Towards Automated Performance Tuning For Complex Workloads*

Kurt P. Brown† Manish Mehta‡ Michael J. Carey† Miron Livny‡
{brown, mmehta, carey, miron}@cs.wisc.edu

†IBM Almaden Research Center ‡University of Wisconsin, Madison

Abstract

In this paper we explore the problem of automatically adjusting DBMS multiprogramming levels and memory allocations in order to achieve a set of per-class response time goals for a complex multiclass workload. We start by describing the phenomena that make this a very challenging problem, the foremost of which is the interdependence between classes that results from their competition for shared resources. We then describe M&M, a feedback-based algorithm for determining the MPL and memory settings for each class independently, and we evaluate the algorithm's effectiveness using a detailed simulation model. We show that our algorithm can successfully achieve response times that are within a few percent of the goals for mixed workloads consisting of short transactions and longer-running ad hoc join queries.

1 Introduction

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, 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 (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. Controlling the performance for such a workload by manually adjusting low-level DBMS performance "knobs" will become increasingly impractical, as has been argued previously [Nikolaou 92, Brown 93b, Weikum 93]. Ideally, a DBMS should be able to accept performance goals for each class as inputs, and should dynamically adjust its own low-level performance knobs using the goals as a guide.

Given performance objectives for each class, there are a number of mechanisms that a 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. Recently, techniques have been proposed for goal-oriented transaction routing [Ferg 93] and goal-oriented buffer management [Brown 93a]. However, 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.

With a choice of knobs to turn, a goal-oriented DBMS is faced with several non-trivial decisions. Perhaps the most obvious one is *which* knob should be turned. For example, if a response time reduction can be accomplished by either a multiprogramming level (MPL) increase or a memory allocation increase, which knob should be used? Should both be used simultaneously? Even if the DBMS can determine which knobs to adjust, it must still decide in which direction and how far each one should be turned. In other words, the DBMS must translate a performance goal specification into a particular resource allocation that will achieve that goal. How should it perform this translation? The problem is further complicated by the fact that classes can interfere with each other's performance through competition for shared resources. Because of this interference, the DBMS may find a knob setting that achieves the goal for one class but at the same time makes it impossible to achieve the goal for some other class.

This paper proposes an algorithm that dynamically adjusts both multiprogramming levels and memory allocations in order to achieve a set of per-class response time goals for complex multiclass workloads. Although the requirement here is for a simultaneous solution[1] across all classes, our algorithm attempts to find a solution for every class independently; at the same time, it tries to avoid solutions for one class that prohibit solutions for other classes. Such an approach significantly simplifies the problem, finds solutions relatively quickly, and eventually discovers a reasonable simultaneous solution in a large number of cases. Similar to other work in this area, the algorithm uses a set of heuristics and estimation techniques to control a feedback mechanism. In addition, it exploits a novel scheme that provides for non-integral MPL limits in order to provide much finer grained control over response times.

The remainder of the paper is organized as follows: We begin by reviewing existing techniques for automatically achieving multiclass performance goals in Section 2. Section 3 gives background information on the problem of goal-oriented MPL and memory management and introduces some of the difficulties that arise in attempting to solve this problem. Our algorithm for adjusting multiprogramming levels and memory

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

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

[1] A *solution* for a class is defined as a set of resource allocations that achieves the response time goal for the class.

allocations is then presented in Section 4. We describe our detailed simulation model in Section 5, and we use this model to analyze the behavior of our algorithm on multiclass workloads involving short transactions and longer-running queries in Section 6. Finally, we discuss future work and summarize our conclusions in Section 7.

2 Related Work

The case for goal-oriented resource management has been argued for distributed computing systems in general [Nikolaou 92] and for database management systems in particular [Brown 93b]. The COMFORT project at ETH Zurich is also directed toward the automation of DBMS performance tuning, and is described in [Weikum 93]. While these position papers describe the motivation for goal-driven systems, very little has been published to date on actual mechanisms for achieving per-class performance goals in a DBMS. A large amount of research exists on the allocation of memory and processors in a DBMS, but the vast majority of this work is directed toward optimizing some system-wide objective (e.g. maximizing buffer hit rates or minimizing individual query execution times). A system-wide performance objective will not, in general, satisfy a set of class-specific goals. The remainder of this section reviews the few published techniques for automatically achieving per-class performance goals in a database or operating system environment.

Two predictive algorithms for goal-oriented transaction routing in distributed transaction processing systems are described in [Ferg 93]. These two algorithms attempt to predict the effect of a routing decision on the response times of each class. Their inputs 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* for any class, where the performance index is defined as the observed response time divided by the response time goal. Because it is normalized relative to the goal, the performance index is a useful indicator of performance that allows comparisons across classes. An objective of minimizing the maximum performance index means that the algorithms do not have to maintain specific response times very accurately. Rather, they need only determine the correct *relative* response times when comparing between different routing possibilities.

Another approach to achieving per-class performance goals, called fragment fencing [Brown 93a], uses disk buffer allocation to explicitly manage buffer hit rates. Fragment fencing maintains per-class statistics on database reference frequencies and observed hit rates, and uses them to determine a minimum number of memory resident pages for each "fragment" of the database. Fragments are normally files or subsets of file pages that have relatively uniform access probabilities. This algorithm uses a feedback mechanism guided by simple predictions to determine the number of memory resident pages for each database fragment required to achieve the response time goals of each class. A limitation of this algorithm is that it cannot satisfy goals for classes which have low buffer hit rates (e.g. sequential scans of infrequently accessed files).

The adaptive memory allocation and MPL adjustment algorithm described in [Mehta 93] showed the importance of integrating memory allocation and load control to achieve good performance. While it does not specifically accept response time goals, the adaptive algorithm is of interest 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 the observed response time to the best possible response time (as obtained by running single queries of that class alone in the system). Fairness is then defined as the absence of variance in this metric across all classes, so the adaptive algorithm's objective of maximizing fairness is similar to minimizing the maximum performance index. 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 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.

IBM's MVS operating system has long provided mechanisms for achieving per-class performance goals [IBM 93]. MVS performance goal specification is much more complicated than simply indicating the desired response times for each class, but the net effect is still the same. Much like the adaptive algorithm of [Mehta 93], the System Resources Manager (SRM) component of MVS uses a set of heuristics to guide a feedback mechanism that determines MPLs and working set sizes for each class. SRM is more aggressive than the adaptive algorithm in achieving its goals – it will swap out active processes (along with their virtual address spaces) in order to achieve its objectives. Swapping out active transactions is an action that may not be desirable (or even possible) in the context of a DBMS, where transactions may need to be aborted in order to free up their resources. The cost of rolling back aborted transactions would likely limit the extent to which this technique could be used in a DBMS. The SRM does not understand disk buffer memory and in this respect it is again similar to the adaptive algorithm.

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." 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 data 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 algorithm described and analyzed in this paper is a first step in this direction.

3 Background

This section provides some background information that will define the problem of goal-oriented MPL and memory management and illustrate the difficulties that must be addressed in order to solve this problem. First, the terms *performance goal, MPL knob,* and *memory knob* are defined. The problem of goal-oriented MPL and memory management is then described, and a solution strategy is outlined. The section closes with an investigation of the MPL and memory management knobs and their effect on query response times using a simulated complex workload.

3.1 Goal Specification

While there are many possible ways to specify a set of performance goals, they will be defined for our purposes as follows: For each workload class, the DBMS will attempt to maintain a user-specified average response time. Because the start and length of the period over which average response times are computed greatly determines whether this goal can be met, we will examine both average and transient response times when evaluating our algorithm. If a response time goal is not specified for a workload class, then the DBMS is expected to "do its best" with respect to that class. While it may be desirable from an administrative point of view to define more than one such "no-goal" class, in this paper we will assume that all no-goal work is collected into a single no-goal class. Note that because we want to provide the best possible performance for the no-goal class, we need to allocate resources in a way that not only meets the performance objectives of the goal classes, but that also minimizes response times for no-goal transactions as well.

We will assume throughout this paper that the system is configured such that it is possible to satisfy the goals for all classes in steady state. In other words, the system does not operate in "degraded mode" [Nikolaou 92] except possibly for transient periods. This "non-degraded mode" assumption is important because it allows us to avoid situations in which the algorithm must decide which class (or classes) should be sacrificed so that others may meet their goals. Realistically, such decisions about the relative importance of each class must come from system administrators as part of a more detailed goal specification. Here, however, we restrict our attention to the more likely scenario in which the specified goals are realistic (on average) with respect to the given configuration.

3.2 The MPL and Memory Knobs

Most current database management systems provide only a single, system-wide MPL knob which is set statically (if they provide one at all). The objective for setting a system-wide MPL knob is to find the "ideal" point between under-utilizing and over-utilizing DBMS resources. A static system-wide MPL knob is not a sufficient mechanism for achieving per-class response time goals, however. Each class will likely have different goals and/or resource demands and must therefore be controlled individually. Our work uses a dynamically adjustable MPL knob for each class. The objective for setting such a per-class MPL knob is not only to prevent over-utilization of resources, but also to achieve each class's response time goal. Unfortunately, it may not be possible to produce a particular average response time value by turning the MPL knob alone. MPL limits are discrete integers and, holding other knobs constant, they will result in a discrete set of response times – none of which may actually equal the goal.

Database management systems typically provide memory knobs as well, but these too are normally system-wide knobs. As was the case for MPL, our work will use a memory knob for each class. We consider two types of classes whose response times can be reduced by allocating additional memory beyond the minimum required to execute: *disk buffer* classes and *working storage* classes. Disk buffer classes benefit from additional memory by experiencing increased buffer hit rates on the database pages that they reference. In contrast, working storage classes use memory for "computational" purposes (e.g. join hash tables or sort work areas); their benefit comes from query processing algorithms that can eliminate disk I/Os in exchange for additional memory, even in the case of negligible buffer hit rates on their input files. While classes certainly exist that benefit from both disk buffers and computational memory, we will assume that such classes can be categorized on the basis of which type of memory is more important in controlling their performance. We will not address classes whose response times are insensitive to memory allocation in this paper, as mechanisms other than memory allocation and multiprogramming levels are required to control the performance of such classes.

3.3 Problem Statement

The objective of goal-oriented MPL and memory management is to find the <MPL, memory> pairs for each class c $<mpl_c, mem_c>$ that allow every class to achieve its goal and leave the largest amount of left-over resources available for the no-goal class. Finding such a set of pairs is a difficult task for a number of reasons, the foremost being the *interdependence* between classes. Classes are interdependent because their response times are determined not only by their own MPL and memory settings, but also by the amount of competition that they experience at shared resources (processors, memory, disks, locks, etc.). The amount of competition seen by a class is, in turn, determined by the MPL and memory settings of *all other* classes. Thus, the response time of any given class is determined both by its own MPL and memory settings and by the settings of all other classes as well. More formally,

$$resp_c = f_c(\overrightarrow{MPL},\ \overrightarrow{MEM})$$

$$\sum_{\forall c} mem_c \leq M$$

where $MPL[c] = mpl_c$ and $MEM[c] = mem_c$, and M stands for the total amount of memory that is available for allocation. Note that each class has a 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 $\overrightarrow{MPL}$ and $\overrightarrow{MEM}$ vectors that 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.

3.4 A Per-Class Solution Strategy

Looking at the problem of goal-oriented MPL and memory management from an implementation perspective, we can classify possible solutions to this problem in one of two ways: algorithms with a *system-wide orientation* and algorithms with a *per-class orientation*. A system-wide orientation means that the algorithm is activated on a global basis (e.g. on a timer pop, or 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. 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 workloads, where response times can 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, we will adopt a per-class orientation because of its superior responsiveness. Additional heuristics can be 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 problem. Instead of having to find the $\overrightarrow{MPL}$ and $\overrightarrow{MEM}$ vectors that achieve the goals for *all* classes in the system, we can independently search for each class's solution (i.e. an $<mpl_c, mem_c>$ pair that achieves its goal). Even for a single class, however, it is difficult to predict response times as a function of MPL and memory. One method that is guaranteed to find a solution is to exhaustively search the entire solution space, trying every possible <MPL, memory> combination. An exhaustive approach may actually be feasible if the search space is small, but obviously becomes too time consuming in the case of multiple knobs where there can be hundreds or thousands of possible combinations. Because our search space is so large, we will settle on a feedback approach which is controlled by heuristics and simple estimation techniques. The heuristics and estimates are used to get "in the ball park", while the feedback mechanism is necessary because any heuristic or prediction will eventually fail on such a complex problem; the system must be continuously observed to insure that the outcome of any knob adjustment is actually what was intended.

3.5 The Effect of MPL and Memory on Response Times

Any feedback mechanism is based on a continual process of observing, adjusting knobs, observing, adjusting, etc. The success of this process is dependent on the feedback *controller*, whose job is to translate observations into the knob adjustments that will eventually achieve the goal. It is much easier to develop good controllers for feedback mechanisms that adjust a single knob (such as those in [Brown 93a] and [Mehta 93]), as the search space is one dimensional; the only decision required is whether to turn the knob "up" or "down." We have two knobs, however, and are thus faced with a two-dimensional search space. In order to design a controller that can move efficiently through this space toward the class's goal, we need to have some idea of how the different points on a two dimensional <MPL, memory> grid relate to response times. In the remainder of this section, we will explore this relationship empirically using a simulated multiclass workload. These simulations will provide some of the background information needed to develop our two-dimensional <MPL, memory> feedback controller.

The simulated workload and configuration that we use here, and throughout the paper, are explained in greater detail in Section 5. For the purposes of this section, a brief overview will be sufficient. The configuration consists of a single 25 MIPS processor, 8 MB of memory, and five 1 GB disks.[2] The workload consists of three classes: "queries," "transactions," and "big queries." The query class represents a consumer of working storage memory. It consists of binary hybrid hash join queries [DeWitt 84] whose performance is related to the amount of memory allocated for their join hash tables. File sizes referenced by the query class are chosen such that their join hash tables can consume 20% of the configuration's memory at their maximum allocation.[3] The transaction class performs random single-record lookups on four common files via B+ tree indices, modeling a disk buffer class. 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 its own unique set of database files, and all files are horizontally partitioned (i.e. fully declustered) across the five disks.

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 requirements. Figures 1 and 2 show two different representations of the query class response times that result from various MPL and memory combinations. Figure 1 shows query response times as a function of memory *per query* (with each query receiving the same allocation), while Figure 2 shows response times as a function of memory *per class* (i.e. 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.

MPL limit	Mem per query (MB)	Query class mem (MB)	Disk util
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 1: 130 sec goal solution characteristics

The most significant phenomenon shown by Figures 1 and 2 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 1 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. For example, the first solution in Table 1 (with an MPL limit of one) consumes 1.64 megabytes of memory and results in a 49% disk utilization, while the last solution (with an MPL limit of four) consumes only half as much memory

[2]This configuration is scaled up in later experiments.

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

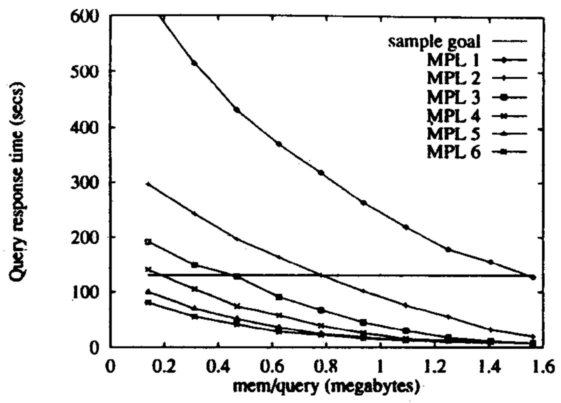

Figure 1: MPL and memory per query

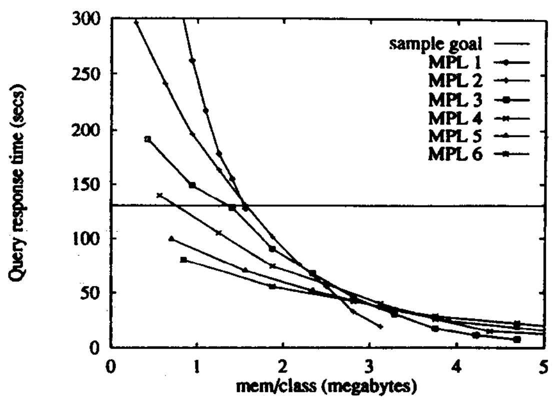

Figure 2: MPL and memory per class

but results in a disk utilization of 75%.

How should a feedback controller decide which of these solutions is the "best" one? This is an important decision because the choice of a solution for any one class will determine the level of competition seen by other classes at shared resources, and thus will also indirectly determine their set of feasible solutions as well. Among those listed in Table 1, 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 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 feedback controller must still understand how the memory and MPL knobs effect response times in order to use those knobs to move a class toward its response time goal. Figures 1 and 2 illustrate this relationship between MPL, memory, and response times. Figure 1 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 1 (larger memory allocations). There, MPL increases (especially beyond two or three) have less of an effect on response times than MPL increases at smaller memory allocations do. This is because execution times are improved enough by the increased 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 2 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 2, we can see that running two queries 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 2, 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. Our algorithm will exploit the Cornell and Yu results, as will be explained in Section 4.

While space limitations prevent a detailed description of the other simulated workloads that we have explored, we can summarize our findings as follows. First, we observed that in nearly all cases, multiple solutions will exist to a class's response time goal. 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 in the system. As a result, a feedback 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 whether response times will increase or decrease.

4 M&M: Goal-Oriented MPL and Memory Management

This section describes M&M, our algorithm for goal-oriented *M* PL and *M* emory management. M&M's key components are a general feedback mechanism and controllers that set the MPL and memory knobs for each class. Before describing these components in detail, we first define a few basic memory management principles that are used by M&M. The first principle is that both working storage *and* disk buffer memory are allocated out of a *single* shared memory pool that is managed by M&M. Without this unification, it would be difficult for M&M to manage the trade-offs between the two types of memory usage. Out of this single shared pool, M&M sets aside a small portion of available 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.

M&M associates a memory pool with each goal class that represents the amount of memory required by the class to meet its goal. The size of this pool 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 the set-aside areas and the goal class pools from the total available memory is called the *unreserved pool*, which is made available to the no-goal class. Any increase in the pool size for a goal class is taken from the unreserved pool, and any decrease is given back to the unreserved pool. If the unreserved pool is empty, no pool increases are allowed, and the no-goal class is forced to execute at its minimum memory requirement (using memory from the set-aside area).

4.1 Feedback Mechanism

M&M's feedback mechanism is largely based on the feedback mechanism of the fragment fencing algorithm, which is described in detail in [Brown 93a]. We will give a brief overview of M&M's feedback mechanism here, primarily highlighting the differences between M&M and fragment fencing.

M&M analyzes the performance of each class independently, adjusting the memory or MPL knobs if required, at well defined intervals. The length of these per-class intervals is determined by a predefined number of transaction completions, and is thus unique to each class. The number of transaction completions is chosen to strike a balance between the need for statistically significant samples (where more completions are better) and the requirement for responsive behavior (where fewer completions are better). M&M's decisions are based on statistical observations computed over the current interval or on exponentially weighted averages of current and past interval statistics (depending on the particular statistic).

To determine whether a class is meeting its goal, its average observed response time is compared to the response time goal for the class. If the observed response time is within some *tolerance band* around the goal (i.e. plus or minus some percentage of the goal), then the goal is considered satisfied. Otherwise, the controller for the class will be invoked to adjust its memory and/or MPL knob(s), and the goal will be checked again after the changes take effect. M&M automatically computes the tolerance band around each class's goal by observing the "natural" variance in response times for each class in steady state. Higher variance will result in a wider tolerance band. Automatically computing a tolerance band for each class allows classes with lower variance to be controlled more aggressively and prevents the algorithm from attempting to manage natural statistical fluctuations within a class.

Because statistics on class behavior will not exist upon a system "cold start", M&M defines a warm-up period for each class of one interval in length in order to allow these statistics to be accumulated. Because of the initial lack of statistical data, M&M cannot decide what the MPL limits for each class should be during the warm-up period. A system administrator must therefore supply the initial MPL limits to M&M on a cold start. For the simulated workloads in this paper, we use a cold-start MPL limit of two for working storage classes and an infinite MPL limit 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.

4.2 Disk Buffer Class Controller

M&M adopts the fragment fencing algorithm [Brown 93a] to control the memory knob for disk buffer classes. Fragment fencing uses observed response times and hit rates for each class to set a minimum number of pages that must remain memory resident for each database fragment[4] referenced by a class; these minimums are called *target residencies*. The buffer manager's native page replacement policy is modified to prevent a page from being chosen for replacement if replacing it would bring the number of memory-resident pages for its fragment below the fragment's target residency. If a class's response time goals are being violated, fragment fencing responds by increasing the target residencies for fragments referenced by the class. Conversely, if goals are being exceeded, target residencies will be lowered. A disk buffer class's pool size is computed as the sum of the target residencies for every fragment referenced by the class.

M&M takes the simplest possible approach in setting the MPL limit for disk buffer classes – it sets them all to infinity. The rationale for this choice is that the actual amount of memory needed by a disk buffer transaction at any particular moment is very small. For example, an index scan requires at most one page per index level and one or a few pages for the indexed data file. Thus, the cost of admitting an additional disk buffer class transaction is very low in terms of the minimum memory required, making it feasible to run disk buffer classes at very high multiprogramming levels without causing a high degree of memory contention.

While fragment 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 fragment fencing is that memory is the bottleneck resource, so it always tries to lower a class's target residencies to the *minimum* possible amount that can achieve its response time goals (i.e. it favors "low memory/high disk" solutions). 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.

We account for this inter-class dependency by allowing M&M to modify fragment fencing's assumption that memory consumption must be minimized. M&M does this by requesting that disk buffer classes enter an *exceed mode*. A disk buffer class in exceed mode will increase its target residencies (and pool size) in order to increase its buffer hit rates and decrease its disk utilization. The 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 no longer the primary reason for violating the goal (i.e. memory or MPL are once again the primary factors), or the unreserved pool is exhausted (i.e. the request for disk utilization reduction failed). As long as a disk buffer class is in exceed mode, fragment fencing will not shed the "excess" memory that is causing its response time goal to be exceeded.

4.3 Working Storage Class Controller

As Section 3 has shown, the huge number of possible <MPL, memory> combinations, the complex relationship of MPL

[4]As mentioned in Section 2, fragments are normally files or subsets of file pages (e.g. one level of an index tree) that have relatively uniform access probabilities.

and memory to response times, and the existence of multiple solutions to a single goal make the design of a working storage controller very challenging. However, Section 3 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 will allow response times to be "fine-tuned" using the MPL knob.

4.3.1 Working Storage Class Controller Heuristics

The most important heuristic was suggested by the discussion of Figure 2: 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. 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.[5] These results form the basis for the first heuristic:

Heuristic 1 *Allocate the maximum memory required by each individual query if this is 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 previously in Figure 1: As MPL 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 point as an average MPL wait queue length of less than 0.5. In other words:

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; 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, if there is no other option, M&M may choose to ignore its disk queue length limiting heuristic. Thus, the third heuristic is:

[5]The Cornell and Yu results apply to hash-based, sort-merge, and nested loops join methods.

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 and how many should execute at max. The final heuristic deals specifically with one effect of an increase in the number of min queries, namely, a corresponding increase in the probability that an arriving query will be allocated its minimum memory requirement. Because a min query requires many more I/Os than one running at max (roughly three times as many in the case of a hash join), any increase in the probability of a minimum allocation for a 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 far outweighs the risk that it will dismiss a possible solution. Thus, our final 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.*

4.3.2 Determining a New <MPL, Memory> Setting

Depending on the current state of the system, the working storage class controller will take one of four actions when invoked by M&M's feedback mechanism 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.

This action is accomplished as described in Section 4.2.

`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 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 3 shows how M&M

decides what action to take and highlights the rational behind these decisions.

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

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

if (there are no disk buffer classes OR
    a previous request failed to reduce disk queue lengths) then
    disksFull = FALSE;// 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++,
                // no disk problem exists so disk-- is useless
elseif (avgMax < URPool) then
    max++;      // heuristic 1 says try to maximize return
                // on memory consumption
eLeif (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, heuristic 4 says min++ may hurt
endif
```

Figure 3: Algorithm to determine new <MPL, memory>.

4.3.3 Non-Integral MPL Limits and MPL Reductions

As we saw in Table 1, solutions to a particular response time goal will 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. If we could somehow set the MPL knob at *non-integral* settings, however, then we could fine-tune response times and find a solution 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 goals are exceeded, and then delaying the admission of the next query by an amount of time that is equal to the amount by which the previous query exceeded its goal. No delay is used 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 Section 4.3.2 is invoked to find the first <MPL, memory> setting that exceeds the response time goal for a class; this setting is called the *home*. During a home search, the delay mechanism is turned off. Once a home is found, the delay mechanism is 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 would be 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.

4.4 Controlling No-Goal Classes

Memory for no-goal class transactions is allocated from the unreserved pool. Thus, each individual no-goal transaction competes for unreserved pool memory on a first-come, first-served basis. If enough memory is available in the unreserved pool to run a no-goal transaction at its maximum demanded memory, then it is allocated its maximum. Otherwise, it takes whatever is available (down to its minimum requirement). Note that even if the unreserved pool is empty, there should still be enough memory available in the *set-aside area* to allow a no-goal query to execute at min.

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 because there is no real basis for selecting appropriate resource allocations without a goal. For our initial version of M&M, we therefore simply limit the no-goal class to an MPL of one at all times.

4.5 Algorithm Summary

In this section, we quickly summarize M&M and describe its policy for allocating memory to individual disk buffer and working storage transactions upon admission into the DBMS. For disk buffer classes, the feedback mechanism periodically compares the observed average response time of the class against its goal. The disk buffer controller is called to revise the target residencies and pool size for the class if it is violating or exceeding its goal. No adjustment is needed if the average response time falls within the dynamically computed tolerance band around the goal, or if the class has entered *exceed mode* and is exceeding its goals in order to reduce disk utilization. Individual disk buffer transactions are admitted immediately upon arrival, and they compete freely for any available (unpinned or unfenced) memory.

The feedback mechanism for working storage classes will periodically monitor response times as well, and will invoke the working storage controller if the class's goal is exceeded or violated. The working storage controller will then adjust one of the three knobs for the class: the number of queries allowed to execute with their maximum memory requirement (*maxMPL*), the number of queries allowed to execute with their minimum memory requirement (*minMPL*), and the pool size for the class. It may also request that disk buffer classes enter exceed mode in order to reduce disk response times. An arriving working storage query is allocated max, in-between, or min, based on the number of queries currently executing at those allocations versus the number allowed.

File name	# recs (x 1000)	rec size
big file	1,600	100
big index	1,600	16
medium file	640	100
medium index	640	16
small file	320	100
small index	320	16
tiny file	8	100
tiny index	8	16
40% query files	10	200
80% query files	41	200
200% query files	102	200

Table 2: Database characteristics

Parameter	Value
# Transaction terminals	100
Mean tran think time	10 sec
# Query terminals	60
Mean query think time	150/1300/3200 sec
# Big Query terminals	2
Big Query think time	0 sec
Number of CPUs	1
CPU speed	25 MIPS
Number of disks	5
Page size	8 KB
Memory size	8 MB (1024 pages)
Disk cylinder size	83 pages
Disk seek factor	0.617
Disk rotation time	16.667 msec
Disk settle time	2.0 msec
Disk transfer rate	6.0 MB/sec
Disk cache context size	4 pages
Disk cache size	24 contexts

Table 3: Simulation parameter settings

Function	# Instr
read rec from buff page	300
write rec to buff page	100
insert in hash table	100
probe hash table	200
test index entry	50
copy 8K msg	10000
start an I/O	1000
apply predicate	100
initiate select	20000
terminate select	5000
initiate join	40000
terminate join	10000

Table 4: Simulation instruction counts

5 Simulation Model

This section provides a detailed description of the simulation model that we will use for evaluating the M&M goal-oriented MPL and memory management algorithm. The external workload source for the system is modeled by a set of simulated terminals. Each terminal submits a stream of transactions of a particular class, one after another. In between submissions, each terminal "thinks" (i.e. waits) for some random, exponentially distributed amount of simulated time. The number of terminals and the think times used in this study were chosen to provide an average disk utilization of 50 to 60% under normal operating conditions.

The simulated disks are modeled after the Fujitsu Model M2266 (1 GB, 5.25") disk drive. Our simulated disk provides a 3/4 MB cache that we divide into twenty-four 32 KB cache contexts for use in prefetching 8K pages for sequential scans. In our model of the disk, which is a slight simplification of the actual Fujitsu disk, the cache is managed in the following manner: Each I/O request, along with the required page number, specifies whether or not prefetching is desired. If so, one context's worth of disk blocks (4 blocks) are read into a cache context after the originally requested data page has been transferred from the disk into memory; the requester is not released until the entire cache context is loaded (i.e. synchronous cache loading is assumed). Subsequent requests for one of the prefetched blocks can then be satisfied without incurring another I/O operation, and a simple round-robin replacement policy is used to allocate cache contexts if the number of concurrent prefetch requests exceeds the number of available cache contexts. The disk queue is managed using an elevator algorithm.

The CPU is scheduled using a round-robin policy with a 5 millisecond time slice. The buffer pool consists of a set of main memory page frames of 8K bytes each. The buffer manager's page replacement scheme is a modified global LRU scheme augmented with three levels of hints. The hints are given by the query execution operators when a page is unfixed, and they define three levels of value as follows: index pages are considered more valuable than data pages, and randomly-accessed data pages are considered more valuable than sequentially-accessed data pages. A memory reservation mechanism allows query execution operators to reserve memory for their working storage, thus preventing those reserved frames from being stolen while their reservation is in effect. This function is used by hash join operators to reserve memory for their hash tables.

The database itself 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 2 lists the files and indices used for all of the experiments in this study. The large, medium, small, and tiny files and indices are used by the "transaction" class (described next). 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.

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 size 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.

The simulated workload for most of this study consists of three classes: "transactions," "queries," and "no-goal queries." "Transactions" represent a disk buffer class with short (subsecond) execution times. They perform single-record index selects on 4 files: big, medium, small, and tiny (see Table 2). 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. We fix the number of transaction terminals at a population of 100 with exponentially distributed think times having a mean value of 10 seconds.

The "query" class models a working storage class with longer execution times (tens of seconds or minutes). The individual queries consist of binary joins of two randomly chosen query files (see Table 2). We use the hybrid hash join algorithm [DeWitt 84] 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 2). The query class terminal population is set at 60 with 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. These think times were chosen to insure that MPL limits up to five or six could be explored before triggering the MPL limiting heuristic (heuristic # 2). Note that the randomness in both the arrival process and memory demand for queries results in a high degree of variance in resource demands within the class.

The last class, the "no-goal query" class, is identical to a "query" class which 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) – since 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 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, we fix the terminal population for no-goal queries at two, with no think time; recall that their MPL limit is fixed at one by M&M.

The important parameters of the simulated DBMS are listed in Tables 3 and 4. Our system is certainly underconfigured with respect to memory; the choice of 8 MB was made purely to achieve tolerable simulation times. We thus include an experiment where we scale up both memory and query file sizes by a factor of 8 (to 64 MB). The 25 MIPS CPU results in CPU utilizations of 50-75%. The number of disks, number of terminals, and think times were chosen to ensure that disk utilizations lie in the 50 to 60% range and to support a reasonable range of feasible multiprogramming levels. The software-related parameters in Table 4 are based on instruction counts taken from the Gamma parallel database system prototype [DeWitt 90], and the disk characteristics approximate those of the Fujitsu Model M2266 disk drive, as stated earlier.

6 Experiments and Results

In this section, we use our simulation model to examine how well M&M can achieve a variety of goals for several variations of a simulated multiclass workload. Each version consists of transactions, queries, and no-goal query classes, as described in Section 5. The difference between each variation is in the average memory demanded by the query class. We will examine workloads where the average maximum query memory demand is 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 ten simulated hours, and the 200% version is executed for twenty hours. We collect and report response time statistics 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. We also include a transient analysis of response times to indicate how the algorithm operates over the entire range of simulated time. For all experiments, we ensure minimums of 170,000 transaction completions, 500 query completions, and 350 no-goal query completions.

The performance metrics that we will use for judging M&M's behavior are the *performance index* of each class and the average response time of the no-goal query class. The performance index of a class is defined as the average response time of the class (over the statistics collection period) divided by its response time goal. Thus, a performance index of one is ideal, an index greater than one indicates a goal violation, and an index less than one indicates that the goal has been exceeded. We use the no-goal class response times 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.

In order to set "reasonable" goals for these workloads, we first ran a series of simulations that explored a wide range of static *minMPL*, *maxMPL*, and pool size settings for the query class in each version of the workload (i.e. for each different average query memory demand). The no-goal class's memory allocation was set at its minimum requirement for these static simulations, and the transaction class was given all of the remaining memory. Using the response times that resulted from these simulations, we then derived a set of per-class goals that spanned a range from "loose" to "tight."

Qry goal (secs)	Tran goal (msecs)	Qry perf index	Tran perf index	No-goal resp (secs)
100	350	0.98	0.78	45
100	275	0.98	0.97	50
50	300	0.97	0.97	61
15	275	0.92	0.99	93
8	300	1.06	0.98	86
8	250	1.08	1.08	130

Table 5: 10-30% memory demand workload

Table 5 shows the results from the base case experiment with per-query memory demands ranging from 10-30% of the configuration memory. Each row represents a different combination of goals for the transaction and query classes. The performance indices in Table 5 show that, except in the case of very tight or very loose goals, both the transaction and query classes are kept to within a few percent of their goals. The first row represents a very loose goal for the transactions (350 msecs). The disk buffer controller (fragment fencing) initially decided that no memory was required to achieve this goal, but its low memory/high disk solution created disk response times that were too high to meet the goal for the query class. M&M then placed the transaction class in exceed mode in order to lower disk response times to a point that allowed the query class to meet its goal. This type of behavior is the reason that loose transaction goals are more likely to be exceeded than tighter transaction goals.

The last row represents an unachievable goal and is used to show how M&M balances resources across the classes in a degraded mode of operation. The response times generated by M&M for this unachievable goal (8.64 seconds, 270 milliseconds, and 130 seconds) are comparable to the best[6] response times achievable with static settings for the query class's *maxMPL*, *minMPL* and pool size (7.6 seconds, 290 milliseconds, and 139 seconds). Note that the no-goal response times are best for the loosest goals (at the top of the

[6]Defining the "best" set of response times for a multiclass workload is a subjective process, since one can usually achieve a better response time for one class by degrading the response times for other classes. Our criteria for selecting the "best" statically obtained response times tries to choose a set of response times which minimizes the distance to the best case response time for each class. There are certainly other valid criteria for making this choice.

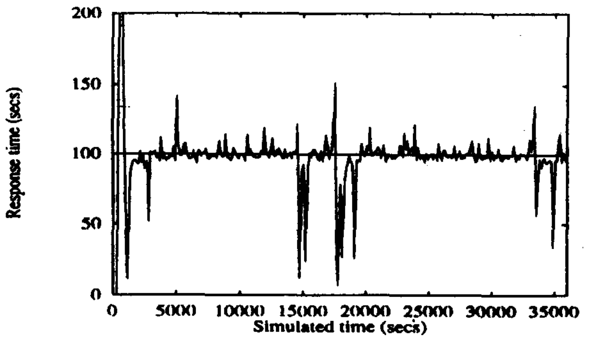

Figure 4: Query resp, 10-30% queries, (100, 275) goal

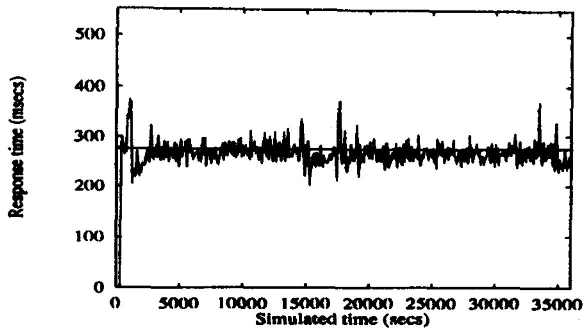

Figure 5: Tran resp, 10-30% queries, (100, 275) goal

table), degrading progressively as the goals tighten.

To examine M&M's transient behavior for this workload, we select the (100, 275) goal pair and graph the average interval response times for both query and transaction classes as a function of time in Figures 4 and 5, respectively. Turning first to the query response times in Figure 4, we can see 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 MPL for the query class. During this adjustment, the delay mechanism was turned off and queries temporarily exceeded their goals by a large amount because the delay had been removed. Since this workload runs with a significant delay (the actual MPL is only 2.1 with an integer MPL limit of 3), a sharp downward spike occurs in the query response times any time the delay is turned off for a knob adjustment. Looking at the transaction response times for this workload in Figure 5, we can see upward spikes at the same points where the query class has temporarily increased its MPL. These MPL increases caused a temporary increase in disk response times for the transaction class. The transaction class compensated for these increases almost immediately, but the transient upward spikes in their response times were unavoidable.

Qry goal (secs)	Tran goal (msecs)	Qry perf index	Tran perf index	No-goal resp (secs)
400	250	0.93	0.98	69
200	350	0.99	0.82	76
200	275	1.01	0.97	87
100	300	1.07	0.93	92
100	275	0.98	0.98	91
50	300	1.12	0.95	102

Table 6: 40-120% memory demand workload

Table 6 shows the results of the base workload when per-query memory demands are larger, ranging from 40-120% of the configuration memory. The results are similar to the previous experiment. Most of the goals are met to within a few percent, except for the last (tightest) goal pair which is unachievable. As before, no-goal response times degrade as the goals get tighter. For the unachievable goal pair, M&M's response times of 56 seconds, 285 milliseconds, and 102 seconds again compares favorably with the best statically obtainable response times: 48 seconds, 400 milliseconds, and 120 seconds. Table 6 shows an unusual result, however; query goals are violated by 7% for the (100, 300) goal pair, while the tighter (100, 275) goal pair is achieved. To explain this violation, we turn to the graph of the transient query class response times for the (100, 300) goal case in Figure 6. Since the average maximum memory demand for this query class is 80% of 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 memory demand to produce a large response time variance for this class, as is shown in Figure 6.

The aforementioned variance in response times for the 40-120% queries has three effects: The first is that M&M computed a much larger tolerance band for these queries (up to ±45%) than it did for the 10-30% queries (±9% at most). While this 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 response times by adjusting one or more 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 query response time, leaving the query class MPL untouched.

The second effect of the large variance is its impact on the average response time statistic. Since our average response time statistics are collected in the second half of the simulation (everything to the right of 17,500 seconds in Figure 6), they include the upward spikes but exclude the largest downward spike in the figure. If the statistics collection window had had started earlier (e.g. around 10,000 seconds), the downward and upward spikes would have canceled and our average response time statistics would have shown that this goal had been achieved. This analysis clearly shows how average response time statistics are sensitive to the positioning of the interval over which they are collected.

A final effect of the increased variance in the query class memory demand can be seen by examining the transaction class response times in Figure 7. Comparing this graph to that of the previous workload (Figure 5), we can see that the transaction response times have a much higher variance for this workload as well. This is because the fluctuations in physical memory demanded by 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.

Table 7 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 last (unachievable) goal. For this workload, we select the unachievable goal pair (200, 350) for our examination of transient behavior, in Figures 8 and 9. Disk queue lengths are higher for this workload than

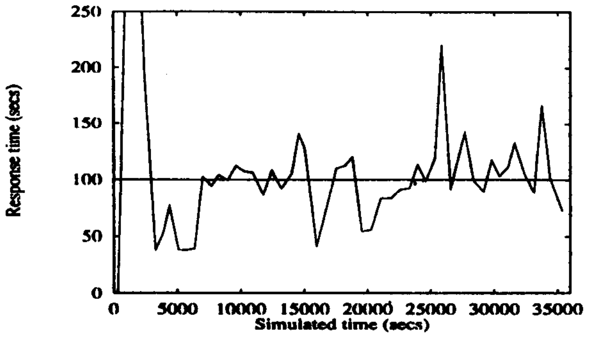

Figure 6: Query resp, 40-120% queries, (100, 300) goal

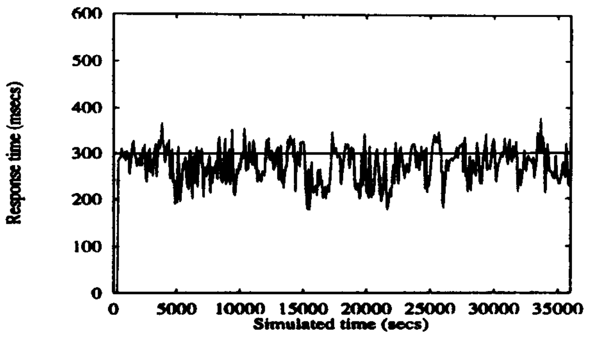

Figure 7: Tran resp, 40-120% queries, (100, 300) goal

Qry goal (secs)	Tran goal (msecs)	Qry perf index	Tran perf index	No-goal resp (secs)
800	250	0.94	1.01	102
600	350	0.96	0.94	53
600	275	0.94	0.98	73
400	350	1.00	0.95	55
400	300	1.03	0.97	65
200	350	1.14	0.95	97

Table 7: 100-300% memory demand workload

the previous two, as these queries spend a larger portion of their execution time writing tuples out to intermediate files (writing is much more expensive that reading because of the ineffectiveness of the disk cache). In order to lower disk queue lengths, M&M requested a disk utilization reduction from the transactions, but this was not successful because the real culprit was the query class. This failed attempt at disk queue length reduction is the reason for the dip in transaction response times in Figure 9 that occurs from 5,000 to 7,500 seconds. Once this attempt failed, M&M ignored the disk queue length limiting heuristic and proceeded to increase the query class MPL. At about 25,000 seconds into the simulation, the query class MPL was raised high enough (to an MPL of five) to eliminate most MPL waiting; at this point the only available action was to increase the query class pool size and let one of the five queries execute in between min and max. The query class's pool size knob was then increased until the unreserved pool was exhausted at about 60,000 seconds into the simulation. At this point, the query goals appear to be finally achieved; since the average response time measurements start at 35,000 seconds, this "success" is not reflected in that measure, however.

Our next experiment tests how well M&M can satisfy goals for a more complex workload consisting of two working storage classes, two disk buffer classes, and a no-goal class. The 60 terminals of the query class used in the base case experiments are split into two query classes of 30 terminals each, and the 100 terminals of the transaction class are split into two transaction classes of 50 terminals each. The transaction class files are replicated so that each transaction class accesses its own files. The same set of files are used by both query classes, and they result in an average memory demand of 20% of configuration memory (with individual query demands ranging from 10-30%).

Table 8 shows various combinations of goals for the more complex workload, arranged in order from loose to tight as determined by the no-goal class response times. There is one case where goals are violated by more than 5%: in goal set #4, transaction class #1's 250 msec goal is violated by 13%. This goal is violated because disk response times are too high to achieve such a tight goal with five classes executing concurrently. However, M&M does achieve the goals for the other three classes in this case. Transaction class goals are exceeded to a greater extent for this workload (by up to 30%) than for previous workloads. As explained previously, looser transaction goals are more likely to exceed than tighter ones, and there are more instances of loose transaction goals in this workload than there were in previous workloads.

While transaction classes may exceed their goals in order to reduce disk response times for other classes, there is no reason why query class goals should be exceeded by a large amount. However, there are two cases in Table 8 where query goals are exceeded by more than 10%: the performance indices of 0.87 in goal sets #1 and #3. While space limitations prevent us from showing the transient response times for these cases, the reason that these goals were exceeded is due to the phenomenon that was shown in Figure 4. Both of the exceeding classes are operating very close to the MPL reduction trigger (*execMPL* $<$ *MPLLimit* -1); 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 4 occur here. (Clearly, the MPL reduction mechanism should be tuned further to reduce the probability of "MPL wobbling.")

Our final experiment verifies that our favorable results for M&M can scale up to larger memory and query file sizes. For this experiment, we increased both the configuration memory and query file sizes by a factor of eight (increasing memory to 64 MB). We then re-ran the base case workload with queries that demand 20% of the configuration memory on average. While space limitations prevent us from explaining the results in detail, M&M achieved the goals for this workload and configuration as well as it did for the smaller configuration. While additional experimentation is needed, these initial studies show that M&M appears to be quite robust in the presence of differing workloads, configurations, and query memory demands.

7 Summary and Future Work

In this paper, we have defined and explored the problem of satisfying per-class response time goals for complex multiclass database workloads. We then described M&M, a feedback-based algorithm for determining the MPL and memory settings for each class independently. M&M builds on the fragment fencing algorithm [Brown 93a] for managing disk buffer classes, adding new mechanisms for controlling working storage classes; these include a heuristic-based controller for determining MPL and memory allocations and an admission delay

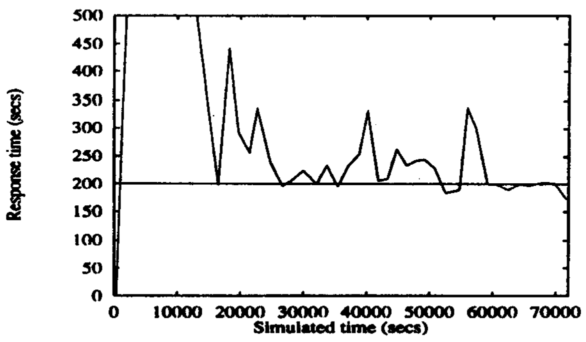

Figure 8: Query resp, 100-300% queries, (200, 350) goal

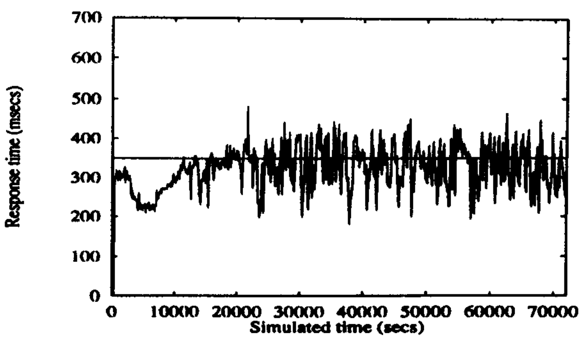

Figure 9: Tran resp, 100-300% queries, (200, 350) goal

Goal set #	Query 1 goal secs	Query 2 goal secs	Tran 1 goal msecs	Tran 2 goal msecs	Query 1 perf idx	Query 2 perf idx	Tran 1 perf idx	Tran 2 perf idx	No-goal resp secs
1	150	200	400	400	0.92	0.87	0.70	0.68	64
2	10	200	350	400	0.90	1.04	0.91	0.78	64
3	50	100	275	350	0.87	0.91	0.99	0.83	96
4	150	175	250	400	1.05	1.03	1.13	0.74	122
5	50	75	275	300	0.95	0.91	1.01	0.99	128

Table 8: More complex workload (10-30% memory demand)

mechanism that allows M&M to set non-integral MPL limits. M&M's per-class orientation has significant advantages in simplicity and responsiveness, while its "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 a detailed simulation model, we explored the steady state and transient performance of M&M and showed that it can achieve goals for a variety of different workloads, configurations, memory demands, and degrees of variance within each goal class.

For future work, we plan to experiment further with different workloads and configurations. In addition, we would like to make several enhancements to M&M, such as fine-tuning its MPL reduction trigger to behave in a more stable manner, developing a less ad hoc heuristic for limiting the number of queries that execute at min (heuristic #4), and allowing no-goal class MPLs to rise above one. Beyond these enhancements, our future work will integrate M&M with goal-oriented processor and disk scheduling mechanisms and will exploit memory adaptive query processing techniques (such as the preemptible hash join and sorting methods of [Pang 93]). Finally, we would like to explore other approaches to specifying goals for low throughput classes. Response time goals for low throughput classes result in long search times for the appropriate solution, as they require a certain number of completions to achieve statistical significance; other, progress-oriented measures of performance are needed to address these types of classes.

Acknowledgements

The authors would like to thank the members of department K55 at IBM Almaden for the generous donation of countless workstation cycles, with particular gratitude to Manish Arya for providing MIPS beyond the call of duty. John McPherson has provided management support for this work even though it was not exactly "in plan." David DeWitt and Peter Haas provided valuable advice and counsel; Bruce Lindsay and Mary Tork Roth provided helpful comments on an earlier version of this paper.

References

[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 Perf. Transaction Sys. Workshop*, Asilomar, CA, Sept 1993.

[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 TKDE*, 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.

[Ferg 93] D. Ferguson, C. Nikolaou, L. Geargiadis, "Goal Oriented, Adaptive Transaction Routing for High Performance Transaction Processing Systems," *Proc. 2nd Int'l PDIS Conf.*, San Diego CA, Jan 1993.

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

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

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

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

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

[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.

[Weikum 93] G. Weikum et al, "The COMFORT Project – Project Synopsis," *Proc. 2nd Int'l PDIS Conf.*, San Diego CA, Jan 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.