Disk Scheduling with Quality of Service Guarantees

John Bruno, José Brustoloni, Eran Gabber, Banu Özden and Abraham Silberschatz
Bell Laboratories, Lucent Technologies
600 Mountain Ave.
Murray Hill, NJ 07974

{jbruno, jcb, eran, ozden, avi}@research.bell-labs.com

Abstract

This paper introduces YFQ, a new disk scheduling algorithm that allows applications to set aside for exclusive use portions of the disk bandwidth. We implemented YFQ as part of the Eclipse/BSD operating system, which is derived from FreeBSD, a version of 4.4 BSD Unix. YFQ's disk bandwidth reservations can guarantee file accesses with high throughput, low delay, and good fairness. Such quality of service (QoS) guarantees to individual applications unfortunately can also hinder global disk scheduling optimizations. We propose and evaluate several disk scheduling enhancements that promote global optimizations and give to YFQ aggregate disk thoughput approaching that of FreeBSD's conventional disk scheduler, which does not provide QoS guarantees. We believe that our enhancements may be helpful also in other disk scheduling algorithms.

1 Introduction

In order to perform adequately regardless of system load, multimedia applications require quality of service (QoS) guarantees, such as minimum disk and network bandwidth. Unfortunately, mainstream operating systems are still geared toward time-sharing workloads, which do do not require such guarantees. The Eclipse project [4, 6] is examining how to modify the schedulers of mainstream systems so as to provide quality of service guarantees. Eclipse's emphasis is on algorithms with high *cumulative service* guarantees, as we explain in the rest of this paragraph. Many scheduling algorithms that provide QoS guarantees have been proposed [7, 13, 8, 15, 9, 10, 1, 2, 14], but they often consider each resource in isolation, neglecting interactions. Unfortunately, this is not realistic due to the "closed-loop" nature of most applications: After completion of a request on one resource (e.g., disk), an application often competes for another resource (e.g., CPU) before being able to make another request on the first resource. In a recent paper [3], we have shown that, in such situations, cumulative service is more meaningful than is the theoretical isolated throughput, because scheduling delays on the different resources can accumulate and make actual throughput much lower than expected.

This paper introduces YFQ, a new disk scheduling algorithm that is easy to implement and allows applications to reserve portions of the disk bandwidth. We describe YFQ in Section 2. YFQ's reservations can guarantee file accesses with high cumulative service, low delay, and good fairness. Such QoS guarantees to individual applications unfortunately can also hinder global disk scheduling optimizations. In Section 3, we propose and experimentally evaluate several disk scheduling enhancements that promote global optimizations and give to YFQ aggregate disk throughput approaching that of a conventional disk scheduler, which does not provide QoS guarantees. We believe that our enhancements may be helpful also in other disk scheduling algorithms. Section 4 discusses related work, and section 5 concludes.

2 The YFQ scheduling algorithm

This section describes the YFQ scheduling algorithm.

YFQ assumes that portions of the scheduled resource (e.g., disk) have been set aside as *resource reservations* (or *reservations*, for short). The portion corresponding to a reservation i is proportional to its *weight*, w_i. A reservation may be used exclusively by one or more processes.

Each I/O request that arrives at the scheduler specifies the respective resource reservation. The scheduler enqueues each request r_i that uses reservation i in the corresponding queue q_i. A reservation i is *busy* if it has I/O requests being serviced or waiting in q_i; otherwise, i is *idle*. A resource is *busy* if it has at least one busy reservation; otherwise, the resource is *idle*.

YFQ associates a *start tag*, S_i, and a *finish tag*, F_i, with each reservation i. S_i and F_i are initially zero. A *virtual*

0-7695-0253-9/99 $10.00 © 1999 IEEE

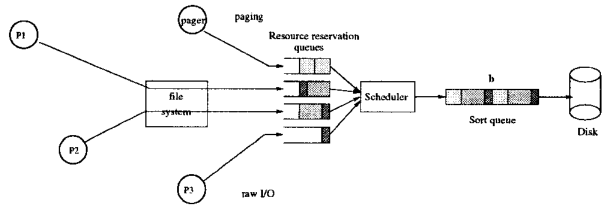

Figure 1: The sort queue allows the disk driver or disk to reorder requests and minimize disk latency and seek overheads

work function, $v(t)$, is defined such that: (1) $v(0) = 0$; (2) While the resource is busy, $v(t)$ is the minimum of the start tags of its busy reservations at time t; (3) When the resource becomes idle, $v(t)$ is set to the maximum of all finish tags.

When a new request r_i that uses reservation i arrives: (1) If q_i was previously empty, YFQ makes $S_i = \max(v(t), F_i)$ followed by $F_i = S_i + \frac{l_i}{w_i}$, where l_i is the data length of request r_i; and (2) YFQ appends r_i to q_i.

YFQ selects for servicing the request r_i at the head of the queue q_i of the busy reservation i with the smallest finish tag F_i. r_i remains at the head of q_i while r_i is being serviced. When r_i completes, YFQ dequeues it; if queue q_i is still non-empty, YFQ makes $S_i = F_i$ followed by $F_i = S_i + \frac{l'_i}{w_i}$, where l'_i is the data length of the request r'_i now at the head of q_i.

Selecting one request at a time, as described above, allows YFQ to provide good cumulative service, delay, and fairness guarantees, as shown in an extended version [5] of this paper. However, such guarantees may come at the cost of excessive disk latency and seek overheads, harming aggregate disk throughput. Therefore, YFQ can be configured to select up to b requests (a *batch*) at a time and place them in a *sort queue*, as shown in Figure 1. The disk driver or the disk itself may reorder requests within a batch so as to minimize disk latency and seek overheads.

3 Tradeoffs between QoS and aggregate throughput

As described in the previous section, setting the batch size $b = 1$ prevents the disk driver or disk from reordering requests, and the resulting QoS is that determined by YFQ. On the other hand, setting b to a large number allows the disk driver or disk to reorder requests extensively, optimizing aggregate throughput, but also undoing YFQ's QoS ordering. This section examines experimentally trade-offs between QoS and throughput and enhancements that increase throughput while preserving QoS.

Maximum Delay [ms]

Task 1		Task 2		Task 3	
YFQ	BSD	YFQ	BSD	YFQ	BSD
12	12				
50	54	50	50		
66	1445	66	75	66	75

Table 1: Pipelining can cause excessive delays under FreeBSD

Throughput [KB/s]

Task 1		Task 2		Task 3	
YFQ	BSD	YFQ	BSD	YFQ	BSD
7815	7815				
1302	1302	1302	1302		
977	45	977	1944	977	1905

Table 2: Pipelining can cause unfairness under FreeBSD

3.1 Experimental set-up

We implemented YFQ and its enhancements as parts of the Eclipse/BSD operating system [4, 6]. Eclipse/BSD is based on FreeBSD 2.2.7, a derivative of 4.4 BSD Unix [11]. Unlike FreeBSD, Eclipse/BSD's CPU scheduler uses the MTR-LS algorithm [3]. This section's measurements were taken on a PC with 266 MHz Pentium II processor, 64 MB RAM, and a 9 GB Seagate ST39173W fast wide SCSI disk. The PC used Eclipse/BSD or FreeBSD. Figures 2 and 3 show that the disk used has a bandwidth of roughly 155 Mbps for sequential raw disk I/O, 36 Mbps for random raw I/O in 64 KB chunks, and 6.5 Mbps for random raw I/O in 8 KB chunks.

3.2 Pipelining, batching, and overlapping

FreeBSD's disk driver may sort outstanding requests in C-SCAN order. If enabled (not the default and not used in this subsection), this feature may allow certain applications to nearly monopolize the disk for extended periods, to the detriment of QoS guarantees for other applications. For example, an application's read request r_0 for a sector on the first track may be placed behind another application's requests for sectors of monotonically increasing address – and in the worst case, the whole disk may be accessed before r_0 is serviced.

FreeBSD's disk driver also *pipelines* up to p requests to the disk, and the disk may reorder such requests so as to reduce its latency and seek overheads (by default, $p = 4$). Reordering may cause widely varying and unacceptable de-

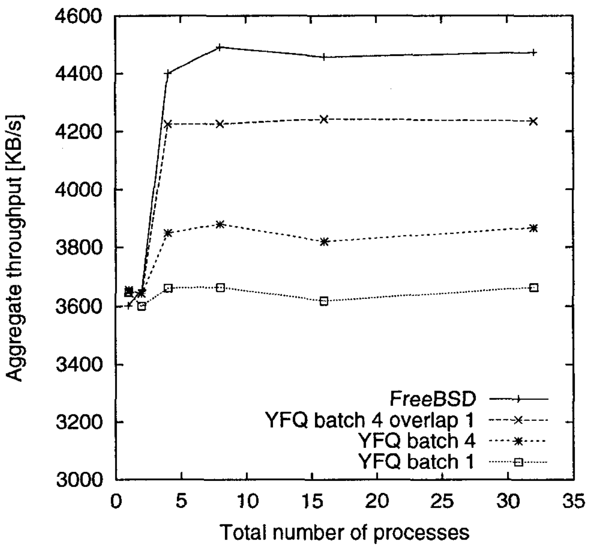

Figure 2: Aggregate throughput with 64 KB I/O requests

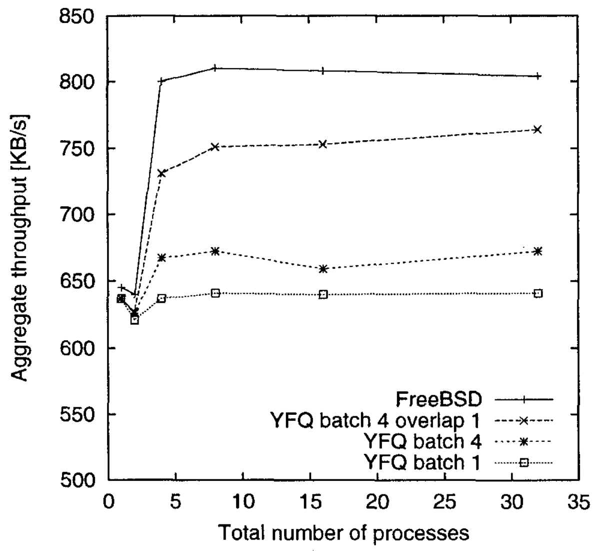

Figure 3: Aggregate throughput with 8 KB I/O requests

lays. For example, Tables 1 and 2 show the maximum delay and throughput, respectively, when one or more tasks read 64 KB chunks sequentially, with disk cache disabled. Task 1 starts at sector 0, while zero or more tasks $j, j > 1$, start at sector 8 G. Under FreeBSD, if there are three tasks, the maximum delay for Task 1 is 1445 ms, while that for Tasks 2 and 3 is 75 ms; the respective throughputs are 45 and greater than 1900 KB/s.

Reordering of requests by the disk driver or disk can cause delays that antagonize not only YFQ but also any other disk scheduling algorithm with QoS guarantees. A common solution to this problem is *batching*. Batching allows multiple outstanding requests and some global disk optimizations. However, because requests in a batch must complete before the next batch, deterioration of QoS is limited by a function of the batch size. In particular, if $b = 1$, neither disk driver not disk can reorder requests, and there is no QoS deterioration. Tables 1 and 2 show that, unlike FreeBSD, YFQ with $b = 1$ gives to three tasks equal maximum delay of 66 ms and throughput of 977 KB/s. But fairness comes at the expense of aggregate throughput: in this case, 2931 KB/s under YFQ vs. 3894 KB/s under FreeBSD.

Figures 2 and 3 show that increasing batch size from 1 to 4 considerably improves YFQ aggregate throughput. With a batch size of 16, YFQ's aggregate throughput further approaches that of FreeBSD. However, a large batch size entails looser QoS bounds.

A refinement to batching, *overlapping*, can improve aggregate throughput without increasing the batch size. In overlapping, the disk scheduler dispatches the next batch when all but one of the present batch's requests complete. If $p = b+1$, the disk can pick first the request in the new batch that is closest to the last request in the present batch, minimizing overheads. Without overlapping, the disk picks first the first request in the new batch, even if that request is far from the last request. Figures 2 and 3 confirm that overlapping considerably improves YFQ's aggregate throughput.

3.3 Breaking ties

A disk scheduler with QoS guarantees may find two or more requests equally preferable for selection. For example, in YFQ, this happens when two or more requests have the same finish tag. In such cases, it makes sense to select the request that minimizes disk latency and seek overheads. Figures 4 and 5 show that this tie breaking rule can dramatically increase YFQ's aggregate throughput. Indeed, with enough concurrency, YFQ's aggregate throughput can become better than that of default FreeBSD (without C-SCAN).

3.4 Fragmentation

I/O delay bounds usually increase with the largest data length allowed in a request. Therefore, systems typically break down "large" I/O requests into multiple "fragment" I/O requests of data length below a certain limit. For example, in FreeBSD disk I/O, the limit is 64 KB. A large I/O

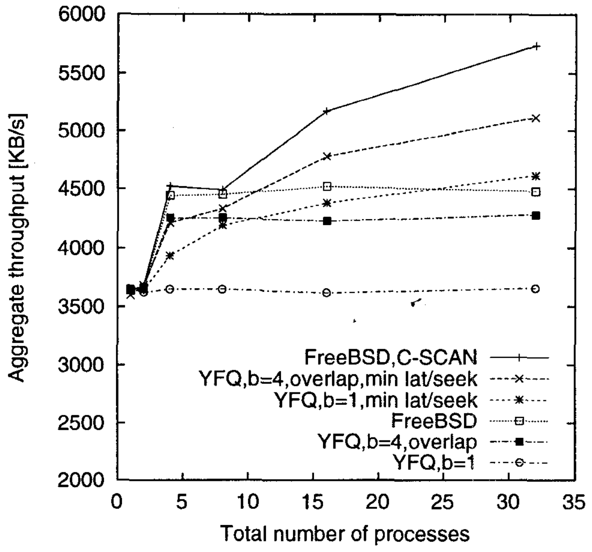

Figure 4: Breaking ties by minimizing latency and seek overheads greatly improves aggregate throughput for 64 KB reads

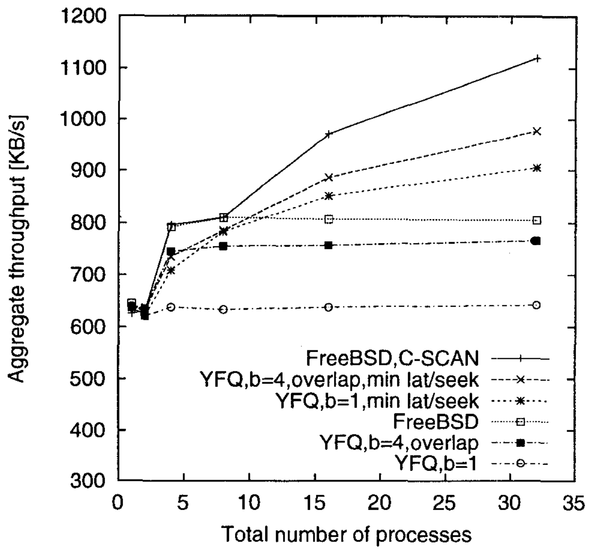

Figure 5: Breaking ties by minimizing latency and seek overheads greatly improves aggregate throughput for 8 KB reads

request completes when all its fragment I/O requests complete.

Because the order of completion of fragments is unimportant, it makes sense to have the disk scheduler consider all fragments equally preferable for selection. In YFQ, this is achieved by making the finish tags of all fragments equal to what would be the finish tag of the original, large I/O request. Such finish tag setting has the benefit of giving other, smaller requests preference over the large I/O request, reducing delays.

If all requests are large, however, it is preferable to maintain them as intact as possible, keeping accesses sequential. Figures 6 and 7 demonstrate this by showing that when fragments of 256 KB requests are interleaved (unmodified FreeBSD and YFQ), the aggregate throughput falls dramatically. We therefore modified the operating system to issue fragments in *groups* of n requests. To prevent interleaving, FreeBSD requires n as large as possible, whereas YFQ requires only $n \geq b+1$. Figure 6 shows that issuing requests as a group greatly improves the aggregate throughput of FreeBSD, whereas Figure 7 shows that this rule, combined with (1) the previous section's tie-breaking rule and (2) making all fragments equally preferable, greatly improves YFQ's aggregate throughput.

However, grouping the fragments of a large request can penalize other, smaller requests. The penalty is less in YFQ because YFQ can favor smaller requests and fewer fragments need to be issued in a group. This is illustrated in Figure 8, where the throughput of a task making 8 KB read requests is measured against the number of other tasks making 256 KB read requests.

3.5 Accounting for disk latency

Servicing of a disk I/O request incurs disk latency and seek overheads and data transfer time. However, YFQ's start and finish tags, as described in Section 2, (roughly) represent only data transfer times. Because latency and seek overheads are proportionally more significant for short data than for long data requests, YFQ may not correctly distribute bandwidth if requests have widely varying sizes.

We therefore modified YFQ so that, for an I/O request of data length l_i, modified YFQ uses $l_i + l_{lat}$ instead of l_i in its calculations. l_{lat} is a fixed value that represents average latency and seek overheads. Figures 9 and 10 show how maximum delay and throughput varied with l_{lat} in an experiment with a task making 64 KB requests and seven tasks making 8 KB requests. Increasing l_{lat} improves the performance of the task making 64 KB requests without appreciably affecting the performance of the tasks making 8 KB requests. The points labeled "64 KB expected" correspond to the performance when all eight tasks make 64 KB requests. The figures suggest that $l_{lat} = 125$ KB would be a reasonable setting for this system.

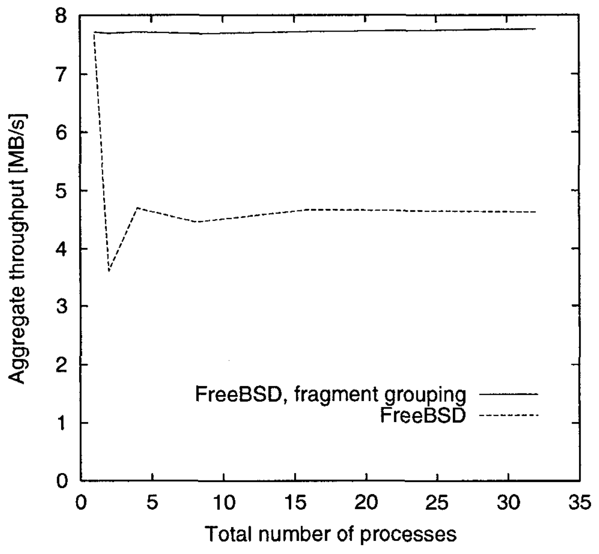

Figure 6: FreeBSD can prevent fragment interleaving by grouping fragments

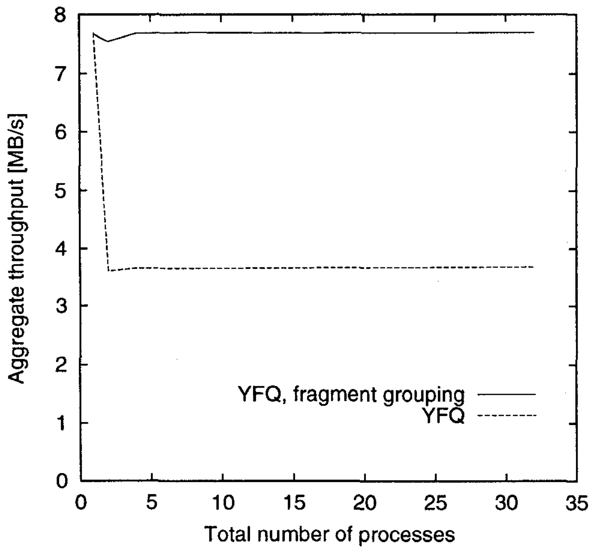

Figure 7: YFQ can also prevent fragment interleaving by grouping fragments

4 Related work

YFQ is an easily implementable approximation to the *generalized processor sharing* (GPS) model of proportional resource sharing [12]. GPS can be briefly defined as follows: Let $s_i(\tau, t)$ denote the service received by reservation i in time interval $[\tau, t]$; then GPS specifies that $s_i(\tau, t)/w_i = s_j(\tau, t)/w_j$ holds for any time interval $[\tau, t]$ during which any pair of reservations i and j are both continuously busy. Numerous algorithms simulate GPS, with different fairness, delay, and cumulative service guarantees [8, 15, 9, 10, 1, 2]. Many GPS approximations rely on elaborate virtual time calculations, which are expensive to implement [8, 1, 2]. On the other hand, GPS approximations that do employ simple virtual time calculations [15, 9, 10] often do not provide desired QoS guarantees (such as cumulative service). YFQ is unique in that it is simple to implement but offers provably high cumulative service guarantees, low delay, and good fairness.

5 Conclusions

We described a new disk scheduling algorithm, YFQ, that allows applications to make and use disk bandwidth reservations, thereby enjoying cumulative service, delay, and fairness guarantees. Such guarantees may cause extra disk latency and seek overheads. We demonstrated experimentally several new scheduling enhancements for reducing such overheads: batching, overlapping, tie-breaking by minimizing disk latency and seek overheads, fragmenting, and fragment grouping. These enhancements may be useful also in other disk schedulers.

References

[1] J. Bennet and H. Zhang. "WF^{2}Q: Worst-Case Fair Weighted Fair Queueing", in *Proc. INFOCOM'96*, IEEE, Mar. 1996, pp. 120-128.

[2] J. Bennet and H. Zhang. "Hierarchical Packet Fair Queueing Algorithms", in *Proc. SIGCOMM'96*, ACM, Aug. 1996.

[3] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. "Move-to-Rear List Scheduling: a New Scheduling Algorithm for Providing QoS Guarantees", in *Proc. Multimedia'97*, ACM, Nov. 1997.

[4] J. Bruno, E. Gabber, B. Özden and A. Silberschatz. "The Eclipse Operating System: Providing Quality of Service via Reservation Domains", in *Proc. Annual Tech. Conf.*, USENIX, June 1998, pp. 235-246.

[5] J. Bruno, José Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. "Disk Scheduling with Quality of Service Guarantees", Technical report, Bell Laboratories, Lucent Technologies, Mar. 1999.

[6] J. Bruno, José Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. "Retrofitting Quality of Service into a Time-Sharing Operating System", to appear in *Proc. Annual Tech. Conf.*, USENIX, June 1999.

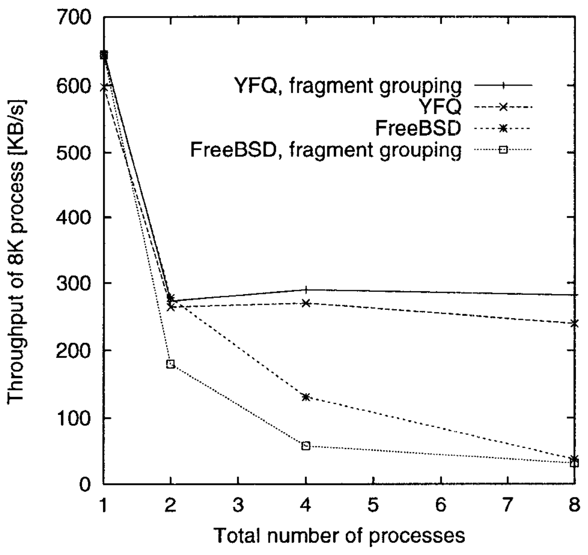

Figure 8: Smaller requests are penalized less on YFQ than on FreeBSD

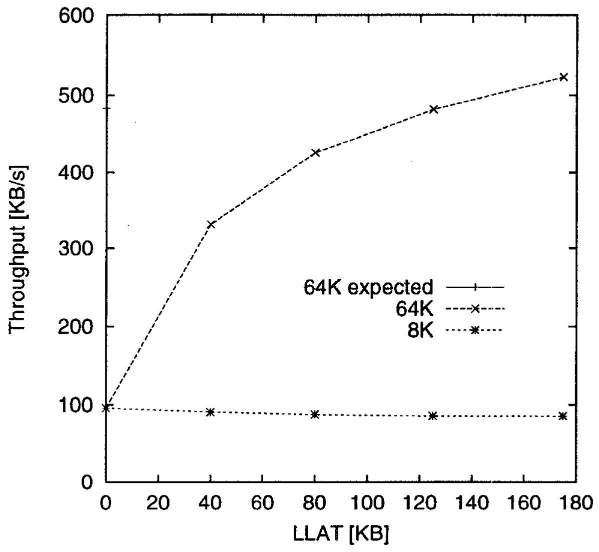

Figure 9: l_{lat} increases throughput when there are requests of different sizes

[7] S. Chen, J. Stankovic, J. Kurose, and D. Towsley. "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems", in *J. Real-Time Systems*, 3:307-336, 1991.

[8] A. Demers, S. Keshav and S. Shenker. "Design and Analysis of a Fair Queueing Algorithm", in *Proc. SIGCOMM'89*, ACM, Sept. 1989, pp. 1-12.

[9] S. Golestani. "A Self-Clocked Fair Queueing Scheme for Broadband Applications", in *Proc. INFOCOM'94*, IEEE, June 1994, pp. 636-646.

[10] P. Goyal, H. Vin and H. Chen. "Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks", in *Proc. SIGCOMM'96*, ACM, Aug. 1996.

[11] M. McKusick, K. Bostic, M. Karels and J. Quarterman. "The Design and Implementation of the 4.4 BSD Operating System", Addison-Wesley Pub. Co., Reading, MA, 1996.

[12] A. Parekh and R. Gallager. "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks–the Single Node Case", in *Transactions on Networking*, ACM/IEEE, June 1993, pp. 344-357.

[13] A. Reddy and J. Wyllie. "I/O Issues in a Multimedia System", in *Computer*, IEEE, Mar. 1994, pp. 69-74.

[14] P. Shenoy and H. Vin. "Cello: A Disk Scheduling Framework for Next Generation Operating Systems", in *Proc. SIGMETRICS'98*, ACM, June 1998.

[15] L. Zhang. "Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks", in *Proc. SIGCOMM'90*, ACM, Sept. 1990.

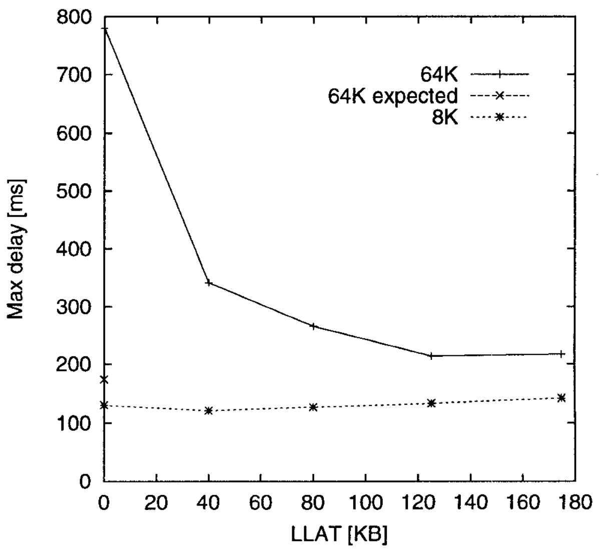

Figure 10: l_{lat} reduces delays when there are requests of different sizes