Computer
Sciences
Department

Goal-Oriented Memory Allocations in Database Management Systems

Kurt P. Brown

Technical Report #1288

September 1995

UNIVERSITY OF
WISCONSIN
MADISON

1288

GOAL-ORIENTED MEMORY ALLOCATION IN DATABASE MANAGEMENT SYSTEMS

By

Kurt Patrick Brown

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

(COMPUTER SCIENCES)

at the

UNIVERSITY OF WISCONSIN – MADISON

1995

Abstract

In order to meet the individual performance goals of each class in a complex multiclass database workload, today's database management systems require the adjustment of a number of low-level performance "knobs," such as buffer pool sizes, multiprogramming levels, data placement, dispatching priorities, etc. As the complexity of database systems is increasing, while their cost is declining at the same time, manually adjusting low-level DBMS performance knobs will become increasingly impractical. Ideally, the DBMS should simply accept per-class performance goals as inputs, and it should adjust its own low-level knobs in order to achieve them; this self-tuning capability is called *goal-oriented resource allocation.*

This thesis makes three contributions in the area of goal-oriented resource allocation for database management systems. First, it defines an overall architecture for goal-oriented resource allocation that includes techniques to insure a stable and responsive system and to accurately gather performance measurement statistics. Second, it presents an algorithm that can adjust per-class disk buffer allocation knobs in order to achieve performance goals for those classes whose performance is primarily affected by their disk buffer hit rates. Finally, it presents an algorithm for controlling the memory allocation and multiprogramming level for those classes primarily affected by their use of sort and join work areas; this algorithm is designed to work in conjunction with the disk buffer memory allocation algorithm in order to provide a comprehensive goal-oriented memory management solution.

Acknowledgements

Now I know why the Academy Awards are so boring. There are so many people who helped me get this thesis finished, that I'm tempted to just say something like "thanks to all of you beautiful little people out there." Instead, I've decided that everyone who helped me get to this point deserves their full name mentioned – and spelled correctly, to boot. Here goes...

First billing in the professor category goes to my advisor, Mike Carey. To me, Mike sets the gold standard in research. While I may not have mastered the use of the semi-colon or the transitional sentence as much as he would have liked, I hope that I've picked up some of his amazing dedication to thoroughness, proper scientific method, and tireless energy in getting to *really* understand a problem. Mike's legendary inability to get upset (except for the occasional editorial exclamation points) is another standard I'll be trying to achieve for a long time. I have also been fortunate to have Miron Livny as my co-advisor, sounding board, assumption-questioner, intellectual sparing partner, and free-lance performance expert. When they didn't end up with me in a Half Nelson, my discussions with Miron always helped to break through some barrier I was struggling with, and always resulted in better papers. In addition to running the "best database research group in the known universe," David DeWitt was a great source of advice, energy, inspiration, fun, motivation, money, and industry scuttlebutt. Yannis Ioannidis can take credit for getting me past the database qualifying exam, and is one of the top teachers I've ever had. In addition to begin a great instructor and research mentor, Jeff Naughton deserves recognition for having the best sense of humor on the seventh floor (and contrary to prevailing opinion, he writes very good code, as well). Mary Vernon deserves an award for teaching someone as probabilistically challenged as myself as much as she did about analytical performance modelling. Stephen Robinson deftly shepherded me through my Industrial Engineering minor, and graciously agreed to sit on my defense committee.

First prize in the fellow database grad student category goes to Manish Mehta, who collaborated with me on most of my research, and can take equal credit for the work presented in Chapter 5. Not only did I learn a lot about databases from Manish, but a lot about how to live life in general. Mike Franklin, Joey Hellerstein, Renee Miller, Hwee-Hwa Pang, Jignesh Patel, Praveen Seshadri, Valery Soloviev, and Odysseas Tsatalos deserve special mention for hours and hours of great collaboration, both geek-related and life-related. I am also indebted to fellow UW databasers Paul Bober, Shaul Dar, Craig Freedman, Eben Haber, Mark McAuliffe, Jussi Myllymaki, Brad Rubin, S. Seshadri, John Shafer, V. Srinivasan, S. Sudarshan, Manolis Tsangaris, Scott Vandenberg, Shivakumar Venkataraman, Seth White, Janet Wiener, Markos Zaharioudakis, and Mike Zwilling.

Kathleen Comerford, Susan Dinan, and Sheryl Pomraning deserve all the credit for shielding me and other unsuspecting database students from various evil bureaucratic tentacles, in addition to their considerable skills in taming the wild beasts of the seventh floor. Lorene Webber and Marie Johnson handle all the other university and grad school details with great skill and even better attitudes. Paul Beebe and his team in the Computer Systems Lab, especially Jim Luttinen and Mitali Lebek, provide the best computing services I've ever had the pleasure to use. Miron and his condor team made the experiments in this, and *many* other Wisconsin theses possible.

My IBM Poughkeepsie management gets credit for the extra years of my life that were added as a result of returning to grad school. First prize goes to Ross Mauri, for getting me into the awesome (and now defunct) Resident Study Program. Virginia Banks ties with Ross for first prize, since she helped my career just as much – and without me even noticing. Dick Mauceri and Judy Lauch carried me in their headcount for years in the hope that some day I might amount to something. Guy Lohman, John McPherson, and Pat Selinger were kind enough to fund my stay at Almaden for 10 very valuable months. I am also grateful to Ambuj Goyal at IBM Yorktown for the Research Initiation Grant that partially supported the work in this thesis.

Other IBMers to whom I owe include: Chamond Liu from IBM Corporate Technical Institutes and Ashok Malhotra from IBM Yorktown for writing two of my grad school letters (at great risk to their own reputations); Don Ferguson from IBM Yorktown for great advice and counsel on all things goal-oriented and otherwise; Manish Arya, Toby Lehman, Bruce Lindsay, Beau Shekita, Bob Yost, and especially Honesty Young from IBM Almaden for valuable aid, assistance, advice, and fun; Peter Gassner and Dave Simmen from DBTI, and Pat Gainer and Steve Schmidt from IBM Toronto for advice, counsel, and assorted goofiness.

Gerhard Weikum and Axel Moenkeberg influenced the work in this thesis from our interesting discussions at ETH in Zurich.

Other geek and non-geek friends are responsible for making my time in Madison (and San Jose) one of the best episodes in my life so far. It's hard to single out anyone in this list because everyone on it is equally important. They are: Lui Chan, Ritu Chander, Mark Craven, Susan Goral, Susan Hert, Christina Margeli, Tia Newhall, Janice Ng, P.S. Rajesh, Ranjani Ramamurthy, Brad Richards, Bill Roth, Adene Sacks, Todd Stern, Cheryl Thompson, Mary Tork Roth, and Martha Townsend.

Finally, I'd like to dedicate this thesis to my parents, Gwen and Richard, and to my sister Heidi. They deserve the *real* credit for getting me this far, and no words can ever repay them.

Contents

Chapter 1

Introduction

In the beginning, there was nothing. And God said ‘Let there be light.’

And there was still nothing. But, you could see it.

– Dave Weinstein

1.1 Background and Motivation

As database management systems continue to increase in function and to expand into new application areas, the diversity of database workloads is increasing as well. In addition to the classic relational DBMS “problem workload” consisting of short transactions running concurrently with long decision support queries [Pirahesh 90, Brown 92, DeWitt 92], we can expect to see workloads comprising an even wider range of resource demands and execution times in the future. New data types (e.g. image, audio, video) and more complex query processing requirements (rules, recursion, user defined operations, etc.) will result in widely varying memory, processor, and disk demands. The performance goals for each workload class will vary widely as well, and may or may not be related to their resource demands. For example, two classes that execute the exact same application and DBMS code could have differing performance goals simply because they were submitted from different departments in an organization. Conversely, even though two classes have similar performance objectives, they may have very different resource demands.

As an example, consider a three-class workload that consists 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”). Because a typical DBMS is tuned to optimize *system-wide* throughput or response time, the performance of each individual class in this example workload will be hard to predict. On the one hand, if the DBMS is designed or configured to allocate the maximum possible memory to sort and join work areas, then the decision support class may perform near its optimum and the TPC-A will likely suffer. One the other hand, if the DBMS favors disk buffer memory over the sort and join work areas, then the TPC-A class will perform near its optimum and the decision support class will be penalized.

In today's database systems, the goals for such a workload 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. As the complexity of database systems is increasing, while their cost is declining at the same time, manually adjusting low-level DBMS performance knobs will become increasingly impractical, as has been argued previously [Nikolaou 92, Brown 93b, Selinger 93, Weikum 93]. Ideally, the DBMS should simply accept per-class performance goals as inputs, and it should adjust its own low-level knobs in order to achieve them; this self-tuning capability is called *goal-oriented resource allocation* [Nikolaou 92].

Given a performance objective for each class in a multiclass workload, there are a number of mechanisms that a goal-oriented DBMS can use to achieve them: load control, transaction routing, CPU and disk scheduling, memory management, data placement, processor allocation, query optimization, etc. Each of these could be driven by performance objectives. A complete solution to the problem of automatically satisfying multiclass performance goals must employ more than one mechanism; each class can have different resource consumption patterns, so the most effective knob for controlling performance may be different for each class. The task for a goal-oriented DBMS is to determine the knob settings for each class that will enable it to achieve its goal, while at the same time providing the maximum amount of "left-over" resources for any class that has no specified goal (i.e. for "best effort" or *no-goal* classes).

1.2 Defining Workload Classes

As defined in this thesis, goal-oriented resource allocation is concerned not with the allocation of resources *within* a class, but *between* competing classes that represent different types of work. In order to allocate resources on a per-class basis, some mechanism must exist to map individual queries and transactions onto a set of workload classes. The processes of defining classes and mapping transactions to classes are critical enough to warrant a brief discussion here, although a detailed treatment is outside the scope of the thesis.

Taking the individual components of an aggregate workload and assigning them to classes is a well-known problem in the field of computer system performance analysis. It is normally performed manually by someone familiar with the workload and the computing system, i.e. a database or system administrator. While there are a wide variety of criteria that can be used in defining classes, Lazowska et al have provided a good summary [Lazowska 84]. They suggest that:

- Classes should consist of transactions that have similar service demands at each system resource. For example, I/O bound transactions should not ordinarily be in the same class as CPU bound transactions.
- Classes must distinguish workload components for which independent performance requirements must be satisfied. For example, if the response time of a particular transaction type is of concern, then it should not be grouped in a single class with other transaction types.
- Classes might be made to correspond to accounting groups or organizational units (e.g. by department or division).

Once the workload classes have been defined, some mechanism must exist to assign a class identifier to each individual database transaction. This process involves defining some function that takes one or more inputs and uses a set of installation-defined rules to generate the class identifier. Example inputs might be the userid or authorization id that submitted the transaction, some user profile information (department id, for example), the network node identifier that submitted the transaction, or a specific transaction or query plan identifier (for precompiled or stored queries). The major difficulty with mapping transactions to classes results from the fact that multiple subsystems need to share information that traditionally has been privately held (i.e. network data, TP monitor data, DBMS data, and operating system data). However, IBM's MVS operating system has provided goal-oriented resource allocation facilities for some time, and its interfaces for specifying goals and mapping transactions to classes serves as an existence proof that this problem can be solved [IBM 93c, IBM 95]. This thesis assumes the existence of similar mechanisms.

1.3 Criteria for Success

Before presenting new mechanisms for achieving multiclass performance goals, it will be helpful to define (abstractly) how these mechanisms should be evaluated. Simply meeting the performance targets for each class is not the only criteria with which to judge a goal-oriented resource allocation algorithm. The following criteria should be satisfied by any goal-oriented resource allocation algorithm before it can be considered for implementation in a real DBMS:

Accuracy: The observed performance for goal classes should be close to their stated goals. A convenient way to quantify accuracy is the *performance index* [Nikolaou 92], which is simply the observed performance metric divided by the performance goal. A performance index of one is ideal, while an index that is greater than or less than one indicates a violated or exceeded goal.

Responsiveness: The number of knob adjustments required to bring a class to its goal should be as small as possible, especially if the interval required between each knob adjustment is relatively long. A responsiveness criteria rules out simplistic exhaustive search strategies that can score high in accuracy, but that 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, all knobs should be left alone once the goals are achieved.

Overhead: A goal-oriented resource manager should minimize the extent to which it reduces overall system efficiency (i.e. its system-wide throughput rating, which is a measure of the system's total capacity for work). 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 degree of 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, 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 directly 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.

Practicality: A viable algorithm should not make 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, nor should it assume that 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.

1.4 Thesis Contributions/Organization

As mentioned earlier, a DBMS has many knobs that can be adjusted to achieve the performance goals for each workload class. Of these knobs, memory allocation is one of the most critical, even when memory is not the bottleneck resource, because it also the affects service demands at the processors and disks. This thesis therefore concentrates on three memory-related knobs: disk buffer allocation, working storage allocation and the multiprogramming level for working storage.

Memory is used for two main purposes in a DBMS: as *disk buffer* memory and as *working storage* memory. Disk buffer memory holds copies of disk pages in the hope that subsequent references to the same disk page in the future will be satisfied from the buffer pool instead of incurring additional disk I/Os. Working storage memory is defined as any memory used for query processing that does not hold copies of (permanent) disk pages. The two primary examples of working storage memory are sort and join work areas – the more memory allocated to these areas, the fewer the number of I/Os required by the sort or join algorithm.

Closely related to memory allocation is the choice of a multiprogramming level. The multiprogramming level (MPL) knob sets a limit on the number of transactions allowed to compete for memory (and other resources as well). An MPL limit is more critical for controlling the allocation of working storage memory than it is for controlling disk buffer memory; this is because disk buffer memory is usually shared among many concurrently executing transactions, while working storage memory is normally only utilized by a single transaction. Admitting an additional transaction that uses working storage memory therefore implies an increase in total memory consumption, while admitting additional transactions that primarily use disk buffer memory may only increase the utilization of existing disk buffer memory pages.

This thesis makes three contributions in the area of goal-oriented resource allocation for database management systems. First, it defines an overall architecture for goal-oriented resource allocation that includes techniques to insure a stable and responsive system and to accurately gather performance measurement statistics. Second, it presents an algorithm that can adjust per-class disk buffer allocation knobs in order to achieve performance goals for those classes whose performance is primarily affected by their disk buffer hit rates. Finally, it presents an algorithm for controlling the working storage allocation and multiprogramming level for those classes primarily affected by their use of working storage memory; this algorithm is designed to work in conjunction with the disk buffer memory allocation algorithm in order to provide a comprehensive goal-oriented memory management solution.

The remainder of this thesis is organized as follows: Chapter 2 describes the overall architecture for goal-oriented resource allocation. Then, a short detour is taken in Chapter 3 to describe the detailed simulation model used to evaluate the algorithms presented in subsequent chapters. Chapter 4 presents an algorithm, called *Class Fencing*, that controls disk buffer allocation; the performance of Class Fencing is evaluated using the simulation model described in Chapter 3. Chapter 5 then describes and evaluates the performance of an algorithm, called *M&M*, for controlling the memory allocation and multiprogramming levels related to working storage. Finally, Chapter 6 summarizes the thesis and points to areas where additional work is needed.

Chapter 2

Goal-Oriented Resource Allocation

Actually, my goal is to have a sandwich named after me.

– Unknown

This chapter presents an overall architecture for goal-oriented resource allocation in database management systems. First, it defines clearly what is meant here by specifying and achieving a performance goal. The components of the architecture are then described, and several techniques that are essential to providing a stable and responsive system are discussed. Finally, the chapter closes with a survey of related work. The architecture presented in this chapter will be used to develop techniques to control the memory allocation and multiprogramming level knobs in Chapters 4 and 5.

2.1 Specifying and Achieving Performance Goals

There are many possible ways to specify database system performance goals. A goal for a transaction class with very short (sub-second) response times is usually expressed in terms of average throughput (measured in transactions per second). On the other hand, performance goals for longer-running transactions, with response times in the tens of seconds or minutes, may be more naturally expressed in terms of an average response time. Response time metrics can be specified as average, maximum, or percentile values. Combinations of multiple metrics are also common, such as a target throughput that is subject to a maximum or a 90th percentile response time constraint. Following other work in this area [Nikolaou 92, Ferg 93], this thesis will adopt an average response time metric. Average response times are not only a commonly used performance metric in themselves, but they are also easily converted into average throughput metrics, given the number of attached terminals (clients) and their average think times.

Not all classes are important enough to justify a performance goal, however. Some work may be of a low enough priority that it should be performed only if excess resources are available after the goals are achieved for goal classes. This type of low-priority work is called, appropriately enough, a *no-goal class*. This thesis assumes that all such low-priority work is collected into a single no-goal class.

The remainder of this section discusses two additional issues related to goal specification. First, it defines the notion of an *observation interval* over which the average response time measurements are taken, then it discusses what should be done in the case when there are not enough resources in the system configuration to satisfy the goals. Finally, it closes by presenting a practical, "additive" approach for achieving per-class response time goals.

2.1.1 Observation Intervals

For any average or percentile metric, it is critical to specify the *observation interval* over which that metric is defined. That is, any such is meaningless unless it also specifies either the number of transactions that contribute to the metric or a time period over which the metric is computed. The desired observation interval is important because it determines the trade-off between stability and responsiveness. With too long an interval, the system will never react to workload changes, and with too short an interval, the system will react to natural statistical fluctuations between the transactions in a class.

In addition to specifying the desired trade-off between stability and responsiveness, another critical factor in choosing the observation interval is the amount of variance between transactions of a class. The greater the variance, the larger the observation interval should be in order to ensure a statistically valid measurement. Obviously, as more diverse types of transactions are included in a class, the response time variance within that class will increase. Ideally, the system should provide a high level "sensitivity" knob to allow the administrator to choose the appropriate balance between stability and responsiveness; the sensitivity setting would be combined with the observed variance in class response times in order to determine the appropriate observation interval. This thesis, however, will treat the observation interval as an input, ignoring the question of whether it is specified manually or with some higher-level mechanism.

Note that in the extreme case of a *maximum* response time goal (i.e. a "100th percentile" goal), the observation interval is equal to one transaction. An observation interval of one essentially implies that the system is to behave as if it were a real-time DBMS, where each *individual* transaction of a class has a performance goal, as opposed to having a longer-term goal for a *class* of transactions. Mechanisms very different from the ones presented in this thesis are required for real-time database systems [Abbott 91, Pang 94b]. Perhaps the key difference between goal-oriented and real-time systems is that goal-oriented systems have an observation interval greater than one transaction; this allows goal-oriented systems to violate goals on individual transactions and still meet their performance targets because they can always "make up" for violations by exceeding the goals on subsequent transactions within a single observation interval. In this thesis, observation intervals will normally be no smaller than the number of transactions required to achieve a statistically significant sample.

2.1.2 Degraded Versus Non-Degraded Modes

If the system configuration is not powerful enough to satisfy the performance goals for all classes in steady state, then it is said to be operating in *degraded mode* [Nikolaou 92]. This thesis concentrates primarily on *non-degraded* modes of operation, and it does so for two reasons. First, if the specified goals are not realistic for the configuration, then either the configuration should be upgraded or the goals should be relaxed; it makes no sense to persistently demand performance objectives that are impossible to achieve. Second, the problem of resource allocation in degraded mode is, in reality, quite different from that of non-degraded mode. The research literature on multiclass resource allocation has proposed methods for distribution of scarce resources that are based on the notion of uniform performance degradation across all classes, either relative to some theoretical optimal performance [Carey 85, Mehta 93, Davison 95] or relative to explicitly stated performance goals [Nikolaou 92, Ferg 93, Chung 94]. However, it is likely that administrators will want much more control in determining how much each class suffers in a degraded mode of operation [Pang 95]. For example, they may want to order classes by their perceived importance so that more important classes receive whatever resources are available and only the less important classes suffer [Nikolaou 92]. More well-understood priority-based allocation techniques can be used to solve this problem (e.g. [Carey 89, Jauhari 90a, Jauhari 90b]).

Even if one assumes a non-degraded steady-state mode of operation, of course it is still important to be able to *detect* unachievable goals. It is not uncommon for a system's workload demands to increase slowly over a period of weeks or months, and it would be valuable warn the administrator when this has occurred (or appears likely). In addition, if the administrator is not very familiar with the workload, it would be helpful to provide some feedback from the DBMS about whether the goals can be achieved or not, as otherwise it would be difficult to determine who was at fault (the administrator, for setting unrealistic goals, or the system, for failing to achieve perfectly reasonable goals). In summary, this thesis takes the approach that there are normally enough resources available to satisfy the goals, but that the system should identify those cases where they cannot be met. Beyond notification, no provisions are made for degraded mode operation (though an industrial-strength implementation of a goal-oriented DBMS should include some priority-based resource allocation mechanism to handle this case).

2.1.3 Achieving Performance Goals

As the following section will make clear, developing resource allocation mechanisms that can achieve per-class average response time goals is a very difficult problem. In order to simplify the problem, this thesis adopts the following practical approach. Rather than developing new resource allocation mechanisms from scratch, the approach taken here is to develop techniques that are *additions* to existing DBMS allocation mechanisms (which are primarily concerned with the efficient management of each resource). If the existing allocation mechanisms cause a class to violate its goal, then the goal-oriented algorithms will kick in and increase the class's allocation until its performance index reaches one. Any class whose allocation has been increased in this manner is not allowed to exceed its performance goal, since this may place the goals for other classes in jeopardy and/or unnecessarily degrade the response time of any no-goal class. Thus, if the performance index ever drops below one for a class whose allocation has been increased in this manner, its allocation will be reduced until its performance index returns to one. On the other hand, if the existing DBMS allocation mechanisms allow a class to "naturally" meet *or exeed* its goal, then nothing is done to modify its allocation. If all classes are meeting or exceeding their goals, then the goal-oriented algorithms will never try to redistribute resources to achieve some secondary objective (such as insuring that all classes are exceeding their goals by the same percentage, for example). Such an "additive" approach allows a goal-oriented DBMS to be built with a minimal amount of effort.

An additive approach implies that the definition of *achieving* performance goals means only that all goal classes experience average response times that are less than or equal to their goal (i.e. their performance indexes are less than or equal to one). The no-goal class response time may or not be minimized under such a definition. While a reasonable effort is made to prevent unnecessary degredation of the no-goal class response time – by insuring that violating classes never be given more resources than they need to achieve their goal – no extra effort is expended to reassign resources from a *naturally* exceeing class to the no-goal class. If the no-goal class response time under the additive approach is truly perceived to be inadequate, it can always be assigned a goal; it will then become eligible to receive any excess resources that may be owned by classes that are exceeding their goals.

2.2 A Goal-Oriented Resource Allocation Architecture

The objective of a goal-oriented DBMS is to find the combination of n resource allocation knob settings $< k1_c, k2_c, k3_c, ...kn_c >$ for each class c that will allow every class to achieve its goal. Finding such a set of knob settings is a difficult task for a number of reasons, with the foremost being the *interdependence* between classes. Classes are interdependent because their response times are determined not only by their own knob settings, but also by the amount of competition that they experience at shared resources (processors, memory, disks, locks, etc.). The amount of competition experienced by a class is determined by the knob settings of *all other* classes. Thus, the response time of any given class is determined both by the setting of its own knobs and by the settings of all other classes as well. More formally,

$$resp_c = f_c(\overrightarrow{k1}, \overrightarrow{k2}, \overrightarrow{k3}, ... \overrightarrow{kn})$$

where $\overrightarrow{ki}$ is a vector that represents the settings of the ith knob for every class. Note that since each class has unique resource consumption patterns, each class has its own unique response time function f_c,

Ideally, it would be possible to derive the response time functions (the f_c's) for each class and then use these functions together with established mathematical optimization techniques in order to determine the $\overrightarrow{ki}$ vectors that will satisfy the goals for all classes and minimize the no-goal response times. Unfortunately, deriving f_c for each class is beyond the current state of the art. While cost-based query optimizers have formulas that can be used to estimate processor and disk service times, these formulas offer no insight into the queuing delays that occur at the system entry point, the CPU, and the disks. Techniques from queuing theory could be applied to account for these delays, but predicting such delays even for a single hash join running alone on a centralized DBMS turns out to be non-trivial due to complexities such as caching disk controllers and intra-operator concurrency [Patel 93]. At best, the application of queuing theory to complex database workloads is a difficult open research challenge.

Because of the difficulty of accurately predicting class response times as a function of resources allocated, the only feasible approach is based on feedback. The general idea is to use the difference between the observed and target response time for a class as input to *controllers* that estimate the knob settings that are needed to bring the class closer to its response time goal. These estimates are repeated again and again until the class is either brought to its goal or it can be determined that the goal is impossible to achieve. One simplistic technique that a controller could use is to exhaustively search the entire solution space, trying every possible knob combination. An exhaustive approach may actually be feasible if the search space is small, but quickly becomes too time consuming in the case of multiple knobs (where there can be hundreds or thousands of possible combinations of settings). The trick is to design controllers that can bring a class close to its goal as quickly as possible while still behaving in a stable manner. Chapter 4 is devoted to developing such a controller for the buffer memory allocation knob, and Chapter 5 presents a controller that handles memory allocation and multiprogramming levels for working storage. The remainder of this chapter describes the design principles and features that are common to both of these controllers.

2.2.1 Per-Class Versus System-Wide Orientation

There are two possible ways to structure a feedback-based goal-oriented resource allocator: either with a *system-wide orientation* and or with a *per-class orientation*. A system-wide orientation means that a controller is activated on a global basis (e.g. every minute or so, or in response to some system-wide event) and, once activated, takes actions based on an analysis across *all* classes. The advantage of such an approach is that it provides the potential for dealing with the interdependence of classes; changes can be made to the system "as a whole." The disadvantage of a system-wide orientation is that it requires, after any resource allocation change, a sufficient waiting period to elapse in order to let the entire system "settle" to a new steady state. This requirement effectively ties the responsiveness of a system-wide algorithm to the slowest-moving class in the system (i.e. the one with the lowest throughput).

In contrast, a per-class orientation means that the algorithm is activated for each class on a time frame that is specific to that class (e.g. the specified observation interval for the class). Once activated, its actions are oriented toward a specific class and are based largely on an analysis of that class in isolation. The advantage of a per-class orientation is that it treats each class independently, allowing fast moving classes to respond quickly without being tied to the behavior of slower classes. Decoupling classes from each other by using a per-class orientation is especially important for complex database workloads, where response times can easily vary by three or four orders of magnitude across classes. The disadvantage of a per-class orientation is that it completely ignores the interdependence between classes.

Despite its disadvantages, this thesis adopts a per-class orientation because of its superior responsiveness.

Additional heuristics are used to compensate for the insensitivity of this approach to inter-class dependencies. Because it ignores inter-class dependencies, a per-class approach greatly simplifies the controller design problem; instead of having to find the $\overrightarrow{ki}$ vectors that achieve the goals for *all* classes in the system, we can independently search for each class's solution (i.e. a $< k1_c, k2_c, k3_c, ...kn_c >$ set that achieves its goal).

To summarize the architecture so far, we advocate an independent feedback controller working on behalf of each goal class. This controller compares the observed average response time for the class against the response time goal after every observation interval. If the class is in violation, it will adjust one or more resource allocation knobs for the class in order to bring it closer to its goal. If the class is meeting its goal, or is exceeding its goal "naturally" using only the underlying DBMS resource allocation policies, nothing is done. If a class is exceeding its goal and its resource allocations have been adjusted, then its allocations are reduced in order to bring the class closer to its goal. Finally, because of the interdependence of classes that share resources, a class's allocation may have to be adjusted to ensure that another class is able to achieve its goal as well (as will be seen in Chapter 5).

Implicit in this architecture are four basic tasks that must be performed for any class. The first is measuring observed response times and any other statistics (e.g. buffer hit rates, queue lengths, device utilizations, etc.) that are required by the controllers; the second is determining when goals are being met, exceeded, or violated; the third is determining which knob(s) should be turned to control the performance of the class; and the forth is determining exactly how to turn the specific knob or knobs. The last task (turning the knob) is specific to the particular resource being controlled, while the first three tasks are common to any controller regardless of what resource it is controlling. These three common tasks will now be described in the following subsections.

2.2.2 Statistics Measurement

The key challenge in statistics measurement is determining *when* to measure them. Because multiclass database workloads are extremely dynamic, measuring them at the wrong time can result in a biased measurement for two reasons: as a result of *state transitions* caused by a change in resource allocation, and as a result of *natural statistical fluctuations* between the individual transactions of a class (that would occur even if all resource allocation knobs remained untouched). One example of a state transition would be the change in average queue lengths at the system entry point, processors, or disks when multiprogramming levels are changed – in this case, no measurements should be taken until the queue lengths stabilize once again.

To avoid measuring state transitions, each class can be treated as a finite state automata (FSA) with well-defined states and transitions between them. The FSA for each class will depend on the particular knobs used to control its performance, but many states are common to all classes regardless of how they are controlled. We discuss some typical states and transitions here, postponing detailed descriptions of the specific FSAs until the controllers for disk buffer and working storage memory are described in Chapters 4 and 5.

- **Warmup**: In this state, the class is waiting for warm-up transients to dissipate either after a cold start or a reset of the goal-oriented allocation mechanism. All classes enter the warmup state on system initialization or reset. After either a fixed time period or some system-defined event that signifies the end of warmup (e.g. the disk buffer becomes full and/or some threshold of files and indexes have been opened), all classes leave the warmup state simultaneously and move to the *history build* state. No action is taken on this transition except to reset all class 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 (e.g., due to a recently changed resource allocation knob). Class statistics are reset on entry to this state and then accumulated until the next state transition. The time spent in the history build state is equal to the length of one observation interval; if response time goals are being met at the end of the interval (or are being exceeded "naturally"), then the class is moved to the *steady state*, otherwise the class's resource allocations will be adjusted, statistics are reset, and the class moves to either the *transition up* or *transition down* states.

- **Transition Up**: A class enters the transition up state if any resource allocation was increased in order to satisfy its goal. This state represents the point in time between when a resource allocation target has increased and when the class has actually adjusted to the new allocation. For example, when a buffer memory allocation target increases, some number of buffer faults must occur in order for a class to accumulate the newly allowed memory. Similarly, when a multiprogramming level is increased, it will take some time for system entry point queue lengths to decrease to a new mean length. A class is moved to the *history build* state upon exit from transition up; no action is taken except to reset all statistics.

- **Transition Down**: This state is similar to transition up, but is entered when resources are decreased. Transition to this state is not necessary in all such cases, however. For example, disk buffer or working storage memory frames can be immediately removed from a class without any time lag. As was the case for transition up, a class is moved to the *history build* state upon exit from transition down; no action is taken except to reset all statistics.

- **Steady State**: A class enters this state when its response time goals are being met (or exceeded "naturally"). The goals are checked again after one observation interval; if they are still being met, then this state is entered again for another observation interval. If the goals are not being met, resource allocations are adjusted, statistics are reset, and the class moves to the *transition up* or *transition down* states.

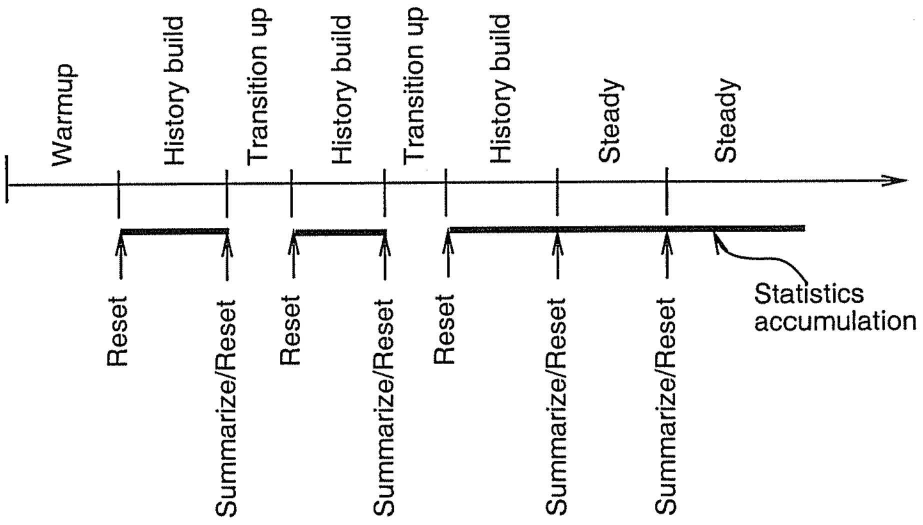

Figure 1: Example state change sequence

Figure 1 shows a possible sequence of state changes for a class over time (moving from left to right). The class starts in warmup state and passes through two knob increases (transition up states) and then meets its goal. In this example, the class spends less time in the transition states than than it does in the history build state. A relatively short transition time is common in the case of disk buffer classes; it may take only a few transactions to fault in enough disk buffer pages to exit the transition up state, whereas the time required in history build state depends on the length of the observation interval (which may require tens or hundreds of transaction completions). The horizontal bars underneath the timeline show when statistics are being accumulated (i.e. when individual transaction response times are being added to running totals used to compute averages, disk queue lengths are being sampled, etc.). At the points marked "reset," all of these accumulated statistics are thrown away and reset. The points marked "summarize/reset" are those points where summary statistics for the observation interval just ending are rolled up, resource allocations may be adjusted, and the statistics are then reset to start off the next observation interval.

While an FSA mechanism can be used to filter out unwanted state transitions from measurements, selective exponential weighting can be used to filter out statistical fluctuations. When statistics are summarized at the end of an observation interval, they are combined with past history as follows:

$$S^{new} \longleftarrow (1-\alpha)S^{prev} + \alpha S^{curr}$$

Here, S^{prev} is the value of a system statistic from the previous observation interval, S^{curr} is the new value as computed at the end of the current observation interval, α represents the percentage value of the present relative to the past, and S^{new} is the resulting weighted value. Based on a sensitivity analysis for a wide range of workloads and controller algorithms in later chapters, a value of .25 for α shows good performance across a wide range of workloads and therefore is adopted as a constant in this thesis. This is the same value that was used in the goal-oriented transaction routing algorithm of [Ferg 93].

Exponential weighting is ideal when a class is in steady state; in this case it is desirable to avoid resource allocation changes in response to the natural statistical fluctuations of a class. However, exponential weighting is not ideal once it is determined that a transition in resource allocation *is* actually called for. Because the burden of history can never be shaken off with exponential weighting, the measurement of a class that just changed its resource allocation may be skewed too much towards its behavior under the previous allocation. To deal with this problem, all history is dropped (i.e. reset) on entry to the history build state. The history build state thus signifies that a class has just completed some resource allocation transition and is now entering a new region of operation, rendering its previous history of no consequence. This type of "selective" exponential weighting gives a good combination of stability in the steady state and responsiveness in transition periods.

2.2.3 Checking Goals

After accurately measuring statistics, the second task common to any resource allocation controller is determining whether a class's goals are being satisfied or not. Due to the natural statistical variance in the response times of transactions within a class, the goals should not be considered satisfied only when the average response time *exactly* equals the goal, as this is unlikely to ever be achieved. Instead, goals are considered satisfied if the observed average response time for a class c is within plus or minus some percentage of the user-specified response time goal for c (i.e. within some tolerance band T_c of the goal). As is typical of any feedback mechanism, T_c turns out to be a sensitive parameter. If there is a large amount of natural statistical variance in the class's response times, T_c must be wide enough to prevent the algorithm from attempting to manage natural statistical fluctuations. However, a narrow T_c should be used with lower variances in order to reduce the number of interval response times that violate the goals. Figures 2 and 3 show how the tolerance for a class should be adjusted to account for the variance in class response times. Figure 2 shows a smaller tolerance band for a class with a moderate response time variance, and Figure 3 shows how this tolerance band must be widened to deal with a larger response time variance.

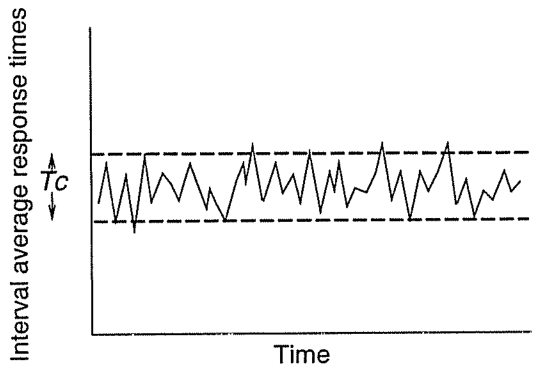

Figure 2: Moderate response time variance

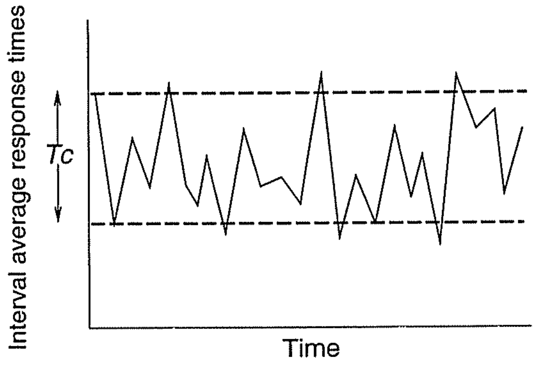

Figure 3: Larger response time variance

Because the value of T_c depends on the workload and the dynamic state of the system, it must be computed dynamically based on the observed standard deviation in response times across multiple intervals. Given a sufficient number of samples, the distribution of average interval response times can be approximated by a normal distribution. T_c is therefore set 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, care must be taken in the standard deviation calculation to avoid including any observations that occur during state transitions, as these observations would act to inflate the algorithm's estimation of the natural variance in the workload; T_c would otherwise become excessively large (loose). Therefore, observations are only added to the running computation of the standard deviation if a class has observed some consecutive number of steady state intervals. A default tolerance band of plus or minus 5% of the response time goal is used until T_c can be computed from actual response time observations. Like any other statistic, T_c is subject to selective exponential weighting.

In addition to ensuring that only the "natural" statistical variance is recorded in the standard deviation calculation, the standard deviation must also be recomputed after a class undergoes a resource allocation transition. This is because the existing sums and sums of squares used to compute the standard deviation are all relative to the previous resource allocation, and are therefore all relative to a different mean response time as well. Combining observations previous to the transition with observations after the transition would 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 value is used temporarily until there have been enough consecutive steady state intervals under the new resource allocation to allow the standard deviation and T_c to be recomputed.

2.2.4 Determining Which Knob to Turn

The final common controller task is determining which knob to turn if a class is not meeting its goal. Since this thesis is concerned only with memory management knobs, it suffices to place each of the workload's classes into one of two categories: disk buffer classes or working storage classes. If a class uses any working storage memory at all, it is considered a working storage class, and the working storage controller is responsible for its performance; otherwise it is considered a disk buffer class, and the disk buffer controller is responsible for its performance. This rudimentary approach obviously ignores those classes in the "grey area" where either the disk buffer or the working storage knobs could be used to control their performance. Ideally, the knob with the "biggest bang for the buck" should be preferred for controlling the class. The techniques used to detect such a knob can be classified under the title of *bottleneck analysis*.

While bottleneck analysis is a challenging area for future work, this thesis is concerned with a much more basic question: can memory knobs be used to control the performance of multiclass database workloads in a way that satisfies the criteria laid out in Chapter 1? Only if this question is answered in the affirmative does it then make sense to delve into the more detailed issue of bottleneck analysis. Therefore, this thesis will adopt the simplistic method for the classification of workload classes as described above; issues related to bottleneck analysis will be discussed in the Future Work section of Chapter 6.

2.2.5 Architecture Summary

At this point, we review the major points of the goal-oriented resource allocation architecture that has just been laid out. Each class is treated independently in order to increase responsiveness and to simplify the problem of determining how to set each knob. Each class operates in a continuous feedback loop with well-defined states, and running statistics (response time, number of I/Os, etc.) are accumulated upon every transaction completion for a class. These statistics are accumulated until the class makes a transition to another state. At appropriate state changes, summary statistics are computed from the running statistics accumulated over the last observation interval, and they are selectively exponentially weighted with summary statistics from previous intervals. A dynamically varying tolerance band around the goal is used to determine if a class is meeting its goal or not. If the class is not meeting its goal, one of two controller algorithms (the disk buffer or working storage controller) is called to make a knob adjustment. The class is placed in a transition state if knobs have been adjusted, and is placed in steady state if its goals are being met. Occasionally, a class may be called upon to adjust its resource allocation in order to allow another class to achieve its goal – such adjustments are required because of the interdependence of classes that share common resources. The entire process just as described repeats indefinitely for every class.

2.3 Related Work

In this section we review the limited amount of previous work in the area of goal-oriented resource management.

2.3.1 The MVS Operating System

The earliest known attempt at goal-oriented resource management for multiclass workloads is IBM's MVS operating system [Lorin 81, Pierce 83, IBM 93c]. The System Resources Manager (SRM) is the component of MVS that is responsible for achieving goals, and like all other proposed algorithms, it is feedback-based. Unlike the architecture presented here, however, it uses a system-wide approach, analyzing all classes at once either on a timer basis or in response to certain system events. The responsiveness problems caused by a system-wide approach are mitigated by the fact that (until the latest MVS release) goals are specified in terms of desired *service rates* (i.e. a class should be able to consume some amount of memory, processor, and disk per unit time). The use of service rate goals frees the SRM from having to wait until a certain number of transactions complete in order to determine whether or not their goals are being satisfied. Unfortunately, service rate goals are much more difficult for an administrator to specify, as it is not at all clear how to translate a response time requirement into a specific set of service rates.

As of the latest MVS release, in addition to service rates (which are now called *velocity* goals [IBM 95]), average and percentile response time goals are now supported. Response time goals are recommended for those classes with a throughput high enough to insure at least 20 completions during the observation interval, and velocity goals are recommended for classes with lower throughputs. In addition, the concept of no-goal classes is now supported (in the form of *discretionary goals*), as is the specification of the relative importance of each class for use in allocating resources in degraded mode.

The MVS SRM has four primary knobs that it controls for each class: multiprogramming level, memory allocation (i.e. working set size), processor scheduling, and I/O subsystem scheduling. It uses a set of fairly simple heuristics to guide the controllers for these knobs [Pierce 83] – unfortunately, detailed information on the heuristics is not available since MVS is a commercial product.

Although it represents a significant example of related work, the MVS SRM is not the answer to the goal-oriented resource allocation problem for mixed database workloads. One of the primary tools that the SRM uses to control resource allocation is swapping processes (along with their virtual address spaces) into and out of memory. Swapping out an active transaction is an action that may not be desirable (or even possible) in the context of a DBMS, as transactions may need to be aborted in order to actually free up their resources. Since it is embedded in the operating system, the SRM does not understand database disk buffer or working storage memory, but instead uses memory allocation as a mechanism to control virtual memory paging rates. While it does not address DBMS knobs, the SRM has been evolving for nearly 20 years, and as such, it represents the most complete solution to goal-oriented resource allocation that exists today.

2.3.2 Goal-Oriented DBMS Research

The earliest published research paper on goal-oriented resource management in a database context was a pioneering paper from Christos Nikolaou's group at IBM Yorktown [Nikolaou 92]. This paper defined the problem of goal-oriented resource allocation, described alternative ways to specify goals, introduced the notion of performance indices, and described work in progress on the problem of goal-oriented resource management for distributed transaction processing systems. The work from this group spawned several algorithms that we review in this section and that influenced subsequent releases of MVS as well as IBM's CICS TP Monitor.

The first offshoot of [Nikolaou 92] was a pair of algorithms for goal-oriented transaction routing in distributed transaction processing systems [Ferg 93]. These two algorithms are feedback-based and use a system-wide orientation. Both algorithms attempt to predict the effect of a transaction routing decision on the response times of each transaction class. The inputs to the algorithms include the average processor, disk, and communication demands for transactions of each class, the number of transactions of each class running on each node, and the observed per-class response times on each node. These inputs are used to estimate the CPU queuing delays and response times that would result from a particular routing decision. A routing is then selected that minimizes the *maximum* performance index (observed response time divided by response time goal) for any class The objective of minimizing the maximum performance index implies that the algorithms do not have to predict specific response times very accurately. Rather, they need only determine the correct *relative* response times when comparing between different routing possibilities.

The second offshoot from [Nikolaou 92] was an algorithm, called *Dynamic Tuning* [Chung 94], for goal-oriented multi-class disk buffer allocation. Dynamic Tuning is also a feedback-based algorithm with a system-wide orientation (their system-wide observation interval is called a *tuning interval*). It operates by comparing the performance indices of each class, and it continuously shifts buffer frames from "rich" classes (those with the lowest performance index) to "poor" classes (those with the highest performance index). This type of "Robin Hood" resource transfer requires a system-wide orientation, as the measurements for all classes must be synchronized in order to insure an accurate system-wide assessment of the relative performance of each class. Dynamic Tuning avoids the aforementioned responsiveness problems of a system-wide orientation because its goals are specified with respect to individual buffer manager get/read page requests (as opposed to end-to-end transaction response times). Thus, the "response times" of all classes are of similar magnitudes (less than or equal to the time required to retrieve a page from disk). The specifics of Dynamic Tuning's controller design will be discussed further in Chapter 4.

2.3.3 Other Related Work

While it does not specifically accept response time goals, the adaptive memory allocation and MPL adjustment algorithm described in [Mehta 93] is relevant here because its objective of maximizing *fairness* is very close to the objective of the goal-oriented transaction routing algorithms described in [Ferg 93]. The adaptive algorithm computes a performance metric for each class which is the ratio of its observed average response time to its best possible response time (as would be obtained by running single queries of that class alone in the system); this is similar to a performance index. Fairness is then defined as the absence of variance in this metric across the set of all classes, so the adaptive algorithm's objective of maximizing fairness is similar to minimizing the maximum performance index[1]. The adaptive algorithm accomplishes its objective by dynamically determining the MPL limit for each class using simple heuristics that guide a feedback mechanism. A memory allocation for each class is then derived from the class's multiprogramming level using another set of heuristics. While the adaptive algorithm addresses memory allocation for purposes such as join hash tables and sort merge work areas, it assumes that all data is disk-resident and thus does not control the allocation of memory for longer-term buffering of disk pages. The adaptive algorithm is also feedback based and uses a system-wide orientation.

Another technique for allocating memory and controlling admission for multi-user query workloads is the dynamic resource broker approach of [Davison 95]. [Davison 95] describes two algorithms, $Broker_M$, and $Broker_M D$, that allocate resources to the highest bidding query operator ($Broker_M$ allocates memory only, and $Broker_M D$ allocates both memory and disk bandwidth). Both algorithms assign an amount of *currency* to each operator that directly reflects its ability to improve whatever system-wide performance objective is of interest. Not only are the admission and initial allocation of query operators determined by a bidding process, but their allocations may also be dynamically adjusted "in-flight" in order to insure that resources are always being used by the highest bidder (i.e. adaptive query processing algorithms [Pang 93a, Pang 93b, Davison 94] are exploited in this scheme). While both $Broker_M$ and $Broker_M D$ were shown to outperform the adaptive algorithm of [Mehta 93], it is not clear how such an approach could be used for goal-oriented allocation. Because of the difficulty of accurately characterizing response time functions, it would seem difficult to develop a bidding currency that would be able to achieve per-class response time goals.

Finally, the COMFORT project at ETH Zurich deserves mention since it was directed toward automated DBMS performance tuning [Weikum 93]. However, its emphasis was on self-tuning algorithms that optimized system-wide objectives, and it did not specifically address the problem of achieving per-class performance goals.

[1] A similar objective function was actually introduced much earlier, in [Carey 85], in the context of work related to load balancing for distributed database queries.

2.3.4 Today's State of the Art

In summary, we note that very few examples of goal-oriented resource management algorithms exist in the literature. Moreover, with the exception of the MVS SRM, the few existing examples all primarily control a single knob. In addition, they all use either prediction or heuristics to guide a feedback mechanism which sets the particular knob that the algorithm manages. The most comprehensive approach (the MVS SRM) is not directed toward a DBMS environment, and because it is part of a commercial product, detailed implementation information is not readily available. Clearly, if automated goal-driven performance tuning for database management systems is to become a reality, comprehensive algorithms need to be developed and evaluated.

The goal-oriented memory and MPL management algorithms presented in [Brown 93a], [Brown 94], and [Brown 95] represent a step in the direction of goal-oriented DBMS resource allocation. These papers form the basis for this thesis and will be presented in Chapters 4 and 5.

Chapter 3

Simulation Model

sim.u.la.tion n sim-y*-'la-sh*n

1 : the act or process of simulating : FEIGNING

2 : a sham object : COUNTERFEIT

– The Webster On-line Dictionary

This chapter provides a description of the simulation model that will be used for evaluating the goal-oriented resource allocation algorithms presented in the following chapters. Because the workloads and configurations required to evaluate the two algorithms are different from each other, this section will concentrate on those features of the simulated DBMS (and its underlying simulated hardware platform) that are common to both algorithms. A detailed specification of the workload and configuration parameters that are unique to each algorithm will be presented later, prior to the performance evaluation sections of Chapters 4 and 5.

3.1 System Configuration Model

The simulated DBMS used in this thesis models a multiple disk, PC-based or workstation-based uniprocessor server. The external workload source for the system is modeled by a fixed set of simulated terminals, so the simulator models a closed queueing system [Lazowska 84]. Each terminal submits a stream of transactions of a particular class, one after another. In between submissions, each terminal "thinks" (i.e. waits) for a random, exponentially distributed amount of simulated time. In all cases, the number of terminals is chosen to provide average disk utilizations in the range of 50 to 60%.

The simulated hardware configuration contains eight disks that are modeled after the Fujitsu Model M2266 (1 GB, 5.25") disk drive [Fujitsu 90]. While the simulated disks include a model of the actual Fujitsu disk cache, the simulated disk caches are disabled in this thesis as a result of our prior experience in prototyping goal-oriented algorithms in DB2/6000 (IBM's relational database for Unix [IBM 93b]). This prototyping work showed that the simulator's disk cache hit rates were much higher than those observed in the real system. The reason for this difference is that the simulator assumed that random, single-page disk accesses would bypass the cache and thus not pollute it with pages that are unlikely to be reaccessed, instead allowing the cache to be mostly dedicated to the prefetching of sequential disk scans. Unfortunately, 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 does not always occur 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 and disable the caches on all simulated disks.

The system's simulated 30 MIPS CPU is scheduled using a round-robin policy with a 5 millisecond time slice, while the disk queue is managed using an elevator algorithm. The buffer pool consists of a set of main memory page frames of 8K bytes each. The buffer manager is modeled after that of DB2/MVS [Teng 84, IBM 93a]. Thus, it utilizes separate LRU chains for sequential and random accesses, and it 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 (four or eight) 8K pages 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. The disk I/O subsystem supports blocked I/O for prefetch requests, i.e. it can concatenate physically adjacent disk blocks and treat them as one disk request (saving both disk seeks and I/O initiation overhead). Only consecutive blocking is supported, however, there is no support for "scatter/gather" I/O in which the pages of an I/O block are not physically adjacent (which DB2/MVS *does* support).

A memory reservation mechanism allows query execution operators to reserve memory for their working storage needs, preventing such 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. Note that the same memory pool is used for both disk buffer and working storage memory here; this design choice will be discussed at some length in Section 5.1.1 of Chapter 5.

Table 1 summarizes the parameters of the simulated configuration that are common to both Chapters 4 and 5. The disk parameters were chosen to approximate those of the Fujitsu Model M2266 disk drive, as stated earlier.

Parameter	Value
Number of CPUs	1
CPU speed	30 MIPS
Number of disks	8
Page size	8 KB
Memory size	24 MB (chap 4), 8 or 64 MB (chap 5)
Prefetch block size (# pages)	8 (chap 4), 4 or 8 (chap 5)
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 1: Simulated instruction counts

3.2 Database Model

The database is modeled as a set of data files (relations), some of which have associated B+ tree indices. These files and indices are modeled at the page level; an extent-based disk storage allocation scheme is assumed, and the B+ tree index pages can be laid out to represent either a clustered or non-clustered index. All database files are fully declustered [Livny 87] over all disks in the configuration (except for those files with fewer pages than there are disks). Detailed descriptions of the file sizes and the types used in subsequent performance experiments will be presented in the performance evaluation sections of Chapters 4 and 5.

3.3 Workload Model

The simulated workloads used in the performance evaluation sections of Chapters 4 and 5 are various combinations of single-tuple index selects, full file scans, index scans, index nested-loop joins and hybrid hash joins [DeWitt 84]. Since the simulator used in this thesis was originally built to model a parallel shared-nothing database system, all operators in a query tree run in parallel within their own lightweight processes and communicate with each other using a message passing paradigm. In this thesis, however, only a single node system is used, so all inter-process messages are bypassed by copying them directly from the sending buffer into the receiving buffer. Table 2 shows the simulated instruction counts used in experiments throughout this thesis; they are based on measurements taken from the Gamma parallel database system prototype [DeWitt 90].

Function	# Instructions
read a record from buffer page	300
write a record to buffer page	100
insert an entry in hash table	100
probe a hash table	200
test an index entry	50
copy an 8K msg	10000
start an I/O	5000
apply a predicate	100
initiate a select/scan	20000
terminate a select/scan	5000
initiate a join	40000
terminate a join	10000

Table 2: Simulated instruction counts

Chapter 4

Disk Buffer Memory

If you want to eat hippopotamus, you've got to pay the freight.

– anonymous IBMer on why IBM softwares uses so much memory

In this chapter, a disk-buffer memory controller algorithm called *Class Fencing* is presented. First, Section 4.1 reviews two previous goal-oriented disk buffer memory allocation algorithms, *Dynamic Tuning* [Chung 94] and *Fragment Fencing* [Brown 93a], highlighting both their features and their limitations. Section 4.2 then presents the Class Fencing algorithm. Class Fencing is based on a concept called *hit rate concavity*, which allows it to be more responsive, stable, and robust (as compared to the previous algorithms), while remaining relatively simple to implement. Section 4.3 describes the simulated workload that is used to evaluate the performance of Class Fencing, and the evaluation itself is presented in Section 4.4.

4.1 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 target allocation of buffer 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 it will be used again in Section 4.2 to explain the new Class Fencing algorithm.

4.1.1 Dynamic Tuning Description

The Dynamic Tuning algorithm [Chung 94] differs from other goal-oriented algorithms ([Ferg 93, Brown 93a, 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 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 then 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 has been 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, with one 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.

4.1.2 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 4 shows a simulated hit rate function for a multi-user index nested loop join workload (solid line). This line was derived by running such a workload over a range of different memory allocations using the detailed DBMS simulation model described in Chapter 3. Also shown in Figure 4 are two different "fittings" (the dashed lines) that were derived by taking a pair of points from the simulated hit rate curve and feeding them into the Belady equation as described in the previous section.

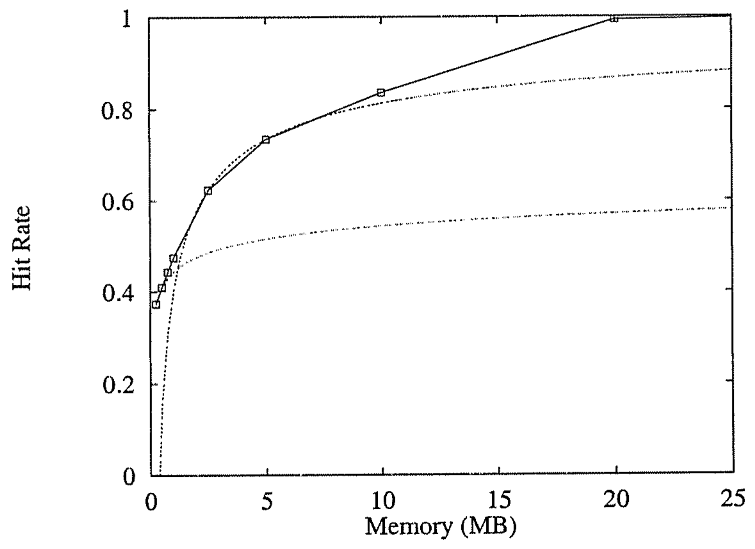

Figure 4: Curves from $1 - a/M^b$

Figure 4 shows that using the Belady equation to predict hit rates at larger memory allocations from observations made at smaller allocations can result in pessimistic estimates, i.e. the predicted hit rate can be much lower than what 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 what is actually needed. This will cause the algorithm to "overshoot" its target goal, and can create unstable 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, discussed next) 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 a high memory allocation 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

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

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].

4.1.3 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 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 the 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 93a], which is their size-normalized access frequency (in references per page per second). Using the uniform reference-based assumption that the hit rate on a particular fragment is identical to its fractional target residency, the fragments referenced by a class are processed in order from hottest to coldest by increasing the target residency for each one in turn until the hit rates for all fragments add up to the overall hit rate required by the class. The hit rate (and target residency) for a higher temperature fragment is increased to 100% before increasing the target residency of any lower-temperature fragment, thus at most one fragment referenced by a class will be partially memory resident. This entire process is reversed when a class's hit rate needs to be decreased: in this case fragments are processed from coldest to hottest, and target residencies are decreased rather than increased.

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 the associated fragment's 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.

4.1.4 Fragment Fencing Issues

A potential problem with a fragment-oriented approach, as noted in [Brown 93a], 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 a fragment's references are indeed 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 93a]; unfortunately, 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 in order to achieve aggressive response time goals, 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 [IBM 93b]) 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.

4.2 Class Fencing

This section describes Class Fencing, an improved goal-oriented buffer management algorithm. Class Fencing adopts the same response time predictor as Fragment Fencing (see Section 4.1.3), i.e. Class Fencing also assumes that miss rate and response time are proportional. Apart from using the same response time predictor, however, Class Fencing differs greatly from Fragment Fencing. The key difference is that, instead of building multiple fences to protect the individual database fragments referenced by a class, Class Fencing builds a *single* fence to protect a class's buffer pages – regardless of which fragment they belong to. In addition, Class Fencing uses a more general hit rate prediction technique based on a notion called *hit rate concavity*. Class Fencing's memory allocation mechanism provides 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, with a separate discussion of two more detailed aspects of the algorithm: estimating memory allocations in the presence of data sharing and computing memory usage statistics. Finally, this section closes with a state transition diagram to summarize the states that a disk buffer class goes through under the control of Class Fencing.

4.2.1 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, where 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, it will be argued 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", such as the one shown in Figure 5, in an optimal hit rate function. Any knee indicates a "mistake" in page replacement, i.e. it implies 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, one needs to know how close today's commercial DBMS replacement policies are to an optimal policy – i.e. when do they make mistakes? A DBMS should certainly 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 therefore prefetch sequentially accessed pages just before they are referenced, and once they are referenced, it can toss or retain these pages based on

[2]A similar theorem has been proven by van den Berg and Towsley [van den Berg 93] for the case of an IRM reference pattern coupled with an LRU replacement policy. 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].

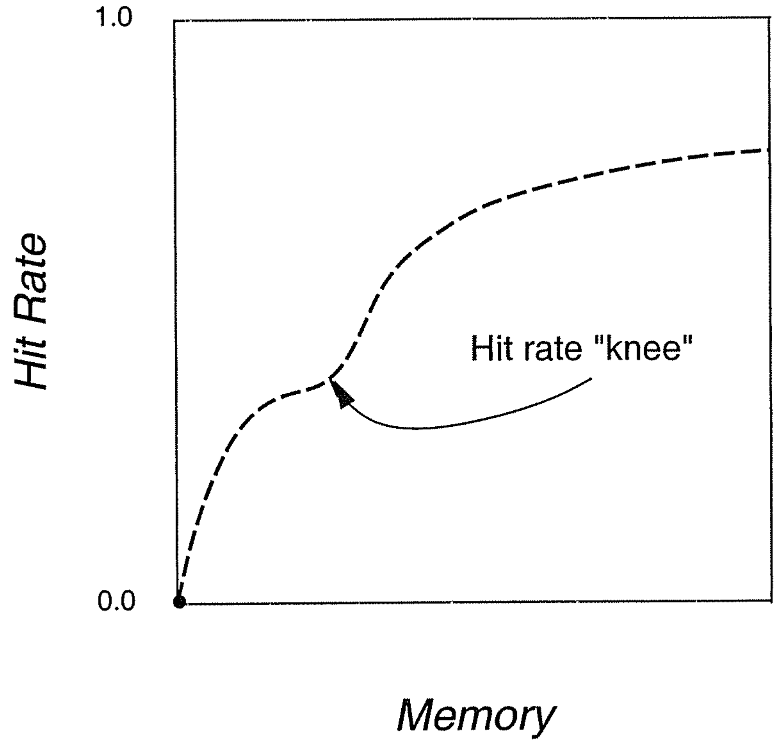

Figure 5: A hit rate function knee

knowledge of the total number of pages that will be scanned [Stone 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 93a].

While it would be impossible to offer any definitive statement about the likelihood of hit rate knees in real-world buffer managers, we conducted a small 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, the hit rate functions were mapped for both the TPC A/B and C benchmarks, as well as for all of the canonical database reference patterns documented in the DBMIN Query Locality Set Model [Chou 85]. None of the observed hit rate functions 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.

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

While acknowledging that hit rate function knees are possible in the real world, the evidence discussed above indicates that they represent pathological cases. Therefore, Class Fencing adopts 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; the impact of non-concave hit rate functions will be addressed after explaining how the concavity assumption is used by Class Fencing to estimate hit rates, which is the topic of the following section.

4.2.2 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 6 illustrates how the required buffer allocation for a class can be predicted with this approach.

The dashed curve in Figure 6 represents a hypothetical hit rate function for a class, over the range 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 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

[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].

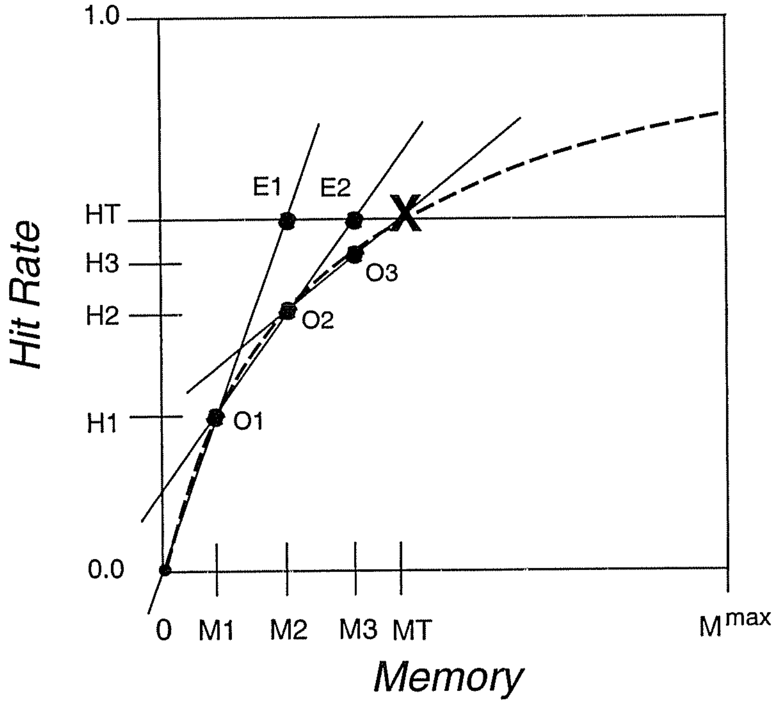

Figure 6: Estimating a concave hit rate function

rate is achieved. If any estimate's line were to cross 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 actual effect of a hit rate knee on Class Fencing's hit rate predictor depends on where the observation points lie 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 7. 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 ideal memory 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 upward direction. In Figure 7, for example, the next estimate after $E2$ would extrapolate between

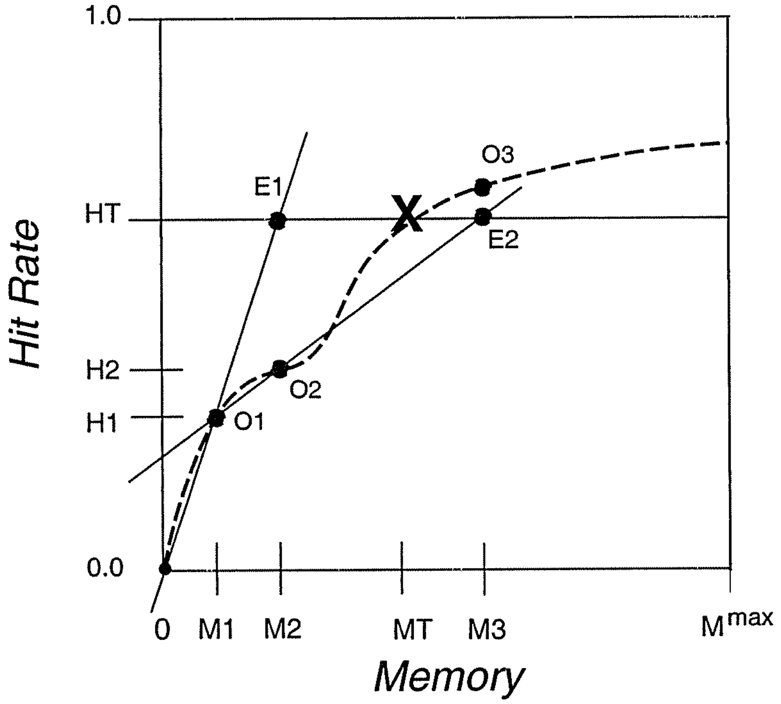

Figure 7: Overshooting a non-concave hit rate function

points $O3$ and $O2$. One more estimate would likely be required to achieve the goal.

4.2.3 Class Fencing's Memory Allocation Mechanism

Class Fencing's memory allocation mechanism represents 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 (called 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 particular replacement policy used 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 buffer allocation mechanism operates as follows. On a buffer miss by a violating class, a free frame is 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 that of a non-goal-oriented system.

[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 on the basis of classes and uses an identical replacement policy (the existing DBMS replacement policy) for each one.

[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 though, since they are likely to be referenced by (and perhaps reassigned to the buffer manager for) another class.

4.2.4 Class Fencing Details

There are two additional details that need to be addressed to complete the 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 necessarily 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, this means that 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. Maintaining $inUse[C]$ can be accomplished by using a bit map for each buffer frame (with one bit per class); if class C's bit is off when a buffer frame is referenced, then $inUse[C]$ is increased by one. When the page residing in a buffer frame is removed, then the $inUse$ counts for any class whose bit was on for that frame are decreased by one.

The percentage p of non-local buffer frames is used to translate total memory usage into pool size as follows. 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 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 6 and 7 should actually be interpreted as the time-weighted count 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 each page-in or page-out of a frame used by a class.

4.2.5 Class Fencing State Transitions

Figure 8 shows the states that a disk buffer class moves through under the control of the Class Fencing algorithm. Ideally, the transition down state never occurs except when there is a reduction in overall system load. If it occurs during the initial search for a solution, then this means that Class Fencing overshot the memory allocation, perhaps because of a convexity in the hit rate function or some other estimation error (e.g. an error in estimating a target hit rate from observed response times, or in estimating a pool size that will result in a target memory allocation).

Note that if Class Fencing determines that goals are unachievable, a transition is made to steady state (as opposed to some specific "unachievable" state). This is done to insure that the goals are continuously checked, in case the system dynamics eventually change such that the goals *can* be achieved. Thus, when a disk buffer class is in steady state, it may actually be meeting, exceeding, or violating its goals.

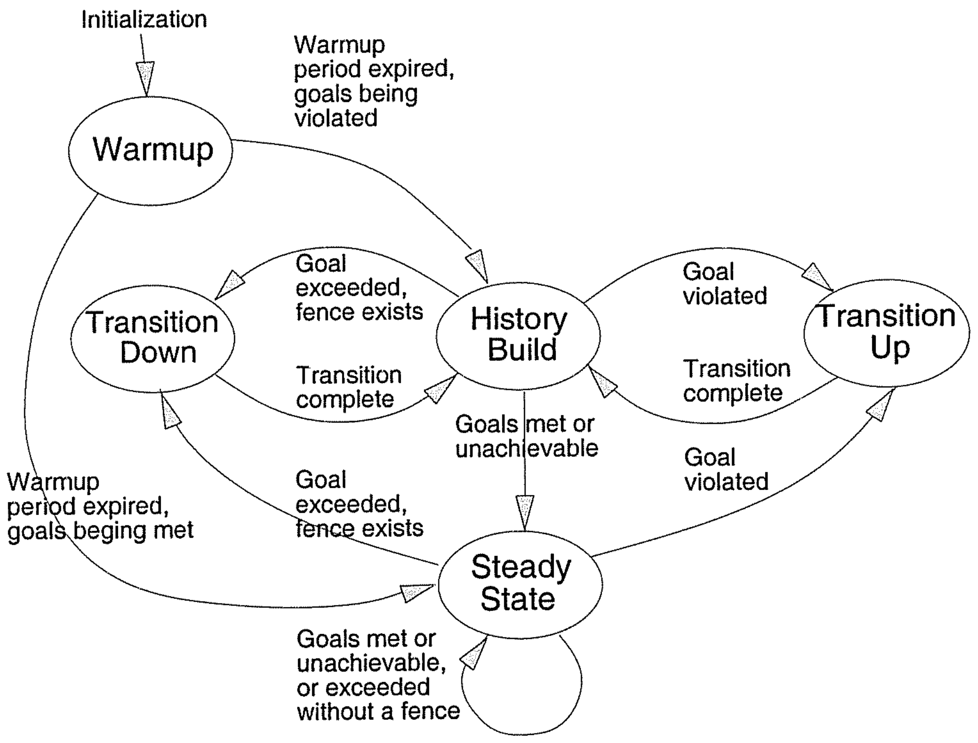
Initialization
Warmup
Warmup period expired, goals being violated
Goal exceeded, fence exists
Transition Down
History Build
Goal violated
Transition Up
Transition complete
Transition complete
Goals met or unachievable
Warmup period expired, goals beging met
Goal exceeded, fence exists
Goal violated
Steady State
Goals met or unachievable, or exceeded without a fence

Figure 8: Class Fencing states

4.3 Experimental Multiclass Workloads

This section describes the simulated workloads used to evaluate Class Fencing. These workloads were chosen to exhibit behaviors that represent challenges for previous goal-oriented buffer management algorithms, including complex and highly skewed page reference patterns and data sharing between classes. The individual classes differ greatly in their sensitivities to buffer hit rate, their degree of response time variance, and in their ranges of achievable response times (by up to two orders of magnitude). All the workloads in this section are formed using various combinations of three workload classes that have been previously defined elsewhere: the TPC-C benchmark from the Transaction Processing Council [TPC 94], and two query types from a previously published performance study of the DBMIN buffer management algorithm [Chou 85]. A 24 MB, eight disk version of the simulated DBMS described in Chapter 3 is used to run the workloads. Detailed descriptions of the database and workload classes are presented in the next two sections.

4.3.1 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 3: 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 3. While the choice of indexes is not specified in the benchmark, those listed in Table 3 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 pool that we use throughout the experiments in Section 5.5. 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 that 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).

4.3.2 Workload Model

The simulated workloads used in the experiments of Section 5.5 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. 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 3 (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 both is highly skewed, giving this workload a relatively high hit rate at a 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.

The TPC-C class is set up to exhibit a relatively high degree of response time variance: one of its five transaction types takes four times as long to execute as the other three, but only comprises 4% of the TPC-C mix. These heavier transactions will cause occasional spikes in the TPC-C class response times. In addition, the TPC-C class is configured with a relatively large number of terminals for this configuration (50), therefore it exhibits a fairly bursty arrival pattern due to its exponentially distributed inter-arrival times (think times).

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 the TPC-C class. The Q2 class is configured with a relatively small number of terminals (10) and exhibits a low degree of response time variance.

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 to be 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 rate than Q2. On the other hand, Q3 represents the class with the largest execution times (multiple tens of seconds, as compared to sub-second execution times for TPC-C). The Q3 class is also configured with 10 terminals and exhibits a low degree of response time variance.

The specific number of terminals and think times for each class are summarized in Table 4. These values were chosen to insure that the TPC-C class is the most aggressive consumer of disk buffer memory (i.e. that it has the highest page reference rate), while maintaining average utilizations of 50-60% across the eight disks.

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

Table 4: Workload parameter settings

4.4 Experiments and Results

In this section, the database and workload classes just described are used to examine how well Class Fencing can achieve a variety of goals for several different multiclass workloads, paying particular attention to its accuracy, stability, responsiveness, and robustness. The performance metrics used 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 1.3, and is a measure of accuracy. The number of knob turns is a measure of responsiveness. The response times of a no-goal class are also shown 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 average no-goal class response time. Selected transient analyses of response times are included to indicate how stable the algorithm is as a function of time.

In order to obtain statistically meaningful simulation results, the simulations in this section are run for 90 simulated minutes. Response time statistics are collected 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.

4.4.1 TPC-C and DBMIN Q2

The first set of Class Fencing experiments pairs the TPC-C and DBMIN Q2 classes together. Three variants of this workload are used: goals for Q2 only, goals for TPC-C only, and goals for both classes.

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 5 shows the results of this experiment. Each row in Table 5 represents a separate simulation run using a different goal for the Q2 class. The columns show the input goal for the Q2 class, the resulting average response time for Q2, Q2's performance index, the average TPC-C (no-goal) class response time, the number of knob turns (intervals) that it took to achieve Q2's goal, and the memory allocation chosen by Class Fencing 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 5 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 are to the different levels of Q2 performance (and memory allocation). Because the TPC-C class has

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

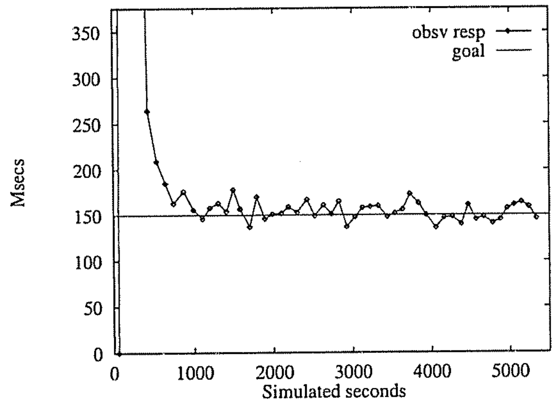

Figure 9: Q2 interval response times

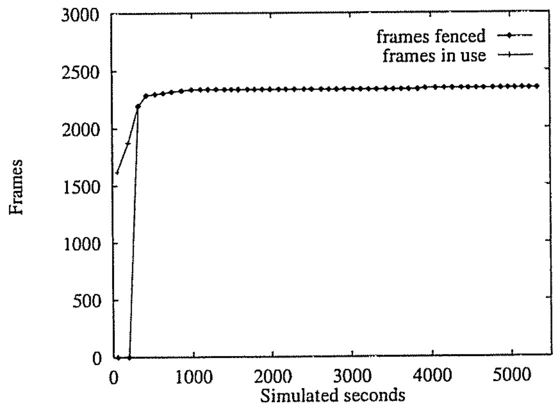

Figure 10: Q2 interval memory allocation

such high locality, large changes in its memory allocation have a minimal effect on its performance.

Note that Table 5'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 9 and 10, which show the Q2 class response time and number of buffer frames allocated as function of time for the 150 msec goal experiment.

Figure 10 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 9 and 10 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.

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

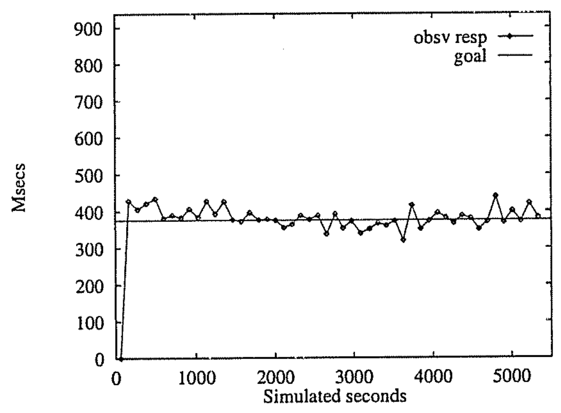

Figure 11: TPC-C interval response times

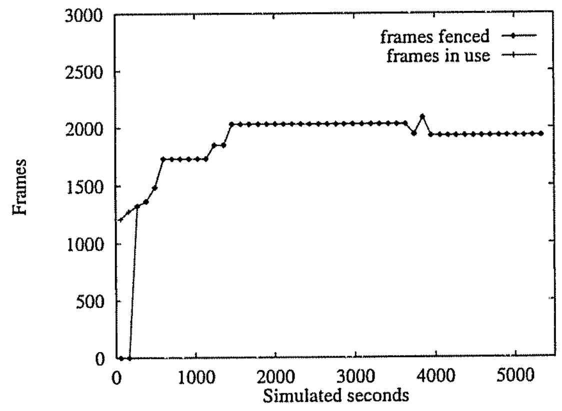

Figure 12: TPC-C interval memory allocation

Goals for TPC-C Only

The 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 6 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 no-goal class behavior of the previous example, Table 6 shows that the Q2 no-goal class response time is extremely sensitive to the memory allocation given to TPC-C. 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 11 and 12 show the transient behavior for the middle (375 msec) goal of Table 6. 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.

Goals for Both Q2 and TPC-C

The last experiment involving 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 (i.e. the problem would be "too easy" to solve, in a sense). 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 into 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.

Table 7 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 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, that are starred in Table 7: 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

TPCC Goal (sec)	Q2 Goal (sec)	TPCC PI	Q2 PI	# TPCC Knob Turns	# Q2 Knob Turns	No-goal Resp (sec)
0.300	25.0	1.03	0.97	4	0	21.4
* 0.300	20.0	1.03	1.10	4	1	27.1
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 7: TPC-C/Q2, with goals for both

can be seen by the poor no-goal class 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. These are cases where the system is operating in degraded mode because the goals are too aggressive for the configuration. Recall from Chapter 1 that the approach adopted in this thesis is to avoid attempts to reallocate memory in order to minimize the maximum performance index in degraded mode. As a result, the performance index for one class may remain higher than that for another class when the system is operating in degraded mode.[8]

4.4.2 DBMIN Q2 and DBMIN Q3

The second group of Class Fencing 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 the pages used by a class can reside both inside and outside its local buffer pool (this estimate was described in Section 4.2.4). As before, we experiment with three variants of this workload: goals for Q2 only, goals for Q3 only, and goals for both Q2 and Q3.

[8]One action that Class Fencing *could* take in degraded mode, however, is to prevent the execution of no-goal transactions altogether. This option is attractive because it is much less likely to harm overall system efficiency than any resource redistribution among executing classes. The Future Work Section of Chapter 6 will revisit this option in the context of other no-goal class enhancements.

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 8: Q2/Q3, with goals for Q2

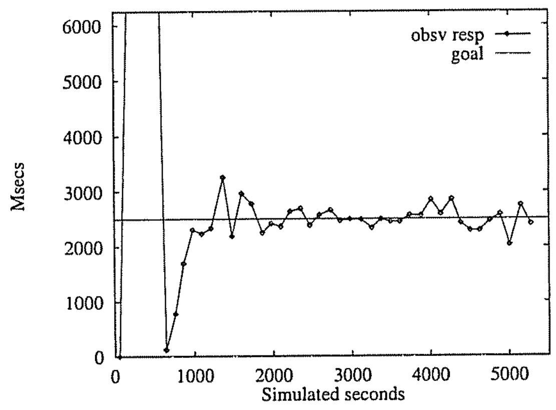

Figure 13: Q2 interval response times

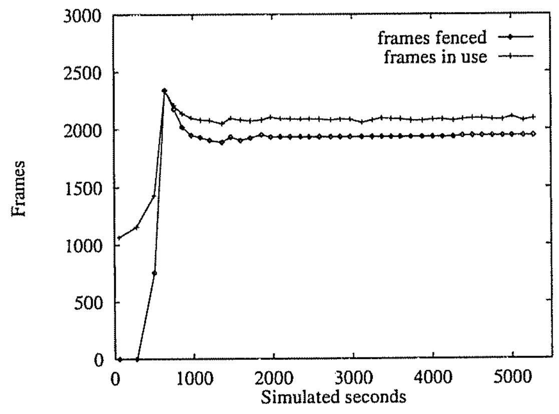

Figure 14: Q2 interval memory allocation

Goals for Q2 Only

Table 8 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. However, Class Fencing is not as responsive here as it was for the TPC-C/Q2 workload; it takes over 10 knob turns to find a solution in some cases. Figures 13 and 14 show the transient response times and memory allocations, respectively, for the 2.5 second goal experiment, which is the least responsive case. 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 a lack of concavity in the hit rate function, but rather to 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 13 and 14 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.

Goals for Q3 Only

Table 9 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 the 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.

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 9: Q2/Q3, with goals for Q3

Goals for Q2 and Q3

Table 10 shows the results of the final Class Fencing experiment, where goals are set for *both* the Q2 and Q3 classes. As before, another Q2-like class is added as a no-goal class in order to consume any resources left over when the combined goals 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, however, it too begins to search for its solution. Although it only took two knob turns to (slowly) find Q3's solution, the search for this 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).

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	20.0	1.03	0.99	2	2	20.7
20.0	10.0	1.06	1.11	2	2	22.7
20.0	5.0	1.04	1.01	1	6	23.9

Table 10: Q2/Q3, with goals for both

4.5 Summary

This Chapter has presented an algorithm for goal-oriented buffer management called Class Fencing. It introduced the concept of hit rate concavity and showed how this concept can be used to develop a simple, robust hit rate predictor that can deal with arbitrary page reference patterns and page replacement policies. A memory allocation mechanism was presented that exhibits low overhead, does not unnecessarily harm the overall system-wide hit rate, and can deal with data sharing between classes. Class Fencing was then evaluated over a range of workloads and goals and was shown to be accurate, stable, and highly responsive for these workloads. The key advantage of Class Fencing is a hit rate predictor that always predicts a *conservative* memory allocation; this allows Class Fencing to bring a class to its target response time very quickly by allocating memory in very large chunks without the fear of overshooting the goal.

Chapter 5

MPL and Working Storage

Fail not for sorrow, falter not for sin,

But onward, upward, till the goal ye win.

– Frances Anne Kemble

This chapter describes M&M, a goal-oriented controller for working storage multiprogramming level (MPL) and memory management. Before describing M&M in detail, two preliminary sections are presented. The first describes how classes controlled by M&M (*working storage classes*) can be made to coexist with *disk buffer classes* (i.e. classes controlled by Class Fencing) and with no-goal classes. The second preliminary section studies the effect of the working storage MPL and memory allocation knobs on response time; this study provides the background needed to develop the heuristics upon which M&M is based. Next, the heuristics and the M&M algorithm that uses them to set the memory and MPL knobs are presented, along with a novel technique that is used to set non-integral MPL limits. Finally, a simulated mixed workload of disk buffer transactions and working storage queries is used to evaluate M&M's performance[1].

5.1 Disk Buffer and Working Storage Coexistence

This section discusses the issues involved in integrating the two types of memory explored in this thesis, namely disk buffer and working storage, either of which can be used by goal and/or no-goal classes. The first issue discussed is how to divide up memory between disk buffers and working storage. Section 4.2.3 of Chapter 4 described an allocation scheme that divided up memory between different disk buffer classes using a *global buffer manager* to manage a *global pool* of memory frames together with class-based *local buffer managers* to manage *local pools*. Both goal and no-goal classes were allowed to use disk buffer frames from the global

[1]An earlier version of M&M that was paired with the Fragment Fencing controller [Brown 93a] for disk buffer classes was described and analyzed in both [Brown 94] and [Mehta 94].

pool, but if a goal class could not meet its goal simply by competing for frames in the global pool, a local pool was set aside for its private use in addition to any global frames it managed to obtain. Goal classes were also allowed to share disk buffer frames residing in the local pools of other classes. The next section describes how this scheme is extended to include working storage as well as disk buffer memory.

After explaining how working storage is integrated into the allocation scheme of Chapter 4, a mechanism for dealing with the interdependence between classes that arises from their competition for shared resources is then described. In particular, the mechanism concentrates on memory and disk resources, as this thesis is implicitly concerned with memory or disk constrained systems (since if the system is processor bottlenecked, memory allocation knobs will not be very effective).

5.1.1 Integrating Working Storage and Disk Buffer Memory

A key assumption of this thesis is that both working storage and disk buffer memory are allocated out of a *single* shared memory pool. Without this unification, it would be very difficult to manage the trade-off between the two types of memory usage. It should be noted that most commercial database management systems use separate disk buffer and working storage pools; in some cases the database administrator is provided with knobs for statically setting the size of both pools, while in other cases the DBMS simply allocates additional *virtual* memory for each individual working storage query. Static knobs provide some flexibility in controlling the relative performance of disk buffer and working storage classes, but like any static partitioning scheme, they can result in the underutilization of memory in an environment where the system load changes dynamically. Allocating virtual memory on demand for each individual working storage query is attractive because of its simplicity, but it essentially sweeps the problem of controlling the relative performance of disk buffer and working storage classes under the rug – the operating system's virtual memory page replacement policy will make an arbitrary (and perhaps unfortunate) trade-off between the two.

Like Class Fencing, M&M also associates a memory pool (of size $poolSize[C]$) with each goal class. However, in M&M's case, the size of this pool represents the amount of *working storage* memory required by the class to meet its goal (as opposed to Class Fencing's pool of *disk buffer* memory). The pool size varies dynamically in response to changing system loads and as each class compensates for interference caused by changes in other classes. Any memory remaining after subtracting all goal class (working storage or disk buffer) pools from the total available memory represents the global pool. Any increase in the pool size for a goal class, be it for working storage or disk buffer memory, is taken from the global pool, and any decrease is given back to the global pool. If the global pool is empty, no pool increases are allowed.

Unlike the local disk buffer pools for goal classes, working storage pools are *not* managed by a local buffer manager. Instead, frames for working storage are simply taken from the global pool (i.e. from the global buffer manager) and given[2] directly to the query operators that require them. Once they are given to a query operator, working storage memory frames are in an *unqueued and reserved* state, and do not belong to any buffer manager. When the operator is finished, it returns its working storage frames back to the global buffer manager. The global buffer manager is therefore the source for *all* newly allocated memory, be it working storage or disk buffers, for both goal and no-goal classes.

It is important to note that the pool boundaries for both disk buffer and working storage classes represent the memory required by an *average* transaction of a class in order to meet the class's goal, and that the actual number of frames used by a class at any *particular* point in time will vary randomly around the average (i.e. around its $poolSize[C]$). Disk buffer transactions can use frames managed by the global buffer manager and other local buffer managers, so disk buffer classes may actually have more frames in use than their pool size would indicate. Likewise, individual working storage transactions may have memory requirements that are either larger or smaller than the average for the class (upon which the pool size is based), and therefore the number of working storage frames used by a class at any particular time may be larger or smaller than its pool size as well. Because of the variance between the pool size and the actual number of frames owned by a class, there may be rare cases when the global buffer manager has no free (unfixed) frames available – in these cases, frames are stolen from local buffer managers in a round-robin fashion.

Memory for no-goal class transactions, for both disk buffer and working storage purposes, is allocated from the global pool on a first-come, first-served basis. A very simple working storage allocation policy is used for no-goal class transactions: If enough memory is available in the global pool to run a no-goal transaction at its maximum demanded working storage memory, then it is allocated its maximum. Otherwise, it takes whatever is available (down to its minimum requirement).

There are two problems with the approach described above. The first problem occurs when the goals are aggressive enough that the goal classes end up consuming most of the global pool. If the global pool drops below the minimum requirements of no-goal transactions, then they may be indefinitely postponed.

[2]The memory reservation mechanism described in Chapter 3 is used for this purpose.

This problem is solved by allocating a small portion of memory to insure that the *minimum* requirements of concurrently executing transactions can be met under normal operating conditions. This *set-aside area* is necessary to insure that MPL limits are the primary admission criteria, and not memory availability.[3]

The second problem is in some sense the "flip side" of the first: Allowing the no-goal class to consume the *entire* global pool can cause excessive memory waits for *goal class* transactions. Memory waits for goal class transactions should be avoided at all costs because they can *dramatically* increase the response time variance of a class, making it much more difficult to control. Consider the following scenario: Suppose that at a certain point, the global pool represents 50% of the available memory. An arriving no-goal query is admitted and allowed to consume the entire global pool to satisfy its working storage requirement. A goal class then decides to increase its pool size, and is allowed to do so immediately because, in theory, 50% of memory is available for this purpose. However, until the no-goal transaction finishes execution, the actual memory frames required to support the increase in the goal-class pool size will not be available. Thus, goal class queries admitted based on the increased pool size may be forced into memory waits until the no-goal transaction completes (which may be quite a while). The problem is that, unlike disk buffer memory, working storage memory can be "locked up" for the entire duration of a query operator; this phenomenon, combined with the variance in memory demands from transaction to transaction, can cause frequent over-commitment of memory.

One way to address the problem of goal-class memory waits is to implement *memory adaptive* query operators, such as those described in [Zeller 90, Pang 93a, Pang 93b, Davison 94], that can dynamically adjust their working storage requirements during the execution of the operation. Using these algorithms, the no-goal class can be "throttled back" to the new, smaller global pool size. However, because these memory adaptive mechanisms are not common in commercial systems as yet, this thesis will not assume their existence, and will adopt a more rudimentary solution in their place. Instead of allowing no-goal class queries to consume 100% of the global pool for working storage, they are instead limited to only 90% of the global pool. The no-goal class is still allowed to consume 100% of the global pool for disk buffer memory however, since disk buffer memory can normally be stolen back from the no-goal class immediately if required. While such a scheme will not prevent memory waits in all cases, it takes enough pressure off of memory demand that goal-class waits are significantly reduced. In addition, a new class state is defined, called *physical memory wait*, that prevents

[3]The actual size of the set-aside area is workload dependent. For example, each disk buffer transaction might require one or two pages and each working storage transaction might require 20-30 pages. These per-transaction minimums would then be multiplied by the estimated maximum MPL for each class to derive the total size of the set-aside area.

statistics collection during transient situations when there are not enough free memory frames to support a pool size increase. A complete state transition diagram for working storage classes is presented in Section 5.3.4 of this chapter.

5.1.2 Resolving Interclass Dependencies

While Class Fencing is an effective mechanism for meeting disk buffer class performance goals, there is a subtle problem with its approach when it operates concurrently with controllers for working storage classes. The basic premise of Class Fencing is that memory is the bottleneck resource, so it always tries to lower a class's pool size to the *minimum* possible amount that can achieve its response time goals (i.e. it favors "low memory allocation/high disk utilization" approaches to achieving goals). If the disks are the bottleneck resource, however, the high disk utilizations that result from this approach may prevent working storage classes from meeting *their* goals, regardless of what their own MPL and memory settings are. This situation is a classic example of an inter-class dependency.

Inter-class dependencies are accounted for in M&M by allowing the algorithm to modify Class Fencing's assumption that memory consumption must be minimized. M&M does this by requesting that one or more disk buffer classes enter an *exceed mode*. A disk buffer class in exceed mode will increase its pool size in order to increase its buffer hit rates and will therefore decrease its disk utilization. The class's pool size will continue increasing in small increments (5% of configuration memory) at each interval until one of two events occur: either disk utilizations are reduced to a point where they are no longer the primary reason for the goal violation of the working storage class (i.e. memory or MPL once again becomes the primary factor), or the global pool is exhausted (i.e. the request for a disk utilization reduction failed). As long as a disk buffer class is in exceed mode, Class Fencing will not force it to shed the "excess" memory that is causing its response time goal to be exceeded. If there are multiple disk buffer classes present in the workload, then they will all respond to the request for disk utilization reduction in parallel. Allowing multiple disk buffer classes to assist in lowering disk utilizations not only increases responsiveness, but acts to equalize performance indexes across the disk buffer classes as well (i.e. one disk buffer class will not be singled out to exceed its goal more than others).

5.2 The Effect of MPL and Memory on Response Times

This section explores the effect of MPL and memory allocation on the response time of working storage transactions. It assumes that an MPL knob exists for *each workload class*, as opposed to a single system-wide MPL knob that is set statically (where the latter is what most commercial database management systems provide today). Whereas the objective for setting a system-wide MPL knob is to find the "ideal" point between under-utilizing and over-utilizing DBMS resources, the objective for setting the per-class MPL knobs is not only to prevent over-utilization of resources, but also to achieve each class's response time goal as well.

Developing a controller that uses two knobs (memory and MPL) to control a class is much more difficult than developing one that adjusts only a single knob (such as those in [Brown 93a], [Mehta 93], and [Chung 94]). The search space for a single-knob controller is one dimensional; the only decision required is whether to turn the knob "up" or "down." With two knobs, the controller is faced with a two-dimensional search space. In order to move efficiently through this space toward a class's goal, the controller needs to have some idea of how the different points on a two dimensional <MPL, memory> grid relate to the class's response time. The remainder of this section explores this relationship empirically using a simulated multiclass workload. These simulations will provide some of the background information needed to understand the principles underlying M&M's two-dimensional <MPL, memory> controller.

The simulated workload used here, and throughout this chapter, is explained in greater detail in Section 5.4. For the purposes of this section, a brief overview will be sufficient. The configuration consists of a single 30 MIP processor, 8 MB of memory, and eight disks.[4] The workload consists of three classes: "queries," "transactions," and "big queries." The query class is a consumer of working storage memory. It consists of hybrid hash join queries [DeWitt 84] whose performance is related to the amount of memory allocated for their in-memory join hash tables. The sizes of the files referenced by the query class are chosen such that their join hash tables consume 20% of the configuration's memory at their maximum allocation.[5] The transaction class performs random single-record lookups on four files via B+ tree indices, and is therefore classified as a disk buffer class. Finally, the big query class is similar to the query class except that its file sizes yield hash tables capable of consuming 80% of the configuration's memory at the maximum allocation. Each class references

[4]The 8 megabytes of configuration memory is scaled up in later experiments.

[5]The maximum memory allocation for a hash join is defined as enough memory to hold a hash table representing the entire (smaller) "build" relation; this is approximately 1.2 times the size of the build relation, including data structure overheads. The minimum memory allocation is the square root of the maximum allocation.

its own unique set of database files, and all files are horizontally partitioned (i.e. fully declustered) across the eight disks.

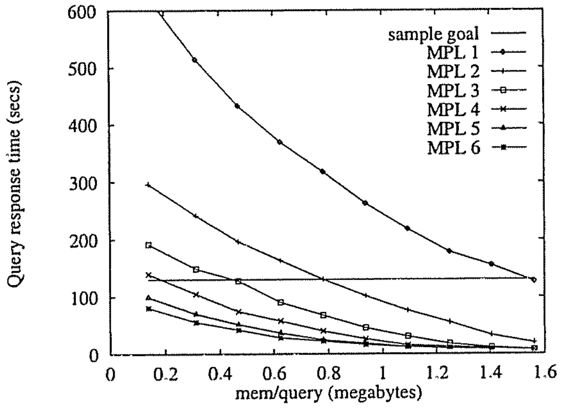

Figure 15: MPL and memory per query

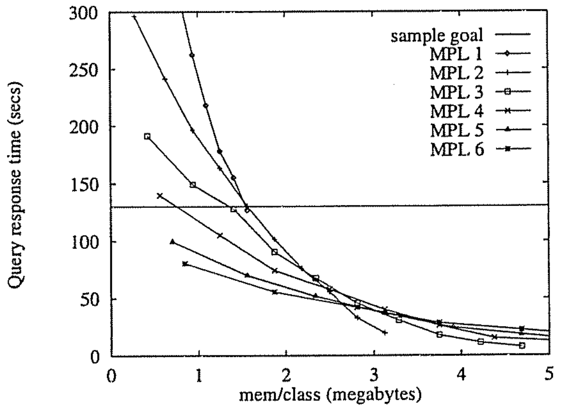

Figure 16: MPL and memory per class

Following a per-class approach, the experiments in this section will examine the effects of a range of MPL and memory combinations for the query class only, while ignoring the effect of these combinations on the transaction and big query classes. The big query class is set at an MPL limit of one query, which is allocated its minimum memory requirement, while the transaction class has no MPL limit and receives whatever memory is left over after the big query and query classes have been allocated their memory reservation requirements. Figures 15 and 16 show two different representations of the query class response times that result from various MPL and memory combinations. Figure 15 shows the average query class response time as a function of memory *per query* (with each query receiving the same allocation), while Figure 16 shows the average query class response time as a function of memory *per class* (i.e. of class MPL times memory per query). Query response times are shown on the y-axes (in seconds) and memory allocations are shown on the x-axes (in megabytes). Each sloping line in the graphs corresponds to a different MPL limit, and the straight horizontal lines represent a 130 second response time goal for the query class.

The most significant phenomenon shown by Figures 15 and 16 is the existence of *multiple solutions* to a particular response time goal. Each of the five points where the 130 second goal line intersects an MPL line represents a possible solution. Table 11 lists the characteristics of each of these solutions. While all of the solutions are equivalent as far as the query class is concerned (since they all achieve its 130 second goal), each one represents a different trade-off between memory and disk consumption by the query class. For example, the first solution in Table 11 (with an MPL limit of one) consumes 1.64 megabytes of memory and results in a 49% average system disk utilization, while the last solution (with an MPL limit of four) consumes only half as much memory but results in a significantly higher disk utilization of 75%.

MPL limit	**Memory per query (MB)**	**Query class memory (MB)**	**Disk utilization**
1	1.64	1.64	49%
2	0.82	1.64	63%
3	0.49	1.47	69%
4	0.20	0.80	75%

Table 11: 130 sec goal solution characteristics

How should a controller decide which of these solutions is the "best" one? This is an important decision, as the choice of a solution for any one class will determine the level of competition seen by other classes at shared resources; thus, it will also indirectly determine their set of feasible solutions. Among those listed in Table 11, the solutions with MPL limits of two and three are poor choices because the others consume either less memory or less disk. Which of the remaining solutions is "best" depends upon what resources are needed by the other classes in the system. Unfortunately, the other classes in the system will *also* have multiple solutions to choose from, and each of *their* solutions will represent a different trade-off between resources. Thus, it is extremely difficult to anticipate the *best* solution for a particular class without determining the solutions for every class simultaneously.

Even if only a single solution existed for any particular response time goal, the controller must still understand how the memory and MPL knobs effect response times in order to use these knobs to move a class toward its response time goal. Figures 15 and 16 illustrate this relationship between MPL, memory, and response times. Figure 15 shows that if memory per query is held constant (i.e. for any vertical line drawn through the graph), an MPL increase will result in a response time improvement for this workload. However, this improvement diminishes as the MPL increases because the benefits of increased concurrency eventually reach a point of diminishing returns. Increasing the MPL is not the only way to improve response time and throughput, however. Increasing memory per query will also allow queries to flow through the system faster. We can see this effect by looking at the right hand side of Figure 15 (larger memory allocations). There, MPL increases beyond two or three have less of an effect on response times than do MPL increases at smaller memory allocations. This is because execution times are improved enough by the greater memory allocation that higher degrees of concurrency are less effective in that region of operation.

The relationship between MPL and memory per class is more complex. Figure 16 shows that if the query class memory is held constant (i.e. for any vertical line), higher MPLs do not necessarily result in lower response times. For example, if we draw a vertical line at three megabytes of memory in Figure 16, we can see that running two queries concurrently provides the best performance (because those queries will be operating at close to their maximum memory requirement). Similarly, for 4.5 megabytes, an MPL of three produces the best response times because there is enough memory to run three queries at their maximum requirement. While it is not shown in Figure 16, this behavior repeats itself for higher MPLs: when there is enough memory to run N queries at their maximum requirement, then an MPL of N provides the best query performance. On the other hand, the best performance for only one megabyte of query class memory is obtained with an MPL of six. In this region of operation, the reduction in MPL queuing provided by a higher MPL outweighs the penalty of a reduced memory allocation per query. These observations support the conclusions of Cornell and Yu [Yu 93], who showed that the best query performance is obtained when queries are allocated either their minimum or maximum memory requirements. M&M exploits the Cornell and Yu results, as will be explained in Section 5.3.

In addition to the workload just described, hundreds of additional workloads were simulated prior to designing the M&M heuristics. The findings from these simulations can be summarized as follows. First, multiple solutions exist to a class's response time goal in nearly all cases. Second, the relationship between MPL, memory per class, memory per query, and CPU/disk utilizations is a complex function of the memory demand of the class, its arrival rate, its goal, and the degree of competition faced by the class from others classes in the system. As a result, a controller cannot easily predict the response time that will result from a particular <MPL, memory> knob setting. In fact, it may not even be able to predict with certainty whether response times will increase or decrease in response to a particular adjustment.

5.3 M&M: A Working Storage Class Controller

As the previous has shown, the huge number of possible <MPL, memory> combinations, the complex relationship of MPL and memory to response times, and the existence of multiple solutions to a single goal make the design of an effective working storage controller very challenging. However, Section 5.2 has also provided some insight for developing heuristics to efficiently prune the search space of possible <MPL, memory> combinations. This section will first explain these heuristics and then show how they are used by M&M to determine MPL and memory knob settings. It will then describe the concept of non-integral MPL limits, which allow response times to be "fine-tuned" using the MPL knob.

5.3.1 M&M Controller Heuristics

The most important heuristic for controlling the performance of working storage classes was suggested by the discussion of Figure 16: If there is enough memory available to run N queries at or near their maximum requirement, then the best response time is obtained with an MPL of N because those queries can execute with optimal performance. At the other end of the spectrum, for small available memory amounts, the best response time is obtained with high MPLs and a per-query memory allocation close to the minimum requirement. These results confirm the memory allocation heuristic derived by Cornell and Yu [Yu 93], which states that the best *return on consumption* is obtained by allocating only the minimum or the maximum memory requirement of any individual query. Return on consumption is a measure of response time improvement versus the space-time cost of memory. In addition, Cornell and Yu also showed that the return on consumption for a maximum allocation is much higher than for a minimum allocation.[6] These results form the basis for the first heuristic:

Heuristic 1 *Allocate the maximum memory required by each individual query if possible; otherwise allocate the minimum requirement. Allocate an amount in between min and max to only one query of a class at any moment, and only if there is no other alternative.*

The next heuristic sets an upper limit on the total MPL for a class, and is based on the behavior of the MPL knob that was observed in Figure 15: As queuing delays decrease, the potential response time benefits of increasing the MPL decrease as well. Clearly, the total MPL limit of the class should not be increased beyond the point at which nearly all MPL queuing delays are eliminated; memory would be underutilized as a result. M&M defines this threshold as being the point were the average MPL wait queue length is less than 0.5. In other words:

[6]Both the min/max heuristic and the conclusion that a join query's return on consumption is largest for a maximum allocation were shown to hold for hash-based, sort-merge, and nested loops join methods [Yu 93].

Heuristic 2 *Do not increase the MPL of a class if there are fewer than 0.5 waiters in its MPL queue, on average.*

M&M's next MPL-limiting heuristic recognizes that an MPL increase implies a cost for the other classes in the system, and that this cost comes in the form of increased competition at shared resources. MPLs should therefore not be allowed to rise so high that resource utilizations become "unreasonable." M&M translates the notion of "reasonable utilization" to "disk queue lengths that are less than or equal to one, on average." In some cases, however, the only possible way to achieve a set of goals will be to run the system with average queue lengths above one. Thus, the third heuristic is:

Heuristic 3 *Do not increase the MPL of a class if average disk queue lengths are greater than one, unless there is no other alternative.*

Given that only minimum or maximum memory requirements are allocated to individual working storage queries, M&M sets the MPL limit for a class by determining how many of its queries should execute at min (*minMPL*) and how many should execute at max (*maxMPL*). The final heuristic deals specifically with one effect of an increase in minMPL, namely, a corresponding increase in the probability that an arriving query will indeed be allocated its minimum memory requirement. Because a min query requires many more I/Os than one running at its maximum memory requirement (roughly three times as many in the case of a hash join [DeWitt 84]), any increase in the probability of a minimum allocation for the queries of a given class will necessarily increase the class's average *execution* time (i.e. response time minus waiting time). Unfortunately, predicting whether the admission of an additional min query will increase or decrease the class's average *response* time is extremely difficult. However, in all of our exploratory simulations, we have observed that any increase in the number of min queries always resulted in a response time increase when the number of max queries was two or greater. While this observation may not apply under all conditions, its value in pruning non-productive combinations of min and max allocations far outweighs the risk that it will dismiss a possible solution. Thus, the final M&M heuristic is:

Heuristic 4 *Never increase the number of queries allowed to run at min if two or more queries are allowed to run at max.*

5.3.2 Determining a New <MPL, Memory> Setting

Depending on the current state of the system, the M&M working storage class controller will take one of four actions in order to reduce the average response time of a class:

`max++` Increase the number of queries allowed to run at max.

The pool size for the class is increased enough to allow one more query to execute at max (based on the *average* maximum requirement of the class). If the number of queries allowed to execute at min (*minMPL*) is non-zero, then *minMPL* is reduced by one and the total MPL limit for the class remains unchanged. If one query of the class had been permitted to execute in between min and max, then this is no longer allowed.

`min++` Increase the number of queries allowed to run at min.

The pool size for the class is increased enough to allow one more query to execute at min (based on the *average* minimum requirement of the class). If one query of the class had been permitted to execute in between min and max, then this is no longer allowed.

`disk--` Request a reduction in disk utilizations from disk buffer classes.

This action is accomplished as described in Section 5.1.2.

`mem++` Increase the memory allocation for the class, allowing one query to execute between min and max.

This action is only taken as a last resort, and is only possible if at least one query is already allowed to execute at min. The pool size for the class is increased by a fixed *step size*, which is set at 5% of configuration memory.

Using the heuristics just derived, Figure 17 shows how M&M decides what action to take and highlights the rationale behind these decisions.

```
int GlobPool;   // current size of the global pool
int avgMax;     // avg maximum memory demand of the class
int maxMPL;     // # of max queries allowed for the class

bool disksFull = (avg disk queue lengths > 1.0);

if (there are no disk buffer classes OR
    a previous request failed to reduced disk queue lengths) then
    disksFull = FALSE;// to ignore heuristic # 3
endif

if (disksFull) then
    disk--;     // heuristic 3 prevents min++ or max++,
                //and heuristic 1 says mem++ is a last resort
elseif (there are fewer than 0.5 waiters in the MPL queue) then
    mem++;      // heuristic 2 prevents min++ or max++,
                //and no disk problem exists (so disk-- won't help)
elseif (avgMax < GlobPool) then
    max++;      // heuristic 1 says try to maximize return on memory consumption
elseif (maxMPL < 2) then
    min++;      // heuristic 1 says min++ if max won't fit,
                //and heuristic 4 says min++ may help
else
    mem++;      // max won't fit, and heuristic 4 says min++ may hurt
endif
```

Figure 17: Algorithm to determine a new <MPL, memory> setting.

5.3.3 Non-Integral MPL Limits and MPL Reductions

As we saw in Table 11, solutions to a particular response time goal normally exist at multiple MPL limits. The amount of memory required to achieve the goal will be different for each MPL, and the exact amount will be difficult to predict. It would therefore seem that unless a search strategy explores a large range of memory knob settings at each integer MPL limit, it will very likely miss these solutions. Unfortunately, M&M only allocates memory in very coarse discrete steps, i.e. based on the average minimum or average maximum memory requirements of a class. If we could somehow set the MPL knob at *non-integral* settings, however, then we could fine-tune response times and find solutions for a wide range of memory knob settings. Given that there can be only an integral number of queries present in the system at any moment, of course, such a non-integral MPL limit would have to apply to the average number of concurrent queries allowed in the system over time.

M&M produces non-integral MPL limits by first locating the lowest integer MPL limit at which the goal for a class is exceeded, and then delaying the admission of the next query by an amount of time that is equal to the amount of time by which the previous query exceeded its goal. No delay is introduced if the previous query violated its goal. By delaying the admission of a new query, the average actual MPL is forced to be some fraction lower than the integer MPL limit. In effect, the delay makes the system behave *as if* each query's response time exactly equals the goal for the class. This delay mechanism is used as follows: The search strategy of Figure 17 is invoked to find the first <MPL, memory> setting that exceeds the response time goal for a class; this setting is called the *home* setting for the class. During a home search, the delay mechanism is turned off. Once a home is found, the delay mechanism is then turned on in order to fractionally reduce the MPL to a point at which the goals are no longer exceeded. If the goals are violated again at any point (for example, due to a change in system load), then the delay mechanism is shut off and a new home search is initiated.

One problem with this delay technique is that the MPL limit for a class could be set too high. For example, if the MPL and memory of a class were set during a period of heavy system load, and then the load drops, the delay would simply be increased to make up for any improvement in response times. As a result, the MPL and pool size for the class might be set too high, and memory would be underutilized. We thus need a way to detect that it has become possible to reduce a class's MPL limit and still exceed its goals. M&M does this by continuously observing the average number of executing queries for each class (*execMPL*). If *execMPL* drops to more than one below the current integer MPL limit ($execMPL < MPLLimit - 1$), then the delay is greater than that which would be produced by a lower maximum MPL; the current MPL limit is therefore reduced by one.[7]

5.3.4 M&M Initialization and State Transitions

Because of the initial lack of statistical data, M&M cannot decide what the MPL limits for each class should be during the system's warm-up period. A system administrator must therefore supply initial MPL limits to M&M on a cold start. For the simulated workloads in this chapter, a cold-start MPL limit of two is used for working

[7] In actual practice, an MPL reduction trigger of $execMPL < MPLLimit - 1$ is a bit too sensitive, especially for those classes running at a point where *execMPL* is very close to $MPLLimit - 1$. After experimenting with different fractions, we settled on a somewhat more conservative MPL reduction trigger of $execMPL < MPLLimit - 1.5$.

storage classes, and an infinite MPL limit is used for disk buffer classes (i.e. no load control). Queries from working storage classes are allocated their minimum memory requirements during warmup, and transactions from disk buffer classes compete freely for any remaining physical memory. Note that the MPL limit for disk buffer queries always remains at infinity, because MPL limiting is a much less effective mechanism for controlling disk buffer allocation than it is for working storage (as was mentioned in Section 1.4).

In general, controlling no-goal class multiprogramming levels is required as well, as the additional competition for shared resources that they provide represents a possible threat to the goal classes. However, it is difficult to decide on a "proper" MPL for the no-goal class since there is no real basis for selecting appropriate resource allocations in the absence of a goal. The initial version of M&M described in this thesis simply limits the no-goal class working storage queries to an MPL of one at all times (no-goal disk buffer queries have no MPL limit).

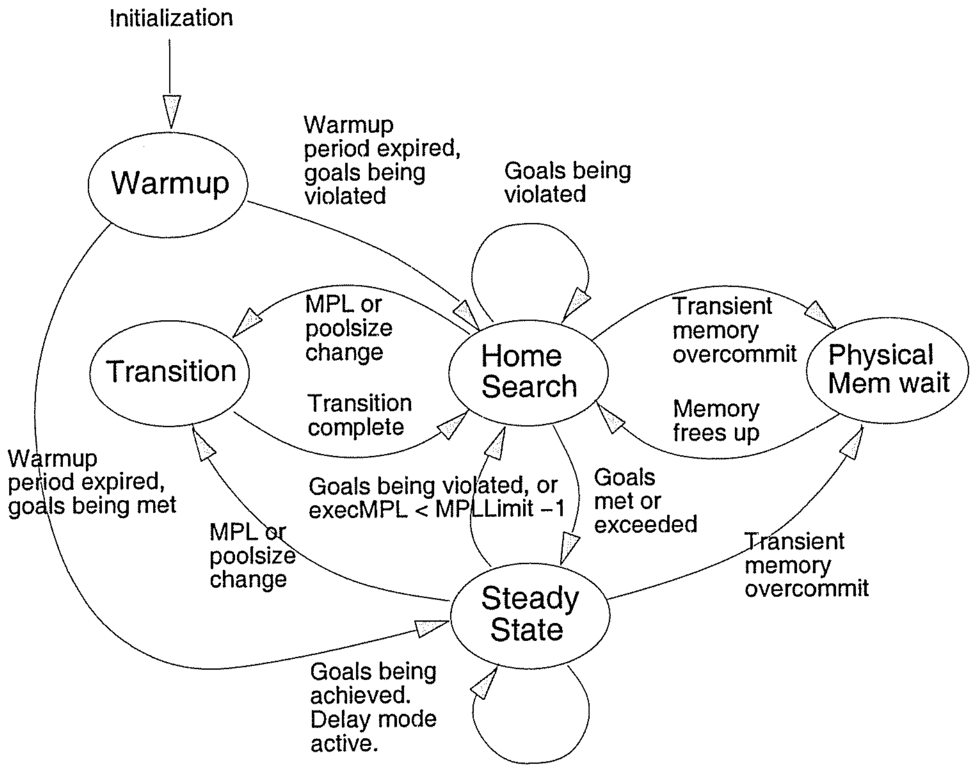
Initialization
Warmup
Warmup period expired, goals being violated
Goals being violated
MPL or poolsize change
Transition
Home Search
Transient memory overcommit
Physical Mem wait
Transition complete
Memory frees up
Warmup period expired, goals being met
Goals being violated, or execMPL < MPLLimit -1
Goals met or exceeded
MPL or poolsize change
Transient memory overcommit
Steady State
Goals being achieved. Delay mode active.

Figure 18: Working Storage class states

Figure 18 shows the states that a working storage class moves through under the control of M&M. There are a number of differences between these states and those of a disk buffer class (as described in Section 4.2.5).

First, note that the transition up and transition down states have been combined into a single *transition* state that occurs whenever there is any MPL or memory change. There is really no difference between an MPL increase or decrease in terms of its system effects, as both will cause a period of adjustment during which mean queue lengths settle to a new equilibrium (e.g. for queues at the DBMS entry point, i.e.MPL queues, or at the processors or disks). The key in both cases is to wait until the queue lengths stabilize. A sophisticated statistical method could be used to detect when a new stable point has been reached, but in practice, simply waiting for one observation interval to expire works reasonably well and is therefore what M&M does.

The other significant difference between the M&M and Class Fencing transitions is the replacement of the history build state by the *home search* state for working storage classes. Both serve the same purpose, which is to build up a statistically valid statistical sample of the class's performance after resource allocation changes. However, the home search state also implies that the delay mechanism described in the previous section has been turned off, and that, instead of trying to achieve an exact response time goal, M&M is looking for the new *home* setting (i.e. the lowest possible MPL at which the goal is being exceeded). Once a new home has been found, a transition to steady state is made, and the delay mechanism is turned on to fractionally reduce the MPL until the actual response time goal is being met.

The final difference is the new *physical memory wait* state that was was discussed in Section 5.1.1. A working storage class enters this state in the case of a pool increase that cannot immediately be supported by available memory frames. The point of this state is to hold off collecting statistics until the actual memory utilized by a class is in line with its newly increased pool size. In order to support a transition into this state, the underlying buffer manager must keep a count of the frames that are in either a fixed or reserved state. If there are too few unfixed, unreserved frames available when a working storage class's pool size is increased, it is placed in this state until enough fixed or reserved frames are freed up (most likely by a no-goal class query that was executing with an outstanding working storage reservation at the time of the pool size increase).

5.4 Experimental Multiclass Workloads

This section describes the simulated workloads that will be used to evaluate M&M in conjunction with Class Fencing. Because these two controllers together represent an algorithm capable of satisfying goals for either disk buffer or working storage classes, all of the workloads used here contain some combination of disk buffer and working storage goal classes; each also includes a no-goal class that consumes working storage. The response times for these classes differ by as much as three orders of magnitude, which is one of the challenging aspects of successfully handling multiclass workloads.

While different flavors of disk buffer classes were used in the evaluation of Class Fencing in Chapter 4, the performance evaluation here will use the same disk buffer class for all experiments and instead will explore different variants of the working storage class (called the "query" class). Three versions of the query class are used: one that can consume an average of 20% of the configuration memory at its maximum memory requirement, a second version whose average maximum requirement is 80% of the configuration memory, and a third whose average maximum is twice the size of the available memory. In all cases, the maximum memory requirements of individual queries will vary uniformly ±50% around the average for the class. These three versions of the query class present M&M with very different options for controlling performance. Whereas running multiple queries at their maximum allocation is an option for the 20% queries, only one 80% query can execute at a maximum allocation. The options for the 200% queries are even more limited since the maximum memory demand is greater than the available memory.

An eight MB, eight disk version of the simulated DBMS configuration described in Chapter 3 will be used to run the workloads. While this configuration is obviously underconfigured at eight megabytes of memory, a small amount of memory was necessary in order to keep simulation times tolerable. In addition, the key parameter of interest here is the size of the working storage memory demands in relation to the available memory, not the actual memory size. In fact, M&M's job becomes easier for larger memory sizes, as the option of running queries at their maximum memory requirement becomes more likely in this case. For smaller memories, M&M's options are more limited. A 64 MB scale-up experiment is included, however, to insure that M&M and Class Fencing can function properly with a larger configuration. The specific database and workload classes used in the subsequent performance analysis are described in the following two sections.

5.4.1 Database Model

As before, the database is modeled as a set of files, some of which have associated B+ tree indices. Index key sizes are 12 bytes, and key/pointer pairs are 16 bytes long. Table 12 lists the files and indices used for all of the experiments in this Chapter. The large, medium, small, and tiny files and indices are used by the "transaction" class (described in the next section). The various "query" files are actually sets of 50 identical files, two of which are randomly chosen for use as inputs for the execution of any particular "query" class transaction.

File name	# recs	rec size
big file	1,600,000	100
big index	1,600,000	16
medium file	640,000	100
medium index	640,000	16
small file	320,000	100
small index	320,000	16
tiny file	8,000	100
tiny index	8,000	16
40% query files	10,240	200
80% query files	40,960	200
200% query files	102,400	200

Table 12: Database characteristics

The sizes of the files and indices used by the transaction class were chosen to result in a wide variety of possible hit rates. The sizes of the query files were chosen primarily to determine the average memory demand of these classes; the 20%, 80%, and 200% files result in average per-query memory demands of 20%, 80%, and 200% of configuration memory, respectively. All files and indices are horizontally partitioned (declustered) across all five disks.

5.4.2 Workload Model

The simulated workload for this study consists of combinations of three classes: "transactions," "queries," and "no-goal queries." As in Chapter 4, the "transactions" represent a disk buffer class with short (sub-second) execution times, although here the transactions are modeled on the TPC-A benchmark [TPC 94] instead of TPC-C. The transaction class performs single-record index selects on 4 files: big, medium, small, and tiny (see Table 12). The file indices range from 1 to 3 levels deep, and accounting for some index nodes with less than full fanout, this implies between 12 and 16 random page references per transaction (with a mean of 13). The number of transaction terminals is fixed at 100, with exponentially distributed think times having a mean value of 10 seconds. Like the TPC-C class of the previous chapter, the transaction class used in this chapter is relatively insensitive to buffer hit rates. Except for the "tiny" file, which is small enough to fit in memory, all of its files are referenced uniformly and are much larger than memory. Thus, the transaction class's files are fairly cold (in terms of their average per-page reference frequencies), and a significant memory investment is required in order to improve the transaction class's response time.

The "query" class models a working storage class with longer execution times (tens of seconds or minutes). The individual queries consist of binary hybrid hash joins of two randomly chosen query files (see Table 12). The hybrid hash join algorithm [DeWitt 84] is used here because it is generally accepted as a good ad hoc join method. Allocating the maximum amount of memory to a join query will allow it to execute with the minimum number of I/Os, i.e. with a single scan of each relation. Allocating less memory (down to a minimum of approximately the square root of the number of file pages) increases the number of I/Os required in a linear fashion. The queries scan the query files with uniformly distributed random selectivities ranging from 33% to 100%, which (after accounting for a hash table overhead expansion factor of 1.2) results in uniformly distributed maximum memory demands ranging from 10-30%, 40-120%, or 100-300% of configuration memory, depending on the particular set of files chosen (see Table 12). The query class terminal population is set at 75, and each terminal has an exponentially distributed think time whose mean depends on the file size assigned to the class: 150 seconds for the 10-30% files, 1300 seconds for the 40-120% files, and 3200 seconds for the 100-300% files. Note that the randomness in both the arrival process and the memory demand for queries results in a high degree of variance in resource demands within the query class.

The last class, the "no-goal query" class, is identical to the query class that references the 40-120% query files. The no-goal query class is used to measure the "goodness" of solutions chosen for the transaction and query classes (which are both goal classes) – as explained in Chapter 4, the more resources that are left available for no-goal queries, the lower their response times and the better the solution. Because no-goal queries are used to evaluate the solutions chosen for the other two goal classes, we simply require that one no-goal query be present in the system at all times. Thus, the terminal population for no-goal queries is fixed at two[8], with no think time; recall that their MPL limit is fixed at one by M&M. Table 13 summarizes the number of terminals and the think times for each class.

5.5 Experiments and Results

In this section, the DBMS simulation model described in Chapter 3 is used to examine how well M&M and Class Fencing together can achieve a variety of goals for several variations of a simulated multiclass workload.

[8]Because startup, termination, and occasional memory queueing delay times can sometimes create a time lag between the completion of a query and the start of the next, even with a zero think time, a terminal population of two insures that there is always a no-goal query present in the system.

Parameter	Value
# Transaction terminals	100
Mean transaction think time	10 sec
# Query terminals	75
Mean query think time	150/1300/3200 sec
# No-Goal Query terminals	2
No-Goal Query think time	0 sec

Table 13: Workload parameter settings

Each version consists of transactions, queries, and no-goal query classes, as described in Section 5.4. The difference between each variation is in the average memory demanded by the query class (20%, 80%, and 200% of configuration memory). In all variants, the actual per-query memory demand varies uniformly ±50% about the mean.

In order to obtain statistically meaningful simulation results, the 20% and 80% memory demand versions of the workload are executed for eight simulated hours, and the 200% version is executed for sixteen hours. Response time statistics are collected and reported for only for the last half 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. A transient analysis of response times is also included to indicate how the algorithm operates over the entire range of simulated time.

The performance metrics used for judging the combined behavior of M&M and Class Fencing are the performance index of each class and the average response time of the no-goal query class. The no-goal class response times are used to roughly indicate the amount of "excess" resources left over after the goal classes have been allocated what they need to meet their goals; the larger the amount of left-over resources, the lower the no-goal response times, as before.

The goals used for the experiments in this section were chosen after exploring a wide range of possible response time combinations for the workloads. From this large set, many of the goal combinations were eliminated because they were either too tight or too loose for the configuration used here. Of the remaining simulations, preference was given to those that showed interesting phenomena. As a result, the reader should not expect to see any "rigorous" pattern in the set of goals for each experiment.

5.5.1 Three-Class Workloads

Table 14 shows the results from the base case experiment with one no-goal class, one transaction goal class, and one query goal class whose per-query memory demands range from 10-30% of the configuration memory. Each row represents a different combination of goals for the transaction and query classes.

Query goal (secs)	Tran goal (msecs)	Query perf index	Tran perf index	No-goal resp (secs)
100	350	0.96	0.80	49
100	250	0.96	1.01	63
50	300	0.96	0.99	77
50	250	0.95	0.99	82
15	300	0.98	0.99	96
15	250	0.98	1.01	117
8	350	0.99	1.02	108
8	250	1.03	1.01	117

Table 14: 10-30% query memory demand workload

The performance indices in Table 14 show that both the transaction and query classes are kept to within a few percent of their goals, and that the no-goal class's response times degrade progressively as the goals tighten, as would be expected. Thus, M&M and Class Fencing are working together successfully for this workload. In fact, the first row shows a case where the transaction goal is being *overachieved.* This is because the first row represents a very loose goal for the transactions (350 msecs). The disk buffer controller (Class Fencing) initially decided that no memory was required to achieve this goal, but the resulting low memory/high disk solution created disk response times that were too high to meet the goal for the query class. M&M then requested that the transaction class enter exceed mode in order to lower the average disk response time to a point that allowed the query class to meet its goal.

To examine M&M's transient behavior for this workload, the exponentially-weighted average observation interval response times from the (100 sec, 250 msec) goal pair experiment of Table 14 are graphed as a function of time; Figures 19 and 20 show these graphs for the query and transaction classes, respectively. The most obvious feature of the query response time graph in Figure 19 is the presence of multiple sharp downward spikes that appear immediately after shorter upward spikes. The upward spikes are transients in arrivals or memory demand that temporarily increased the system load enough to trigger an increase in the MPL for the query class. During this adjustment, M&M's delay mechanism was turned off while a new *home* setting was

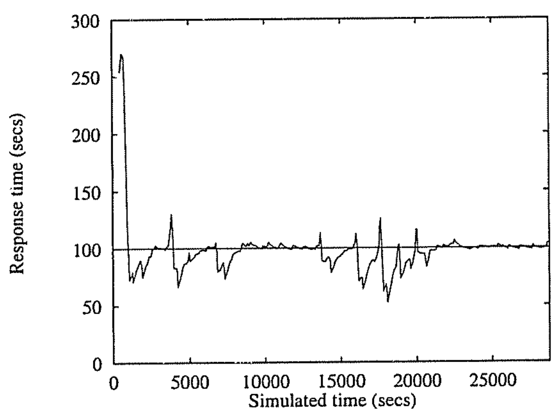

Figure 19: Query response times, 10-30% queries, (100 sec, 250 msec) goal

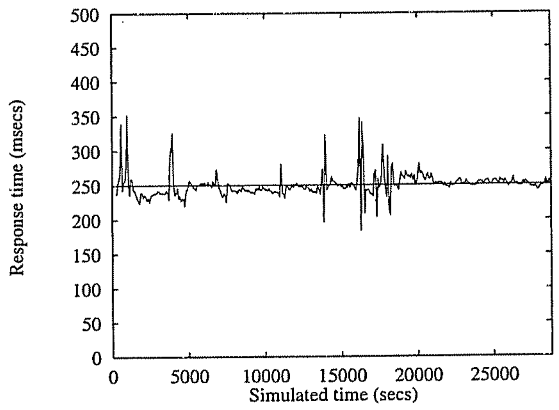

Figure 20: Transaction response times, 10-30% queries, (100 sec, 250 msec) goal

being searched for. The downward spikes reflect the fact that queries can temporarily exceed their goals any time the delay mechanism is turned off due to a knob adjustment.

An examination of the transaction class response times for this workload in Figure 20 shows upward spikes appearing at the same points where the query class temporarily increased its MPL. These MPL increases caused a temporary increase in disk response times for the transaction class. Although the Class Fencing controller compensated for these increases almost immediately, the transient upward spikes in the transaction class response times were unavoidable. On average, however, Figures 19 and 20 both indicate that M&M and Class Fencing operate in a very stable manner for this workload.

Query goal (secs)	**Tran goal (msecs)**	**Query perf index**	**Tran perf index**	**No-goal resp (secs)**
300	250	0.99	1.01	113
200	350	0.96	0.78	137
200	275	0.91	0.99	124
100	275	1.10	1.01	148
100	250	1.01	1.02	91
75	350	0.96	0.80	162
75	275	1.11	1.03	156

Table 15: 40-120% query memory demand workload

Table 15 shows the results of the base workload when the per-query memory demands are significantly larger, ranging from 40-120% of the configuration memory. The results are similar to the previous experiment, with most of the goals being held to within a few percent. The transaction class exceeds by approximately

20% for the (200 sec, 350 msec) goal pair because it was placed in exceed mode in order to allow the query class to meet its goal. The last goal pair of (75 sec, 275 msec) represents an unachievable goal; in this case, the transaction class's pool size is too large to allow the queries to run with an MPL of five ($maxMPL = 1, minMPL = 4$), which is what they need to achieve their goal. As before, the no-goal response class times degrade as the goals get tighter. Table 15 shows an unusual result, however; query goals are violated by 10% for the (100 sec, 275 msec) goal pair, while the tighter (100 sec, 250 msec) goal pair is achieved. To explain this violation, the transient query class response times for the (100 sec, 275 msec) goal case are graphed in Figure 21. The points in Figure 21 represent the actual interval average response times, and the wobbly line represents the selective exponentially-weighted average of those averages (as explained in Section 2.2.2).

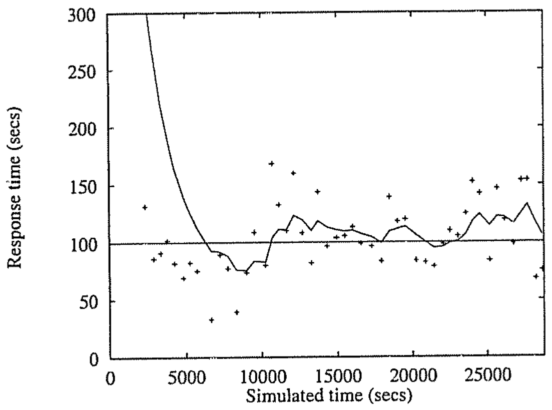

Figure 21: Query response times, 40-120% queries, (100 sec, 275 msec) goal

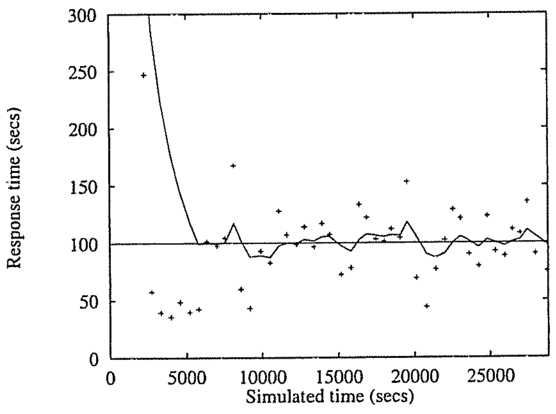

Figure 22: Query response times, 40-120% queries, (100 sec, 250 msec) goal

Since the average maximum memory demand for this query class is 80% of the configuration memory, the chances of an individual query (admitted at max) having to wait for such a large amount of memory to free up are significant. These frequent memory waits combine with an already large variation in the per-query memory demand to produce a large response time variance for this class, as is shown in Figure 21. This high variance in response times for the 40-120% queries has two effects: The first is that M&M computed a much larger tolerance band for these queries (up to ±30%) than it did for the 10-30% queries (±5% at most). Recall from Section 2.2.3 that the tolerance band is used to decide if a class is meeting its goal (i.e. goals are being met if the observed average response time falls within plus or minus some percentage of the goal), and its width is a function of the class's response time variance. The ±30% tolerance band computed for this workload means that even if the goals are being violated by 10%, M&M considers this "close enough" to avoid taking any action. The same ±30% tolerance band is in effect for the tighter (100 sec, 250 msec) goal pair, but the system state in this case just happened to place the query class's mean response time much closer to the center of its tolerance band, as Figure 22 shows.

While a ±30% tolerance band may seem excessive, it is the key reason that M&M is able to do *as well* as it does for this workload. Without this wide tolerance, M&M would be forced to act on transient increases and decreases in query response times by adjusting the MPL and/or memory knobs. These knob adjustments would increase the variance of the class even more, creating a system too unstable to control. Instead, M&M only occasionally asks for a reduction in disk response times; this action is sufficient to address the worst upward spikes in the average query response time, leaving the query class MPL untouched. In addition, it should be noted that over the long term, even the (100 sec, 275 msec) goal's average query response times will eventually much closer than 10% from the goal. This is because any long term goal violation will increase the sensitivity of the controller to short term goal violations (because exponential weighting considers the past value of a statistic as well as its current value); as soon as one of these short term violations occurs, an MPL and/or memory adjustment will be made and the class will enter a new region of operation much closer to its actual goal.

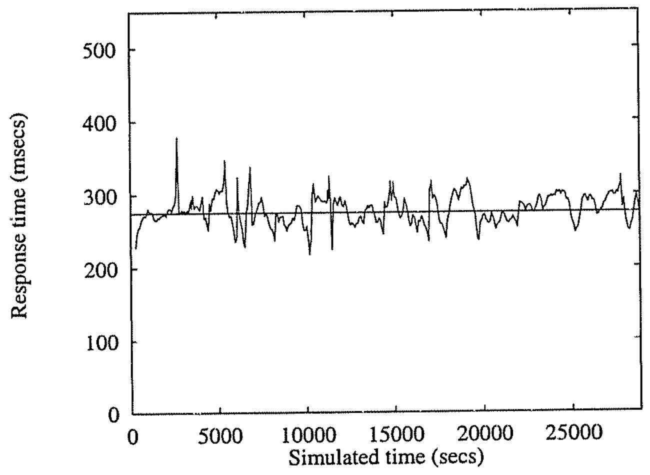

Figure 23: Transaction response times, 40-120% queries, (100 sec, 275 msec) goal

Another effect of the increased variance in the query class memory demand can be seen by examining the transaction class response times for the (100 sec, 275 msec) goal pair in Figure 23. Comparing this graph to that of the previous (20% query memory demand) workload (Figure 20), we can see that the transaction response times have a much higher variance for this workload as well. This is because the fluctuations in the physical memory demand of the query class also cause fluctuations in the amount of buffer pool memory available to the transaction class. Thus, an increase in the variance in one class can be "transmitted" to another class via interactions at shared resources.

Query goal (secs)	Tran goal (msecs)	Query perf index	Tran perf index	No-goal resp (secs)
700	250	0.90	0.98	135
600	350	0.96	0.69	151
600	250	1.02	0.93	160
500	300	1.00	0.92	130
500	250	0.95	1.04	132
400	350	0.90	0.73	154
400	250	0.95	1.05	163

Table 16: 100-300% memory demand workload

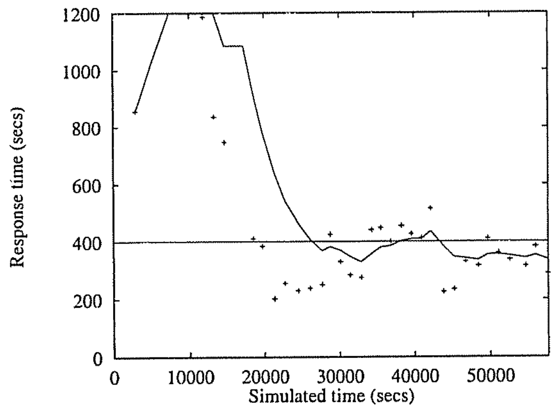

Figure 24: Query response times, 100-300% queries, (400 sec, 350 msec) goal

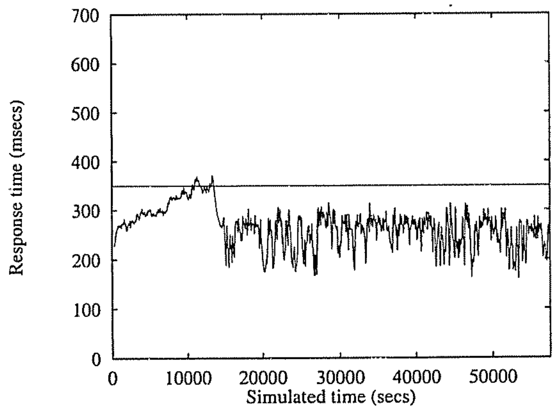

Figure 25: Transaction response times, 100-300% queries, (400 sec, 350 msec) goal

Table 16 shows that similar results are obtained for the 100-300% memory demand queries. All goals are achieved to within a few percent except for the 350 msec transaction class goals, which are exceeded by up to 30%. In the 350 msec transaction goal cases, the transactions are running in exceed mode in order to allow the query class to increase its multiprogramming level high enough to achieve its goal. For this workload, the (400 sec, 350 msec) goal pair is selected for an examination of M&M's transient behavior, which is shown in Figures 24 and 25. Looking at the transaction response times in Figure 25, one can clearly see the point (about 15,000 seconds into the simulation) at which M&M requested that the transactions enter exceed mode.

Once the transactions were able to decrease their disk utilizations, the query class was able to raise its MPL from five to seven, which then allowed it to achieve its 400 second goal. Here again, we see the transaction class response times exhibiting an even higher variance than was the case for the 80% query memory demand workload. Here, however, the reason for the variance is not the random fluctuations in the memory demands of the query class (as all of the 200% queries run at a minimum allocation here, and the query class only consumes about 20% of the available memory). Instead, the transaction class's response time variance is the result of a high variance in their disk response times. With a multiprogramming level of seven, the query class's disk utilization is fairly bursty, and this effect is transmitted to the transaction class via the shared disks.

5.5.2 A More Complex Workload

The next experiment tests how well M&M can satisfy goals for a more complex mixed workload consisting of two working storage classes, two disk buffer classes, and a no-goal class. The two (essentially identical) query classes are replicas of the query class used in the previous experiments, and consist of 30 terminals each. Similarly, the two transaction classes are replicas of the transaction class used previously, and consist of 50 terminals each. The think times used in the three-class workloads are retained here. The transaction class files are replicated so that each of the two transaction classes accesses its own files. The set of query files is shared by the two query classes, and they result in an average memory demand of 20% of the configuration memory (with individual query demands ranging from 10-30%).

Goal set #	Qry 1 goal (secs)	Qry 2 goal (secs)	Trn 1 goal (msecs)	Trn 2 goal (msecs)	Qry 1 perf index	Qry 2 perf index	Trn 1 perf index	Trn 2 perf index	No-goal resp (secs)
1	150	175	300	400	0.92	0.94	0.93	1.00	164
2	150	175	250	400	0.92	0.94	1.11	1.00	181
3	50	175	300	400	0.88	0.93	0.99	0.99	95
4	50	100	275	350	0.98	0.99	1.13	1.08	96
5	10	200	350	400	0.88	0.90	1.01	0.99	130
6	10	100	400	400	0.99	0.91	1.01	0.99	145

Table 17: More complex workload (10-30% memory demand)

Table 17 shows various combinations of goals for this more complex workload. It shows one case where goals are violated by more than 10%: in goal set #4, transaction class #1's 275 msec goal is violated by 13%. This goal is violated because disk response times are too high there to achieve such a tight goal with five classes executing concurrently. However, M&M does achieve the goals for the other three classes fairly well in this case.

There are two cases in Table 17 where query goals are exceeded by more than 10%. This occurs in goal sets #3 and #5, where the performance indices are 0.88. The reason that these goals were exceeded is due to the same phenomenon that was shown in Figure 19. Both of the exceeding query classes are operating very close to their MPL reduction trigger ($execMPL < MPLLimit - 1.5$); this operating region is unstable because their MPLs tend to "wobble" up and down. Every time a new MPL is chosen, the delay mechanism is shut off and the same sharp drop in response times that was displayed in Figure 19 occurs here. (Clearly, the MPL reduction mechanism should be tuned further to reduce the probability of such "MPL wobbling.")

5.5.3 Scale-up Experiment

The final experiment in this chapter verifies that the favorable results for the combination of M&M and Class Fencing can scale up to larger memory and query file sizes. For this experiment, the configuration memory and query file sizes are both increased by a factor of eight (increasing memory to 64 MB). The base case workload is then rerun with queries that demand an average of 20% of the configuration memory. Table 18 shows that M&M and Class Fencing achieved the goals for this workload and configuration as well as they did for the smaller configuration.

Query goal (secs)	Tran goal (msecs)	Query perf index	Tran perf index	No-goal resp (secs)
500	300	0.96	0.87	299
300	350	0.97	0.75	334
200	350	0.99	0.79	105
200	250	0.97	0.98	251
100	350	0.93	0.92	363
100	300	1.03	0.97	312
100	250	0.91	0.99	430

Table 18: Scaled-up workload (10-30% memory demand)

5.6 Summary

This Chapter has presented M&M, an algorithm for goal-oriented working storage allocation and MPL management. Section 5.1 first explained how working storage and disk buffer memory management can coexist with each other, and introduced the notion of an *exceed mode* to resolve the interclass dependencies that arise because of competition between classes at shared disks. Section 5.2 then explored the effect of the working storage allocation and MPL knobs on the response times of working storage classes, showing how multiple solutions to a single response time goal can exist when more than one knob is used to control performance. This background information was then used to derive the heuristics upon which M&M is based. Section 5.3 then described the M&M algorithm, showing how it uses its heuristics to help prune the large search space of possible MPL and memory allocation combinations. A novel technique for fine tuning response times using non-integral MPL settings was also described. Finally, Section 5.5 explored the steady state and transient performance of the M&M and Class Fencing algorithms when combined to handle multi-class workloads. The results of these initial studies showed that M&M appears to be quite accurate, stable, and robust in the presence of differing workloads, configurations, and query memory demands.

Chapter 6

Conclusions

I have made good judgements in the Past.

I have made good judgements in the Future."

– Vice President Dan Quayle

This chapter first reviews the material presented in this thesis, summarizing the general architecture presented in Chapter 2, the Class Fencing disk buffer controller presented in Chapter 4, and the M&M working storage controller presented in Chapter 5. It then closes with a discussion of a number of areas for future work.

6.1 Thesis Summary

Motivated by the challenge presented by multiclass database workloads, this thesis has advocated the adoption of *goal-oriented resource allocation* mechanisms within a DBMS in order to satisfy user-specified per-class performance goals for these workloads. Chapter 1 defined a set of criteria with which to judge such mechanisms – namely: accuracy, responsiveness, stability, overhead, robustness, and practicality. Chapter 2 then presented an architectural foundation that can be used to build individual *controllers* for specific DBMS resources. This architecture features a *class-based, feedback-oriented* approach that greatly simplifies the problem of multiclass resource allocation while providing the appropriate responsiveness for high-throughput classes. It exploits well-defined *states and transitions* for each class to prevent statistics collection from being biased by transitions in a class's resource allocation, and employs *selective exponential weighting* of statistics to prevent a controller from managing the natural statistical fluctuations of a class. Also, it includes a method for dynamically computing a *tolerance band* around a class's goal to allow a controller to tightly manage classes with low response time variance while exhibiting less sensitivity for classes with high response time variance. While individual resource controllers are responsible for correctly setting the allocation knobs for a class at *particular* points in time, the general mechanisms just described are responsible for insuring that a class's resource allocation knobs are adjusted in a stable and responsive fashion *continuously* over time, regardless of what resource is being controlled.

Chapter 3 explained the behavior of the detailed DBMS simulation model used to evaluate the performance of the two algorithms presented in subsequent chapters. The simulator includes realistic models of page reference patterns, buffer management mechanisms (including prefetch), disk behavior, and query operator implementations.

Chapter 4 presented a goal-oriented controller for the disk buffer memory knob called *Class Fencing*. Class Fencing is based on the notion of *hit rate concavity*, which uses a simple straight line approximation to predict a class's buffer hit rate as a function of memory while at the same time providing a conservative memory estimate. Using Chapter 3's detailed simulation model, the steady-state and transient performance of Class Fencing was investigated for various multiclass workloads and goal combinations. These experiments have shown that Class Fencing is able to hold most classes to within a few percent of their goals, and to do so in a stable manner. Class Fencing was also shown to be very responsive 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. Class Fencing is also fairly robust because its primary assumption, hit rate concavity, applies to a wide range of workloads. Finally, Class Fencing is able to handle arbitrary page reference skew and can detect unachievable hit rates, both of which were stumbling blocks for the earlier Fragment Fencing algorithm.

Finally, Chapter 5 presented a second goal-oriented controller, called M&M, that manages allocations and multiprogramming levels for working storage memory. M&M works in conjunction with the Class Fencing controller for managing disk buffer classes in order to provide a comprehensive goal-oriented memory allocation solution for multiclass database workloads. Section 5.3.1 developed a set of heuristics that allow M&M to prune unproductive MPL/memory combinations in its search for a *home* setting that comes closest to meeting or exceeding the goal for a working storage class. M&M uses a novel admission delay mechanism that allows it to lower the home MPL to a *non-integral limit* in order to fine tune the class's response time at its home setting, thereby bringing it closer to its goal. In addition, M&M's *exceed mode mechanism* for disk buffer classes allows it to deal with the interdependence between classes that results from their interactions at shared disks. Using Chapter 3's detailed simulation model, the steady state and transient performance of M&M was explored in Section 5.5. These experiments showed that the combination of M&M and Class Fencing can achieve goals for a variety of different workloads, configurations, memory demands, and degrees of variance within each goal class.

6.2 Future Work

The general architecture and the specific controllers for disk buffer and working storage memory described in this thesis provide a firm foundation for a general solution to the goal-oriented DBMS resource allocation problem. However, plenty of work still remains. This section will discuss some of this work, including improvements to M&M's admission delay mechanism, more advanced management of no-goal and disk buffer classes, controllers for other resource allocation knobs, bottleneck analysis, and user interface issues.

6.2.1 M&M Delay Mechanism Enhancements

As was shown in the performance analysis section of Chapter 5 (Section 5.5), M&M's admission delay mechanism provides an effective way to exploit the normally coarse-grained MPL knob in order to fine tune working storage class response times. However, Section 5.5 also showed a problem with the delay mechanism: When the delay is shut off at the beginning of a search for a new home MPL for a given class, the class exceeds its response time goal. In fact, if a class is running with any non-zero delay, then it seems premature to increase memory and/or MPL to compensate for a transient response time spike before decreasing (or eliminating) the delay time; decreasing the delay could be just as effective (and much less disruptive) a mechanism for responding to transients. Currently, M&M does not treat admission delay as a "first class" knob; it is either on or off, and its length is pre-determined on a query-by-query basis. Instead, the delay could be set based on a *running deficit* for the class that accumulates over the entire observation interval.

6.2.2 Disk Buffer and No-Goal Class Improvements

In this thesis, disk buffer classes (both goal and no-goal varieties) are assigned a static MPL limit of infinity, while no-goal classes that consume working storage are assigned a static MPL limit of one. Clearly, these static settings are inadequate. While the MPL limit for disk buffer classes is not likely to be an effective knob for controlling *their* performance, it may very well be an effective knob for controlling the performance of *other* classes because it can act to limit competition at the processors and disks. For example, instead of allowing disk buffer classes to exceed their goals with an increased memory allocation in order to reduce disk utilizations, another option that might make sense is to limit their MPL, especially if they are already exceeding their goals. Controlling disk buffer class MPL limits is a challenging problem, however, since it turns the existing one-dimensional disk buffer controller into a two-dimensional controller with a much larger solution space (and more than one possible solution).

Limiting no-goal transactions that consume working storage memory to an MPL of one is another M&M restriction that needs to be removed. No-goal class working storage MPLs should be allowed to increase as long as they do not represent a threat to the goal classes. Finding the point at which they become a threat is the challenge here. Similarly, there is no reason to assume that the MPL of the no-goal class always should always be non-zero. As was pointed out in Section 4.4, one option that would make sense when the system enters a degraded mode of operation is to prevent the execution of *any* no-goal transactions. Thus, the MPL limit for no-goal classes should really be allowed to vary from zero to N.

A final area for no-goal class improvement stems from the "additive" approach to building a goal-oriented DBMS that was defined in Chapter 2 and adopted throughout the thesis. While the additive approach simplifies the problem of designing a goal-oriented DBMS, it does not necessarily result in the best response times for the no-goal class. If the existing DBMS resource allocation mechanisms allow a goal class to "naturally" exceed its goal, then the additive approach dictates that no action should be taken. However, it may be possible to reduce the no-goal class response times if excess resources are taken from an exceeding goal class and reassigned to the no-goal class. Of course, because of the interdependence of classes that share resources, it is difficult to determine if the no-goal class performance would actually be improved by such a reassignment. More sophisticated mechanisms are needed to determine what, if anything, can be done to insure the miminim no-goal response time (subject to meeting the goals for all goal classes).

6.2.3 Other Resources Besides Memory

While effective MPL and memory management will likely be the critical point of control for most workload classes, processor and disk management must be addressed as well. Two key questions for processor and disk management are how performance goals should be translated into scheduling parameters (for example, dynamic priorities or percentages of resource utilizations) and how much control should be exerted at the load controller versus the scheduling of the resource itself. Load control is probably more critical, but goal-oriented scheduling will likely be required to "mop up" after any poor load control decisions, or to deal with short-term load transients. Another challenge in the area of goal-oriented processor and disk scheduling algorithms is insuring that efficient resource utilization is not adversely affected by the goal-oriented scheduling policies. For example, in disk-bound systems, minimizing disk head motion is critical to maximizing throughput at the disk. A goal-oriented disk scheduling algorithm must therefore take care to avoid unnecessarily increasing head motion; if the allocation process itself consumes too much bandwidth, then deciding how much disk bandwidth to allocate to each class will become a non-issue. While there are other resources that could also be used to control DBMS performance, the memory, MPL, processor, and disk knobs together will likely represent an adequate solution for most workloads.

In addition to managing the DBMS resources, the resource *demands* of individual transactions could be manipulated by a *goal-oriented query optimizer.* While there has been no published work to date on query optimization techniques for satisfying multiclass performance goals, some early work has been done on run-time selection of query plans based on resource availability [Hong 91, Ioannidis 92]. Given an optimizer that can generate more than one plan to execute a particular query, how should these plans be chosen at run time in order to satisfy class-based response time goals? What kind of plans should be generated? Could an optimizer instead generate a single plan that attempts to achieve a particular *target* response time (as opposed to one that *minimizes* response time)? While optimizers have sophisticated cost models to predict query execution times, they only need to be correct when comparing the *relative* performance of two plans. It is not clear if these cost models can be used for the much more difficult problem of accurately predicting the actual response time of a particular plan. While the issues involved in goal-oriented query optimization seem daunting, the fact that the optimizer has so much control over the resource demands of individual queries is enough to justify their exploration.

6.2.4 Bottleneck Analysis

Given a choice of knobs to turn for each class, the knob that controls the bottleneck resource of the class should be used first. Determining which resource is the bottleneck for a class is called *bottleneck analysis.* One approach to this problem is simply to sum up the times spent at various resources and pick the resource with the largest sum as the bottleneck. This solution is unattractive for two reasons: first, the overhead involved in collecting resource utilization statistics at such a detailed level may be prohibitive, and second, it ignores the fact that resources are likely to be utilized in parallel. A sampling approach is probably a better alternative, although detecting the state of a class during a particular sampling probe would require a non-trivial amount of engineering. For example, individual processes/threads have to be mapped to the classes they are working on behalf of, and some easy way to identify the state of each process/thread must exist (e.g. waiting for locks, processors, disks, network connections, executing on a processor, etc.).

Assuming that the engineering issues associated with sampling can be solved, memory usage still presents a knotty problem. While memory waits and memory allocations can be sampled, the relative *value* of the two different memory types to a class (disk buffers or working storage) is more difficult to evaluate. Here, the value of an additional memory page is measured by its ability to decrease the number of disk I/Os for a class. One possible way to detect the relative values of disk buffer versus working storage might be to compare the average number of buffer faults to the average number of disk I/Os for each class; whichever is greater could indicate the more important memory type. Unfortunately, the fact that a class experiences more buffer misses than other types of I/Os (or vice-versa) says nothing about how the class's I/O rate would be affected by an increased disk buffer (or working storage) memory allocation. For example, if a class references very cold files, then an increase in disk buffer allocation will not be a very effective control, regardless of the number of buffer faults experienced by the class. More research is needed to develop a better predictor that can judge the relative value of the working storage and buffer memory knobs.

6.2.5 User Interface Issues

In some sense, the whole point of this thesis is to raise the level of abstraction of a DBMS's performance tuning interface. The idea is not to eliminate all knobs in order to develop a self-tuning DBMS that always "knows what's best" in any situation (an elusive goal, to be sure), but rather to hide the low-level knobs behind an *autopilot* [Brown 93b] that can translate high-level performance requirements into low-level knob settings. Because this thesis concentrates on the mechanisms needed to achieve this objective for three specific knobs, the user-interface it presents leaves something to be desired. For example, requiring an administrator to specify an observation interval for each class is clearly inadequate, as it is difficult to determine a proper setting for this parameter by hand. As mentioned in Chapter 2, the observation interval should really be determined automatically from a class's response time variance, perhaps combined with a higher-level *sensitivity* knob (although even the sensitivity knob should provide a reasonable default behavior).

Given the objective of a high-level performance tuning interface, one can justifiably ask whether per-class response time goals themselves represent a high-level or a low-level knob. While response time goals are certainly higher level than the memory or MPL knobs, less sophisticated users may not even be able to *define* their workload classes, let alone specify goals for them. Unsophisticated users do not necessarily imply that response time goals and goal-oriented resource allocation methods are worthless at the low end of the DBMS market, however. Default class definitions based on observed response times or resource consumption, the ability to specify *relative* response times (i.e. one class should be twice as fast as another), and innovative graphical interfaces all represent possible ways to increase the ease-of-use of a performance tuning "auto-pilot." While more work certainly remains on additional resource allocation mechanisms, perhaps the "real money" to be made in goal-oriented DBMS research lies in innovative approaches to the performance tuning user-interface.

Bibliography

[Abbott 91] R. K. Abbott, *Scheduling Real-Time Transactions: A Performance Evaluation*, PhD Thesis, Princeton University, 1991 (Princeton CS TR-331-91).

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

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

[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 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.

[Brown 93a] 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 93b] K. Brown, M. Carey, M. Livny, "Towards an Autopilot in the DBMS Performance Cockpit," *Proc. 5th Int'l High Performance Transaction Processing Workshop,* Asilomar, CA, September 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.

[Brown 95] K. Brown, M. Carey, M. Livny, "Goal-Oriented Buffer Management Revisited," submitted for publication, July, 1994.

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

[Carey 85] M. Carey, M. Livny, and H. Lu, "Dynamic Task Allocation in a Distributed Database System," *Proc. of the Fifth Int'l. Conf. on Distributed Computing Systems,* Denver, CO, May 1985.

[Carey 89] M. Carey, R. Jauhari, M. Livny, "Priority in DBMS Resource Scheduling," *Proc. 15th Int'l. VLDB Conf.*, Amsterdam, The Netherlands, August 1989.

[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.

[Chou 85b] H. Chou, *Buffer Management of Database Systems*, PhD Thesis, University of Wisconsin, Madison, 1985.

[Chu 72] W. Chu and H. Opderbeck, "The Page Fault Frequency Replacement Algorithm," *Proc. 1972 AFIPS Fall Joint Computer Conf.*, Vol 41, AFIPS Press, Montvale, NJ, Dec. 1972.

[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.

[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, August 1989.

[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.

[Davison 94] D. Davison, G. Graefe, "Memory-Contention Responsive Hash Joins," *Proc. 20th Int'l VLDB Conf,* Santiago, Chile, September 1994.

[Davison 95] D. Davison, G. Graefe, "Dynamic Resource Brokering for Multi-User Query Execution," *Proc. ACM SIGMOD Conf,* San Jose, CA, May 1995.

[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.

[Denning 80] P. Denning, "Working Sets Past and Present," *IEEE Transactions on Software Engineering,* SE-6(1), January 1980.

[Easton 75] W. Easton, "Model for Interactive Data Base Reference String," *IBM Journal of Research and Development*, 19(6), November 1975.

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

[Effel 84] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management," *ACM TODS,* 9(4), Dec. 1984.

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

[Ferg 93] D. Ferguson, C. Nikolaou, L. Geargiadis, "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.

[Fujitsu 90] Fujistsu America, Inc. M2265 Technical Manual, part number 41FH5048E-01, Fujistsu America Technical Assistance Center, San Jose, CA, 1-800-826-6112.

[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.

[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.

[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.

[IBM 93c] IBM Corporation, *MVS/ESA Version 4.3 Initialization and Tuning Guide GC28-1643*, IBM Corporation, Poughkeepsie NY, March 1993.

[IBM 95] IBM Corporation, *MVS/ESA Version 5 Planning: Workload Management GC28-1493*, IBM Corporation, Poughkeepsie NY, March 1995.

[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.

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

[Jauhari 90a] R. Jauhari, M. Carey, M. Livny, "Priority-Hints: An Algorithm for Priority-Based Buffer Management," *Proc. 16th Int'l VLDB Conf.*, Brisbane, Austrailia, August 1990.

[Jauhari 90b] R. Jauhari, *Priority Scheduling in Database Management Systems,* PhD Thesis, University of Wisconsin, Madison, 1990 (available as UW Madison CS Technical Report CS-TR-90-959).

[Kaplan 80] J. Kaplan, "Buffer Management Policies in a Database Environment,", Masters Thesis, UC Berkeley, 1980.

[Lazowska 84] E. Lazowska, J. Zahoran, G.S. Graham, K. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models,* Prentice Hall, Englewood Cliffs, NJ, 1984.

[Livny 87] M. Livny, S. Koshafian, H. Boral, "Multi-Disk Management Algorithms," *Proc. ACM SIGMETRICS Conf.*, Alberta, Canada, May 1987.

[Lorin 81] H. Lorin and H. Deitel, *Operating Systems* (chapter 9: Resource Management), Addison Wesley, Reading MA, 1981.

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

[Mehta 94] M. Mehta, *Resource Allocation in Parallel Shared-Nothing Database Systems,* PhD Thesis, University of Wisconsin, Madison, 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.

[Pang 93a] H. Pang, M. Carey, M. Livny, "Partially Preemptible Hash Joins," *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," *Proc. 19 Int'l VLDB Conf.*, Dublin, Ireland, August 1993.

[Pang 94a] H. Pang, M. Carey, M. Livny, "Managing Memory for Real-Time Queries," *Proc. ACM SIGMOD '94 Conf.*, Minneapolis MN, May 1994.

[Pang 94b] H. Pang, *Query Processing in Firm Real-Time Database Systems,* PhD Thesis, University of Wisconsin, Madison, 1994.

[Pang 95] H. Pang, M. Carey, M. Livny, "Query Scheduling in Real-Time Database Systems", to appear in *Trans. of Knowledge and Data Engineering*, August 1995.

[Patel 93] J. Patel, M. Carey, M. Vernon, "Accurate Modeling of the Hybrid Hash Join Algorithm," *Proc. ACM SIGMETRICS '94*, Nashville, TN, May 1994.

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

[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 82] G. Sacco and M. Schkolnick, "A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model" *Proc. 8th Int'l VLDB Conf.*, Mexico City, September 1982.

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

[Selinger 93] P. Selinger, "Predictions and Challenges for Database Systems in the Year 2000," *Proc. 19th Int'l VLDB Conf,* Dublin, Ireland, August 1993.

[Stone 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.

[Weikum 93] G. Weikum et al, "The COMFORT Project – Project Synopsis," *Proc. 2nd Int'l Conf. on Parallel and Distributed Information Systems*, San Diego CA, January 1993.

[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, August 1990.