Goal-Oriented Buffer Management Revisited*

Kurt P. Brown   Michael J. Carey   Miron Livny
{*brown, carey, miron*}@*cs.wisc.edu*

Computer Sciences Department, University of Wisconsin, Madison, WI

Abstract

In this paper we revisit the problem of achieving multi-class workload response time goals by automatically adjusting the buffer memory allocations of each workload class. We discuss the virtues and limitations of previous work with respect to a set of criteria we lay out for judging the success of any goal-oriented resource allocation algorithm. We then introduce the concept of *hit rate concavity* and develop a new goal-oriented buffer allocation algorithm, called *Class Fencing*, that is based on this concept. Exploiting the notion of hit rate concavity results in an algorithm that not only is as accurate and stable as our previous work, but also more responsive, more robust, and simpler to implement.

1 Introduction

In a multiclass database workload, each class exhibits different resource consumption patterns, and each may have its own performance goal. For example, a three-class workload might consist of TPC-A-like transactions, critical decision support queries, and non-critical data mining queries. The performance goals for this workload might specify an average response time of one second for the transactions, one minute for the decision support queries, and no specific goal for the data mining queries (i.e. "best effort"). Today, these goals would be achieved by manually tuning various low level "knobs" provided by the DBMS, possibly including buffer pool sizes, multiprogramming levels, data placement, dispatching priorities, prefetch block sizes, commit group sizes, etc. Ideally, the DBMS should accept per-class performance goals as inputs, and it should adjust its own low-level knobs in order to achieve them.

1.1 Goal-Oriented Basics

Among the many knobs that can be used to control response times in a DBMS, memory allocation is perhaps the most important because it determines the amount of disk bandwidth consumed. If we assume that all other knobs remain fixed, the problem of goal-oriented memory allocation can be stated as follows: For each class with an average response time goal, a memory allocation must be found such that its observed response time is as close as possible to its goal, while at the same time maximizing the amount of memory available for any no-goal class (we assume that all classes without performance goals are combined into a single no-goal class). In this paper we will restrict ourselves to the allocation of *disk buffer* memory, i.e. memory that is used to store copies of disk pages, as opposed to other types of *working storage* memory (e.g. sort work areas, join hash tables, etc.).

*This work was partially supported by the IBM Corporation through a Research Initiation Grant.

Because of the complexities inherent in real-world database management systems and workloads, accurately predicting the disk buffer allocation required to achieve a particular response time is extremely difficult. Therefore, the general approach common to all goal-oriented resource allocation work [Pierce 83, Ferg 93, Brown 93, Brown 94, Chung 94] is the notion of feedback coupled with "best guess" estimation. The idea is to observe the actual response times of a class relative to its response time goal, and to use the difference between the two as an input to an estimator that will adjust resource allocation knobs in order to move the class closer to its goals. This process of observing, estimating, and adjusting knobs is repeated continuously at regular intervals. The length of these intervals can be expressed as a predefined number of transaction completions, and should be chosen to strike a good balance between responsiveness and statistical stability [Brown 93].

1.2 Criteria for Success

How successfully a class meets its response time goal is not the only criteria with which to judge a goal-oriented resource allocation algorithm. In our view, the following criteria must be satisfied by any goal-oriented resource allocation algorithm before it can be seriously considered for an implementation in an industrial-strength DBMS.

Accuracy: The observed average response times for goal classes should be close to their goals. A convenient way to quantify accuracy is the *performance index* [Nikolaou 92], which is simply the average observed response time divided by the average response time goal. A performance index of one is ideal, greater than one indicates a goal violation, and less than one indicates an exceeded goal. Other metrics are needed to deal with different forms of goals (e.g. 90th percentile response time limits), but in this paper we will restrict ourselves to average response time goals.

Responsiveness: The number of knob adjustments (intervals) it takes to bring a class to its goal should be as small as possible, especially if the interval between each knob adjustment is relatively long (multiple minutes). A responsiveness criteria rules out simplistic exhaustive search strategies that can score high in accuracy, but may require *lots* of time to search for the solution.

Stability: The variance in the response times of goal classes should not increase significantly relative to a system without goal-oriented allocation mechanisms. Thus, for a stable workload, once the goals are achieved, all knobs should be left alone.

Overhead: A goal-oriented memory manager should minimize the extent to which it reduces overall system efficiency. Overhead can be tested by taking the observed class response times for a particular workload running on a non-goal-oriented system and using them as goals for the same workload running on a goal-oriented system. One of the classes can be chosen arbitrarily as a no-goal class; any response time degradation in this class will then indicate the reduction in system capacity (assuming the goals for the other classes can be met).

Robustness: The system should handle as wide a range of workloads as possible, while avoiding any knob adjustments for a class that cannot be controlled by the given knob. For example, if a class is dominated by large file scans and the DBMS has an effective prefetching strategy, then the response time for such a class will not be controllable via the buffer allocation knob because the prefetcher will guarantee a very high hit rate with very little memory. As another example, any increase in the multiprogramming level knob for a class that only rarely queues for admission into the DBMS is not likely to affect the response time for the class either.

Practicality: The algorithm should not make any unrealistic assumptions about the workload or the DBMS in general. For example, it would be unreasonable to assume that all workloads are static and therefore amenable to off-line analysis. Likewise, the algorithm should not place too many restrictions on the behavior of the basic resource allocation mechanisms of the DBMS and/or OS, or assume it has full control over all aspects of those mechanisms.

It should be noted that these criteria will normally be in conflict (stability versus responsiveness, responsiveness versus overhead, etc.), and therefore a goal-oriented resource allocation algorithm necessarily represents a careful balance between them.

1.3 Our Work

In earlier work, we described a goal-oriented buffer management algorithm called *Fragment Fencing* [Brown 93]. Encouraged by our initial simulation studies, we built a prototype version of Fragment Fencing in DB2/6000, IBM's commercial relational database for Unix [IBM 93b]. We then experimented with our simulated workloads and some additional workloads, including the TPC-B, TPC-C, and multi-user TPC-D benchmarks [TPC 94]. Our prototype performed as expected for the types of workloads that we had used in our simulation studies, and accurately held classes to their goals in a stable manner.

While our initial experiments were encouraging, our experiments with the TPC workloads uncovered two problems with Fragment Fencing. The first problem was the fact that our prototype exhibited certain unanticipated overheads when coupled with DB2/6000's page replacement policy. The second problem was the difficulty of extending Fragment Fencing to handle a wider range of workloads than it had been designed for. Specifically, our experience with a wider range of workloads made us realize that lifting one of Fragment Fencing's simplifying assumptions (uniform access within database fragments) would require a different, and more general, approach to predicting buffer hit rates. This paper describes a new algorithm, called *Class Fencing*, that features a hit rate predictor based on a notion that we call *hit rate concavity*. Exploiting the notion of hit rate concavity results in an algorithm that not only is as accurate and stable as Fragment Fencing, but also more responsive, more robust, and simpler to implement. In addition, Class Fencing's memory allocation mechanism eliminates the overhead problem that we discovered in our Fragment Fencing prototype.

In the next section, we review the Fragment Fencing algorithm in more detail and discuss some of the insight that we gained from implementing it in an industrial-strength DBMS. We also discuss the only other goal-oriented memory management algorithm of which we are aware, *Dynamic Tuning* [Chung 94]. We then explain the notion of hit rate concavity and describe the Class Fencing algorithm in Section 3. In Section 4, we describe a detailed simulation model that is used in Section 5 to analyze the performance of Class Fencing. Finally, we summarize our findings and discuss our planned future work in Section 6.

2 Previous Approaches

Goal-oriented buffer allocation algorithms can be described abstractly in terms of three components: a *response time estimator* that estimates response time as a function of buffer hit rate, a *hit rate estimator* that estimates buffer hit rate as a function of memory allocation, and a *buffer allocation mechanism* that is used to divide up memory between the competing workload classes. The basic idea behind existing goal-oriented buffer allocation algorithms is to first use the response time estimator (in the inverse) to determine a target buffer hit rate that can achieve the response time goal. Next, the hit rate estimator is used (in the inverse) to determine a buffer allocation that can achieve this target hit rate. Finally, the buffer allocation mechanism is used to give each class its required memory. These steps are repeated continuously for each class in the hope that each successive estimate will bring the classes closer to their response time goals. This abstract framework will be used in the remainder of this section to describe the Dynamic Tuning and Fragment Fencing algorithms, and will be used again in Section 3 to explain the Class Fencing algorithm.

2.1 Dynamic Tuning Description

The Dynamic Tuning algorithm [Chung 94] differs from other goal-oriented algorithms ([Ferg 93, Brown 93, Brown 94]) in one important respect: response time goals are specified with respect to low-level buffer management requests (i.e., in terms of target service times for individual get/read page requests) as opposed to overall transaction response times. Dynamic Tuning's low level of goal specification allows it to use the following simple linear estimate to predict buffer request response times:

$$R^{est} = (1.0 - HIT^{est}(M)) \times D$$

$HIT^{est}(M)$ is the estimated hit rate for the class that will result from a memory allocation M, and D is the average time required for moving a page from disk to memory.

To estimate hit rate as a function of memory, the Dynamic Tuning algorithm adopts observations from Belady's virtual memory study [Belady 66], modeling the hit rate function as $1 - a/M^b$, where M is the memory allocation and the constants a and b are specific to a particular combination of workload and buffer page replacement policy. To compute a and b, Dynamic Tuning observes the hit rates that result from the two most recent memory allocations, plugs these observations into the model, and solves the two simultaneous equations that result. With a specific a and b in hand, Dynamic Tuning can use the inverse of the Belady equation to estimate the memory required to achieve a particular target hit rate.

Once a target memory allocation for a class is determined, Dynamic Tuning uses this allocation as the size of a buffer pool partition that is dedicated to the class. The entire buffer pool is essentially partitioned into separate pools for each class that are managed by completely autonomous buffer managers. The size of each pool is allowed to vary dynamically in response to changing system loads.

2.2 Dynamic Tuning Issues

While the Belady equation used by Dynamic Tuning's hit rate estimator is a good approximation to the general shape of most hit rate functions, it is not always a good "fit" for any particular function. To illustrate how well the curve-fitting approach used by Dynamic Tuning's hit rate estimator works with an actual hit rate function, Figure 1 shows a simulated hit rate function for a multi-user index nested loops join workload (solid line). It was derived by running such a workload over a range of different memory allocations using the detailed DBMS simulation model that we describe in later in Section 4. Also shown in Figure 1 are two different "fittings" that were derived by taking a pair of points from the simulated hit rate curve and feeding them into the Belady equation described above (dashed lines).

Figure 1 shows that using the Belady equation to predict hit rates at larger memory allocations from observations at smaller allocations can result in pessimistic estimates, i.e. the predicted hit rate can be much lower than would actually be achieved. Normally, pessimistic estimates are safer than optimistic ones, but this is not the case for goal-oriented buffer allocation – a pessimistic hit rate estimate will result in a memory allocation that may be much larger than is actually needed. This will cause the algorithm to "overshoot" its target goal, and can create unstable

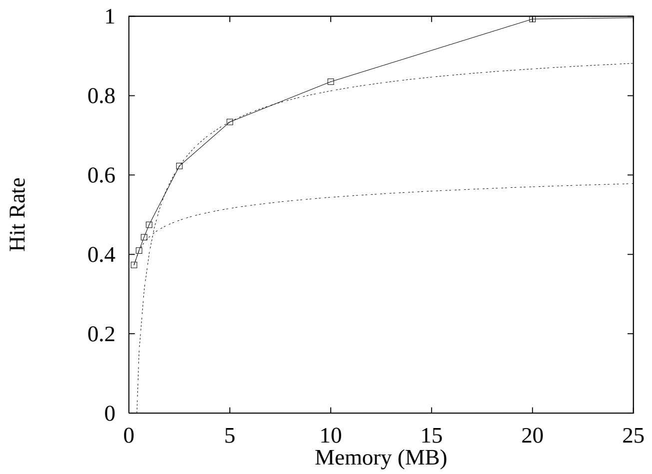

Figure 1: Curves from $1 - a/M^b$

oscillations in a class's performance. Instead, a goal-oriented memory allocator should err on the side of a smaller allocation in order to maximize stability. Dynamic Tuning overcomes this problem (as does Fragment Fencing) by only changing memory allocations in small chunks;[1] this policy prevents unstable behavior, but can result in poor responsiveness because it takes many knob turns to achieve high memory allocations when such an allocation is necessary to achieve a tight response time goal.

While the model equation used by Dynamic Tuning is a reasonable choice, there are inherent problems with any curve-fitting approach. Whatever the model equation, any real curve that doesn't fit the model will be estimated with low accuracy. It is also difficult to determine if a curve-fitting estimate will be optimistic or pessimistic. More complex, higher order functions have the same problem; the model equation will only give a good estimate for those hit rate curves that "look similar" to the model equation. Real hit rate curves have a wide range of shapes and are difficult to capture accurately with a single analytical model.

Dynamic Tuning's approach to memory allocation is to partition the buffer pool and assign each class to a partition managed by its own buffer manger, as described earlier. This approach is simple and effective for classes that do not share data, but some provision needs to be made for classes that *do* share buffer pages. Sharing is not discussed in [Chung 94].

2.3 Fragment Fencing Description

Fragment Fencing's response time estimator makes the simplifying assumption that response time and buffer miss rate are directly proportional. Using this relationship, the target hit rate estimated to achieve the response time

[1] 2.5% of the total buffer pool was used in [Chung 94]. Similarly, Fragment Fencing caps its per-step changes in memory allocation at 10% of the buffer pool [Brown 93].

goals is computed as:

$$HIT^{target} = 1.0 - (M^{obsv} * (R^{goal}/R^{obsv}))$$

where R^{obsv} and R^{goal} are the observed response time and response time goals, respectively, and M^{obsv} is the observed miss rate that occurs with the observed response time. While real response times are functions of many other variables besides buffer hit rate (including disk/cpu/network queueing and service times, lock waits, MPL waits, etc.), assuming a linear relationship is reasonable in the case of a disk-bound class.

Rather than estimating the overall hit rate function for each class, Fragment Fencing estimates it in a piecemeal fashion for each *fragment* of the database that is referenced by the class. A fragment is defined as all of the pages within a relatively uniform reference unit, e.g. a single relation or a single level of a tree-structured index. A uniform reference probability is assumed across the pages of a fragment, and the hit rate of a fragment is therefore estimated to be equal to the percentage of the fragment that is memory resident. Fragment Fencing's goal is to determine, for each fragment, the minimum number of pages that must be memory-resident in order to achieve an overall target hit rate for the class. These minimum amounts are called *target residencies* and are analogous to a working set size for each fragment.

When a class's hit rate needs to be increased by some amount, all of the fragments referenced by the class are sorted in order of decreasing *class temperature* [Copeland 88, Brown 93], which is their size-normalized access frequency (in references per page per second). Using the assumption that the hit rate on a particular fragment is identical to its target residency, the fragments referenced by a class are processed in order from hottest to coldest by increasing the target residency for each one until the hit rates for all fragments add up to the overall hit rate required by the class. This process is reversed when a class's hit rate needs to be decreased.

Once Fragment Fencing's hit rate estimator has determined a target residency for each database fragment referenced by a class, some mechanism is needed to enforce these target residencies. This is done by modifying the existing DBMS's buffer replacement policy to first ask the Fragment Fencing component if removing a page from memory would violate a minimum residency target; if so, the page is not replaced. This type of "passive" allocation allows Fragment Fencing to co-exist with any type of buffer replacement policy, be it global or local. It is passive in the sense that it does not explicitly direct the appropriate pages *into* the buffer pool; it only prevents their ejection from the pool by the DBMS's native replacement policy.

2.4 Fragment Fencing Issues

A potential problem with a fragment-oriented approach, as noted in [Brown 93], is what happens when references within a fragment are not uniform. Since Fragment Fencing measures the actual hit rates of each fragment, it can easily test for violations of the uniform reference assumption by comparing the estimated hit rate to the actual hit rate. If they are significantly different, it is clear that the fragment is being referenced non-uniformly. However, once confronted with the knowledge that fragment references are non-uniform, it is not clear what the fragment's memory allocation should be. Additionally, it is not clear what an average per-page reference frequency means when references are non-uniform within a fragment. The more frequently referenced pages of a fragment will certainly have a higher temperature than the average for the fragment, and therefore sorting fragments by a fragment-wide metric is not very meaningful.

Another problem with Fragment Fencing has to do with its "passive" memory allocation mechanism. Keeping the DBMS's replacement policy "in the dark" with regard to which buffer frames are fenced or not provides a high degree of independence from the underlying replacement policy [Brown 93], but it also has the potential for significant overhead. Because the replacement policy is unaware of which frames are fenced, it is forced to waste time inspecting frames that *seem* like good candidates – only to be overruled by Fragment Fencing. For example, if 80% of the buffer pool is fenced off, then 80% of the candidate pages for replacement will be overruled. This problem is particularly troublesome for clock-based replacement policies (like that of DB2/6000) because fenced frames may cluster together physically in the buffer table; when the clock hand moves into such a cluster, it may have to inspect a large number of consecutive frames before finding one that can be replaced. In order to eliminate this overhead, all fenced frames must somehow be removed from consideration for replacement.

3 Class Fencing

In this section, we describe Class Fencing, our improved goal-oriented buffer management algorithm. Class Fencing adopts the same response time predictor as Fragment Fencing (see Section 2.3), i.e. Class Fencing also assumes that miss rate and response time are proportional. However, it uses a more general hit rate prediction technique based on a notion that we call *hit rate concavity.* Class Fencing's memory allocation mechanism allows for data sharing between classes and represents a compromise between the rigid partitions of Dynamic Tuning and the passive fences of Fragment Fencing. The remainder of this section describes the concept of hit rate concavity and then explains how this concept is used by Class Fencing to predict buffer hit rates. Class Fencing's memory allocation mechanism is then described, and the section closes by discussing two more detailed aspects of the algorithm: estimating memory allocations in the presence of data sharing and computing memory usage statistics.

3.1 The Hit Rate Concavity Assumption

Class Fencing estimates the buffer hit rate that will result from a particular buffer allocation by exploiting the following *concavity theorem*[2]:

> Regardless of the database reference pattern, hit rate as a function of buffer memory allocation is a *concave function* under an optimal replacement policy.

The concavity theorem says that the *slope* of the hit rate curve never increases as more memory is added to an optimal buffer replacement policy. An optimal buffer replacement policy is defined as one that always chooses the least valuable page to replace (e.g. Belady's MIN algorithm [Belady 66]). While optimal replacement policies are not realizable in practice because they require knowledge of future reference patterns, we will argue shortly that the behavior of industrial-strength DBMS replacement policies are "optimal enough" that hit rate concavity applies to them as well.

An informal proof of the concavity theorem can be stated as follows: The slope of the hit rate curve represents the marginal increase in hit rate obtained by adding an additional page of memory. The steeper the slope, the higher the "value" of a particular page (as measured by its ability to increase the hit rate).[3] An *optimal* buffer replacement policy must choose pages for memory residency in decreasing order of their value in order to achieve the highest hit rate for a given amount of memory. Thus, since the slope measures value, and value cannot increase as more memory is added, neither can the slope. Note that concavity also implies that there are no "knees" in an optimal hit rate function. Any knee indicates a "mistake" in page replacement, i.e. implying that lower-valued pages (to the left of the knee) were made memory resident before higher-valued pages (to the right of the knee).

In order to make use of the concavity theorem in a real DBMS, we must ask how close today's commercial DBMS replacement policies are to an optimal policy – i.e. when do they make mistakes? A DBMS should make fewer page replacement mistakes than an operating system (where hit rate knees are common) for two reasons: knowledge of future page reference behavior, and the presence of indexes. A DBMS knows when accesses are going to be sequential versus random. It can prefetch sequentially accessed pages just before they are referenced, and once they are referenced, it can toss or retain these pages based on knowledge of the total number of pages that

[2]A similar theorem has been proven for the case of an IRM reference pattern coupled with an LRU replacement policy by van den Berg and Towsley [van den Berg 93]. To our knowledge, no one has explicitly stated it in the form we do here, although some previous work has exploited the notion of concavity in any case [Dan 95].

[3]This notion of page value is synonymous with the concept of *marginal gain* defined in [Ng 91].

will be scanned [Stonebraker 81]. Random accesses to pages are generally made via indexes,[4] and there are a number of techniques available to insure that more valuable index pages are not replaced by less valuable data pages [Haas 90, O'Neil 93, Johnson 94]. For index pages themselves, it is possible to use reference frequency statistics or information about the last few references to insure that more valuable index pages are not replaced by less valuable index pages [Copeland 88, O'Neil 93, Brown 93].

While it would be impossible to offer any definitive statement about the likelihood of a hit rate knee in real-world buffer managers, we conducted an empirical study of two simulated buffer managers, one modeled after DB2/MVS [Cheng 84, Teng 84, IBM 93a] and the other modeled after DB2/6000 [IBM 93b]. For both of these buffer managers, we mapped the hit rate functions for the TPC A/B and C benchmarks, as well as all of the canonical database reference patterns documented in the DBMIN Query Locality Set Model [Chou 85]. None of them showed a knee. Additional empirical evidence for concavity is provided by Dan et al [Dan 95], where hit rate functions derived from actual traces of DB2/MVS customers were also seen to be free of knees.

While we acknowledge that hit rate function knees are possible in the real world, we believe that they represent pathological cases. Therefore, Class Fencing will adopt the assumption that hit rate concavity holds for the most commonly occuring workloads running on a typical commercial DBMS. Of course, it must also be prepared to accept the failure of this assumption. We will address the impact of non-concave hit rate functions after explaining how the concavity assumption is used by Class Fencing to estimate hit rates, which we now turn our attention to.

3.2 Estimating Hit Rates Using the Concavity Assumption

The hit rate concavity assumption is useful because it enables a simple straight line approximation to be used to predict the memory required to achieve a particular hit rate.[5] Only the last two hit rate observations are needed, and the accuracy of the estimate improves with each new hit rate observation at larger memory allocations. Moreover, unlike a curve-fitting estimator, a straight line approximation always predicts a conservative *lower bound* for its memory allocation. Figure 2 illustrates how the required buffer allocation for a class can be predicted with this approach.

The dashed curve in Figure 2 represents a hypothetical hit rate function for a class, from zero pages up to some maximum memory allocation M^{max} (some large percentage of the total buffer pool). The horizontal line labeled HT represents a target hit rate that the hit rate estimator receives as input (from the response time estimator). The idea is to move the class as quickly as possible to the "X", which represents the memory allocation MT that results in the

[4] This is certainly true for relational systems, but less so for object-oriented systems that support navigational access.

[5] A straight line approximation of buffer hit rate functions was also used in [Chen 93] to predict a memory allocation that maximizes *marginal gain* [Ng 91].

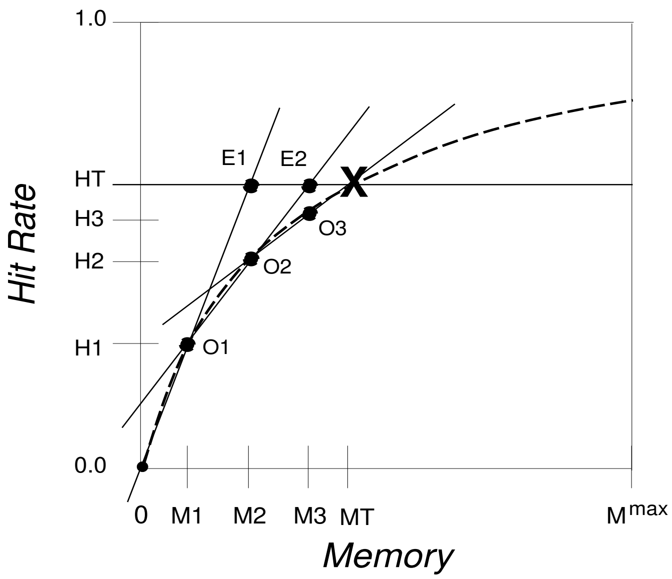

Figure 2: Estimating a concave hit rate function

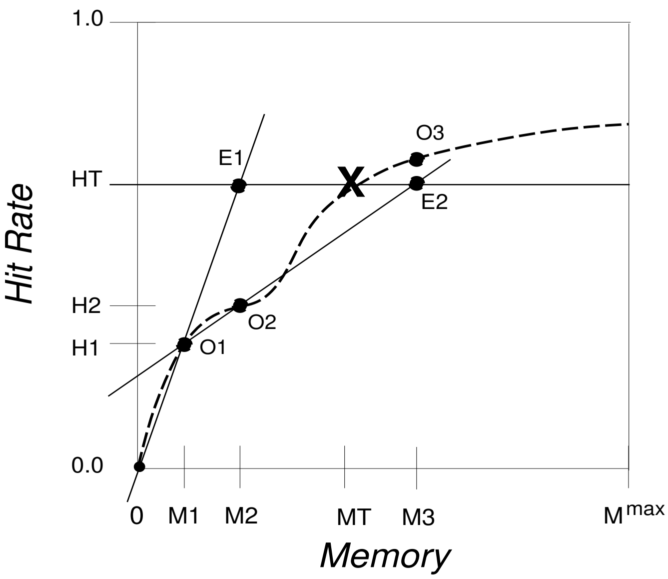

Figure 3: Overshooting a non-concave hit rate function

target hit rate HT. The point labeled O1 indicates the initial observed hit rate H1 of the class with its "naturally occuring" memory allocation M1 – this allocation is what the existing non-goal-oriented DBMS memory allocation policy would "naturally" give to the class in the context of the current workload. To estimate the memory required to achieve the target hit rate HT, a line extending from the origin through O1 is computed; the point at which this line intersects the target hit rate (E1) represents a *lower bound* (M2) on the memory allocation that can achieve the target hit rate HT. If the actual memory allocation required to achieve HT was *less* than M2, this would mean that the concavity assumption had been violated. After increasing the class's memory allocation to M2 and waiting long enough to insure statistical stability, a second observation O2 occurs and another estimate E2 is computed using points O1 and O2. Estimate E2 predicts a required memory allocation of M3. With one more estimate using points O2 and O3, the target hit rate is achieved. If any estimate line were to intersect the M^{max} limit instead of the target hit rate, then the target hit rate is unachievable.

Assuming that concavity holds, Class Fencing's hit rate predictor allows it to aggressively allocate memory in large increments because it can be confident that it will not "overshoot" or be mislead by unachievable hit rate targets. Large memory allocation increments mean that Class Fencing can be extremely responsive, especially in the case of very tight goals. If the concavity assumption does not hold, however, then there may be knees in the hit rate curve. The effect of a hit rate knee on Class Fencing's hit rate predictor depends on where the observation points are relative to the knee. If they are straddling the knee (with one point on either side of it), then the slope computed across the two points will still be fairly accurate. In the worst case, one of the observations will lie directly in the knee. Such a worst case scenario is illustrated in Figure 3. In this case, the computed slope will be too low, the estimated memory allocation ($M3$) no longer represents a lower bound, and Class Fencing will overshoot the allocation for the class. In order to correct for these (hopefully rare) cases, Class Fencing must therefore incorporate code to estimate in the *downward* as well as the upward direction. In Figure 3, for example, the next estimate after $E2$ would extrapolate between points $O3$ and $O2$. One more estimate would likely be required to achieve the goal.

3.3 Class Fencing's Memory Allocation Mechanism

Class Fencing's memory allocation mechanism is a compromise between the rigid partitions of Dynamic Tuning and the passive fences of Fragment Fencing. Instead of building individual fences around each database fragment referenced by a class, a *single* fence is built to protect *all* of the pages referenced by the class, regardless of which fragment they belong to. The choice of which pages belong inside versus outside the fence is made by a buffer manager that is local to the class. A separate global buffer manager manages pages for no-goal classes as well as any "less valuable" unfenced pages that belong to goal classes. The global buffer manager is the source for all "victim" frames necessary to satisfy any page miss. Note that since the global buffer manager contains no fenced frames, no additional overhead is required on a page replacement decision in order to deal with fenced frames. Finally, a single buffer frame table and associated disk-page-to-buffer-frame mapping table is shared by all buffer managers, both global and local.

For each goal class that cannot meet its goal "naturally" by competing for frames in the global buffer manager (a *violating class*), a separate and identical instance of the existing DBMS replacement policy is cloned to manage a set of frames that are then protected from replacement by other competing classes. The choice of replacement policy is irrelevant to the Class Fencing algorithm; it is simply replicated when a goal class is in violation. Each violating class C has a limit determined by Class Fencing's hit rate predictor, *poolSize[C]*, that represents the maximum number of buffer frames that can be managed by class C's local buffer manager. The global buffer manager also has a pool size, *poolSize[GLOBAL]*, and the sum of the local and global pool sizes equals the total amount of DBMS buffer pool memory. Any pool size increase for a goal class implies a corresponding decrease in the pool size for the global buffer manager, and any local pool size decrease implies a global pool size increase of identical size. Like Dynamic Tuning, Class Fencing's goal is to set a pool size for each violating class so that it can meet its goal. Unlike Dynamic Tuning, however, only the replacement policy is replicated for each class; the common frame table and mapping table enables buffered pages to be shared across classes.[6]

Class Fencing's allocation mechanism operates as follows. On a buffer miss by a violating class, a free frame is

[6] A similar sharing technique was used by the DBMIN algorithm [Chou 85]. Class Fencing differs from that approach in that DBMIN partitioned memory on the basis of *file instances* and used a different replacement policy for each instance. Class Fencing partitions around classes and uses an identical replacement policy (the existing DBMS replacement policy) for each one.

stolen from the global buffer manager and then reassigned to the local buffer manager for the violating class. If the local buffer manager now exceeds its *poolSize* limit, then *its* replacement policy is called upon to choose a frame to donate back to the global buffer manager, where it is treated as recently referenced.[7] On a buffer miss by a no-goal class, or by a goal class that "naturally" meets its goal with the existing buffer allocation mechanism, one of the frames managed by the global buffer manager is chosen for replacement; the referenced page is read into that frame and assigned to the global buffer manager. The page then stays in memory until the global buffer manager's replacement policy decides to eject it. If all classes can meet their goals with the existing allocation mechanism, then no local buffer managers exist and the system's behavior is indistinguishable from a non-goal-oriented system.

3.4 Class Fencing Details

There are two additional details that need to be addressed to complete our description of Class Fencing. The first stems from the fact that a class that shares data with other classes can use buffer frames that are controlled either by its local buffer manager or by some other buffer manager. Any page referenced by a class while it is still in memory is considered "in use" by the referencing class, regardless of which buffer manager controls the page. Thus, the pool size for a class only represents a *lower bound* on the number of frames used by the class; it does not represent the *total* number of frames in use by the class. On the other hand, Class Fencing's hit rate estimator is based on the *total* number of frames used by a class, regardless of which buffer manager they reside in. For sharing classes, some mechanism is needed to translate the hit rate estimator's proposed memory allocation into a (necessarily smaller) pool size value for the class. For non-sharing classes, this translation is not needed because the pool size is the same as the number of frames in use for these classes.

Predicting the number of frames in use by a sharing class given a particular pool size is relatively simple. Whenever a class C is observed, the percentage p of non-local buffer frames that it is using can be computed as follows:

$$nonLocal[C] = bufSize - poolSize[C]$$

$$p = (inUse[C] - numLocal[C]) \ / \ nonLocal[C]$$

Here, $inUse[C]$ is a running count of the total number of frames used (referenced) by the class at any moment (regardless of which buffer manager controls them), $numLocal[C]$ is the number of frames currently managed by the class's local buffer pool, and *bufSize* is the total number of DBMS buffer pool frames. When the pool size is increased for the class, the miss rate of the class decreases, and therefore the rate at which the class asks for frames

[7] Actually, if the page chosen for replacement by a local buffer manager comes from a database fragment that is not shared by any other classes, it can be safely tossed out of the buffer pool immediately. This is because there is no chance of harming the buffer hit rate of any other class by doing so. Shared pages must remain resident in the global buffer before they are ejected because they are likely to be referenced by (and reassigned to the buffer manager for) another class.

from the global buffer manager also decreases. The currently observed percentage p thus represents an upper bound on the percentage of frames that this class will utilize outside of its local buffer pool after its pool size is increased. Using p as an upper bound, the estimated number of frames utilized by a class C, given its current *poolSize[C]* and a local pool size increase of $\Delta poolSize$ (which causes a decrease in *nonLocal[C]*), can be computed as

$$inUse[C]^{est} = poolSize[C] + \Delta poolSize + p \cdot (nonLocal[C] - \Delta poolSize)$$

Solving this equation for $\Delta poolSize$ allows Class Fencing to determine the new local pool size that is likely to result in the targeted overall buffer allocation for a class.

The final algorithmic detail addresses the fact that the count of frames used by a given class can vary dramatically over time. As just explained, the current value of *inUse[C]* represents the (transient) amount of memory used by a class C at a particular time. Therefore, instead of using *inUse[C]* directly, a *time-weighted* frame count is actually used instead. This means that the X-axes of Figures 2 and 3 should actually be interpreted as the time-weighted counts of frames in use during the current observation interval; the time-weighted frame count for a class is reset at the end of every observation interval. The cost of maintaining a time-weighted frame count for each class is low, as a single floating point multiply is all that is needed on every page-in or page-out of a frame that is used by a class.

4 Simulation Model

This section provides a detailed description of the simulation model that we will use for evaluating Class Fencing. We describe the simulated DBMS configuration, database, and workload.

4.1 System Configuration Model

The external workload source for the system is modeled by a set of simulated terminals. Each terminal submits a stream of transactions of a particular class, one after another. 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 provide average disk utilizations of 50 to 60%.

The simulated configuration contains eight disks that are modeled after the Fujitsu Model M2266 (1 GB, 5.25") disk drive. However, we have disabled their disk caches in this simulation study as a result of our experience with the Fragment Fencing implementation in DB2/6000. It turns out that our simulated disk cache hit rates were much higher than what we observed in the real system. The reason for the difference is that our simulator assumed that random, single-page disk accesses would not pollute the cache with pages that are unlikely to be reaccessed, allowing the cache to be mostly dedicated to the prefetching of sequential disk scans. Protecting sequential scans from concurrent random accesses in this manner requires the cooperation of the DBMS, O/S, disk driver software, and disk controller firmware. This degree of cooperation rarely occurs in the real world, especially with products built to be portable across a wide range of hardware and software platforms. Given this situation, it is safer to assume the worst-case disk cache behavior, so we therefore disable caches on all simulated disks for this study.

The system's simulated 30 MIP CPU is scheduled using a round-robin policy with a 5 millisecond time slice, and the disk queue is managed using an elevator algorithm. The buffer pool consists of a set of 3072 main memory page frames of 8K bytes each (24 MB). While a 24 megabyte buffer pool is on the low end for our workload and configuration, it is appropriate to study Class Fencing in a memory-constrained environment since buffer hit rates are a significant performance factor; if memory were unconstrained, then hit rates would be too high to observe the effects of any memory management decisions. The buffer manager is modeled after that of DB2/MVS [Teng 84, IBM 93a]. It utilizes separate LRU chains for sequential and random accesses, and includes an asynchronous prefetcher which operates as follows: At the initiation of a file or index leaf page scan, the prefetcher asynchronously orders the next block of pages (eight 8K pages in our case) to be prefetched. When the penultimate page in the prefetch block is referenced, an I/O for the next block of pages is asynchronously scheduled. This approach enables the prefetcher to stay just ahead of the scanning process while using a minimal amount of memory.

4.2 Database Model

File Name	Tuple Bytes	# of Tuples	# 8K Pages	% Buf Pool	% of Refs	Access Type
customer file	655	30K	2500	81.4	2.2	U/H
customer index (ID)	12	30K	89	2.9	1.5	U/H
customer index (name)	24	30K	45	1.5	2.9	U
district file	95	10	1	0.0	1.5	-
history file	46	33K	186	6.1	0.6	A
history index	12	33K	50	1.6	1.3	A
item file	82	100K	1011	32.9	7.4	U/C
item index	16	100K	197	6.4	14.9	U/C
new order file	8	9K	10	0.3	0.8	90/10
new order index	12	9K	15	0.5	1.6	90/10
order file	24	30K	88	2.9	0.8	90/10
order index	16	30K	60	2.0	1.7	90/10
order line	54	300K	1987	64.7	8.6	90/10
order line index	16	300K	589	19.3	25.6	90/10
stock file	306	100K	3847	125.2	11.7	U
stock index	16	100K	197	6.4	15.2	U
warehouse file	89	1	1	0.0	1.5	-
DBMIN file A	182	100K	2223	72.3	-	U
DBMIN unclustered idx A	12	100K	147	4.8	-	U
DBMIN clustered idx A	12	100K	147	4.8	-	U
DBMIN file B	182	100K	2223	72.3	-	U
DBMIN unclustered idx B	12	100K	147	4.8	-	U
DBMIN clustered idx B	12	100K	147	4.8	-	U

Table 1: Database characteristics

The database model consists of a two-part database, with one part taken directly from the TPC-C benchmark [TPC 94] using a scale factor of one (one warehouse), and the other drawn from a previously published performance study of the DBMIN buffer management algorithm [Chou 85]. The DBMIN portion of the database is a subset of the original Wisconsin Benchmark Database [Bitton 83], except that here we scale up the number of tuples in each relation by a factor of ten.

The TPC-C benchmark represents an order-entry application for a wholesale distribution company. Its files and associated B+ tree indexes are summarized in Table 1. While the choice of indexes is not specified in the benchmark, those listed in Table 1 represent a typical implementation. For indexes, the **Tuple Bytes** column indicates the size of a key/pointer pair. The **% Buf Pool** column expresses the size of each file as a percentage of the 24MB buffer that we use throughout the experiments in Section 5. The **% of Refs** column indicates the percentage of the database references that are directed at a particular file by the TPC-C benchmark transactions. Note that over half of the references are directed at the stock, stock index, and order line index files. Finally, the **Access Type** column indicates the type of page reference pattern that occurs on a file: U implies a uniformly random page reference probability, 90/10 means that 90% of the references are directed at 10% of the file, and A stands for append-only access. U/H and U/C are special distributions defined in the TPC-C benchmark and can be roughly described as uniform with hot spots and uniform with cold spots, respectively (see [TPC 94] for a detailed description).

All the database files are fully declustered over the eight disks in the configuration (except for those files with fewer than eight pages).

4.3 Workload Model

The simulated workloads used in the experiments of Section 5 are composed of different combinations of the TPC-C workload together with several DBMIN query classes. Because we are primarily interested in the page reference patterns of these classes, all of the workload classes are read-only.[8] The specific behavior of the classes is described in the following paragraphs.

TPC-C: This simulated workload class faithfully duplicates the reference patterns of the TPC-C benchmark as specified in [TPC 94]. TPC-C models an order-entry business and is composed of a mix of five different transaction types. These queries are mostly index scans of varying selectivities that produce the reference frequencies and patterns shown in Table 1 (a detailed description is provided in [TPC 94]). As stated earlier, TPC-C exhibits a high degree of locality. The order line and order line index files receive over one third of the references, and the access to

[8]Note to reviewers: Our final version of the paper will also summarize results with read/write workloads and will employ an asynchronous page cleaner in the simulated buffer manager. Unfortunately, this part is our simulator is currently broken.

both is highly skewed, giving this workload a relatively high hit rate at low cost in memory. As a result, its response times can only be varied over a relatively narrow range without a huge investment in memory. Note that because of its skewed references within database fragments, TPC-C violates Fragment Fencing's uniform reference assumption and therefore its performance cannot be controlled by Fragment Fencing.

DBMIN Query 2 (Q2): The Q2 class is a non-clustered index scan of DBMIN file A with a 1% selectivity [Chou 85]. Because file A and its index can fit entirely in memory, this class is very sensitive to its buffer hit rate and is therefore more easily controlled than TPC-C.

DBMIN Query 3 (Q3): The Q3 class is an index nested loops join of DBMIN files A and B [Chou 85]. File A is scanned using a clustered index with a 2% selectivity, and file B is scanned directly. When Q2 and Q3 are running together in the same workload, they share the common file A, causing their performance is somewhat linked. The total number of database pages referenced by a Q3 query is about 50% larger than the buffer pool, so Q3's performance is slightly less sensitive to its buffer hit rates than Q2.

4.4 Parameter Summary

Parameter	Value
# TPC-C terminals	50
Mean TPC-C think time	5 sec
# Q2 terminals	10
Mean Q2 think time	10 sec
# Q3 terminals	10
Mean Q3 think time	10 sec
Number of CPUs	1
CPU speed	30 MIPS
Number of disks	8
Page size	8 KB
Memory size	24 MB (3072 pages)
Disk cylinder size	83 pages
Disk seek factor	0.617
Disk rotation time	16.667 msec
Disk settle time	2.0 msec
Disk transfer rate	3.1 MB/sec

Table 2: Simulation parameter settings

Function	# Instr
read rec from buff page	300
write rec to buff page	100
test index entry	50
start an I/O	5000
apply predicate	100
initiate select	20000
terminate select	5000
initiate join	40000
terminate join	10000

Table 3: Simulated instruction counts

The important simulation parameters for this study are listed in Tables 2 and 3. The 30 MIP CPU results in CPU utilizations of 50-75%. The number of disks, number of terminals, and think times were chosen to ensure that disk utilizations lie in the 50 to 60% range. The software-related parameters in Table 3 are based on instruction counts taken from the Gamma parallel database system prototype [DeWitt 90], and the disk characteristics approximate those of the Fujitsu Model M2266 disk drive (as stated earlier).

5 Experiments and Results

In this section, we use our simulation model to examine how well Class Fencing can achieve a variety of goals for several different multiclass workloads. In order to obtain statistically meaningful simulation results, we execute the simulations for 90 simulated minutes. We collect response time statistics only for the last hour of the simulation in order to factor out the solution searching time from the averages, as the averages are meant to indicate steady-state behavior. We also include a few transient analyses of response times to how stable the algorithm is over time.

The performance metrics that we will use for judging Class Fencing's behavior are the *performance index* of each goal class and the *number of knob turns* (i.e. different memory allocations) that it takes to reach a point where the class's goal is achieved for three consecutive observation intervals. The performance index of a class is defined as the average response time of the class (over the hour-long statistics collection period) divided by its response time goal, as described in Section 1.2, and is a measure of accuracy. The number of knob turns is a measure of responsiveness. We also show the response times of any no-goal class in order to roughly indicate the amount of "excess" resources left over after the goal classes have been given what they need to meet their goals; the larger the amount of left-over resources, the lower the no-goal response times.

5.1 TPC-C and DBMIN Q2

Our first set of experiments pairs the TPC-C and DBMIN Q2 classes together. We experiment with three variants of this workload: goals set for Q2 only, goals for TPC-C only, and goals for both classes.

5.1.1 Goals for Q2 Only

The first TPC-C/Q2 experiment sets a range of goals for the Q2 class, allowing the TPC-C class's response time to "float" as a no-goal class. Table 4 shows the results of this experiment. Each row in Table 4 represents a separate simulation run using a different goal for the Q2 class. The columns show the input goal, the resulting average response time for the Q2 class, its performance index, the average TPC-C (no-goal) class response time, the number of knob turns (intervals) that it took to achieve the goal, and the resulting memory allocation for the Q2 goal class (out of a total of 3072 8K buffer frames). The interval length used for the Q2 class is 100 completions, which (depending upon the goal and resulting throughput for the class) translates to anywhere from about 150 to 225 seconds.

The performance indexes in Table 4 show that Class Fencing can achieve the goals fairly accurately for the Q2 class – to within four percent at most. The last row in the table represents a goal that is satisfied "naturally" by the system's buffer manager. An interesting aspect of this workload is how insensitive the TPC-C response times

Q2 Goal (sec)	Q2 Resp (sec)	Q2 PI	TPC-C Resp (sec)	# of Knob Turns	Q2 Mem Alloc (pages)
0.150	0.153	1.01	0.436	7	2336
0.250	0.259	1.04	0.426	4	2293
0.500	0.494	0.99	0.421	3	2249
0.700	0.705	1.01	0.425	5	2220
1.000	0.981	0.98	0.421	4	2193
2.000	1.957	0.98	0.423	3	2102
5.000	4.820	0.96	0.437	6	1919
10.000	5.770	0.58	0.436	0	0

Table 4: TPC-C/Q2, with goals for Q2

are to the different levels of Q2 performance (and memory allocation). Because TPC-C has such high locality, large changes in memory allocation have a minimal effect on its performance.

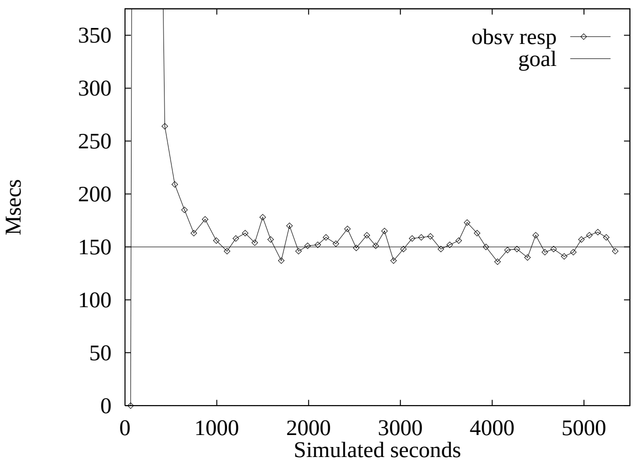

Figure 4: Q2 interval response times

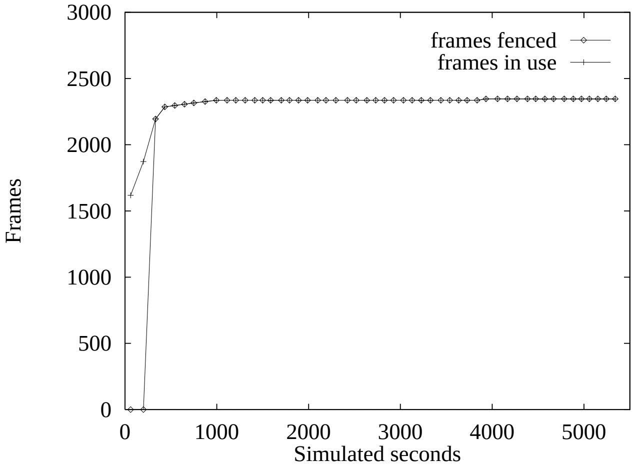

Figure 5: Q2 interval memory allocation

Note that Table 4's tightest achievable goal, 150 msecs, takes more knob turns to achieve than do the other goals (7 knob turns versus an average of 4). The reason for this is that when hit rates are very high, very small changes in memory allocation can bring about large relative differences in miss rates (since so few I/Os are occurring). When hit rates are very high, Class Fencing is forced to take smaller steps in order to prevent an overshoot, and this is why very tight goals may require more knob turns than looser ones. This behavior can be seen in Figures 4 and 5, which show the Q2 class response time and number of buffer frames allocated as function of time for the 150 msec goal experiment. Figure 5 shows that while the first memory allocation brought the class very close to its goal, six more very small adjustments were required to achieve a stable solution. From a responsiveness standpoint, the "number of knob turns" measure is imperfect since it does not recognize the magnitude of each knob adjustment. In this case, the goal was reached for the most part after four adjustments, with the remainder providing additional fine tuning. Figures 4 and 5 also show that Class Fencing behaves in a very stable manner for this workload – it is clear that once the solution is found, the memory knob is left untouched.

5.1.2 Goals for TPC-C Only

TPCC Goal (sec)	TPCC Resp (sec)	TPCC PI	Q2 Resp (sec)	# of Knob Turns	TPCC Mem Alloc (pages)
0.300	0.301	1.00	24.0	5	2816
0.350	0.345	0.99	16.7	5	2305
0.375	0.374	1.00	14.2	4	1731
0.400	0.399	1.00	11.6	3	1378
0.425	0.425	1.00	5.1	0	0

Table 5: TPC-C/Q2, with goals for TPC-C

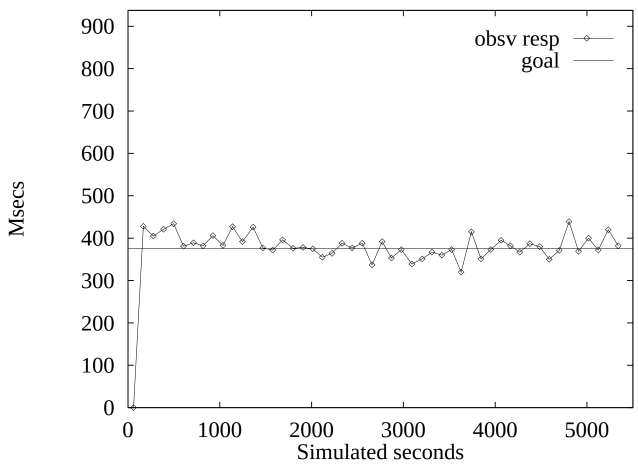

Figure 6: TPC-C interval response times

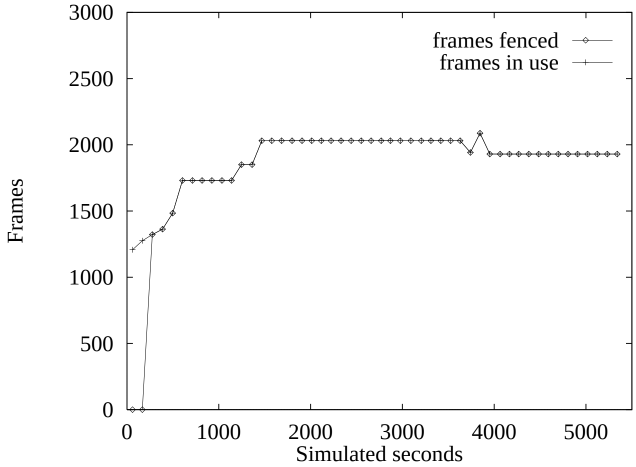

Figure 7: TPC-C interval memory allocation

Our second experiment in this set uses the same workload as the previous experiment, but reverses the roles of the two classes. Here, goals are set for the TPC-C class, while the Q2 acts as a no-goal class. The observation interval for the TPC-C class is set to 1000 completions, which translates to about 160 seconds at the throughputs exhibited in these experiments. Table 5 shows the results of a series of simulations for this workload. As before, each row represents a simulation run with a different goal for the TPC-C class. Because of its high locality, TPC-C has a much narrower range of possible response times, so there are fewer rows in this table. As before, Class Fencing achieves the goals to within a few percent using only a few knob adjustments. In contrast with the TPC-C class behavior of the previous example, Table 5 shows that the Q2 no-goal class response time is extremely sensitive to the TPC-C memory allocation. As the TPC-C class's goals are loosened, the Q2 class is able to achieve better performance using the additional leftover buffer memory. Figures 6 and 7 show the transient behavior for the middle (375 msec) goal of Table 5. For this goal, it only takes four intervals to find a solution, but because of the high variance of TPC-C (both in transactions' service demands and arrival rates), continuing adjustments need to be made in order to maintain the average response time. These adjustments are relatively stable, however, and do not cause any oscillating behavior.

5.1.3 Goals for Both Q2 and TPC-C

The last experiment with this workload provides goals for *both* the TPC-C and Q2 classes. A third class is added as a no-goal class to consume any left-over resources in the case where both classes have a loose goal. This is necessary because otherwise the goals would have to be set such that all of memory is exactly consumed by both TPC-C and Q2; if some memory was left over, then one class would always naturally exceed its goal and the experiment would behave as if there only one class with a goal. The no-goal class for this experiment is another Q2-like class that references a distinct file from the Q2 goal class. To maintain the same aggregate system load, the original ten Q2 class terminals are split in to two groups: four belong to the Q2 goal class, and six are assigned to the Q2-like no-goal class. More terminals are assigned to the no-goal class to make it a slightly more aggressive competitor for buffer frames. One consequence of the reduced number of Q2 goal class terminals is a lower throughput for the Q2 goal class (0.2 versus 0.6 queries per second for a 700 msec goal). A lower throughput increases the time required to gather statistically valid measurements, and therefore implies a longer time interval between knob turns; the longer the interval between knob turns, the more critical it is to find a solution in as few turns as possible.

TPCC Goal (sec)	Q2 Goal (sec)	TPCC PI	Q2 PI	# TPCC Knob Turns	# Q2 Knob Turns	No-goal Resp (sec)
* 0.300	20.0	1.03	1.10	4	1	27.1
0.300	25.0	1.03	0.97	4	0	21.4
0.400	20.0	1.00	1.04	3	1	17.2
* 0.400	15.0	1.05	0.99	5	2	29.2
0.500	10.0	1.01	1.02	2	3	23.2
* 0.500	5.0	1.06	0.99	4	3	27.6
0.700	5.0	0.99	1.01	0	3	22.8
0.700	2.0	1.03	0.96	1	5	23.5

Table 6: TPC-C/Q2, with goals for both

Table 6 shows the results of this experiment, including columns for both the TPC-C and Q2 class response time goals, their resulting performance indexes, the number of knob turns it took to find the solutions, and the resulting no-goal class response time. The performances indexes for this workload are mostly within a few percent of the goals for this workload as well, indicating that Class Fencing is successfully doing its job. Three exceptions are the tightest goal combinations marked that are starred in Table 6: 0.300/20.0, 0.400/15.0, and 0.500/5.0. These three goal pairs together consume most of the available memory, leaving very little for the no-goal class (as can be seen by the poor no-goal performance in these cases). The performance indexes for these tight goal combinations show some goals being violated by as much as ten percent because of the shortage of buffer memory. Class Fencing is not designed to make any attempt to reallocate memory in order to minimize the maximum performance index in cases like these where the goals are likely too aggressive for the system as configured. This situation is called *degraded* *mode* [Nikolaou 92]. Like Fragment Fencing, Class Fencing assumes that the system will not be required to operate in degraded mode except during short transient periods. (Otherwise, if the specified goals are truly important, the system configuration must be upgraded to provide the capacity required to meet the goals in steady-state.)

5.2 DBMIN Q2 and DBMIN Q3

Our second group of experiments pairs the Q2 and Q3 DBMIN classes together. These two classes share a common file, so their performance is somewhat linked. As a result, this workload is more challenging than the TPC-C/Q2 workload because Class Fencing must use a *third* estimate when there is sharing between classes. In addition to the response time and hit rate estimates, it must now estimate the total memory utilized per class for a given fence size, as pages used by a class may reside inside or outside of its local buffer pool. This estimate was described in Section 3.4. As was was the case for TPC-C/Q2, we experiment with three variants of this workload: goals for Q2 only, goals for Q3 only, and goals for both Q2 and Q3.

5.2.1 Goals for Q2 Only

Q2 Goal (sec)	Q2 Resp (sec)	Q2 PI	Q3 Resp (sec)	# of Knob Turns	Q2 Mem Alloc (pages)
0.110	0.110	1.00	34.689	4	2342
0.300	0.296	0.97	33.108	8	2268
0.500	0.496	0.99	32.497	5	2224
1.000	1.000	1.00	32.256	10	2135
2.500	2.498	1.00	31.151	13	1935
5.000	4.982	1.05	31.202	6	1667
10.000	9.596	0.96	27.604	3	1138
15.000	13.956	0.96	8.393	0	0

Table 7: Q2/Q3, with goals for Q2

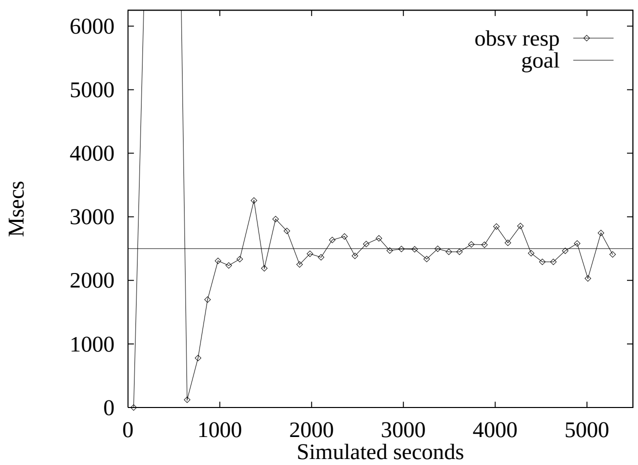

Figure 8: Q2 interval response times

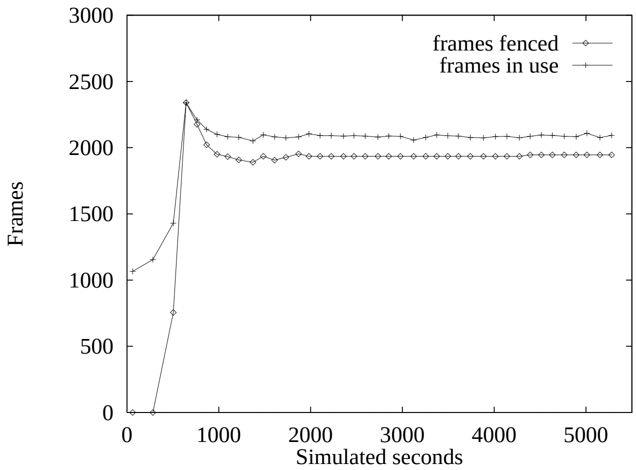

Figure 9: Q2 interval memory allocation

Table 7 shows the steady-state results for this workload when goals are set only for the Q2 class, with the Q3 class acting as a no-goal class. Class Fencing is fairly accurate for this workload as well, holding the Q2 class to within five percent of its goal, with at most a five percent error. However, Class Fencing is not as responsive for this workload as it was for the TPC-C/Q2 workload; it takes as many as 13 knob turns to find a solution for the 2.5 second goal. Figures 8 and 9 show the transient response times and memory allocations, respectively, for the 2.5 second goal experiment. The reason that Class Fencing took so long to find a solution is that the memory allocation initially overshot the final solution by about 12%, and it then took 12 more knob adjustments to correct it. The over-allocation was not due to any lack of concavity in the hit rate function, but simply a combination of errors from all three of Class Fencing's estimates. The long correction time is due to a phenomenon discussed earlier: in the region where hit rates are very high (> 93% in this case), Class Fencing tends to make very small adjustments because its estimators assume (correctly so) that small memory allocation changes may cause very large fluctuations in hit rate and response time at high hit rates. Figures 8 and 9 also show that the 13 knob turn measure sounds much worse than it is; after only the fourth knob adjustment, the class is within five percent of the final solution.

5.2.2 Goals for Q3 Only

Q3 Goal (sec)	Q3 Resp (sec)	Q3 PI	Q2 Resp (sec)	# of Knob Turns	Q3 Mem Alloc (pages)
01.00	0.99	0.99	16.0	9	2764
01.25	1.25	1.00	15.7	4	2552
01.50	1.46	0.97	15.9	6	2505
01.75	1.80	1.03	15.8	8	2366
02.00	1.98	0.99	16.1	11	2358
03.00	3.02	1.01	16.1	8	2124
04.00	3.98	1.00	16.1	3	1891
05.00	5.23	1.05	16.1	8	1511
10.00	8.39	0.84	14.4	0	0

Table 8: Q2/Q3, with goals for Q3

Table 8 shows the steady-state results for this workload when goals are set instead for the Q3 class, with the Q2 class now acting as a no-goal class. Here too, response times are held to within a few percent of the the goals (with at most a five percent error). As was the case when goals were set for the Q2 class, it takes more knob turns for this workload than it did for the TPC-C/Q2 workload. This occurs for reasons just described; the cumulative errors from all three of Class Fencing's estimates can lead to an initial overshoot followed by a slow correction in the goal class's memory allocation.

5.2.3 Goals for Q2 and Q3

Table 9 shows the results of our final experiment, where goals are set for *both* the Q2 and Q3 classes. As before, we add another Q2-like class as a no-goal class in order to consume any resources left over when the combined goals

Q2 Goal (sec)	Q3 Goal (sec)	Q2 PI	Q3 PI	# Q2 Knob Turns	# Q3 Knob Turns	No-Goal Resp (sec)
1.0	50.0	1.00	0.94	4	0	23.4
* 5.0	30.0	1.21	1.17	2	3	28.3
5.0	40.0	0.98	0.85	2	0	23.8
* 10.0	30.0	1.10	1.12	3	2	26.5
10.0	40.0	0.99	1.00	3	0	24.2
15.0	25.0	1.08	1.00	3	4	25.7
15.0	20.0	1.06	1.08	3	7	30.2
20.0	5.0	1.04	1.01	1	6	23.9
20.0	10.0	1.06	1.11	2	2	22.7
20.0	20.0	1.03	0.99	2	2	20.7

Table 9: Q2/Q3, with goals for both

for the Q2 and Q3 classes do not require the entire buffer pool. Instead of ten Q2 and ten Q3 terminals, there are six Q2, six Q3, and seven no-goal terminals for this experiment (which lowers the Q2 and Q3 class throughputs and makes them more difficult to control). Except for the starred unachievable goal pairs, Class Fencing is reasonably accurate for this workload. The biggest error is that the 10 second goal for the Q3 class is violated for the 20/10 goal pair by 11%. The reason for this violation is as follows: A solution is first found (slowly) for the Q2 class's 20 second goal. Initially the ten second Q3 goal is loose enough to be satisfied without any fence. Once the Q3 class is affected by the increase in Q2's memory allocation, it too begins to search for its solution. Although it only took two knob turns to (slowly) find Q3's solution, the search for its solution was started late enough that it did not complete before the steady-state statistics collection period had begun. On the other hand, the five second Q3 goal (of the 20/5 goal pair) was sufficiently tight that the search for its solution began simultaneously with that for Q2; a five second goal also increased the throughput of the Q3 class, so it moved much more quickly (even though it required six knob turns to find its solution).

6 Summary and Future Work

In this paper, we described the problem of goal-oriented DBMS buffer management and defined a set of criteria with which to evaluate solutions to this problem. We reviewed two existing solutions, Dynamic Tuning and our own Fragment Fencing scheme, and described a new algorithm called Class Fencing that attempts to overcome the limitations of these prior solutions. Class Fencing is based on the notion of *hit rate concavity*, which allows a simple straight line approximation to predict a class's buffer hit rate as a function of memory while at the same time guaranteeing a conservative memory estimate. Using a detailed simulation model, we explored the steady-state and transient performance of Class Fencing for a various multiclass workloads and goal combinations.

Our experiments have shown that Class Fencing is stable and as accurate (like Fragment Fencing), holding most classes to within a few percent of their goals without excessive knob twiddling. Class Fencing was also shown to be very responsive (much more so than Fragment Fencing) because its hit rate estimator does not have to be restricted to allocating memory in small chunks. Responsiveness is a key advantage of Class Fencing, allowing it to find solutions with very few knob turns. In addition, Class Fencing eliminates the primary overhead of Fragment Fencing (unnecessarily examining fenced frames for replacement) as well as some of its other, smaller overheads (e.g. tracking the hit rates and memory residency of each database fragment for each class). While the best test of overhead lies in an implementation, Class Fencing's favorable overheads relative to Fragment Fencing gives us cause for optimism on this point. Class Fencing is also fairly robust because its primary assumption, hit rate concavity, applies to a wide range of workloads. It can handle arbitrary skew and can also detect unachievable hit rates, both of which were stumbling blocks for Fragment Fencing.

In the future, we need to experiment with a wider range of workloads to determine the outer limits of Class Fencing's applicability. We also plan on enhancing Class Fencing's behavior when the system is operating in degraded mode (i.e., when the goals are too aggressive for the configuration). It is not uncommon for a system's workload demands to increase slowly over a period of weeks or months, and it would be would be nice for the algorithm to degrade gracefully and warn the administrator when this has occurred (or appears likely). Finally, we plan on replacing the Fragment Fencing component of our more general M&M algorithm [Brown 94] and then re-evaluating M&M's performance on even complex multiclass workloads. (In addition to the buffer memory knob, M&M also controls working storage memory and multiprogramming levels).

Acknowledgements

The authors would like to thank Praveen Seshadri and Mark McAuliffe for valuable comments on previous versions of this paper. John McPherson and Pat Selinger from IBM Almaden provided generous support for our Fragment Fencing prototype, and the entire DB2/6000 team provided tireless and patient assistance with endless technical questions.

References

[Belady 66] L. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal*, 5(2), July 1966.

[Bitton 83] D. Bitton, D. DeWitt, C. Turbyfill, "Benchmarking Database Systems – A Systematic Approach," *Proc. 9th Int'l VLDB Conf*, Florence, Italy, October 1983.

[Brown 93] K. Brown, M. Carey, M. Livny, "Managing Memory to Meet Multiclass Workload Response Time Goals," *Proc. 19th Int'l VLDB Conf*, Dublin, Ireland, August 1993.

[Brown 94] K. Brown, M. Mehta, M. Carey, M. Livny, "Towards Automated Performance Tuning for Complex Workloads," *Proc. 20th Int'l VLDB Conf*, Santiago, Chile, September 1994.

[Chen 93] C. Chen, N. Roussopoulos, "Adaptive Database Buffer Allocation Using Query Feedback," *Proc. 19th Int'l VLDB Conf*, Dublin, Ireland, August 1993.

[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, August. 1985.

[Chung 94] J. Chung, D. Ferguson, G. Wang, C. Nikolaou, J. Teng, "Goal Oriented Dynamic Buffer Pool Management for Database Systems," *IBM Research Report RC19807*, October, 1994.

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

[Corbato 68] F. Corbato, "A Paging Experiment with the Multics System," Project MAC memo MAC-M-384, Massachusetts Institute of Technology, Boston, MA, July 1968.

[Dan 95] A. Dan, P.S. Yu, J.-Y. Chung, "Characterization of Database Access Pattern for Analytic Prediction of Buffer Hit Probability," *VLDB Journal*, 4(1), January 1995.

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

[Easton 79] M. Easton, P. Franaszek, "Use Bit Scanning in Replacement Decisions," *IEEE Transactions on Computing,* 28(2), February 1979.

[Ferg 93] D. Ferguson, C. Nikolaou, L. Geargiadis, K. Davies, "Goal Oriented, Adaptive Transaction Routing for High Performance Transaction Processing Systems," *Proc. 2nd Int'l Conf. on Parallel and Distributed Systems*, San Diego CA, January 1993.

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

[IBM 93a] IBM Corporation, *IBM Database 2 Version 3 Performance Monitoring and Tuning SC26-4888*, IBM Corporation, San Jose CA, December 1993.

[IBM 93b] IBM Corporation, *Database 2 AIX/6000 Administration Guide SC09-1571*, IBM Corporation, North York, Ontario, Canada, October 1993.

[Johnson 94] T. Johnson, D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," *Proc. 20th Int'l VLDB Conf,* Santiago, Chile, September 1994.

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

[Nikolaou 92] C. Nikolaou, D. Ferguson, P. Constantopoulos, "Towards Goal Oriented Resource Management," *IBM Research Report RC17919*, April 1992.

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

[Pierce 83] B. Pierce, "The Most Misunderstood Parts of the SRM," *Proc. SHARE 61* (IBM users group), New York NY, August 1983.

[Stonebraker 81] M. Stonebraker, "Operating System Support for Database Management," *CACM,* 24(7), July, 1981.

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

[TPC 94] Transaction Processing Performance Council, *TPC Benchmark C, Revision 2.0, 20 October 1993*, and *TPC Benchmark D, Working Draft 7.0, 6 May 1994*, C/O Shanley Public Relations, 777 N. First St, San Jose, CA.

[van den Berg 93] J. van den Berg, D. Towsley, "Properties of the Miss Ratio for a 2-Level Storage Model with LRU or FIFO Replacement Strategy and Independent Reference," *IEEE Trans. on Computers*, 42(4), April 1993.