MANAGING MEMORY FOR REAL-TIME QUERIES

HweeHwa Pang† *Michael J. Carey* *Miron Livny*
Computer Sciences Department
University of Wisconsin - Madison
Madison, WI 53706

ABSTRACT — The demanding performance objectives that real-time database systems (RTDBS) face necessitate the use of priority resource scheduling. This paper introduces a *Priority Memory Management* (PMM) algorithm that is designed to schedule queries in RTDBS. PMM attempts to minimize the number of missed deadlines by adapting both its multiprogramming level and its memory allocation strategy to the characteristics of the offered workload. A series of simulation experiments confirms that PMM's admission control and memory allocation mechanisms are very effective for real-time query scheduling.

1. INTRODUCTION

The real-time database system (RTDBS) performance objective of minimizing the number of missed deadlines can be very demanding. This is particularly so in *firm* RTDBSs [Hari90], where jobs lose all value once their deadlines expire. In order to accomplish their objective, RTDBSs employ multiprogramming so that all of their resources can be utilized productively to service incoming jobs. Moreover, RTDBSs use priority scheduling to resolve any resource contention that arises from multiprogramming. While the problem of scheduling real-time transactions has been extensively studied, largely from a concurrency control perspective, executing multiple queries that require large amounts of computational memory (e.g., hash tables for joins or tournament trees for external sorts) introduces admission control and memory allocation issues that have yet to be addressed.

Many queries can simply read their operand relation(s) once and produce results directly if given their maximum required memory. Alternatively, as long as the memory allocation of the queries meet certain minimum requirements, they can also be run with less memory by writing out temporary files and subsequently reading them back in for further processing. For instance, a hash join can either execute with its maximum required memory, which is slightly greater than its inner relation size, or it can run in an additional pass with as few buffer pages as the square root of its inner relation size [Shap86]. In order to derive the benefits of multiprogramming, it may be necessary for an RTDBS to admit some queries with less than their maximum memory allocations. If too many queries are admitted, however, the resulting additional I/Os could lead to thrashing, making high concurrency harmful instead of helpful. Multiprogramming is therefore a two-edged sword, and RTDBSs require a priority-cognizant admission control mechanism to protect them against thrashing.

Having determined which queries to admit, the next issue that an RTDBS faces is memory allocation. While the highest-priority query at a given CPU or disk will use that resource exclusively, memory must be shared among all of the admitted queries. When the total maximum memory requirement of the admitted queries exceeds the available memory, the RTDBS must decide on the amount of memory to give each query. This decision needs to take into account queries' timing requirements to ensure that urgent queries receive their required resources in time to meet their deadlines. In addition, the effectiveness of memory allocation in reducing individual queries' response times should be considered so as to make the best use of the available memory [Corn89, Yu93].

In this paper, we introduce a *Priority Memory Management* (PMM) algorithm that is designed to schedule queries in firm RTDBSs. PMM does not assume any advance knowledge of workload characteristics or query execution times, as such knowledge is usually not available in a database system. Instead, PMM controls the number of queries that may gain admission at any time by dynamically choosing a target multiprogramming level (MPL) to balance the demands on the system's memory, CPU, and disks. Moreover, PMM can either insist that queries be admitted only with their maximum memory allocations, or it can give higher-priority queries their maximum required memory while allowing lower-priority queries to run with their minimum requirements. Both the target MPL and the memory allocation policy are chosen based on past system behavior. The Earliest Deadline policy [Liu73], which gives higher priority to queries whose deadlines are more imminent, is used to guide the admission and memory allocation decisions of PMM.

The remainder of this paper is organized as follows: Section 2 briefly discusses related work, and the PMM algorithm is introduced in Section 3. A detailed simulator of a firm RTDBS, intended for studying the performance of the PMM algorithm, is described in Section 4. Section 5 presents the results of a series of experiments showing that, over a wide range of workloads, PMM offers an effective solution to the memory management problem that arises in scheduling real-time queries. Finally, our conclusions are presented in Section 6.

2. BACKGROUND

This section briefly describes several studies reported in the literature that are related to our work. We first review related work on query scheduling, and we then devote the rest of the section to the dynamic query processing algorithms that PMM relies upon as primitives.

This work was partially supported by a scholarship from the National Univ. of Singapore (NUS), and by an IBM Research Initiation Grant.

† Present address: Institute of Systems Science, NUS.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

SIGMOD 94- 5/94 Minneapolis, Minnesota, USA
© 1994 ACM 0-89791-639-5/94/0005..$3.50

2.1. Query Scheduling

While a number of studies have addressed real-time transaction scheduling [eg., Abbo88, Hari90, Huan89] and disk scheduling [Abbo89, Abbo90, Care89, Chen91, Kim91], to the best of our knowledge no work has dealt with query scheduling issues in RTDBSs. The work that is most relevant to our work here is reported in [Corn89, Yu93]. That work examined the effect of memory allocations on query response times in traditional (non-real-time) database systems, and concluded that giving some of the queries their maximum required memory, while allocating the minimum possible memory to the rest, leads to near-optimal memory usage. This result is incorporated in the memory allocation strategies of PMM.

2.2. Memory-Adaptive Query Primitives

In a priority scheduling environment such as an RTDBS, large queries involving operations like hash joins and external sorts face the prospect of having memory taken away and/or allocated to them during their course of execution. In anticipation of such memory fluctuations, this study will employ the adaptive hash join and external sorting algorithms that we found to deliver the best performance among a range of alternatives that we investigated in a recent pair of studies [Pang93a, Pang93b]. The two algorithms are briefly summarized here.

The hash join algorithm that PMM employs was introduced in [Pang93a] as *Partially Preemptible Hash Join* (PPHJ) with *late contraction, expansion*, and *priority spooling*. PPHJ splits the pair of input relations into a set of partitions, as is done in traditional hash joins as well. At any one time during join execution using PPHJ, some of these partitions may be *expanded*, i.e., held in hash tables in memory, while others are *contracted*, i.e., resident on disk. When asked by the memory manager to free up buffers, PPHJ can do so by reducing the number of expanded partitions. Moreover, if extra memory becomes available while the outer (probing) relation is being split, PPHJ can expand contracted partitions so that outer relation tuples that hash to these partitions can be joined directly and then discarded, thus avoiding some I/Os.

The external sorting algorithm that PMM employs begins by using replacement selection to split the operand relation into sorted runs; these sorted runs are then repeatedly merged into longer runs until only a single run remains. These are the usual phases of an external sorting algorithm. What makes the algorithm adaptive is that, during the merging process, an executing merge step can be split into sub-steps that fit within the remaining memory if memory reductions occur [Pang93b]. Conversely, existing merge steps can be combined into larger steps (i.e., steps that merge more runs at once) to take advantage of any excess buffers that become available.

3. PRIORITY MEMORY MANAGEMENT

In firm RTDBSs [Hari90], queries become worthless if they fail to complete by their deadlines. Consequently, the primary performance objective of an RTDBS is to minimize the number of missed deadlines without intentionally discriminating against any particular type of queries. In order to achieve this objective, resource scheduling decisions in these systems have to be priority-driven. The Priority Memory Management (PMM) algorithm is a priority-cognizant algorithm designed to regulate memory usage for firm real-time query workloads.

The PMM algorithm consists of an admission control component and a memory allocation component. Both components employ the Earliest Deadline (ED) scheduling policy [Liu73], so queries that are more urgent are given higher priority in admission and memory allocation decisions than queries whose deadlines are further away. The ED policy is adopted here, instead of policies that take into account query execution times, because (accurate) execution time information is usually not available a priori in a database system. The admission control component sets the target multiprogramming level (MPL) by statistical projection from past miss ratios and their associated MPL values. In cases where the statistical projection method fails, PMM falls back on a heuristic that chooses the MPL based on desirable resource utilization levels. The memory allocation component operates using one of two strategies — a Max strategy that assigns to each query either its maximum required memory or no memory at all, and a MinMax strategy that allows some low-priority queries to run with their minimum required memory while the high-priority ones get their maximum. The choice of memory allocation strategy is based on statistics about the workload characteristics that PMM gathers. Since both the MPL setting and memory allocation strategy choices have to be tailored to the characteristics of the workload, PMM constantly monitors the workload for changes that may necessitate adjustments to its decisions. The details of the algorithm are presented below. The key parameters of PMM, which will be explained as they appear in the following description, are summarized in Table 1.

3.1. Admission Control

The task of the admission control mechanism is to determine the MPL based on current operating conditions. In order to minimize the *miss ratio*, defined as the proportion of queries that fail to complete by their deadlines, the MPL has to be high enough so that the CPU and disk resources can be fully exploited. However, the MPL should not be so high as to cause the system to experience thrashing. The relationship between MPL and miss ratio thus follows the shape of a concave curve. PMM attempts to locate the optimal MPL, i.e., the MPL that leads to the lowest miss ratio on this curve, through a combination of *miss ratio projection* and a *resource utilization heuristic*, revising its MPL setting after every *SampleSize* queries are served by the system. The two components of the MPL determination method are presented below.

Parameter	Meaning	Default
SampleSize	Re-evaluation frequency (# of query completions)	30
$[Util_{Low}, Util_{High}]$	Range of "desirable" CPU/ disk utilization levels	[0.70, 0.85]
$Adapt_{ConfLevel}$	Conf. level of statistical tests for PMM adaptation	95%
$Change_{ConfLevel}$	Conf. level of statistical tests for workload changes	99%

Table 1: PMM Algorithm Parameters

3.1.1. Miss Ratio Projection

The miss ratio projection method approximates the relationship between MPL and miss ratio by a concave quadratic equation; this equation is used to set the system's target MPL. A quadratic equation is used here because it stabilizes faster than higher-order equations, while still capturing the general shape of the concave curve. After every *SampleSize* query completions, PMM measures the miss ratio, $miss_i$, that the current MPL, mpl_i, produces. Based on this pair of values, together with past miss ratios and their associated MPL settings, a new quadratic equation is calculated according to the least squares method [Drap81]. It is important to note that PMM does not actually have to keep track of individual miss ratio readings, but only the values of k, $\Sigma\, mpl_i$, $\Sigma\, mpl_i^2$, $\Sigma\, mpl_i^3$, Σmpl_i^4, $\Sigma\, miss_i$, $\Sigma\, mpl_i \times miss_i$, and $\Sigma\, mpl_i^2 \times miss_i$, where k is the number of times PMM is invoked. After approximating the equation, a new MPL value is chosen according to the type of curve obtained:

Type 1: *The curve has a bowl shape.* In this case, the curve has a minimum. Therefore, the target MPL is set to the minimum of the curve. (This is the expected case after the algorithm has been operating for a while.)

Type 2: *The curve is monotonic decreasing*, i.e. higher MPLs lead to lower miss ratios. This indicates that the optimal MPL is beyond the highest MPL tried so far. Since the curve may not be valid if extrapolated too far, the projection method selects an MPL that is one above this largest attempted MPL. Next, PMM applies the resource utilization heuristic (described below) to see if an even higher MPL may be warranted. If so, the MPL suggested by that heuristic is adopted; otherwise PMM sticks to the MPL that the miss ratio projection method picked.

Type 3: *The curve is monotonic increasing.* The MPL computation procedure for this case is just the opposite of the procedure for Type 2 curves. Here the projection method tentatively selects an MPL that is one unit below the smallest MPL that has been tried so far. Next, a second MPL is obtained using the resource utilization heuristic. The two MPLs are then compared, and the smaller of the two is adopted.

Type 4: *The curve has a hill shape.* Occasionally the fitted curve takes on this shape due to randomness in the observed miss ratios caused by inherent workload fluctuations. When this happens, the projection method fails and PMM resorts to the resource utilization heuristic.

An attractive feature of the miss ratio projection method is that the MPL values that it picks improve over time: Initially, the shape of the fitted curve is largely influenced by random workload fluctuations. As time progresses and more miss ratio readings are obtained, the fitted curve will gradually stabilize and its optimum will close in on the optimal MPL. At this point, the system can be expected to deliver good performance so long as there are no significant changes in the workload characteristics. (Workload changes will be addressed in Section 3.3).

3.1.2. Resource Utilization Heuristic

The resource utilization (RU) heuristic attempts to help the system achieve low query miss ratios by keeping the utilization of the most heavily loaded resource among the CPU and disks within some "desirable" range, $[Util_{Low}, Util_{High}]$, thus avoiding situations where the bottleneck resource is either under-utilized or near saturation. The heuristic extrapolates from the current MPL and utilization to predict a new MPL that is likely to bring the utilization into the middle of the $[Util_{Low}, Util_{High}]$ range by applying the following formula[1]:

$$MPL_{New} = \frac{Util_{Low} + Util_{High}}{2 \times Util_{Current}} \times MPL_{Current}$$

The linear dependency between MPL and utilization that this formula assumes is based on the observation that the utilization of a resource increases approximately linearly with the MPL until the resource is near saturation, at which point the utilization levels off. Since neither the RU heuristic nor the miss ratio projection method are likely to push the utilization way above $Util_{High}$ to saturation, the above formula should provide satisfactory MPL estimates most of the time. Even in regions where the linear dependency assumption does not hold, the RU heuristic is still useful in steering the MPL setting in the direction of the optimal MPL since utilization increases monotonically with MPL.

As described, one of the values that the RU heuristic uses to compute the new MPL is the utilization of the most heavily loaded resource at the current MPL. Due to random workload fluctuations, the utilization over the duration of the current batch of *SampleSize* queries may not be indicative of the resource's overall average utilization at that MPL. For this reason, the heuristic actually averages the utilization values that have been obtained so far instead of relying only on the most recent utilization reading. Conceptually, PMM computes the average utilization at the current MPL, denoted as $Util_{Current}$ in the formula above, by first obtaining a straight line from every pair $\langle util_i, mpl_i \rangle$ of observed utilization values and their associated MPLs by using the least squares method [Drap81], again applying the linearity assumption. The average utilization is then taken from the fitted line as the rate that corresponds to the current MPL. For the purposes of computing the straight line, PMM records the values of k, $\Sigma\, mpl_i$, $\Sigma\, mpl_i^2$, $\Sigma\, util_i$, and $\Sigma\, mpl_i \times util_i$, where k denotes the number of times PMM is invoked.

3.2. Memory Allocation

As described above, queries like hash joins and external sorts each have a maximum and a minimum memory requirement. Given its maximum required memory, such an operation can read its operand relation(s) and generate

[1] An alternative to using this formula would have been to simply choose the MPL value on the fitted line that corresponds to the desired utilization level. However, due to workload fluctuations, the fitted line may not reflect the true relationship between MPL and utilization very well. This is especially a problem at the start, where few statistics are available, and where, unfortunately, PMM has to rely on the RU heuristic because it does not yet have sufficient statistical data to apply the miss ratio projection method. We therefore ruled out this alternative.

results directly. Given only its minimum required memory, which is typically much lower than its maximum, the operation instead has to process its operand relation(s), write out intermediate results to temporary files, and then read these files back for further processing before the final results can be produced. The maximum memory requirement of an external sort is the size of its operand relation [Shap86], whereas it can run with as few as three memory pages by doing multiple merge passes. In the case of a hash join, the maximum memory requirement and the minimum memory demand for two-pass operation are $F||R||$ and $\sqrt{F||R||}$, respectively, where $||R||$ is the inner (building) relation size and F is a fudge factor that reflects the overhead of a hash table [Shap86].

When the total maximum memory requirement of the admitted queries exceeds the available memory, the memory allocation component is responsible for determining the amount of memory to allot to each query. As mentioned previously, the memory allocation decisions of PMM are based on the ED policy, so queries that are more urgent are always given buffers ahead of queries with looser deadlines. At any given time, PMM adopts one of two memory allocation strategies: the Max strategy or the MinMax policy. With the Max strategy, queries are either allocated enough memory to satisfy their maximum demands or else they are given no buffers at all. When operating in MinMax mode, however, PMM is able to admit more queries by meeting the maximum memory demands for only some of the more urgent queries, allowing the rest of the queries to execute with their minimum required memory. The reason for doing MinMax allocation, as opposed to simply dividing the available memory proportionally among the admitted queries, is that MinMax leads to more effective use of memory then proportional allocation (as was shown in [Corn89, Yu93]); this will be verified quantitatively in Section 5.1.

The MinMax allocation process is conceptually carried out in two passes. Starting from the highest-priority query, PMM first gives each query just enough memory for it to begin execution. If there are leftover buffers at the end of this pass, PMM makes another pass through the list of admitted queries, again beginning with the highest-priority query. In the second pass, the allocation of each query in turn is topped up to its maximum. The allocation process terminates when either all of the available memory has been allocated or all of the queries have received their maximum allocations. Consequently, at the end of this memory allocation process, the higher-priority queries will have their maximum allocations while the lower-priority queries just have their minimum. The only possible exception is the query that gets the last few memory pages in the second pass, which may receive an allocation somewhere in between its minimum and maximum demands. In a running system, of course, queries do not arrive all at once; rather, they come and go over time. Since ED assigns priorities to queries according to their urgency, the memory allocation of a query can therefore vary between maximum, minimum, or no allocation as higher-priority queries enter and leave the system, but over time it will settle on the maximum allocation as the query's deadline draws close. The initial variations are the reason why we require the dynamic query processing techniques described in Section 2.

The Max strategy, by insisting on the maximum memory allocation, eliminates the thrashing problem that can result when additional (lower-priority) queries are admitted at the expense of requiring some of the higher-priority queries to run with less than their maximum memory allocations. Consequently, PMM does not explicitly limit the MPL when it is in Max mode. Instead, PMM admits as many queries at their maximum allocations as memory permits. A possible pitfall of Max is that it may severely restrict the MPL if every query requires a substantial amount of memory in order to run at its maximum allocation. In contrast, MinMax assigns to some or all of the admitted queries as little as their minimum memory demand, thus enabling the system to achieve the target MPL that the admission control component sets Whether Max or MinMax performs better depends on the workload characteristics and the system configuration — Max is preferable if memory is abundant and the bottleneck resource type is CPU or disk, whereas MinMax is more suitable for memory-constrained situations.

The PMM algorithm uses a feedback mechanism to monitor the state of the system, and it revises its choice of allocation strategy as necessary. Initially, the Max mode is selected. After serving every *SampleSize* queries, PMM checks the system state and switches to MinMax if all of the following conditions are met: (1) one or more queries in this batch missed their deadlines; (2) the utilizations of the CPU and disks are below $Util_{Low}$, which indicates that none of these resources are likely to be a bottleneck; (3) there is a non-zero admission waiting time, suggesting that there is memory contention; and (4) on the average, the execution time of a query is shorter than its time constraint (the difference between its deadline and its arrival time) so that the longer execution times that will result from switching to MinMax are likely to be feasible. In checking for condition (3), PMM carries out a large-sample test [Devo91] for the mean waiting time at a confidence level of $Adapt_{ConfLevel}$. Condition (4) is tested in a similar fashion, except that here the test is on the difference between the execution time and time constraint. After switching to MinMax, PMM then monitors the target MPL. If it drops to or falls below the average MPL that was realized in Max mode, PMM reverts to the Max strategy. This entire process is repeated continuously.

3.3. Dealing with Workload Changes

PMM attempts to minimize query miss ratios by tailoring its MPL setting and memory allocation strategy to the system's workload and resource configuration. Consequently, it is necessary for PMM to discard the statistics that it has gathered and to re-adapt itself when the workload undergoes a significant change. In order to detect workload changes, PMM constantly monitors the following workload characteristics: (1) the average maximum memory demand of queries; (2) the average number of I/Os that each query issues to read its operand relation(s)

(the number of I/Os that are expended to write and read intermediate results depends on memory allocation decisions, and thus is not an inherent characteristic of the workload); and (3) the average normalized time constraint, defined as the ratio of the time constraint to the number of I/Os needed to read the operand relation(s). After every *SampleSize* query completions, PMM carries out a large-sample test with a confidence level of $Change_{ConfLevel}$ [Devo91] on each monitored workload characteristic to see if its present value differs significantly from its last observed value. If so, PMM concludes that a workload change has taken place. Since every workload change prompts PMM to restart itself, $Change_{ConfLevel}$ is set to a high value to reduce the chances of PMM wrongly reacting to inherent workload fluctuations.

3.4. An Example

Having presented the PMM algorithm in detail, we now finish by illustrating it with a simple example. Suppose that the first batch of *SampleSize* queries produces point *a* in Figure 1(a) under the Max strategy, and suppose that PMM concludes that Max is inappropriate and decides to switch to MinMax. At this point, the RU heuristic suggests a higher MPL, from which we derive the point *b* after the next batch of query completions. Once more, the RU heuristic leads PMM to raise its MPL setting, which results in point *c* after the third batch of queries. Having collected three observations, PMM can now apply the miss ratio projection method. The quadratic equation that is computed from the three points is shown by the Type 2 curve (see Section 3.1.1) in Figure 1(a). This curve causes PMM to experiment with an even higher MPL, the consequence of which is indicated by point *d* in Figure 1(b). Applying the projection method again, PMM now obtains a Type 1 curve. Since the optimum of the curve is likely to be near the optimal point, PMM adopts the MPL value associated with this optimum for its next MPL setting. As this process continues and more observations are gathered, the fitted curve will gradually stabilize and lead PMM to the best MPL for the given workload.

4. DATABASE SYSTEM SIMULATION MODEL

To aid in our ongoing research on real-time databases, we have constructed a simulation model of a centralized database system. The model, shown in Figure 2, has five components: a *Source* that generates the system's workload and collects statistics on completed queries; a *Query Manager* that models the execution details of queries, including hash joins and external sorts; a *Buffer Manager* that implements an LRU replacement policy and the PMM algorithm; and a *CPU Manager* and a *Disk Manager* that

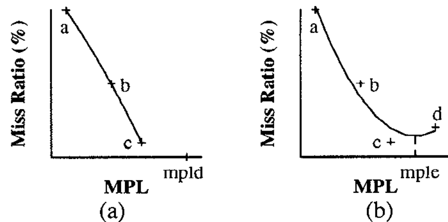

Figure 1: Admission Control Decision-Making

are responsible for managing the system's CPU and disks, respectively. The simulator is written in DeNet [Livn90].

4.1. Database and Workload Model

Table 2 summarizes the database and workload parameters that are relevant to this study. Our objective is to simulate a stream of external sorts and/or hash joins on different relations. To facilitate this, the database consists of *NumGroups* groups of relations. Each group *i* has $RelPerDisk_i$ clustered relations per disk. The size of the $RelPerDisk_i$ relations are chosen at equal intervals from $SizeRange_i$. For example, if $RelPerDisk_i$ = 5 and $SizeRange_i$ = [100, 200] pages, group *i* will have 5 relations with sizes equal to 100, 125, 150, 175, and 200 pages, respectively, on every disk. To minimize disk head movement, all relations assigned to the same disk are randomly placed on its middle cylinders; temporary files are allotted either the inner or the outer cylinders.

In this study, the workload comprises *NumClasses* classes of queries. Each class *j* has the following characteristics: It may be made up of external sorts, in which case $RelGroup_j$ specifies a group of database relations from which queries in class *j* draw their operand relations. Alternatively, the class may consist of hash joins. In the second case, every query in the class randomly chooses two relations by taking one relation from each of the two relation groups listed in $RelGroup_j$. The smaller of the two chosen relations is the inner relation, **R**, of the join, while its outer relation, **S**, is the larger relation. The type of queries that form the class (sort or hash join) is indicated by the parameter $QueryType_j$. Query submissions from the class follow a Poisson process with a mean arrival rate of λ_j. The *Source* module assigns a deadline to each new query *Q* from class *j* in the following manner:

$$Deadline_Q = StandAlone_Q \times SlackRatio_Q + Arrival_Q$$

where $Deadline_Q$, $StandAlone_Q$, $SlackRatio_Q$ and $Arrival_Q$ are the deadline, stand-alone execution time, slack ratio

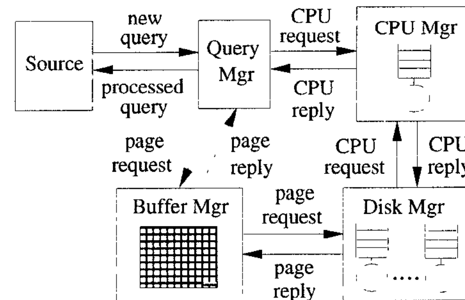

Figure 2: Database System Model

Database	Meaning
NumGroups	Number of relation groups in the database
$RelPerDisk_i$	Number of relations per disk for group *i*
$SizeRange_i$	Range of relation sizes for group *i*
TupleSize	Tuple size of relations in bytes
Workload	**Meaning**
NumClasses	Number of classes in the workload
$QueryType_j$	Type of class *j* queries (hash join or ext. sort)
$RelGroup_j$	Operand relation group(s) for class *j* queries
λ_j	Arrival rate of class *j* queries
$SRInterval_j$	Range of slack ratios for class *j* queries
F	Fudge factor for hash joins

Table 2: Database and Workload Model Parameters

and arrival time of query Q, respectively. The stand-alone execution time of a query is the time it would take to execute alone in the system with its maximum memory allocation, i.e., without experiencing any contention from other queries. The slack ratio, $SlackRatio_Q$, varies uniformly in the range specified by $SRInterval_j$, and it controls the tightness of the query's assigned deadline.

4.2. Physical Resource Model

The parameters that specify the physical resources of our model, which consist of CPU, disks and memory, are listed together with their default values in Table 3. The CPU, which has a MIPS rating of *CPUSpeed*, is scheduled by the Earliest Deadline (ED) discipline. Table 4 gives the costs of the various CPU operations involved in the execution of hash joins and external sorts.

Turning to the disk model parameters in Table 3, *NumDisks* specifies the number of disks attached to the system. Every disk manages its own queue by the ED policy; any disk requests that ED assigns the same priority to are serviced according to the elevator algorithm. Each disk has a 256-KByte cache for use in prefetching pages. To keep the per-page I/O cost low, all queries capitalize on this facility, fetching *BlockSize* pages on each sequential I/O that incurs a disk cache miss (except during the merge phase of an external sort). Moreover, whenever queries have enough buffers, they spool their outputs so that pages are flushed to disk in blocks. The access characteristics of the disks are also given in Table 3. The total time required to complete a disk access is:

$$\text{DiskAccess} = \text{Seek} + \text{RotateDelay} + \text{Transfer}$$

As in [Bitt88], the time required to seek across n tracks is:

$$\text{Seek Time}\ (n) = \text{SeekFactor} \times \sqrt{n}$$

Finally, the system has a total buffer pool size of **M** pages. A reservation mechanism allows query operators, including sorts and joins, to reserve buffers for use as workspaces. These reserved buffers are managed by the operators themselves, while page replacement for non-reserved buffers is handled according to the LRU policy.

Parameter	Meaning	Default
CPUSpeed	MIPS rating of CPU	40 MIPS
NumDisks	Number of disks	10
SeekFactor	Seek factor of disk	0.000617
RotationTime	Time for one disk rotation	16.7 msec
NumCylinders	Number of cylinders per disk	1500
CylinderSize	Number of pages per cylinder	90 pages
PageSize	Number of bytes per page	8 KBytes
BlockSize	Number of pages requested on each sequential I/O	6
M	Total number of buffer pages	2560 pages

Table 3: Physical Resource Model Parameters

Operation	# Instructions
Common Operations —	
Start an I/O operation	1000
Initiate a sort or join	40,000
Terminate a sort or join	10,000
Hash Joins —	
Hash tuple and insert into hash table	100
Hash tuple and probe hash table	200
Hash tuple and copy to output buffer	100
External Sorts —	
Copy a tuple to output buffer	64
Compare two keys	50

Table 4: Number of CPU Instructions Per Operation

5. EXPERIMENTS AND RESULTS

In this section, our database system simulator will be used to evaluate the performance of the Priority Memory Management (PMM) algorithm. For comparison purposes, we shall also examine three static memory allocation algorithms: Max, MinMax-N, and Proportional-N. The Max algorithm always employs the Max strategy in its memory allocation decisions. MinMax-N admits the N highest-priority queries, dividing the available memory among these N queries according to the MinMax policy. A special case of MinMax-N is MinMax-∞, which admits as many queries as the available memory allows by not explicitly limiting the MPL. We shall refer to MinMax-∞ simply as MinMax, as it will be frequently used to compare against PMM. Note that PMM is an adaptive algorithm that dynamically chooses between the Max algorithm and the MinMax-N algorithm, where N is the target MPL setting. The final algorithm to which PMM will be compared, Proportional-N, behaves like MinMax-N, except that Proportional-N gives the N admitted queries the same percentage of their maximum buffer requirements subject to the condition that the memory allocation of an admitted query must at least equal its minimum requirement. As in the case of MinMax, we shall simply refer to Proportional-∞ as Proportional. For ease of reference, the various algorithms are listed in Table 5.

We will begin our evaluation of PMM with a baseline experiment, with further experiments being carried out by varying a few parameters each time. The performance metric of interest here is the average query miss ratio, which is the percentage of queries that fail to complete by their deadlines. Unless stated otherwise, each experiment was run for 10 simulated hours, allowing a minimum of 2000 query completions. We also verified that the size of the 90% confidence intervals for miss ratios (computed using the batch means approach [Sarg76]) was within a few percent of the mean in almost all cases.

5.1. Baseline Experiment

In the first experiment, we simulate an environment where, except for occasional overloads, there are abundant CPU and disk capacities for the given workload; thus, memory is the bottleneck resource. This is achieved by letting *CPUSpeed* and *NumDisks* be 40 MIPS and 10, respectively, and by setting **M** to 2560 pages (20 MBytes). The workload consists of one class of hash join queries. Each join has two operand relations, **R** and **S**, where $\|R\|$ varies uniformly between 600 and 1800 pages and $\|S\|$ is selected from the range [3000, 9000] pages. Moreover, the slack ratio interval is set to [2.5, 7.5]. The database and workload parameters are summarized in Table 6, while the rest of the resource parameters are kept at their default settings of Table 3.

Indicator	Algorithm
Max	Max algorithm
MinMax-N	MinMax with an MPL limit of N
MinMax	MinMax with no MPL limit
Proportional-N	Proportional with an MPL limit of N
Proportional	Proportional with no MPL limit

Table 5: Algorithms for Comparison with PMM

Database	Value	Workload	Value
NumGroups	2	*NumClasses*	1
$RelPerDisk_1$	3	$QueryType_1$	Hash join
$SizeRange_1$	[600, 1800]	$RelGroup_1$	{1, 2}
$RelPerDisk_2$	3	λ_1	varied
$SizeRange_2$	[3000, 9000]	$SRInterval_1$	[2.5, 7.5]
TupleSize	256 bytes	*F*	1.1

Table 6: Database and Workload Settings (Baseline Experiment)

Figure 3 plots the miss ratios for Max, MinMax, Proportional, and PMM as a function of the arrival rate. The figure shows that MinMax consistently delivers the lowest miss ratio for this experiment, followed very closely by PMM. Proportional performs satisfactorily initially, achieving a near 0% miss ratio at $\lambda = 0.04$ queries/second. As the arrival rate increases, however, the performance of Proportional deteriorates rapidly until, at $\lambda = 0.08$ queries/second, Proportional produces a hefty 25% miss ratio, which is almost double that of MinMax and PMM. The worst algorithm is Max, which matches the performance of Proportional only under lighter load conditions. As the workload mounts, Max degenerates even faster than Proportional, missing four times as many deadlines as MinMax and PMM. These observations clearly show that the choice of memory allocation algorithm can have a very significant impact on the system miss ratio. To understand the behaviors of the four algorithms, we shall analyze each in turn with the aid of Figures 4 and 5, which give the disk utilizations and average observed MPLs (as opposed to the target MPL set by PMM, which serves to limit the maximum MPL in the system) respectively, and Table 7, which lists the admission waiting time, execution time and total response time for the various algorithms.

Let us first examine the Max algorithm. This algorithm admits queries only if they can be allotted enough buffers to satisfy their maximum requirements. For the workload used in this experiment, Max allows less than 2 queries to be admitted at the same time (see Figure 5) since each query requires an average of 1321 buffers ($F \times 1200$ pages for **R** plus one I/O buffer). This makes memory the bottleneck for Max, as evidenced by the high admission waiting times recorded in Table 7. The tight MPL limit imposed by Max prevents the RTDBS from exploiting its disk and CPU resources to cope with the heavier load as the arrival rate increases from 0.04 to 0.08 queries/second, which explains why, unlike the other three algorithms, Max's disk utilization barely rises. This ineffective resource usage leads to the observed sharp growth in the miss ratio of Max.

In contrast to Max, MinMax attempts to reduce query miss ratios by increasing the MPL. This is achieved at the expense of running queries with memory allocations that are less than their maximum, which increases the demands on the CPU and the disks. By giving queries their minimum required memory, MinMax could admit up to an average of 69 queries at the same time (on the average, the minimum memory requirement per query is $\sqrt{F\|R\|}$ pages + 1 I/O buffer = 37 pages), thus allowing much higher average MPLs as Figure 5 shows. Moreover, the increased CPU and disk demands that result have little harmful effect here, as the disk utilization barely exceeds 45% even at an arrival rate of 0.08 queries/second, indicating that there are abundant CPU and disk capacities to service all the admitted queries. The overall result is that MinMax uses the system's resources much more effectively than Max. As shown in Table 7, the higher execution times that MinMax produces are more than compensated for by the large reduction in admission waiting times, thus resulting in total response times that are significantly lower than the response times of Max. This accounts for MinMax's superior miss ratios in Figure 3.

Like MinMax, Proportional attempts to reduce query response times by not insisting on maximum memory allocation as an admission criterion. This is why Proportional also produces higher MPLs than Max. The difference between Proportional and MinMax is that Proportional

Arrival Rate	0.04	0.05	0.06	0.07	0.08
Max					
Waiting	12.4	36.4	81.4	107.3	117.3
Execution	39.5	35.4	32.9	25.9	22.4
Total	51.9	71.8	114.3	133.2	139.7
MinMax					
Waiting	0.0	0.0	0.0	0.0	0.0
Execution	40.9	45.5	53.1	68.3	92.1
Total	40.9	45.5	53.1	68.3	92.1
Proportional					
Waiting	0.0	0.0	0.0	0.0	0.0
Execution	52.9	61.2	75.8	92.4	110.8
Total	52.9	61.2	75.8	92.4	110.8
PMM					
Waiting	3.0	3.3	3.7	3.9	4.0
Execution	40.2	45.1	52.5	66.3	89.4
Total	43.2	48.4	56.2	70.2	93.4

Table 7: Average Timings (seconds) for Baseline Experiment

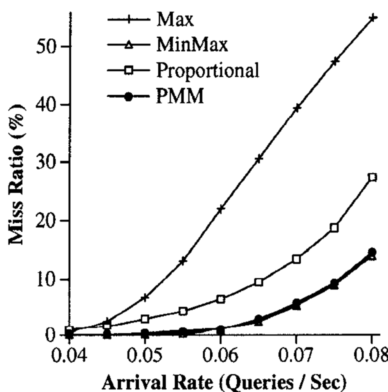

Figure 3: Miss Ratio (Baseline)

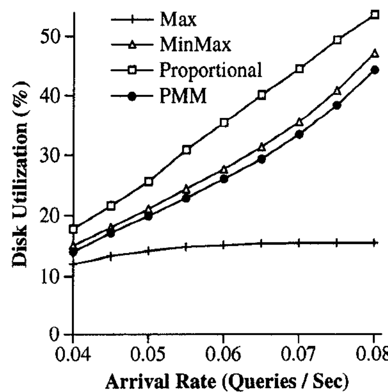

Figure 4: Disk Utilization (Baseline)

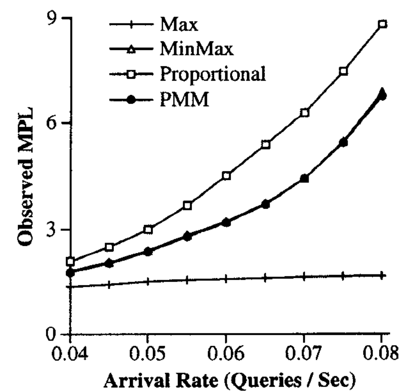

Figure 5: MPL (Baseline)

divides up memory among the admitted queries in proportion to their demands, rather than running low-priority queries with minimum allocations while giving high-priority queries their maximum required memory (as in MinMax). Unfortunately, the faster execution times that the low-priority queries enjoy from receiving more than their minimum required memory are overwhelmed by the execution time penalty that the high-priority queries pay as a result of being forced to run with less-than-maximum memory allocations. The average execution time that Proportional produces is therefore higher than that of MinMax. The longer query execution times also cause an increase in the number of queries that are running concurrently, as Figure 5 shows, which in turn reduces the memory allocation of each query. This increases the queries' reliance on the CPU and disks, resulting in further increases in the queries' execution times. Consequently, Proportional utilizes memory much less effectively than MinMax. As mentioned earlier, similar observations about the inferiority of Proportional-style policies were made in [Corn89, Yu93] in a non-real-time context.

We now turn our attention to the PMM algorithm. In order to understand how PMM adapts itself to the workload, we examine Figure 6, which traces the target MPL settings of PMM over the initial 10 hours of operation at an arrival rate of 0.075 queries/second. PMM starts with Max, but it quickly detects that this allocation strategy is not satisfactory because it leads to a very limited MPL while leaving the CPU and disks grossly underutilized. This causes PMM to switch to MinMax mode to make a higher MPL possible. The target MPL is first set to 25, following the suggestion of the Resource Utilization heuristic. Once PMM has gathered three miss ratio observations, it invokes the miss ratio projection method, which quickly steers the target MPL to the vicinity of 10 where it stabilizes. This MPL is sufficiently loose to admit all of the queries into the system most of the time, as the low 4-second admission waiting time in Table 7 suggests. Indeed, Figure 5 shows that PMM consistently achieves high MPL settings, thus enabling it to behave like the MinMax algorithm. This is why PMM manages to closely match the performance of MinMax, which offers the best miss ratios for this experiment.

Having studied the performance trade-offs of the memory allocation algorithms, we now briefly examine the demand that these algorithms place on the system's underlying memory-adaptive query processing primitives. Figure 7 shows, as a function of the arrival rate, the average number of times that a query's memory allocation changes under each algorithm. The Max algorithm either executes queries with their maximum required memory or it suspends them. In contrast, the other three algorithms do expose executing queries to changes in their memory allocations. Under MinMax (and hence PMM, since it mimics MinMax in this experiment), the allocation of a query may vary between its minimum and maximum memory requirements initially, gradually stabilizing at the maximum only as its deadline draws near. The algorithm that generates the most memory fluctuations is Proportional, which always distributes memory proportionally among all admitted queries, therefore subjecting them to memory changes throughout their lifetimes.

To summarize the results of this experiment, we can derive the following conclusions about situations where memory is the bottleneck resource of an RTDBS: First, insisting on maximum memory allocation as an admission criterion is undesirable. Instead, an RTDBS needs to be willing to run queries at memory allocations that are below their maximum requirements so that enough queries can be admitted to take advantage of the RTDBS's disk and CPU resources. This is facilitated by memory-adaptive query processing techniques (such as those of [Pang93a, Pang93b]) that permit queries to execute efficiently in the face of memory fluctuations. Among the algorithms that do not insist on maximum memory allocations, Proportional allocation leads to very large miss ratios and should be avoided. This is why PMM employs MinMax allocation when it detects that running queries with sub-maximal memory allocations is beneficial. Finally, PMM seems to be capable of finding the right MPL setting and memory allocation strategy within a few iterations, achieving low query miss ratios by balancing the load on the system's various resources.

5.2. Moderate Disk Contention

In the next experiment, we investigate how PMM performs when disk contention becomes more of a consideration in memory allocation decisions, though memory remains the bottleneck resource. The number of disks is reduced here to 6, while the rest of the parameters remain at their settings from the baseline experiment. We will exclude the Proportional algorithm since it is inferior to MinMax. The performance statistics for the remaining three algorithms, Max, MinMax-N and PMM, are given in Figures 8, 9 and 10, which plot as a function of the arrival rate their miss ratios, disk utilizations, and observed MPLs, respectively. These figures show that the behavior of Max is essentially the same as in the baseline experiment. We shall therefore not discuss Max here, instead focusing on MinMax and PMM, both of whose behaviors differ significantly from those observed previously.

We first analyze the performance of the MinMax algorithm. Figure 8 shows that MinMax no longer provides the best performance. In fact, MinMax now misses many more deadlines than PMM under heavy loads. The performance deterioration of MinMax here is due to its unrestrained admission policy. In this experiment, where disk contention is not negligible, the system does not always have enough disk capacity for all of the queries that MinMax admits. This is evidenced by the higher average disk utilizations in Figure 9, which exceed 70% under heavy loads. As a result, some of the low-priority queries remain essentially inactive even after being allotted memory because they do not get the opportunity to access the disks under the priority scheduling policy. This unproductive use of memory unnecessarily forces higher-priority queries to run below their maximum memory allocations and increases their dependence on the CPU and disks, resulting in the observed rise in MinMax's miss ratios.

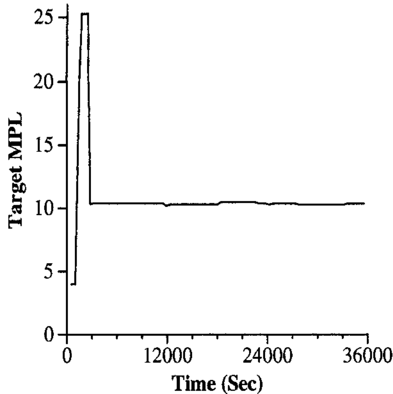

Figure 6: PMM MPL (λ = 0.075)

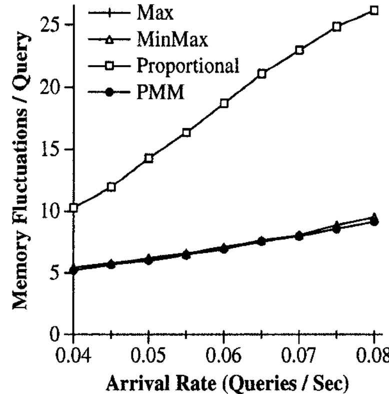

Figure 7: Memory Fluctuations (Baseline)

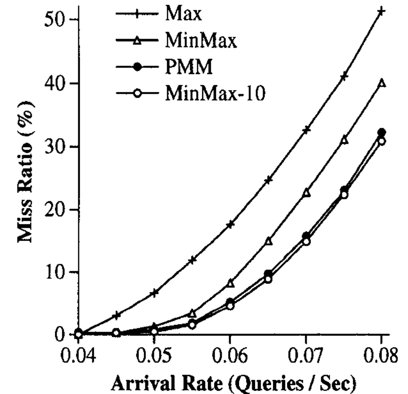

Figure 8: Miss Ratio (Disk Contention)

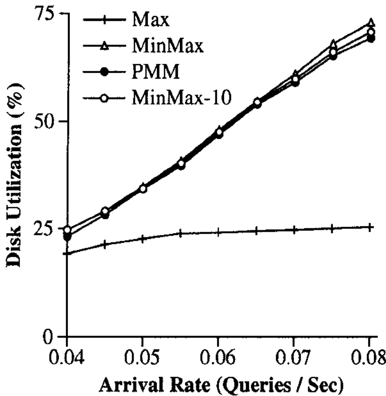

Figure 9: Disk Util. (Disk Contention)

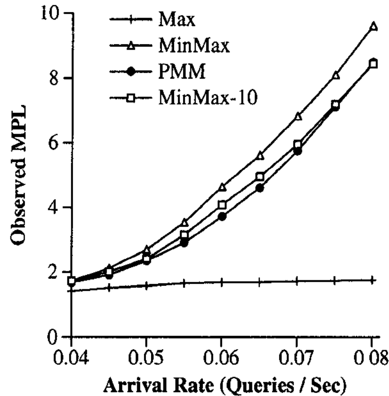

Figure 10: MPL (Disk Contention)

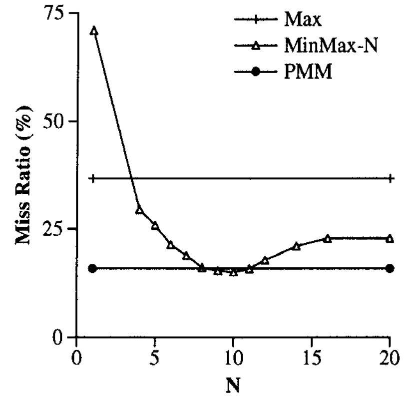

Figure 11: MinMax-N, λ = 0.07 (Disk Contention)

Since MinMax performs unsatisfactorily here, we must examine other MinMax-N variants in order to explain PMM's performance. Figure 11 plots the miss ratios produced by MinMax-N as a function of N for an arrival rate of 0.07 queries/second. The MinMax-N variants that are included cover the entire spectrum of trade-offs. At one end, the MinMax-N algorithms with low N values are similar to Max, as every admitted query is able to run with maximum memory allocation due to the low MPL settings. At the other end of the spectrum is MinMax-20, which essentially performs like MinMax (not shown)[2]. Figure 11 shows that the best performance for this workload is achieved by MinMax-10, which utilizes the CPU and disks much more effectively than Max by admitting more queries into the system — but not so many queries that thrashing occurs, as is the problem with MinMax. We also conducted a series of experiments like Figure 11 at other arrival rates, and the results of those experiments unanimously confirmed that MinMax-10 indeed delivers the best performance for the present workload.

Having identified MinMax-10 as the best MinMax-N algorithm for this experiment, we now proceed to evaluate PMM against MinMax-10. The curves in Figure 10 show that the observed average MPLs for PMM remain consistently close to those of MinMax-10. This indicates that PMM succeeds at bringing its MPL setting to the proximity of the best MPL value, which explains why PMM outperforms both Max and MinMax. In fact, Figure 8 shows that PMM manages to meet almost as many deadlines as MinMax-10 over the entire range of arrival rates that we investigate, delivering miss ratios that are worse than those of MinMax-10 by at most 2%.

The results of this experiment show that, while Max leads to under-utilization of the CPU and disks in memory-constrained situations, MinMax can produce thrashing when disk contention is not negligible. Therefore, some trade-off between Max and MinMax, i.e., a MinMax-N algorithm, is needed. Since the best MinMax-N algorithm depends on the system configuration and workload characteristics, which are usually not known in advance, the right MinMax-N algorithm to employ has to be dynamically selected. PMM demonstrated its ability here to quickly find the appropriate MinMax-N algorithm by steering itself to the best MPL setting.

5.3. Workload Changes

The first two experiments lead us to the conclusion that PMM performs well for relatively stable real-time workloads. The objective of this experiment is to find out how quickly PMM adapts to workload changes. This is achieved by subjecting the various memory allocation algorithms to a workload that alternates between two classes of hash joins, Small and Medium, every 2 to 5 simulated hours. For the Small class, $||R||$ ranges between

[2] Theoretically, MinMax allows up to an average MPL of 69 for this workload. In practice, the chances of having more than 20 queries in the system at the same time here is so rare that, for all practical purposes, MinMax-20 is the same as MinMax.

Database	Value	Workload	Value
NumGroups	4	*NumClasses*	2
$RelPerDisk_1$	3	$QueryType_{Medium}$	Hash join
$SizeRange_1$	[600, 1800]	$RelGroup_{Medium}$	{1, 2}
$RelPerDisk_2$	3	λ_{Medium}	0.07
$SizeRange_2$	[3000, 9000]	$SRInterval_{Medium}$	[2.5, 7.5]
$RelPerDisk_3$	3	$QueryType_{Small}$	Hash join
$SizeRange_3$	[50, 150]	$RelGroup_{Small}$	{3, 4}
$RelPerDisk_4$	3	λ_{Small}	2.8
$SizeRange_4$	[250, 750]	$SRInterval_{Small}$	[2.5, 7.5]

Table 8: Database and Workload Settings (Workload Changes)

50 and 150 pages, while ||S|| ranges from 250 to 750 pages. The characteristics of the Medium class are the same as those of the baseline workload. These two classes pose different demands on the system's resources. On one hand, it takes an average of only 111 memory pages to satisfy the maximum demand of each hash join from the Small class. Thus the disks, rather than the memory, are the bottleneck, and the Max algorithm is therefore appropriate for this class. On the other hand, the system is memory-constrained with the Medium class, making a MinMax-N algorithm more desirable, as we saw previously. In order to highlight the performance trade-offs between the various algorithms, the arrival rates of the two classes are chosen so that the RTDBS is forced to operate under relatively heavy load conditions. The database and workload parameters are listed in Table 8. For this experiment, the number of disks is again set to 6, with the rest of the resource parameters set to the values listed in Table 3.

Figures 12, 13, and 14 display the miss ratios of the three algorithms as a function of time, while Figure 15 traces the observed MPL under PMM. Figures 12 to 14 also give the average miss ratio over each interval along the top of each figure. Comparing the two static algorithms, we notice that MinMax's unrestrained admission policy again causes it to perform poorly: Whereas Max produces average miss ratios of 16% and 33% for the Small and Medium classes[3], respectively, MinMax produces average miss ratios of 37% and 23% for the two classes. In contrast to MinMax, PMM is able to capitalize on the system's disk and CPU resources without suffering from thrashing. By dynamically selecting its MPL setting and memory allocation strategy, PMM outperforms both Max and MinMax for the Medium class, missing only 15% of its queries on the average. Moreover, PMM successfully detects workload changes, switching back to Max mode for the Small class, so its average miss ratio for Small queries is just as low as that of the Max algorithm. Similar experiments under lighter loads revealed essentially the same trade-offs between the three algorithms; while the magnitudes of the differences were smaller there, the relative performance of the algorithms was the same as that seen here. We therefore conclude that PMM not only performs well under stable workloads, but is also capable of adapting to workload changes.

5.4. Desirable Resource Utilization Levels

One of the input parameters of PMM is the range of desirable resource utilizations, $[Util_{Low}, Util_{High}]$. Up to this point, all of our experiments have used the range [0.70, 0.85] for this parameter. The choice of 0.85 for $Util_{High}$ is reasonable because, with resources being more than 85% utilized, the system most probably does not have enough capacity to service all of the admitted queries, so thrashing is likely to occur. The appropriate setting for $Util_{Low}$ is not as obvious, however. To study the sensitivity of PMM to the $Util_{Low}$ setting, we carried out an experiment where $Util_{Low}$ was varied from 0.50 to 0.80. The results showed that PMM delivers approximately the

[3] The average miss ratio of the Medium class is derived by averaging the miss ratios over the three time intervals where the workload is made up of Medium queries.

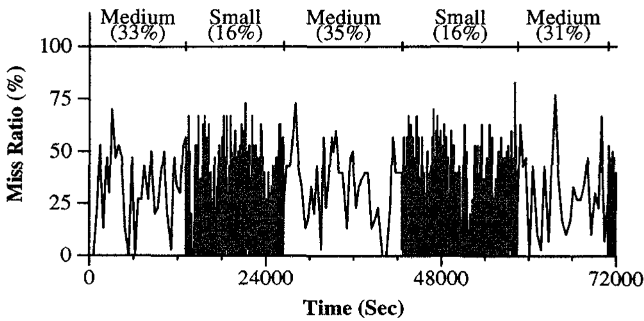

Figure 12: Max Miss Ratio (Workload Changes)

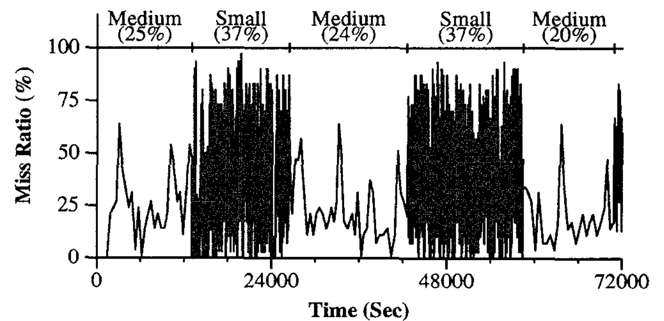

Figure 13: MinMax Miss Ratio (Workload Changes)

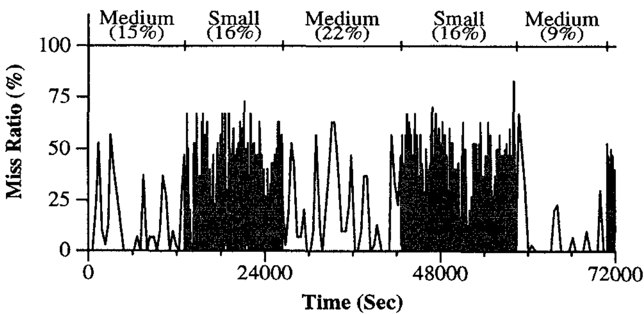

Figure 14: PMM Miss Ratio (Workload Changes)

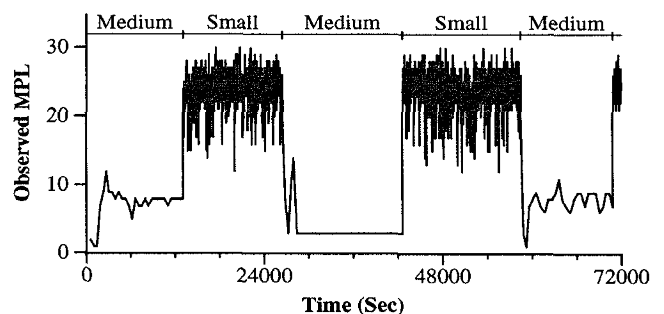

Figure 15: PMM MPL (Workload Changes)

same performance for the different $Util_{Low}$ values. This is not surprising, as PMM relies on the desirable resource utilization levels to set its MPL only during the initial period after startup. Since the precise value of $Util_{Low}$ does not matter, the default setting of 0.70 suffices.

5.5. Other Query Types

While we have demonstrated the capability of PMM for handling workloads that consist of hash joins, the PMM algorithm is designed to be a general memory management algorithm for RTDBSs; it is not limited to handling only hash joins. To verify that PMM is capable of handling other types of queries, we repeat the baseline experiment using external sorts. Each query in this new workload sorts a single relation **R**, where $||R||$ ranges from 600 to 1800 pages. All of the other workload and resource parameters (except arrival rates) remain as they were in the baseline experiment. Here we include the Proportional algorithm once again for completeness of our evaluation.

The miss ratios of Max, MinMax, Proportional, and PMM for this workload are shown in Figure 16. Comparing this figure with Figure 3, we notice that Max performs much worse here. This is because the load that they place on the disks and CPU is lighter here, while the memory demands of the queries are about the same as before; on the average, each external sort only has to read in a 1200-page relation, whereas the average hash join in the baseline experiment had to deal with a 1200-page inner relation plus a 6000-page outer relation. Consequently, memory is a much more critical resource here, thus resulting in a situation that is even more favorable to the liberal admission policies employed by the other algorithms. Again, we see that PMM is able to select the appropriate MPL setting and allocation strategy, achieving miss ratios that are just about as low as those obtained by MinMax.

5.6. Multiclass Workload

Our last experiment is designed to study how PMM performs when presented with a multiclass workload. We again simulate a workload that consists of two classes of hash joins, Small and Medium. The characteristics of the two classes are as listed in Table 8. However, instead of alternating between the two classes as in the "Workload Changes" experiment, here we activate both classes *together*. We fix the arrival rate of the Medium class at 0.065 queries/second and vary the arrival rate of the Small class. With the exception of the number of disks, which is raised to 12 to accommodate the heavier load here, the resource parameters remain as in the baseline experiment.

Figure 17 shows the overall *system* miss ratios produced by Max, MinMax, and PMM. Interestingly, here the system miss ratio curve of PMM resembles that of MinMax initially, but gradually switches to follow that of Max as λ_{Small} increases. This behavior arises because PMM chooses its MPL and memory allocation strategy according to the average characteristics of the workload, which naturally affords the class that has a higher arrival rate a greater influence on its choices. Consequently, PMM adopts the MinMax strategy, which is more suitable for Medium queries, only when λ_{Small} is low. As λ_{Small} rises, PMM allows the increasing influence of Small queries to sway it to Max mode. While operating in this mode is very effective in minimizing the system miss ratio, as Figure 17 shows, it severely limits the MPL of the Medium class and causes a disproportionally large number of Medium queries to miss their deadlines. This bias is clearly evident in Figure 18. Since such biased behavior may not be acceptable for certain applications, we are now working on augmenting PMM with a mechanism to allow an RTDBS system administrator to specify the desired relative *class* miss ratios to support applications that require "fairer" real-time query services.

5.7. Scalability of Results

In order to limit simulation costs, we intentionally chose to use small relation and memory sizes in our experiments. This raises questions about the scalability of our results to larger systems: How would larger memory and relation sizes affect the performance of the various algorithms? Would PMM still be able to choose appropriate MPL settings and memory allocation strategies quickly? To explore these issues, let us consider a scenario with the memory and relation sizes of Experiment 2 (the moderate disk contention case) scaled up by a factor of 10, and with the arrival rates reduced by the same factor in order to maintain the resource utilizations at their previous levels.

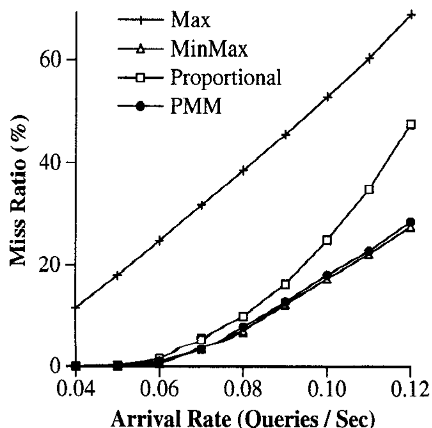

Figure 16: Miss Ratio (External Sort)

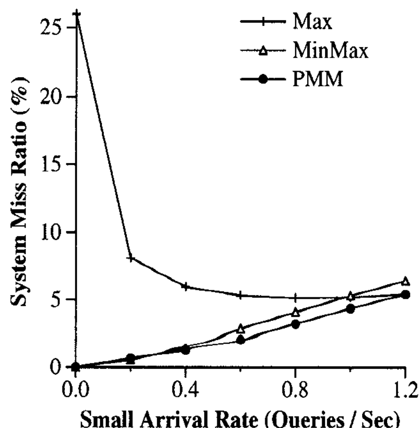

Figure 17: System Miss Ratio (Multiclass)

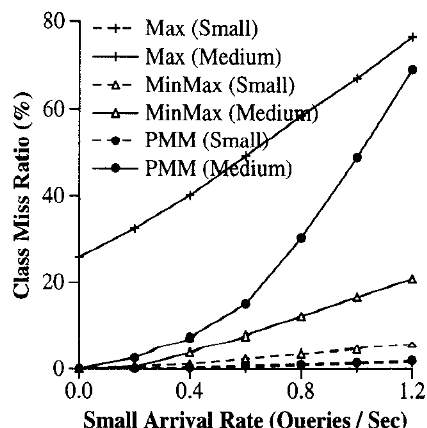

Figure 18: Class Miss Ratio (Multiclass)

In the case of Max, these changes should have no impact on the miss ratios since the maximum allocation of each query, $F\|R\|$, is unchanged relative to the memory size. In contrast, the MinMax algorithm would be affected by the larger sizes. This is because the average query's minimum required memory is only $\sqrt{10}$ times larger than before, while the maximum required memory and the system memory have been increased by a factor of 10. Admitting extra queries with their minimum allocations would thus have a lesser impact on the memory allocations of high-priority queries. Consequently, the detrimental effect of MinMax would be reduced considerably, leading MinMax to deliver miss ratios that are closer to those of the best MinMax-N algorithm. However, as we increase the arrival rate, the disadvantage of MinMax will still eventually overwhelm its benefits. There is therefore still a need for a mechanism to regulate query admissions.

Turning our attention to PMM, we first observe that PMM will still decide against using Max, as the behavior of the Max strategy is not affected by the larger sizes. Once in MinMax mode, PMM will require roughly the same number of query completions as before to find the right MPL setting. Therefore, the qualitative behavior of PMM should remain the same as in Experiment 2. To verify this, we carried out two different sets of experiments — a set of medium-scale experiments, reported in this paper, and a set of small-scale experiments that involved database and memory sizes that were ten times smaller. The two sets of experiments produced essentially the same qualitative algorithm behavior. We therefore expect our results to scale up to even larger memory and relation sizes; PMM should be just as effective for larger systems as it was for the workloads and configurations that we have experimented with here.

6. CONCLUSION

In this paper, we have focused on the problem of scheduling queries in firm real-time database systems (RTDBS). As a solution to this problem, we have proposed a *Priority Memory Management* (PMM) algorithm that aims to minimize the number of missed deadlines by adapting both the multiprogramming level (MPL) and the memory allocation strategy of an RTDBS according to feedback on system behavior. This eliminates the need for any advance knowledge of workload characteristics or query execution times, which is usually not available in a database system. Instead, the setting of the MPL is determined primarily by a statistical projection method, called miss ratio projection, which is supplemented by a resource utilization heuristic when the statistical method fails. PMM incorporates two memory allocation strategies — a Max strategy under which each query receives either its maximum required memory or no memory at all, and a MinMax strategy that allows some queries to run with their minimum required memory while others get their maximum. Both strategies employ the Earliest Deadline (ED) policy so that queries whose deadlines are more imminent are given memory ahead of queries that are less urgent. The choice of memory allocation strategy is based on statistics about the workload characteristics that PMM gathers; in order to ensure that its MPL setting and memory allocation strategy choices remain appropriate, PMM constantly monitors the workload for changes that may necessitate adjustments to those decisions.

Using a detailed RTDBS simulation model, we studied the performance of PMM under workloads that comprised both hash joins and external sorts. For comparison purposes, we also examined two static algorithms based purely on the Max and MinMax allocation strategies. Our experiments revealed that while the static algorithms perform satisfactorily under very light loads, neither algorithm is adequate in overload situations. In contrast, PMM is able to dynamically reach the right compromise between Max and MinMax, consistently delivering low miss ratios. Moreover, PMM achieves this quickly enough so that it works well even for fluctuating workloads. While we only experimented with queries that perform either external sorting or hash join operations, PMM is designed to schedule general query workloads effectively by balancing their demands on the system's memory, CPU, and disks. In particular, PMM can be extended to handle complex database queries that use external sorting and hash joins as building blocks, such as queries with aggregates, group-by clauses, and/or order-by clauses. Therefore, we conclude that the admission control and memory allocation mechanisms of PMM should be very useful for RTDBS query scheduling.

REFERENCES

[Abbo88] R. Abbott, H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation", *Proc. 1988 VLDB Conf*

[Abbo89] R. Abbott, H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data", *Proc. 1989 VLDB Conf.*

[Abbo90] R. Abbott, H. Garcia-Molina, "Scheduling I/O Requests with Deadlines. A Performance Evaluation", *Proc. 1990 RTSS Symp.*

[Bitt88] D. Bitton, J. Gray, "Disk Shadowing", *Proc. 1989 VLDB Conf.*

[Brow93] K. Brown, M. Carey, M. Livny, "Managing Memory to Meet Multiclass Workload Response Time Goals", *Proc. 1993 VLDB Conf.*

[Care89] M.J. Carey, R. Jauhari, M. Livny, "Priority in DBMS Resource Scheduling", *Proc. 1989 VLDB Conf*

[Chen91] S. Chen, J.A. Stankovic, J.F. Kurose, D. Towsley, "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems", *Real-Time Systems Journal 3(3)*, 1991

[Corn89] D. Cornell, P. Yu, "Integration of Buffer Management and Query Optimization in a Relational Database Environment", *Proc. 1989 VLDB Conf*

[Devo91] J.L. Devore, *Probability and Statistics for Engineering and the Sciences*, Brooks/Cole Pub. Co., 1991, pp. 283-301, 326-335.

[Drap81] N.R Draper, H. Smith, *Applied Regression Analysis*, John Wiley & Sons, Inc., 1981, pp. 70-136.

[Hari90] J R. Haritsa, M.J. Carey, M Livny, "On Being Optimistic about Real-Time Constraints", *Proc. 1990 PODS Symp.*

[Huan89] J. Huang, J.A. Stankovic, D. Towsley, K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing", *Proc. 1989 RTSS Symp.*

[Kim91] W. Kim, J. Srivastava, "Enhancing Real-Time DBMS Performance with Multiversion Data and Priority Based Disk Scheduling", *Proc. 1991 RTSS Symp.*

[Liu73] C. Liu, J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *ACM Journal*, January 1973.

[Livn90] M. Livny, "DeNet User's Guide, Version 1.5", *Computer Sciences Department, University of Wisconsin, Madison*, 1990

[Pang93a] H. Pang, M.J. Carey, M. Livny, "Partially Preemptible Hash Joins", *Proc. 1993 SIGMOD Conf.*

[Pang93b] H. Pang, M.J. Carey, M. Livny, "Memory-Adaptive External Sorting", *Proc. 1993 VLDB Conf.*

[Sarg76] R. Sargent, "Statistical Analysis of Simulation Output Data", *Proc. 1976 Symp. on Simulation of Computer Systems.*

[Shap86] L.D. Shapiro, "Join Processing in Database Systems with Large Main Memories", *Trans. on Database Systems 11(3)*, 1986.

[Yu93] P.S. Yu, D.W. Cornell, "Buffer Management Based on Return on Consumption In a Multi-Query Environment", *VLDB Journal 2(1)*, 1993.