Performance Comparison of Thrashing Control Policies for Concurrent Mergesorts with Parallel Prefetching

Kun-Lung Wu and Philip S. Yu

IBM T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598

James Z. Teng

IBM Programming Systems
P. O. Box 49032
San Jose, CA 95161

Abstract

We study the performance of various run-time thrashing control policies for the merge phase of concurrent mergesorts using parallel prefetching, where initial sorted runs are stored on multiple disks and the final sorted run is written back to another dedicated disk. Parallel prefetching via multiple disks can be attractive in reducing the response times for concurrent mergesorts. However, severe *thrashing* may develop due to imbalances between input and output rates, thus a large number of prefetched pages in the buffer can be replaced before referenced. We evaluate through detailed simulations three run-time thrashing control policies: (a) disabling prefetching, (b) forcing synchronous writes and (c) lowering the prefetch quantity in addition to forcing synchronous writes. The results show that (1) thrashing resulted from parallel prefetching can severely degrade the system response time; (2) though effective in reducing the degree of thrashing, disabling prefetching may worsen the response time since more synchronous reads are needed; (3) forcing synchronous writes can both reduce thrashing and improve the response time; (4) lowering the prefetch quantity in addition to forcing synchronous writes is most effective in reducing thrashing and improving the response time.

1 Introduction

Parallel prefetching via multiple disks can be attractive in an I/O bound environment. This is especially the case for sequential data references, as it not only overlaps the CPU and I/O operations but also increases the throughput of read I/Os through I/O parallelism. External sorts, or mergesorts, are one of the most time-consuming database operations in query processing and most of their operations are sequential and I/O intensive. In particular, consider the merge phase of an external sort. Assume that the sort phase uses some kind of tournament tree [9] to sort the tuples into multiple *runs*, where the tuples in each run are sorted according to the sort key. During the merge phase, with sufficient buffer these sorted runs are read into the memory from disks, merged and then sequentially written back to another disk in the final sorted order. Thus, parallel prefetching can be an effective approach to improving the response time for the merge phase of mergesorts.

Note in this paper, the input I/O parallelism comes from the assumption that the multiple sorted runs from the sort phase are spread over multiple disks. However, each run, including the final sorted one, resides only on one disk. We do not assume that each run can interleave among multiple disks as in the disk array environment [14].

In addition to parallel prefetching on reads, the I/O efficiency can also be improved by deferring the disk writes and batching them to the same disk. These are referred to as *deferred writes*, in contrast to *synchronous writes* where a write request must be completed before further processing on the merge task can be resumed. With parallel prefetching and the final sorted run residing on a single disk, the imbalance between the number of input and output streams can cause a large number of dirty pages to accumulate in the buffer and thus result in severe *thrashing*. Namely, prefetched pages are stolen, i.e. replaced, by other pages before they are referenced. When thrashing occurs, not only the previous work of prefetching the pages is wasted, but also a *synchronous read* (i.e. the merge process is put into a wait state until the read is completed) is needed for each stolen page. Hence, the advantages of overlapping the CPU and I/O as well as I/O parallelism can be greatly compromised.

The number of pages prefetched from each disk is referred to as *prefetch quantity* (PFQ). Since disk seek time and latency are typically the dominant factors in disk I/O time, a larger PFQ results in a smaller average I/O cost per page. However, with parallel prefetching, a large prefetch quantity for multiple concurrent merges may aggravate the effect of thrashing by widening the I/O imbalance.

Besides PFQ, the degree of thrashing can also be affected by the number of runs allocated to a merge task. Given a PFQ, the smaller the number of runs allocated, the lower the degree of potential thrashing can be. However, if an insufficient number of runs is allocated and thus multiple passes over some or all the data are needed to complete a merge task, the response time may be significantly worsened. This type of phenomenon is common in the query processing of relational databases, such as hash joins [5] and mergesorts. This adds an extra dimension to the complexity of the concurrent merge problem. As an example, consider the merge phase of a mergesort. A merge task can take multiple steps to complete depending upon the number of runs allocated, which in turn depends on the buffer availability. For instance, to merge 16 sorted runs, one can accomplish the task in one step by merging all 16 runs in one step. Alternatively, due to buffer limitation or other constraints, one can first merge 8 of the initial sorted runs to produce an intermediate output run, and then merge the remaining 8 initial runs with the intermediate output run.

Therefore, to minimize the response time while avoiding thrashing, both PFQ and the number of steps determined by the buffer availability have to be considered. Although some optimal trade-off may be devised for a single merge, in a concurrent mergesort environment further complexities arise from buffer contentions among multiple merge requests, which can be of very different sizes. Moreover, the PFQ used by the buffer manager is a system-wide quantity (e.g. see [17] on IBM's DB2); it may be adjusted by system load, but cannot be tuned for each merge request separately. Since the buffer requirements and the time from when a page is prefetched until it finally gets processed are different for merge requests of different sizes, a run-time control mechanism is needed to monitor and handle thrashing dynamically.

Thus, in this paper we study various run-time thrashing control policies for concurrent merges using parallel prefetching, where the initial sorted runs are stored on multiple input disks and the final output is written back to another dedicated disk. We examine run-time thrashing control policies under two different scheduling algorithms, one using a *fixed* PFQ and the other using *flexible* PFQs. The flexible PFQ algorithm allows PFQ to be readjusted accordingly whenever a new job is scheduled for execution; however, the fixed PFQ algorithm does not. Three run-time control policies are considered: (a) disabling prefetching, (b) forcing synchronous writes, i.e. disabling deferred writes, and (c) adjusting/lowering the prefetch quantity in addition to forcing synchronous writes. Thrashing control is triggered when the total number of nonreplaceable buffer pages reaches a pre-defined threshold. These different approaches to thrashing control are evaluated through detailed simulations. The results show that disabling prefetching can control thrashing but may worsen the response time, and lowering the prefetch quantity in addition to forcing synchronous writes is most effective in both thrashing control and response time improvement.

A large body of literature exits concerning specifically with the I/O performance of the merge phase of an external sort [9, 1, 4, 8, 10, 13, 16]. However, these papers have only dealt with a single sort or merge job, and have not considered the interactions of concurrent merges. Moreover, they have not studied the problem of thrashing due to parallel prefetching. There also exists other work related to allocating buffer for general relational database queries [12, 6, 2, 18, 15]. However, their emphases were different from ours and again they did not study the thrashing problem.

The paper is organized as follows. Section 2 describes the system model. Section 3 presents the scheduling algorithms considered in this paper. We then examine the various thrashing control policies in Section 4. The simulation model is described in Section 5. In section 6, we illustrate the usefulness of prefetching and the potential thrashing problem. Performance comparisons of these policies are presented in Section 7.

2 The system model

The system consists of a CPU, a buffer, multiple input disks storing the initial sorted runs of all merge requests, and a group of dedicated output disks storing the sorted run of a merge step (see Figure 1). Note that the general results of this paper remain the same even when the

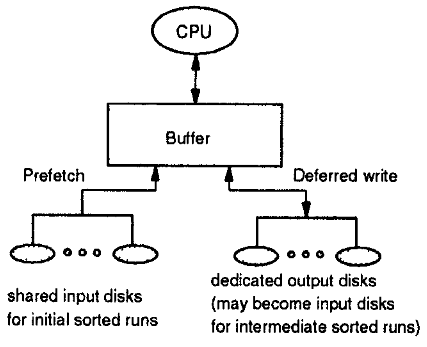
CPU
Buffer
Prefetch
Deferred write
shared input disks for initial sorted runs
dedicated output disks (may become input disks for intermediate sorted runs)

Figure 1: The system architecture.

system has multiple CPUs, since the merge tasks are I/O bound, not CPU bound. Both the CPU and disks are assumed to use the FCFS discipline. The input disks are evenly shared by all merge requests, but each output disk is dedicated to a merge step at a given time. If more than one merge step is needed, the dedicated output disk during a previous step may become an input disk later on.

To merge N sorted runs, at least the N active pages (the pages currently being merged), one from each run, have to be in the buffer. These active buffer pages are *pinned* and cannot be replaced until *unpinned* after the data are depleted by the merge process. The process of locating the active page of a run in the buffer is referred to as a *GetPage*. When one of the active pages is depleted, a GetPage request is issued for the next page of the same run. If it is already in the buffer due to prefetching, then the buffer page is pinned and the merge process continues. If not, a synchronous read is issued and the merge process enters a wait state until the read is completed.

The buffer manager maintains a deferred write queue for the output of each ongoing merge and a system-wide LRU chain linking all the buffer pages. An asynchronous write request is issued when a certain number of dirty pages have been generated for a merge. The number of dirty pages written out in each write I/O is referred to as *deferred write quantity* (DWQ). Clean buffer pages containing newly fetched data from the input disks and dirty pages produced by the merge process are placed at the top of the LRU chain. However, after an active page is depleted, it is placed at the bottom of the LRU chain since the page is no longer needed. Also, after a disk write is completed, the clean pages are placed at the bottom of the LRU.

In addition to the active pages currently being merged, buffer pages being prefetched or synchronously read are also pinned and cannot be replaced. These pages are unpinned after the read I/Os are completed. When a buffer page is needed (for data fetched from disks or for merged output data), the buffer manager always gets one from the bottom of the LRU. Notice that thrashing occurs when a prefetched page reaches the bottom of the LRU and is replaced with new data before it is referenced by the merge process, resulting in a synchronous read for the replaced page later on.

A prefetch request is issued after a prefetch triggering page becomes active and is located in the buffer. A prefetch triggering page is the page whose page number is a multiple of the current PFQ. For example, if PFQ is 8, then pages 0, 8, 16, ... are prefetch triggering pages. When page 8 becomes active and is located in the buffer, a prefetch request for pages 16-23 is issued. Therefore, if N runs are allocated, then at least $2 \times N \times \text{PFQ}$ buffer pages are required. In our implementation, the initial 8 pages are fetched together, i.e., pages 1-7 are prefetched when page 0 is synchronously read from the disk. In the simulations of this paper, we assumed PFQ can be 0, 1, 2, 4, 8 or 16. A PFQ value of 0 means prefetching is disabled.

Table 1 summarizes the notation and definitions for the workload and system parameters. The number in the parenthesis at the end of a definition is the default value used in the simulation if not otherwise specified. Table 2 summarizes other notation used for scheduling and thrashing control.

3 Scheduling algorithms

For an arriving merge task, a scheduling algorithm (or run allocation algorithm) is first executed to decide the number of runs to be merged for this task. If the number of allocated runs is smaller than requested, then more than one merge step is needed to complete the job. In this study we adopt an optimal merge pattern, similar to the one in [7], in which the k smallest length runs are merged in each step if k runs are allocated. Since it is generally undesirable to complete a merge in multiple steps, we set a maximum number of steps S_{max} that a merge has to be completed and two values for S_{max} were studied: 1 and 2.

Two different scheduling algorithms are considered, one using a fixed PFQ and the other using flexible ones. The scheduler maintains the total number of runs, $R_{allocated}$, allocated so far in the system and decides whether or not a merge can be scheduled based on the following criteria.

Notation	**Definition** (Default values)
λ	merge arrival rate, Poisson interarrival time distribution (0.03 arrivals/sec)
B	buffer size (2,000 pages)
R	mean initial runs per merge, uniform distribution between $R/2$ and $3R/2$ (30 runs)
L	run length (300 pages)
PFQ	prefetch quantity, can be 0, 1, 2, 4, 8 or 16 (16 pages)
DWQ	deferred write quantity (32 pages)
C_{set_up}	CPU cost for initiating a read or write I/O request (5,000 instructions)
C_{merge}	CPU cost for generating a sorted dirty page (10,000 instructions)
C_{mgm}	CPU cost for managing the page table when a GetPage is a buffer hit (100 instructions)
D	number of input disks (10 disks)
MPL	maximum number of active merges in the system (10)
M	CPU MIPS (40)
T_{seek}	average disk seek time (16 ms)
$T_{latency}$	average disk latency time (8.35 ms)
$T_{transfer}$	average disk transfer time for a page (0.91 ms)
S_{max}	maximum number of steps for a merge to finish (1 step)
P_{max}	maximum prefetch quantity (16)

Table 1: Workload and system parameters

Notation	Definition
R_{max}	maximum number of runs the buffer can support under current PFQ
R_i	no. of runs requested by a merge
$R_{allocated}$	the total number of runs currently allocated
$round\_2(y, P_{max})$	round y down to the nearest number that is a power of 2, but less than P_{max}
B_{NR}	total number of buffer pages that are nonreplaceable (pinned or dirty)
P_1, P_2	thresholds used for disabling and enabling prefetching, respectively
W_1	threshold used for forcing synchronous writes
P_{min}	PFQ can be lowered by half if it is greater than P_{min}

Table 2: Other notation used for scheduling and thrashing control.

Let R_{max} represents the maximum number of runs the buffer can support under the current PFQ.

$$R_{max} = \lfloor \frac{B}{(2 \times \text{PFQ})} \rfloor. \tag{1}$$

Note that a factor of 2 is needed in the denominator due to the double buffering requirement: while one set of pages is being consumed, the next set is being prefetched.

For the fixed PFQ algorithm, a merge task requesting R_i runs can be scheduled if

$$R_{max} \geq R_i + R_{allocated} \tag{2}$$

and $S_{max} = 1$ or 2, or if

$$R_{max} \geq \lceil \frac{(R_i + 1)}{2} \rceil + R_{allocated} \tag{3}$$

and $S_{max} = 2$. When $S_{max} = 2$, the fixed PFQ algorithm tries to allocate R_i runs if possible. If not, then it tries to allocate $\lceil (R_i + 1)/2 \rceil$ runs if possible. If not, the job cannot be scheduled and has to wait in a queue. Upon the completion of a merge, the request at the head of the waiting queue is scheduled for execution, if it is schedulable.

In our study, once a merge is allocated N runs for its first step, at most N runs are allocated for its later step, if more than one step is needed. The second step is started once the first step finishes (the intermediate sorted run is completely written to a disk). Thus, the optimal runs needed to finish a merge in 2 steps is $\lceil (R_i + 1)/2 \rceil$. For example, to merge 12 runs in 2 steps optimally, 7 runs have to be merge in the first step and the rest of 5 initial runs plus the intermediate run (total of 6 runs) are merged in the second step. If less than 7 runs, say 6, are merged in the first step, it cannot be finished in 2 steps, since at most 6 runs can be merged in the second step and there are still 7 runs to be merged. On the other hand, if more than 7 runs are merged in the first step, then more work needs to be done since more data are processed twice.

Let P_{max} be the maximum PFQ allowed. We define $round\_2(y, P_{max})$ as a function that rounds y down to the nearest number that is a power of 2, but less than P_{max}, e.g., $round\_2(5, 16) = 4$, $round\_2(15, 16) = 8$, and $round\_2(45, 16) = 16$.

For the flexible PFQ algorithm, the scheduling criteria is as follows. If R_i runs are requested, then it is schedulable if

$$round\_2(\frac{B}{((2 \times (R_{allocated} + R_i))}, P_{max}) \geq 1 \tag{4}$$

and $S_{max} = 1$ or 2, or if

$$round\_2(\frac{B}{2 \times ((R_{allocated} + \lceil (R_i + 1)/2 \rceil))}, P_{max}) \geq 1 \quad (5)$$

and $S_{max} = 2$. Equations 4 and 5 say that so long as the resulting PFQ is at least 1 then the job can be scheduled by the flexible PFQ algorithm. Unlike the fixed PFQ algorithm, in the flexible PFQ algorithm the system-wide PFQ changes accordingly after a merge request is scheduled using the following formula:

$$round\_2(\frac{B}{2 \times R'_{allocated}}, P_{max}), \quad (6)$$

where $R'_{allocated}$ is the total number of runs allocated after the merge is scheduled.

4 Thrashing control policies

Three run-time thrashing control policies were examined. The first one, referred to as *disabling prefetch policy*, is to disable prefetching temporarily when $B_{NR} \geq P_1$, and enable it again when $B_{NR} \leq P_2$, where B_{NR} is the total number of nonreplaceable buffer pages including those which are pinned or dirty. P_1 and P_2 are two thresholds that are pre-defined. While prefetching is temporarily disabled, no prefetch requests are issued when a prefetch triggering page becomes active. Disabling prefetching slows down the generation of dirty pages in the buffer since the CPU must wait for some synchronous read I/Os to complete, during which dirty buffer pages are pushed out by asynchronous output I/Os.

The second policy is to periodically force synchronous writes. This is referred to as *forcing synchronous write policy*. Since the major reason for thrashing is because most of the buffer pages become dirty (hence not replaceable), forcing synchronous writes can reduce thrashing. Unlike the disabling prefetch policy, forcing synchronous write policy only uses a single threshold. When $B_{NR} \geq W_1$, a synchronous write flag is turned on and all the currently active merges are marked for synchronous writes. The synchronous write flag is turned off immediately after the merge tasks are marked. After a merge is marked for synchronous writes, the very next dirty page generated for that merge is synchronously written onto the disk. Because of a dedicated output disk for each merge step and a FCFS policy, a synchronous write would clean up all the dirty pages of the merge when the write is finished. Also, after a synchronous write request is issued, the merge process releases the CPU and waits until the synchronous write completes. Thus, a synchronous write also temporarily stops prefetching for the merge task. However, unlike the disabling prefetch policy, no additional synchronous reads are needed once the merges resume execution.

The third policy is referred to as *lowering PFQ policy*. Similar to the forcing synchronous write policy, we turn on a synchronous write flag when $B_{NR} \geq W_1$ and mark active merges for synchronous writes. In addition, if the current PFQ is greater than P_{min} then it is lowered by half when the merge tasks resume execution. Instead of totally disabling prefetching as in the first policy, we lower PFQ by half. After it is lowered, the prefetch quantity is doubled again when a new merge task arrives.

5 Simulations

A discrete event-driven simulator consisting of three major components was developed. The first major component is a detailed buffer manager maintaining the LRU stack. It tracks the status of each buffer page, whether it is pinned, unpinned, dirty, or clean. It also monitors the triggering conditions to activate the thrashing control policy in use. These triggering conditions include P_1 and P_2 for the disabling prefetch policy, and W_1 for the forcing synchronous write policy and the lowering PFQ policy.

The second major component implements the scheduling algorithms. The scheduler allocates runs for a merge request if it is schedulable, maintains the total number of runs currently allocated, and re-computes the system PFQ when necessary. In order to control the multiprogramming level, two special waiting queues were implemented to control the total number of merge tasks in the system, one called *allocation queue* and the other *entry queue*. At any time, there can be at most MPL merges in the system, either being processed or waiting to be scheduled in the allocation queue. When a new merge arrives and there are already MPL tasks in the system, it is placed at the entry queue, and is not considered to be in the system. After the entry queue overflows, the arriving request is simply discarded. In our implementation, the sizes of the allocation queue and entry queue are both set to be MPL.

The third major component implements the merge process, including the thrashing control policies and the queueing in the CPU and disks based on the system configuration described in Section 2. When the triggering conditions are detected by the buffer manager, the thrashing control policy will be activated or deactivated. The merge process depletes the active pages from each run and generates dirty pages in the buffer. After an active page is depleted, a GetPage for the next page of the same run is issued. If found in the buffer, the page is pinned and the merge process resumes. Otherwise, the merge process is blocked and waits until the page is read in from disk. Each input disk and output disk maintain a separate request queue, where a total number of D input disks is evenly shared by all the merge requests and a dedicated output disk is assumed for each merge step.

The CPU service times are constants that correspond to the CPU MIPS rating (M MIPS) and the specific instruction pathlengths given in Table 1, including C_{merge} for generate a dirty page, C_{set_up} for initiating a read or write I/O request (both synchronous and asynchronous), and C_{mgm} to pin a page. The I/O service time (not including the queueing time) is estimated as follows. Since the input disks are shared by many runs, the I/O service time of a disk read is computed as

$$T_{seek} + T_{latency} + T_{transfer} \times N_{read}, \qquad (7)$$

where T_{seek}, $T_{latency}$ and $T_{transfer}$ are the average disk seek time, average latency time and average transfer time per page, respectively, and N_{read} is the number of pages read. However, since output is written to a dedicated disk, the I/O service time of a disk write is

$$T_{latency} + T_{transfer} \times N_{write}, \qquad (8)$$

where N_{write} is the number of pages written. That is to say the seek component is eliminated. For disk service times, $T_{seek} = 16$ ms, $T_{latency} = 8.35$ ms and $T_{transfer} = 0.91$ ms were used [3].

The merge request arrival process is assumed to be Poisson with rate λ. Each merge request contains a certain number of runs (uniformly distributed among $R/2$ and $3R/2$ runs, where R is the mean initial runs per merge) and each run contains a fixed number L of pages. The sort key value is assumed to be uniformly distributed among all the initial runs. Therefore, for the first merge step each run has the same depletion rate since each run initially contains the same number of pages. However, if more than one step is needed, the depletion rate of an intermediate sorted run is faster. In fact, it is in proportion to its length. For example, the depletion rate of an intermediate run that is the result of merging 10 initial runs is 10 times that of an initial run.

Finally, in all the simulation results reported in this paper, confidence intervals were obtained (but not shown on the graphs) using the method of batch means [11]. Assuming independence of the estimates derived from each batch, confidence intervals were generated. For most cases, the estimated confidence intervals are very tight, except at very high thrashing levels. Nevertheless, the 90% confidence interval is consistently within 10% of the point estimates for almost all the data-points shown.

6 The thrashing phenomenon

In this section, we illustrate the effectiveness of parallel prefetching and the potential problem of thrashing associated with it. Figures 2 and 3 show the average response times and corresponding synchronous read ratios under different PFQs. (Note the unit of response time in this paper is second.) Two buffer sizes of 10,000 and 2,000 pages were considered. For this figure, we used the fixed-PFQ allocation algorithm with $S_{max} = 1$. Other workload and system parameters were $\lambda = 0.02$ merges/sec, $R = 30$ runs, $L = 200$ pages, and $D = 10$ disks. If prefetching is not used, the average response time is 158.89 seconds for both buffer sizes (not shown in Figures 2 and 3), more than 5 times as much as when PFQ is as small as 1. As shown in Figure 2, if the buffer size is large, the average response time generally improves as PFQ increases. However, if the buffer size is not large enough, severe thrashing starts developing when PFQ increases. As thrashing occurs, the average response time worsens, offsetting the effectiveness of parallel prefetching (see the case of $B = 2,000$ pages when PFQ is 8 and 16 in Figure 2). Figure 3 shows the ratio of the number of synchronous reads over the total number of GetPages. The increase in the synchronous read ratio is a direct result of thrashing. For the case of $B = 2,000$ pages, thrashing begins to develop when PFQ is 4 and becomes more severe as PFQ increases. The small synchronous read ratios on the cases of smaller PFQs represent the synchronous reads from the first page of every sorted run during each merge step.

Thrashing also becomes more severe when more input disks are used. Note that more input disks reduce the interference among the concurrent merges, thus effectively increasing the I/O parallelism in the system. Figures 4 and 5 show the average response time and synchronous read ratio for different PFQs when the buffer size is 2,000 pages. For a smaller PFQ, such as 1 or 2, the response time improves as more input disks are used, since more I/O parallelism can be employed. However, for a larger PFQ such as 8 or 16, the response time worsens as more input disks are used, because thrashing becomes more severe with a higher degree of parallel prefetching.

7 Performance comparisons

In this section, we compare the performance of the three different run-time thrashing control policies by first examining each policy separately and then comparing them. We consider cases of fixed PFQ and flexible PFQ algorithms, and single step and multiple step merges. Various sensitivity analyses were done for different workload and system parameters, such as the merge arrival rate (λ), the mean initial runs per merge (R), the run length (L), the buffer size (B), the CPU MIPS (M), and the number of input disks (D). However, we only show a subset of the results in the following subsections as they convey similar messages.

7.1 Disabling prefetching

We first examine the performance of the disabling prefetch policy, where prefetching is disabled when $B_{NR} \geq P_1$ and enabled again when $B_{NR} \leq P_2$. Since a merge request is simply discarded if the entry queue overflows, we also show the *completion ratio* of the system whenever it is below 1.0. The completion ratio was computed as the ratio of the total number of completed jobs over the sum of the total number of completed and discarded jobs. In our simulations, we also measured the degree of thrashing, which was computed as the ratio of the number of prefetched pages stolen before referenced over the total number of pages prefetched.

The impact of P_1

Figures 6 and 7 demonstrate the impact of different prefetch-disabling thresholds on the average response time and completion ratio using the fixed PFQ algorithm with $S_{max} = 1$. Figures 8 and 9 show the corresponding charts on thrashing and synchronous read ratio. $P_1 = 0.9 \times B$, $0.8 \times B$, and $0.7 \times B$ were used. P_2 was set to be $(P_1 - 0.1) \times B$ for all three cases. (We did simulations on different P_2's, and found no significant difference.) The "no control" case in all the figures means that no run-time thrashing control policy was used. In general, even though thrashing is reduced to a larger degree as shown in Figure 8, the response time worsens (see Figure 6) and the completion ratio drops as prefetching is disabled with a smaller P_1 (see the case of $P_1 = 0.7 \times B$). Note that with $P_1 = 0.7 \times B$, the average response time when the disabling prefetch policy was used is worse than when no control policy was used. This is because the corresponding synchronous read ratio increases significantly due to the fact that prefetching is disabled more frequently with a smaller P_1. For example, with PFQ = 16 if prefetching is disabled, there are 16 pages need to be synchronously read from the disk every time a prefetch triggering page is accessed.

The impact of scheduling algorithms

The same three P_1 values were used to conduct simulations for the flexible PFQ algorithm. The general trend is similar. Namely, depending on the chosen threshold, the average response time using the disabling prefetch policy may only improve slightly or for most cases become worse than using no control policy. Therefore, disabling prefetching in general is not an effective run-time thrashing control policy. To show the sensitivity to a different parameter, Figures 10 and 11 compare the average response time and completion ratio using fixed PFQ and flexible PFQ algorithms for different run lengths. $P_1 = 0.9 \times B$ was used for both scheduling algorithms. There is a big performance difference between the two scheduling algorithms when the system is more heavily loaded, i.e., the run length is larger (400 and 500 pages per initial run). For the fixed PFQ algorithm with $S_{max} = 1$ and PFQ = 16, a large number of merge arrivals have to wait in the allocation queue or entry queue before they can be scheduled for execution. However, for the flexible PFQ algorithm, most of arrivals can be started with a smaller PFQ.

7.2 Forcing synchronous writes

The impact of W_1

Here, we study of the performance of the forcing synchronous write policy, where once $B_{NR} \geq W_1$ a synchronous write flag is turned on and all the active merges are marked for synchronous writes. We first examine the impact of W_1. Figures 12 and 13 show the average response time and completion ratio for the cases of $W_1 = 0.9 \times B$, $0.7 \times B$, and $0.5 \times B$. In general, the average response time reduces as W_1 decreases, but this trend stops as W_1 continues to decrease. This is because, if W_1 is too small, a merge process may be unnecessarily stopped to flush its dirty pages even if thrashing is not about to occur, thus increasing its write I/O time. On the other hand, with a large W_1 (such as $0.9 \times B$), the improvement in response time can be marginal, since too many dirty pages have been built up in the buffer and as a result a substantial amount of prefetched pages have been stolen. However, the optimal W_1 is dependent on the workload and changes dynamically.

The impact of scheduling algorithms

The performance of forcing synchronous writes with different W_1's under the flexible PFQ algorithm is similar to that under the fixed PFQ algorithm. Namely, the response time improves initially as W_1 decreases but it stops improving as W_1 continue to decrease. Figure 14 compare the performance of both scheduling algorithms using the forcing synchronous write control policy with $W_1 = 0.5 \times B$. From Figures 10 and 14, we notice that the forcing synchronous write policy is significantly more effective in controlling thrashing and improving the response time, as compared with the disabling prefetch policy.

7.3 Lowering the prefetch quantity

In this section, we evaluate the performance of the lowering PFQ policy. Note that in the lowering PFQ policy, the system-wide PFQ is lowered by half if PFQ $> P_{min}$ after synchronous writes are forced. In this policy, after the PFQ is lowered, it is doubled again on each subsequent arrival of a merge task until it reaches P_{max}. Since PFQ is re-computed each time a new merge task is scheduled for execution, raising PFQ or not may not be an important issue for the flexible PFQ algorithm. Figure 15 shows the performance of the lowering FPQ policy with $W_1 = 0.5 \times B$ and $P_{min} = 1, 4, 8$ and 16 using the fixed PFQ algorithm. Note that when $P_{min} = P_{max} = 16$, PFQ will never be lowered, hence only synchronous writes are forced. As shown in Figure 15, compared with just forcing synchronous writes, lowering PFQ can further improve the system response time as long as PFQ does not become too small, such as 1 or 2. With a small PFQ, the merge process may depletes the current set of prefetched pages before the prefetching of the next set of pages is completed, thus increasing the waiting time. Lowering PFQ in addition to forcing synchronous writes can reduce the input I/O rate, thus reducing thrashing, especially when P_{max} is large.

7.4 Comparisons of three thrashing control policies

In this section, we compare the performance of the three thrashing control policies. Figures 16 and 17 show the performance of the three different thrashing control policies using the fixed PFQ algorithm. For the disabling prefetching policy, $P_1 = 0.9 \times B$ was used; for the forcing synchronous write policy, $W_1 = 0.5 \times B$ was used; and for the lowering PFQ policy, $W_1 = 0.5 \times B$ and $P_{min} = 4$ were used. The lowering PFQ policy is the most effective thrashing control policy of the three. The larger the run length, the more significant the performance advantage of the lowering PFQ policy becomes. Similar results were also obtained for the flexible PFQ algorithm.

The impact of multiple merge steps

In the simulations of this paper, we set a maximum step S_{max} for each merge to finish. If available buffer pages are not large enough for a merge request, it cannot be scheduled for immediate execution and must wait in the allocation queue. For the fixed PFQ scheduling algorithm, such waiting time could be large. Thus it may improve the average response time if $S_{max} > 1$, even though more work is needed in a multi-step merge. We set the maximum step to 2 and conducted simulations for the three run-time thrashing control policies. The results show that (a) the relative order of the three run-time control policies when $S_{max} = 2$ is similar to that when it is 1, i.e., the lowering PFQ policy is in general the most effective policy; (b) when no run-time control measure is used, increasing the maximum step to 2 can significantly improve the response time for the fixed PFQ algorithm, because when $P_{max} = 16$ and $S_{max} = 1$ many merge tasks have to wait in the allocation queue; and (c) when the flexible PFQ algorithm is used, the maximum step has less effect on the average response time; and (d) when the lowering PFQ policy is used, the response time is less sensitive to S_{max}. Figure 18 shows the average response time for the fixed PFQ algorithm with the maximum step equal to 1 and 2 (for the cases of using the lowering PFQ policy, $W_1 = 0.5 \times B$ and $P_{min} = 4$).

8 Conclusions

In this paper we studied the problem of thrashing due to parallel prefetching and the imbalance between input I/O streams and output I/O streams, and compared the performance of various run-time thrashing control policies for the merge phase of concurrent mergesorts, where initial sorted runs are stored on multiple disks and the final sorted run is written back to another dedicated disk. In a concurrent mergesort environment, which is sequential I/O bound, parallel prefetching can be used to improve the input I/O efficiency. However, thrashing may develop. Thrashing can significantly compromise the effectiveness of parallel prefetching.

We examined run-time control policies under two different scheduling algorithms, one using a fixed prefetch quantity and the other using flexible prefetch quantities. Three run-time control policies involving (a) disabling prefetching, (b) forcing synchronous writes, and (c) lowering PFQ in addition to forcing synchronous writes were studied. Thrashing control was activated when the total number of nonreplaceable buffer pages reaches a pre-defined threshold. These different approaches to thrashing control were evaluated through detailed simulations. The results show that, first, thrashing indeed can severely degrade the response time for concurrent merges using parallel prefetching. With a buffer of limited size, thrashing exists so long as the input and output rates are imbalanced. Thus, a run-time thrashing control is required to dynamically monitor and control thrashing. Second, though effective in reducing the degree of thrashing, disabling prefetching generally worsens the response time since more synchronous reads are needed. Third, forcing synchronous writes in general both reduce thrashing and improve the response time. However, if the threshold for forcing synchronous writes is too small, the response time may worsen. Fourth, lowering the prefetch quantity in addition to forcing synchronous writes is most effective in both reducing thrashing and improving the response time.

In addition to comparing the three thrashing control policies, we also studied the impacts of different scheduling approaches on the response time. Compared with fixed PFQ, flexible PFQ generally reduces thrashing and narrows the gap among the three policies. Although the relative order of the different thrashing control policies are similar under single-step and two-step merges, allowing a maximum step of 2 can reduce the thrashing effect and improve the average response time.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] H. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proc. of Very Large Data Bases*, pages 127–141, 1985.

[3] E. I. Cohen, G. M. King, and J. T. Brady. Storage hierarchies. *IBM Systems Journal*, 28(1):62–76, 1989.

[4] D. J. DeWitt, D. Bitton, H. Boral, and W. K. Wilkinson. Parallel algorithms for relational database operations. *ACM Trans. on Database Systems*, 8(3):324–353, 1983.

[5] D. J. DeWitt et al. Implementation techniques for main memory database systems. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 1–8, 1984.

[6] C. Faloutsos, R. Ng, and T. Sellis. Predictive load control for flexible buffer allocation. In *Proc. of Very Large Data Bases*, pages 265–274, 1991.

[7] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1984.

[8] B. R. Iyer and D. M. Dias. System issues in parallel sorting for database systems. In *Proc. of Int. Conf. on Data Engineering*, pages 246–255, 1990.

[9] D. E. Knuth. *The Art of Computer Programming, Vol 3: Sorting and Searching.* Addison-Wesley, 1973.

[10] S. C. Kwan and J. L. Baer. The I/O performance of multiway mergesort and tag sort. *IEEE Trans. on Computers*, 34(4):383–387, 1985.

[11] S. S. Lavenberg, editor. *Computer Performance Modeling Handbook.* Academic Press, 1983.

[12] R. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 387–396, 1991.

[13] V. S. Pai and P. J. Verman. Prefetching with multiple disks for external mergesort: Simulation and analysis. In *Proc. of Int. Conf. on Data Engineering*, pages 273–282, 1992.

[14] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 109–116, 1988.

[15] G. M. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Trans. on Database Systems*, 11(4):473–498, 1986.

[16] B. Salzberg. Merging sorted runs using large main memory. *Acta Informatica*, 27:195–215, 1989.

[17] J. Z. Teng and R. A. Gumaer. Managing IBM Database 2 buffers to maximize performance. *IBM Systems Journal*, 23(2):211–218, 1984.

[18] P. S. Yu and D. W. Cornell. Buffer management based on return on consumption in a mulit-query environment. *VLDB Journal.* to appear.

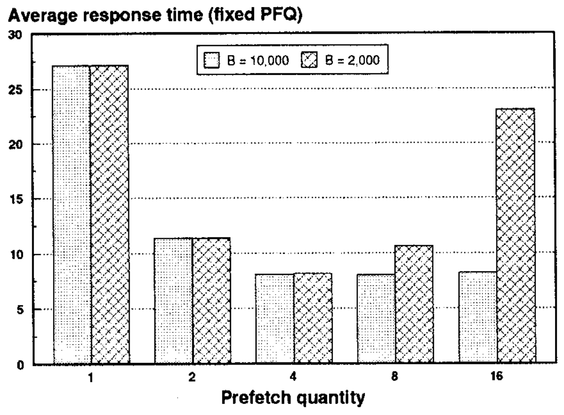

Figure 2: The impact of thrashing on the average response time.

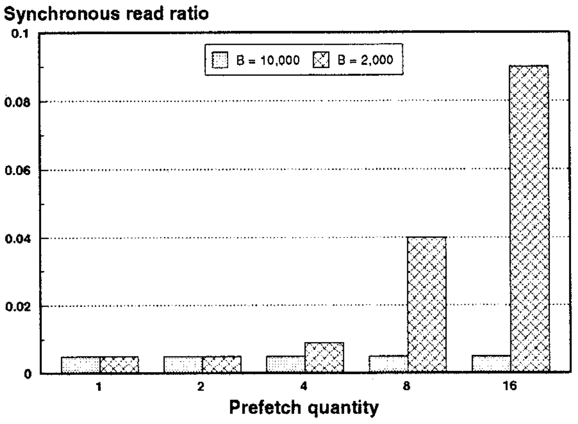

Figure 3: The impact of thrashing on synchronous read ratio.

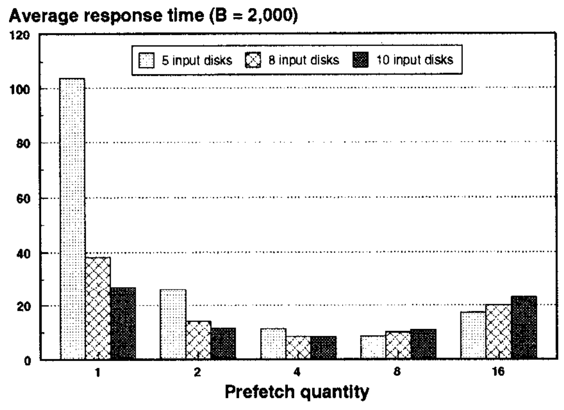

Figure 4: The impact of input parallelism on response time.

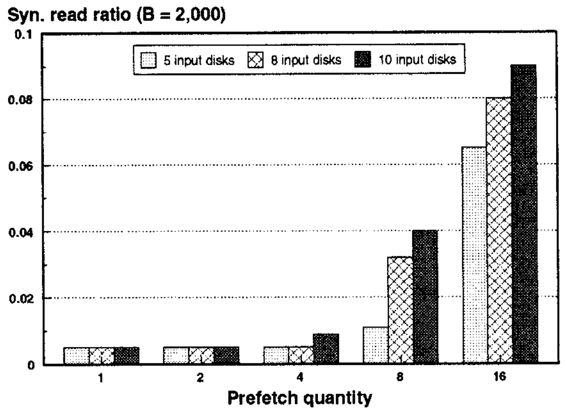

Figure 5: The impact of input parallelism on synchronous read ratio.

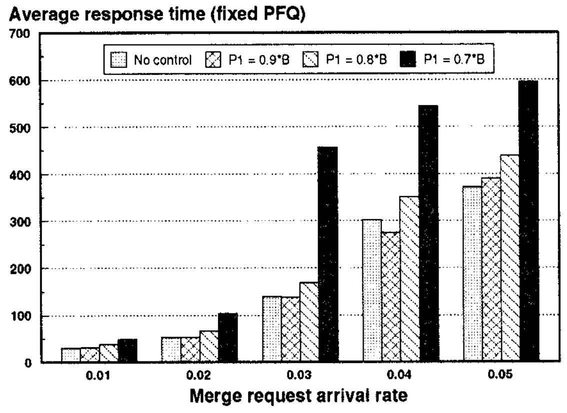

Figure 6: The impact of prefetch-disabling thresholds on average response time.

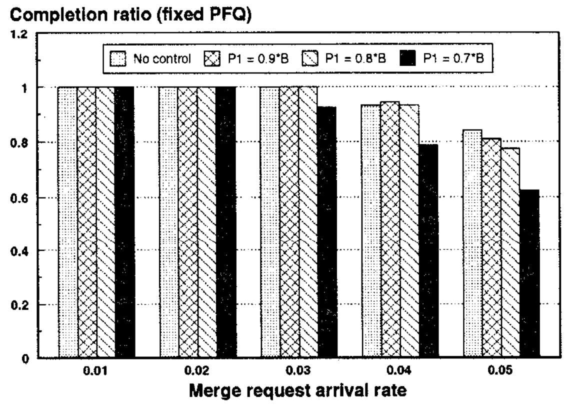

Figure 7: The impact of prefetch-disabling thresholds on completion ratio.

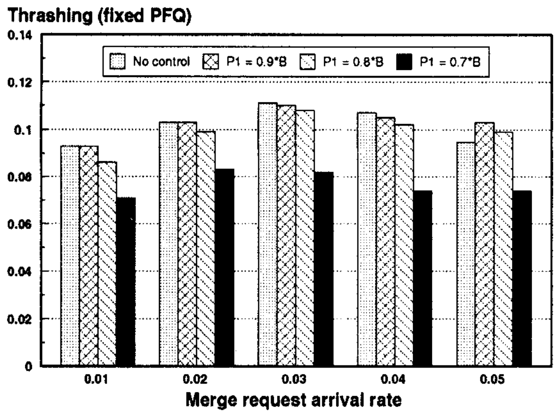

Figure 8: The impact of prefetch-disabling thresholds on degree of thrashing.

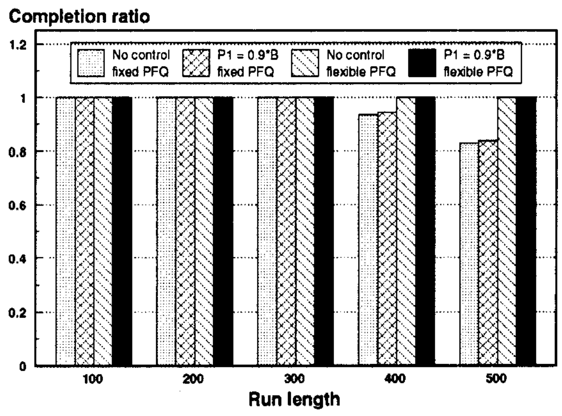

Figure 11: The impact of scheduling algorithms on completion ratio using disabling prefetch policy.

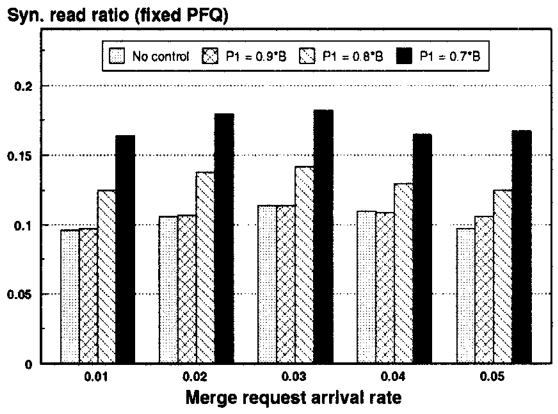

Figure 9: The impact of prefetch-disabling thresholds on synchronous read ratio.

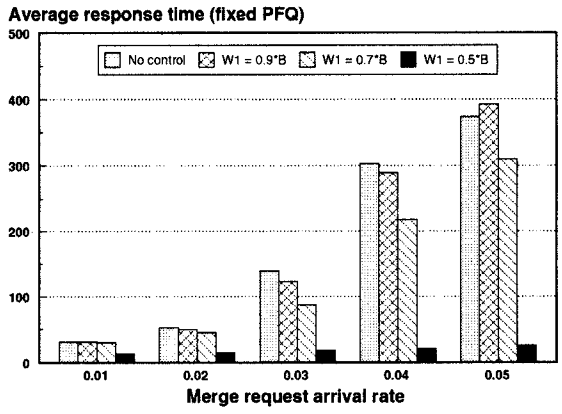

Figure 12: The impact of synchronous-write thresholds on response time.

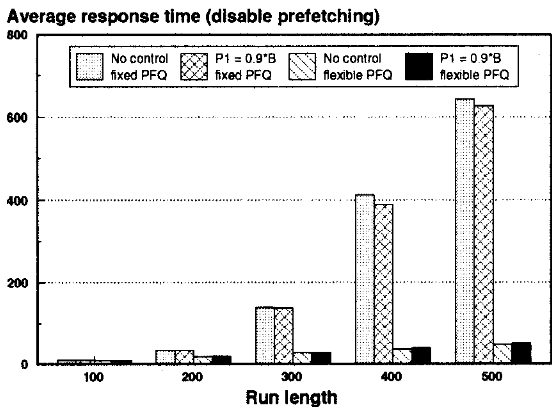

Figure 10: The impact of scheduling algorithms on response time using disabling prefetch policy.

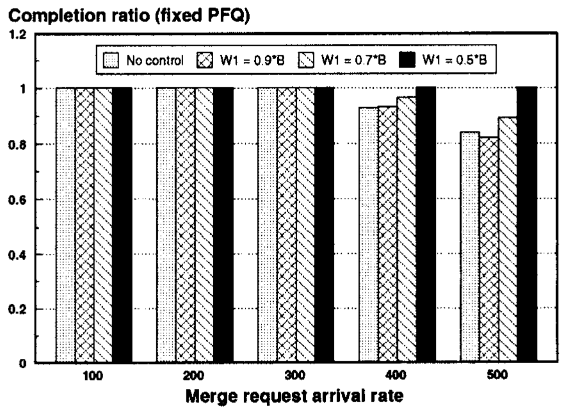

Figure 13: The impact of synchronous-write thresholds on completion ratio.

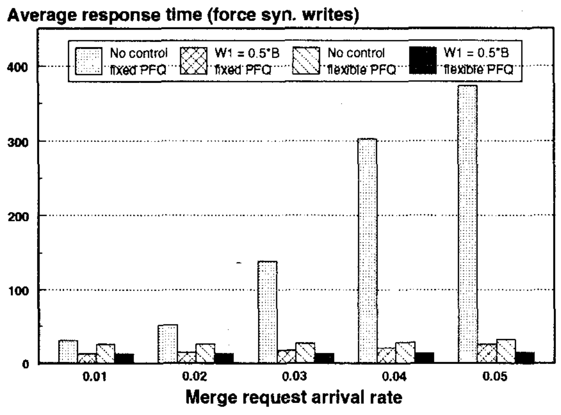

Figure 14: The impact of scheduling algorithms on response time using forcing synchronous write policy.

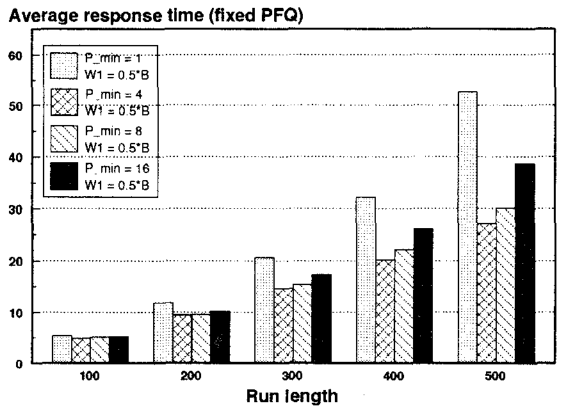

Figure 15: The impact of P_{min} on response time using the lowering PFQ policy.

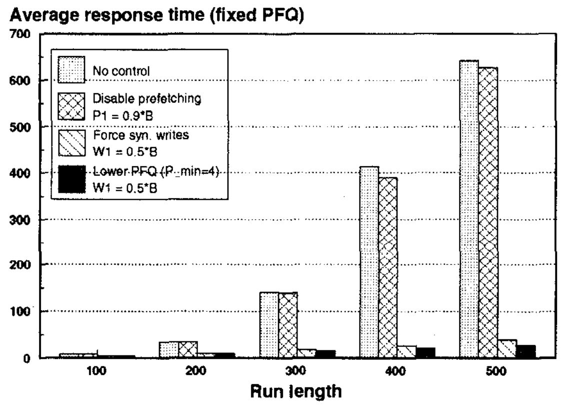

Figure 16: Average response times of the three thrashing control policies using fixed PFQ algorithm.

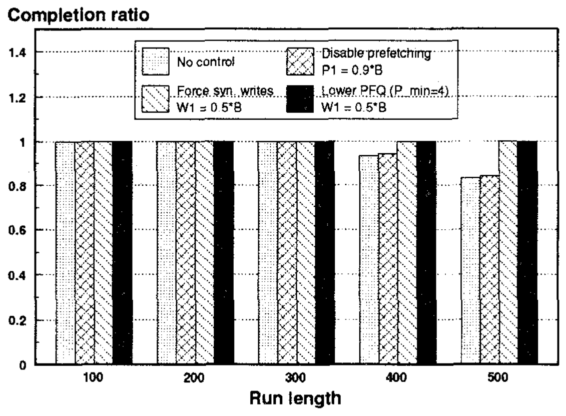

Figure 17: Completion ratios of the three thrashing control policies using fixed PFQ algorithm.

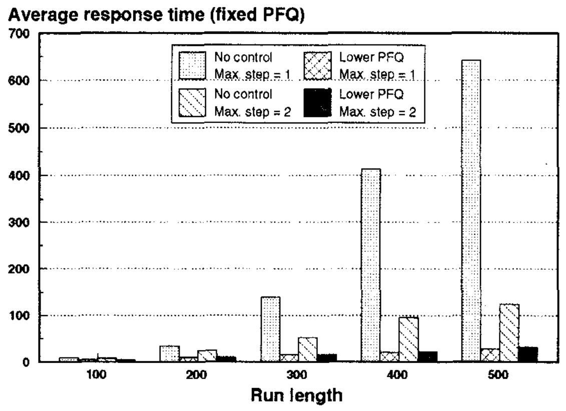

Figure 18: The effect of S_{max} on response time using the fixed PFQ algorithm ($W_1 = 0.5 \times B$ and $P_{min} = 4$ for the lowing PFQ cases).