Managing Memory to Meet Multiclass Workload Response Time Goals

Kurt P. Brown
Michael J. Carey
Miron Livny

Technical Report #1146

April 1993

Managing Memory to Meet Multiclass Workload Response Time Goals*

Kurt P. Brown† Michael J. Carey Miron Livny

Computer Sciences Department
University of Wisconsin, Madison
{brown,carey,miron}@cs.wisc.edu

Abstract

In this paper we propose and evaluate an approach to DBMS memory management that addresses multiclass workloads with per-class response time goals. It operates by monitoring per-class database reference frequencies as well as the state of the system relative to the goals of each class; the information that it gathers is used to help existing memory allocation and page replacement mechanisms avoid making decisions that may jeopardize performance goals.

1 Introduction

A widening range of application areas, as well as requirements for data sharing and continuous operation, are contributing to an increase in the diversity of workloads that a DBMS must be able to cope with. However, providing adequate performance for each class in a multiclass DBMS workload is still an open problem [Pirahesh 90, Brown 92, DeWitt 92]. A multiclass workload is characterized by distinct classes of work that may have widely varying resource demands, each with its own performance objective. A DBMS that is unaware of these performance objectives may penalize one class or another in an unpredictable way. Consider the issue of buffer page replacement, for example. A replacement policy based on recency of reference will tend to penalize workload classes with low locality; one based on frequency of reference may be biased against workload classes with low arrival rates; and a policy which uses hints about the relative value of pages based on their type (e.g. index or data) will be biased against whatever workload class uses the "wrong" page type. In order to avoid such "hard-wired" biases, a DBMS must be able to accept performance objectives for each class as inputs, and to use those goals as the basis for its resource management decisions.

Given a set of performance objectives for each class, there are a number of mechanisms that a DBMS can use to achieve them: load control, CPU scheduling, disk scheduling, and memory management. While a complete solution to the problem of satisfying performance goals in a multiclass environment would likely include all of these options, in this paper we investigate the use of memory allocation and page replacement mechanisms for this purpose. It is well known that memory management is a critical factor in database system performance, which accounts for the

*This work was partially supported by the IBM Corporation through a Research Initiation Grant. An abridged version of this paper is to appear in the *Proceedings of the 19th Int'l VLDB Conference*, Dublin, Ireland, Aug 1993.

†Supported by an IBM Resident Study Fellowship.

large volume of ongoing research in this area [Chou 85, Sacco 86, Cornell 89, Robinson 90, Ng 91, Falou 91, Yu 93, O'Neil 93]. However, none of the previous work specifically addresses how memory management can be used to achieve per-class performance objectives for a multiclass workload.

There are two ways that memory can be used to improve DBMS performance: for buffering disk pages, and for working storage areas (join hash tables, sort work areas, etc.). At any point in time, some number of pages are being used for disk buffers (the *disk buffer region*), and some are being used for working storage (the *working storage region*). A DBMS memory allocation policy is responsible for two decisions: it must decide how many pages to devote to disk buffers versus working storage, i.e. it must logically "draw a line" between the disk buffer region and the working storage region; and it must allocate memory within the working storage region among competing transactions.[1] A page replacement policy is responsible for deciding which specific disk pages should reside in the disk buffer region at any point in time.

In a multiclass environment these decisions need to be driven by per-class performance goals, as stated earlier. For workloads that vary over time, they need to be dynamic as well. Otherwise, the response time goals will at best be satisfied only on average, where the average is defined over a large enough interval of time to eliminate any workload variance. For many workloads, this time frame would have to be extended to days, or even weeks. In contrast we would like performance goals to be satisfied over time frames on the order of tens of seconds or minutes.

In this paper, we propose and evaluate an approach to DBMS memory management called *fragment fencing* that specifically addresses multiclass workloads with per-class performance objectives. It is designed to be used in conjunction with existing page replacement and allocation mechanisms and acts to prevent allocation or replacement decisions that could violate the performance objectives of a class. Fragment fencing operates by periodically monitoring per-class database reference frequencies as well as the state of the system relative to the goals of each class; it then uses this information to dynamically set the boundary between the disk buffer and working storage regions of memory, and to guide the allocation of pages within the disk buffer region to different *fragments* of the database.

The remainder of the paper is organized as follows: We begin by reviewing existing memory management techniques in Section 2. The fragment fencing algorithm is then presented in Section 3. We describe the simulation model used to evaluate fragment fencing in Section 4, and we show the results of that evaluation in Section 5. Section 6 discusses some additional issues and possible extensions to fragment fencing, and our conclusions and future plans are summarized in Section 7.

2 Related Work

With respect to database memory management, the only relevant work which specifically addresses multiclass workloads are commercial systems, such as IBM's DB2 [Cheng 84, Teng 84], which provides basic mechanisms to partition its buffer pool and to place different portions of the database in specific partitions. DB2's page replacement policy is

[1] It would also be responsible for allocating memory within the disk buffer region as well, if a *local* allocation policy is used there. More commonly, a *global* allocation policy is used for the disk buffer region, and individual disk buffer pages are never explicitly assigned to any individual transaction.

local within each partition, so competition between the different pools is eliminated. While in theory, this mechanism could be used to satisfy multiclass performance objectives, there are two problems in using it for this purpose. First, it is static in nature, so it cannot respond to workload variance and shifts. Second, the connection between response time goals for each workload class and which parts of the database to place in each partition, as well as the relative sizes of each partition, must be somehow determined manually by the database administrator. Ideally, we would like the DBMS to perform these tasks dynamically, based on the current system state and the response time goals.

We categorize recent developments in database buffer management into three categories: *modified global LRU, frequency-based,*and *local query analysis.* The modified global LRU approaches extend a basic global LRU allocation and replacement mechanism by permitting query operators to provide hints to the buffer manager about the relative "value" of a page. For example, index pages could be considered more valuable than data pages, as in the Domain Separation algorithm [Reiter 76]; randomly accessed pages could be treated as more valuable than sequentially accessed pages, as in the DB2 Buffer Manager [Cheng 84, Teng 84]; or the inner relation of a nested loop join could be preferred over the outer, as in the Starburst Buffer Manager [Haas 90]. Information on the value of a page is then combined with information on recency of reference and used as input to guide page replacement decisions. These approaches are attractive because they address the major limitations of pure global LRU with a minimum amount of work. However, the hints are based on static heuristics that are unrelated to response time goals, and therefore may be inappropriate in a multiclass environment.

The second category of memory management approaches combines information on *frequency* of reference with *recency* of reference into the replacement criteria. This is logical because recency of reference is a good basis for replacement when database references exhibit temporal locality, while frequency of reference is best when references are skewed, but uncorrelated [Coffman 73]. Real database reference behavior is a combination of both. The Frequency Based Replacement policy (FBR) [Robinson 90] and the LRU-K algorithm [O'Neil 93] are examples of this approach, tracking frequency statistics on a page-by-page basis. The Bubba parallel database prototype [Boral 90] can be placed in this category as well, but unlike FBR and LRU-K, which are both dynamic, Bubba statically determined a boundary between that portion of memory which is managed by frequency of reference (the file cache), and that which is managed by recency (normal global LRU). This boundary is determined off-line by a "5 Minute Rule" type of analysis [Gray 87]. The Bubba scheme tracks frequency information on a per-file basis and uses a size-normalized frequency metric called *temperature* [Copeland 88] (references per second per megabyte). Entire files are statically placed in the file cache in decreasing order of temperature. By statically or dynamically combining frequency and recency into the replacement policy, these approaches each provide better performance than pure LRU while avoiding any requirements for "hint-passing".

Examples of the local query analysis approach are Hot Set [Sacco 86], DBMIN [Chou 85], Marginal Gains [Ng 91], Predictive Load Control [Falou 91], and Threshold [Yu 93]. All of these algorithms use information in the query plan to determine the optimal amount of memory to allocate on a local basis (to queries, subqueries, or query/file combinations). The Hot Set, DBMIN, Marginal Gains, and Predictive Load Control approaches all address disk buffer memory allocation, and the Threshold algorithm addresses working storage allocation. However, none of them address the trade-off between the two types of memory. Interestingly, although some of these algorithms

use response time *predictions* internally (e.g. Predictive Load Control and Threshold), none of them are driven by response time *goals*. An obvious question is whether these approaches can be modified to be *driven* by their response time predictions instead of just using them as a means to another end. Unfortunately, the difficulty with trying to drive them by their response time predictions is that they can be quite inaccurate when trying to predict transient response times, especially in a multiclass environment where each class has widely varying resource demands. Buffer hit rates, communication delays, lock waits, and queuing at the disk and CPU are all factors that can significantly affect the performance of a query when it runs concurrently with other work.

3 Fragment Fencing

3.1 Overview

Before we can explain how fragment fencing works, we must first define the terms *performance goal* and *fragment*. While there are many possible ways to specify a performance goal, it will be defined for our purposes as follows: for each workload class, the DBMS will attempt to maintain a user specified average response time. Of course, some response times will exceed the goal and some will be below it, but the average of all response times for a class should approach the goal as the number of transaction completions increases. If a response time goal is not specified for a workload class, then we expect the DBMS to "do its best" with respect to that class. In addition, because we are primarily interested in allocation and replacement policies in this study, we do not allow any work to be postponed by a load controller; it must be allowed to execute upon arrival, even if it has no goal specified.

A *fragment* is a statically determined set of database pages that have relatively uniform access probabilities. It is simply a generalization of any distinct external storage structures used by a DBMS, and its actual definition would be DBMS-specific. A fragment could correspond to the operating system files that store the database, or it could be composed of a subset of file pages. One example of a file that could be broken up into multiple fragments is a tree-structured index. Each level of the index tree could be a separate fragment because the pages in each level have distinct access probabilities. A relational DBMS that stored multiple relations in the same operating system file would likely define each relation as a fragment. For the rest of the paper, we assume that the term fragment refers either to a single index level or an entire data file.

Given a set of response time goals for each workload class, and a set of fragments that each class references, the basic idea behind fragment fencing is to achieve the response time goals for a class by individually controlling the hit rates on the fragments referenced by the class. For each fragment, the algorithm determines a *target residency*, which is the minimum number of the fragment's pages that should remain memory resident in order to meet response time goals. Response times for each class are continuously checked by the algorithm at well defined intervals and if a class is not meeting its goal, then the target residencies for fragments referenced by that class are increased. If a class is over-performing relative to its goal, the target residencies are decreased. The actual amount of each fragment to retain in memory is determined using two inputs: the observed access frequencies of each fragment (those with higher access frequencies are favored for memory residency), and a "best guess" as to the response time improvement that will result when the fragment's memory residency is increased. The details of this process are discussed in Section

3.4.

Target residencies for each fragment are enforced by modifying the existing (*base*) replacement policy to avoid stealing a page if that would bring the number of memory resident pages below the target for a fragment. Enforcing target residencies thus provides a passive way to "fence off" fragments from the possibility of replacement when they would otherwise be chosen by the existing replacement criteria.

Any individual fragment may transiently be in one of three states: *in deficit* (below target), *on target*, or *in surplus* (exceeding its target). A fragment can be in deficit immediately after its target residency increases, and will remain so until enough pages are faulted in to meet its target. If the demand for memory is low, fragments may exceed their targets and will then be susceptible to stealing by the normal page replacement mechanism when the demand for memory rises. Just like fragments, the system as a whole can be in one of three states: it is *in deficit* when one or more fragments are in deficit, *in surplus* when no fragments are in deficit and one or more are in surplus, and *on target* when no fragments are in deficit or surplus.

At any particular moment, the sum of the target residencies for every fragment in the database is called the *resident volume*, and the size of the remaining portion of memory is called the *unreserved volume*. The resident volume dynamically determines a "line" that sets a *minimum* size for the disk buffer region of memory. The resident volume should obviously not grow so large as to consume all of available memory. At the very least, enough must be set aside to satisfy the minimum requirements of the average set of concurrently executing transactions. Therefore, we limit the resident volume to 80% of available memory.[2] Memory which is not reserved for caching fragments can be allocated either for working storage or for additional disk buffer pages, as determined by the base allocation policy. The base allocation policy is responsible for insuring that the sum of all allocated working storage does not exceed the unreserved volume. Figure 1 illustrates these concepts. Note that the line separating the memory reserved for caching fragments and the unreserved memory is dynamic, whereas the line which defines the amount of memory set aside for minimum transaction requirements is static.

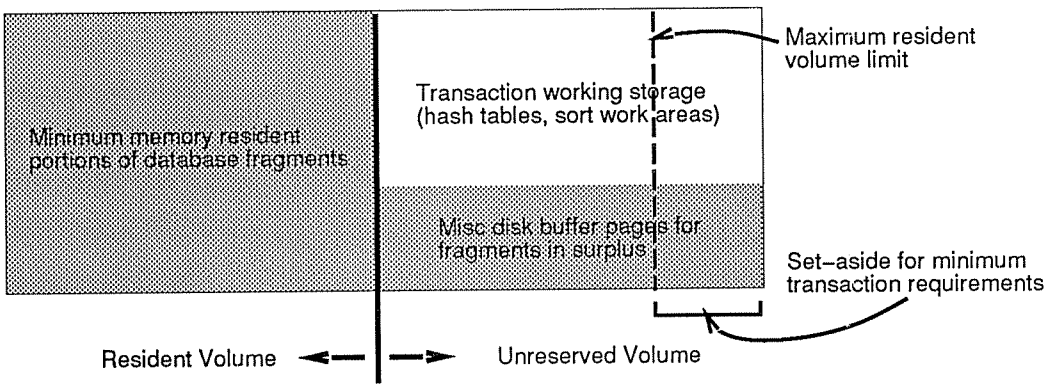

Figure 1: Logical Memory Layout

[2] Athough this constant would be DBMS dependent, for this study we chose 80% as a reasonable limit.

3.2 Implementation Details

Fragment fencing maintains the following state data about classes and fragments:

Global data:		
N_{def}	# of fragments in deficit	(observed)
N_{sur}	# of fragments in surplus	(observed)
$Resvol$	Resident volume	(calculated)
For each fragment f:		
$Size_f$	Size, in pages	(input)
Res_f^{curr}	Current # of memory resident pages	(observed)
Res_f^{target}	Target # of memory resident pages	(calculated)
$SzDiskRes_f$	$Size_f - Res_f^{target}$	(calculated)
For each workload class c:		
R_c^{goal}	Response time goal	(input)
I_c	Observation interval length	(input)
IO_c^{obsv}	Avg # disk I/Os (buffer misses) per transaction during observation interval I_c	(observed)
R_c^{obsv}	Avg transaction response time during observation interval I_c	(observed)
For each fragment/class combination f, c:		
$Refs_{f,c}$	# references to frag f by an average class c transaction during observation interval I_c	(observed)
$Hits_{f,c}$	# buffer hits on frag f by an average class c transaction during observation interval I_c	(observed)
$Misses_{f,c}$	$Refs_{f,c}$ - $Hits_{f,c}$	(calculated)

The observation interval length I_c indicates the frequency at which response time goals should be checked by the algorithm. For example, with an interval of 100 completions, each group of 100 individual transaction response times are averaged together to form an *interval response time*, which is then compared against the user specified response time goal. I_c is actually maintained as two values: one is a number of transaction completions, and the other records the number of seconds that elapsed during those completions. Shorter intervals result in more responsive behavior and longer intervals result in more stable behavior. Ideally, the tradeoff between stability and responsiveness should perhaps be decided by the user and not the DBMS, but in the initial version of fragment fencing, we explicitly set the interval size by hand for each workload (see Section 4.3).[3] The observed I/Os and response times (IO_c^{obsv} and R_c^{obsv}) as well as the reference and hit counts ($Refs_{f,c}$ and $Hits_{f,c}$) are all relative to the current observation interval only, and are reset to zero at the start of every interval.

On every buffer reference to a fragment f from a class c, the algorithm increments $Refs_{f,c}$. For a buffer hit or miss it increments $Hits_{f,c}$ or IO_c^{obsv}, respectively. Res_f^{curr} is also updated for the current fragment, if necessary, as well as for any fragment whose page was replaced. N_{def} and N_{sur} are also updated if any page movement between disk and memory changes the state of a fragment.

On every transaction completion for a class c which has a response time goal specified, the observed response time for the transaction is added to the running average for the class. If the current interval, I_c, has expired, then the next action to take is based on the current job class state:

[3] We explore the sensitivity of fragment fencing to different observation interval sizes in Section 5.2.

- **Warmup**: The class is waiting for the buffer to fill up after a system cold start. All job classes enter the warmup state on system initialization, and they all leave the warmup state simultaneously when the replacement policy first kicks in, moving to the *history build* state. No action is taken on this transition except to reset all statistics.

- **Transition Up**: A class enters this state if any target residency was increased in order to satisfy its goal. When the system leaves the deficit state ($N_{def} = 0$), the class is moved to the *history build* state. No action is taken except to reset all statistics.

- **Transition Down**: This state is similar to transition up, but is entered when target residencies were decreased. The class is moved to the *history build* state when $N_{sur} = 0$. No action is taken except to reset all statistics.

- **History Build**: A class enters this state from the *warmup*, *transition up*, or *transition down* states. Movement to the history build state is required in order to achieve a statistically significant sample of the newly obtained system state (a recently changed resident volume). The time in this state is set to a number of transaction completions that provides statistical significance. We currently set it to 50 in all cases, but this length could also be dynamically determined for each class using sampling techniques [Haas 91]. If response time goals are being met at the end of 50 completions, then the class is moved to *steady state*, otherwise new target residencies are set, statistics are reset, and the class moves to *transition up* or *transition down*.

- **Steady State**: A class enters steady state when its response time goals are being met. The goals are checked again after I_c completions; if they are still being met, then this state is entered once again to wait another I_c completions. If the goals are not being met, new target residencies are set, statistics are reset, and the class moves to *transition up* or *transition down*.

3.3 Checking the Goals

If the observed average response time for a class c is within plus or minus some percentage of the user-specified response time goal (i.e. with some tolerance band, T_c of the goal), then the goals are considered to be satisfied. Otherwise, if the observed response times are higher than the goal, target residencies for one or more fragments referenced by class c are increased and the class is placed in the *transition up* state. If the observed response time is lower then the goal, then one or more target residencies are decreased and the class is placed in the *transition down* state. While our definition of performance goals allows a class to over-achieve, we still want to lower target residencies if we can. The motivation for this is to insure that the amount of memory available for working storage is always as large as possible.

As is typical of any feedback mechanism, T_c turns out to be the most sensitive parameter for fragment fencing. If there is a large amount of "natural" variance in the class's response times, then T_c must be wide enough to prevent the algorithm from attempting to manage natural statistical fluctuations. A narrow T_c should be used with lower variances in order to reduce the number of interval response times that violate the goals.

The value of T_c cannot be set a priori, as it depends on the workload and the dynamic state of the system. Therefore, the algorithm computes it dynamically based on the observed standard deviation across multiple intervals.

Given a sufficient number of samples, the distribution of average interval response times can be approximated by a normal distribution. We therefore set T_c such that it includes 90% of the area under a normal distribution curve (i.e. T_c is plus or minus 1.65 times the observed standard deviation). However, we must take care in the standard deviation calculation to avoid including any observations that occur during transitions in resident volume. These observations would act to inflate the algorithm's estimation of natural variance in the workload, and T_c would then become excessively large (loose). Therefore, observations are only added to the running computation of standard deviation if the workload class has observed some consecutive number of steady state intervals. A default tolerance band (currently set to plus or minus 10%) is used until T_c can be computed from actual response time observations.

In addition to insuring that we record only "natural" variance, we must also recompute the standard deviation for a class after it undergoes any transition in target residencies. This is because the existing sums and sums of squares used to compute the standard deviation are all relative to a previous set of target residencies, and therefore they are all relative to a different mean response time as well. Combining observations previous to the transition with observations after the transition will also result in a higher estimation of variance than is occurring naturally in the workload. Thus, on any transition, the running sums and sums of squares used to compute the standard deviation are reset, and the previous T_c is used temporarily until there have been enough consecutive steady state intervals under the new target residencies to allow the standard deviation to be recomputed.

3.4 Changing Target Residencies

If a class is not meeting its goals, then the fragment fencing algorithm makes an "informed guess" regarding new target residencies that would move it towards its goal. Its guesses are based on a simplistic model of transaction behavior that consists of two assumptions:

1. Transaction response times are directly proportional to the number of I/Os that they require (the *I/O dominance assumption*).

2. Hit rates observed on a particular fragment will be equal to the percentage of that fragment which is memory resident (the *hit rate assumption*).

The most common reason for a violation of the first assumption (I/O dominance) is that the bottleneck resource for a particular workload class may be something other than the disk. The extent to which the second (hit rate) assumption holds depends on the degree to which accesses within the fragment are uniformly distributed and on how the base replacement policy deals with different access patterns. Because the algorithm is continually observing the system and readjusting target residencies based on those observations, violations of these two assumptions are not critical. However, extreme cases can cause the algorithm to "try too hard," meaning that it could increase the resident volume even when there is very little benefit in terms of response time improvements.

Using the state data maintained for each class and fragment, together with the model of transaction behavior just described, new target residencies for a class c are determined in two steps: calculating the change in I/Os required for the class, and setting target residencies in order to achieve that I/O increase or reduction. The change in the

number of I/Os for an average transaction of class c is computed using the I/O dominance assumption as follows:

$$IO_c^{target} = IO_c^{obsv}/(R_c^{obsv}/R_c^{goal})$$
$$\Delta IO_c = IO_c^{obsv} - IO_c^{target}$$

Note that ΔIO_c will be positive if I/Os are to be reduced, or negative if they are to be increased. In order to dampen the feedback mechanism, we limit ΔIO_c to at most 20% of IO_c^{obsv} on any individual change in target residencies (thus I/O deltas larger than 20% require multiple observation intervals to be achieved).

Setting Target Residencies

Every fragment has a certain observed frequency of reference by the transactions of a class, and the fragments with higher reference frequencies should be favored for memory residency over those with lower frequencies. On the other hand, some fragments are much larger than others; therefore, for a given frequency of reference, small files should be favored over larger ones, as their per-page reference frequencies will be higher. The notion of *temperature* [Copeland 88] combines both of these factors into a single number of references per second per megabyte. We extend the definition of temperature to record access frequencies for a specific class instead of for the system as a whole, and we call the resulting metric *class temperature*. Each fragment has a class temperature for every class that references it.

If a class is not meeting its response time goals, ΔIO_c will be positive, and target residencies will need to be increased. In this case, fragments are sorted in decreasing order of class temperature ("biggest bang for the buck" first). If a class is exceeding its goals, ΔIO_c will be negative, and target residencies will need to be decreased. Here, fragments will be sorted in increasing order of class temperature ("lowest bang for the buck" first).

Each fragment f is then processed in sorted order. First, the absolute change in the fragment's hit rate (as compared to its current hit rate) that is required to achieve ΔIO_c is computed as:

$$\Delta hitrate_{f,c} = \begin{cases} MIN(1.0,\ \Delta IO_c/Misses_{f,c}) & \text{if } \Delta IO_c > 0 \\ MAX(-1.0,\ \Delta IO_c/Hits_{f,c}) & \text{otherwise} \end{cases}$$

If the absolute value of $\Delta hitrate_{f,c}$ is greater than 1.0, this means that fragment f is not currently experiencing enough buffer misses (hits) from class c to completely satisfy the required ΔIO_c, so the next fragment in the sorted list will need to be investigated as well. Otherwise, fragment f can accomplish the change in I/O by itself; in this case, the increase (or decrease) in hit rate is simply equal to the ratio of ΔIO_c to $Misses_{f,c}$ (or $Hits_{f,c}$ for a hit rate decrease). Finally, the hit rate assumption is used to translate hit rate changes into absolute target residency changes (as compared to the current target residency) as follows:

$$\Delta Res_f^{target} = \begin{cases} SzDiskRes_f * \Delta hitrate_{f,c} & \text{if } \Delta hitrate_{f,c} > 0 \\ Res_f^{target} * \Delta hitrate_{f,c} & \text{otherwise} \end{cases}$$

Changes in target residencies and total resident volume are limited to 10% of available memory, in order to dampen the feedback mechanism.

To illustrate the process just described, consider a class c with a response time goal of 1 second and an observed response time of 1.5 seconds. Suppose that class c references two fragments, f_1 and f_2, with an average of 5 buffer

misses on f_1 and 25 misses on f_2, for an average of 30 disk I/Os per class c transaction. We first compute a target number of I/Os that would result in a 1 second (goal) response time as $IO_c^{target} = IO_c^{obsv}/(R_c^{obsv}/R_c^{goal}) = 30/(1.5/1.0) = 20$, and thus $\Delta IO_c = IO_c^{obsv} - IO_c^{target} = 30 - 20 = 10$. Assuming that fragment f_1 has the higher class temperature, we compute the required increase in f_1's hit rate as $\Delta hitrate_{f,c} = MIN(1,\ \Delta IO_c/Misses_{f,c}) = MIN(1, 10/5) = 1$. Becase $\Delta hitrate_{f,c} = 1$, f_1 cannot satisfy the change in I/O all by itself. We make all of f_1 memory resident, taking care of 5 out of the 10 I/Os that we are trying to eliminate, leaving a ΔIO_c of 5 which must be satisfied by fragment f_2. The required change in hit rate for f_2 is $MIN(1, 5/25) = 0.2$. Suppose that f_2 is 200 pages and that it has a current target residency of 100 pages. Therefore, if we need to increase f_2's hit rate by 20%, we must bring in 20% of its 100 disk resident pages, resulting in a new target residency for f_2 of 120 pages.

4 Simulation Model

The simulator that we use for our performance study of fragment fencing was built as part of an ongoing investigation into resource allocation and scheduling for parallel database systems. For this study, however, we define a very simple centralized configuration that consists of one processing node with a single CPU, memory, and two disks. The remainder of this section provides a more detailed description of the relevant portions of the current simulation model, and concludes with a table of the simulation parameter settings used for this study.

4.1 Hardware and Software Configuration Model

Terminals

The simulated terminals model the external workload source for the system. Each terminal submits a stream of transactions of a particular class, one after another. As each transaction is formulated, the terminal sends it to the DBMS for execution and then waits for a response before continuing on to the next transaction. In between submissions, each terminal "thinks" (i.e. waits) for some random (exponentially distributed) amount of simulated time. The number of terminals and the think times used in this study were chosen to insure an average disk utilization of 50 to 60% under normal operating conditions.

Disks

The simulated disks are modeled after the Fujitsu Model M2266 (1 GB, 5.25") disk drive. This disk provides a 256 KB cache that we divide into eight 32 KB cache contexts for use in prefetching 8K pages for sequential scans. In our model of the disk, which is a slight simplification of the real disk, the cache is managed in the following manner: Each I/O request, along with the required page number, specifies whether or not prefetching is desired. If so, one context's worth of disk blocks (4 blocks) are read into a cache context after the originally requested data page has been transferred from the disk to memory. The requester is not released until the entire cache context is loaded, however (synchronous cache loading). Subsequent requests to one of the prefetched blocks can then be satisfied without incurring an I/O operation. A simple round-robin replacement policy is used to allocate cache contexts if the number of concurrent prefetch requests exceeds the number of available cache contexts. The disk queue is

File name	# records	Record size	# pages	Fraction of memory
big file	100,000	100	1234	2.400
big index	100,000	16	196	0.380
medium file	40,000	100	493	0.960
medium index	40,000	16	79	0.150
small file	10,000	100	123	0.240
small index	10,000	16	20	0.040
tiny file	1,000	100	12	0.020
tiny index	1,000	16	2	0.004
query files	20,100	200	502	0.980

Table 1: Database characteristics

managed using an elevator algorithm.

CPU and Memory Management

The CPU is scheduled using a round-robin policy with a 5 msec time slice. The buffer pool models a set of main memory page frames, 8K bytes each. We use two base replacement and allocation policies in this study: pure global LRU, and a modified global LRU scheme augmented with 3 levels of hints. The hints are given by the query execution operators when a page is unfixed, and define 3 levels of value as follows: index pages are considered more valuable than data pages, and randomly accessed data pages are considered more valuable than sequentially accessed data pages. Pages are chosen for replacement in the following order: unused frames (not mapped to any database page), sequentially accessed data pages using an MRU criteria, randomly accessed data pages using an LRU criteria, and finally, index pages using an LRU criteria. A memory reservation mechanism allows query execution operators to reserve memory for their working storage, preventing those reserved frames from being stolen while the reservation is in effect. This function is used by hash join operators to reserve memory for their hash tables.

4.2 Database Model

The database is modeled as a set of files, each of which can can have one or more associated B+ tree indices. All of the indices used in this study are unclustered secondary indices, implying that accesses to the data pages through an index scan occur in a random (versus sequential) pattern. Key sizes are 12 bytes, and key/pointer pairs are 16 bytes. Table 1 lists the files and indices used for all of the experiments in this study. The large, medium, small, and tiny files are used by the transaction and batch classes (which are described in the next section). The query files consist of a set of 200 identical files and reside on a different disk than the transaction/batch files to limit any competition at the disk from the query class. There will be a small amount of disk interference between the query class and the other classes, however, because its hash join intermediate bucket files are written to randomly chosen disks. There are also two sets of the transaction/batch files, each on a separate disk, to eliminate the disk interference between transaction and batch classes or between multiple transaction classes.

4.3 Workload Model

Since we are primarily interested in the effects of page replacement decisions and working storage allocation on transaction performance, the key workload characteristics are page reference patterns and working storage requirements. Therefore, our simulated workload classes are relatively simple examples of variations in these two characteristics. For the purposes of this paper, we define a workload as any pair of the following classes: transactions, queries, or batch.

Transactions

The transaction workload class models page reference behaviors typical of transactions in the TPC-A benchmark [Gray 91]. They perform nonclustered, single record index selects on 4 files: big, medium, small, and tiny (see Table 1 above). Since all of our indices are 2 levels deep, this adds up to a total of 12 random page references per transaction. Although each file is accessed the same number of times per transaction, their differing sizes insures that some will have higher per-page access rates (i.e. higher temperatures) than others. Transactions require no working storage, and the key factor in their performance is their buffer hit rate.

For every experiment in the performance analysis section that includes transactions, we fix the number of terminals submitting transactions at a population of 100. Their think times are exponentially distributed with 15 second means. These two values were chosen such that average disk utilizations remain in the 50-60% range. The resulting transaction throughput is approximately 5 completions per second, and depending on the response times experienced, there are an average of 0.5 to 1.5 transactions resident in the system at any moment with peaks of 10-12. Enough memory is set aside to insure that at no point is a transaction forced to wait for memory, as we do not wish to address load control issues in this initial study.

The interval over which the average transaction response times are computed is set to 300 completions (about 60 seconds). This interval represents a balance point that allows the fragment fencing algorithm to provide a high degree of responsiveness while at the same time exhibiting very stable behavior with respect to changes in target residencies. As mentioned earlier, we will explore the effect of varying observation interval lengths in section 5.2.

Batch

The batch workload class consists of a single sequential scan of the medium data file. Obviously, the medium file has a fairly high temperature for this class. Because of this one hot file, the word "batch" is somewhat of a misnomer; while real batch workloads can normally be characterized by sequential scans, the files they reference are typically of a fairly low temperature. For this study, however, straight sequential scans of low temperature files are uninteresting because their buffer hit rates are near zero. This class is actually more of a stand-in for any type of workload that can be characterized by sequential accesses to a small portion of the database and very low working storage requirements. As before, because we wish to exclude load control issues from this study, we fix the number of terminals submitting batch queries to one, and we set its think time to zero.

The interval over which average batch response times is computed is set at 30 completions in length (about 60 seconds). The rationale for this interval is the same as that for the transactions: it represents a good balance point

between responsiveness and stability.

Queries

We model a query workload using binary relational join operators on two randomly chosen query files (see Table 1). Since we want to ignore any possible effect of query optimization decisions, the inner and outer join files are always of the same size here. We use the hybrid hash join algorithm [DeWitt 84] because it is generally accepted as a good ad hoc join method. Since the query files are nearly the same size as the configuration memory, allocating all of available memory to a join query will allow it to execute the with the minimum number of I/Os (a single scan of each relation). Allocating less memory (down to a minimum of 28 pages for these files) increases the number of I/Os required in a linear fashion. Since the queries choose their two join files from a set of 200, no single query file will have a very high access rate, and therefore the primary factor in their performance is the amount of working storage allocated to them as opposed to their buffer hit rates (which are essentially zero).

Since queries can demand and be allocated large portions of memory, the potential for more than one simultaneous query arrival would complicate our study of replacement policies with issues related to load control. Setting aside memory to avoid possible memory waits, as was done for the transaction class, is not feasible for queries since they can use such large amounts of memory. We therefore restrict the number of terminals that submit queries here to one at all times. We set the think time for this terminal to zero when studying steady state behavior (Section 5.1), because in this case it doesn't really matter if there are some points in time when a query is present or not – only average values are of interest. In our analysis of fragment fencing's transient behavior (Section 5.2), we investigate the effects of varying the query think time.

No average response time computation interval is needed for the query class. Since query performance cannot be affected by changes in disk buffer hit rates, we do not set any goals for them and we expect the DBMS to "do its best" for this class.

4.4 Parameter Summary

The important parameters of the simulated DBMS are listed in Table 2. The MIPs rating is typical of high-end workstations or mid-range computers and was chosen so that CPU utilizations could be kept below 10% in order to insure that the two workload classes primarily compete for memory, not CPU cycles. The number of terminals and think times were chosen to insure that disk utilizations lie in the 50 to 60% range. The memory size of 4 megabytes is obviously small, but was chosen to limit the amount of simulation time required for the performance studies. This does not limit the applicability of our performance analyses however, since the important factor is not the absolute size of memory but its size relative to the database and the working sets of concurrent transactions. The software parameters are based on instruction counts taken from the Gamma parallel database prototype [DeWitt 90]. The disk characteristics approximate those of the Fujitsu Model M2266 disk drive, as described earlier.

Parameter	Value	Parameter	Value
Transaction terminals	100	Mean transaction think time (exponential)	15 sec
Query terminals	1	Query think time	0 (varied)
Batch terminals	1	Batch think time	0
Number of CPUs	1	# instructions to read record off of buffer page	300
CPU speed	50 MIPS	# instructions to write record into buffer page	100
Number of disks	2	# instructions to insert record in hash table	100
Page size	8 KB	# instructions to probe hash table	200
Memory size	4 MB (512 pages)	# instructions to apply a predicate	100
Disk cylinder size	83 pages	# instructions to test an index entry	50
Disk seek factor	0.617	# instructions to copy 8K msg	10000
Disk rotation time	16.667 msec	# instructions to start an I/O	1000
Disk settle time	2.0 msec	# instructions to initiate select	20000
Disk transfer rate	3.09 MB/sec	# instructions to terminate select	5000
Disk cache context size	4 pages	# instructions to initiate join	40000
Disk cache size	8 contexts	# instructions to terminate join	10000

Table 2: Simulation parameter settings

5 Fragment Fencing Performance

In this section, we use the simulation model described previously to examine the both the steady state and the transient performance of fragment fencing. The steady state analysis addresses the basic question of how well fragment fencing can achieve response time goals for various workloads and system configurations. We explore four different pairings of the workload classes described in Section 4.3: transactions with queries, batch with queries, transactions with transactions, and transactions with batch. Half the cases specify goals for only one of the two classes, and the other half specify goals for both. Besides varying workloads and goals, we also explore the effects of different base replacement policies as well as varying levels of competition at the disk or CPU between the two classes. The transient analysis section explores the behavior of fragment fencing over time and addresses questions of stability and responsiveness that are always a concern for systems that exploit feedback. Holding the workload and configuration constant there, we explore two parameters: the length of the observation interval and the stability of the workload.

5.1 Steady State Behavior

The performance metric we adopt for judging steady state behavior is the average response time for each workload class. All of the experiments in this section execute the workload for 50 simulated minutes and collect statistics for only the final 30 minutes of simulated time in order to remove warm-up transients from the averages. We insure a minimum of 15,000 transaction completions, 500 batch job completions, and 50 query completions.

The results of each experiment in this section are presented in tables of a similar format, with a column for the average response time of each class. Every row represents a different response time goal. For comparison purposes, we include rows labeled "alone" that show the response time of each class when it is executed alone in the system, as well as rows labeled "base" that show results when no goal is specified for either class. The "alone" rows represent

lower bounds on the response times that can be expected for each class, and the "base" rows show how the base replacement policy acts without any assistance from the fragment fencing algorithm.

Additionally, if the query class is present, we add a column showing the amount of working storage allocated to the hash join under the guidance of fragment fencing. We also need to split the "base" row into two cases when queries are present, because without any guidance from the fragment fencing algorithm, the base allocation policy is free to decide on its own how much working storage to allocate to a hash join operator. We explore two cases: minimum, which is the minimum allocation required for the join to execute, and maximum, which is all of memory except for that portion which is set aside for "system" use and to insure that no transaction memory waits occur (20% of memory, or about 100 frames).

Transactions & Queries

We begin this section with a set of four experiments using a mix of transactions and queries. The detailed behavior and parameters of each workload class were described previously in Section 4.3. The first experiment isolates the effects of adding fragment fencing to an existing memory manager. Pure global LRU is used as the base memory manager to show the effects of fragment fencing as distinct from any other "hints" about the relative value of a page. We also insure that memory is the only resource where the two classes compete to any significant degree. This is accomplished by segregating the data referenced by each class onto separate disks and by setting the CPU speed such that processor utilizations are 10% or below (50 MIPS). Table 3 shows the resulting response times and memory allocations for this first experiment.

Examining the first two rows in Table 3, we see the impact of adding queries to a transaction workload: transaction response times double – even when those queries are allocated the absolute minimum amount of working storage. Since there is no significant contention at the disk or CPU in this experiment, the only reason for the change is a drop in transaction buffer hit rates when queries are added, resulting in an increase in the average I/Os per transaction from about 2 to about 4. This hit rate decrease is due to the inability of pure global LRU to distinguish the more frequently accessed transaction pages from the less valuable pages accessed by the queries.

The second and third rows of Table 3 show the effects of adding fragment fencing to a pure global LRU memory manager. While the 60 msec goal is not achievable for the transactions, their average response time of 71 msecs under fragment fencing approaches their stand-alone performance of 64.5 msecs. The reason is, of course, the increase in buffer hit rates provided by fragment fencing. In these same two rows we can see that query performance improves as well, even though the amount of memory allocated to the queries is the same (28 pages) with or without fragment fencing. This is because the transaction response times, with fragment fencing trying to enforce a 60 msec goal, are nearly halved relative to the pure LRU case. The response time improvement for the transactions lowers their average number in the system from 0.9 to 0.5, reducing what little competition the queries experience at the CPU and disk from the transactions (hash join buckets are written to a randomly chosen disk, which occasionally causes some interference at the disk between the two classes). Looking at the remaining rows, we can see that fragment fencing manages to meet the goals fairly well, with at most a 1% violation for the 80 and 150 msec cases.

The second experiment dealing with transactions and queries retains the workload and configuration of the

previous one. This time, however, the base memory manager uses a 3-level global LRU policy instead of pure LRU. Table 4 shows the results of this experiment. If we examine the first two rows of this table and compare them to the previous experiment (Table 3), we can see that adding hints on page type and reference patterns significantly reduces the impact of adding queries to a transaction workload relative to the pure LRU case. This is largely because sequential flooding[4] is eliminated via the hints. The second and third rows of Table 4 show that the additional guidance provided by fragment fencing still results in improved transaction response times. Even though the 3-level LRU base replacement policy can now distinguish the more valuable transaction data from the less valuable query data, it still does not discriminate between more frequently accessed and less frequently accessed data *within* the transaction class. The rest of Table 4 shows results similar to the previous experiment with pure LRU, except that all of the query response times are correspondingly lower. Looking at the query memory allocation column of Tables 3 and 4, we see that transaction response time goals under a 3-level LRU policy can be achieved with much less memory, leaving more left over to allocate to the queries. This shows clearly how fragment fencing's feedback approach allows it to adapt to the behavior of the base memory manager. The fencing algorithm doesn't know that response times are improved because the base replacement policy is smarter; it just knows that it has to keep less data in memory here to achieve the response time goals. For the rest of the paper, we will adopt a 3-level LRU replacement policy since it is clearly superior to pure LRU.

The final two experiments with a transaction/query workload examine the effects of increased competition between the two workload classes at resources other than memory: CPU and disk. Table 5 shows the effects of the increased disk competition that results from placing all the data of both classes on a single disk. Table 6 shows the effects of increased CPU competition by decreasing the MIP rating from 50 to 8. CPU utilizations rise from 10% or less in the previous experiments to between 50 and 75%.

Both tables 5 and 6 show a similar phenomenon. Response times rise uniformly relative to the more powerful system configuration used in Table 4, and the more aggressive response time goals in the top few rows become unachievable. Both of these effects are due to the increased competition between the two classes at the CPU or the disk (disk response times nearly double due to queueing delays). The amount of memory made available to the queries drops as well, indicating that the fencing algorithm is trying to compensate for the response time increases by retaining more and more of the database in memory. Similar to the pure LRU versus 3-level LRU case, the fragment fencing algorithm only knows that response times are higher for some reason, and the only thing it can do is to increase the memory resident portion of the database. Even though goal oriented CPU scheduling might be a more effective way to control response times in this case, we can see that fragment fencing still performs better for *both* classes than the base 3-level LRU strategy does by itself ("Base (min)" row).

Batch & Queries

The second workload that we examine is a combination of "batch" jobs, which consist of file scan operations, and hash join queries (see section 4.3 for detailed workload descriptions and parameters). Because the batch jobs access a "hot" file, fragment fencing can be effective in controlling their response times. Normally, a hint-based memory

[4] *Sequential flooding* is a problem characteristic of a pure LRU replacement policy. Processes performing sequential scans can flood the buffer pool with pages that are not likely to be reaccessed, displacing pages with a much higher probability of reaccess.

	Avg tran resp (msec)	Avg qry resp (sec)	Qry mem (pages)
Tran alone	64.5	-	-
Base (min)	138.4	33.7	28
Goal 60 ms	71.0	30.5	28
Goal 80 ms	80.8	25.3	100
Goal 100 ms	98.8	21.8	203
Goal 150 ms	151.6	16.6	350
Goal 200 ms	196.9	15.6	392
Base (max)	232.3	15.6	412
Qry alone	-	11.7	412

Table 3: Pure LRU, separate disks

	Avg tran resp (msec)	Avg qry resp (sec)	Qry mem (pages)
Tran alone	54.9	-	-
Base (min)	75.1	30.8	28
Goal 60 ms	60.8	30.0	28
Goal 80 ms	77.6	22.1	181
Goal 100 ms	101.2	19.2	273
Goal 150 ms	146.4	14.7	399
Goal 200 ms	182.5	14.8	411
Base (max)	182.1	14.5	412
Qry alone	-	11.7	412

Table 4: 3-level LRU, separate disks

	Avg tran resp (msec)	Avg qry resp (sec)	Qry mem (pages)
Tran alone	55.0	-	-
Base (min)	135.8	58.9	28
Goal 60 ms	112.8	53.2	28
Goal 80 ms	112.8	53.2	28
Goal 100 ms	112.8	53.2	28
Goal 150 ms	146.7	44.3	174
Goal 200 ms	201.0	43.2	280
Base (max)	333.2	43.5	412
Qry alone	-	11.7	412

Table 5: 3-level LRU, disk interference

	Avg tran resp (msec)	Avg qry resp (sec)	Qry mem (pages)
Tran alone	68.7	-	-
Base (min)	103.0	36.8	28
Goal 60 ms	91.0	36.1	28
Goal 80 ms	91.0	36.1	28
Goal 100 ms	100.1	29.2	116
Goal 150 ms	153.8	22.9	300
Goal 200 ms	190.0	19.8	395
Base (max)	230.3	19.9	412
Qry alone	-	15.7	412

Table 6: 3-level LRU, slow CPU

	Avg Batch resp (sec)	Avg qry resp (sec)	Qry mem (pages)
Batch alone	0.80	-	-
Base (min)	3.12	37.7	28
Goal 1.3 sec	1.42	37.7	28
Goal 1.6 sec	1.71	32.8	58
Goal 1.9 sec	1.96	28.4	140
Goal 2.5 sec	2.65	22.6	284
Goal 2.8 sec	2.86	22.5	300
Base (max)	3.27	17.0	412
Qry alone	-	11.7	412

Table 7: Batch & Queries

	T1 resp (msec)	T2 resp (msec)
T1 alone	55.0	-
T2 alone	-	55.0
T1 Goal 150 msec	131.1	60.6
T1 Goal 125 msec	126.3	60.6
T1 Goal 100 msec	102.2	60.1
T1 Goal 80 msec	91.4	64.8
Base	106.4	106.0

Table 8: Trans 1 & Trans 2 (60 msec goal)

manager would consistently penalize this batch workload because all of its data is accessed sequentially. On the other hand, a frequency-based memory manager would consistently favor a batch workload of this sort, because it understands that retaining frequently accessed data will increase overall hit rates. Table 7 shows the results of this experiment. If we look at the first two rows, we see a phenomenon that is identical to the first experiment (Table 3), where queries were added to a transaction workload under the pure LRU replacement strategy: response times for the batch class are more than doubled. In the first experiment, transaction response times doubled because the base replacement policy couldn't distinguish between the pages of each workload class, and buffer hit rates for the transactions thus dropped significantly. By adding hints on page reference patterns to the replacement policy, the problem with the transaction/query workload was fixed. However, those same hints are useless for the batch/query workload in this experiment because both classes access the same type of data with the same reference pattern (sequential data file scans).

Looking at the average response times for the batch workload in Table 7, we see that fragment fencing can achieve the goals reasonably well for this workload, with at most a 6% violation in the 2.5 second case.

Transactions & Transactions

The third workload that we examine here is a combination of two TPC-A-like transaction classes, each with their own response time goals. The behavior of both workload classes is identical to that which was described in Section 4.3, except that each references an identically sized but distinct set of files to eliminate data sharing effects. The data used by the two transaction classes is segregated on separate disks as well, so there is no competition at the disk between the two classes.

This experiment investigates the behavior of fragment fencing when the goals for both classes can be achieved, and also when they cannot. Table 8 shows the response times that result. Class T2's goals are fixed at 60 msecs, which is very close to the lower bound of 55 msecs. Class T1's goals are progressively tightened until they become impossible to achieve, which occurs at the 80 msec goal. Because the behavior of each class in this workload is very similar, goals for both classes are violated when either one cannot be achieved. Class T2's performance suffers a bit more relative to class T1, however. It turns out that this is purely a matter of chance. Since fragment fencing does not have a notion of priority between classes, the first class to violate its goals will win the race for any remaining memory; which class violates its goal first simply depends on the random arrival processes of each class.

Another interesting result of this experiment is the behavior of fragment fencing with extremely tight goals (the 80/60 msec case) relative to the performance of the base 3-level LRU replacement policy. Although the base replacement policy gives each class the same performance, it is significantly worse (see the "Base" row of the table) than when fragment fencing is activated. The reason is the same as in the second experiment (Table 4): the base replacement policy has no information about the relative frequencies of reference among pages with the same "hint" level. The situation becomes even worse with two classes because of the interference between them (external thrashing). Because fragment fencing tracks the frequency of reference to each fragment, it can guide the base replacement policy into making more intelligent replacement decisions.

	B resp (sec)	T resp (msec)
Tran alone	-	55.0
Goal 2.70	2.70	60.8
Goal 2.30	2.40	82.3
Goal 2.00	2.06	115.9
Goal 1.70	1.76	140.3
Goal 1.40	1.40	162.2
Batch alone	0.80	-
Base	2.70	60.8

Table 9: Trans (150 ms) & Batch

	B resp (sec)	T resp (msec)
Tran alone	-	55.0
Goal 2.70	2.70	60.8
Goal 2.30	2.40	82.3
Goal 2.00	2.06	99.7
Goal 1.70	1.75	107.0
Goal 1.40	1.75	107.0
Batch alone	0.80	-
Base	2.70	60.8

Table 10: Trans (100 ms) & Batch

	B resp (sec)	T resp (msec)
Tran alone	-	55.0
Goal 2.70	2.70	60.8
Goal 2.30	2.39	79.9
Goal 2.00	2.06	85.9
Goal 1.70	2.07	85.8
Goal 1.40	2.07	85.8
Batch alone	0.80	-
Base	2.70	60.8

Table 11: Trans (80 ms) & Batch

Transactions & Batch

The final workload that we investigate in this section is a combination of TPC-A-like transactions and "batch" jobs that consist of scans over "hot" files. Tables 9, 10, and 11 fix the transaction class goals at 150, 100 and 80 msecs respectively. Transaction and batch response times in these tables are shown under the "T resp" and "B resp" columns respectively. For each of the three transaction class goals, the batch class goals are varied from loose to tight. Table 9 shows the response times that result with the loosest transaction class goal (150 msecs). In all cases, the batch class goals are met, and in all except the last row, the transactions out perform their 150 msec target. This is because the base replacement policy is favoring the transaction class's pages over the batch class's pages, allowing the transaction class to use all of the memory that is not required to meet the batch response time goals. Tables 10 and 11 show the same experiment with the transaction class goals set at 100 and 80 msecs, respectively. In the 100 msec transaction case, batch class goals are unattainable beyond 1.75 seconds. Under the tightest transaction goals of 80 msecs (Table 11), the batch goals are unattainable beyond 2 seconds.

An interesting aspect of this workload is how fragment fencing can modify the base replacement policy's treatment of each class: always favoring transactions over batch. With fragment fencing, the base replacement policy can be "coerced" to favor the transactions less and less, allowing the batch class to move closer to its stand-alone response time.

We conclude our examination of steady state performance by noting that fragment fencing seems capable of successfully achieving steady state response time goals for these example workloads, and that in many cases it can provide better performance for the classes that do not specify any goal as well (relative to the base buffer manager's stand-alone performance). We have also seen that fragment fencing is able to adjust to different degrees of intelligence in the base buffer manager, and to high device utilizations which violate its simplistic model of transaction behavior. Thus, fragment fencing appears quite promising as a mechanism to provide users or system administrators with the ability to automatically tune a DBMS according to a set of application-level performance requirements.

5.2 Transient Behavior

There are many possible ways to satisfy an average performance metric over some specified time interval. For example, a one second average response time goal over some interval could be satisfied such that 80% of the transactions in the interval experience a quarter second response time, while 20% of the transactions experience 4 second response times. Since only the average value of the metric and the interval over which it should be computed were specified, we cannot say if this particular way of satisfying the goal is good or bad. Most likely a more complete performance specification would include more information on the distribution of response times that are considered "good," perhaps by specifying standard deviations, percentiles, or maximums.

Even though fragment fencing currently lacks mechanisms to specify or act on a more detailed specification of response time goals, we *can* state a simple requirement for its transient behavior in any case: it should not introduce more variance in the workload than would exist if fragment fencing were not activated. Since the only way fragment fencing can introduce variance is to change the size of the resident database volume, we need to see if there is excessive movement of the line separating the resident volume from the non-resident volume. We explore two variables which could cause the algorithm to adjust the resident volume excessively: the length of the interval used to compute average response times, and the length of the think time between arrivals of resource intensive queries. Both experiments use the same workload as in the first steady state experiment: TPC-A-like transactions mixed with hash join queries (see Section 4.3 for detailed workload descriptions).

The length of the observation interval can cause excessive movement of the resident volume line for basic statistical reasons. When the interval is shorter, fewer transactions are used to compute the average observed response times at each interval completion. As in any statistical sample of a large population, the smaller the sample size, the larger the variance that will be observed between each sample. Small observation intervals can therefore present a picture of a very unstable system to the fragment fencing algorithm. The challenge is not to over-react and try to manage what are purely statistical fluctuations in the system load. Larger observation intervals help to mask these natural fluctuations in load, and thus provide a much more stable input to the algorithm. In this case, the algorithm will be less likely to attempt to over-manage the system.

To give an idea of the input that the fragment fencing algorithm is attempting to deal with, Figure 2 shows a graph of the average transaction response times over intervals of 50 completions with fragment fencing turned off. The X axis is a count of transaction completions, and the Y axis is the average response time over each 50-transaction interval. The upper line in the graphs shows the behavior of transactions when the queries are allocated their maximum amount of working storage, leaving very little for the transaction class data. The lower line shows the results of a minimum memory allocation to the queries, with most of memory being allocated to transaction data. Note that even this picture shows less variance than is actually occurring in the system, since it represents averages over 50 transaction completions. We use an interval length of 50 completions as our lower bound since any smaller intervals start to lose statistical significance.[5] One phenomenon that can be seen in this graph is the relationship between the amount of memory available to the transactions and the resulting variance in their response

[5] While the actual number of samples required by any statistical analysis depends on the amount of error that can be tolerated, sample sizes less than 30 or 40 are normally considered "small."

times. The more memory, the lower the probability of disk I/Os, and the lower the variance becomes. An interesting implication of this phenomenon for fragment fencing is that as the resident volume increases because of tighter and tighter response time goals, the variance decreases. Surprisingly, this means that loose response time goals are actually more difficult to manage than tight ones are.

We show the effects on resident volume of varying the observation interval lengths in Figures 3, 4, and 5 for interval lengths of 50, 100, and 300 completions, respectively. The X axis of these graphs shows transaction completion counts (time), and the Y axis shows the resident volume in pages. The maximum resident volume allowed is about 400 pages (80% of the memory in the configuration). Each line in the graphs represents a different goal for the transaction class. Higher lines (larger resident volumes) correspond to tighter response time goals, and lower lines correspond to looser goals. The throughput of the transactions in this experiment is approximately 5 per second, so the intervals of 50, 100, and 300 completions translate to 10, 20, and 60 seconds. While it is difficult to develop a precise metric to gauge the relative "goodness" of each of these graphs, they show how the stability of the algorithm improves as the observation interval lengthens. Even though there are more fluctuations with smaller intervals, the algorithm seems firmly anchored around a central point in each case. We also experimented with intervals greater than 300, but the results were essentially identical to the 300 case and we therefore omit them here.

The implication of this analysis is that for a workload class with sufficiently high throughput (greater than 5 per second), an observation interval of around one minute or larger provides very stable performance. For workload classes with lower throughput, however, there is going to be a larger trade-off between stability and responsiveness. While lower throughput workloads (e.g. batch jobs) seem likely to experience much lower natural variance in response times, and can therefore perhaps deal with a smaller observation interval, the proper setting of this parameter for low throughput workloads remains an area for further investigation.

The second variable that we investigate here is the gap between query arrivals. We explore deterministic query class think times ranging from 10 to 120 seconds for a transaction workload with a response time goal of 70 milliseconds. The transactions have a throughput of approximately 5 per second, so the number of transaction completions that could occur during the gap between query arrivals varies from 50 to 1500. The interval over which we compute the average response times is 100 completions (20 seconds at 5 transactions per second). This interval size is smaller than that recommended by the previous analysis, but it allows us to exaggerate the effects of query think time slightly by increasing the responsiveness of the fencing algorithm. If we look at query think time simply as another way to introduce variance in the system load, then obviously a large enough observation interval could cancel the effects of any think time-related variance as well. For the purposes of this experiment, however, we want to limit the dampening effect of a longer observation interval (even though it is a perfectly valid way to address the problem).

Figure 6 shows the size of the resident volume as a function of time for each of four query think time values. The two straight lines at the top of the graph are the 10 and 30 second think time results. Since these think times are similar in length to the observation interval, their effects are completely dampened by the averaging that occurs over the observation interval, as explained in the previous analysis. The resident volume for the 30 second think time line is lower than the 10 second think time line because as the query think time increases, there are more periods where the transactions do not have to compete for memory and thus their response times improve as a result. The fencing

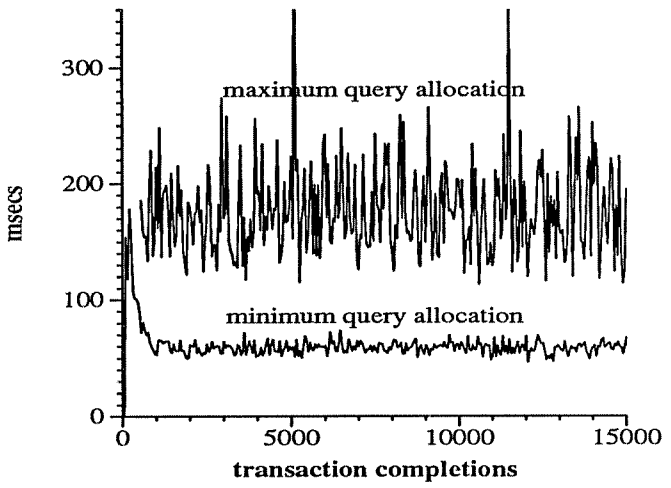

Figure 2: Tran resp times: 50 completion interval

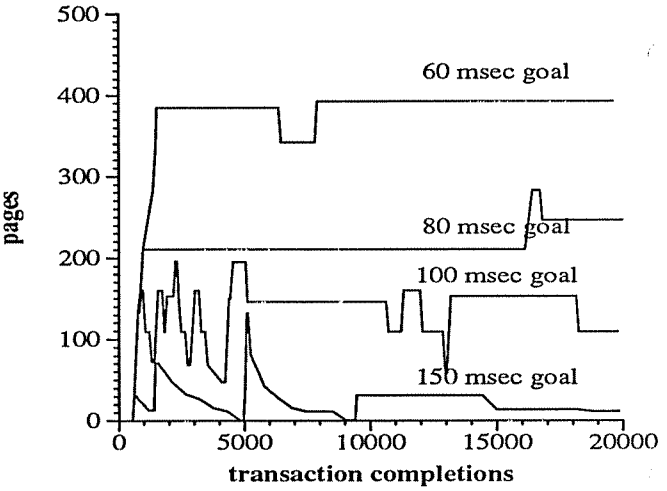

Figure 3: Res volume, 50 compl intervals (10 sec)

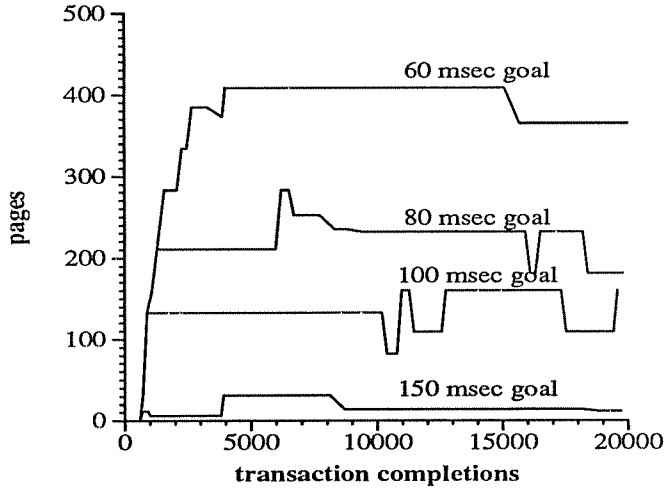

Figure 4: Res volume, 100 compl intervals (20 sec)

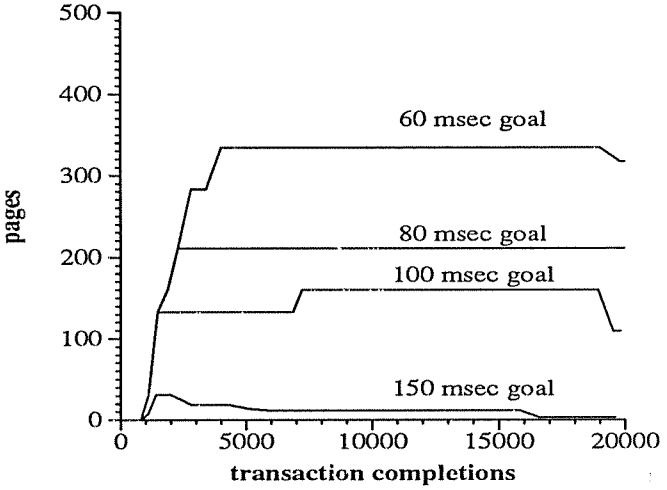

Figure 5: Res volume, 300 compl intervals (60 sec)

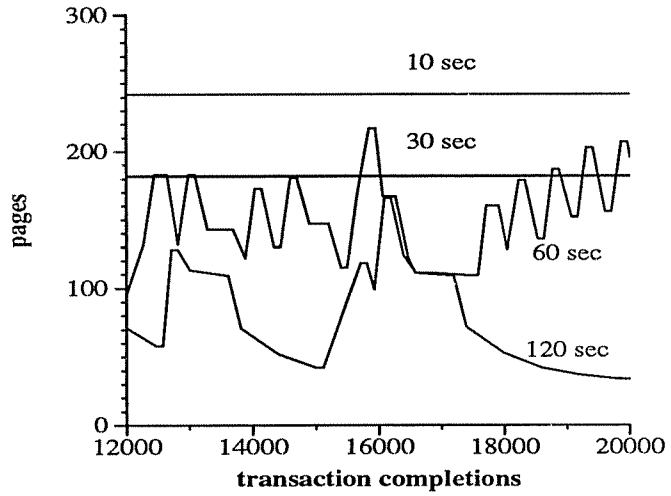

Figure 6: Resident volume: various think times

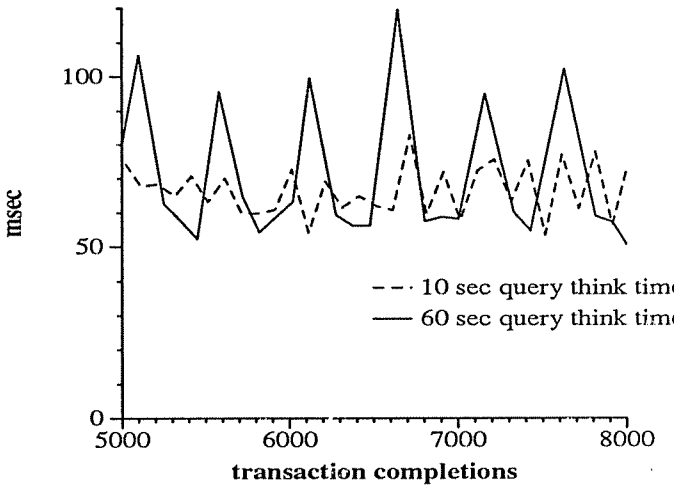

Figure 7: Tran resp times: various think times

algorithm reacts to this by reducing the resident volume required to maintain the 70 m sec transaction response time goal.

The next two (wobbly) lines in Figure 6 show 60 and 120 second query think times, in order of decreasing average resident volume. The 60 second think time is just large enough for three observation intervals to occur during the gap in between queries. These three observations are enough to convince the fencing algorithm to reduce the resident volume required, only to raise it again in the following interval. The 120 second think time line is similar, but the time span between lowering and raising the resident volume becomes larger. The 60 second think time represents a worst case scenario for this experiment (with its 20 second observation interval length), as there is no benefit there to lowering the resident volume – it will have to be raised again almost immediately.

Figure 7 shows the effect of excessive movement of the resident volume line on transaction response times. The line with higher variance corresponds to the worst case 60 second query think time, and the line with the lower variance corresponds to a more favorable scenario involving a 10 second query think time. The 10 second think time represents a favorable case because the resident volume line is never moved here, and any variance in response times is thus due to natural statistical fluctuations in the transaction workload itself. Clearly, lowering the resident volume in the 60 second think time case is a bad idea; the additional variance introduced could even cause the average response time goals to be jeopardized. We plan on investigating this issue further in our future work.

6 Issues and Extensions

In this section, we briefly discuss some important remaining challenges for the fragment fencing approach and our current thoughts on how we plan to address them. One key challenge is to address violations of the algorithm's assumption that the hit rate for a fragment is equal to the fraction of the fragment that is memory resident. Violations of this assumption could be caused either by non-uniform reference patterns (e.g. temporal locality, correlated references, append-only access, etc.) or by deficiencies in the base replacement policy (e.g. LRU for a loop that cannot fit in memory). Such violations currently can cause fragment fencing to continue increasing the target residency of a fragment even when little or no hit rate increase results from doing so. However, the information needed to detect a violation of this assumption is already collected by the algorithm (i.e., it keeps the % residency and the observed hit rate for each fragment), so it should not be too difficult to improve the algorithm in this regard.

Another area for improvement is to address violations of the algorithm's assumption that a transaction's response time is linearly related to the number of I/Os that it requires. As we saw in the steady state performance analysis (Tables 5 and 6), violations of this assumption translate into a larger resident volume being required to achieve a given goal. In fact, fragment fencing may try *too hard* to achieve a goal when this assumption is violated, increasing the resident volume by larger and larger amounts in order to achieve only small improvements in response times. As for the hit rate assumption above, the algorithm should be modified to check the validity of this I/O dominance assumption before acting on it. This can be accomplished by monitoring the average observed disk response time per class; by multiplying this quantity by the average number of disk I/Os for a class, the algorithm can identify classes for which I/O time is a relatively small component of the overall average response time.

Still another challenge lies in addressing potential problems caused by low temperature fragments, as these may also cause the fragment fencing algorithm to increase the target residency of a fragment by large amount for only a small return. If a workload class performs a large number of I/Os, but on very "cold" data, then even filling up all of available memory with the class's data would not significantly reduce the number of I/Os required by transactions of the class. An example of this type of behavior would be a batch job that sequentially scanned a very large database. An obvious approach to addressing this issue is to check for some minimum temperature before increasing the target residency. The algorithm already determines whether a single fragment can completely satisfy any required change in I/O, or if multiple fragments are required. This decision can easily be extended to determine if *any* set of fragments referenced by the class can satisfy the required change in I/O.

Finally, as seen in the transient performance analysis (Figure 6), long-running classes with large working storage requirements (such as hash joins) can present special challenges with respect to the transient behavior of fragment fencing. Once fragment fencing gives away some working storage to a long-running hash join, it can suffer the consequences of that decision for long time to come. The situation would be exacerbated further if the relative response times of such queries are many orders of magnitude larger than those of the competing goal classes. (Our performance analysis only considered response time ratios of up to 100 or so between classes). While it is unlikely that fragment fencing can ever be prevented from making mistakes, there are certainly ways to limit the penalty of doing so. One promising possibility is the exploitation of memory-adaptive query processing algorithms, e.g. memory adaptive hash join and sorting methods [Zeller 90, Pang 93a, Pang 93b]. These join methods can dynamically adapt to changes in the amount of available working storage during execution, so fragment fencing could actually "take back" some of the working storage from long running queries when it is necessary increase the resident volume while such queries are active.

In summary, the primary pathology of fragment fencing is the possibility of its attempting large increases in the resident volume in return for small improvements in I/Os or response times for certain classes. By modifying the algorithm to first check its assumptions, and to react to violations that it detects, it is likely that such problematic behavior can be avoided. In addition, memory-adaptive schemes appear promising as a way to address the problem of long-running consumers of working storage.

7 Conclusions and Future Work

In this paper we have explored the potential of using memory allocation and page replacement mechanisms to implement per-class performance goals for multiclass workloads. We described an algorithm called *fragment fencing* that takes as input a set of per-class response time goals and a description of the data and index fragments that make up the database. The algorithm that we described observes the per-class reference frequencies and monitors the state of the system relative to its stated goals; the information that it gathers is used to help existing buffer allocation and page replacement mechanisms to avoid making decisions that may violate the goals.

Using a detailed simulation model, we studied both the steady state and transient performance of fragment fencing when it is coupled with a modified global LRU memory manager with three levels of "hints." Our results

showed fragment fencing to be capable of successfully achieving steady state response time goals for a number of example multiclass workloads. For workloads where one of the classes did not specify any goals, fragment fencing usually provided better performance than the base buffer manager alone for the non-goal class as well. Moreover, by coupling fragment fencing with a pure global LRU replacement mechanism, we demonstrated that the approach is able to coexist with base buffer managers with varying degrees of intelligence. Fragment fencing was able to achieve the same goals with an LRU scheme as it did with the more intelligent 3-level LRU scheme, although at a higher cost in terms of the amount of memory dedicated to fragment caching. Finally, we explored violations of fragment fencing's simple assumptions regarding transaction behavior as well as possible enhancements to limit the impact of these violations. We conclude that fragment fencing appears quite promising as a way to provide users or system administrators with the ability to tune a DBMS according to a set of application-level performance requirements.

Besides the extensions listed in the previous section, our future work will explore additional mechanisms for dealing with conflicting goals between classes, for allowing more detailed specifications of response time goals (such as maximums and percentiles), and for limiting the penalty incurred as a result of workload shifts (via persistent statistics). We also plan on coupling fragment fencing with algorithms that handle load control and working storage allocation among competing queries in order to explore the performance of multiple concurrent queries competing with transactions and batch classes [Mehta 93], and we plan on integrating fragment fencing with goal-oriented CPU and disk scheduling mechanisms as well. The information collected by the algorithm on hit rates and percent residencies for individual fragments could also be a useful input to recently proposed techniques for run-time selection of query plans [Hong 91, Ioann 92]. Finally, we would like to exploit the capabilities of memory-adaptive query processing techniques, e.g., preemptible hash join and sorting methods [Pang 93a, Pang 93b].

Acknowledgements

The authors would like to thank Manish Mehta, Mike Franklin, Hwee-Hwa Pang, and Joe Hellerstein for many helpful discussions and comments on previous versions of this paper.

References

[Boral 90] H. Boral et al, "Prototyping Bubba: A Highly Parallel Database System," *IEEE Trans. on Knowledge and Data Engineering*, 2(1), March 1990.

[Brown 92] K. Brown, M. Carey, D. Dewitt, M. Mehta, J. Naughton, "Resource Allocation and Scheduling for Mixed Database Workloads," Computer Sciences Technical Report #1095, Department of Computer Sciences, University of Wisconsin, Madison, July 1992 (available via anonymous ftp from `ftp.cs.wisc.edu`).

[Cheng 84] J. Cheng et al, "IBM Database 2 Performance: Design, Implementation, and Tuning," *IBM Systems Journal*, 23(2), 1984.

[Chou 85] H. Chou and D. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proc. 11th Int'l VLDB Conf.*, Stockholm, Sweden, Aug. 1985.

[Coffman 73] E. Coffman and P. Denning, *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs NJ, 1973.

[Copeland 88] G. Copeland, W. Alexander, E. Boughter, T. Keller, "Data Placement in Bubba," *Proc. ACM SIGMOD '88 Conf.*, Chicago, IL, June 1988.

[Cornell 89] D. Cornell and P. Yu, "Integration of Buffer Management and Query Optimization in a Relational Database Environment," *Proc. 15th Int'l VLDB Conf.*, Amsterdam, The Netherlands, Aug, 1989.

[DeWitt 84] D. DeWitt et al, "Implementation Techniques for Main Memory Database Systems," *Proc. ACM SIGMOD Conf.*, Boston, MA, June 1984.

[DeWitt 90] D. DeWitt et al, "The Gamma Database Machine Project," *IEEE Trans. on Knowledge and Data Engineering*, 2(1), March 1990.

[DeWitt 92] D. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Processing," *CACM*, 35(6), June, 1992.

[Falou 91] C. Faloutsos, R. Ng, T. Sellis, "Predictive Load Control for Flexible Buffer Allocation," *Proc. 17th Int'l VLDB Conf.*, Barcelona, Spain, Sept. 1991.

[Graefe 89] G. Graefe and K. Ward, "Dynamic Query Evaluation Plans," *Proc. ACM SIGMOD '89 Conf.*, Portland, OR, May 1989.

[Gray 87] J. Gray and F. Putzolu, "The 5 Minute Rule for Trading Memory for Disk Access and the 10 Byte Rule for Trading Memory for CPU Time," *Proc. ACM SIGMOD '87 Conf.*, San Francisco, CA, 1987.

[Gray 91] J. Gray ed., *The Benchmark Handbook*, Morgan Kaufmann, San Mateo CA, 1991.

[Haas 90] L. Haas et al, "Starburst Mid-Flight: As the Dust Clears," *IEEE Trans. on Knowledge and Data Eng.*, 2(1), March 1990.

[Haas 91] P. Haas, A. Swami, "Sequential Sampling Procedures for Query Size Estimation," *Proc. ACM SIGMOD '92 Conf.*, San Diego, CA, June 1992.

[Hong 91] W. Hong and M. Stonebraker, "Optimization of Parallel Query Execution Plans in XPRS," *Proc. 1st Int'l PDIS Conf.*, Miami, FL, Dec. 1991.

[Ioann 92] Y. Ioannidis, R. Ng, K. Shim, T. Sellis, "Parametric Query Optimization," *Proc. 18th Int'l VLDB Conf.*, Vancouver, B.C., Aug. 1992.

[O'Neil 93] E. O'Neil, P. O'Neil, G. Weikum, "The LRU-K Page Replacement Algorithm For Database Disk Buffering," to appear *Proc. ACM SIGMOD '93 Conf.*, Washington D.C., May 1993.

[Mehta 93] M. Mehta and D. DeWitt, "Dynamic Memory Allocation for Multiple-Query Workloads," to appear *Proc. 19 Int'l VLDB Conf.*, Dublin, Ireland, Aug 1993.

[Ng 91] R. Ng, C. Faloutsos, T. Sellis, "Flexible Buffer Allocation Based on Marginal Gains," *Proc. ACM SIGMOD '91 Conf.*, Denver, CO, May 1991.

[Pang 93a] H. Pang, M. Carey, M. Livny, "Partially Preemptible Hash Joins," to appear *Proc. ACM SIGMOD '93 Conf.*, Washington D.C., May 1993.

[Pang 93b] H. Pang, M. Carey, M. Livny, "Memory Adaptive External Sorts and Sort-Merge Joins," to appear *Proc. 19 Int'l VLDB Conf.*, Dublin, Ireland, Aug 1993.

[Pirahesh 90] H. Pirahesh, et al, "Parallelism in Relational Database Systems: Architectural Issues and Design Approaches," *IEEE 2nd Int'l Symposium on Databases in Parallel and Distributed Systems*, Dublin, Ireland, July 1990.

[Reiter 76] A. Reiter, "A Study of Buffer Management Policies For Data Management Systems," MRC Technical Summary Report #1619, Mathematics Research Center, University of Wisconsin, Madison, March 1976.

[Robinson 90] J. Robinson and M. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," *Proc. SIGMETRICS '90 Conf.*, Boulder, CO, May 1990.

[Sacco 86] G. Sacco and M. Schkolnick, "Buffer Management in Relational Database Systems," *ACM TODS*, 11(4), December 1986.

[Teng 84] J. Teng and R. Gumaer, "Managing IBM Database 2 Buffers to Maximize Performance," *IBM Systems Journal*, 23(2), 1984.

[Yu 93] P. Yu and D. Cornell, "Buffer Management Based on Return on Consumption in a Multi-Query Environment," *VLDB Journal*, 2(1), Jan 1993.

[Zeller 90] H. Zeller, J. Gray, "An Adaptive Hash Join Algorithm for Multiuser Environments" *Proc. 16th Int'l VLDB Conf.*, Melbourne, Australia, Aug. 1990.