busy (i.e., no period of low activity), additional problems may be presented by this large queue. In our trace, the CPU is idle after

	Elevator Sort	Priority #1	Priority #2
Execution Time (ms)	555193	479435	477755
% of Base	100.0	86.35	86.05
False Idle Time (ms)	114101	35848	34357
Context Switches	6523	14026	14122
Critical Requests	4	4	4
Time-Limited Requests	17777	17777	17777
Avg. Time Limit (ms)	34.9	44.6	44.1
% Satisfied In Time	89.14	66.11	66.32
Avg. Response Time (ms)	36.3	43.3	43.7
Max. Response Time (ms)	1035.81	159.37	123.40
Non-Critical Requests	8364	8364	8364
Avg. Response Time (ms)	118	14960	16307
Avg. Access Time (ms)	7.58	14.10	14.11
% of Base	100.00	186.02	186.15
Buffer Hits	13157	8056	8045
Avg. Seek Distance	148.2	409.2	409.7
Avg. Seek Time (ms)	3.44	9.70	9.72

Table 3: Performance Data for the Disk Scheduling Algorithms.

the completion of these two processes. This allows for the backlog of time-noncritical requests to be serviced. Due to the delaying nature of the file system, not all of the write requests which make up the file have even been initiated at this point. In all of our experiments, the last I/O request of the compressed files was completed at approximately the same time.

The improvement in system performance is derived mainly from the 68-70% reduction in false idle time. While the average response time for time-limited requests has increased, its variation has been reduced. In particular, the lengthy response times which are seen during bursts of activity do not affect the time-limited requests under a class-based scheme nearly as much as with the elevator algorithm. This is evident in the order of magnitude reduction in maximum response time for time-limited requests. The average response time for time-limited requests is much lower for the elevator sort, but most of these time-limited requests far undershoot the time limit (due to hitting in the read buffer). Having the request complete just prior to the time limit is no worse in terms of system performance than having it complete in zero time.

While system performance is increased, the performance of the I/O subsystem (as measured by component metrics) is degraded. The average seek times are increased by over 180%. This occurs because the sorting algorithm deviates from seek-optimal ordering to expedite high-priority requests. In addition, the number of hits in the read buffer is reduced by over 35%. The reason for this is less obvious. Rather than letting processes block until most or all of the time-noncritical requests complete, the class-based algorithms force the requests for which processes would block to be handled immediately. Time-noncritical write requests and time-limited read requests are therefore interleaved, which reduces the performance of the disk read buffer. The increased seek times and reduced buffer hit ratio account for only an 86% increase in access times. The remainder of the increase in average response time is due to queue times. Part of this increase is a secondary effect of the increased access times. More importantly, however, because time-limited requests are given precedence over time-noncritical requests, the processes continue to execute and create new I/O requests. This increases the rate at which I/O requests are generated, which in turn increases the queue times in the I/O subsystem.

Finally, Priority #1, which returns to the previ-

	Elevator Sort		Priority #2	
	Preempt	Non-preempt	Preempt	Non-preempt
Execution Time (ms)	555,193	553,006	477,755	463,392
% of Base	100.0	99.61	86.05	83.47
False Idle Time (ms)	114,101	111,182	34,357	21,034
Context Switches	6523	6219	14,026	10,745
Critical Requests	4	4	4	4
Time-Limited Requests	17,777	17,777	17,777	17,777
Avg. Time Limit (ms)	34.9	31.0	44.1	29.5
% Satisfied In Time	89.14	84.15	66.32	61.11
Avg. Response Time (ms)	36.3	38.8	43.7	43.5
Max. Response Time (ms)	1035.81	1616.69	123.40	133.02
Non-Critical Requests	8364	8364	8364	8364
Avg. Response Time (ms)	118	1936.8	16307	20782
Avg. Access Time (ms)	7.58	8.30	14.11	14.21
% of Base	100.00	109.50	186.15	187.47
Buffer Hits	13,157	12,506	8045	8913
Avg. Seek Distance	148.2	181.7	409.7	492.2
Avg. Seek Time (ms)	3.44	4.22	9.72	11.22

Table 4: Performance Data Related to CPU Preemption upon I/O Completion.

ous location in the elevator sort after completing high-priority requests, performs slightly worse than Priority #2, which moves to the nearest cylinder with a request and continues to SCAN. This results in a longer seek in the former case, which could potentially delay the servicing of a time-limited request which arrives before the current request has completed.

6.2 Preempting upon I/O Completion

While system performance is improved by reducing the amount of false idle time, we did find that the improvement was not equal to the reduction in false idle time. The major reason for this is a large increase (over 115%) in the number of context switches, most of which add to false computation time. The additional context switches are caused by a reduction in the percentage of time-limited read requests which are satisfied within their time limits. Whereas the elevator algorithm tends to allow the processes to block until a large burst of time-noncritical activity passes, the priority algorithms do not. Although the class-based algorithms give priority to time-limited (and time- critical) requests, they can not predict when these requests will arrive. The result is an interleaving of low-priority requests and high-priority requests, which in turn results in the increase in missed time limits and context switches.

Table 4 provides data related to whether or not processes which are blocked waiting for I/O requests should preempt the currently running process when they are awakened. The class-based schemes tend to cause an increased number of process switches due to missing time limits, since the time-limited and time-critical I/O requests which cause these context switches will tend to complete before the next process reaches a stopping point. Preempting the current process increases false computation time and can increase the interleaving of sequential read requests with other requests. A policy which does not preempt an active process in favor of a newly enabled process improves system performance by about 3% when using the class-based algorithms. The improvement comes from a 23% reduction in the number of context switches and an 11% increase in the number of read buffer hits. These improvements also streamline process execution, resulting in a 35-40% reduction in false idle time. It is interesting to note that with a conventional sorting algorithm (represented by the elevator

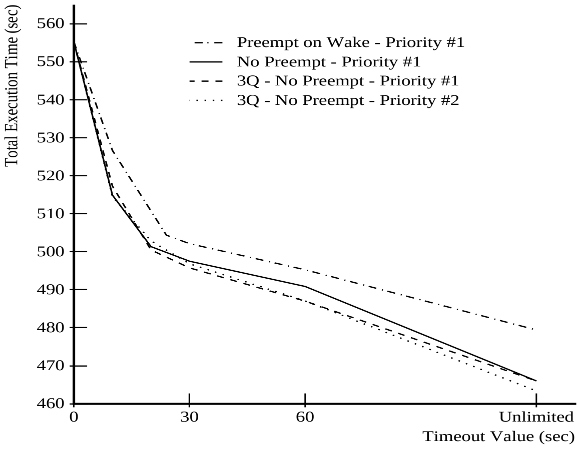

Figure 1: System performance using timeouts

sort), this decision provides a small improvement in performance (about 0.4%). The implication is that it may be necessary to re-examine process scheduling methodology when implementing these new disk scheduling algorithms.

6.3 Timeouts

Scheduling algorithms which expedite the completion of time-critical and time-limited I/O requests get the majority of their performance by postponing the completion of time-noncritical requests. If the system is extremely busy (as is the case in our environment), time-noncritical requests may be starved for extended periods of time. Depending on the guarantees which the system must provide to its users and the problem of resource (e.g., main memory) contention, excessive request completion times may be unacceptable. Figure 1 shows total execution time for two different timeout schemes (with some parameter alterations) given different timeout values. A timeout value of zero corresponds to the shortest seek algorithm employed by the high priority queue (all requests are immediately timed out). An unlimited timeout value (or no timeout value) corresponds to the original priority schemes.

The first scheme (represented by **Preempt on Wake - Priority #1** and **No Preempt - Priority #1**) simply causes a request to be moved from the low-priority queue to the high-priority queue if it waits in the queue for longer than the timeout value. The second scheme (identified by **3Q**) maintains a third queue for partially timed-out requests. When a low-priority request has waited for longer than half of the timeout value, it is moved to the timed-out queue. Requests are chosen from the timed-out queue before the low-priority queue. Requests are still chosen from the high-priority queue first. This scheme improves performance by allowing the disk to selectively handle low-priority requests which are in danger of becoming high-priority requests before the other low-priority requests. The improvement to system performance is minimal (less than 1% difference between **No Preempt - Priority #1** and **3Q - No Preempt - Priority #1**).

Figure 1 also indicates that preemption on wakeup still degrades performance when timeouts are in effect. On the other hand, the Priority #2 scheme performs slightly worse than the Priority #1 scheme for some timeout values. We attribute this to the fact that Priority #1 tends to handle older requests by continuing its elevator sort from the location at which the SCAN was interrupted.

Timeouts significantly reduce the performance of the class-based priority sorting algorithms when the I/O subsystem becomes saturated (as is the case here). These algorithms still provide significant improvements to system performance, but the improvements are reduced because of the scheduling restrictions imposed by timeouts. As discussed in Section 3, the timed-out time-noncritical requests are not time-limited in our taxonomy. The timeouts are simply a design restriction for the I/O subsystem. In many environments, unlike our experimental environment, system activity is quite bursty in nature [McNu86] [Zhou88] [Baker91] [Ruem93]. We believe that periods of low activity will allow time-noncritical requests to be serviced in the background (i.e., without interfering with the completion of time-critical and time-limited requests).

7 Conclusions

We have introduced a taxonomy of I/O requests, dividing them into three sets — time-critical, time-limited, and time-noncritical. This classification is based upon the fashion in which a process interacts with a particular I/O request. That is, the classes are defined by whether or not the creating process waits for the request to complete and, if so, at what point this waiting begins. We believe that recognizing the classes to which I/O requests belong and taking advantage of that information can lead to significant improvements in system performance. The initial study of disk scheduling algorithms presented in this paper supports this premise.

We also promote the use of a system-level model for studying the effect of I/O subsystem performance on system performance. We have described the initial implementation of a system-level simulator, the process-flow model, and presented data from several experiments. Using this model it was found that changing the disk scheduling algorithm to incorporate I/O request classification and ordering the requests on the basis of this information can significantly improve system performance (13-17% in our experiments). The majority of this improvement comes from a 68-82% decrease in false idle time.

Giving priority to time-critical and time-limited requests over time-noncritical requests can lead to starvation of time-noncritical requests if the system is saturated (as it was in our environment). The use of timeouts can limit this starvation, but only at the cost of reducing the performance gains. Fortunately, in many environments the system activity is very bursty [McNu86] [Zhou88] [Baker91] [Ruem93]. Periods of relative inactivity should allow for completion of time-noncritical requests.

Our experimentation also suggests that changing disk scheduling algorithms to incorporate class information may require re-examining current process scheduling algorithms. Preempting the currently executing process in favor of a process newly enabled by the completion of an I/O request was found to reduce system performance by as much as 3% when employing class-based disk scheduling algorithms.

It would be difficult to reach such conclusions if we focused only on the component metrics of the I/O subsystem, because the changes we have made to the scheduling algorithm actually reduce performance as seen by the I/O subsystem. In our experiments, the average service time increased by 86% and average response time increased by two orders of magnitude. Further, traditional I/O subsystem simulation does not provide any information about how system performance is affected by changes in I/O subsystem performance. We believe that using a system-level model is superior to traditional I/O subsystem simulation for these reasons.

Our study of disk scheduling algorithms is far from complete. First, only a single workload was studied. More workloads must be examined before drawing final conclusions about scheduling algorithms. Second, more disk scheduling algorithms must be considered. It has been noted that sorting based on both seek time and rotation time is often superior to sorting on seek time alone [Selt90] [Jaco91]. Also, we have not yet exploited the difference between time-critical requests and time-limited requests. Third, many state-of-the-art disk drives are equipped with request queues. Queueing requests at the disk (or at the disk controller) can significantly reduce the turn-around time between requests by not requiring system involvement between every pair of I/O accesses.

In order to generalize our results and address new items of interest, we intend to instrument a number of systems under a variety of workloads both to capture traces and to collect simple run-time statistics. Also, we will implement class-based disk scheduling algorithms in several systems to validate the results in this paper.

Finally, we expect to refine the process-flow model to increase our capability to study I/O performance. Both the I/O subsystem simulation and the process-flow simulation will be expanded. We plan to study shared-memory multiprocessors as well as uniprocessors. The system side will be augmented to include simulation of system calls related to the storage management software (e.g., file systems, virtual memory, and system calls which directly request I/O). Controllers, buses and other hardware options will be included in the I/O subsystem side. We believe that the process-flow model shows great promise as a tool for studying I/O subsystem performance in terms of system performance.

8 Acknowledgements

We wish to acknowledge both the financial and technical support of NCR Corporation. NCR has been extremely generous with equipment gifts that have been invaluable to our experimental work. In addition, discussions with Jim Browning, Rusty Ransford and Charles Gimarc, all of NCR, have been very useful in understanding the NCR system used in the experiments reported in this paper. We also thank Richie Lary of Digital Equipment Corporation, David Jaffe of Micro Technology Incorporated, and Joe Pasquale of the University of San Diego for their insights on various I/O issues. As always, we appreciate the other members of our research group at the University of Michigan for providing the stimulating environment that characterizes our workday.

Finally, our research group is very fortunate to have the financial and technical support of several additional industrial partners. We are pleased to acknowledge them. They include Intel, Motorola, Scientific and Engineering Software, HaL, Digital Equipment Corporation, Micro Technology Incorporated and Hewlett-Packard.

References

[Baker91] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, J. Ousterhout, "Measurements of a Distributed File System", *SOSP Proceedings*, 1991, pp. 198-212.

[Cars92] S. Carson, S. Setia, "Analysis of the Periodic Update Write Policy for Disk Cache", *IEEE Transactions on Software Engineering*, January, 1992.

[Cars92a] S. Carson, S. Setia, "Optimal Write Batch Size in Log-Structured File Systems", *Proceedings of USENIX File Systems Workshop*, 1992, pp. 79-91.

[Chiu78] W. Chiu, W. Chow, "A Performance Model of MVS", *IBM System Journal*, Vol. 17, No. 4, 1978, pp. 444-463.

[DDI90] *Device Driver Interface/Driver-Kernel Interface (DDI/DKI) Reference Manual*, UNIX System V/386 Release 4, AT&T, 1990.

[Geis87] R. Geist, S. Daniel, "A Continuum of Disk Scheduling Algorithms", *ACM Transactions on Computer Systems*, February 1987, pp. 77-92.

[Geis87a] R. Geist, R. Reynolds, E. Pittard, "Disk Scheduling in System V", *Performance Evaluation Review*, May 1987, pp. 59-68.

[Haig90] P. Haigh, "An Event Tracing Method for UNIX Performance Measurement", *CMG Proceedings*, 1990, pp. 603-609.

[Jaco91] D. Jacobson, J. Wilkes, "Disk Scheduling Algorithms Based on Rotational Position", Hewlett-Packard Technical Report, HPL-CSP-91-7, Feb. 26, 1991.

[Kim86] M. Kim, "Synchronized Disk Interleaving", *IEEE Transactions on Computers*, November 1986, pp. 978-988.

[McKu84] M. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, August 1984, pp. 181-197.

[McNu86] B. McNutt, "An Empirical Study of Variations in DASD Volume Activity", *CMG Proceedings*, 1986, pp. 274-283.

[McVo91] L. McVoy, S. Kleiman, "Extent-like Performance from a UNIX File System", *Winter USENIX Proceedings*, 1991, pp. 1-11.

[Mill91] E. Miller, R. Katz, "Input/Output Behavior of Supercomputing Applications", *Proceedings of Supercomputing*, 1991, pp. 567-576.

[Oust90] J. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast as Hardware?", *Summer USENIX Proceedings*, 1990, pp.247-256.

[Rich92] K. Richardson, M. Flynn, "TIME: Tools for Input/Output and Memory Evaluation", *Proceedings of the Hawaii International Conference on Systems Sciences*, 1992, pp. 58-66.

[Ruem93] C. Ruemmler, J. Wilkes, "UNIX Disk Access Patterns", *Winter USENIX Proceedings*, 1993.

[Seam69] P. Seaman, R. Soucy, "Simulating Operating Systems", *IBM System Journal*, No. 4, 1969, pp. 264-279.

[Selt90] M. Seltzer, P. Chen, J. Ousterhout, "Disk Scheduling Revisited", *Winter USENIX Proceedings*, 1990, pp. 313-324.

[Teor72] T. Teorey, T. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies", *Communications of the ACM*, March 1972, pp. 177-184.

[Zhou88] S. Zhou, "A Trace-Driven Simulation Study of Dynamic Load Balancing", *IEEE Transactions on Software Engineering*, Volume 14, No. 9, September 1988, pp. 1327-1341.