Dynamic Memory Allocation for Multiple-Query Workloads

Manish Mehta *David J. DeWitt*
Computer Science Department, University of Wisconsin-Madison
mmehta@cs.wisc.edu

ABSTRACT — This paper studies the problem of memory allocation and scheduling in a multiple query workload with widely varying resource requirements. Several memory allocation and scheduling schemes are presented and their performance is compared using a detailed simulation study. The results demonstrate the inadequacies of static schemes with fixed scheduling and memory allocation policies. A dynamic adaptive scheme which integrates scheduling and memory allocation is developed and is shown to perform effectively under widely varying workloads.

1. Introduction

An important reason for the popularity of relational database systems is their ability to process ad-hoc queries posed by the users. Past research on query processing has dealt with issues of query optimization, scheduling, and resource allocation. Unfortunately, most of the work in this area has concentrated on processing single queries and does not consider multi-user issues. A particularly important issue that has largely been ignored is memory allocation for multiple-query workloads.

In the past, the majority of the memory allocation policies proposed have concentrated on allocating the 'proper amount' of memory to a single operator or query and have ignored the presence of other concurrently executing queries. Such 'localized' allocation techniques, which ignore global system behavior, can lead to poor performance and reduced throughput as the appropriate memory allocation for a query cannot be determined in isolation. Another drawback is that the memory allocation policy is largely independent of the query scheduling policy. To the best of our knowledge each of these schemes schedules queries in a first-come, first-served manner, delaying the actual execution only until sufficient memory becomes available (e.g. DBMIN [Chou85]). As we will demonstrate, this decoupling of query scheduling from memory allocation can have disastrous effects on the overall performance of the system.

The problem of processing multiple-query workloads becomes even more complex when the workload consists of different types of queries, each with widely varying memory requirements. As we will demonstrate, each type of query has a different sensitivity to how memory is allocated, and an effective memory allocation policy must carefully consider these differences if the overall performance of the system is to be maximized.

In this paper, we study the relative performance of various query scheduling and memory allocation policies. We compare schemes that perform scheduling and memory allocation independently to a new dynamic algorithm that integrates the two decisions. The performance of all the policies is compared using a detailed simulation model.

The rest of the paper is organized as follows. Section 2 discusses related work. The workloads studied in the paper are presented in Section 3 followed by a discussion of the metric used to compare the various policies in Section 4. Sections 5 and 6 describe the various scheduling and memory allocation policies, respectively. Section 7 presents the simulation model. The performance of the policies under various workloads is presented in Section 8 which is followed by our conclusions and future work in Section 9.

2. Related Work

The subject of memory allocation in database systems has been studied extensively. A significant portion of this work is related to allocating buffers to queries in order to minimize the number of disk accesses. The hot set model [Sacc86] defines the notion of a *hot set* for nested-loop join operations and tries to pre-allocate a join's hot-set prior to its execution. The DBMIN algorithm [Chou85] extends the idea of a hot-set by estimating the buffer allocation per file based on the expected pattern of usage. The DBMIN algorithm was further extended in Marginal Gains allocation [Ng91] and later it was employed to perform predictive load control based on disk utilization [Falo91]. Each of these schemes, with the exception of [Falo91], ignore the effects of other concurrently executing queries and thus make *localized* decisions for each query. In addition, none of these algorithms handle the allocation of memory to hash joins.

[1] This research was supported by the IBM Corporation through a Research Initiation Grant and through a donation by NCR.

**Proceedings of the 19th VLDB Conference
Dublin, Ireland, 1993**

The only work, to our knowledge, that directly handles memory allocation among concurrently executing hash-join queries is [Corn89, Yu93]. The authors introduce the concepts of *memory consumption* and *return on consumption* which are used to study the overall reduction in response times due to additional memory allocation. A heuristic algorithm based on these ideas is proposed for memory allocation. The algorithm's performance is determined by three runtime parameters. The problem with the heuristic is that the performance of the algorithm is very sensitive to the value of these parameters and the authors do not discuss how to set their parameter values. Also, the performance is based on average response times which, as discussed in Section 4, is not an appropriate metric for multiple class workloads.

All these schemes use a first-come first-served policy to service the arriving queries. As will be shown later, this can lead to bad performance for a multi-query workload.

Query scheduling algorithms proposed in [Chen92a, Schn90] handle scheduling of only single complex join queries and do not consider multiple queries. The batch scheduling algorithms studied in [Meht93a] cannot be directly applied as batch scheduling algorithms attempt only to maximize overall throughput and do not consider the impact on query response times.

Adaptive hash join algorithms, which can adapt to changing memory requirements have been proposed by [Zell90]. The implications of adapting to memory changes have not been investigated in the context of a multiple query workload. It is not very evident as to when taking memory from an executing query and giving it to some other query benefits the overall performance of the system. The same drawback lies in other schemes which dynamically change query plans at run-time [Grae89a].

3. Workload Description

In this study we consider only queries involving a single join and two selections. This simple workload allowed us to study the effects of memory allocation and load control without considering other complex query scheduling issues like pipelining and intra-query parallelism. For example, while there are several different execution strategies for complex queries such as left-deep, right-deep, and bushy scheduling [Schn90], nothing is known about their relative performance in a multiuser environment. Including queries with multiple joins would have made it impossible to separate the effects of memory allocation from other query scheduling issues. In addition most database systems (with the exception of Gamma [DeWi90] and Volcano [Grae89b]), execute multiple-query joins as a series of binary joins and do not pipeline tuples between adjacent joins in the query tree. Such simplified workloads have also been used previously in [Falo91, Ng91, Yu93].

Another simplification in the workload is that the selections were executed by scanning the data file and do not use indices. Also, the same join selectivity is used for each query — the number of output tuples produced by the join is one half of the number of the tuples in the outer relation. Inclusion of indices and varying join selectivity changes only the overall response time of the query and not the memory requirements of the queries. As a result, ignoring these simplifications does not affect the results qualitatively.

All joins were executed using the hybrid-hash join algorithm [2] [DeWi84, Shap86]. This algorithm operates in two phases. In the first phase, called the *build* phase, the inner relation is partitioned into n buckets $I_1, I_1, ... I_n$. The tuples that hash into bucket I_1 are kept in an in-memory hash table. Tuples that hash to one of the remaining buckets are written back to disk, with a separate file being used for each bucket. The number of buckets, n, is selected to be the minimum value such that each bucket I_i will fit into the memory space allocated to it at run time.

In the second phase, called the *probe* phase, the outer relation is also partitioned into n buckets $O_1, O_1, ... O_n$. Tuples that hash into bucket O_1, are joined immediately with the tuples in I_1. The other tuples are written to their corresponding bucket file on disk. If n=1, the algorithm terminates. Otherwise, the algorithms proceeds to join I_i and O_i for i=2,...,n. As the tuples from bucket I_i are read, an in-memory hash table is constructed. Then, O_i is scanned and its tuples are used to probe the hash table constructed from I_i looking for matching join attribute values. Results tuples are written back to disk. If the size of the inner relation is K pages, then the amount of memory available for the join must be at least $\sqrt{K}$ (also n is proportional to $\sqrt{K}$).

In a multiquery environment (composed of join queries from a number of different users or even perhaps join operations from a complex query composed of multiple joins), the memory requirements of each query can vary widely. To facilitate our study, we decided to classify each join as one of three different types: small, medium, or large. The idea is simple. Small queries are ones whose operands will fit in memory in most workloads. Medium queries are those which could always run in one pass (i.e. n=1) if they are alone in the system. Large queries are those whose execution always require multiple buckets. The workloads studied in this paper consist of mixes of queries from these three classes. As we will show later, in order to maximize the throughput of a system executing such a workload, it is necessary to carefully control the allocation of memory to queries from each of these different classes of queries.

[2] In the conclusions, we discuss the application of our techniques when other join algorithms are used.

The following sections describe the classes in more detail.

3.1. Small Query Class

The first query class represents join queries with minimal memory requirements. As such, this class of queries can always be expected to execute in one pass of the hybrid hash algorithm (i.e. no disk buckets are created during the build phase of the algorithm). For our experiments, we set the average memory requirement for this class to be 5% of the size of main memory with the actual requirement varying from 1 to 10% of the memory.

3.2. Medium Query Class

The second class represents join queries whose operands are about the same size as main memory. For our experiments we assumed that medium queries range from 10% to 95% of memory so an average of two such queries can fit into memory simultaneously. Thus, the memory requirements of this class of queries are nearly an order of magnitude larger than the small class. However, since the memory requirements of these queries are substantial compared to the total amount of main memory available, in the presence of other concurrently executing queries, medium queries will frequently require multiple join passes (depending on the actual scheduling and memory allocation policy employed).

3.3. Large Query Class

The last class represents queries whose operands are larger than the amount of physical memory available and thus always require multiple buckets for their execution. The large query sizes varied from the size of the memory upto four times the memory size.

4. Performance Objective and Metric

This section describes the performance metric used to compare the various memory allocation and scheduling policies examined in this paper. The metric is presented here in order to simplify the presentation of the algorithms described in the following section.

We spent a significant amount of time and effort trying to find a reasonable goal for a multi-class workload. The performance goals that are typically used for single class workloads are not adequate as they are biased towards one class or another. For example, combining the different query types into a single class and maximizing overall throughput biases the metric towards the smaller queries which have a higher throughput; algorithms that maximize the throughput of the smaller queries at the expense of the other classes will fare better. Similarly, minimizing the average response time is inadequate as the value of the metric is determined mainly by the large queries with long execution time and hence high response times.

In the absence of any universally accepted relative importance of the three workload classes (such as might be defined by an organization like the TPC council), we wanted a performance goal that weights the relative priority of the different classes equally. The performance goal chosen for this study is **fairness**.

The first step in measuring fairness is to obtain, for each class, the ratio of the observed average response time to average response time of the class when instances of the class are executed alone in the system. This normalizes the effect on the metric of the actual values of the response times of the different query types, which may differ by orders of magnitude.[3] Fairness is then defined as the standard deviation in the ratios obtained for each of the three classes in the first step. A higher standard deviation means a larger difference in the response time changes. This implies that the performance of some class was disproportionately worse when compared to other classes in the system and that the system was *unfair* towards that class.

In addition, we also measured the mean percentage change in response time across all three classes; a lower mean implies that, overall, the classes had better performance and that their observed response times were closer to their stand-alone response times. Whenever necessary, the response times for each individual class are shown along with the Fairness metric.

5. Memory Allocation Policies

We begin this section by describing three different ways of allocating memory to a hash join operation: *minimum*, *maximum*, and *available*. Next, we discuss how these alternatives can be applied to each of the three query classes: small, medium, and large. The resulting combinations provide a wide range of alternatives for controlling the allocation of memory to queries in a multiuser environment.

5.1. Join Memory Allocation

The amount of memory allocated to a hybrid hash join can range from the square root of the size of the inner relation to the actual size of the relation [DeWi84]. In a multi-query environment, allocating more memory to a join reduces not only the execution time of the operation but also disk and CPU contention as the query performs fewer I/O operations. The drawback of allocating more memory is that it may cause other queries to wait longer for memory to become available. Allocating less memory has the opposite effect. It increases execution time and disk contention but it may reduce the waiting time of other queries. Also, the overall multi-programming level in the system may be higher as the memory requirement of each individual query is reduced and more of them can be executed in parallel. We studied the effect of three allocation schemes on query performance.

[3] Normalization can also be done using expected response times (as in [Ferg93]) or the optimizer estimate of the query response time. Since we did not want to invent arbitrary expected response times for the queries we did not choose the first approach and the absence of a real optimizer prevented us from using the second.

Minimum

In this scheme, the join is always allocated the minimum memory required to process the join (i.e. the square root of the size of the inner relation). Hence a partitioning phase and a joining phase are always required. In effect, except for buckets I_1 and O_1, both relations get read twice and written once. This scheme is the least memory intensive but causes the most disk/CPU contention.

Maximum

In the *maximum* scheme (which only works for *small* and *medium* queries), each join is allocated enough memory so that its inner relation (i.e. the temporary relation resulting from the selection) will fit in memory. No tuples from the inner or outer relations get written back to bucket files on disk and the join is processed by reading each relation once. This allocation scheme minimizes the response time of the query but is also the most memory intensive. Also, since enough memory may not be available when the query enters the system, it may have to wait longer to begin execution. *Large* queries cannot use this scheme as the size of their inner relations is much greater than the amount of main memory available.

Available

The previous two schemes ignore how much memory is actually available at run-time. The *Available* scheme determines a query's allocation at run-time by examining available memory when the query is ready for execution. The query is given whatever memory is available subject to the constraint that the amount of memory allocated is at least the minimum required by the query (i.e. the square root of the size of the inner relation) and no more than the maximum. An allocation between two these extremes is interesting because it increases the number of tuples in bucket I_1 and O_1 and hence reduces the number of tuples that have to be read twice from disk (and written back once).

5.2. Impact of Memory Allocation on Each Class

Combining these schemes in all possible ways for each of the three query classes gives us 18 possible allocation schemes (3 small schemes * 3 medium schemes * 2 large schemes). As studying all possible combinations was not feasible, the number of combinations was reduced by removing some schemes for the *small* and *large* queries that are either redundant or that we expected to perform badly.

Queries in the *small* class require small amounts of memory to execute. From some preliminary experiments, we observed that in most cases the queries always got the maximum memory required. Also, as the memory needed by these queries is quite small (on the average 1/20th the size of physical memory), the queries did not wait long for memory to become available. As a result, the *Maximum* allocation scheme was always superior to the other schemes for this class of queries. Hence, we elected to only use this scheme for small queries.

Similarly, using Available for queries in the *large* class is a very bad idea as these queries can consume all of the memory, blocking out all other queries. Thus, only *Minimum* was used for these queries.

6. Scheduling Policies

The response time of a single query is determined not only by the execution time of the query but also by how queries are scheduled for execution. The scheduling decision gains added significance if the response times of the queries vary significantly. We considered three schedule policies, *FCFS*, *Responsible*, and *Adaptive*, which are described below.

6.1. FCFS

This simple scheduling policy is illustrated in Figure 1. Queries from all classes are directly sent to the memory queue as they arrive into the system. The memory queue is served in first-come first-served (FCFS) order and queries execute whenever sufficient memory is available. We present results for this scheduling policy in combination with all three medium-query memory allocation schemes - *Maximum*, *Available* and *Minimum*. The *FCFS-Maximum* combination can be particularly bad for *small* queries as they may need to wait behind *large* queries with much higher response times. The *medium* queries also suffer as the policy tends to restrict the multi-programming level (MPL) of medium queries. On the other hand, the *FCFS-Minimum* combination may cause very high system contention as a lot of queries can fit into the system if minimum memory is allocated to each. Moreover, all three schemes are inherently biased towards larger queries. The larger queries take up more memory and block out the smaller queries. In addition, the larger queries take longer to execute and thus are present in the system longer. This means that the average MPL of the larger queries could become disproportionately higher compared to the smaller queries as time progresses and larger queries tend to consume an increasing fraction of the system resources.

6.2. Responsible

We term the second scheduling policy *Responsible* as it is biased in favor of the smaller queries. As shown in Figure 2, small queries are sent directly to the Memory queue as in the FCFS scheduling policy. Medium and large queries on the other hand are queued in separate queues to prevent them from blocking small queries. These scheduler calculates the average amount of memory consumed by the small queries and leave that amount of memory for use by the small queries. The leftover memory is divided among the medium and large queries according to their memory allocation policy (e.g.

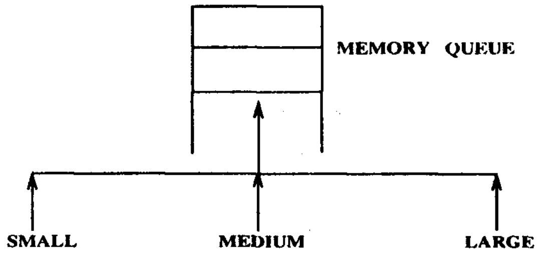

Figure 1: FCFS Scheduling

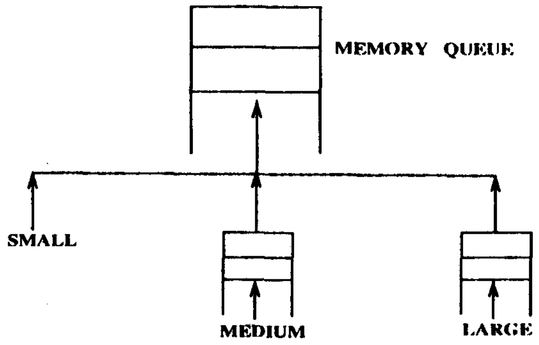

Figure 2: Responsible Scheduling

minimum for the large queries and available for the medium queries). In order to prevent starvation, each class has a minimum MPL of one. The amount of memory being used by small queries is calculated by multiplying the average MPL of the small queries by their average memory requirement. *Responsible* is thus biased towards the small queries as the small queries are sent directly to the memory queue while the medium and large queries wait in separate queues for execution.

6.2.1. Adaptive

FCFS scheduling lets the scheduling decision be determined by memory availability and is biased towards larger queries. *Responsible* is biased in favor of small queries. As we will demonstrate later, both schemes can exhibit very poor performance. The *Adaptive* scheduling scheme, on the other hand, tries to allocate resources such that the overall goal of fairness is achieved to the maximum extent possible. The basic mechanism used by Adaptive is the control of the MPL of each class of queries. For each class, an MPL queue is maintained in which incoming queries wait if the current MPL of the class is above a dynamically determined level (this level will fluctuate over time under the control of the Fairness Metric). Each MPL queue is serviced independently in FIFO order. When a query belonging to a particular class completes, the next waiting query in the queue is moved from the class's MPL queue to the memory queue. Figure 3 illustrates the various queues managed by the Adaptive scheduling algorithm.

As mentioned before, small queries always execute at maximum memory and large queries always execute at their minimum. Using these two properties, the algorithm calculates the average amount of memory consumed by queries from both classes. The remainder is then distributed among the medium queries. As an example, assume that there is 32 MB of memory available, that maximum MPL of the small class is 10 and that the small queries consume, on the average, 0.5 MB of memory. Similarly, assume that the large query MPL is 2 and that they require 3 MB on the average. Thus, the memory available to the medium queries is 21 MB (32 - 10*0.5 - 2*3). If the current maximum medium MPL is 5, then each medium query receives 4.2 MB of memory. Dividing the memory in this manner ensures that the smaller queries are not blocked out by the medium and large queries. Also, note that the memory allocated to a query is related directly to the MPL of its class. The adaptive algorithm thus integrates query scheduling and memory allocation.

The adaptiveness of the algorithm arises from the fact that the MPLs of the query classes are determined dynamically based on system parameters. Periodically, the algorithm checks the average response times for each of the query classes and evaluates the Fairness Metric (Section 4). If the variance is high, implying that some class is doing much better than the other two, the algorithm takes a compensating action so that the offending class is *throttled back*. The decision may mean increasing the MPL of the class that is doing worst so that it receives more resources or decreasing the MPL of the offending class to reduce its resource consumption. The actual algorithm is presented in more detail in Figure 4.

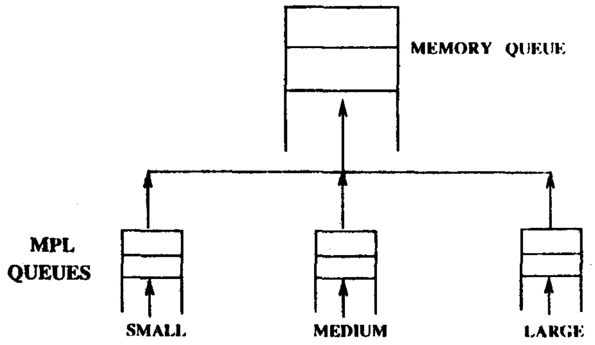

Figure 3 - Adaptive Scheduling

```
if (Activate) {
        calculate the average response times for each class
        calculate Fairness Metric (Dev) and mean Response Time change (MRT)
        calculate difference of percentage change from the mean
            SP  = difference in percentage change for small class
            MP  = difference in percentage change for medium class
            LP  = difference in percentage change for large class
        if (Dev > DevThreshold) {
            sort SP, MP and LP (a higher value means the corresponding class has poorer performance)
            switch {
                SP > MP > LP   —>  increase small MPL || decrease large MPL || decrease medium MPL
                MP > SP > LP   —>  increase medium MPL || decrease large MPL || decrease small MPL
                LP > SP > MP   —>  decrease medium MPL || increase large MPL || decrease small MPL
                LP > MP > SP   —>  decrease small MPL || increase large MPL || decrease medium MPL
                SP > LP > MP   —>  increase small MPL || decrease medium MPL || decrease large MPL
                MP > LP > SP   —>  decrease small MPL || increase medium MPL || decrease large MPL
            }
        }
}
```

Figure 4: Pseudo Code of the Adaptive Algorithm

We followed a few general principles in deciding the action to be taken in response to the state the system is in. The increase and decrease of MPLs was done under the constraints that the MPL for a class can never be zero and that the corresponding memory associated with the class cannot exceed the total memory size. Also, when the algorithm must decide what class should have its maximum MPL adjusted, the algorithm always tries to change the MPL of the smaller query class first. This is because the smaller queries have shorter response times and changing their MPL makes the system respond to the change faster.

As can be seen in Figure 4, there are two parameters that control how often the adaptive algorithm adjusts the query workload. The algorithm is invoked every time the variable, Activate, is set to TRUE. If invoked too frequently, the algorithm is susceptible to transients in the workload; on the other hand infrequent activation makes the algorithm slow to respond to workload changes. For the purposes of this paper, Activate was set to TRUE at the completion of each *medium* query.

The second parameter, *DevThreshold*, controls how much of a deviation the algorithm tolerates before making a compensating action. A low threshold means that the system will respond to even very low changes in the deviation and thus may react too quickly to a transient change. A very high threshold means that the algorithm does not change the runtime parameters even if the deviation is very high and some class is performing relatively worse. The threshold value was set to 2.0 for the purposes of the performance study. A sensitivity analysis of the algorithm towards these two parameters can be found in Section 8.4.

During execution, the adaptive algorithm needs to compute the average observed response times for each of the three query classes in order to calculate the percentage change from the ideal case. This requires that the algorithm be provided with the response time of a representative query from each class, run with the MPL set to 1. Also, if the activation frequency of the algorithm is such that an activation can occur before a single query of some class has completed execution, the algorithm needs a way of calculating the expected response time of the class. The solution used in this study is to require as input two numbers for each class when the queries are run in a multi-user mode: *Response Time* and the observed *Disk Read Response Time*. If the system is I/O bound, the response time is in most cases proportional to the disk response time so that we can compare the current *Disk Read Response Time* to the above number and estimate the expected response time accordingly. This technique could be replaced by another estimation technique without changing the algorithm. Even if the estimate is quite inaccurate, the penalty is not severe as the estimate is no longer needed once a **single** query from the class has completed.

7. Simulation Model

The simulator used for this work was derived from a simulation model of the Gamma parallel database machine which had been validated against the actual Gamma implementation. The simulator is written in the CSIM/C++ process-oriented simulation language [Schw90].

The simulator models a centralized database system. The system consists of a single processing node, composed of one CPU, memory, one or more disk drives, and a set of external terminals from which queries are submitted. As queries arrive in the system they are first routed to a special scheduler task that controls the scheduling and execution of all transactions present in the system. The database is modeled as a set of relations that are declustered [Ries78, Livn87] over all the disk drives. The simulator models the database system as a closed queueing system [Reis80].

7.1. Terminals

The terminals model the external workload source for the system. Each terminal sequentially submits a stream of queries of a particular class. After each query is formulated, the terminal sends it to the scheduler task for execution and then waits for a response before submitting another query.

7.2. Processing Node

The processing node in the system is modeled as a single CPU, four disk drives, and a buffer pool. The CPU uses a round-robin process scheduling policy. The buffer pool models a set of main memory page frames. Page replacement in the buffer pool is controlled via the LRU policy extended with *love/hate* hints (like those used in the Starburst buffer manager [Haas90]). These hints are provided by the various relational operators when fixed pages are unpinned. For example, *love* hints are given by the index scan operator to keep index pages in memory; *hate* hints are used by the sequential scan operator to prevent buffer pool flooding. In addition, a memory reservation system under the control of the scheduler task allows memory to be reserved in the buffer pool for a particular operator. This memory reservation mechanism is used by the join operators to ensure that enough memory is available to prevent their hash table frames from being stolen by other operators.

The simulated disk models a Fujitsu Model M2266 (1 GB, 5.25") disk drive. This disk provides a 256 KB cache that is divided into eight 32 KB cache contexts for use in prefetching pages for sequential scans. In the disk model, which slightly simplifies the actual operation of the disk, the cache is managed as follows: 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 pages (4 blocks) are read into a cache context as part of transferring the page originally requested from the disk into memory. Subsequent requests to one of the prefetched blocks can then be satisfied without incurring an I/O operation. A simple round-robin replacement policy is used to allocate cache contexts if the number of concurrent prefetch requests exceeds the number of available cache contexts. The disk queue is managed using an elevator algorithm.

The key parameters of the processing nodes, along with other configuration parameters, are listed in Table 1. The system memory which is 4MB is obviously small but was chosen only to keep the simulation time low. The crucial factor for performance is not the actual size of the memory but the relative size of the queries to the the system memory which was scaled accordingly. The software parameters are based on actual instruction counts taken from the Gamma prototype. The disk characteristics approximate those of the Fujitsu Model M2266 disk drive described earlier.

Configuration/Node Parameter	**Value**
Number of Disks	4 disks
CPU Speed	20 MIPS
Memory	4 MB (varied)
Page Size	8 KB
Disk Seek Factor	
Disk Rotation Time	16.667 msec
Disk Settle Time	2.0 msec
Disk Transfer Rate	3.09 MB/sec
Disk Cache Context Size	4 pages
Disk Cache Size	8 contexts
Disk Cylinder Size	83 pages
Tuple Size	200 bytes
CPU Cost Parameter	**No. Instructions**
Initiate Join	40000
Terminate Join	10000
Terminate Select	5000
Apply a Predicate	100
0.617	Read Tuple
Write Tuple into Output Buffer	100
Probe Hash Table	200
Insert Tuple in Hash Table	100
Start an I/O	1000
Copy a Byte in Memory	1

Table 1: Simulator Parameters and Values

7.3. Scheduler

The scheduler task accepts queries from terminals and implements the scheduling and memory allocation algorithms discussed in Sections 5 & 6. A query may wait either in it's MPL queue if the MPL is too high or for memory in the Memory queue. The scheduling policy determines the order in which these queues are serviced. Once the query is scheduled for operation, the scheduler also coordinates the startup and termination of all the operators in the query.

7.4. Operators

The queries in the work reported here use only two basic relational operators: select and join. Results tuples are stored back in the database after the queries complete. Each select process also spawns an additional scan process that reads pages from the disk and passes them on to the select process. The select process then applies the filtering predicate(s) on the scanned pages and passes the qualifying tuples on to the join process. The simulator models the actual execution in detail with accurate modelling down to tuple-level operations.

8. Performance Results

This section presents the results of our performance evaluation of the different scheduling and memory allocation policies. First, we describe the characteristics of the base workload and the results obtained. Then a variety of other workloads are examined.

8.1. Base Workload

The workload parameters used for all the experiments are shown in the Table 2. The think times for the query classes were chosen arbitrarily except that the larger classes have larger think times to keep their MPL relatively low. The number of terminals for each class was varied such that the system moved from a lightly loaded state to a heavily loaded state. In addition, results are also presented where the ratio of the terminals of different classes was varied to change the mix of the workload.

Class	Number of Terminals	Think Time
Small	0-100	20 sec
Medium	0-80	150 sec
Large	0-10	450 sec

Table 2: Experimental Parameters

The terminal ratio for the first experiment was set at 10:8:1 (small:medium:large). However, this does not represent the ratio of the MPLs; due to differences in think times and execution times, the actual ratio is closer to 5:2:1. Furthermore, the MPL per class varies with the different algorithms. Figures 5-7 show the response times of the different classes as the number of small terminals is varied from 10 to 50; the medium class terminals change accordingly from 8 to 40 and the large class terminals range from 1 to 5.

As discussed in Section 6, each FCFS scheme (solid lines) is biased towards larger queries. Thus, the small queries show very high response times as they get blocked behind the larger queries as the load increases. FCFS-Maximum, which gives most memory to the medium queries blocks the most small queries and performs the worst. FCFS-Minimum, which gives the least memory to the medium queries, blocks the small queries the least and performs better while FCFS-Available performs in between the two schemes. The medium queries fare similarly under each of the FCFS policies. The response times increase as the load increases in each case. The effect on the large queries is very different; FCFS-Maximum, which minimizes disk contention (by limiting the number of small and medium queries in the system), has the best response time among the three FCFS schemes followed by FCFS-Available and FCFS-Minimum.

The Responsible schemes (dashed lines) all do a very good job handling the small queries while making the medium queries suffer. The reason is that these schemes leave memory for the smaller queries. This leaves less memory for medium queries which get queued up. The large queries also get queued up but the response times do not increase correspondingly because of two reasons. Firstly, there are less number of large queries in the system and secondly, the low medium query MPL reduces disk contention significantly. Consequently, the large queries have low execution times leading to smaller response times.

The Adaptive scheme behaves responsibly towards the small queries and prevents excessive blocking behind larger queries. Also, unlike the Responsible schemes, the algorithm is able to keep the medium queries from being penalized by carefully controlling class MPLs. Adaptive leads to highest response times for the large queries. Since large queries have much higher base response times

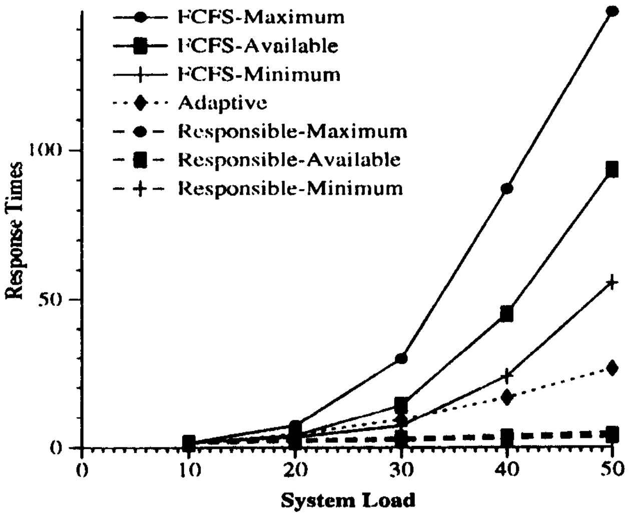

Figure 5: Small Query Class

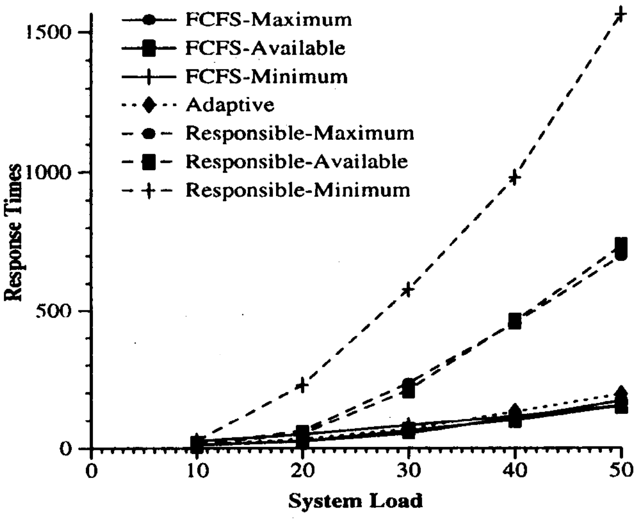

Figure 6: Medium Query Class

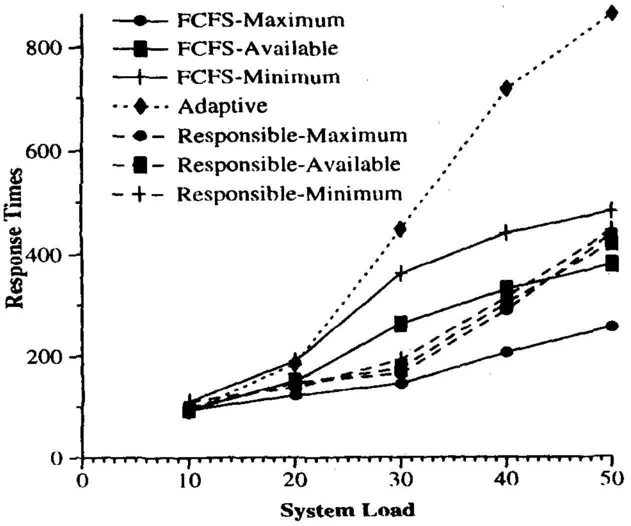

Figure 7: Large Query Class

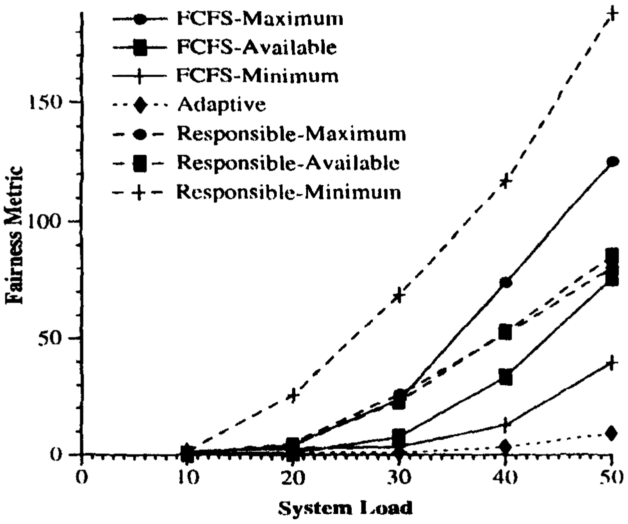

Figure 8: Fairness

compared to the small and medium queries, a high response time for large queries is desirable for fairness.

The performance of the various schemes with regard to the fairness metric is shown in Figure 8. The FCFS schemes do poorly as they are unfair towards the small query class while the Responsible schemes are bad because of their unfairness towards queries from the medium query class. The adaptive algorithm is the fairest among all the schemes across the entire load range as it prevents the smaller queries from being blocked and at the same time does not penalize the larger classes.

We now present a series of six other workloads and compare the performance of the algorithms for each.

8.2. Doubling the Terminals

In this section we examine the performance of the algorithms for three workloads in which the number of terminals for a particular query class is doubled compared to the base case above. These workloads vary the ratio of the various queries in the workload and test how well the algorithms adapt to various workloads. Only the fairness metric numbers are presented for these three workloads. Most of the response time numbers were qualitatively similar to the base case; any differences are explicitly mentioned in the discussion of each workload.

8.2.1. Doubling Small Terminals

In this workload, we increased the ratio of small to the medium terminals from 10:8 to 20:8 (the reader should keep in mind that due to the difference in think times of the two classes the actual MPLs are more in the range of 10:2). Figure 9 shows the performance of the various schemes on this workload.

The FCFS schemes, which are unfair towards the smaller queries, perform relatively worse for this workload compared to the base case because this workload has more small queries, so the penalty of blocking smaller queries is even higher. Since the Responsible schemes are biased towards small queries, this workload makes them even more unfair to the medium queries and their performance also worsens. The Adaptive algorithm, which is able to do a fair job of distributing memory among among the classes, shows the best overall performance.

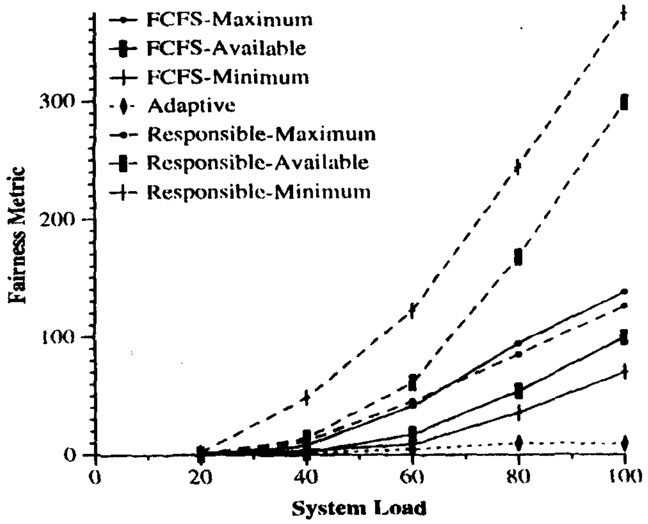

Figure 9: Fairness Metric: Small Terminals Doubled

8.2.2. Doubling the Medium Terminals

In this workload we doubled the number of terminals for the medium class. The performance of the various schemes is shown in Figure 10. As in the two previous workloads, compared to the Adaptive algorithm, the FCFS and the Responsible schemes perform much worse.

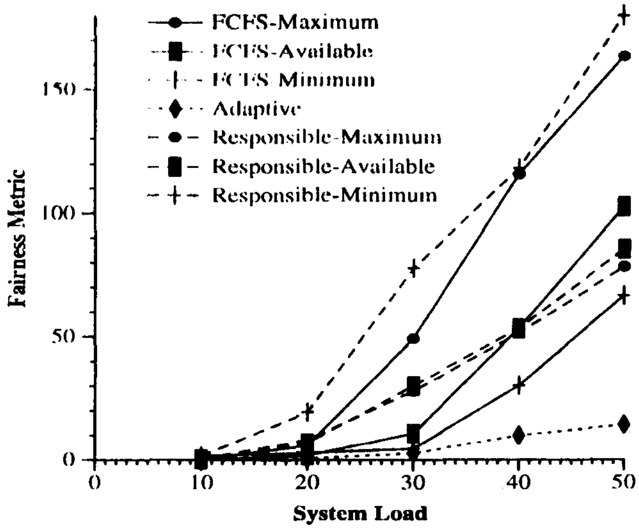

Figure 10: Fairness metric: Medium Terminals Doubled

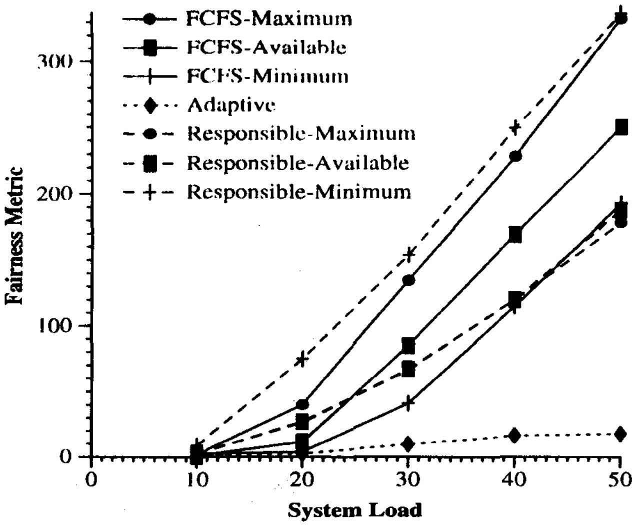

Figure 11: Fairness Metric: Large Terminals Doubled

The FCFS schemes block more small queries because the workload has a larger number of medium queries. The Responsible schemes, which exhibited poor performance for the medium queries in the base case above, just get worse as the number of medium queries increases. The adaptive algorithm adapts to the increased medium workload and is thus able to do a better job at reducing the variance among the different query classes.

8.2.3. Doubling the Large Terminals

Figure 11 shows the performance of the different schemes for a workload in which the number of large terminals has been doubled. The numbers show that the FCFS and Responsible schemes are much worse than the Adaptive scheme. The FCFS schemes do not explicitly control the MPL of the large query class and large queries are always able to execute with very little waiting for memory. The Responsible schemes perform badly for the medium queries and *also* for the large queries. Since the number of large terminals is now doubled, the large queries also get queued up leading to much higher response times. Adaptive is again the fairest algorithm as restricts the larger queries from dominating the system and prevents excessive queueing at the same time.

8.3. Reduced Number of Query Types

The previous three experiments varied the ratio of terminals among the different query classes, but the workload always contained queries from all three classes. We next tried workloads in which only two of the three classes were present. The performance of the algorithms on these workloads is presented next. The Responsible schemes, which have been shown to perform quite badly in the previous workloads, have been omitted so that the differences between the other algorithms can be shown and discussed in more detail. The performance of the workload with only Small and Large queries has also been omitted because the FCFS and Adaptive schemes perform the same for this workload. The interested reader can look at [Meht93b] for the performance figures of this workload.

8.3.1. Small-Medium Workload

This workload consists only of small and medium queries. Figures 12 and 13 show the response times of the small and medium classes, respectively.

The performance of the algorithms for the small class is the same as before. The Adaptive algorithm shows the best response time followed by FCFS-Minimum, FCFS-Available and FCFS-Maximum. The performance of these schemes on queries from the medium class is quite different though. FCFS-Maximum has the best response times for the medium queries. This is quite surprising because FCFS-Maximum makes the medium queries wait the longest for memory yet it has the best response time for the class. The reason is that making the medium queries wait for memory lets the queries execute in only one pass, reducing the disk contention significantly. The reduction in disk contention more than compensates for the time spent waiting for memory to become available. While FCFS-Available and FCFS-Minimum induce fewer memory waits for queries from the medium query class, they may cause the medium queries to execute with multiple join passes. As the system is already loaded with other small queries, the added disk contention causes the schemes to perform worse. Adaptive performs similar to FCFS-Minimum except at high loads where it penalizes the medium queries in order to achieve lower response times for small queries.

The fairness metric for all the algorithms is shown in Figure 14. In addition, we also present the mean percentage change obtained for the two classes of queries in Figure 15. Not only does the adaptive scheme achieve

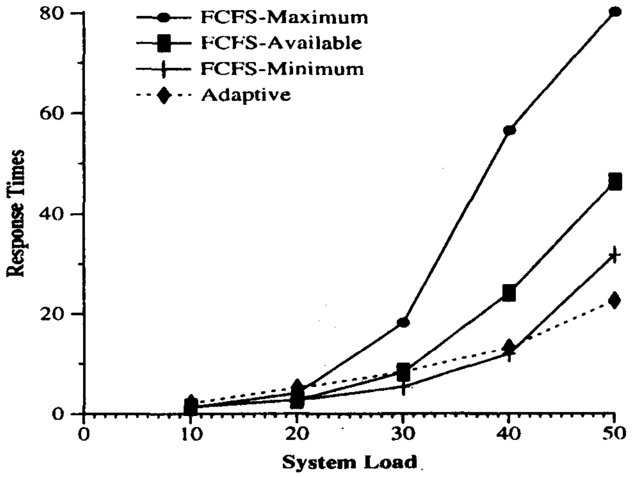

Figure 12: Small-Medium Workload, Small Class

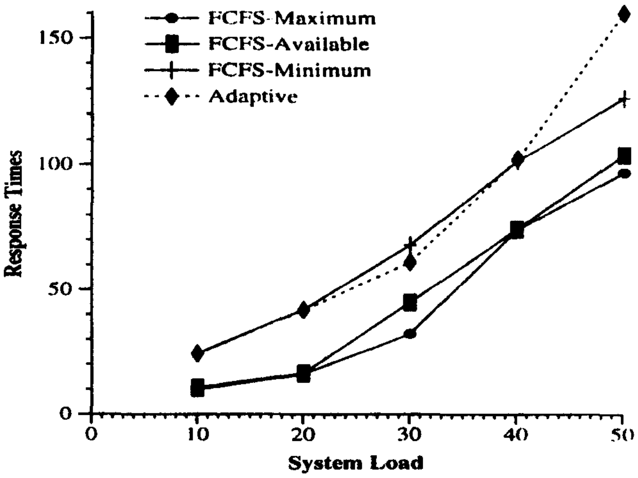

Figure 13: Small-Medium Workload, Medium Class

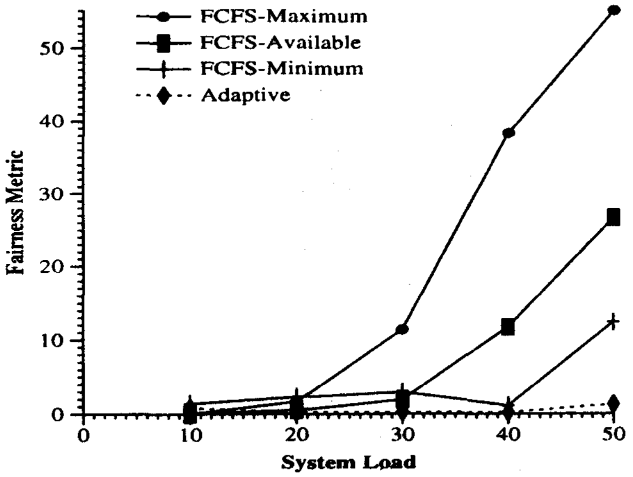

Figure 14: Small-Medium Workload, Fairness

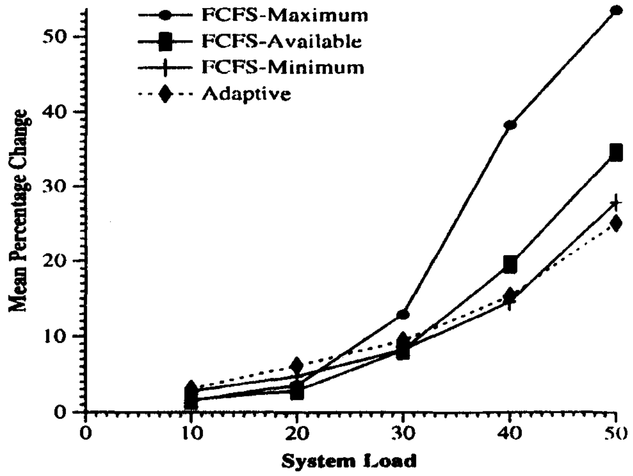

Figure 15: Mean Percentage Ratio

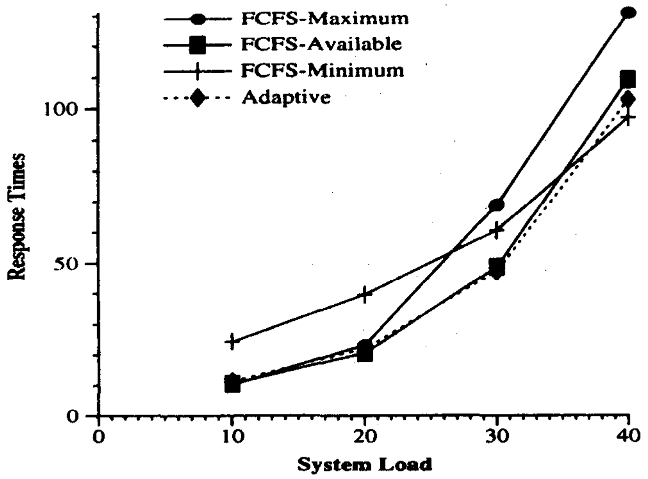

Figure 16: Medium-Large Workload, Medium Class

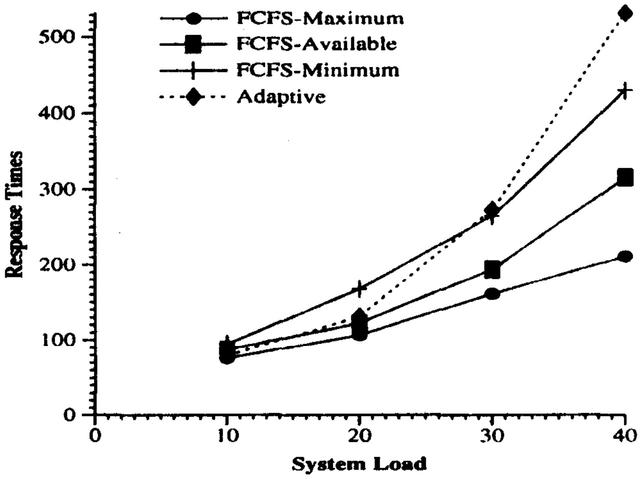

Figure 17: Medium-Large Workload, Large Class

the best fairness but it also provides the best mean percentage change compared to all the FCFS algorithms.

8.3.2. Medium-Large Workload

The response times of Adaptive and FCFS schemes for the Medium-Large workload, which has no small queries, are shown in Figure 16 and 17. Figure 18 presents the fairness achieved by the algorithms while the mean percentage change is shown in Figure 19. The graphs demonstrate that FCFS-Maximum performs well for medium queries at low loads since the execution times and queueing times are small due to a low number of queries in the system. However, at high loads, queueing times increase substantially as queries wait for memory leading to high response times. FCFS-Minimum, on the other hand, is the worst scheme for medium queries at low loads due to high execution times. As the load increases, FCFS-Minimum performs well because the queueing times remain low as more queries can be executed concurrently compared to other schemes. FCFS-Available performs similar to FCFS-Maximum for low loads and FCFS-Minimum for higher loads. The Adaptive algorithm also shows good performance throughout the range and behaves like FCFS-Maximum for low loads and like FCFS-Minimum for high loads.

FCFS-Maximum is the best algorithm for large queries because low disk contention due to low medium MPLs leads to low response times for large queries. FCFS-Minimum causes the most disk contention and is the worst scheme for large queries while FCFS-Available performs in between. The response times of the large queries for Adaptive algorithm increase rapidly as load increases and the algorithm performs the worst at high loads. The reason for this degradation is a drawback of the manner in which memory gets allocated to medium queries in Adaptive. Adaptive makes the large queries wait in order to improve the performance of the medium queries. However, due to the manner in which medium queries are allocated memory, the extra memory is not utilized fully and the performance of the medium queries does not improve much. Medium query memory is divided equally among concurrent medium queries and ,in several cases, there is not much of an improvement in allocating memory between the minimum and maximum needed by the query. We plan to investigate the performance of the heuristic proposed in [Yu93] where a query gets either the maximum or the minimum memory required in the future. Even though it leads to high mean percentage changes at high loads (figure 19), Adaptive is the best algorithm for fairness followed by FCFS-Minimum, FCFS-Available and FCFS-Maximum as shown in figure 18.

8.4. Sensitivity Analysis

The performance of the *Adaptive* algorithm is controlled by the values of two parameters — *DevThreshold* and *Activate*. *Activate* determines the frequency with which the Adaptive algorithm is executed and *DevThreshold* determines the variance in response times tolerated by the Adaptive algorithm. The algorithm makes a change in MPLs only if the current fairness value is above the DevThreshold. The sensitivity of the algorithm to the values of these parameters is investigated in the next two experiments. The performance of the algorithm was studied for two workloads containing queries from all three classes. The first workload containing 20 small, 16 medium and 1 large terminal represents a lightly loaded system, while the second workload which has 50 small, 40 medium and 1 large terminal represents a heavily loaded system.

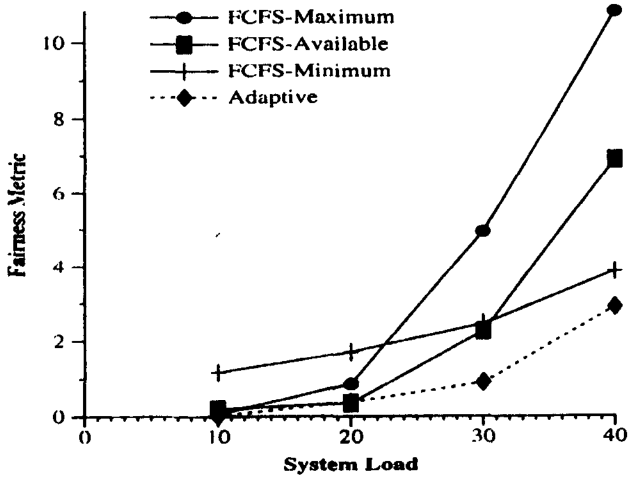

Figure 18: Medium-Large Workload, Fairness

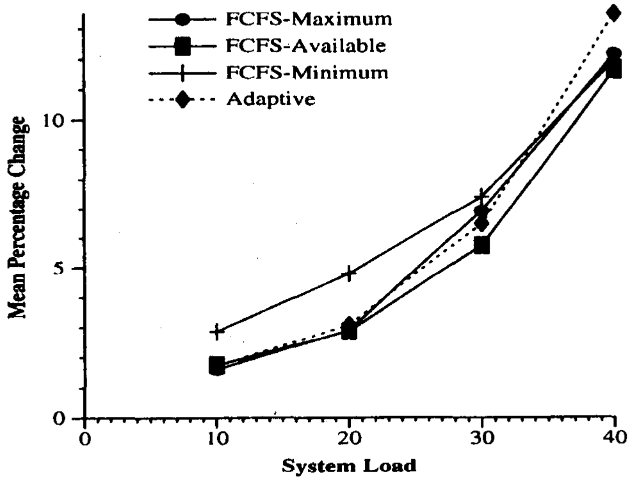

Figure 19: Mean Percentage Change

8.4.1. Sensitivity to Activate

Figure 20 shows the sensitivity of the Adaptive algorithm to the frequency with which the algorithm is executed. The graphs show the performance of the algorithm as the execution interval is varied from 5 to 640 seconds (simulated execution time) for the two workloads described before. Increasing the activation period implies that Adaptive is executed less often and reacts much more slowly to the workload. The algorithm does not fare badly for the lightly loaded system because the system load is so low that performance does not deteriorate even with the Adaptive scheme being activated less often. On the other hand, for the heavily loaded system, the performance is more sensitive to the activation period. If the activation period is low (less than 100), the algorithm is over-reactive to the system state and the performance degrades. As the activation period increases the performance becomes less reactive and the performance improves. Increasing the period beyond 400 seconds again causes the algorithm to degrade as the algorithm is now under-reactive. The region of good performance corresponds to about the completion time of a medium query. Hence activating the algorithm on every medium query completion seems to be a good heuristic.

8.4.2. Sensitivity to DevThreshold

Figure 21 shows the performance of Adaptive as the DevThreshold is varied from 0.5 to 32. As the DevThreshold increases, Adaptive changes the MPLs of the classes less frequently and performance degrades. In the lightly loaded case, fairness decreases as DevThreshold is increased to 8. As DevThreshold is increased further, Adaptive cannot perform any worse and the fairness value stablizes at 3.65. As expected, Adaptive performs much worse for the heavily loaded system. For a threshold values below 1.0 the algorithm takes actions very frequently and fairness deteriorates as the algorithm is unstable. As the threshold value is increased further the performance improves with the lowest value achieved at 2.0. However, the algorithm degrades fast as the threshold value is increased beyond 3. This shows that Adaptive is most sensitive to the value of DevThreshold and reasonable performance can be achieved if the value is between 1 and 3.

The boundary between small and medium queries was fixed at 10% for the experiments presented so far. Additional experiments about the sensitivity of the algorithm to the boundary between small and medium queries have been omitted here due to lack of space and the results can be found in [Meht93b].

9. Conclusions and Future Work

This paper has investigated the problem of memory allocation for a multiple query workload consisting of queries belonging to different classes with very different resource requirements. The results obtained demonstrate that the performance of the memory allocation policy is closely linked to the scheduling policy used to service arriving queries. Three memory allocation policies and two scheduling policies were described. The three memory allocation and two scheduling policies can be combined to produce six different schemes, each of which makes separate allocation and scheduling decisions. Their performance was compared to that of a new adaptive algorithm that combines the two decisions. A simulation study demonstrated that the adaptive algorithm provides excellent performance under a wide variety of workloads and, in several cases, performs much

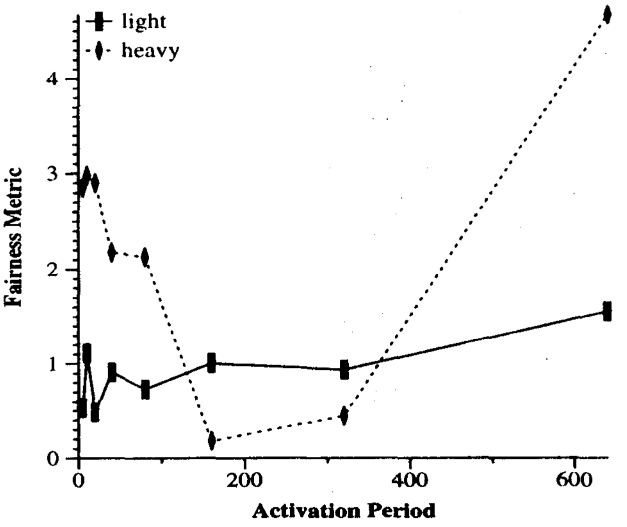

Figure 20: Sensitivity to Activate

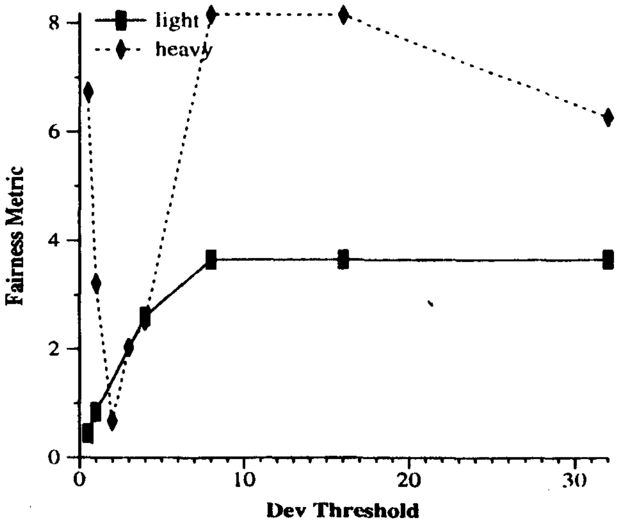

Figure 21: Sensitivity to DevThreshold

better than any of the other schemes. We also demonstrated that an intelligent combination of allocation and scheduling decisions can give substantial gains in performance.

There are several possible extensions to the adaptive algorithm presented in this paper. In several cases, fairness is not the only criteria when executing a multi-class workload. Often workloads consist of queries of different types with each type having a fixed response time or throughput goal. We want to investigate how the present scheme can be extended to handle such performance objectives. All the queries in our workloads were hash join queries. However, the adaptive algorithm can also be applied to queries using other join methods. The Adaptive algorithm uses observed response time estimates to control the MPL of each of the classes. This could be used even with other join methods like nested-loop and sort-merge. The division of queries into the three classes would also be valid for Sort-Merge and Nested Loop join. In the case of sort-merge joins, queries could be divided into small, medium and large classes based on the size of the memory needed for sorting the relations. The size of the inner relation could be used to make the division for Nested-Loop join queries.

We also want to investigate whether the policies that we have developed can be adapted to processing mixes of transactions and queries. Finally, an important future objective is to implement the adaptive scheme described in this paper and to measure its performance on some real database workloads.

10. Acknowledgements

The authors would like to acknowledge Kurt Brown for the idea of the fairness metric used in this paper and for reviewing the paper and Jim Gray for suggesting the idea of varying the system load to study performance.

11. References

[Chen92a] Ming-Syan Chen, et. al., "Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins", *Proc. 18th VLDB Conf.*, to appear, Vancouver, Canada, August 1992.

[Chou85] H. Chou, D. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems", *Proc. 11th Int'l VLDB Conf.*, Stockholm, Sweden, Aug. 1985.

[Corn89] D. Cornell, P. Yu, "Integration of Buffer Management and Query Optimization in a Relational Database Environment", *Proc. 15th Int'l VLDB Conf.*, Amsterdam, The Netherlands, Aug, 1989.

[DeWi84] D. DeWitt, et al, "Implementation Techniques for Main Memory Database Systems," *Proc. ACM SIGMOD Conf.*, Boston, MA, June 1984.

[DeWi90] D. DeWitt, et al, "The Gamma Database Machine Project", *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

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

[Falo91] C. Faloutsos, R. Ng, T. Sellis, "Predictive Load Control for Flexible Buffer Allocation", *Proc. 17th Int'l VLDB Conf.*, Barcelona, Spain, Sept. 1991.

[Grae89a] G. Graefe, "Dynamic Query Evaluation Plans", *Proc. ACM SIGMOD '89 Conf.*, Portland, Oregon, 1989.

[Grae89b] J. Gray, ed., "Volcano: An extensible and parallel dataflow query processing system.", *Computer Science Technical Report*, Oregon Graduate Center, Beavorton, OR, June 1989.

[Haas90] L. Haas et. al., "Starburst Mid-Flight: As the Dust Clears", *IEEE Trans. on Knowledge and Data Eng.*, 2(1), March, 1990.

[Livn87] M. Livny, S. Khoshafian, and H. Boral, "Multi-Disk Management Algorithms", *Proc. ACM SIGMETRICS Conf.*, Alberta, Canada, May 1987.

[Meht93a] M. Mehta, V. Soloviev, D. DeWitt, "Batch Scheduling in Parallel Database Systems", *9th Int'l Conf. on Data Engineering*, Vienna, April 1993.

[Meht93b] M. Mehta, D. DeWitt, "Dynamic Memory Allocation for Multiple-Query Workloads", *Technical Report*, University of Wisconsin-Madison, June, 1993.

[Ng91] R. Ng, C. Faloutsos, T. Sellis, "Flexible Buffer Allocation Based on Marginal Gains", *Proc. ACM SIGMOD '91 Conf.*, Denver, CO, May 1991.

[Reis80] M. Reiser, and S. Lavenberg, "Mean Value Analysis of Closed Multichain Queueing Networks", *JACM*, 27(2), April, 1980.

[Ries78] D. Ries, and R. Epstein, "Evaluation of Distribution Criteria for Distributed Database Systems", *UCBERL Technical Report M78/22*, UC Berkeley, May 1978.

[Sacc86] G. Sacco, M. Schkolnick, "Buffer Management in Relational Database Systems", *ACM TODS*, 11(4), December 1986.

[Schn90] D. Schneider, and D. DeWitt, "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines," *Proc. 16th VLDB Conf.*, Melbourne, Australia, Aug. 1990.

[Schw90] H. Schwetman, *CSIM Users' Guide*, MCC Technical Report No. ACT-126-90, Microelectronics and Computer Technology Corp., Austin, TX, March 1990.

[Shap86] L. Shapiro, "Join Processing in Database Systems with Large Main Memories", *ACM TODS*, 11(3), Sept. 1986.

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

[Zell90] H. Zeller, J. Gray, "An Adaptive Hash Join Algorithm for Multiuser Environments", *Proc. 16th Int'l VLDB Conf.*, Melbourne, Australia, Aug. 1990.