Predictive Load Control for Flexible Buffer Allocation

Christos Faloutsos and **Raymond Ng** and **Timos Sellis**
Department of Computer Science
University of Maryland
College Park, Maryland 20742, U.S.A.

Abstract

Existent buffer allocation methods use pre-calculated parameters to make buffer allocation decisions. The main idea proposed here is to design an *adaptable* buffer allocation algorithm that will automatically optimize itself for the specific query workload. To achieve this adaptability, we propose using run-time information, such as the load of the system, in our buffer allocation decisions. In particular, based on a simple queueing model, we use *predictors* to estimate whether a buffer allocation will improve the performance of the system. Simulation results show that the proposed method consistently outperforms existent algorithms.

1 Introduction

Previous proposals on buffer allocation are based either exclusively on the availability of buffers at runtime[4, 6, 9] or on the access patterns of queries[1, 8]. In [7] we propose a unified approach for buffer allocation in which both factors are taken into consideration. Simulation results show that this added flexibility in buffer allocation achieves good performance.

However, one weakness common to all the above approaches is that they are *static*. That is, they utilize parameters, which are calculated beforehand, for a given workload. The values for these parameters are not necessarily optimal as the workload varies. In this paper, we propose a new family of flexible buffer management techniques that are adaptable to the workload of the system. The basic idea of our approach is to use *predictors* to predict the effect a buffer allocation decision will have on the performance of the system. These predictions are based not only on the availability of buffers at runtime and the characteristics of the particular query, but also on the dynamic workload of the system. Two predictors are considered in this paper: *throughput* and *effective disk utilization*. Simulation results show that buffer allocation algorithms based on these two predictors perform better than conventional ones.

In Section 2 we review related work and motivate the research described in this paper. In Section 3 we present formulas for computing the expected number of page faults for different types of database references, and outline flexible buffer allocation strategies. Then in Section 4 we introduce the predictors and present the policies for predictive allocation schemes. Finally, we present in Section 5 simulation results that compare the performance of our algorithms with other allocation methods. We conclude in Section 6 with a discussion on ongoing research.

2 Related Work and Motivation

In relational database management systems, the buffer manager is responsible for all the operations on buffers, including load control. That is, when buffers become available, the manager needs to decide whether to activate a query from the waiting queue and how many buffers to allocate to that query. If too few buffers are given, the query will cause many page faults, and the response time will be too high. If too many buffers are given, other queries waiting to get into the system will be blocked out, and the throughput of the whole system will decrease. Achieving the "golden cut" is the objective of a buffer allocation algorithm.

As depicted in Figure 1 which outlines the buffer manager and its related components, the buffer pool area is a common resource and all queries – queries currently running and queries in the waiting queue – compete for the buffers. Like in any competitive environment, the principle of supply and demand, as well as protection against starvation and unfairness must be employed. Hence, in principle, the number of buffers assigned to a query should be determined based on the following factors:

1. the *demand* factor – the space requirement of the query as determined by the access pattern of the

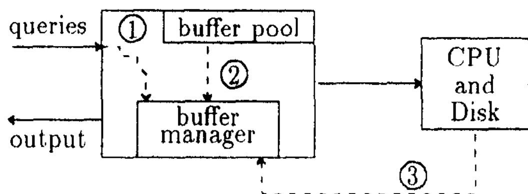

Figure 1: Buffer Manager and Related Components

algorithms	dynamic workload	access patterns of queries	availability of buffers at runtime
FIFO, LRU, Random, etc.	–		√
Hot-Set, DBMIN	–	√	
MG-x-y	–	√	√
proposed algorithms	√	√	√

Table 1: Classification of Allocation Algorithms

query (shown as path (1) in Figure 1),

2. the *buffer availability* factor - the number of available buffers at runtime (shown as path (2) in Figure 1), and
3. the *dynamic load* factor - the characteristics of the queries currently in the system (shown as path (3) in Figure 1).

Based on these factors, previous proposals on buffer allocation can be classified into the following groups, as summarized in Table 1.

Allocation algorithms in the first group consider only the buffer availability factor. They include variations of First-In-First-Out (FIFO), Random, Least-Recently-Used (LRU), Clock, and Working-Set[4, 6, 9]. However, as they focus on adapting memory management techniques used in operating systems to database systems, they fail to take advantage of the specific access patterns exhibited by relational database queries, and their performance is not satisfactory[1].

Allocation strategies in the second group consider exclusively the demand factor, or more specifically the access patterns of queries. They include the Hot-Set model designed by Sacca and Schkolnick[8], and the strategy used by Cornell and Yu[3] in the integration of buffer management with query optimization. This approach of buffer allocation is culminated in the work of Chou and DeWitt[1]. They introduce the notion of a *locality set* of a query, i.e. the number of buffers needed by a query without causing many page faults. They propose the DBMIN algorithm that makes allocation equal to the size of the locality set. Simulation results show that DBMIN outperforms the Hot-Set strategy and the algorithms referred to in the first group.

While the strength of DBMIN and other algorithms referred to in the second group lies in their consideration of the access patterns of queries, their weakness arises from their oblivion of runtime conditions, such as the availability of buffers. This imposes heavy penalties on the performance of the whole system. For instance, the buffer utilization is low and the response time is high[7].

The above deficiencies have led us to propose *flexible* buffer allocation techniques. In [7] we design a class of allocation methods, named *MG-x-y*, which simultaneously take into account the access patterns of queries and the availability of buffers at runtime. Simulation results show that this added flexibility gives better performance than DBMIN[7].

However, all the algoritms mentioned above are static in nature, and they cannot adapt to changes in system loads and the mix of queries using the system. This weakness motivates us to investigate how to incorporate the dynamic workload factor into the buffer allocation decision. In this paper, we introduce predictive load control based on the notion of predictors that estimate the effect a buffer allocation will have on the performance of the system. These predictions not only take into account the access patterns of queries and the availability of buffers, but they also include dynamic workload information such as the characteristics of queries currently running in the system and queries being kept in the waiting queue.

3 Models for References and Flexible Buffer Allocation

In this section we first present mathematical models for relational database references. As we shall see in Section 4, these models provide formulas for predictive load control. Then we review DBMIN and outline the class of flexible buffer allocation algorithms MG-x-y which are precursors to the predictive buffer allocation strategies proposed in this paper.

3.1 Models for Relational Database References

In [1] Chou and DeWitt show how page references of relational database queries can be decomposed into sequences of simple and regular access patterns. They identify four major types of references: random, sequential, looping and hierarchical. A random reference consists of a sequence of random page accesses. A selection using a non-clustered index is one example. The following definitions formalize this type of references.

Definition 1 A *reference Ref* of length k to a relation is a sequence $< P_1, P_2, \ldots, P_k >$ of pages of the relation to be read in the given order. □

Definition 2 A *random reference* $\mathcal{R}_{k,N}$ of length k to a relation of size N is a reference $< P_1, \ldots, P_k >$ such that for all $1 \leq i, j \leq k$, P_i is uniformly distributed over the set of all pages of the accessed relation, and P_i is independent of P_j for $i \neq j$. □

In a sequential reference, such as a selection using a clustered index, pages are referenced and processed one after another without repetition.

Definition 3 A *sequential reference* $\mathcal{S}_{k,N}$ of length k to a relation of size N is a reference $< P_1, \ldots, P_k >$ such that for all $1 \leq i, j \leq k \leq N$, $P_i \neq P_j$. □

When a sequential reference is performed repeatedly, such as in a nested loop join, the reference is called a looping reference.

Definition 4 A *looping reference* $\mathcal{L}_{k,t}$ of length k is a reference $< P_1, \ldots P_k >$ such that for some $t < k$,
i) $P_i \neq P_j$, for all $1 \leq i, j \leq t$, and
ii) $P_{i+t} = P_i$, for $1 \leq i \leq k - t$
The subsequence $< P_1, \ldots, P_t >$ is called the *loop*, and t is called the *length* of the loop. □

Finally, a hierarchical reference is a sequence of page accesses that form a traversal path from the root down to the leaves of an index. In [7] we show how the analyses for random, sequential and looping references can be applied to hierarchical references and other more complex types of references. Hence, due to space limitations, here we present mathematical models only for the three types of references. In particular we give formulas for computing the expected number of page faults using a given number of buffers s.

Definition 5 Let $Ef(Ref, s)$ denote the expected number of page faults caused by the reference Ref using s buffers, where Ref can be a random, sequential or looping reference. □

Random References: For a random reference, the expected number of page faults is given by:

$$Ef(\mathcal{R}_{k,N}, s) = \sum_{f=1}^{k} f * P(f, k, s, N) \qquad (1)$$

where $P(f, k, s, N)$ denotes the probability that there are f faults in k accesses to a relation of size N using s buffers. The probability $P(f, k, s, N)$ can be computed by the recurrence equations listed in [5, 7]. Here due to space limitations, we only show the formula that gives a very close approximation to $Ef(\mathcal{R}_{k,N}, s)$:

$$Ef(\mathcal{R}_{k,N}, s) \approx \begin{cases} N * [1 - (1 - 1/N)^k], & k < k_0 \\ s + (k - k_0) * (1 - s/N), & k \geq k_0 \end{cases} \qquad (2)$$

where $k_0 = \ln(1 - s/N) \,/\, \ln(1 - 1/N)$. Note that all these formulas make the uniformity assumption whose effects are discussed in [2].

Sequential References: Trivially

$$Ef(\mathcal{S}_{k,N}, s) \;=\; k. \qquad (3)$$

Looping References: Using the Most-Recently-Used replacement policy, which is proved to be optimal in [7], we have:

$$Ef(\mathcal{L}_{k,t}, s) = t + (t - s) * t * (k/t - 1)/(t - 1), \qquad (4)$$

if $s \leq t$, and

$$Ef(\mathcal{L}_{k,t}, s) = t \qquad (5)$$

otherwise.

Next we review DBMIN and the MG-x-y policy, which generalizes and improves DBMIN by permitting flexible buffer allocations to references.

3.2 Generic Load Control and DBMIN

During load control, a buffer manager determines whether a waiting reference can be activated, and decides how many buffers to allocate to this reference. Throughout this paper, we use the term **admission policy** to refer to the first decision and the term **allocation policy** to refer to the second one. Once the admission and allocation policies are chosen, a buffer allocation algorithm can be outlined as follows.

Algorithm 1 (Generic) Whenever buffers are released by a newly completed query, use the given admission policy to determine whether the query Q at the head of the waiting queue can be activated. If this is feasible, the buffer manager decides the number of buffers s that Q should have, based on the allocation policy. After Q is activated, continue with the next query, until no more query can be activated. Notice that only Q can write on these buffers; these buffers are returned to the buffer pool after the termination of Q. □

Throughout this paper, we use A to denote the number of available buffers, and s_{min} and s_{max} to denote respectively the minimum and maximum numbers of buffers that a buffer allocation algorithm is willing to assign to a reference.

For DBMIN, the admission policy is simply to activate a query whenever the specified number of buffers are available, that is $s_{min} \leq A$. As for the allocation policy, it depends on the type of the reference. For ease of presentation, here we only consider the most basic references: looping, sequential and random. For a looping reference, DBMIN specifies that exactly t buffers have to be allocated, i.e. $s_{min} = s_{max} = t$. For a sequential and a random reference[1], DBMIN specifies $s_{min} = s_{max} = 1$. Table 2 summarizes the admission and allocation policies of DBMIN. Note that the

[1]In [1], it is proposed that a random reference may be allocated 1 or b_{yao} buffers where b_{yao} is the Yao estimate

allocation algorithms	allocation policy						admission policy
	looping		random		sequential		
	s_{min}	s_{max}	s_{min}	s_{max}	s_{min}	s_{max}	
DBMIN	t	t	1	1	1	1	$s_{min} \leq A$
MG-x-y	$x\% * t$	t	1	y	1	1	$s_{min} \leq A$
predictive methods	$f(load)$	t	$f(load)$	b_{yao}	1	1	$s_{min} \leq A$

Table 2: Characteristics of Buffer Allocation Algorithms

inflexibility of DBMIN is illustrated by the fact that the range $[s_{min}, s_{max}]$ degenerates to a point. In other words, DBMIN never tolerates sub-optimal allocations to looping references, and it never allows random references the luxury of many buffers even if these buffers are available. These problems lead us to the development of the flexible MG-x-y policy discussed next.

3.3 Flexible Buffer Allocation: MG-x-y

The MG-x-y buffer allocation policy [7] uses two parameters x and y. In particular, for a looping reference, MG-x-y uses $s_{min} = x\% * t$ and $s_{max} = t$, where x is in the range of 0% to 100%. As MG-x-y tries to allocate as many buffers as possible to a reference, MG-x-y can allocate flexibly and sub-optimally if necessary. For random references, MG-x-y uses $s_{min} = 1$ and $s_{max} = y$, where y is a value related to the Yao estimate b_{yao}. Thus, MG-x-y allows random references the luxury of many buffers, if these buffers are available. See Table 2 for a summary. Note that MG-100-1 (i.e. x=100%, y=1) is the same as DBMIN. In [7] we show that if we allow more flexible values for x and y than DBMIN, MG-x-y gives higher throughputs.

The significance of the MG-x-y policy is to show the benefits of flexible buffer allocation. However, as the x and y parameters are determined beforehand according to the mix of queries to use the system, they are generally not easy to find, and they are vulnerable to changes in the mix of queries. In the rest of this paper, we explore further the idea of flexible buffer allocation, and we propose allocation algorithms that dynamically choose the s_{min} and s_{max} values using run-time information. Thus, apart from the fact that the proposed algorithms are as flexible as MG-x-y, they have the extra advantage of not relying on any parameters to be chosen in advance, and they can adapt to changing workloads. The main idea is to use a queueing model to give predictions about the performance of the system, and to make the s_{min} and s_{max} parameters vary according to the state of the queueing model. In the next section, we describe the proposed queueing model, as well as the ways the model can be used to perform buffer allocation in a fair (FCFS), robust and adaptable way.

on the average number of pages referenced in a series of random record accesses[11]. However, it is unclear under what circumstances a random reference should be allocated b_{yao} buffers[1]. Regardless, in reality, the Yao estimates are usually too high for allocation. For example, for a blocking factor of 5, the Yao estimate of accessing 100 records of a 1000-record relation is 82 pages. Thus, DBMIN almost always allocates 1 buffer to a random reference.

4 Predictive Buffer Allocation

4.1 Predictive Load Control

As described in the previous section, both DBMIN and MG-x-y are static in nature and their admission policy is simply: $s_{min} \leq A$, where s_{min} is a pre-defined constant, for each type of reference. Here we propose predictive methods that use dynamic information, so that s_{min} is now a function of the workload, denoted by $f(load)$ in Table 2. More specifically, a waiting reference is activated with s buffers, if this admission is *predicted* to improve the performance of the current state of the system. In more precise notations, suppose Pf denotes a performance measure (e.g. throughput), $\overrightarrow{Ref_{cur}}$ represents all the references (i.e. queries) $Ref_1, \ldots, Ref_n$ currently in the system, with $\overrightarrow{s_{cur}} = s_1, \ldots, s_n$ buffers respectively, and Ref is the reference under consideration for admission. Then s_{min} is the smallest s that will improve the Pf predictor: $Pf(\overrightarrow{Ref_{new}}, \overrightarrow{s_{new}}) \geq Pf(\overrightarrow{Ref_{cur}}, \overrightarrow{s_{cur}})$, where $\overrightarrow{Ref_{new}} = \overrightarrow{Ref_{cur}} \cup Ref$, $\overrightarrow{s_{new}} = s_1, \ldots, s_n, s$, and the symbol $Pf(\vec{R}, \vec{s})$ denotes the performance of the system with $\vec{R}$ active references and $\vec{s}$ buffer allocations. Thus, the reference Ref is admitted only if it will not degrade the performance of the system[2].

In this paper we consider two performance measures or *predictors*: *throughput TP* and *effective disk utilization EDU*. Next, we analyze the above predictors and discuss the motivation behind our choices. But first we outline a queueing model that forms the basis for these predictors. At the end of this section, we discuss how these predictors can be incorporated with various allocation policies to give different predictive buffer allocation algorithms. In section 5 we present simulation results comparing the performance of these predictive algorithms with MG-x-y and DBMIN.

[2]With one only exception; see Section 4.4 for a discussion.

4.2 Queueing Model

We assume a closed queueing system with two servers: one CPU and one disk. Within the system, there are n references (jobs) $Ref_1, \ldots, Ref_n$ whose CPU and disk loads are $T_{C,i}$ and $T_{D,i}$ respectively for $i = 1, \ldots, n$. Furthermore, Ref_i has s_i buffers. Therefore, if every disk access costs t_D (e.g. 30 msec), and the processing of a page after it has been brought in core costs t_C (e.g. 2 msec), we have the following equations:

$$T_{D,i} = t_D * Ef(Ref_i, s_i) \qquad (6)$$

$$T_{C,i} = t_C * k_i \qquad (7)$$

where k_i is the number of pages accessed by Ref_i, and $Ef(Ref_i, s_i)$ can be computed using the formulas listed in Section 3.

The general solution to such a network can be calculated; see for example [10, pp. 451-452]. It involves an n-class model with each job being in a class of its own. But while it gives accurate performance measures such as throughput and utilizations, this solution is expensive to compute, since it requires exponential time on the number of classes. As ease of computation is essential in load control, we approximate it with a single-class model; all the jobs come from one class, with the overall CPU load T_C and the overall disk load T_D being the averages of the respective loads of the individual references. T_C and T_D may be the harmonic or geometric means depending on the predictors to be introduced in the following.

4.3 Predictor TP

Since our ultimate performance measure is the throughput of the system, a natural predictor is to estimate the throughput directly. In general, there are two ways to increase the throughput of a system: increase the multiprogramming level mpl, or decrease the disk load of the jobs by allocating more buffers to the jobs. However, these two requirements normally conflict with each other, as the total number of buffers in a system is fixed. Hence, for our first predictor TP, we propose the following admission policy:

Admission Policy 1 (TP) Activate the reference if optimal allocation is possible; otherwise, activate only if the reference will increase the throughput. □

To implement the above policy, we provide formulas to compute the throughput. The solution to the single class model is given in [10]:

$$TP = U_D / T_D. \qquad (8)$$

U_D is the utilization of the disk

$$U_D = \rho \frac{\rho^n - 1}{\rho^{n+1} - 1} \qquad (9)$$

where ρ is the ratio of the disk load versus the CPU load

$$\rho = T_D / T_C. \qquad (10)$$

To derive the average loads T_C and T_D, we use the harmonic means of the respective loads

$$1/T_C = 1/n * \sum_{i=1}^{n} 1/T_{C,i} \qquad (11)$$

$$1/T_D = 1/n * \sum_{i=1}^{n} 1/T_{D,i}. \qquad (12)$$

Since the equations of the queueing systems are based on the concept of "service rate" which is the inverse of the load, using the harmonic means of the loads is equivalent to using the arithmetic means of the rates. Notice that the calculation of the throughput requires $O(1)$ operations, if the buffer managers keeps track of the values T_D and T_C.

4.4 Predictor EDU

Although very intuitive, using the estimated throughput as the criterion for admission may lead to some anomalies. Consider the situation when a long sequential reference is at the head of the waiting queue, while some short, optimally allocated random references are currently running in the system. Now admitting the sequential reference may decrease the throughput, as it increases the average disk load per job. However, as the optimal allocation for the sequential reference is only one buffer, activating the sequential reference is reasonable. Exactly for this reason, Admission Policy 1 is "patched up" to admit a reference with s_{max} buffers, even if this admission decreases the throughput.

This anomaly of the predictor TP leads us to the development of our second predictor – **Effective Disk Utilization** (EDU). Consider the following point of view of the problem: There is a queue of jobs (i.e. references), a system with one CPU and one disk, and a buffer pool that can help decrease the page faults of the jobs. Assuming that the disk is the bottleneck (which is the case in all our experiments, and is usually the case in practice), a reasonable objective is to make the disk work as efficiently as possible. In general, there are two sources of inefficient uses of the disk: (1) the disk is sitting idle because there are very few jobs, or (2) the disk is working on page requests that could have been avoided if enough buffers had been given to the references causing the page faults. The following concept captures these observations.

Definition 6 The *effective disk utilization EDU* is the portion of time that the disk is engaged in page faults that could not be avoided even if the references are each assigned its optimum number of buffers (infinite, or, equivalently s_{opt} which is the maximum number of buffers usable by a reference). □

Hence, for our second predictor EDU, we use the following admission policy (Ref is the reference at the head of the waiting queue):

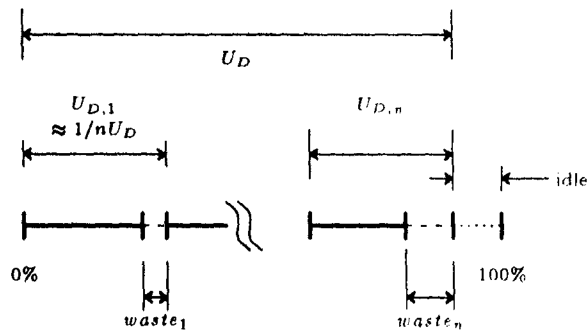

Figure 2: Effective disk utilization

Admission Policy 2 (EDU) Activate Ref if it will increase the effective disk utilization. □

Mathematically, the effective disk utilization is expressed by:

$$EDU = (\sum_{i=1}^{n} U_{D,i}) - (\sum_{i=1}^{n} U_{D,i} * w_i) \quad (13)$$

where $U_{D,i}$ represents the disk utilization due to Ref_i and w_i is the portion of "avoidable" (or "wasted") page faults caused by Ref_i:

$$w_i = \frac{Ef(Ref_i, s_i) - Ef(Ref_i, \infty)}{Ef(Ref_i, s_i)}. \quad (14)$$

For practical calculations, we use s_{opt} instead of ∞; clearly, s_{opt} is 1, t and b_{yao} for sequential, looping and random references respectively. Informally, Equation (13) specifies that at every unit time, the disk serves Ref_i for $U_{D,i}$ units of time. Out of that, Ref_i wastes $w_i * U_{D,i}$ units of time. Summing over all jobs, we get Equation (13).

Figure 2 illustrates the concept of effective disk utilization. The horizontal line corresponds to a 100% disk utilization; the dotted portion stands for the idle time of the disk, the dashed parts correspond to the "wasted" disk accesses and the sum of the solid parts corresponds to the effective disk utilization.

Note that, for I/O bound jobs, every job has approximately an equal share of the total disk utilization U_D, even though the jobs may have *different* disk loads. Thus, we have the following formula:

$$U_{D,i} = U_D/n, \quad (15)$$

which simplifies Equation (13) to:

$$EDU = U_D - U_D/n * (\sum_{i=1}^{n} w_i). \quad (16)$$

Notice that we have not yet used a single-class approximation. We only need this approximation to calculate the disk utilization U_D. Using the exact n-class model [10], we find out that the *geometric* averages seem to give a better approximation to the the disk utilization. Thus, the average CPU and disk loads are given by:

$$\mathcal{T}_C = \sqrt[n]{\prod_{i=1}^{n} \mathcal{T}_{C,i}} \quad (17)$$

$$\mathcal{T}_D = \sqrt[n]{\prod_{i=1}^{n} \mathcal{T}_{D,i}}. \quad (18)$$

Based on these equations, the disk utilization U_D can be computed according to Equations (9) and (10). As for the TP predictor, the calculation of EDU requires $O(1)$ steps, if the buffer manager keeps track of the loads $\mathcal{T}_C$, $\mathcal{T}_D$ and of the total "wasted" disk accesses $\sum_{i=1}^{n} w_i$.

4.5 Predictive Buffer Allocation Algorithms

Thus far we have introduced two predictors: TP and EDU. We have presented the admission policies based on these predictors and provided formulas for computing these predictions. To complete the design of predictive buffer allocation algorithms, we propose three allocation policies, which are rules to determine the number of buffers s to allocate to a reference, once the reference has passed the admission criterion.

Allocation Policy 1 (Optimistic) Give as many buffers as possible, i.e. s=min(A, s_{max}). □

Allocation Policy 2 (Pessimistic) Allocate as few buffers as necessary to random references (i.e. s_{min}), but as many as possible to sequential and looping references. □

The optimistic policy tends to give allocations as close to optimal as possible. However, it may allocate too many buffers to random references, even though these extra buffers may otherwise be useful for other references in the waiting queue. The pessimistic policy is thus designed to deal with this problem. But a weakness of this policy is that it unfairly penalizes random references. In particular, if there are abundant buffers available, there is no reason to let the buffers sit idle and not to allocate these buffers to the random references.

Allocation Policy 3 (2-Pass) Assign tentatively buffers to the first m references in the waiting queue, following the pessimistic policy. Eventually, either the end of the waiting queue is reached, or the m+1 - th reference in the waiting queue cannot be admitted. Then perform a second pass and distribute the remaining buffers equally to the random references that have been admitted during the first pass. □

In essence, when the 2-Pass policy makes allocation decisions, not only does it consider the reference at

query type	query operators	selec-tivity	access path of selection	join method	access path of join	reference type
I	select(A)	10 %	clustered index	–	–	$\mathcal{S}_{50,500}$
II	select(B)	10 %	non-clustered index		–	$\mathcal{R}_{30,15}$
III	select(C)	1 %	non-clustered index		–	$\mathcal{R}_{30,150}$
IV	select(A) ⋈ B	1 %	sequential scan	index join	non-clustered index on B	$\mathcal{R}_{100,15}$
V	select(B) ⋈ C	10 %	sequential scan	index join	non-clustered index on B	$\mathcal{R}_{30,150}$
VI	select(A) ⋈ B	4 %	clustered index	nested loop	sequential scan on B	$\mathcal{L}_{300,15}$

Table 3: Summary of Query Types

relation A	10,000 tuples
relation B	300 tuples
relation C	3,000 tuples
tuple size	182 bytes
page size	4K

Table 4: Details of Relations

	I	II	III	IV	V	VI
mix 1	10%	10%	–	10%	–	70%
mix 2	10%	45%	–	45%	–	–
mix 3	10%	30%	–	–	30%	30%
mix 4	–	–	50%	–	50%	–

Table 5: Summary of Query Mixes

the head of the waiting queue, but it also takes into account as many references as possible in the rest of the queue.

The three allocation policies can be used in conjunction with both TP and EDU, giving rise to six potential predictive buffer allocation algorithms. As a naming convention, each algorithm is denoted by the pair "*predictor-allocation*" where "*predictor*" is either TP or EDU, and "*allocation*" is one of: o, p, 2, representing optimistic, pessimistic and 2-Pass allocation policies respectively. For instance, EDU-o stands for the algorithm adopting the EDU admission policy and the optimistic allocation policy. Note that all the above algorithms follow the generic description of Algorithm 1. And once the number of buffers s allocated to a reference has been decided, all buffer allocation algorithms operate in exactly the same way as DBMIN, that is, those s buffers are exclusively used by the reference, until it terminates.

5 Simulation Results

In this section we present simulation results evaluating the performance of predictive allocation algorithms in a multiuser environment. As Chou and DeWitt have shown in [1] that DBMIN performs better than the Hot-Set algorithm, First-In-First-Out, Clock, Least-Recently-Used and Working-Set, we only compare the

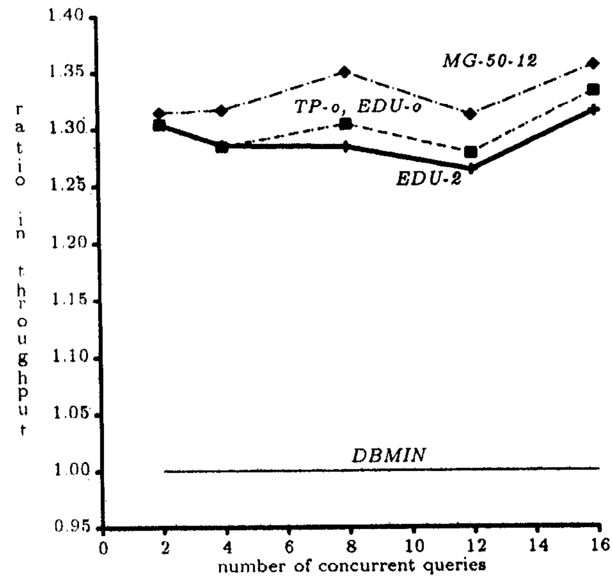

Figure 3: Mix 1 – no Data Sharing

predictive algorithms with MG-x-y and DBMIN. Furthermore to highlight the comparisons, we only present simulation results for TP-o, EDU-o, EDU-2, and the best set of parameters for MG-x-y.

5.1 Details of Simulation

In order to make direct comparison with DBMIN, we use the simulation program Chou and DeWitt used for DBMIN, and we experiment with the same types of queries. Table 3 summarizes the details of the queries that are chosen to represent varying degrees of demand on CPU, disk and memory [1]. Table 4 and Table 5 show respectively the details of the relations and the query mixes we used. In the simulation, the number of concurrent queries varies from 2 to 16 or 24. Each of these concurrent queries is generated by a query source which cannot generate a new query until the last query from the same source is completed. Thus, the simulation program simulates a closed system[3]. Further-

[3]Besides buffer management, concurrency control and transaction management is another important factor affecting the performance of the whole database system. While the simulation package does not consider transaction man-

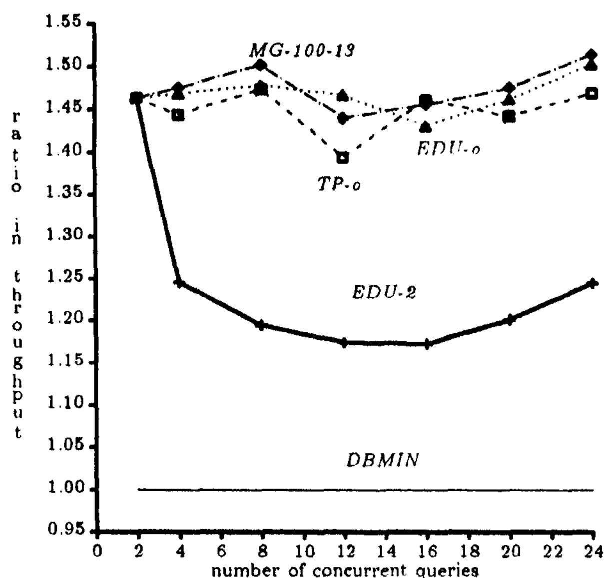

Figure 4: Mix 2 - no Data Sharing

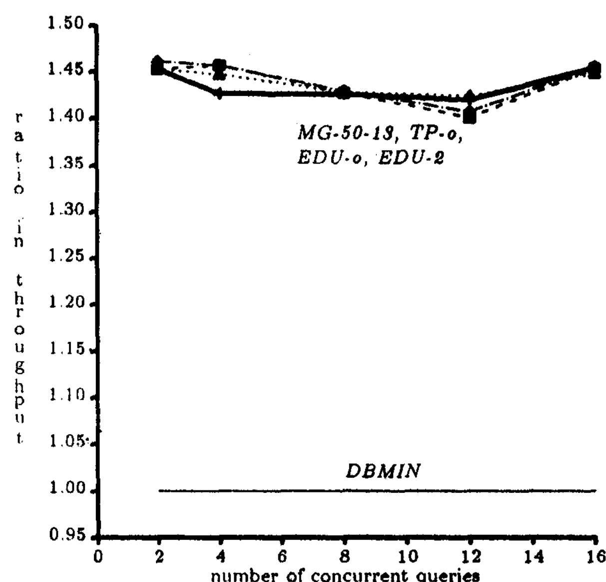

Figure 5: Mix 1 - Full Data Sharing

more, to highlight the increase or decrease in throughput relative to DBMIN, the throughput values shown in the following figures are normalized by the values of DBMIN, effective showing the ratio in throughput.

5.2 Allocations to Looping References

The first mix of queries consisted of 10% each of queries of type I, II and IV (sequential, random and random, respectively), and 70% of queries of type VI (looping references). The purpose of this mix is to evaluate the effectiveness of predictive allocations to looping references, i.e. query type VI. Figure 3 shows the relative throughput using a total number of 35 buffers.

When compared with DBMIN, all our algorithms perform better. Because of the low percentages of sequential and random references in this mix of queries, the improvement shown by our algorithms can be attributed to effective sub-optimal allocations to looping references. Furthermore, TP-o, EDU-o and EDU-2 all perform comparable to the best MG-x-y scheme for this mix of queries.

5.3 Allocations to Random References

The second mix of queries consisted of 45% each of queries of type II and IV (random references), and 10% of queries of type I (sequential references). This mix of queries is set up to evaluate the performance of allocations to random references. Figure 4 shows the relative throughput using again a total number of 35 buffers.

Again when compared with DBMIN, all our algorithms give better performance. This demonstrates the effectiveness of flexible buffer allocations to random references. In particular, when buffers are available, random references are benefited by allocations of extra buffers. Though better than DBMIN, EDU-2 does not perform as well as our other algorithms. This is because every time during the first pass of allocations (cf. Allocation Policy 3), EDU-2 has the tendency of activating many random references. As a result, the number of buffers per random reference allocated by EDU-2 is lower than that allocated by our other algorithms, thereby causing more page faults and degrading overall performance.

5.4 Effect of Data Sharing

In the simulations carried out so far, every query can only access data in its own buffers. To examine the effect of data sharing on the performance of our algorithms relative to DBMIN, we also run simulations with varying degrees of data sharing. Figure 5 shows the relative throughput of our algorithms running the first mix of queries with 35 buffers, when each query has read access to the buffers of all the other queries, i.e. full data sharing.

Compared with Figure 3 for the case of no data sharing, Figure 5 indicates that data sharing favors our algorithms. The same phenomenon occurs for other query mixes we have used. In fact, this phenomenon is not surprising because it is obvious that with data sharing, the higher the buffer utilization, the higher

agement, see [1] for a discussion on how the transaction and lock manager can be integrated with a buffer manager using DBMIN. Since our algorithms differ from DBMIN only in load control, the integration proposed there also applies to a buffer manager using our algorithms.

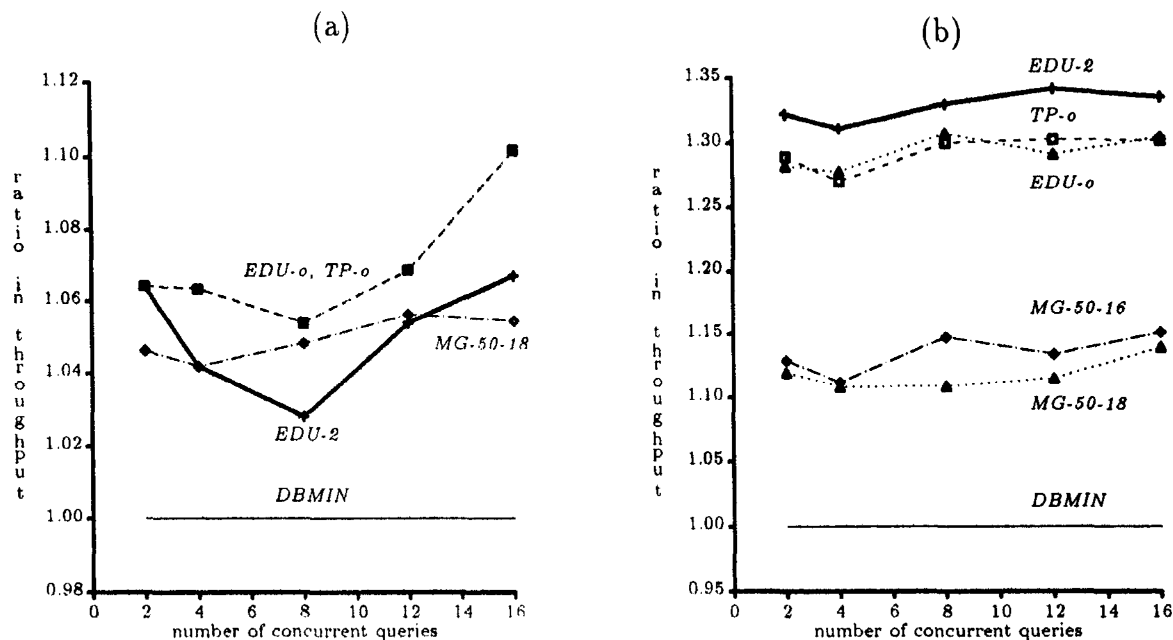

Figure 6: Adaptability: (a) Stage 1 – Mix 4, (b) Stage 2 – Mix 3

the throughput is likely to be. In other words, the inflexibility of DBMIN in buffer allocation becomes even more costly than in the case of no data sharing.

5.5 Comparisons with MG-x-y – Adaptability

Among all the simulations we have shown thus far, the predictive allocation algorithms TP-o, EDU-o and EDU-2 perform approximately as well as the best of MG-x-y. The reason is that we have a fixed mix of queries, with few types of queries, and we have selected carefully the x and y parameters that are best suited for this specific mix. The weaknesses of the MG-x-y policy stem exactly from the rigidity of the x and y parameters. The simulations described below demonstrate the weaknesses of MG-x-y and the effectiveness of the predictive algorithms.

The first weakness of MG-x-y lies in the fact that each MG-x-y scheme has only *one* x and *one* y value for *all* kinds of looping and random references. But now consider Query Type II and V respectively: Query Type II ($\mathcal{R}_{30,15}$) has a low Yao estimate of 13 and a high selectivity of making 30 random accesses on 15 pages; Query Type V ($\mathcal{R}_{30,150}$) has a high Yao estimate of 27 and a low selectivity of making 30 random accesses on 150 pages. For a query of the first type, it is beneficial to allocate as close to the Yao estimate as possible. But for a query of the second type, it is not worthwhile to allocate many buffers to the query. Thus, for any MG-x-y scheme, using one y value is not sufficient to handle the diversity of queries. This problem is demonstrated by running a simulation on the third query mix which consists of the two kinds of random references mentioned above (Query Type II and query Type V). Figure 6(b) shows the relative throughput of running this mix of queries with 30 buffers. When compared with the best result of MG-x-y (i.e. MG-50-16 in this case), the predictive algorithms perform better, handling the diversity of queries more effectively.

The second major weakness of MG-x-y is its inability of adjusting to changes in query mixes. Figure 6 shows the result of running a simulation that consists of two stages. In the first stage, the query mix consists of random references only (i.e. mix 4). As shown in Figure 6(a), the best result of MG-x-y (i.e. MG-50-18 in this case) performs comparably to the predictive algorithms. But when the second stage comes and the query mix changes from mix 4 to mix 3, MG-50-18 cannot adapt to the changes, as illustrated by Figure 6(b). On the other hand, the predictive algorithms adjust appropriately.

5.6 Summary

Our simulation results indicate that predictive allocation strategies are more effective and more flexible than DBMIN, with or without data sharing. They are capable of making allocation decisions based on the characteristics of queries, the runtime availability of buffers, and the dynamic workload. When compared with the MG-x-y algorithms, the predictive methods are more adaptable to changes, while behaving as flexible as the MG-x-y schemes. For simple query mixes, they typically perform as well as the *best* MG-x-y scheme; for complicated or changing query mixes, they clearly perform better.

As for the two predictors TP and EDU, both of them perform quite well. While EDU is probably more accurate for a single disk system, TP is more extendible to multi-disk systems, and is slightly easier to compute. Considering allocation policies, the winners are the 2-Pass approach and the optimistic one. The pessimistic approach gives results only slightly better than DBMIN in most cases. The 2-Pass approach performs well in most situations, with the exception of heavy workloads consisting primarily of random references. In this case, the 2-Pass policy degenerates to the pessimistic one, because there is no left-over buffers to allow for a second pass. A practical disadvantage of the 2-Pass policy is that it cannot activate queries instantaneously because queries admitted in the first pass may have to wait for the second pass for additional buffers. Thus, it is slower than the policies that only require one pass. Finally, the optimistic allocation policy performs very well in most situations. In addition, it is easy to implement and is capable of making instantaneous decisions.

6 Conclusions

The main idea in this paper is to incorporate runtime information in the buffer allocation algorithm. We propose using a simple, but accurate single-class queueing model to predict the impact of each buffer allocation decision. The proposed approach builds on the concept of flexibility, that was introduced by the MG-x-y policy [7], and it improves the MG-x-y policy further: it treats the x and y values as variables, which are determined by the load of the system so as to maximize the throughput or disk utilization.

The strong points of the proposed methods are as follows.

- They require no parameter values to be determined statically. They are adaptable, performing well under any mix of queries, as well as when the workload characteristics change over time.
- The methods are fair (FCFS) and fast, requiring $O(1)$ calculations for each performance prediction.
- Simulation results show that they perform at least as well, and usually much better than existent methods.
- Finally, the two performance measures (TP and EDU) perform equally well, and both the optimisitic and the 2-pass allocation policies are effective.

Future work in adaptive buffer management includes the extension of our predictors for systems with many disks, and the analytical study of the case where data sharing is allowed.

Acknowledgements

We thank H. Chou and D. DeWitt for allowing us to use their simulation program for DBMIN so that direct comparison can be made. This research was partially sponsored by the National Science Foundation under Grants DCR-86-16833, IRI-8719458, IRI-8958546 and IRI-9057573, by DEC and Bellcore, by the Air Force Office for Scientific Research under Grant AFOSR-89-0303, and by University of Maryland Institute for Advanced Computer Studies.

References

[1] H. Chou and D. DeWitt. *An Evaluation of Buffer Management Strategies for Relational Database Systems*, VLDB 1985.

[2] S. Christodoulakis. *Implication of Certain Assumptions in Data Base Performance Evaluation*, TODS (9) 2 (1984).

[3] D. Cornell and P. Yu. *Integration of Buffer Management and Query Optimization in Relational Database Environment*, VLDB 1989.

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

[5] C. Faloutsos, R. Ng and T. Sellis. *Predictive Load Control for Flexible Buffer Allocation*, Technical Report CS-TR-2622, University of Maryland, College Park (1991).

[6] T. Lang, C. Wood and E. Fernandez. *Database Buffer Paging in Virtual Storage Systems*, TODS (2) 4 (1977).

[7] R. Ng, C. Faloutsos and T. Sellis. *Flexible Buffer Allocation based on Marginal Gains*, SIGMOD 1991.

[8] G. Sacca and M. Schkolnick. *A Mechanism for Managing the Buffer Pool in a Relational Database System using the Hot Set Model*, VLDB 1982.

[9] S. Sherman and R. Brice. *Performance of a Database Manager in a Virtual Memory System*, TODS (1) 4 (1976).

[10] K.S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, Prentice Hall, Inc., Englewood Cliffs, NJ (1982).

[11] S. Yao. *Approximating Block Accesses in Database Organizations*, CACM (20) 4 (1977).