BUFFER MANAGEMENT OF DATABASE SYSTEMS

by

Hong-Tai Chou

Computer Sciences Technical Report #597

May 1985

BUFFER MANAGEMENT
OF
DATABASE SYSTEMS

by

HONG-TAI CHOU

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1985

ABSTRACT

The performance of a database system is strongly influenced by its buffer management. Common tactics for memory management in virtual memory systems, such as the LRU replacement strategy, are known to be less suitable in a database environment.

In this dissertation, we present a new query behavior model, the query locality set model (QLSM), for database systems. Like the hot set model, the QLSM has an advantage over the stochastic models due to its ability to predict future reference behavior. However, the QLSM avoids the potential problems of the hot set model by separating the modeling of reference behavior from any particular buffer management algorithm.

Based on the query locality set model, we propose a new algorithm, termed the Query Locality Set (QLS) algorithm, for database buffer management. Using the file instance as the basic unit for buffer allocation and management, the QLS algorithm implements a memory policy that is tailored to the individual needs of the queries. By operating on a per-file basis, it adapts better than the hot set algorithm to a query's buffer needs, making more efficient use of the available buffers.

To evaluate buffer management algorithms in a multi-user environment, we use a performance evaluation methodology that employs a hybrid model which combines features of both trace-driven and distribution-driven simulation models. Using this model for simulation experiments, we compare the performance of six buffer management algorithms: RAND, FIFO, CLOCK, WS (Working Set algorithm), HOT (Hot Set algorithm), and QLS. The simulation results indicate that the latter two algorithms, HOT and QLS, provide significantly better performance than the other algorithms. And in comparison, QLS provided about 10% more throughput than HOT for the tests conducted.

ACKNOWLEDGEMENTS

I am deeply indebted to my advisor, Professor David DeWitt, who provided generous support, excellent guidance, continued encouragement, and a superb working environment. I would also like to thank Professors Mike Carey, Livny Miron, and Mary Vernon for their helpful comments and suggestions.

The query tracing and simulation experiments might never have come to pass without the Crystal multicomputer environment. I would like to thank the primary investigators and other members of the Crystal project. Tad Lebeck, Nancy Hall, and Tom Virgilio provided patient and valuable support.

Finally, my special thanks to my wife Hai-Hsien, who provided not only unremitting patience and understanding but also valuable technical support and suggestions during these past five years.

This research was partially supported by the Department of Energy under contract #DE-AC02-81ER10920 and the National Science Foundation under grant MCS82-01870.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. The Problem - Management of Buffers in Database Systems

In a memory hierarchy, buffering of data between different levels is one of the most important factors in determining the performance of the entire system. Likewise, the performance of a database system is strongly influenced by its buffer management. Common tactics for memory management in virtual memory systems, such as the LRU replacement strategy, are less suitable in database systems [Ston81]. The need for a better buffer management algorithm, coupled with the encouraging results of recent studies in this area, have motivated our study on buffer management for database systems.

The idea of using a large backing store to complement the limited capacity of primary memory is certainly not new. In the classical paper by von Neumann et al. [Burk63], a large automatic "subsidiary memory" was suggested for this purpose. The first realization of this idea appeared in the demand paging supervisor of the ATLAS computer [Foth61] [Kilb62] in the late 50's. Despite earlier pessimism [Fine66] [Coff68], demand-paged virtual memory systems have been successfully implemented on several pioneer computers, including the Ferranti ATLAS [Foth61] [Kilb62], the IBM M44/44X [Braw68], and MULTICS [Dale68] [Bens72].

The success of dynamic storage systems has brought forth a wide research interest in the area of virtual memory management for the past quarter century. Various subjects related to virtual memory systems have been extensively studied, both analytically and

empirically. These studies have contributed to the maturity of a number of areas, including program behavior modeling, page replacement algorithms, memory partitioning, and load control.

Compared to its counterpart in virtual memory systems, buffer management in database systems[1] has not received a great deal of attention. Many existing database systems either use a buffer management algorithm designed for virtual memory systems, e.g. System R [Astr76], or simply rely on the buffer management of the underlying operating system, e.g. INGRES [Ston76]. However, conventional virtual memory policies, such as the LRU algorithm, were found to be generally unsuitable for a relational database environment [Ston81] [Sacc82]. Not only are the reference patterns of database queries different from those of programs, but also the regularity of the query reference behavior enables the prediction of future references. Equipped with this advance knowledge on reference patterns, a database buffer management algorithm has an advantage over conventional virtual memory algorithms that are based on stochastic reference models.

The hot set model [Sacc82], proposed by Sacco and Schkolnick, is a notable example of a query behavior model that departs from conventional Markov chain program models. By using advance knowledge on a query's reference behavior, the hot set model is able to predict the memory requirement of a query under the LRU algorithm. The availability of this information greatly simplifies the problems of memory partitioning and load control since we can partition the memory according to the individual needs of queries and regulate the load so that the aggregate memory requirement of active queries does not exceed the available memory in the system.

---

[1] In database systems, the focus of buffer management is on the buffering of data pages; while buffer management in virtual memory systems deals with both instruction (code) and data pages.

Although the predictive power of the hot set model has been demonstrated in [Sacc82], extensive testing of the model and its related buffer management algorithm are required to evaluate the hot set model. As studies indicate [Kapl80], the LRU algorithm is not a good page replacement strategy for certain reference patterns of database queries. Thus, the close interaction between the hot set model and the LRU algorithm leaves room for further improvement.

In this dissertation we first review the important results from the studies of virtual memory management and database buffer management. We then present a new query behavior model, the query locality set model (QLSM). Like the hot set model, the QLSM has an advantage over the stochastic models due to its ability to predict future reference behavior. However, the QLSM avoids the potential problems of the hot set model by separating the modeling of reference behavior from any particular buffer management algorithm. Based on this model of query behavior, we propose a new buffer management algorithm, termed the Query Locality Set (QLS) algorithm, for database systems. Using the file instance as the basic unit for buffer allocation and management, the QLS algorithm implements a buffer management policy that is tailored to the individual needs of the queries.

To evaluate buffer management algorithms in a multi-user environment, we use a performance evaluation methodology based on a hybrid model that combines features of both trace-driven and distribution-driven simulation models. Using this model for simulation experiments, we compare the performance of six buffer management algorithms: RAND, FIFO, CLOCK, WS (Working Set algorithm), HOT (Hot Set algorithm), and QLS.

1.2. Organization of the Dissertation

The rest of the dissertation is organized as follows:

Chapter 2 reviews some important results on modeling reference behavior of programs and database systems. A new reference behavior model, the query locality set model (QLSM), for relational database systems is also presented.

In Chapter 3, we review some important results from the studies of main memory management, focusing on three important issues: load control, memory partitioning and page replacement. Several algorithms for database buffer management are also discussed. In the latter part of the chapter, we present a new buffer management algorithm, the Query Locality Set (QLS) algorithm, for database systems.

Chapter 4 presents a performance evaluation methodology for evaluating buffer management algorithms in a multi-user environment. Using this model for simulation experiments, the performance of six buffer management algorithms are evaluated.

Chapter 5 investigates issues related to the integration of a QLS-based buffer manager and two other major components of a database system: the query optimizer and the transaction manager. Finally, the conclusions of our study and some suggestions for future research are presented in Chapter 6.

CHAPTER 2

MODELS OF REFERENCE BEHAVIOR

A program behavior model provides a basis for determining a program's working information at a given time and predicting what it will be at a future time. It helps us understand the dynamic behavior of programs and is useful in both designing and evaluating policies for virtual memory systems. Similarly, a query behavior model for database systems is useful in designing and evaluating buffer management policies. In this chapter, we shall review some important results on modeling behavior of programs and database systems. After surveying various behavior models in the literature, a new query behavior model for database systems, called the Query Locality Set Model (QLSM), will be proposed and discussed in the last part of this chapter.

2.1. Models of Program Behavior

The concept of **Working Set**, introduced by Denning [Denn68a], plays an important role in the study of memory management. Intuitively, the working set[1] of a process is the minimum subset of its program pages that must reside in main memory in order for the process to execute efficiently. It is observed that many programs, to varying degrees, obey the **principle of locality** [Denn70], which states:

L1. During any interval of time, a program references non-uniformly over its pages,

[1] To be more precise, the set of pages favored by a process at a given time (during its execution) is called its **locality set** [Denn80]. A **working set** is an observed estimate of the locality set.

some pages being favored over others.

L.2. The frequency with which a given page is referenced tends to change slowly in time.

L.3. References in the immediate past and the immediate future tends to be highly correlated, whereas the correlation between references tends to become smaller as the distance between them increases.

The principle of locality provides a basis for predicting a program's memory demand of the immediate future from that of the immediate past. Following this principle, Denning [Denn68a] defined a process's working set $\mathbf{W(t,\tau)}$ at time t to be the set of pages referenced by the process during the process time interval $[t-\tau,t]$. The parameter τ is called the **window size**.

Based on the definition of working set $\mathbf{W(t,\tau)}$, an analysis for program behavior in terms of working set properties was presented in [Denn72a]. Consider a program with n pages, and let $\mathbf{N}=\{1,2,..,n\}$ be the set of all pages of the program. The dynamic behavior of the program for a given input can be modeled by its **reference string**, which is a sequence $r_1 r_2 \cdots r_t \cdots$, $r_t \in N$. The meaning of $r_t=i$ is that page i is referenced at time t; thus t measures the process's execution time or virtual time. If r_j and r_{j+x} in the reference string $r_1 r_2 \cdots r_j \cdots r_{j+x} \cdots$ are two successive references to page i, x is called an **interreference interval** for page i. Under the assumptions that reference substrings are stationary and references are asymptotically uncorrelated[2], equations that established the

[2] A reference substring is stationary if references within the substring satisfy the condition:
$\mathbf{Pr}[r_{t+x}=j \mid r_t=i]=\mathbf{Pr}[r_{t+z+x}=j \mid r_{t+z}=i]$, for any $z>0$, $i,j \in N$.
References are asymptotically uncorrelated if for all $t>0$, r_t and r_{t+x} become uncorrelated as $x \to \infty$.

relationships between the mean working set size $s(\tau)$, the missing-page rate $\mathbf{m}(\tau)$ and the interreference density function $\mathbf{f}(\tau)$ were derived:

$$s(\tau+1)-s(\tau)=\mathbf{m}(\tau).$$

$$\mathbf{m}(\tau+1)-\mathbf{m}(\tau)=-\mathbf{f}(\tau+1).$$

$$s(\tau)=\sum_{z=0}^{\tau-1}\mathbf{m}(z)=\sum_{z=0}^{\tau-1}\sum_{y>z}\mathbf{f}(y).$$

Informally, the interreference-interval density function is the negative slope of the missing-page rate function which, in turn, is the slope of the mean working set size function.

The derivation above provides a unified view towards the formalization of program behavior. On the other hand, it also suggests a hierarchy for classifying program behavior models (Figure 2.1). There are four levels in the hierarchy: mean working set size, missing-page rate, interreference interval, and page reference strings; each provides a finer characterization of program behavior than its predecessor.

A working set size function describes a program's dynamic memory requirement in terms of the number of page frames required to hold the working set of the program. This class of models is useful for studying memory partitioning policies in multiprogrammed

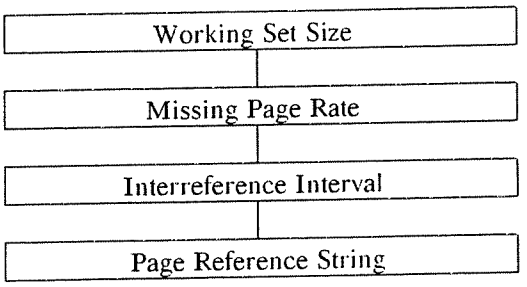

Figure 2.1 Hierarchy of Program Behavior Models

systems where detailed paging behavior is not required. Coffman and Ryan [Coff72], for example, demonstrated how dynamic (as oppose to static) memory partitioning can improve memory utilization by using a locality model, in which a program's working set size is modeled as a stationary Gaussian (normal) process.

Missing-page rate is another manifestation of program behavior and has been the subject of many studies [Bela69] [Cham73] [Denn73] [Ghan75b] [East77] [East78]. **Lifetime function L(x)** [Bela69], which gives the mean execution interval (in the process's virtual time) between page exceptions under memory constraint **x**, is a program behavior model based on the missing page rate. Two empirical lifetime functions, Belady's formula [Bela69] and Chamberlin's formula [Cham73], have been proposed and used in several studies [Bela69] [Cham73] [Aror73] [Ghan75b]. Lewis and Shedler [Lewi73] have also derived a semi-Markov model for sequences of page exceptions. Like the working set size models, models in this class are useful for studying issues of multiprogrammed systems, including memory partitioning and control of multiprogramming level.

The models discussed as far are useful for studying memory contention among concurrent processes in a multiprogrammed environment. However, they do not contain enough information on the paging behavior of an individual process, and hence are inadequate for studying such issue as page replacement algorithms. One solution is to formulate page references as an arrival process[3], and impose a stochastic structure on the intervals between page references. The renewal model [Coff73] [Opde75], in which references to each page constitutes a renewal process, is one such model, and has been used to analyze the performance of the working set replacement algorithm.

---

[3] The term "process" here refers to a sequence of random events, not the execution of a program.

The last class in the hierarchy of program models is the page reference model. A page reference model describes the details of a program behavior at the page level, and uses a stochastic process to characterize the page reference activities of the program. Due to their wide use in the study of memory management, we shall examine individually some important page reference models for program behavior.

2.1.1. Independent Reference Model (IRM)

The independent reference model (IRM) [Aho71] treats a reference string $r_1 r_2 \cdots r_t \cdots$ as a sequence of independent random variables with identical distributions. Under this model, a fixed reference probability is associated with each page in a program. That is, $\mathbf{Pr}[r_t = i] = c_i$ for **all** t. A special case of the IRM is called the random reference model (RRM) [Opde75], in which the reference probability of all pages are the same (i.e. $\mathbf{Pr}[r_t = i] = C$ for **all** $i \in N$). Due to its simplicity, IRM has been extensively analyzed [Aho71] [King71] [Gele73] [Coff73] [Smit76b]. However, since it fails to capture the temporal locality of programs, IRM has also been shown to be a poor model for program behavior [Spir72] [Arvi73] [Lenf75].

2.1.2. Locality Model (LM)

The locality model [Denn72b] tries to capture program locality explicitly through the concept of **locality sets** $\{L_i\}$, each representing a set of favored pages. Under this model, the dynamic behavior of a program is described by a sequence of pairs $(L_1, t_1), (L_2, t_2), \cdots, (L_i, t_i), \cdots$, where L_i is the ith locality set and t_i is the holding time in L_i. A stochastic structure can be imposed by a transition matrix $[\mathbf{Pr}(L_i, L_j)]$ among locality sets and a set of holding time distributions $H_i(t)$ for each L_i. An early form of the locality

model was presented by Shemer and Shippey [Shem66] in which "paging unit" and "paging unit transition matrix" were used to describe locality sets and the transition of localities, respectively. The so-called very simple locality model (VSLM) [Spir72] is another special case of the LM in which a fixed size locality and a geometric holding time distribution are assumed[4]. Although experimental results show that the VSLM is a better model than the IRM, the simple LRU model, which will be discussed next, is superior to both.

2.1.3. LRU Stack Model (LRUSM)

The LRU stack model (LRUSM) originated from the LRU replacement algorithm [Matt70]. Central to the model is the **LRU stack**, which is a dynamic list that arranges the referenced pages from top to bottom by decreasing recency of reference. The position at which a referenced page was found in the stack before being promoted to the top is called its **stack distance**. Another interpretation of the stack distance is the number of distinct pages referenced since the page was last referenced. The LRU stack model describes the dynamic behavior of a program by a distance string, i.e. sequence of stack distances, $\mathbf{d}_1\mathbf{d}_2 \cdots \mathbf{d}_t \cdots$ generated by some stochastic process.

Shedler and Tung [Shed72] proposed a first order Markov process for generating the distance string. There are **N** states in the model, among which the first **f** states are favored. Their model is relatively complex and has not been empirically validated.

In an attempt to improve the LRU stack model, Arvind et. al. [Arvi73] used separate stacks for instruction and data spaces, and defined a random process for switching between stacks. Despite the added complexity, this approach did not result in a better model.

---

[4] That is, $|\mathbf{L}_i| = \mathbf{l}$ and $\mathbf{H}_i(t) = (1-\lambda)^{t-1}\lambda$ for all **i**, where **l** and λ are two parameters of the model.

The most popular form of LRUSM is the simple LRU stack model (SLRUM), which has been subjected to extensive study [Coff72] [Denn72b] [Oden72] [Spir72] [Coff73]. Under this model, a fixed probability is associated with each stack distance, i.e. $\mathbf{Pr[d_t = i] = b_i}$ for **all t**. SLRUM provides a good model for predicting references coming from a locality in a program. However, it fails to describe the disruptive behavior during transitions of localities, and thus inaccuracy of the model has been observed [Arvi73] [Lenf75].

2.1.4. Phase/Transition Model

An approach that explicitly models the transitional behavior of programs is based on the **principle of decomposability** [Cour75]. This approach, known as the **phase/transition model** [Denn75b], decomposes a reference string into a sequence of intervals called **phases**, during which the program is assumed to be in equilibrium. The phase holding times and associated locality sets are described by a macromodel, while the reference pattern within each phase is described by a micromodel. The phase/transition model is very similar to the locality model except that the phase/transition model was derived with a better theoretical background and places more emphasis (at least conceptually) on the disruptive nature of the transitional behavior. Denning and Kahn [Denn75b] showed that even simple forms of the phase/transition model are able to reproduce observed program behavior, which the IRM and the LRUSM fail to model. The study of program localities by Madison and Batson [Madi76] reinforces the concept of phases and transitions.

2.2. Models of Query Behavior for Database Systems

After looking at various program behavior models, we now examine their counterpart in database systems. Basically, the query behavior models to be discussed in this section are extensions to the program behavior models.

2.2.1. A Working Set Model for Database Systems

In establishing a framework for evaluating database systems, Rodriguez-Rosell and Hildebrand [Rodr75] extended the working set model to hierarchical databases. In their approach, a database system is viewed as consisting of three levels, the logical level, the encoding (access structure) level, and the physical block level. A working set definition for each of the three levels was given in terms of target entities, path entities and data blocks, respectively. The study also reported some working set statistics obtained by applying the extended working set definitions to trace data gathered from IMS, IBM's Information Management System. In a later study, Rodriguez [Rodr76] reported more experimental results from the IMS trace data. An important conclusion was that database reference strings have strong sequentiality and weak locality.

One major contribution of the database working set model is that it relates database reference behavior to program behavior in multiprogrammed systems, thus allowing the tools and results in that area to be applied to the study of database systems. However, the model is heavily biased toward hierarchical databases. It is not clear how this approach can be extended to other higher-level data models.

2.2.2. Easton's Model

In a study to model the reference strings generated by an interactive database system AAS[5], Easton [East75] proposed a Markov chain model for database reference behavior. The model has **n** states, one for each page in the database, and the transition probabilities

$$\mathbf{P_{ii}} = \mathbf{r} + (1-\mathbf{r})\lambda_i,$$

$$\mathbf{P_{ij}} = (1-\mathbf{r})\lambda_j, \ \mathbf{i} \neq \mathbf{j}$$

where $0 \leq \mathbf{r} < 1$, $\sum_{i=1}^{n} \lambda_i = 1$, and $\lambda_i > 0$, for $\mathbf{i} = 1, \cdots, \mathbf{n}$. The intuitive meaning of the transition probabilities is that with probability $(1-\mathbf{r})$, the next reference is chosen according to distribution $\{\lambda_i\}$. Based on this model, Easton derived expressions for working set functions $\mathbf{s}(\tau)$ and $\mathbf{m}(\tau)$, and showed that the results agree with observations of an AAS trace for large window sizes (τ's). Note that this model is a degenerated VSLM with a locality size of one, and the IRM is a special case of this model with $\mathbf{r} = 0$.

In a later study, Easton [East78] generalized the above model through the concept of **reference clusters**, which is an observed phenomenon that once a page is referenced, there are often additional references to it within a relatively short time. A reference to a page is defined to be **primary** if the time since the page was last referenced exceeds a particular value τ. Otherwise, the reference is **secondary**. The generalized model describes the behavior of primary references, and ignores the secondary references under the assumption that they will not cause any page fault. The model states that once a cluster has ended, the time to the initiation of the next clustered is a random variable with a geometric distribution. Formally stated, there exist a finite τ and probabilities $\{\rho_i\}$ such that for all $\mathbf{T} \geq \tau$,

---

[5] AAS is an IBM database system known as the Advanced Administrative System.

$\mathbf{Pr}\{\mathbf{r}_t = \mathbf{x}_i \mid \mathbf{r}_{t-T} \neq \mathbf{x}_i, \mathbf{r}_{t-T+1} \neq \mathbf{x}_i, \cdots, \mathbf{r}_{t-1} \neq \mathbf{x}_i\} = \rho_i,\ \mathbf{i} = 1, \cdots, \mathbf{n}.$

Intuitively, $1/\rho_i$ is the mean interval between clusters of references to page **i**. Again, the expressions for working set functions $\mathbf{s}(\tau)$ and $\mathbf{m}(\tau)$ derived under the new model were validated by empirical results from trace strings. Note that the first Easton model is a special case of the second, with $\tau = 1$ and $\rho_i = (1-\mathbf{r})\lambda_j$.

Both models discussed in the preceding paragraphs have demonstrated success in determining working set functions $\mathbf{s}(\tau)$ and $\mathbf{m}(\tau)$. However, as Easton pointed out [East78], the models are not suited to addressing such issues as determining the distribution of intervals between successive page faults, since the models do not explicitly deal with the correlation between references of different pages. Furthermore, the models do not provide enough details on page reference behavior, and hence are inadequate for studying buffer replacement policies.

2.2.3. Hot Set Model

Most of the behavior models we have discussed so far are Markov chain models that describe the behavior of a process[6] by some stochastic process. Based on the assumption that no advance knowledge is available [Denn68a], these models predict a process's future behavior by past statistics or simply by probability distribution. However, there are exceptions to this assumption, for example, in relational database systems [Astr76] [Ston76]. A relational database system provides its users with a high-level non-procedural interface and lets its optimizer [Wong76] [Seli79] decide how the data will be accessed. Thus, the reference patterns for data pages can be predicted at the time the optimizer selects an access plan for a query.

---

[6] A process in this context refers to the execution of either a program or a query.

The hot set model proposed by Sacco and Schkolnick [Sacc82] is a query behavior model for relational database systems that integrates the advance knowledge of reference patterns into the model. In this model, a set of pages over which there is a looping behavior is called a **hot set**. If a query is given a buffer large enough to hold its hot sets, its processing will be efficient because the pages referenced in a loop will stay in the buffer. On the other hand, a large number of page faults may result if the memory allocated to a query is insufficient to hold a hot set. Plotting the number of page faults as a function of buffer size, we can observe a discontinuity around the buffer size where the above scenario takes place. There may be several such discontinuities in the curve, and each is called a **hot point**.

In a nested loops join in which there is a sequential scan on both relations, a hot point of the query is the number of pages in the inner relation plus one. The formula is derived by reserving enough buffers to hold the entire inner relation, which will be repeatedly scanned, plus one buffer for the outer relation, which will be scanned only once. If, instead, the scan on the outer relation is an index scan, an additional buffer is required for the leaf pages of the index. Following similar arguments, the hot points for different queries can be determined.

Applying the predictability of reference patterns in queries, the hot set model provides a more accurate reference model for relational database systems than a stochastic model. However, the derivation of the hot set model is based partially on the LRU replacement algorithm, which is inappropriate for certain looping behavior. In fact, the MRU (Most-Recently-Used) algorithm, the opposite to the LRU algorithm, is more suited for cycles of references [Thor72], because the most-recently-used page in a loop is the one that will not be re-accessed for the longest period of time. Going back to the nested loops join example, the number of page faults will not increase dramatically when the number of buffers drops

below the "hot point" if the MRU algorithm is used. In this respect, the hot set model does not truly reflect the inherent behavior of some reference patterns, but rather their behavior under the LRU algorithm.

2.3. The Query Locality Set Model

As discussed in the previous section, the hot set model, which integrates knowledge on reference patterns as part of the model, is better at describing the reference behavior of database systems than other stochastic models. Based on similar assumptions, we propose a new query behavior model for relational database systems which we shall refer to as the Query Locality Set Model (QLSM). However, unlike the hot set model, the QLSM is not tied to any particular buffer replacement algorithm. The main focus of the QLSM is to characterize the "inherent" behavior of database queries rather than to describe the paging behavior under a certain buffer management policy.

In the following, we shall start with a description of the storage organizations used in our analysis. Based on the storage model, page reference patterns found in relational database systems will be characterized and discussed. Finally, the classification of page reference patterns will be applied to the analysis of a number of database operations.

2.3.1. Storage Organizations and Access Methods

Without loss of generality, we shall use disks as the storage medium for databases in our discussion. Each disk is physically partitioned into pages, which are the basic unit of access. **Files** and **records** are the logical structures imposed on disks and pages, respectively. A file is a portion of a disk that consists of logically related pages. Each page, in turn, contains a number of records. We are not concerned with the exact structure of a

record, except that it contains a number of fields whose values may be used as search keys for accessing records. Roughly speaking, a file corresponds to a relation (or table), and a record corresponds to a tuple in the relational data model.

There are storage organizations that require additional space for maintaining auxiliary access structures, such as indices. Some storage organizations mix access structures and data together, even on the same page[7]. However, many other equally powerful storage organizations do not place such a restriction on the underlying implementations. The storage system of System R [Blas77] and the Frame Memory design [Marc81], for instance, both have a clear separation between control and data pages. In the following, we shall assume that the control and the data portion of a storage organization are placed (at least conceptually) in different files. For example, a file with a B^+-tree index can be implemented by two files; one for storing data records, and the other for storing key information and pointers to the data records. We consider the following storage organizations:

(1) Heap Organizations

This is the simplest form of data organization in which there is no logical ordering among records. Adding a record to a heap file, with no checking for database constraints, consists simply of appending the record to the end of the file. However, retrieving a record that satisfies a certain condition is expensive. A sequential scan seems to be the only meaningful way of accessing the file.

(2) Ordered Sequential Organizations

This organization is similar to the heap organization except that there exists a logical ordering among records in the file. Ordered files are useful for range scans and

---

[7] For example, a DL-tree [Lome83] may have indices and data compressed together in leaf pages.

merge joins [Blas77]. Like the heap organization, a sequential scan seems to be the only pattern of accesses[8].

(3) Indexed Organizations

An index is usually structured as a tree. Examples in this category include ISAM [Mart75], B^+-tree [Come79], digital tree [Knut73], K-D-B tree [Robi81], and so on. The leaves of an index tree are assumed to be linked to facilitate sequential scans. Based on the physical ordering of data records, an index can be classified as either **clustered** or **non-clustered**. For a clustered index, the physical ordering of data records is generally the same as that of their key values. Furthermore, an index is said to be **unique** if there is no duplication of keys, and **non-unique** otherwise.

(4) Hashed Organizations

A hashed organization uses some key transformation technique to locate records. While it provides good random access time, it has poor performance on sequential scans. Although there exist many variants, such as extendible hashing [Fagi79] and linear hashing [Litw80], we shall use the term **hashing** to refer to all the key-transformation techniques. To simplify discussion, we shall assume that hashing is used only when the key field of the data file is unique. Random access is the only reasonable reference pattern to this organization.

2.3.2. A Classification of Page Reference Patterns

Although a database system usually deals with a large amount of data, the set of operations required to manipulate databases is surprisingly small. Furthermore, the patterns of

[8] A binary search on a sorted file is possible if the ordering and disk locations of pages are directly available, say, from a page table. However, it is generally inappropriate because it necessitates time consuming disk seek operations [Mart75].

page references found in database operations are very regular and predictable. In the following, we shall propose a taxonomy for classifying page reference patterns exhibited by access methods and database operations. This classification is applicable to both data page and control page accesses, and is useful for identifying the importance of a page for the purpose of buffer management.

2.3.2.1. Sequential References

In a sequential scan, pages are referenced and processed one after another. In many cases, a sequential scan is done only once without repetition. For example, during a selection operation on an unordered relation, each page in the file is accessed once and then thrown away forever (so to speak). A single page frame provides all the buffer space that is required. We shall refer to such a reference pattern as **straight sequential** (SS).

Local re-scans may be observed in the course of a sequential scan in certain operations. That is, once in a while, a scan may back up a short distance and then start forward again. This can happen in a merge join [Blas77] in which records with the same key value in the inner relation are repeatedly scanned and matched with those in the outer relation. We shall call this pattern of reference **clustered sequential** (CS). Obviously, records in a cluster (a set of records with the same key value) should be kept in memory at the same time if possible.

In some cases, a sequential reference to a file may be repeated several times. In a nested loops join, for instance, the inner relation is repeatedly scanned until the outer relation is exhausted. We shall call this a **looping sequential** (LS) pattern. The entire file that is being repeatedly scanned should be kept in memory if possible. If the file is too large to fit in memory, the most-recently-used page is the best candidate for replacement since it will

not be re-accessed for the longest period of time.

2.3.2.2. Random References

A random reference pattern consists a series of independent accesses. For example, in an index scan through a non-clustered index, the reference pattern of the data pages can be described by the Independent Reference Model (IRM) with equal access probability for each data page. The total number of distinct pages accessed in a series of random accesses can be estimated by Yao's formula [Yao77]:

$$\mathbf{b(m,p,k)} = \begin{cases} \mathbf{m}\left[1 - \prod_{\mathbf{i}=1}^{\mathbf{k}} (\mathbf{n}-\mathbf{p}-\mathbf{i}+1)/(\mathbf{n}-\mathbf{i}+1)\right] & \textbf{when } \mathbf{k} \le \mathbf{n}-\mathbf{p} \\ \mathbf{m} & \textbf{when } \mathbf{k} > \mathbf{n}-\mathbf{p} \end{cases}$$

where **n** is the total number of records in the file, **m** is the number of pages in the file, **p=n/m** is the blocking factor (number of records on a page) and **k** is the number of accesses.

There are cases when a locality of reference exists in a series of "random" accesses. This may happen in the evaluation of a join in which a file with a non-clustered and non-unique index is used as the inner relation, while the outer relation is a sorted (or clustered) file with non-unique keys. The reference string generated by accesses to the data pages of the inner relation can be represented by as a regular expression $s_1^{n_1} \cdot s_2^{n_2} \cdot \ldots \cdot s_m^{n_m}$ where each cluster s_i is a sequence of random variables (page numbers). We shall call this pattern of reference **clustered random** (CR). The reference behavior of a clustered random reference is similar to that of a clustered sequential scan. If possible, each page containing a

record in a cluster should be kept in memory at the same time.

2.3.2.3. Hierarchical References

A hierarchical reference is a sequence of page accesses that form a traversal path from the root down to the leaves of an index. If the index is consulted only once as in an ad hoc query, one page frame is enough for buffering all the index pages. We shall call this a **straight hierarchical** (SH) reference. There are cases in which a tree traversal is followed by a sequential scan through the leaves. To distinguish these cases from SH references, we shall call these references either **hierarchical with straight sequential** (H/SS) if the scan on the leaves is SS, or **hierarchical with clustered sequential** (H/CS) otherwise. Note that the reference patterns of an H/SS reference and an H/CS reference are similar to those of an SS reference and a CS reference, respectively.

During the evaluation of a join in which the inner relation is indexed on the join field, repeated accesses to the index structure may be observed. We shall call this pattern of reference **looping hierarchical** (LH). In an LH reference, pages closer to the root are more likely to be accessed than those closer to the leaves. The access probability of an index page at level i, assuming the root is at level 0, is inversely proportional to the ith power of the fan-out factor of an index page. Therefore pages at an upper level (which are closer to the root) should be favored to stay in buffer over those at a lower level. In many cases, the roots is perhaps the only page worth keeping in memory since the fan-out of an index page is usually high.

2.3.3. Reference Behavior of Database Operations

After discussing how to classify page reference patterns, we shall demonstrate how this classification can be applied to analyzing the reference behavior of database operations. In essence, the reference behavior of a database operation is a composition of a number of simple reference patterns. In the following, we shall start with the discussion of three relational database operations, projection, selection and join. Analysis of sort operations will be given next, followed by a discussion of aggregate functions.

A projection, without duplicate elimination[9], usually involves a source file and a file to hold the projected result. Fields within a source record may be eliminated, transposed, or perhaps replicated. However, no matter what is required for the projection, the page reference pattern to the source file is always straight sequential. Appending records to the output file is also sequential. Note that if a projection is optimized as part of another operation, it requires no buffers at all.

When no access structures are available, a straight sequential scan is the only way to execute a selection operation. However, if an index exists, we can then traverse down the index and do a sequential scan over some leaf pages (ie., an H/SS reference). In this case, the reference pattern to the data pages is straight sequential if the index is clustered, and random otherwise. If hashing is used, references to the hash table and to the data pages are both random.

There are a variety of algorithms for evaluating joins. Despite the variety, they share one common pattern: extract a record from the first file, and use the value of its join field (as

---

[9] Removing duplicates from a file can be done by first sorting the file and then scanning it sequentially to eliminate adjacent duplicates. We shall delay the discussion of the sort operation until later in the section.

the search key) to do a selection on the second file. The access pattern to the first file is always straight sequential[10]. Therefore we shall concentrate on the reference pattern to the second file.

(1) For the nested loops algorithm, the reference pattern to the second file is looping sequential.

(2) If an index exists on the second file, several combinations of reference patterns to the index and data pages are possible: (i) H/CS and CS, when the first file is clustered and non-unique[11], while the index to the second file is clustered; (ii) H/CS and CR, when the first file is clustered and non-unique, while the index to the second file is non-clustered; (iii) H/SS and SS, when the first file is clustered and unique, while the index to the second file is clustered; (iv) H/SS and random, when the first file is clustered and unique, while the index to the second file is non-clustered; (v) LH and random, otherwise.

(3) If the second file is hashed on the join field, the reference patterns to both the hash table and the data pages are random.

(4) For the sort merge algorithm, the reference pattern on the second file is either straight sequential or clustered sequential, depending on whether the join attributes of the outer relation are unique.

A common way to implement an external sort utility is to use an N-way sort-merge algorithm [Braw70] [Knut73]. The file to be sorted is first partitioned into N-page segments called runs, each of which are sorted by an internal sort routine. Then every N input runs

---

[10] In the sort merge join algorithm, both source files may have to be sorted first. We shall factor out the effect of sort and delay its discussion until later.

[11] A file is clustered if the records in the file are sorted or nearly sorted. A file is unique if its key field is unique (i.e. no duplicate keys).

are merged into a larger (output) run in each iteration. The whole process terminates when there is only one input run left. The number of iterations required for the merging process depends on the value of **N**. To sort a file with **M** pages, $\lceil \log_N M \rceil - 1$ merging iterations are needed. Since each iteration requires one complete scan of the entire file, using a larger **N** is clearly preferred.

During an iteration of the merging process, accesses to each of the **N** runs are equally frequent. We need one page frame for each run. Except for prefetching, additional page frames will not improve the performance as the locality of reference in each run is so predominant. If one has extra page frames, it is more beneficial to increase the value of **N** rather than allocating more page frames to each run. Thus, the number of page frames to use is determined by the availability of free page frames in the system. An **N**-way merge sort is possible when (**N**+1) page frames are available. A buffer replacement strategy may be unnecessary for a sort operation since it is easy for a sort utility to do its own buffering. However, we can still fit sort into our framework by viewing each run in the merging phase as a separate file under sequential scan. The reference pattern to the source file in the initial sorting phase can also be treated as straight sequential scan.

Many database systems provide simple aggregate operations, such as MIN, MAX, SUM, AVG, COUNT, and so on. Despite the difference in semantics, their reference patterns to the source file are the same, namely, straight sequential. For aggregate functions that involve grouping records before aggregation, two implementations are possible. The first method is to sort the source file on the grouping attribute and then do a sequential scan on the sorted file to calculate the aggregate value for each group. The second and more efficient method is to scan the source file sequentially while using an auxiliary structure, such as a B^+-tree or hash table, to hold the intermediate aggregate value for each group. Thus,

the reference behavior of these aggregate operations has been covered by the previous discussion.

In this section, we have presented a new query behavior model, the query locality set model, for database systems. Using a classification of page reference patterns, we have shown how the reference behavior of common database operations can be described as a composition of a set of simple and regular reference patterns. Like the hot set model, the QLSM has an advantage over the stochastic models due to its ability to predict future reference behavior. However, the QLSM avoids the potential problems of the hot set model by separating the modeling of reference behavior from any particular buffer management algorithm. In the next chapter, we shall describe the construction of a predictive buffer management algorithm based on the QLSM.

CHAPTER 3

MAIN MEMORY MANAGEMENT

Memory management is concerned with the management of a critical resource in a computer system, namely the main memory (or primary memory). It is one of the most studied areas in computer science. Numerous papers on this subject have appeared in the literature in the past quarter century. In this chapter, We shall review some important results from these studies and propose a new approach to database buffer management. Section 1 is a survey of memory policies for virtual memory systems. Their counterpart in database systems, namely buffer management algorithms, will be discussed in section 2. A new buffer management algorithm, which is based on the query locality set model discussed in the previous chapter, will be presented in the last part of this chapter.

3.1. Memory Management in Virtual Memory Systems

By executing several jobs concurrently, a multi-programmed system can improve its resource utilization, and hence its **throughput**, i.e. the number of jobs completed per second. However, when the number of active jobs exceeds certain limit, a collapse in performance can occur due to excessive paging activity. This phenomenon is known as **thrashing** [Denn68b]. Therefore, the purpose of **load control** is to maximize resource utilization while preventing the system from entering the thrashing state.

Another issue in memory management is how main memory shall be allocated among competing jobs. Some might tend to think that **memory partitioning** is simple and can readily be dealt with by dividing the main memory evenly among all active jobs. However,

Belady and Kuehner [Bela69] have shown that even a simple biased partitioning scheme can improve the throughput of a computer system by 10 to 15 percent. Thus, this apparently simple problem can have a significant effect on the performance of a computer system.

One remaining question is what action should be taken when a page requested is not in memory and there are no page frames available. Usually, a resident page (i.e. a page in main memory) is chosen for removal by a **replacement algorithm**. The objective of such a replacement algorithm is two-fold. On one hand, it should keep those pages that are currently being used in main memory to reduce paging traffic. On the other hand, it should remove those resident pages that are unlikely to be re-used to ensure the efficiency of memory utilization.

Load control, memory partitioning and page replacement are three closely related aspects of a memory policy (Figure 3.1). In fact, as we shall see, there are memory policies that use a single mechanism to deal with all three problems. However, for ease of discussion, they will be treated as three different issues to be examined separately in this section.

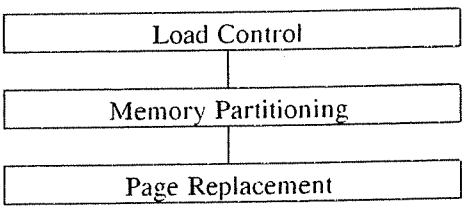

Figure 3.1 Issues of Main Memory Management

3.1.1. Replacement Algorithms

The replacement algorithm was called the "learning" program when it was incorporated in the "one-level storage system" (i.e. virtual memory system) of ATLAS in the late 50's [Kilb62]. Since then, numerous replacement algorithms have been proposed, and some of them have been implemented in real systems. These algorithms can conveniently be classified into two major types, lookahead and non-lookahead (Figure 3.2). For completeness, we shall examine both types of algorithms in more detail. The results of some empirical studies will also be summarized to conclude the discussion of replacement algorithms for virtual memory systems.

3.1.1.1. Lookahead Replacement Algorithms

A lookahead replacement algorithm requires a priori knowledge of the actual page requests before program execution. Therefore, it is generally impossible to implement one

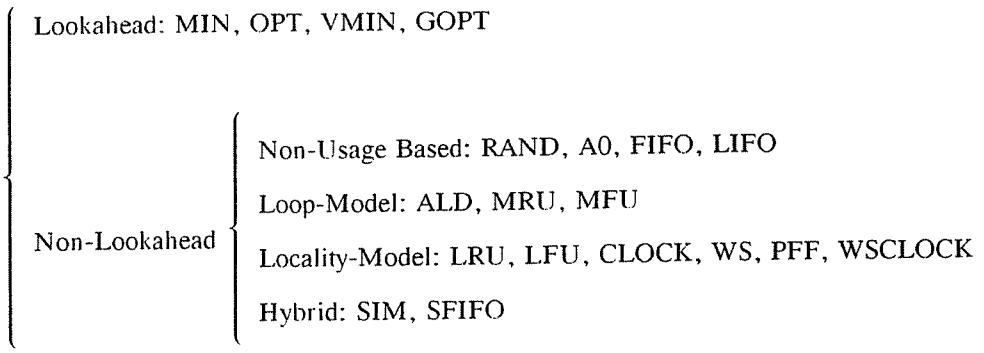

Figure 3.2 Classification of Replacement Algorithms

in practice. However, an optimal lookahead replacement algorithm can provide useful information for evaluating real systems. For example, the MIN algorithm presented by Belady [Bela66] is an optimal replacement algorithm which provides a lower bound on page fault rates under fixed memory allocation[1]. Mattson et al. [Matt70] presented another algorithm, OPT, which also yields the minimum number of page faults for a given reference string. In essence, both algorithms remove the resident page with the longest time to its next reference when a free page frame is required.

The VMIN algorithm presented by Prieve and Fabry [Prie76] is an optimal replacement algorithm for variable space allocation. After each reference, VMIN removes the page just referenced if and only if the page will not be re-used within the next **R/U** time units, where **R** is the cost of a page fault and **U** is the cost of retaining a page in memory per time unit. Note that **R/U** can be viewed as the size of a time window into the future, and is in contrast to the working set parameter τ, which is the size of a backward time window. Using the page fault rate as a function of (average) memory sizes as the criterion[2], Prieve and Fabry have demonstrated the optimality of VMIN. Denning and Slutz [Denn78b] later presented an extended version of VMIN, called the generalized optimal policy (GOPT), which minimizes the aggregated retention and swapping costs.

3.1.1.2. Non-lookahead Replacement Algorithms

Although complexity of implementation and run-time overhead are commonly used criteria for characterizing replacement algorithms, it is convenient to categorize replacement

---

[1] Number of page faults is usually the criterion for evaluating replacement algorithms under fixed memory allocation. A different characterization of the optimality of replacement algorithms was presented by Pomeranz [Pome71].

[2] Under mild assumptions, this criterion is equivalent to the time-space (time integration of memory sizes) criterion.

algorithms according to the reference pattern they anticipate. From this viewpoint, existing non-lookahead algorithms can further be classified into four sub-types: non-usage based, loop-model, locality-model and hybrid. In the following, we shall examine each sub-type in more details.

A non-usage based algorithm anticipates, more or less, a random reference pattern. It does not base its replacement decisions on programs' past history. Consequently such an algorithm is usually simple to implement and has low run-time overhead since no usage statistics need to be maintained. Examples of non-usage based algorithms include:

(1) RAND [Bela66]: When a page frame is needed, randomly select a page for replacement. This algorithm is justifiable if programs obey the random reference model.

(2) A0 [Aho71]: When a page frame is needed, replace the page that has the lowest probability of being accessed. Aho, Denning and Ullman [Aho71] have shown that A0 is optimal if program behavior satisfies the IRM[3].

(3) FIFO (First-In-First-Out) [Bela66]: When a page frame is needed, replace the oldest page in memory. FIFO is based on the assumption that programs tend to follow instructions in sequence so that the page which has been in memory longest is least likely to be re-used. Another argument for FIFO is that it is easier to maintain a cyclic counter than to generate a random number.

(4) LIFO (Last-In-First-Out): When a page frame is needed, replace the youngest page in memory. LIFO, the opposite to FIFO, has little logical justification. However, LIFO may occasionally out-perform some other algorithms. For example, LIFO is better than LRU when the available space is not enough to hold all the pages that are being

[3] Lew [Lew76] later extended their work by formulating page replacement as an optimal control problem, which can be solved by dynamic programming.

repeatedly scanned.

The loop-model algorithms look for cycles of references in programs. These algorithms assume that a program's behavior is dominated by iterations (e.g. DO loops in FORTRAN or FOR loops in PASCAL). Some examples are:

(1) ALD (ATLAS Loop Detection) [Kilb62]: The ALD algorithm keeps statistics on every page's presence in and absence from main memory. Based on these statistics, a page is removed if it is projected to be no longer active in the current loop.

(2) MRU (Most-Recently-Used): Replace the Most-Recently-Used page when a page frame is needed. This method yields the minimum number of page faults for cyclic references because the MRU page in a loop is the one that will not be re-used for the longest period of time.

(3) MFU (Most-Frequently-Used): Replace the Most-Frequently-Used page when a page frame is needed. This method is similar to MRU, except that a usage count is used as the basis for predicting the future access time of a page.

The locality-model algorithms are designed to capture "locality of reference" in programs. These algorithms are more popular than other algorithms due to the frequently observed locality in real programs. However, they are also more expensive since statistics for tracking the locality need to be maintained. Some well-known examples are:

(1) LRU (Least-Recently-Used) [Bela66]: When a page frame is required, remove the page that has not been referenced for the longest period of time. Denning, Spirn and Savage [Denn72b] have shown that under the simple LRU model, the LRU algorithm is optimal when the reference probabilities associated with stack distances are non-increasing.

(2) LFU (Least-Frequently-Used) [Matt70]: When a page frame is required, remove the page that is least frequently used. This method is similar to LRU, except that a usage count is used as the basis for deciding the priority of a page.

(3) CLOCK [Bela66]: This method requires a reference bit for each page frame in main memory. When a page frame is needed, a pointer resumes a cyclic scan through the page frames, resetting the reference bit of used frames and choosing the first unused frame for replacement. CLOCK is an approximation to LRU with a simpler implementation. An early version was presented by Belady [Bela66]. Later it was adopted in MULTICS under the name "First-In-Not-Used-First-Out" (FINUFO) [Corb68]. A detailed analysis of the CLOCK algorithm and some variations of it were presented by Easton and Franaszek [East79].

(4) WS (Working Set) [Denn68a]: The WS algorithm is based on the concept of "working set". This algorithm assigns a resident set to a process that is identical to its observed (or estimated) working set. Under the time-window WS algorithm, pages that have been referenced in the past τ time units (in the process's virtual time) are retained in memory[4]. The WS algorithm tends to over-allocate memory during phase transitions. This problem is alleviated in a modified version of WS, the Damped Working Set (DWS) algorithm proposed by Smith [Smit76a], which trys to "clip off" the overshoot of working sets during transitions.

(5) PFF (Page Fault Frequency) [Chu72]: The PFF algorithm proposed by Chu and Opderbeck is an approximation to WS which re-computes a process's working set only

[4] The original time-window WS algorithm [Denn68a] is actually a special case of the generalized WS policy (GWS) [Denn78b], which keeps as the resident set those pages whose retention costs do not exceed their retrieval costs. However, due to its wide use in the literature, WS will be used to refer to the time-window WS unless otherwise stated.

at page fault time in order to reduce the run-time overhead. Under the PFF algorithm, a process is given an additional page frame if the interval between successive page faults is smaller than a certain threshold. Otherwise, pages that have not been referenced within the threshold interval are removed at the page fault time[5].

(6) WSCLOCK [Carr81]: The WSCLOCK algorithm proposed by Carr and Hennessy is a compromise between the WS and CLOCK algorithms. Its purpose is to gain the performance advantage of WS and the simplicity of CLOCK. The scanning structure of WSCLOCK is similar to that of CLOCK. However, the WS principle is applied to the page frame which is currently being scanned. In other words, the status of the page frame is checked against the working set information of its owner process. A page is available for replacement when it is determined that the page is no longer in the working set of the owner process.

The hybrid algorithms attempt to achieve the advantages of two or more different algorithms by combining them into one integrated algorithm. An example is the SIM algorithm proposed by Thorington and Irwin [Thor72]. SIM is an adaptive algorithm that simultaneously simulates several decision rules (e.g. RAND, FIFO, LRU, etc.) at run time. The decision rule "on duty" is the one which yielded the best performance in the last time frame. Thus the SIM algorithm can be viewed as a "parallel" hybrid algorithm. In contrast, there are hybrid algorithms in which the main memory is managed by a two-stage mechanism [Turn81] [Baba82]. Under such an algorithm, most replacement decisions are made at the first stage by a replacement policy with a low implementation cost, e.g. FIFO. The poor performance of the first policy is compensated at the second stage by a better policy with a

[5] The parameter θ used in the actual algorithm is the page fault frequency, which is the reciprocal of the interval between successive page faults.

higher implementation cost, e.g. WS. The intent is to achieve the performance of the second policy while keeping the cost close to that of the first policy. An typical example is the SFIFO (Segmented FIFO) algorithm, a combination of FIFO and LRU, proposed by Turner and Levy [Turn81].

3.1.1.3. Comparative Evaluation of Replacement Algorithms

One of the earliest simulation studies of replacement algorithms was conducted by Belady [Bela66]. By simulating programs written for an IBM 7094/94, Belady found that RAND and FIFO generated about two to three times as many page faults as his MIN algorithm. In the same study, ALD was found to perform slightly better than RAND and FIFO. Nonetheless, Belady concluded that too much reliance on cumulative information is not worthwhile. His viewpoint was later supported by experiments done on the ATLAS [Bayl68] in which ALD generated about 10 percent more page faults than LRU. In a paging study on an IBM 360/50, Coffman and Varian [Coff68] found that LRU yielded a performance within 30 to 40 percent of that of Belady's MIN. In a later study by Thorington and Irwin [Thor72], LRU was reported to perform better than FIFO and LFU, but not as well as the SIM algorithm. Although CLOCK was initially designed as an approximation to LRU, evidence indicates that the two algorithms are comparable in performance. This was first suggested by Belady's simulation results [Bela66], and later confirmed by Grit and Kain's experiments [Grit75]. Similar results were also obtained by Easton and Franaszek [East79].

By simulating programs running on the UCLA SIGMA-7, Chu and Opderbeck [Chu72] showed that WS has a better space-time product than does LRU, and their PFF algorithm is comparable to WS in performance. Using trace data from programs running under TSS/360, Prieve and Fabry [Prie76] found that WS generated up to 50 percent more

page faults than VMIN. Similar results on WS and VMIN were also obtained by Smith [Smit76a]. Using reference strings generated by an Inter-Reference Interval Model (IRIM), Carr and Hennessy [Carr81] demonstrated that their WSCLOCK algorithm is comparable to WS in performance.

3.1.2. Memory Partitioning

Consider a multi-programmed system with **M** pages and **n** active processes, denoted by $P_1, \ldots, P_n$. Associated with each process P_i at time **t** is its resident set $Z_i(t)$, containing $z_i(t)$ pages. The allocation of the main memory can be described by a partition vector

$$Z(t) = \left(Z_1(t), \ldots, Z_n(t)\right)$$

under the constraint $\sum_{i=1}^{n} z_i(t) \leq M$. In the general case in which each $z_i(t)$ is a time varying function, $Z(t)$ is termed a **dynamic partition**. In contrast, $z_i(t)$ is a constant z_i under a **fixed partition** Z. Denning and Graham [Grah74] [Denn75a] proposed a classification scheme which further divides memory management policies into two fixed-partition and three dynamic-partition classes (Figure 3.3).

An **equi-partition** is a fixed partition in which every process is given the same amount of memory, that is $z_i = M/n$ for all **i** [Matt68]. Otherwise, a fixed partition is an **imbalanced partition** in which the main memory is not equally divided among active processes. Imbalanced partitioning is a viable solution to memory allocation if the memory demands of different programs are static and predictable. Even when all processes have identical memory requirements, an imbalanced partition may still be more efficient than an equi-

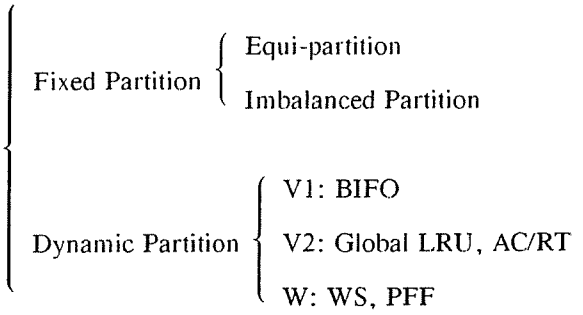

Figure 3.3 Classification of Memory Partitioning

partition. This can be attributed to the convexity of lifetime functions[6]. Operating within a convex region, a multi-programmed system can increase the average interval between page faults, and hence the efficiency, by relocating more memory to one process, since the marginal value of a page frame to that process is higher. This argument has repeatedly been suggested by several studies [Cham73] [Denn75a] [Ghan75b][7].

One main advantage of fixed partitioning is its low implementation cost. However, this advantage can be offset by the loss of storage utilization for programs with large variations in locality size. This effect has been analyzed by several studies. Coffman and Ryan [Coff72] showed how variable partitioning can improve storage efficiency when the working set sizes of programs are modeled by a Gaussian process. Using the simple LRU model for program behavior, Oden and Shedler [Oden72] found that an increase is obtainable in the average execution interval between page faults under variable partitioning. Similar results were

[6] Belady and Kuehner [Bela69] found that the lifetime functions of many programs are nonlinear, and can be approximated by $\mathbf{L(s)=as^k}$, where **a** varies with the individual program and **k** has a value in the vicinity of 2.

[7] Ghanem actually showed a stronger result that gives the condition under which imbalanced partitioning is more efficient than equi-partitioning and vice versa.

obtained by Denning and Spirn [Denn73] in a study on dynamic storage partitioning. Thus, the general conclusion is that dynamic partitioning is more efficient than fixed partitioning.

Based on their correlation with locality changes of programs, dynamic-partition policies can be grouped into three sub-classes:

(1) V1: The memory partition **Z(t)** is a time varying function, but with no explicit correlation to the reference patterns of the active processes. An example of a class V1 policy is a "biasing" scheme proposed by Belady and Kuehner [Bela69]. Their BIFO (Biased FIFO) algorithm, a modified version of FIFO, selects one of the competing processes to be the favored process, whose pages are exempted from replacement consideration for a period of **p** page faults. The privilege is then passed on to the next process selected, and so on. Despite the somewhat arbitrary nature of the algorithm, a gain of 10 to 15 percent in throughput over the simple FIFO was observed.

(2) V2: Variation in **Z(t)** is directly correlated with the aggregate behavior of the active processes, but the individual locality of each process is not explicitly identified. The global LRU algorithm, in which all the pages in main memory are ordered in a global LRU stack for replacement purpose, is a typical example of a class V2 policy. Another example is the AC/RT algorithm proposed by Belady and Tsao [Bela73]. In this algorithm, two control parameters are maintained for each process: a memory demand indicator RT (Round-Trip frequency) defined as the relative frequency that the page demanded is the one which has been most recently replaced; and a memory utilization indicator AC (Activity Count) defined as the average fraction of resident pages being referenced between page faults. The AC/RT algorithm steals a page from the demanding process itself if its RT value (memory requirement) is low. Otherwise, AC/RT increases the memory of the process by selecting a page from another program

with the lowest AC value (memory utilization).

(3) W: The resident sets **Z(t)** are assigned according to the (estimated) working set of each active process. The WS algorithm, which assigns a resident set to a process that is identical to its working set, is perhaps the most well-known example of a class W policy. The PFF algorithm is another well-known example[8]. Going one step further, Ghanem [Ghan75a] showed how optimal partitioning[9] can be achieved by selecting an optimal set of window sizes $\{\tau_i\}$ such that

$$\frac{\ddot{w}_i(\tau_i)}{\dot{w}_i(\tau_i)} = \lambda$$

for all i, where $\ddot{w}_i(\tau_i)$ and $\dot{w}_i(\tau_i)$ are the first and second derivatives of the working set size function of process i, respectively.

In addition to presenting a classification of memory partitioning policies, Denning and Graham [Denn75a] also gave a relative ranking among the five classes of policies, from worse to best: equi-partition, fixed imbalanced partition, V1, V2 and W. As mentioned earlier, imbalanced partitions are more efficient than equi-partitions due to the convex property of lifetime functions. Similarly, the convexity argument also explains why V1 partitions are generally better than fixed partitions, although they do not correlate memory allocation with program behavior. V2 policies improve storage efficiency by relocating memory according to the dynamic requirements of processes. This claim is substantiated by experiments conducted by Oliver [Oliv74] in which global LRU generated less page faults than local LRU with equi-partition. Class W policies are more efficient than V2 policies partly because they attend to the individual memory needs of each process, and partly due to their inherent load

---

[8] Chamberlin et al. [Cham73] suggested a similar method in which page frames are allocated to processes in such a way that all their page fault rates are kept equal.

[9] His goal was to minimize the total number of page faults of the system.

control. Empirical studies [Rodr73] [Opde74] [Smit76a] have provided evidence that W policies are superior to V2 policies.

3.1.3. Load Control

To attain its performance, a computer system needs to regulate not only memory allocation but also the multiprogramming level (MPL) of the system. Without such regulation, thrashing [Denn70] can occur and may severely degrade the performance of the system. Rodriguez-Rosell and Dupuy [Rodr72] reported the results of experiments which revealed how the performance of CP-67, an operating system running on the IBM 360/67, degraded when the number of logged-in users exceeded a certain value. Using a queuing model, Arora and Kachhal [Aror73] illustrated through a stochastic analysis how CPU idle time increases as the number of concurrent jobs grows beyond certain limit. Thus, the basic need for a load control is to restrict the number of active jobs in a system within some limit $\mathbf{n}_{max}$ in order to prevent thrashing. Empirical data [Weiz69] [Rodr72] [Rodr73], supported by later queuing network modeling [Bran74] [Bade75] [Denn75a], also indicates the existence of an optimal MPL $\mathbf{n}_{opt}$ at which the performance of the system is at its peak. A more ambitious load controller attempts to locate such an optimal MPL and adjusts its load accordingly. Due to the dynamic nature of user load, an adaptive mechanism is often required to re-estimate the optimal MPL $\mathbf{n}_{opt}(t)$ from time to time. At any rate, the function of a load controller is to regulate the level of multi-programming according to certain constraint in order to ensure the performance of the system. Based on the nature of their control criterion, load controllers can be categorized into three classes: static, feedback and predictive (Figure 3.4).

- Static: OS/VS1, LT/RT
- Feedback: 50% Rule, L=S, Discrete Optimal Control
- Predictive: Knee Criterion

Figure 3.4 Classification of Load Control

The simplest load control is to set a fixed limit $\mathbf{n}_{max}$ on the number of concurrent jobs that are allowed to compete for system resources. We call this a **static** load control. It has been used on several IBM 360/370 operating systems [Madn74], including OS/360-MFT, OS/VS1, and so on. Given a system configuration, the value of $\mathbf{n}_{max}$ can be determined by some optimization technique, such as integer programming [Matt68], or simply by experience. Usually, the value of $\mathbf{n}_{max}$ is conservatively chosen to be smaller than any possible MPL that might lead to thrashing. Therefore, a system's memory tends to be under-utilized under a static load control. Another static rule is called the LT/RT (Loading Task/Running Task) control [Carr81], in which the number of loading tasks[10] (processes) is limited according to the number of paging devices. It has been observed that many programs have a distinct initial loading phase in which many new pages are referenced [Jose70]. Therefore it is sensible to limit the number of loading processes to prevent them from saturating the paging devices. LT/RT can be used with other load control mechanisms as an auxiliary constraint to deal specifically with the loading phase of processes.

The second approach to load control is based on the feedback control technique. It requires a feedback parameter which, in our case, is an indicator that reflects the current

[10] A process is said to be in its loading phase until it has executed for τ_0 units of CPU time, where τ_0 is a parameter of the RT/LT scheme.

loading of the system. Device utilization and page fault rate have long been recognized as effective load indicators for virtual memory systems [Braw68] [Cham73]. An example of a feedback load control that is based on device utilization is the "50% rule" [Bade75] [Lero76]. This criterion constraints the load so that the paging devices are busy about half the time. A criterion that uses lifetime function (the reciprocal of page fault rate) as the feedback parameter is the "L=S criterion" [Denn76a][11]. Under the "L=S criterion", the load is regulated so that the average lifetime of the system is at least as large as the swap (access) time of the paging device. A detailed analysis and a comparison of the above two criteria have been presented in [Denn76b]. More recently, Blake [Blak82] applied the discrete optimal control technique to load control in a virtual memory system. Using a new feedback parameter, called the **thrashing level**[12], Blake showed how to formulate load control as a discrete control problem, which can then be solved by a dynamic programming technique. The advantage of this method is that its run-time overhead is low since the decision table for the control mechanism can be calculated in advance.

A predictive load controller activates a new process only if the activation will not overload the system. It makes such a prediction by analyzing the information supplied by the memory policy or other reliable sources. An example of a predictive load control is the "knee criterion" [Denn75a], which requires that each process be operating under the **knee**[13] of its lifetime curve. In other words, a process is given a resident set whose size is

---

[11] An early version was proposed by Chamberlin et al. [Cham73] which suggested that the average system page fault rate **U** be kept between limits $\mathbf{U}_{min}$ and $\mathbf{U}_{max}$.

[12] The thrashing level at time t is defined as: THL(t)=VMQ(t)/MPL(t) where VMQ(t) is the length of the virtual memory (paging device) queue and MPL(t) is the degree of multi-programming at time t.

[13] The **knee** of a lifetime curve is defined as the highest point of tangency between a line from the origin and the curve [Denn75b]. The space-time (time integration of the resident set) of a process is minimized when it operates under the knee [Denn78a].

the same as the "knee". The resident set sizes thus determined form a basis for a predictive load control: the activation of a process is delayed if its resident set size exceeds the number of available page frames. What this amounts to is a WS algorithm with multiple window sizes, one for each process. The implied complexity makes this approach impractical. However, it has been found in a number of experiments that one global window size suffices to come within 10 percent of minimum in the space-time of all the programs [Denn78a]. Thus, the one-parameter (one-window-size) WS algorithm can approximate a multi-parameter WS algorithm with only a minor loss of efficiency. In this respect, the one-parameter WS algorithm is a practical implementation of the "knee criterion". A predictive load control is inherently more stable than a feedback load control because the former can prevent overload, whereas the latter can only response to overload after the condition has been detected.

3.2. Buffer Management for Database Systems

Many early studies of database buffer management focused on the double paging problem[14], which arises from managing a buffer pool on top of a virtual memory system [Fern78] [Lang77] [Sher76a] [Sher76b] [Tuel76]. Denning made a comment on these studies [Denn80]:

> The double paging problem is the consequence of a flaw in the architecture of the computer; it is not an interesting subject of memory management research.

Although we do not totally agree with such a strong statement, managing a buffer pool on top of a virtual memory system does not appear particularly suitable either.

---

[14] The double paging anomaly [Gold74] was initially discovered in paged operating systems (e.g. IBM's OS/VS2) which were running under a paged virtual machine system (e.g. VM/370).

Another focus of research in this area is to find buffer management policies that "understand" database systems and know how to exploit the predictability of database reference behavior. As Stonebraker pointed out [Ston81], conventional algorithms, although effective for virtual memory systems, are not necessarily suitable for database systems. Several buffer management algorithms for database systems have been proposed in the past. We shall examine some of these algorithms in this section.

3.2.1. Domain Separation Algorithm

Consider a query that randomly accesses records through a B^+-tree index. The root page of the B^+-tree is obviously more important than a data page since it is accessed with every record retrieval. Based on this observation, Reiter [Reit76] proposed a buffer management algorithm, called the **domain separation** (DS) algorithm, in which pages are classified into types. Each type of pages is separately managed in an associated domain of buffers. When a page of a certain type is needed, a buffer is allocated from the corresponding domain. If none are available for some reason, e.g. all the buffers in that domain have I/O in progress, a buffer is borrowed from another domain. Buffers inside each domain are managed using the LRU discipline. Reiter suggested a simple page type assignment scheme: assign one domain to each non-leaf level of the B-tree structure, and one to the leaf level together with the data. Empirical data[15] showed that this DS algorithm out-performed the LRU algorithm by 8 to 10 percent in throughput.

Using the terminology for virtual memory systems, the DS algorithm is a multi-class LRU algorithm with class V2 memory partitioning and no provision for load control. This

[15] In Reiter's simulation experiments, a shared buffer pool and a workload consisting of 8 concurrent users were assumed.

characterization of the DS algorithm reveals the weaknesses of the algorithm. First of all, the concept of domain is static, and the algorithm fails to reflect the dynamics of page references because the importance of a page may vary in different queries. It is obviously desirable to keep a data page resident when it is being repeatedly accessed in a nested loops join. However, this is not the case when the same page is being accessed in a sequential scan. Second, the DS algorithm ignores the relative importance between different types of pages. An index page will be over-written by another incoming index page under the DS algorithm, although the index page is potentially more important than a data page in another domain. Memory partitioning is another potential problem. Partitioning buffers according to domains, rather than queries, can not prevent interference among competing queries. For example, pages that are important to a query may be forced out when another query is doing a fast sequential scan that touches a lot of pages which will not be re-used. Lastly, a separate mechanism needs to be in to prevent thrashing since the DS algorithm has no built-in facilities for load control.

Several extensions to the DS algorithm have been proposed. The group LRU (GLRU) algorithm, proposed by Hawthorn [Nybe84], is similar to DS, except that there is a fixed priority ranking among different groups (domains). A search for a free buffer always starts from the group with the lowest priority. Another alternative, presented by Effelsberg and Haerder [Effe84], is to dynamically vary the size of each domain using a WS-like partitioning scheme. Under this scheme, pages in domain i which have been referenced in the last τ_i references are exempt from replacement consideration. The "working set" of each domain may grow or shrink depending on the reference behavior of the user queries. Although empirical data indicated that dynamic domain partitioning can reduce the number of page faults (of the system) over static domain partitioning, Effelsberg and Haerder concluded that

there is no convincing evidence that page-type-oriented schemes[16] are distinctly superior to global algorithms such as LRU and CLOCK.

3.2.2. "New" Algorithm

In a study to find a better buffer management algorithm for INGRES [Ston76], Kaplan [Kapl80] made two observations about the reference patterns of queries. First, the priority to be given to a page is not a property of the page itself, but of the relation to which it belongs. Second, each relation needs a "working set". Based on these observations, Kaplan designed an algorithm, called the "new" algorithm, in which the buffer pool is subdivided and allocated on a per-relation basis. In this "new" algorithm, each active relation is assigned a resident set which is initially empty. The resident sets of all active relations are linked in a priority list with a global free list at the top. When a page fault occurs, a search is initiated from the top of the priority list until a suitable buffer is found. The faulting page is then brought into the buffer and added to the resident set of the relation. The MRU discipline is employed within each relation. However, each relation is entitled to one active buffer which is exempt from replacement consideration. The ordering of relations is determined, and may be adjusted subsequently, by a set of heuristics. A relation is placed near the top if its pages are unlikely to be re-used. Otherwise, the relation is protected by being placed near the bottom. Results from Kaplan's simulation experiments suggested that the "new" algorithm performed much better than the UNIX[17] buffer manager normally used by INGRES. However, in a trial implementation [Ston82], the "new" algorithm failed to improve the performance of IFS-INGRES, an experimental version of INGRES. One factor

[16] The DS algorithm is called a page-type-oriented buffer allocation scheme in [Effe84].

[17] UNIX is a Trademark of Bell Laboratories.

that accounted for the failure of the "new" algorithm in the IFS-INGRES experiment was that the "new" algorithm was modified and applied to the management of the buffers in user processes rather than the system buffers; hence the potential data sharing between successive queries was lost. Furthermore, the standard IFS-INGRES did not use the UNIX systems buffers. A modified LRU algorithm, similar to the DS algorithm, was used by the standard IFS-INGRES to manage the buffer space of the IFS-INGRES process.

The "new" algorithm is essentially a multi-class MRU replacement algorithm for a single-user environment[18]. The algorithm presented a new approach to buffer management, an approach that tracks the locality of a query through relations. However, the algorithm itself has several weak points. The use of MRU is justifiable only in limited cases. The rules suggested by Kaplan for arranging the order of relations on the priority list were based on simple intuitions; further justification and extensions are needed[19]. Furthermore, under high memory contention, searching through a priority list for a free buffer can be expensive. Extending the "new" algorithm to a multi-user environment presents additional problems. Although Kaplan claimed that such an extension is straightforward, it is not at all clear how to set up a priority among relations from different queries that are running concurrently.

3.2.3. Hot Set Algorithm

The **hot set** algorithm [Sacc82] is based on the hot set model described in the previous chapter. It was proposed as an alternative to the global LRU (actually CLOCK) algorithm currently used in System R. In the hot set algorithm, each query is provided with a separate

---

[18] This was appropriate for INGRES since INGRES was structured as user processes running on top of the UNIX operating system, which did not support sharing of data space among processes.

[19] Even Kaplan admitted that "the best way to carry out this reordering of the relation descriptors has not yet crystallized".

list of buffers that are managed using an LRU discipline. The number of buffers each query is entitled to is predicted according to the hot set model. That is, a query is given a local buffer pool of size equal to its **hot set size**. A new query is allowed to enter the system if its hot set size does not exceed the available buffer space. Thus, the hot set algorithm is a local LRU algorithm with imbalanced memory partition and predictive load control.

As mentioned in the previous chapter, the use of LRU in the hot set model lacks a logical justification. There exist cases where LRU is the worse possible discipline under tight memory constraints. The hot set algorithm avoids this problem by always allocating enough memory to ensure that references to different data structures within a query will not interfere with one another. Thus it tends to over-allocate memory, which implies that memory may be under-utilized. Another related problem is that there are reference patterns in which LRU does perform well and yet is unnecessary since another discipline with a lower overhead can perform equally well. Lastly, the hot set algorithm can not respond well to phase transitions of queries since the algorithm allocates the maximum amount of buffers ever needed by a query[20].

3.3. QLS - A Buffer Management Algorithm Based on the QLSM

In the previous chapter, the regularity and predictability of query reference behavior have been illustrated through the query locality set model. It is advantageous to integrate such information into a database buffer manager. To explore the potential of this approach, we have designed a new buffer management algorithm, which we call the Query Locality Set (QLS) algorithm.

---

[20] This problem can be alleviated by sub-dividing a query into a number of sub-queries and adjusting memory allocation at the beginning of each sub-query.

In the QLS algorithm, buffers are allocated and managed on a per file instance[21] basis (Figure 3.5). The set of buffered pages associated with a file instance is referred to as its **locality set**. Each locality set is separately managed by a discipline selected according to the intended usage of the file instance. Active instances of the same file are given different buffer pools which are independently managed. However, as we will explain later, all these file instances share the same copy of a buffered page whenever possible through a global table mechanism. If a buffer contains a page that does not belong to any locality set, the

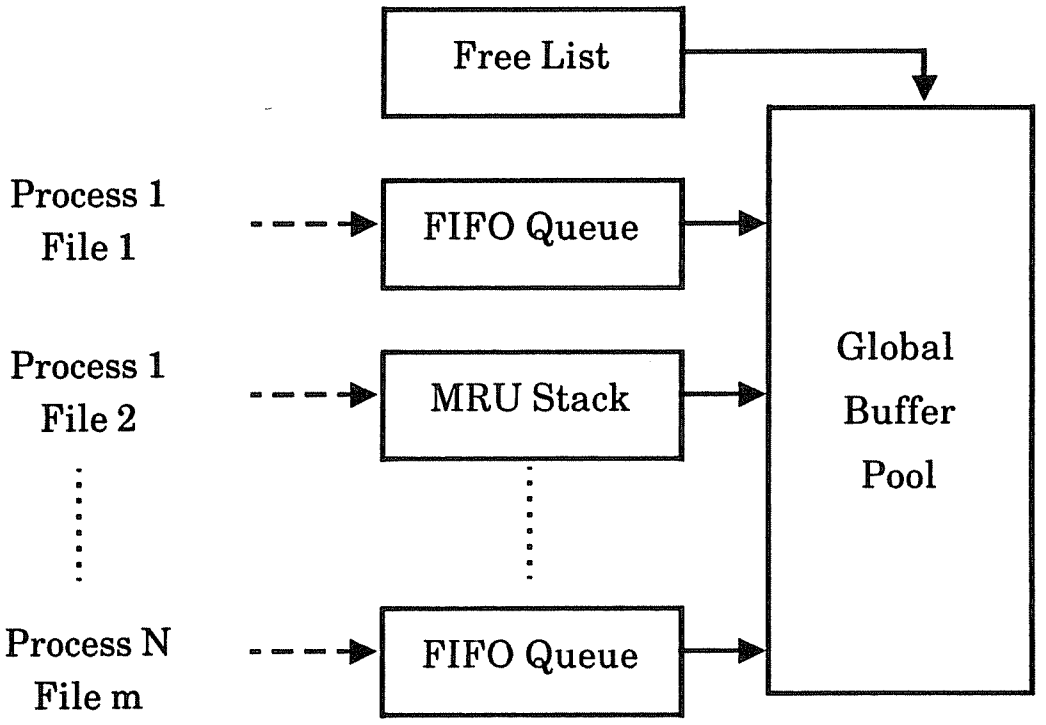

Figure 3.5 QLS Buffer Organization

buffer is placed on a global free list. For simplicity of implementation, we restrict that a page in the buffer can belong to at most one locality set. A file instance is considered the owner of all the pages in its locality set. To allow for data sharing among concurrent queries, all the buffers in memory are also accessible through a global buffer table. To simplify the description of the algorithm, we shall use the following notation:

N, the total number of buffers (page frames) in the system;

$\mathbf{l_{ij}}$, the maximum number of buffers that can be allocated to file instance **j** of query **i**;

$\mathbf{r_{ij}}$, the number of buffers allocated to file instance **j** of query **i**.

Note that **l** is the desired size for a locality set, while **r** is the actual size of the locality set.

At start up time, QLS initializes the global table and links all the buffers in the system together on the global free list. When a file is opened, its associated locality set size and replacement policy are given to the buffer manager. An empty locality set is then initialized for the file instance. The two control variables **r** and **l** associated with the file instance are initialized to 0 and the given locality set size, respectively.

When a page is requested by a query, the global table is searched first, and then the associated locality set is adjusted. There are three possible cases:

(1) The page is found in both the global table and the locality set:

In this case, only the usage statistics need to be updated if necessary as determined by the local replacement policy.

(2) The page is found in the global table but not in the locality set:

If the page already has an owner, the page is simply given to the requesting query and no further actions are required. Otherwise, the page is added to the locality set of the

file instance, and r is incremented by one. Now if r > l, a page is chosen and released according to the local replacement policy, and r is set back to l. Usage statistics are updated as required by the local replacement policy.

(3) The page is not in memory:

A disk read is scheduled to bring the page from disk into a buffer allocated from the global free list. After the page is brought into memory, proceed as in case 2.

Note that the local replacement policies associated with file instances do not cause actual swapping of pages. Their real purpose is to maintain the image of a query's "working set". Disk reads and writes are issued by the mechanism that maintains the global table and the global free list.

The load controller is activated when a file is opened or closed. Right after a file is opened, the load controller checks to see if $\sum_i \sum_j l_{ij} < N$ for all active queries i and their file instances j. (Note that the inequality guarantees that a free buffer is always available when a page fault occurs.) If the condition is met, the query is allowed to proceed; otherwise, it is suspended and placed at the front of the waiting queue. Buffers associated with a suspended query are released to avoid possible deadlock situations. When a file is closed, buffers associated with its locality set are released. The load controller then activates the first query on the waiting queue if this will not cause the above condition to be violated.

QLS can be viewed as a combination of the WS algorithm and Kaplan's "new" algorithm in the sense that the locality set associated with each file instance is similar to the working set associated with each process. However, the size of a locality set is determined in advance, and need not be re-calculated as the execution of the query progresses. This predictive nature of QLS is close to that of the hot set algorithm. But, unlike the hot set

algorithm which allocates buffers statically, QLS uses a dynamic partitioning scheme, in which the total number of buffers assigned to a query may vary as files are open or closed. To summarize, QLS may be considered a local hybrid algorithm with class W memory partitioning and predictive load control. What remains to be described is the selection of local replacement policies and the sizes for the locality sets. Using the query locality set model as a framework, we shall demonstrate how local policies for locality sets can be determined.

Sequential References:

As mentioned in the previous chapter, there are three types of sequential reference patterns: straight sequential, clustered sequential, and looping sequential.

(1) Straight Sequential Reference:

For a straight sequential reference, in which each page is brought into memory only once, the locality set size is obviously 1. When a requested page is not found in the buffer, the page is fetched from disk and overwrites whatever is in the buffer. For convenience, we shall call this simple replacement policy the SB (Single Buffer) algorithm.

(2) Clustered Sequential Reference:

During the evaluation of a merge join, a clustered sequential reference may be observed on the inner relation. Clearly, it is desirable to keep the members of a cluster (i.e. records with the same key) in memory. Thus, a proper size for the locality set is the size of the largest cluster divided by the blocking factor (i.e. the number of records per page). To account for the worst case boundary conditions, one additional buffer may be needed. Provided that enough space is allocated, FIFO and LRU both yield the minimum number of page faults. If run-time overhead is taken into con-

sideration, FIFO is a better policy than LRU for a clustered sequential reference.

(3) Looping Sequential Reference:

When a file is being repeatedly scanned in a looping sequential reference, MRU is the best replacement algorithm[21] as discussed previously. It is beneficial to give the file as many buffers as possible, up to the point where the entire file can fit in memory. Hence the locality set size corresponds to the total number of pages in the file.

Random References:

There are two types of random reference patterns identified by the query locality set model: independent random references, and clustered random references.

(1) Independent Random Reference:

When the records of a file are being randomly accessed, say through a hash table, the choice of a replacement algorithm is immaterial since all the algorithms perform equally well [King71] [Gele73]. What is left to be determined is a proper size for the locality set. Yao's formula [Yao77] provides an estimate of the total number of pages referenced **b** in a series of **k** random record accesses. This sets up an (estimated) upper bound on the locality set size. In the cases where page references are sparse, there is no need to keep a page in memory after its initial reference. Thus, there are two reasonable sizes for the locality set, 1 and **b**, depending on the likelihood that each page is re-referenced. We define $\mathbf{r} = \frac{\mathbf{k}-\mathbf{b}}{\mathbf{b}}$ as the **residual value** of the pages in a

---

[21] It is perhaps worth noting that the performance of LIFO is close to that of MRU in this particular case. The difference in missing page rate **M** is

$$\mathbf{M(LIFO)-M(MRU)} = \frac{\mathbf{p-r+1}}{\mathbf{p}} - \frac{\mathbf{p-r}}{\mathbf{p-1}} = \frac{\mathbf{r-1}}{\mathbf{p^*(p-1)}}$$

where **p** is the number of pages in the file and **r** is the number of buffers.

file. The locality set size is 1 if $\mathbf{r} \leq \beta$, and **b** otherwise. Thus, β is the threshold above which pages are considered to have a high probability of being re-referenced. Due to the recursive nature of Yao's formula, its computational cost may be too expensive. If this is a concern, there are formulas that give reasonable approximations to Yao's formula with less computational overhead [Whan83].

(2) Clustered Random Reference:

The reference pattern of a clustered random reference is similar to that of a clustered sequential reference. The only difference is, in a clustered random reference, records in a "cluster" are not physically adjacent, but randomly distributed over the file. The locality set size in this case can be approximated by the size of the largest cluster[22].

Hierarchical References:

The remaining reference patterns are hierarchical references to indices. The first three types of hierarchical references are straightforward and will be discussed together. However, the case of looping hierarchical references is more complex and will be treated separately.

(1) Straight Hierarchical, H/SS, and H/CS References:

In a straight hierarchical reference or an H/SS reference, an index page is traversed only once. Thus, the SB algorithm for the sequential reference is also adequate in these two cases. Similarly, the discussion of the clustered sequential reference is applicable to H/CS reference, except that each member in a cluster is now a key-pointer pair rather than a data record.

---

[22] A more accurate estimate can be derived by applying Yao's formula to calculate the number of distinct pages referenced in a cluster.

(2) Looping Hierarchical Reference:

In a looping hierarchical reference, an index is repeatedly traversed from the root to the leaf level. In such a hierarchical reference, pages near the root are more likely to be accessed than those at the bottom [Reit76]. Consider a tree of height **h** and with a fan-out factor **f**. Without loss of generality, let us assume that the tree is complete, i.e. each non-leaf node has **f** sons. During each traversal from the root at level 0 to a leaf at level **h**, one out of the f^i pages at level i is referenced. Therefore pages at an upper level (which are closer to the root) are more important than those at a lower level. Consequently, an ideal replacement algorithm should keep the (accessed portion of the) upper levels of a tree resident and multiplex the rest of the pages using a scratch buffer. As to how many levels to keep in memory, we can again resort to the concept of "residual value" which we have defined for the random reference pattern. Let $\mathbf{b_i}$ be the number of pages accessed at level i as estimated by Yao's formula. The size of the locality set can be approximated by $(1+\sum_{i=1}^{j} \mathbf{b_i})+1$, where **j** is the largest **i** such that $\frac{\mathbf{k}-\mathbf{b_i}}{\mathbf{b_i}} > \beta$. In many cases, the root is perhaps the only page worth keeping in memory, since the fan-out of an index page is usually high. If this is true, the LIFO algorithm with few buffers may deliver a reasonable level of performance as the root is alway kept in memory.

In this section, we have presented a new buffer management algorithm, the QLS algorithm, for database systems. Using the file instance as the basic unit for buffer allocation and management, the QLS algorithm implements a memory policy that is tailored to the individual needs of the queries. However, despite its theoretical support from the query locality model, the efficiency of QLS remains to be evaluated. In the next chapter, we shall

present a performance evaluation of the QLS algorithm and compare its performance with that of other algorithms.

CHAPTER 4

EVALUATION OF BUFFER MANAGEMENT ALGORITHMS

Evaluating a computer system through direct measurement is both accurate and creditable [Saue81]. Benchmarking, for example, has recently been applied to the evaluation of several database systems [Bitt83] [Bora84]. The problem with direct measurement is that it is usually a computationally expensive procedure, and it is only possible when the system to be evaluated is already operational. Analytic modeling is a cost-effective alternative for estimating the performance of computer systems. However, accuracy is often traded for simplicity of the equations in order to keep the solution tractable. Since our objective is to compare the performance of different buffer management algorithms in a multi-user environment, neither of these two approaches seems appropriate. Thus, simulation was chosen for evaluating the performance of a number of selected buffer management algorithms in our study.

In this chapter, we shall discuss how our simulation experiments were conducted as well as the results of the experiments. This chapter is organized into five sections. The simulation model used in the experiments and the six selected buffer management algorithms are described in sections 1 and 2, respectively. Section 3 evaluates the performance of the algorithms using the empirical data obtained from the simulation experiments. Complementary to these empirical results is a simple cost analysis of some of the algorithms derived in section 4. Finally, the results of the evaluation are summarized in the last section of this chapter.

4.1. Performance Evaluation Methodology

There are two types of simulations that are widely used [Sher73]: **trace-driven simulations** which are driven by traces recorded from a real system, and **distribution-driven simulations** in which events are generated by a random process with a certain stochastic structure. A trace-driven model has several advantages, including creditability and fine workload characterization which enables subtle correlations of events to be preserved. However, selecting a "representative" workload is difficult in many cases. Furthermore, it is hard to characterize the interference and correlation between concurrent activities in a multi-user environment so that the trace data can be properly treated in an altered model with a different configuration. To avoid these problems, we employed a hybrid simulation model which combines features of both trace-driven and distribution-driven models. In this hybrid model, the behavior of each individual query is described by a trace string, and the system workload is dynamically synthesized by merging the trace strings of the concurrent queries.

Another component of our simulation model is a simulator for a database system which manages three important resources: a CPU, an I/O device, and memory. When a new query arrives, a load controller (if it exists) decides, depending on the availability of resources at the time, whether to activate or delay the query. After a query is activated, it then circulates in a loop between the CPU and the I/O device, competing for resources until it finishes its execution. After a query terminates, another new query is generated by the workload model. An active query, however, may be temporarily suspended by the load controller when the condition of overloading is detected.

The page fault rate has frequently been used to measure the performance of a memory policy. However, minimizing the number of page faults in a multiprogrammed environment

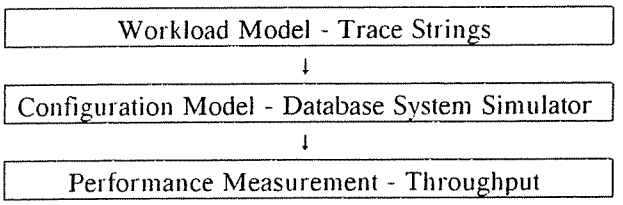

Figure 4.1 A Simulation Model for Database Systems

does not guarantee optimal system behavior. Although a time-space product is usually a better criterion in this case, it does not relate directly to the overall performance of the system. Thus, throughput, measured as the average number of queries completed per second, has been chosen as the performance metric in our study. In the following subsections, we shall discuss in detail the three aspects of the simulation model: workload characterization, the configuration model and performance measurement (Figure 4.1).

4.1.1. Workload Synthesis

The first step in developing a workload was to obtain single-query trace strings by running queries on the **Wisconsin Storage System** (WiSS) [Chou83]. While WiSS supports a number of storage structures and their related scanning operations, WiSS does not directly support a high-level query interface; hence, the test queries were "hand coded". The **Wisconsin Database** [Bitt83], a synthetic database with a well-defined distribution structure, was used in the experiments. Several types of events were recorded (with accurate timing information) during the execution of each query, including page accesses, disk I/O's, and file operations (i.e. the opening and closing of files).

A trace string can be viewed as an array of event records, each of which has a tag field that identifies the type of the event. There are six important event types: **page read, page**

write, **disk read**, **disk write**, **file open**, and **file close**. Their corresponding record formats are:

- **Page read** and **write**

page read / write	file ID	page ID	time

- **Disk read** and **write**

disk read / write	file ID	page ID	time

- **File open**

file open	file ID	locality set size	replacement policy

- **File close**

file close	file ID

Disk reads or **writes** come in pairs, each of which brackets the time interval of a disk operation. The times originally recorded were real (elapsed) times of the system. For reasons to be explained later, **disk read** and **write** events were removed from the trace strings, and the times of other events were adjusted accordingly. In essence, the times in a modified trace string reflect the virtual (or CPU) times of a query.

Since accurate timing (on the order of 100 microseconds) is required to record the events at such a detailed level, tracing was done on a dedicated VAX[1]-11/750 [Digi80] under a very simple operating system kernel designed for the CRYSTAL multi-computer system [DeWi84b]. To reduce the overhead of obtaining the trace strings, events were recorded in main memory and written to a WiSS file after tracing had ended.

---

[1] VAX is a Trademark of Digital Equipment Corporation.

In the methodology proposed by Boral and DeWitt [Bora84] for evaluating the performance of database systems in a multi-user environment, three important factors were identified: the number of concurrent queries[2], the degree of data sharing, and the query mix. The number of concurrent queries in each of our simulation runs is a fixed constant in the range of 1 to 32. To study the effects of data sharing, 32 copies of the test database were replicated. Each copy was stored in a separate portion of the disk. Three levels of data sharing were defined according to the average number of concurrent queries accessing a copy of the database:

(1) full sharing: all queries access the same copy of the database.

(2) half sharing: two queries share a copy of the database.

(3) no sharing: every query has its own copy of the database.

The approach to query mix selection used in [Bora84] is based on a dichotomy of the consumption of two system resources, CPU cycles and disk bandwidth. We extended their query classification scheme by considerating one additional resource, main memory (Table 4.1)[3]. After some initial testing, six queries were chosen as the base queries for synthesizing multi-user workloads. The first two are simple selection queries, and the remaining are two-way join queries with a selection operation on one of the source relations. The CPU and disk consumptions of the queries were calculated from the single-query trace strings, and the corresponding memory requirements were estimated by the hot set model (Table 4.2). Table 4.3 contains a summary description of the queries.

---

[2] The term multiprogramming level (MPL) was used in [Bora84]. However, since it is desirable to distinguish the external workload condition from the internal degree of multiprogramming, "number of concurrent queries" (NCQ) is used here instead. Using our definitions, MPL $\leq$ NCQ under a buffer manager with load control.

[3] The types of queries with low CPU requirement and high memory requirement were not included since they are less likely to occur in practice.

Query Type	CPU Requirement	Disk Requirement	Memory Requirement
I	Low	Low	Low
II	Low	High	Low
III	High	Low	Low
IV	High	High	Low
V	High	Low	High
VI	High	High	High

Table 4.1 Query Classification

Query Number	CPU Usage (seconds)	Number of Disk Operations	Hot Set Size (4K-pages)
I	.53	17	3
II	.67	99	3
III	2.95	53	5
IV	3.09	120	5
V	3.47	55	17
VI	3.50	138	24

Table 4.2 Representative Queries

At simulation time, a multi-user workload is constructed by dynamically merging the single-query trace strings according to a given probability vector which describes the relative frequency of each query type. The trace string of an active query is read and processed by the CPU simulator, one event at a time, when the query is being served by the CPU. For a **page read** or **write** event, the CPU simulator advances the query's CPU time according to the time in the event record, and then it forwards the page request to the buffer manager. If the requested page is not found in the buffer, the query is blocked while the page is being fetched from the disk. The exact ordering of the events from the concurrent queries are determined by the behavior of the simulated system and the times recorded in the trace strings.

Query Number	Query Operations	Selectivity Factor	Access Path of Selection	Join Method	Access Path of Join
I	select(A)	1%	clustered index	-	-
II	select(B)	1%	non-clustered index	-	-
III	select(A) join B	2%	clustered index	index join	clustered index on B
IV	select(A') join B	10%	sequential scan	index join	non-clustered index on B
V	select(A) join B'	3%	clustered index	nested loops	sequential scan over B'
VI	select(A) join A'	4%	clustered index	hash join	hash on result of select(A)

A,B:10K tuples; A':1K tuples; B':300 tuples; 182 bytes per tuple.

Table 4.3 Description of Base Queries

4.1.2. Configuration Model

The general structure of the configuration model is similar to those used in [Opde74] and [Poti77] (Figure 4.2). Three hardware components are simulated in the model: a CPU, a disk, and a pool of buffers. A round-robin scheduler is used for allocating CPU cycles to competing queries. The CPU usage of each query is determined from the associated trace string, in which detailed timing information has been recorded. In this respect, the simulator's CPU has the characteristics of a VAX-11/750 CPU. The simulator's kernel schedules disk requests on a first-come-first-serve basis. In addition, an auxiliary disk queue is maintained for implementing delayed asynchronous writes, which are initiated only when the disk is about to idle.

The disk times recorded in the trace strings tend to be smaller than what they would be in a "real" environment for two reasons: (1) the database used in the tracing experiments

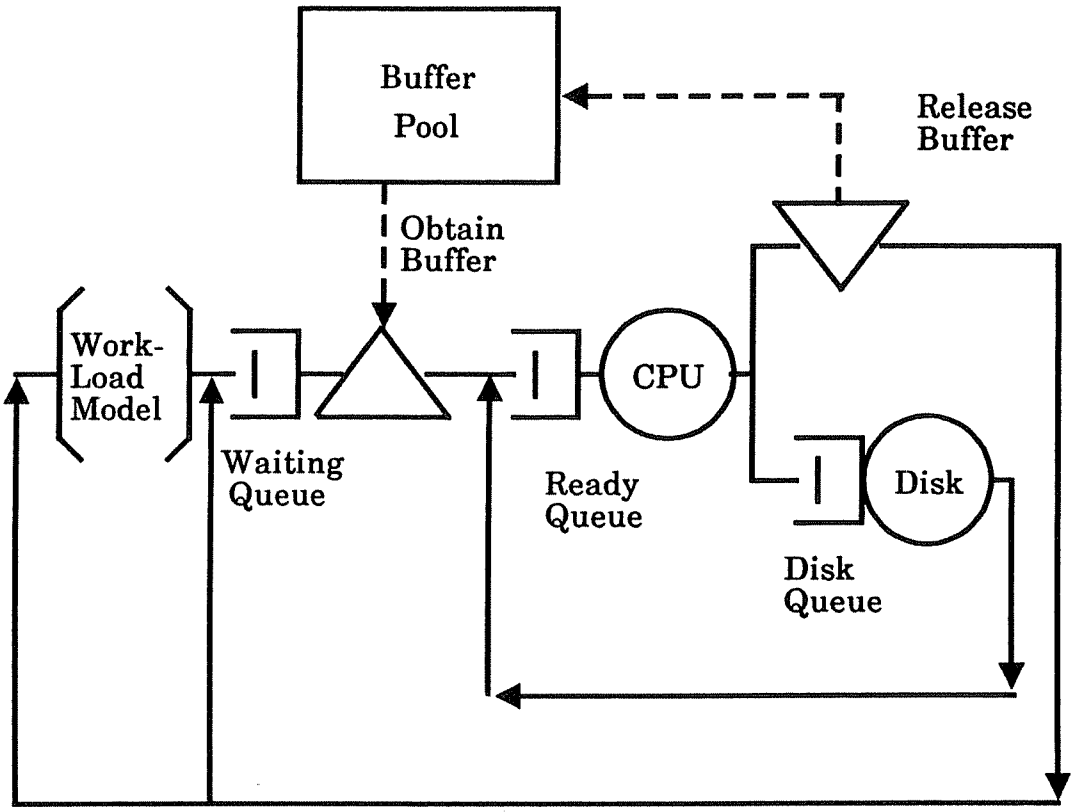

Figure 4.2 Simulation Model

was relatively small; and (2) disk arm movements are usually less frequent on a single-user system than in a multi-user environment. Furthermore, requests for disk operations are affected by the buffer management algorithm used. Therefore, the disk times recorded were replaced by times from a stochastic disk model in which a random process for disk head positions is assumed. In the disk simulator, the access time of a disk operation is calculated from the timing specifications of a Fujitsu Eagle disk drive [Fuji82]. On the average, it

takes about 27.6 ms to access a 4K page on an Eagle disk.

The buffer pool is under the control of the buffer manager, which uses one of several buffer management algorithms. However, the operating system can fix a buffer in memory to prevent replacement when an I/O operation is in progress. The size of the buffer pool for each simulation run is determined by the formula

$$8 \cdot \frac{\sum_i p_i t_i h_i}{\sum_i p_i t_i}$$

where p_i is the issuing probability of query type i, t_i and h_i are the CPU requirement and the hot set size of query type i, respectively. The intent was to saturate the memory at a load with eight concurrent queries so that the effect of overloading on performance can be observed under different buffer management algorithms.

To speed up the simulation runs, the simulator was coded in the C language [Kern78], and it uses the hardware queue instructions of the VAX [Digi81]. Maintaining several buffer management algorithms, each with its own version of the operating system, would be a tedious task. Therefore, the operating system in the simulator was designed to have a table-driven interface to the buffer manager so that it can be used for all of the buffer management algorithms.

4.1.3. Performance Measurements

As mentioned earlier, throughput is used as the performance metric for evaluating the performance of the buffer management algorithms. Due to the stochastic nature of the workload model, each simulation experiment has to run long enough to provide meaningful measurements. In addition, statistical analysis of the measurements is required to establish

the validity of the results. After some trial runs, our goal was set to limit each confidence interval [Bask73] to within ± 5% of the mean throughput at the 90% confidence level.

There are three alternatives for estimating confidence intervals: batch means, independent replications, and the regenerative method [Sarg76]. Although batch means is the least rigorous method of the three, it was chosen as it can easily be programmed into the simulator. The number of batches in each simulation run was set to 20. Analysis of the throughput measurements indicates that many of the confidence intervals fell within ± 1% of the mean throughput. For those experiments in which thrashing occurred, the length of a batch was extended to ensure that all confidence intervals were within 5% of the mean.

4.2. Buffer Management Algorithms

Six buffer management algorithms, divided into two groups, were included in the experiments. The first group consisted of three simple algorithms: RAND, FIFO, and CLOCK[4]. They were chosen because they are typical replacement algorithms and are easy to implement. It is interesting to compare their performance with that of the more sophisticated algorithms to see if the added complexity of these algorithms is warranted. Beside QLS, WS (the working set algorithm), and HOT (the hot set algorithm) were included in the second group. WS is one of the most efficient memory policies for virtual memory systems [Denn78a], so it is interesting to know how well it performs when applied to a database system. HOT was chosen to represent the class of algorithms that have previously been proposed for database systems, since it is more complete, both technically and theoretically, than other algorithms that we have examined. Also, we are unaware of previous performance

[4] LRU was not included since CLOCK provides similar performance at a lower implementation cost.

results for HOT, making such results interesting in their own right.

All the algorithms in the first group are global algorithms in the sense that the replacement discipline is applied globally to all the buffers in the system. Common to all three algorithms is a global table that contains, for each buffer:

(1) the identity of the residing page, and

(2) a flag indicating whether the buffer has an I/O operation in progress.

Additional data structures or flags may be needed depending on the individual algorithm. Implementations of RAND and FIFO are typical, and need no further explanation. The CLOCK algorithm used in the experiments gives preferential treatment to dirty pages, i.e. pages that have been modified. During the first round of a scan, an unreferenced dirty page is scheduled for write, whereas an unreferenced clean page is immediately chosen for replacement. If no suitable buffer is found in the first complete scan, dirty and clean pages are treated equally during the second scan. None of the three algorithms has a built-in facility for load control. However, we will investigate later how a load controller may be incorporated and what its effects are on the performance of these algorithms.

The algorithms in the second group are all local policies, in which replacement decisions are made within each local buffer pool. There is a local table associated with each query or file instance for maintaining its resident set. Buffers which do not belong to any resident set are placed in a global LRU list. To allow for data sharing among concurrent queries, a global table, similar to the one for the global algorithms, is also maintained by each of the local algorithms in the second group. When a page is requested, the global table is searched first, and then the appropriate local table is adjusted if necessary. As an optimization, an asynchronous write operation is scheduled whenever a dirty page is released back

to the global free list. All three algorithms in the second group base their load control on the (estimated) memory demands of the submitted queries. A new query is activated if there is sufficient free space left in the system. On the other hand, an active query is suspended when over-commitment of main memory has been detected. Buffers associated with a suspended process are released to avoid potential deadlocks. We adopted the deactivation rule implemented in the VMOS operating system [Foge74] in which the faulting process (i.e. the process that was asking for more memory) is chosen for suspension[5]. In the following, we shall discuss implementation decisions that are pertinent to each individual algorithm in the second group.

(1) The **working set algorithm**:

To make WS more competitive, a two-parameter WS algorithm was implemented. That is, each process is given one of two window sizes depending on which is more advantageous to it. The two window sizes, τ_1 = 10ms and τ_2 = 15ms, were determined from an analysis of working set functions on the single-query trace strings (see Appendix A). Instead of computing the working set of a query after each page access, the algorithm implemented re-calculates the working set only when the query encounters a page fault or has used up its current time quantum. This implementation is very close to the "ideal" WS algorithm since releasing pages not in the current working set is unnecessary while the query has control over the CPU.

(2) The **hot set algorithm**:

HOT was implemented according to the outline described in [Sacc82]. The hot set sizes associated with the base queries were hand-calculated according to the hot set

[5] We also implemented the deactivation rule suggested by Opderbeck and Chu [Opde74] which deactivates the process with the least accumulated CPU time. However, no noticeable differences in performance were observed.

model (see Table 4.2 above). They were then stored in a table which is accessible to the buffer manager at simulation time.

(3) The **QLS algorithm**:

The locality set size and the replacement policy for each file instance were manually determined. They were then passed (by the program that implemented the query) to the trace string recorder at file open times while the single-query trace strings were being recorded. At simulation time, the QLS algorithm uses the information recorded in the trace strings to determine the proper resident set size and replacement discipline for a file instance at the time the file is opened.

4.3. Simulation Results

In order to make a thorough evaluation of the buffer management algorithms, a number of parameters were included in the simulation:

(1) the buffer management algorithm,

(2) the number of concurrent queries,

(3) the query mix, and

(4) the degree of data sharing.

In addition, the effects of a feedback load controller on the performance of the three simpler algorithms were also included in the experiments.

To cover such a wide domain, several thousand simulation runs, which took a total of 3,000 VAX 11/750 CPU hours[6], were performed. The CRYSTAL multi-computer system [DeWi84b], on which we gathered the single-query trace strings, was used again for the simulation runs. On the average, five dedicated VAX 11/750's were simultaneously used

for the simulations.

For ease of discussion, we shall further organize this section into five sub-sections. The first sub-section evaluates the performance of the buffer mangement algorithms for the six query types. The effects of query mix and data sharing are examined in the second and third sub-sections, respectively. We then describe the implementation of a feedback load controller and examine its effectiveness in the fourth sub-section. Finally, the results of the simulation are summarized in the last part of this section.

4.3.1. Performance for Six Query Types

The purpose of the first set of experiments was to compare the performance of the different algorithms for the six query types. In these experiments, it was assumed that there is no data sharing among concurrent queries. The results are plotted in Figures 4.3 (a) through (f). The x axis is the number of concurrent queries and the y axis is the throughput of the system measured in queries per second. There are six curves in each graph, one for each of the buffer management algorithms.

The presence of thrashing for the three simple algorithms is evident[7]. A sharp degradation in performance can be observed in most cases. RAND and FIFO yielded the worst performance, although RAND is perhaps more stable than FIFO in the sense than its curve is slightly smoother than that of FIFO. Before severe thrashing occurred, CLOCK was generally better than both RAND and FIFO.

---

[6] This corresponds to 15,000 simulated hours, as the simulator runs about 5 times faster than the simulated system.

[7] This is why the data points for the three simple algorithms were gathered only up to 16 concurrent queries. It is very time-consuming to gather throughput measurements with a $\pm$ 5% confidence interval when the simulated system is trapped in a thrashing state.

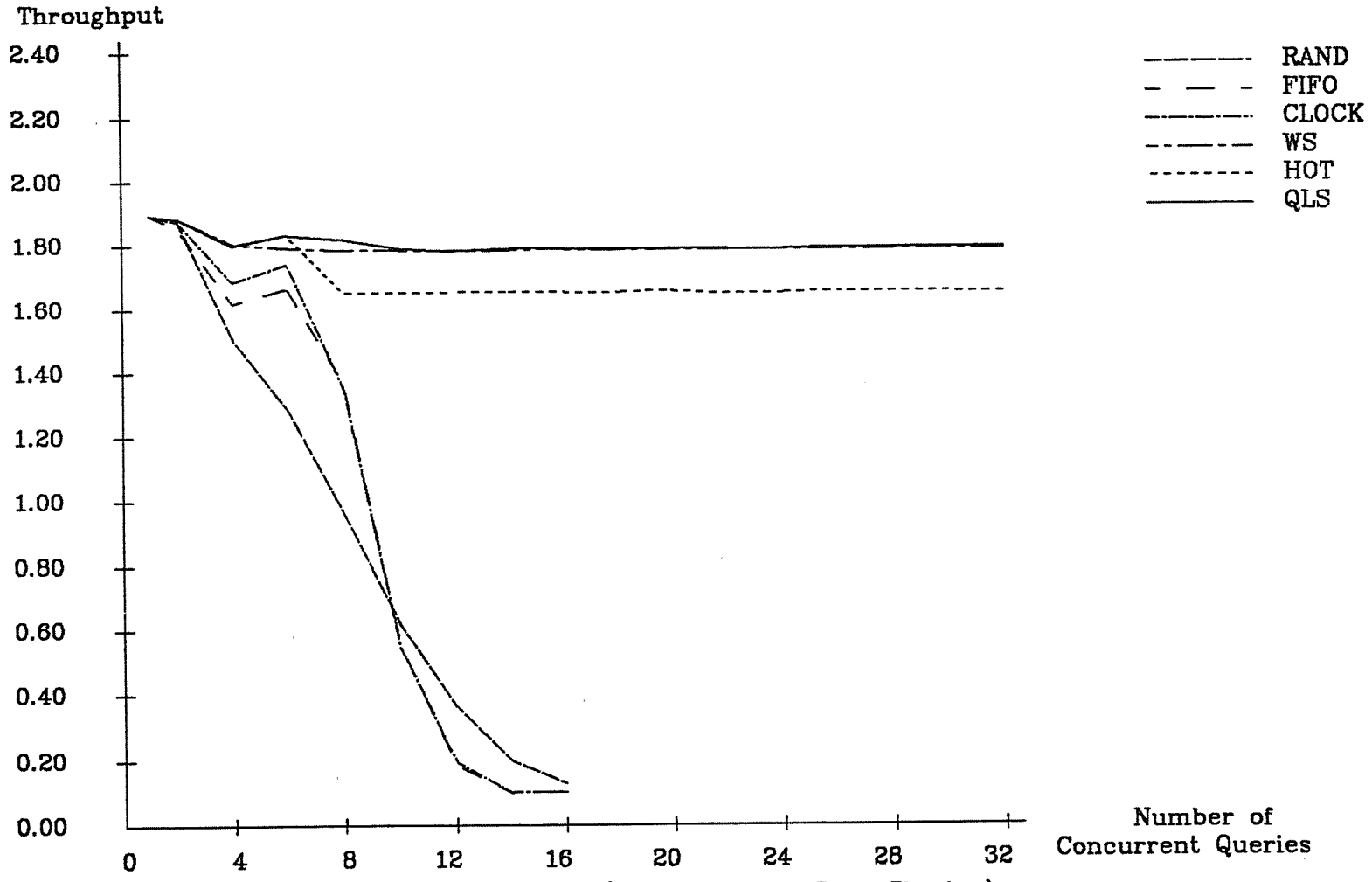

Figure 4.3 (a) Query Type I (25 Buffers, No Data Sharing)

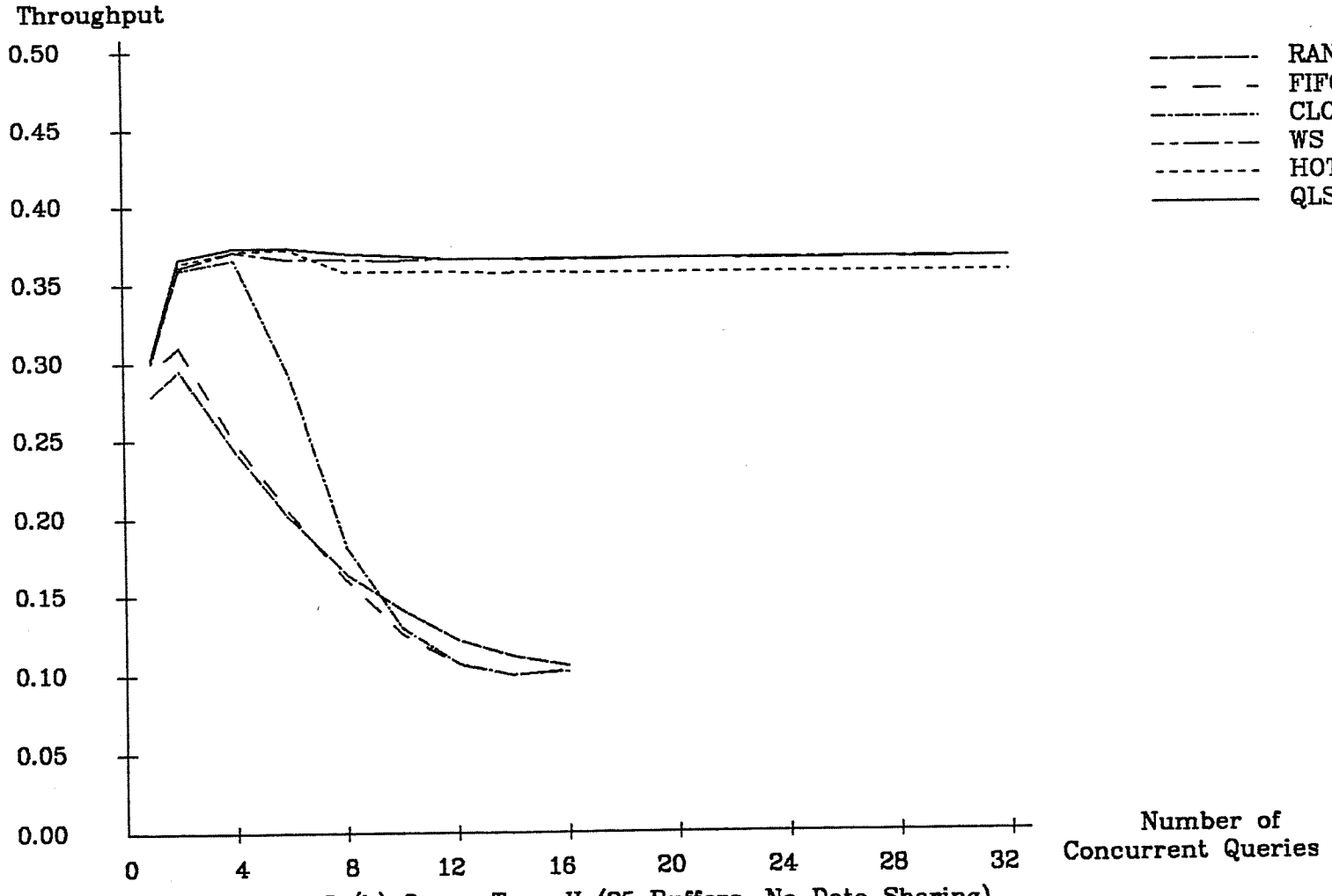

Figure 4.3 (b) Query Type II (25 Buffers, No Data Sharing)

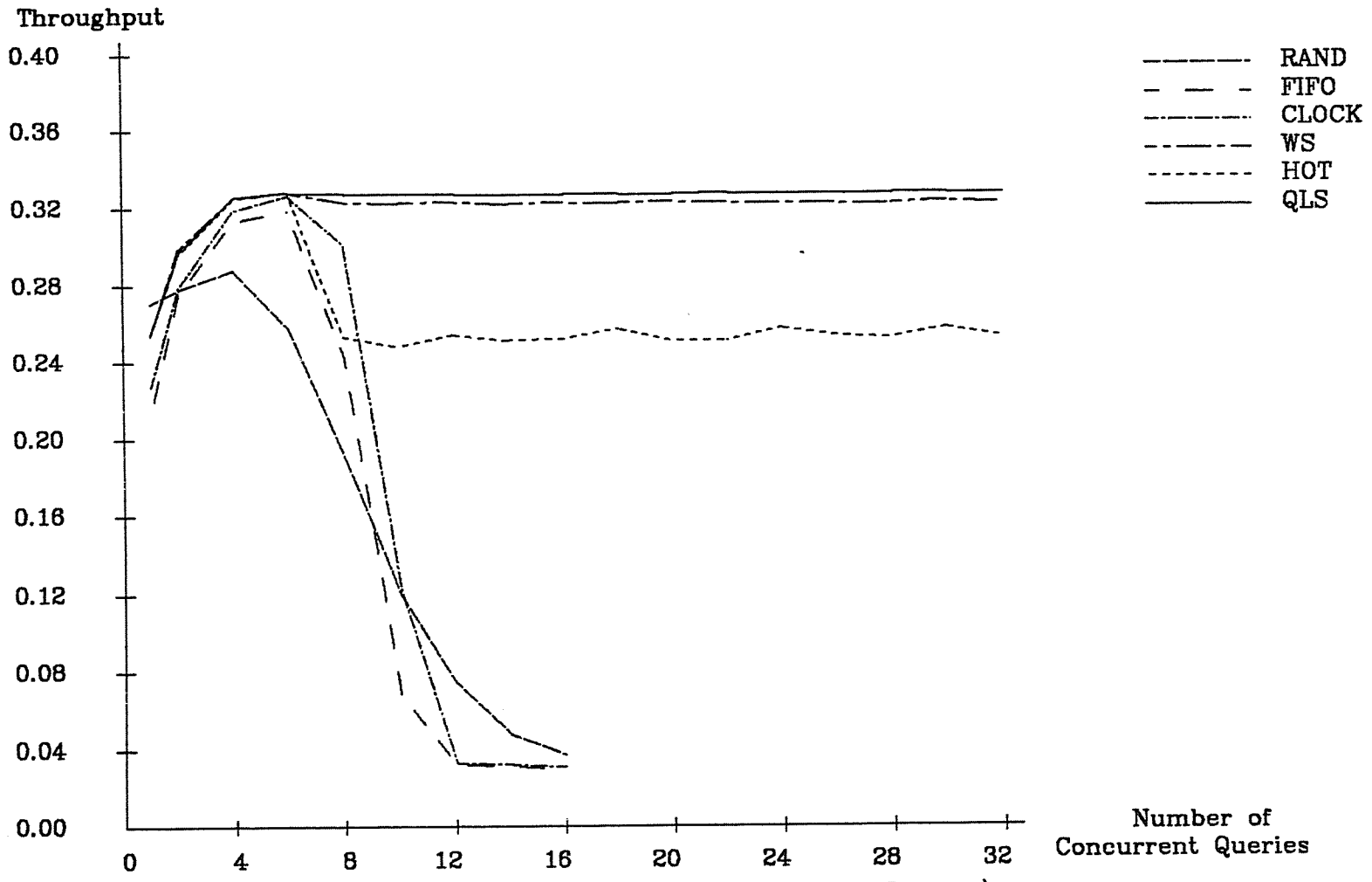

Figure 4.3 (c) Query Type III (40 Buffers, No Data Sharing)

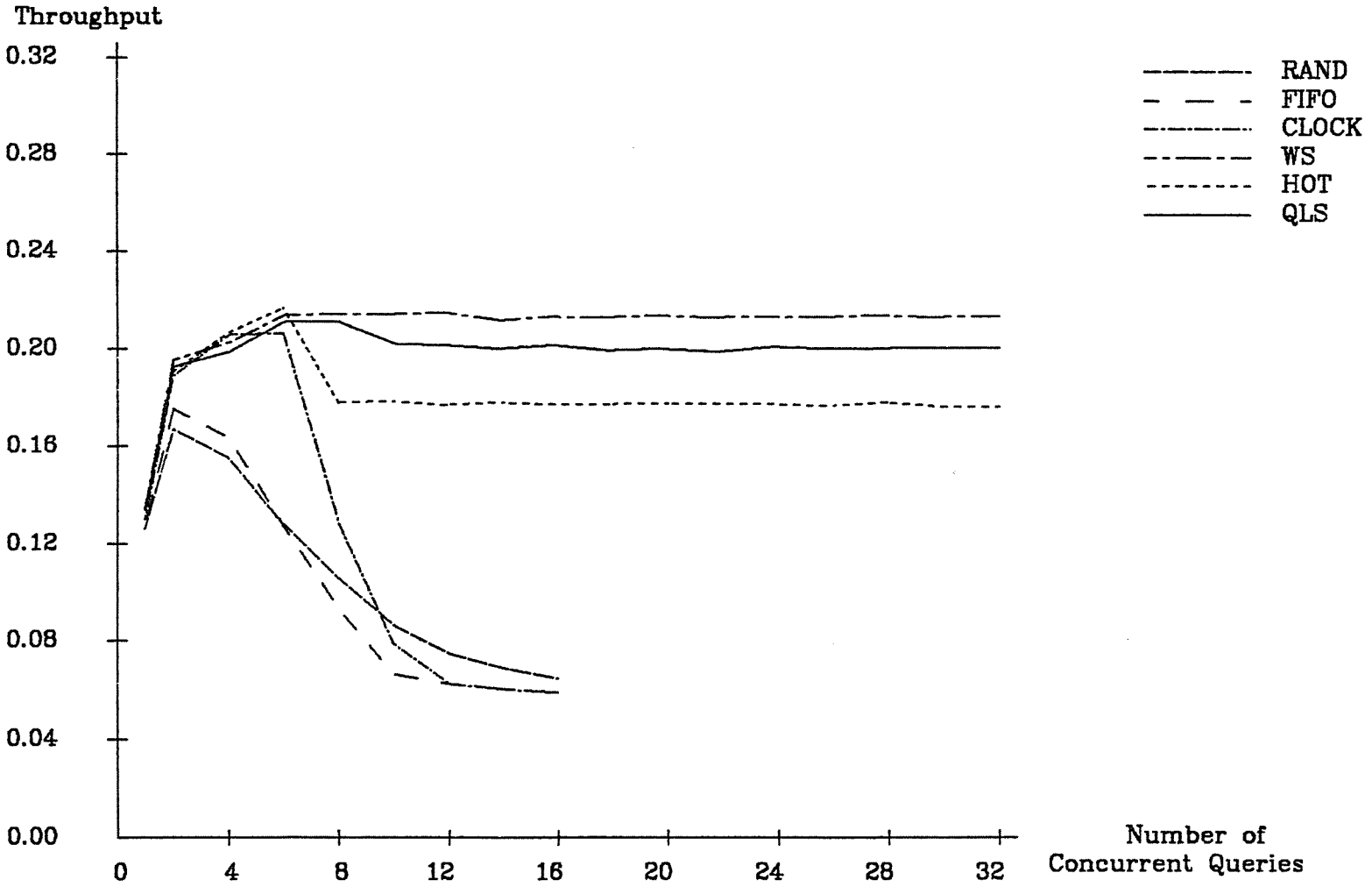

Figure 4.3 (d) Query Type IV (40 Buffers, No Data Sharing)

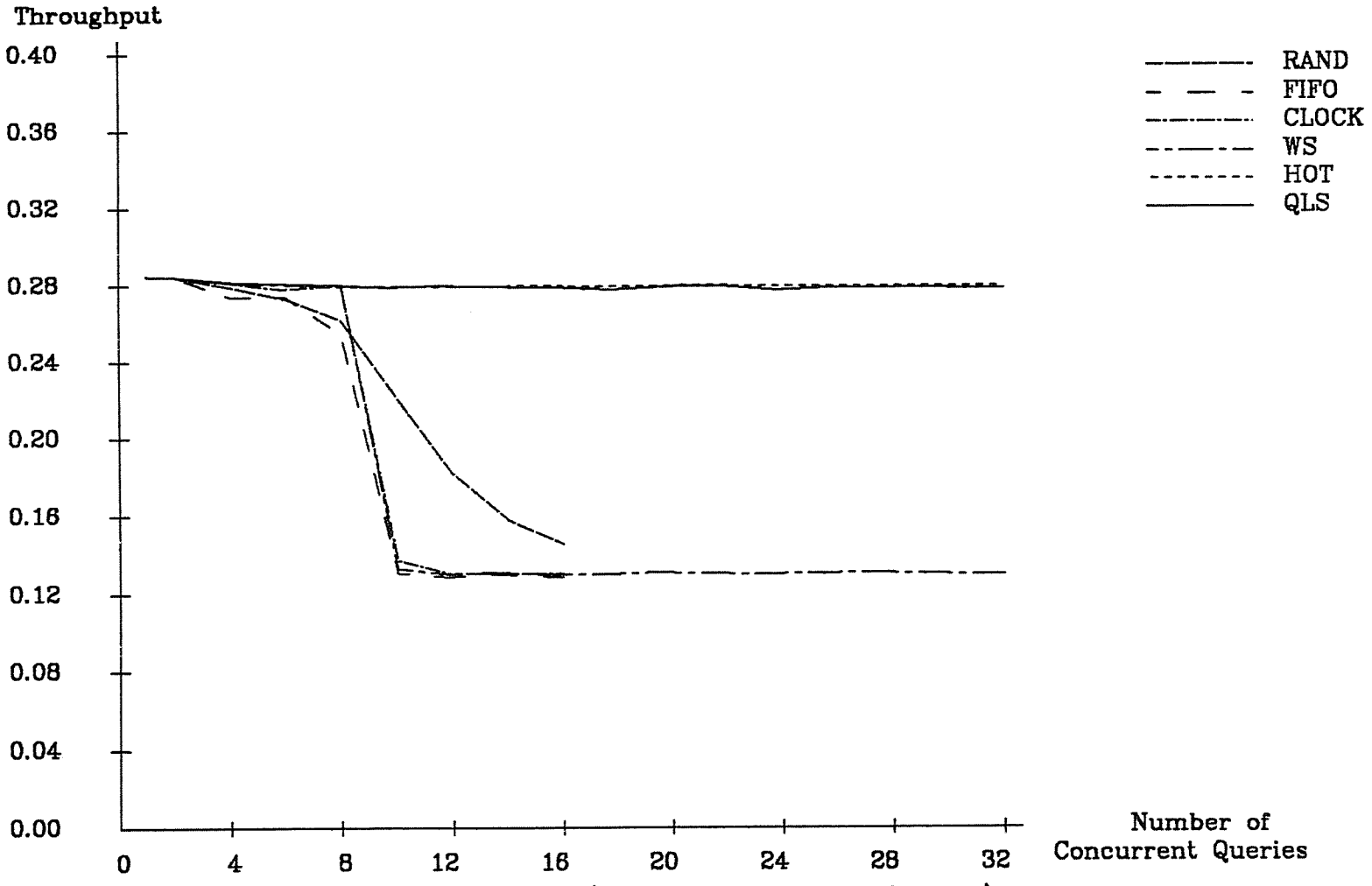

Figure 4.3 (e) Query Type V (140 Buffers, No Data Sharing)

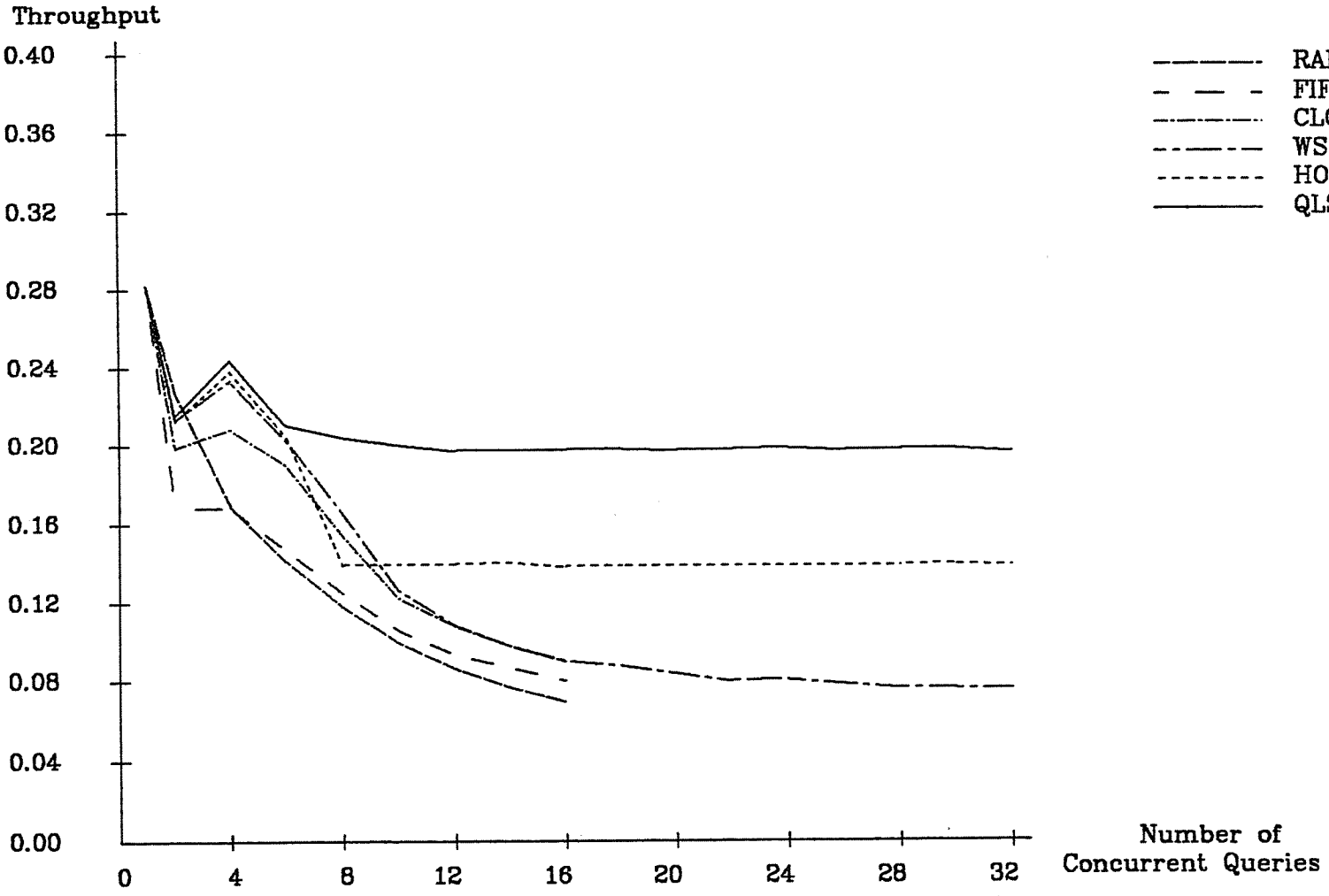

Figure 4.3 (f) Query Type VI (200 Buffers, No Data Sharing)

WS performed quite well for the first four queries. However, it began to thrash for queries V and VI. The reason is that the two window sizes chosen for WS captured only small localities and failed to include the large loops (on the inner relation of the join) in queries V and VI. Selecting a window size large enough to include the main loop of a join is rather complicated since it depends on the size of the inner relation, the cost of processing a tuple, and many other factors.

HOT did not thrash as WS did for queries V and VI. However, its performance was worse than that of WS and QLS for some queries, such as for query III. There are several possible reasons for this degradation:

(1) The storage allocation in HOT is static; hence buffers may be under-utilized during certain phases that require less space than what is allocated.

(2) The hot set model does not apply well to the intermediate files that are being generated during the execution of a query. For example, for a selection query with a low selectivity factor, a page for storing the selected tuples may be forced out by the LRU policy while a source page which is no longer in use is kept in memory because it has been more recently accessed.

(3) An implicit assumption of the hot set model is that the interface between the buffer manager and the higher level software is a block-at-a-time interface. If a tuple-at-a-time interface is used instead, the reference patterns of queries may not be as regular as the hot set model expects. In a hash join operation, for example, crossing a page boundary in the outer relation may cause a page of the inner relation to be flushed out, although the page of the outer relation is never needed again. In many other similar situations, HOT is sensitive to small perturbations in the reference string.

QLS yielded the best performance for most queries. An exception is that WS outperformed QLS for query IV. Query IV consists of a range selection, in which tuples with a key value that falls within a pair of bounds are selected, followed by an index join. QLS assumed that the hierarchical references to the index are randomized over the entire range of key values. Only two buffers, one for the root and one as a scratch buffer, were allocated for the index accesses by QLS, since QLS projected that a leaf page had a low probability of being re-used. (The index tree, not counting the data level, was only two-levels deep.) However, the actual references to the index were clustered in a narrow strip of the index tree because the range of the search keys had been narrowed down by the previous selection. WS was able to keep the accessed portion of the tree resident, whereas QLS miscalculated and failed to do the same. Note that QLS might have performed better if the results of the range selection were sorted, since the smaller locality set could then fit in two buffers.

4.3.2. Effects of Query Mix

Although comparing the performance of the algorithms for different query types provides insight into the efficiency of each individual algorithm, it is more interesting to compare their performance under a workload consisting of a mixture of query types. Instead of conjecturing what a "real" workload should look like, three query mixes were defined to cover a wide range of workload characterization:

Query Mix	Type I	Type II	Type III	Type IV	Type V	Type VI
M1	16.67%	16.67%	16.67%	16.67%	16.66%	16.66%
M2	25.00%	25.00%	12.50%	12.50%	12.50%	12.50%
M3	37.50%	37.50%	6.25%	6.25%	6.25%	6.25%

Experiments similar to those for the homogeneous workloads were conducted for the three query mixes. The results, as shown in Figure 4.4, are consistent with those of the previous experiments. Thrashing is again observed for the simple algorithms, although the degradation is not as steep as before. The slowdown in performance degradation may be explained as follows: The three simple algorithms are all global algorithms in which the buffers allocated to each query are not isolated for page replacement purposes. Some types of queries may compete more favorably than others under high memory contention. Consequently, queries that managed to get more buffers might still be making reasonable progress while others had begun to thrash. This argument is consistent with the analysis of memory contention by Smith [Smit80] which showed that an imbalance of memory allocation can result under heavy memory contention as a self-stabilizing mechanism to resist thrashing.

WS did not perform well because it failed to capture the main loops of the joins in queries V and VI. Its performance improved as the frequency of queries V and VI decreased. The efficiency of HOT was close to that of QLS. When the system was lightly loaded, QLS was only marginally better than the rest of the algorithms. However, as the number of concurrent queries increased to 8 or more, HOT and WS were about 7 to 13% and 25 to 45% worse than QLS, respectively. Since mixed workloads are more likely to be found in real environments than homogeneous workloads, we discuss from now on only experiments with mixed queries.

4.3.3. Effects of Data Sharing

To study the effects of data sharing on the performance of the algorithms, two more sets of experiments, each with a higher degree of data sharing, were conducted. In the experiments with half data sharing, an average of two queries were simultaneously accessing

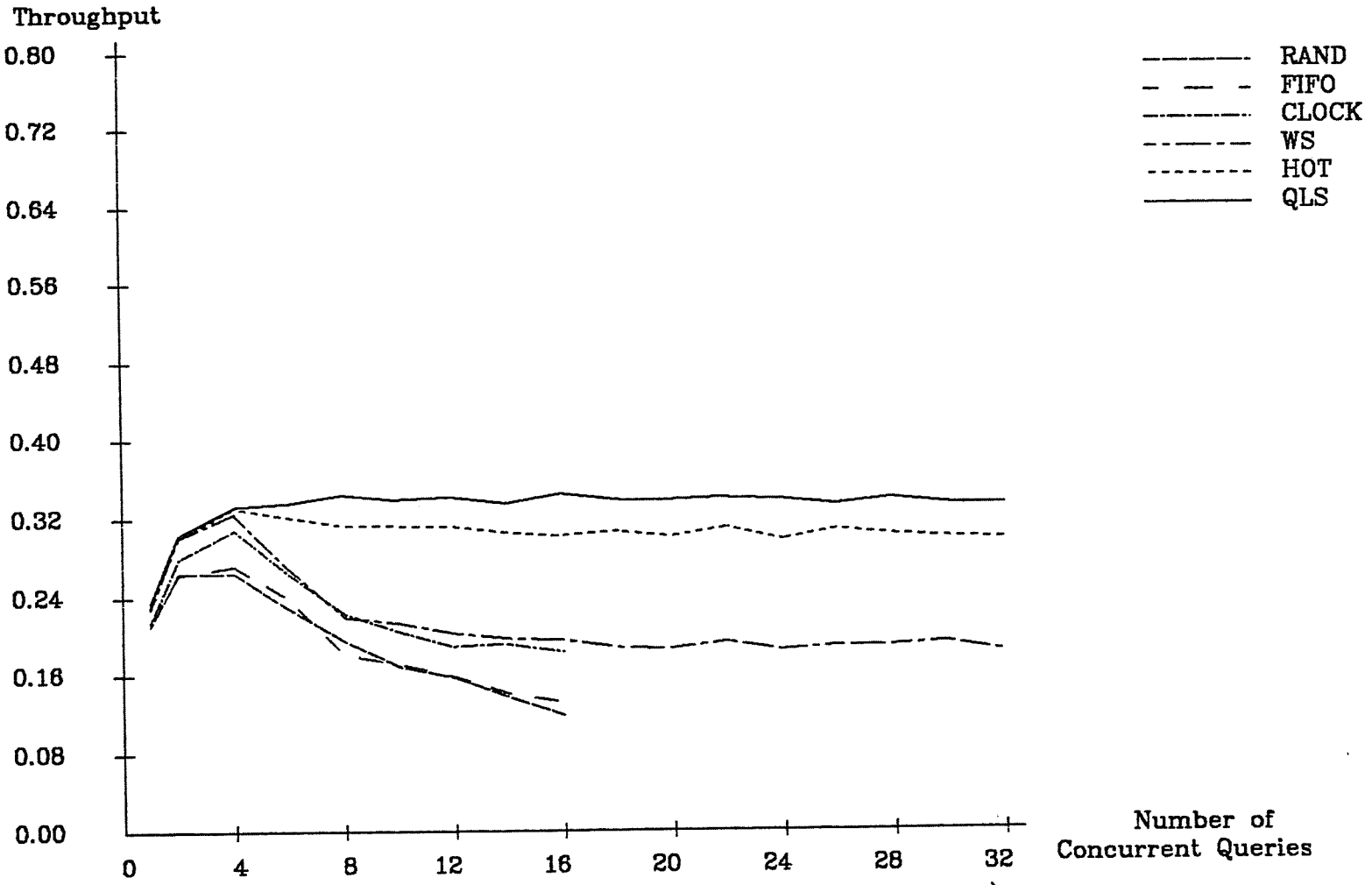

Figure 4.4 (a) Query Mix M1 (100 Buffers, No Data Sharing)

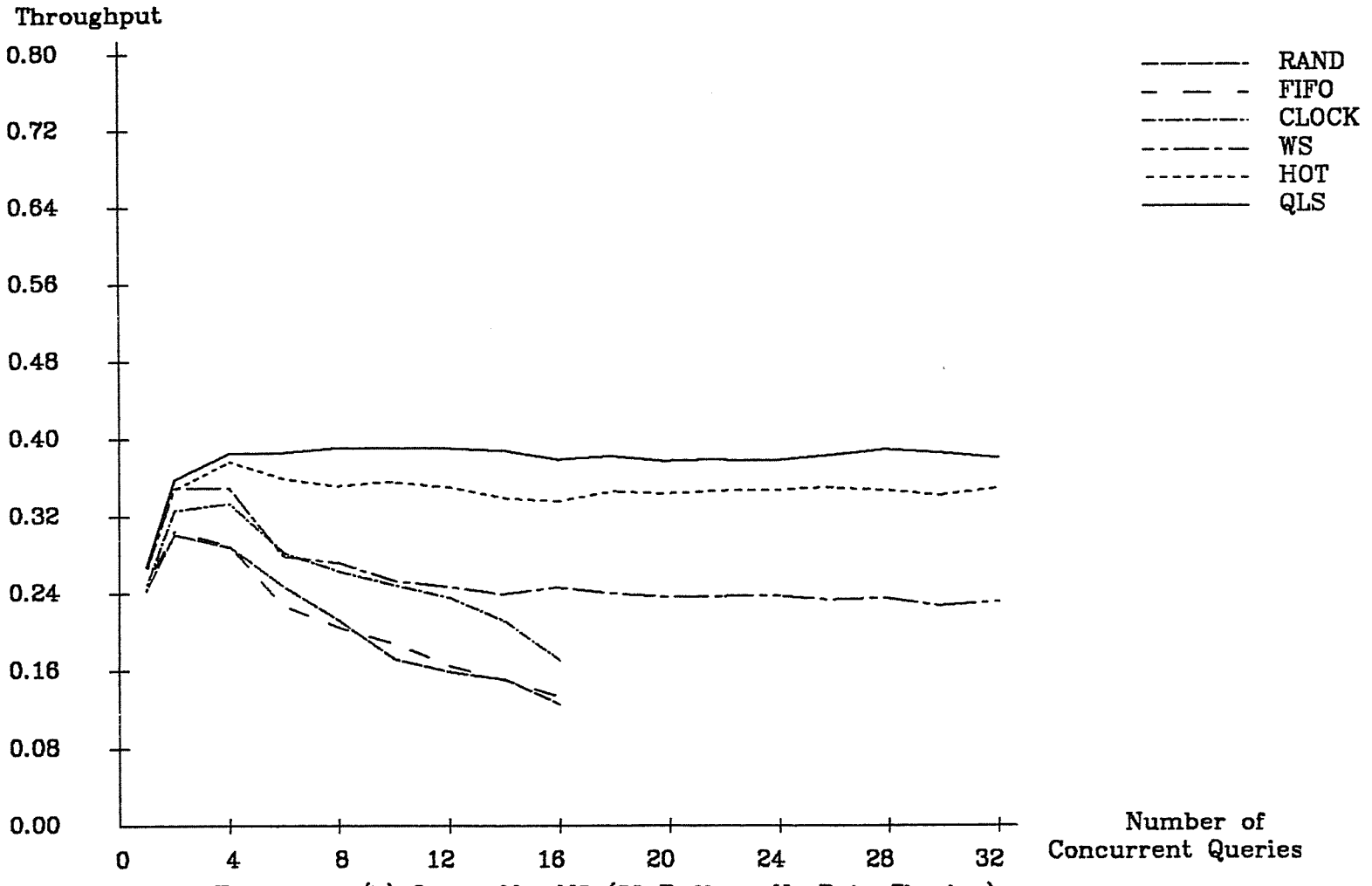

Figure 4.4 (b) Query Mix M2 (80 Buffers, No Data Sharing)

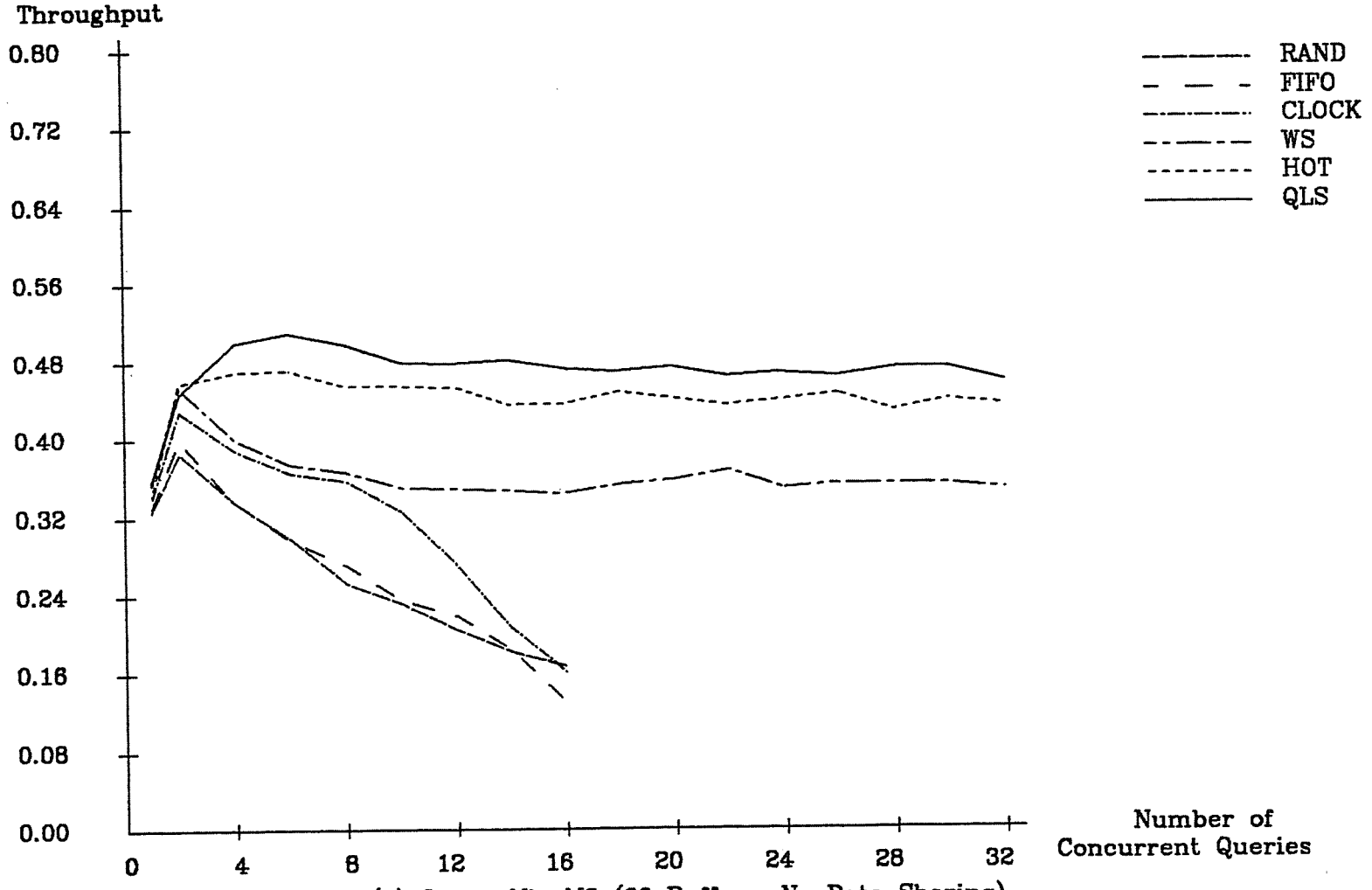

Figure 4.4 (c) Query Mix M3 (60 Buffers, No Data Sharing)

each copy of the database. In the case of full data sharing, all concurrent queries were sharing the same database copy. The results are plotted in Figures 4.5 and 4.6. It can be observed that, for each of the algorithms, the throughput increases as the degree of data sharing increases. This reinforces the view that allowing for data sharing among concurrent queries is important in a multi-programmed database system [Reit76] [Bora84].

The relative performance of the algorithms for half data sharing is similar to that for no data sharing, but this is not the case for full data sharing. For query mixes M1 and M2, the efficiencies of the different algorithms were close. Because every query accessed the same copy of the database, it was easy for any algorithm to keep the important portion of the database in memory. Not surprisingly, RAND and FIFO performed slightly worse than the other algorithms due to their inherent deficiency in capturing locality of reference. For query mix M3, however, the performance of the different algorithms again diverged. This may be attributed to the fact that small queries dominated the performance for query mix M3. The "working" portion of the database becomes less distinct as many small queries are entering and leaving the system. (In contrast, the larger queries, which intensively access a limited set of pages over a relatively long period of time, played a more important role for query mixes M1 and M2.) Therefore, algorithms that made an effort to identify the localities performed better than those that did not.

4.3.4. Effects of Load Control

As was observed in the previous experiments, the lack of load control in the simple algorithms led to thrashing under high workloads. It is interesting to find out how effective those algorithms will be when a load controller is incorporated. The "50% rule" [Lero76], in which the paging device is kept busy about half the time, was chosen for its simplicity of

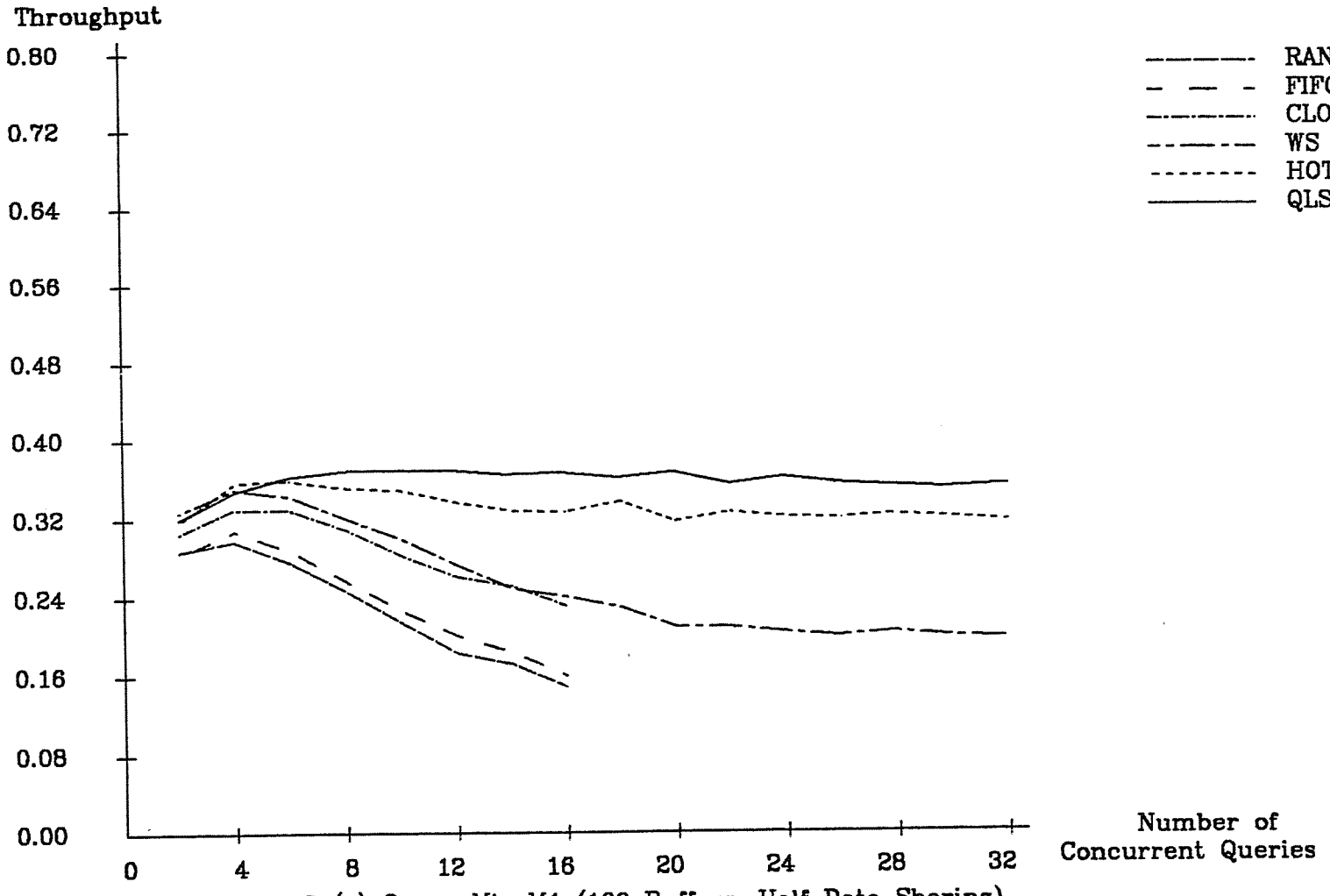

Figure 4.5 (a) Query Mix M1 (100 Buffers, Half Data Sharing)

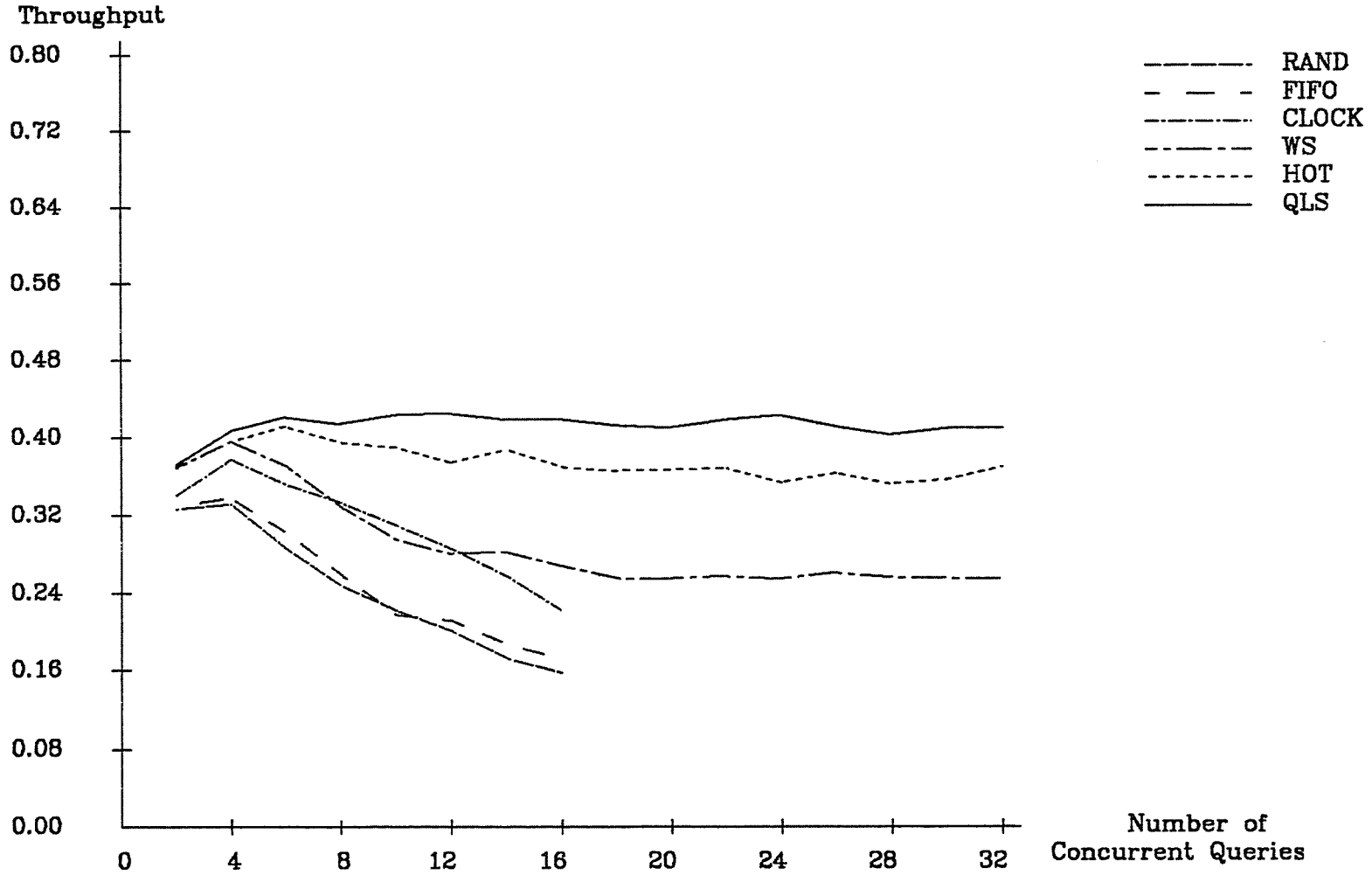

Figure 4.5 (b) Query Mix M2 (80 Buffers, Half Data Sharing)

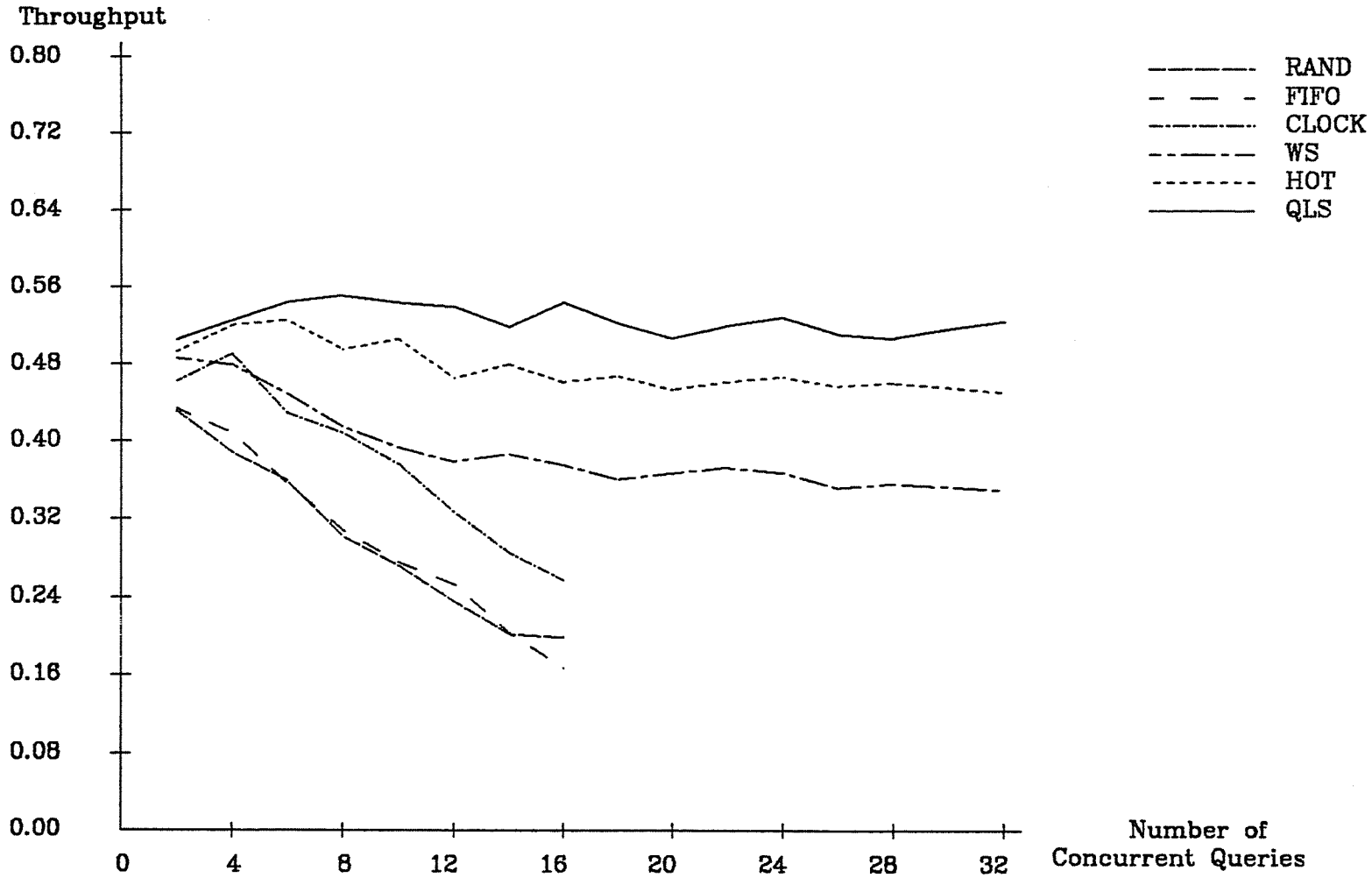

Figure 4.5 (c) Query Mix M3 (60 Buffers, Half Data Sharing)

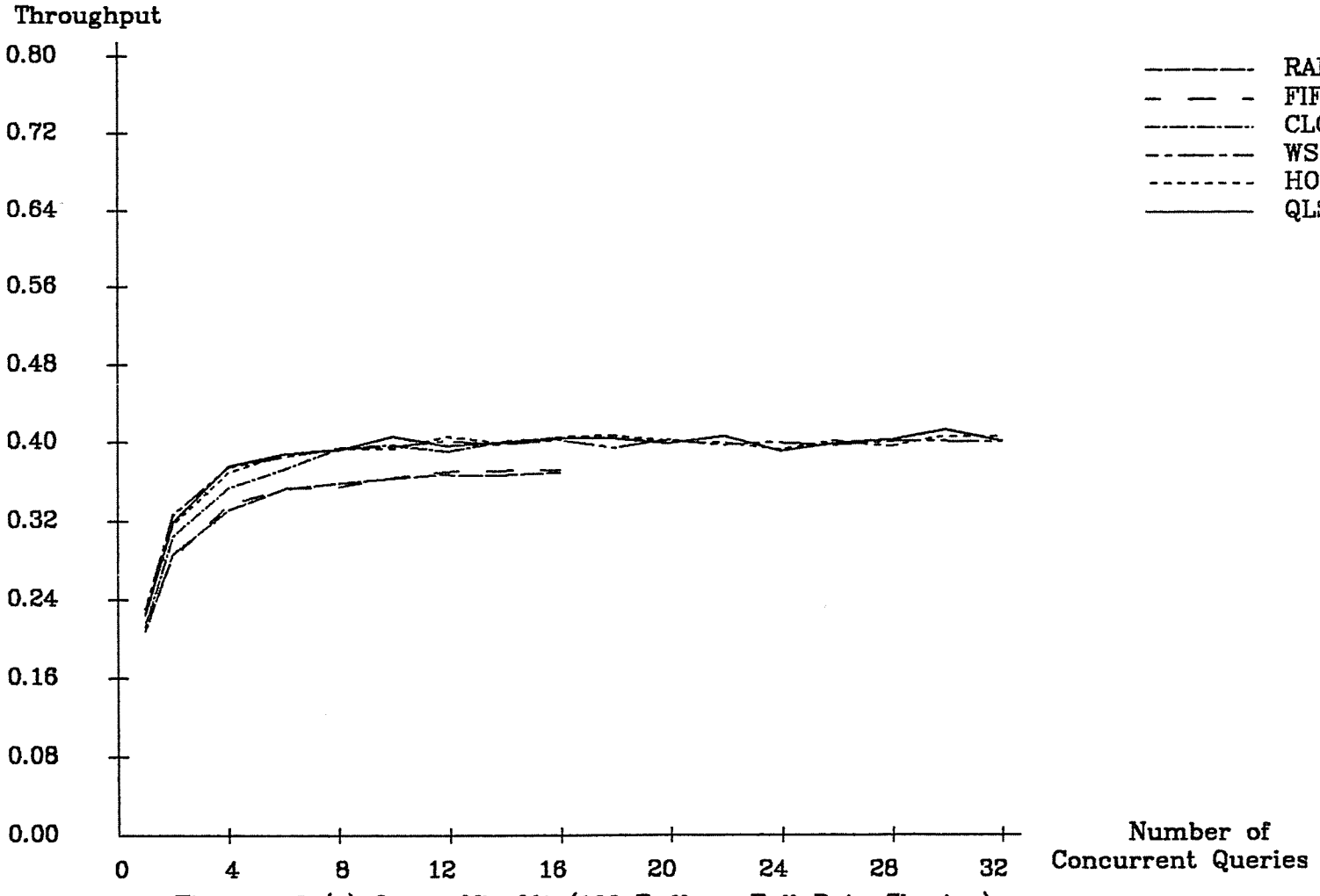

Figure 4.6 (a) Query Mix M1 (100 Buffers, Full Data Sharing)

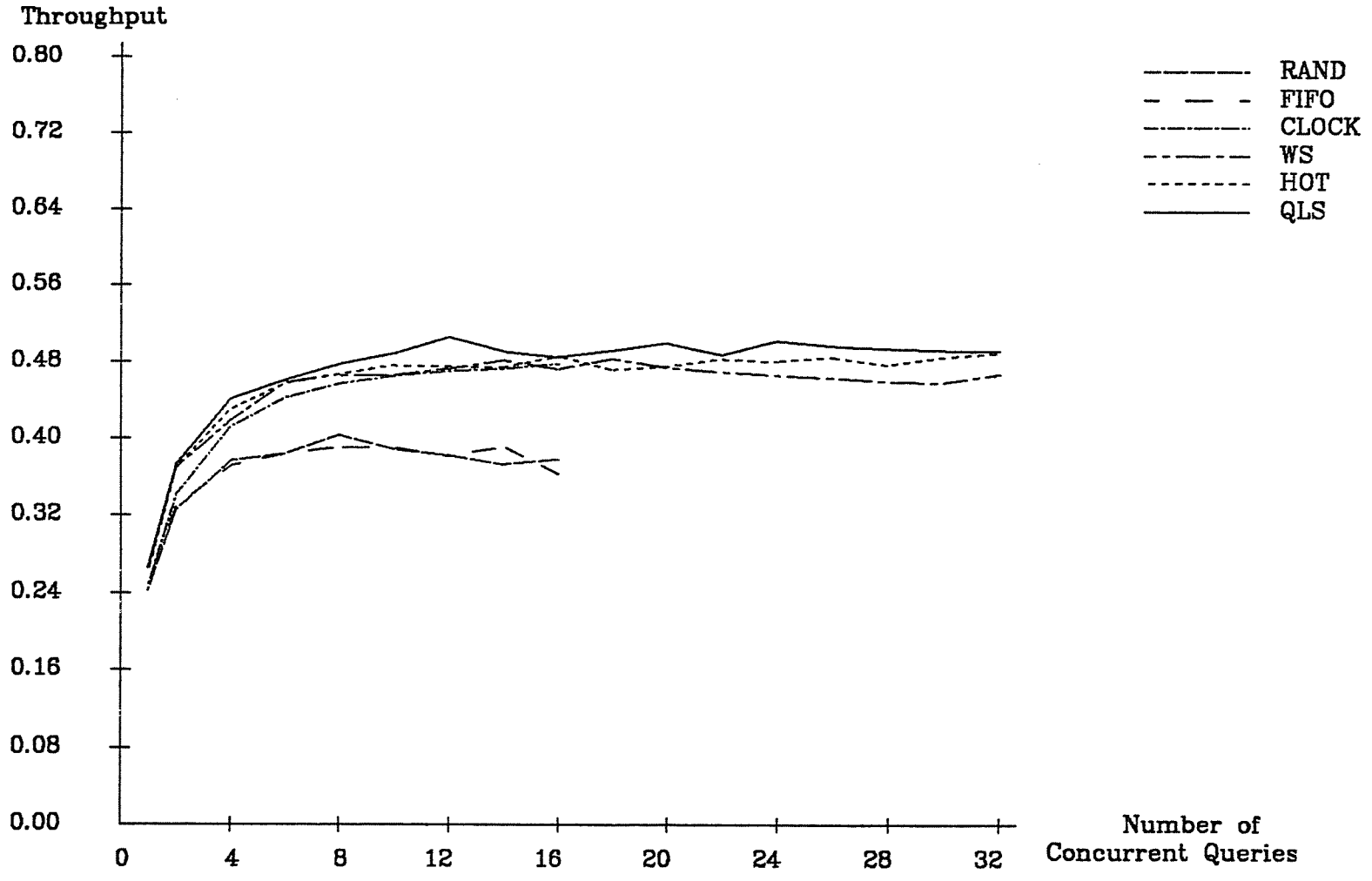

Figure 4.6 (b) Query Mix M2 (80 Buffers, Full Data Sharing)

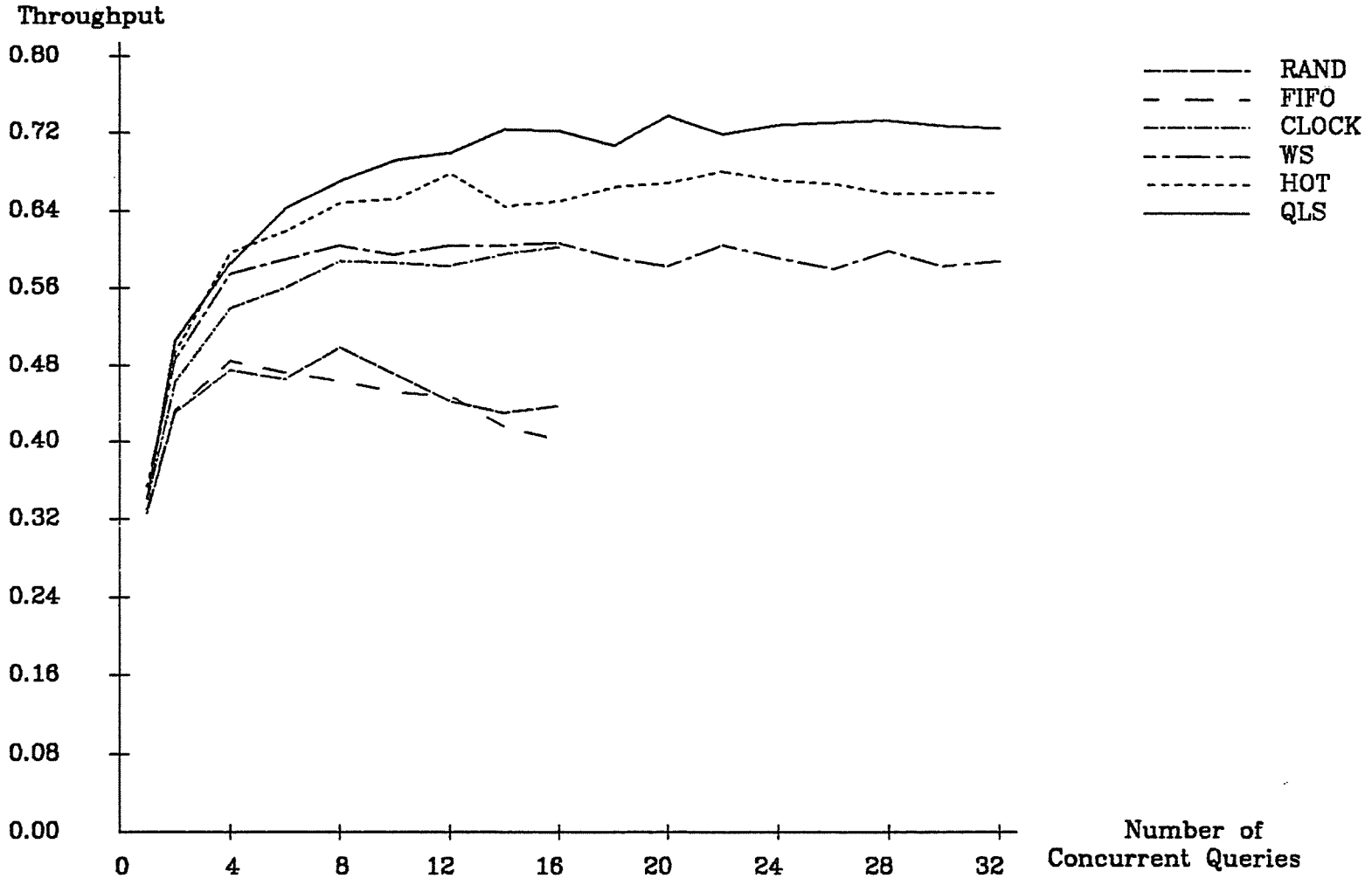

Figure 4.6 (c) Query Mix M3 (60 Buffers, Full Data Sharing)

implementation and because it is supported by empirical evidence [Denn76b].

A load controller which is based on the "50% rule" usually consists of three major components:

(1) an **estimator** that measures the utilization of the paging device,

(2) an **optimizer** that analyzes the measurements provided by the estimator and decides what load adjustment is appropriate, and

(3) a **control switch** that activates or deactivates processes according to the decisions made by the optimizer.

The measurements provided by the estimator are in the form of a confidence interval $[u^-, u^+]$. Each confidence interval is checked against a threshold U_{th}[8], which is the desired level of device utilization. The optimizer bases its decisions on the following rules:

(1) retain the current multi-programming level if $U_{th} \in [u^-, u^+]$,

(2) increase the multi-programming level if $U_{th} > u^+$, and

(3) decrease the multi-programming level load if $U_{th} < u^-$.

Setting the value of U_{th} to 50% is reasonable for virtual memory systems because the objective is to maximize CPU utilization by avoiding queuing delay of processes at the paging device. For a database system, however, a higher value of U_{th} is more desirable since we want to maximize both CPU and disk utilization. We tried several values of U_{th}, ranging from 50% to 100%. The results, as shown in Figure 4.7, suggest that performance improvement can be achieved by increasing the utilization of the disk up to a certain point. (Although only the results for query mix M1 are shown, the results for query mixes M2 and

[8] For virtual memory systems, the value of U_{th} is usually around 50%. This is why the criterion is called the "50% rule".

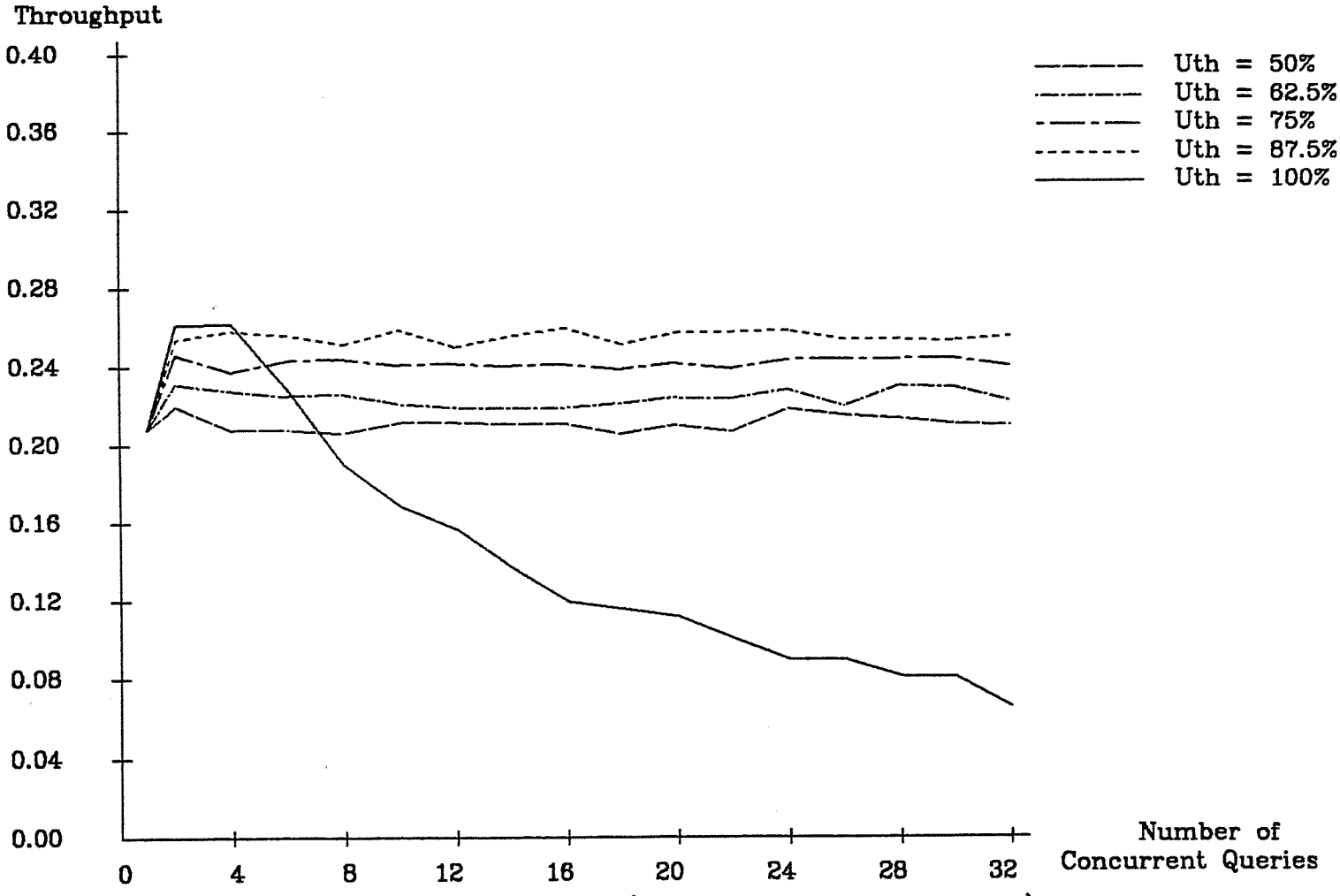

Figure 4.7 (a) Query Mix M1, RAND (100 Buffers, No Data Sharing)

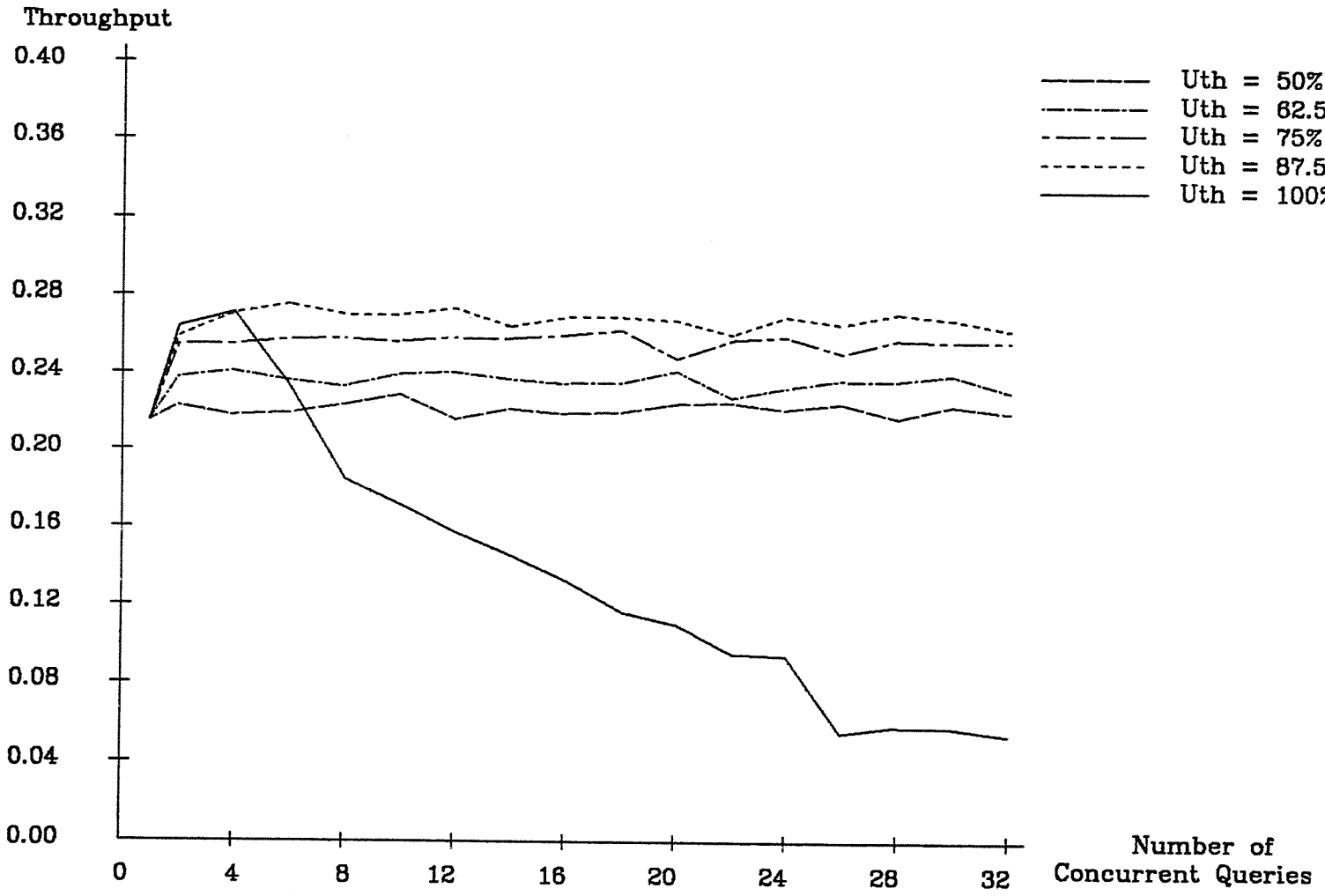

Figure 4.7 (b) Query Mix M1, FIFO (100 Buffers, No Data Sharing)

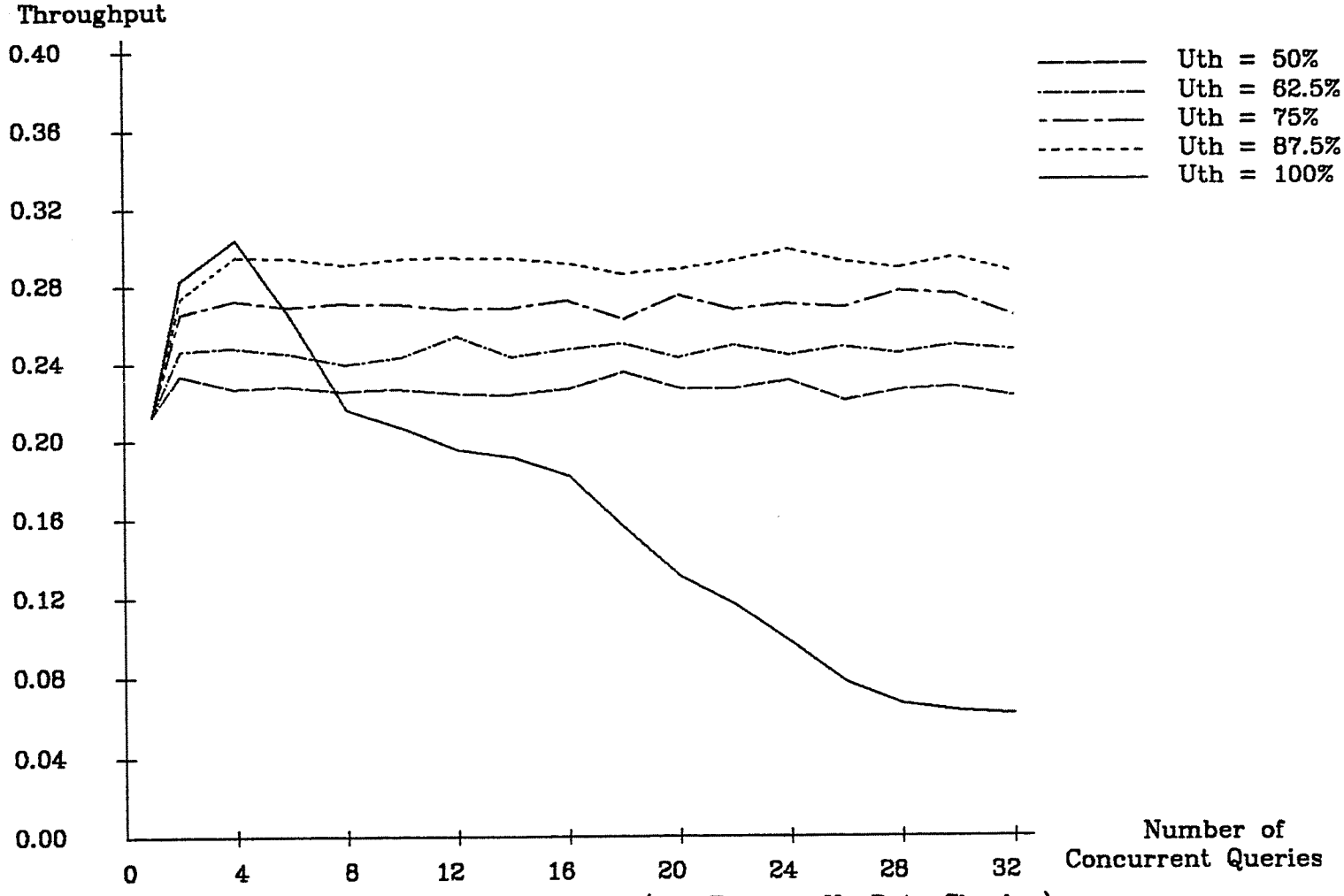

Figure 4.7 (c) Query Mix M1, CLOCK (100 Buffers, No Data Sharing)

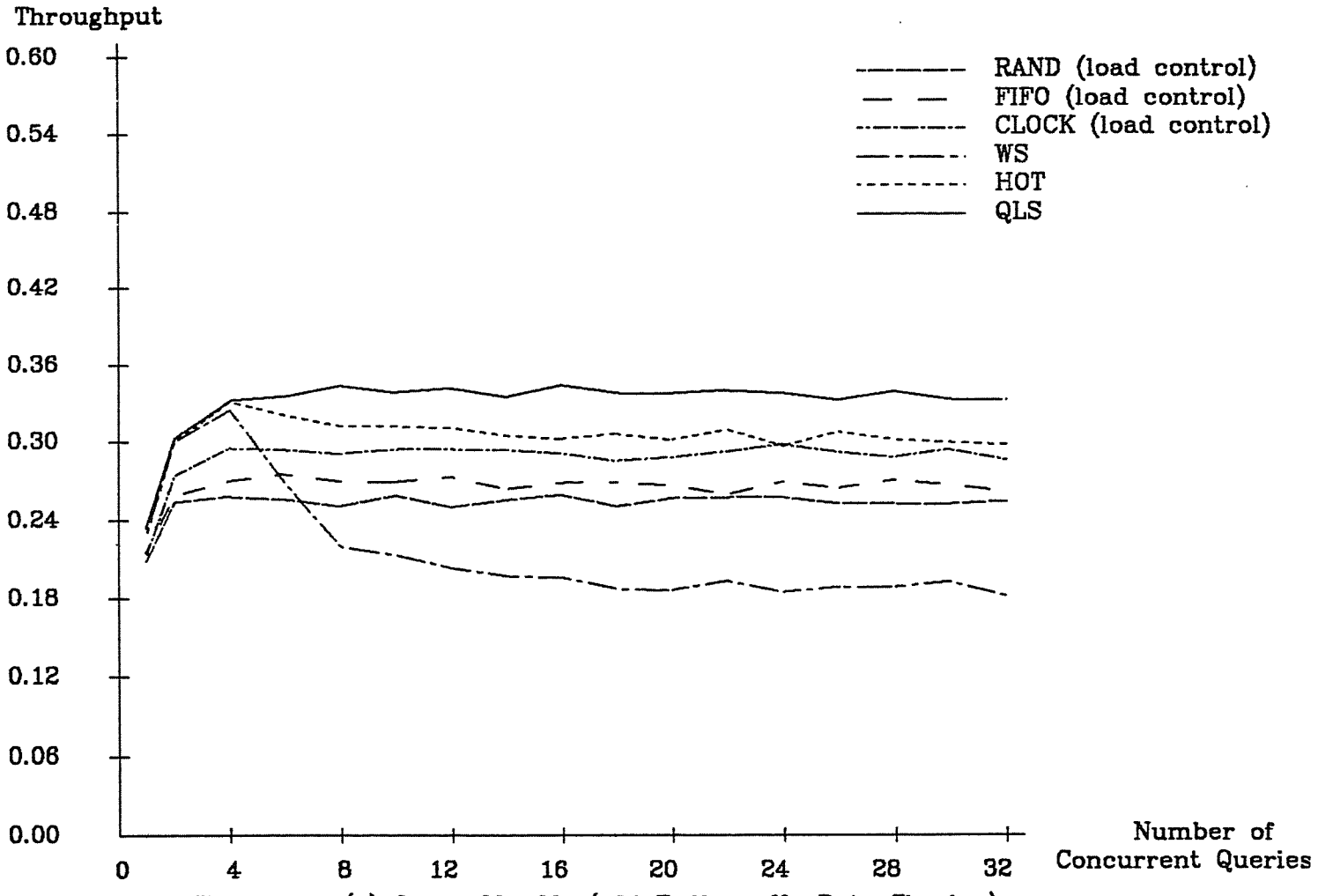

Figure 4.8 (a) Query Mix M1 (100 Buffers, No Data Sharing)

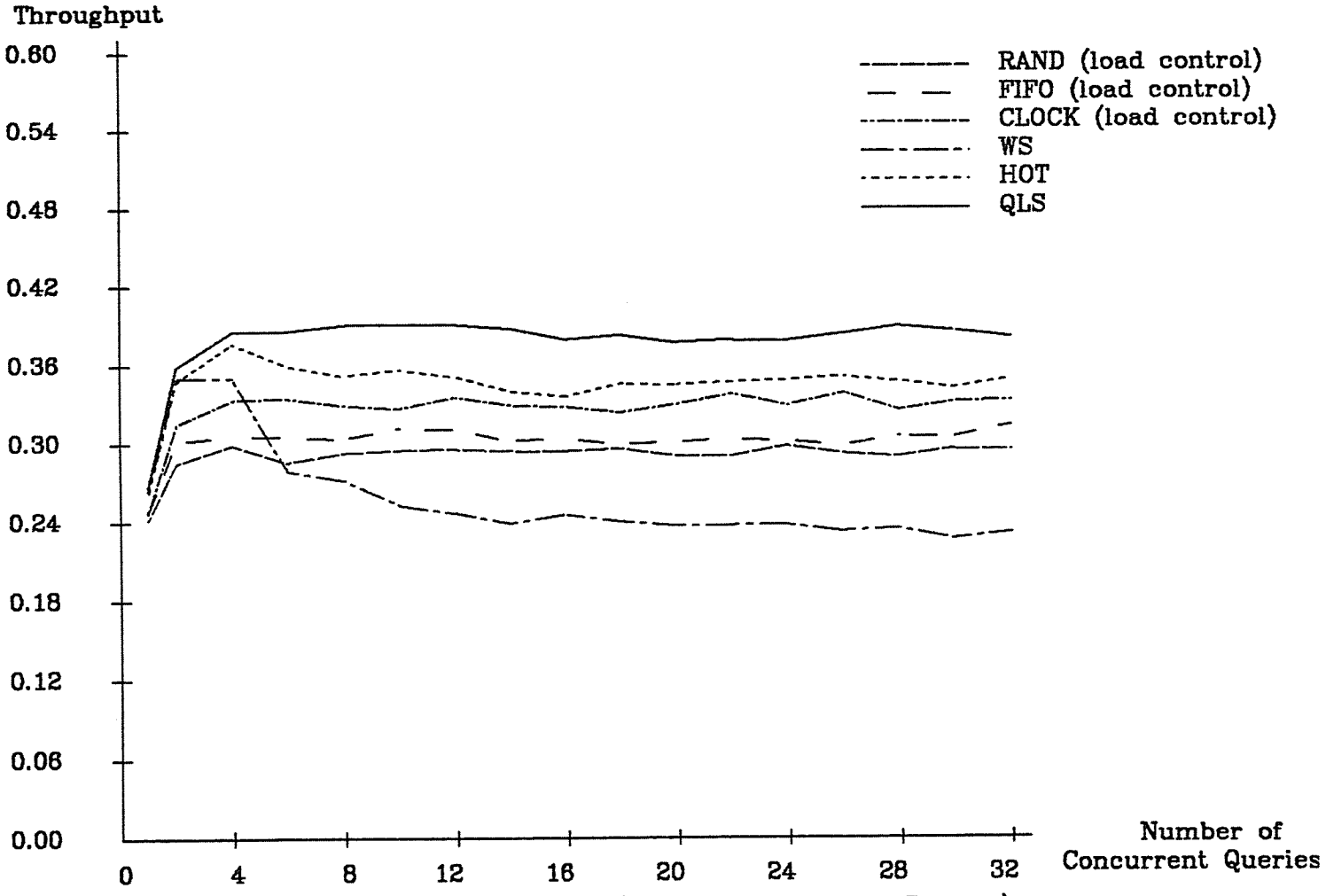

Figure 4.8 (b) Query Mix M2 (80 Buffers, No Data Sharing)

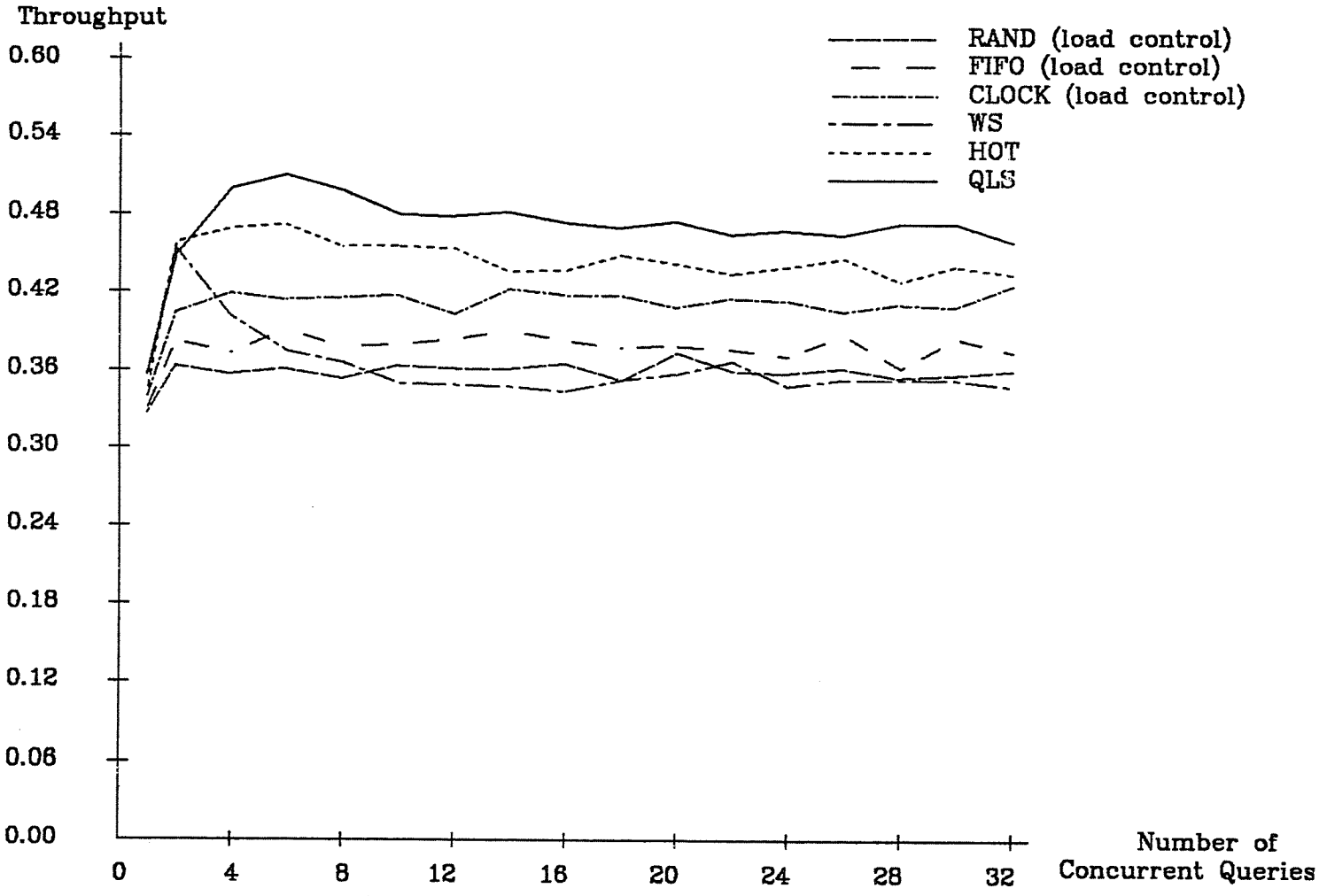

Figure 4.8 (c) Query Mix M3 (60 Buffers, No Data Sharing)

M3 were similar.) There is virtually no load control when U_{th} is set to 100% as the lower bound of a confidence interval can never exceed 100%. In fact, using a value higher than 90% is worthless since a typical confidence interval, as measured in the experiments, is around $\pm 10\%$ of the mean value.

The empirical data show that a load controller can indeed prevent thrashing. With load control, every simple algorithm in the experiments outperformed the WS algorithm (Figure 4.8). The performance of CLOCK with load control even came close to that of HOT. However, the results should not be interpreted too literally. There are potential problems with such a load control mechanism which arise from the feedback nature of the load controller:

(1) Run-time overhead can be expensive if sampling is done too frequently. On the other hand, the optimizer may not respond fast enough to adjust the load effectively if analyses of the measurements are not done frequently enough. The interval between measurements is usually on the order of 100 milliseconds [Bade75]. Although a smoothing type (or an incremental) technique can be used to cut down the costs of the analysis, estimating the device utilization is still a computationally expensive procedure.

(2) Unlike the predictive load controllers, a feedback controller can only respond after an undesirable condition has been detected. This may result in unnecessary process activations and deactivations that might otherwise be avoided by a predictive load control mechanism.

(3) A feedback load controller does not work well in an environment with a large number of small transactions which enter and leave the system before their effects can be assessed. It can be seen in Figure 4.8 that the load controller based on the "50%

rule" becomes less effective as the percentage of small queries increases. Note that the so-called "small queries" (i.e. queries I and II) in our experiments still retrieve 100 tuples from the source relation. The disadvantages of a feedback load controller are likely to be more apparent in a system with a large number of single-tuple queries or queries that access a few tuples (e.g., debit-credit transactions).

4.4. Cost Analysis

In the previous section, we have compared the performance of the six buffer management algorithms through a large number of simulation experiments. However, the evaluation is still incomplete, as the cost associated with each algorithm was not included in the simulation model. To better evaluate the algorithms, a cost analysis of the algorithms is presented in this section. We restrict our analysis to the last three algorithms since the more sophisticated algorithms, especially QLS and HOT, have a significant performance advantage over the simple algorithms.

Due to the complexity involved in analyzing a multi-user system, we were initially considering direct cost measurements of a prototype implementation of the buffer management algorithms. Since the implementation of the algorithms in the simulator contains sufficient details to be considered a "realistic" implementation, measuring the time the simulator spends in the buffer manager may provide a reasonable estimate of the cost of a buffer management algorithm. A major problem with this approach, however, is that the buffer manager was coded in such a way that the different algorithms can share the same interface and data structures. Hence, direct cost measurements from the simulator do not necessarily reflect the inherent costs of the algorithms.

Modeling is the next alternative after direct measurement has been ruled out. Since we are more interested in comparing (rather than quantifying) the performance of the algorithms, an abstract analysis seems adequate for our purpose. Thus, a simple model, in which the activities of a buffer management algorithm are described as a set of basic operations, is used in our analysis.

As described in section 2 of this chapter, all three algorithms (WS, HOT, and QLS) maintain a global table and one local table for each process or file instance. Accesses to the global and local tables are similar in all three algorithms. (Note that locating the local table for a file is no more difficult than locating the local table for a process.) The main difference among these algorithms is in the way that they maintain the local tables (i.e. the resident sets). There are three possible cases for each page request:

Case 1. The page is found in the local table (which also implies that the page is resident in the buffer).

Case 2. The page is found in memory but not in the local table.

Case 3. The page is not in memory.

To formulate the cost of each algorithm for these three cases, we shall use the notation $\mathbf{C_i(A)}$ to represent the cost of algorithm **A** for case **i**. In addition, we shall use $\mathbf{c_{op}}$ to represent the cost associated with operation **op**, which is one of the following operations:

(1) **su** - stack (or table) entry update.

(2) **vu** - variable update.

(3) **cp** - variable comparison.

Using the notation $\mathbf{C_i(A)}$ and $\mathbf{c_{op}}$, we shall first derive the cost formulas for the WS algorithm:

$$C_1(WS) = c_{vu}$$

$$C_2(WS) = c_{su} + c_{vu}$$

$$C_3(WS) = rss \cdot c_{cp} + (rss - wss) \cdot c_{su} + C_2(WS)$$

where rss and wss represent the resident set size and working set size of the faulting process, respectively. In case 1, only a time stamp needs to be updated as the page is already in the working set of the process. In case 2, a new stack entry with a fresh time stamp needs to be added to the working set. Case 3 is more complicated since it results in a context switch, at which time the resident set is compacted to be the same as the working set[9]. The first term in C_3 is the cost to compute the working set, and the second term is the cost to remove pages that are no longer in the working set. The last term is for adding the faulting page to the working set.

Since a fixed-size LRU stack is maintained for each process, the cost functions for HOT are straight-forward:

$$C_1(HOT) = C_2(HOT) = C_3(HOT) = c_{su} + c_{vu}.$$

In all three cases, the response of HOT is the same; namely, update an LRU stack and its associated stack pointer. If pushing the bottom entry off an LRU stack is considered a separate operation, C_2 and C_3 should be modified to:

$$C_2(HOT) = C_3(HOT) = 2 \cdot c_{su} + c_{vu}.$$

Four replacement policies were used by QLS in the experiments: SB, RAND, LIFO[10] and MRU. The cost functions of QLS are:

---

[9] We have ignored the case where a process is preempted by the scheduler because such a preemption should be less frequent than a page fault.

[10] As an approximation, index pages in hierarchical references were managed by LIFO.

$$C_1(QLS) = \begin{cases} c_{vu} & \text{for MRU} \\ 0 & \text{otherwise,} \end{cases}$$

$$C_2(QLS) = C_3(QLS) = \begin{cases} c_{vu} & \text{for SB} \\ c_{su} & \text{for RAND} \\ c_{su} + c_{vu} & \text{for LIFO and MRU.} \end{cases}$$

MRU is the only policy that needs to update a recency pointer for each page access. When a page is missing from the local table, SB simply replaces the old page while RAND randomly picks a victim for replacement. Under the same situation, both LIFO and MRU need to update the local stack and its associated pointer.

Although the above analysis is not a rigid derivation, it does give us a general idea of the relative cost of the algorithms. Judging from the formulas, the cost of the WS algorithm is higher than that of HOT unless the page fault rate is kept very low. In comparison, QLS is less expensive than both WS and HOT, as fewer usage statistics need to be maintained.

4.5. Conclusions

In this chapter, we have compared the performance of six buffer management algorithms using the results of a number of simulation experiments. Homogeneous workloads help us understand the advantages and potential problems associated with each buffer replacement algorithm, while mixed workloads provide a clearer view of the relative performance of the different algorithms. The experiments on data sharing show that there is indeed a performance advantage associated with allowing sharing among concurrent queries.

Without load control, the performance of the three simpler algorithms (CLOCK, RAND, FIFO) suffered from thrashing at higher loads. A trial implementation of the "50%

rule" was shown to be effective in controlling thrashing. However, as we pointed out, there are cases where such a feedback load controller may fail. Furthermore, its run-time overhead can be very expensive.

As expected, the three more sophisticated algorithms - WS, HOT, and QLS - performed better than the simple algorithms. However, WS did not perform as well as "advertised" for virtual memory systems [Denn78a], especially when its run-time overhead is considered. HOT and QLS generally performed better than the first four algorithms. In comparison, QLS, which has a lower run-time overhead, provided 7 to 13% more throughput than HOT over a wide range of operating conditions for the tests conducted.

CHAPTER 5

SYSTEM INTEGRATION ISSUES

In the preceding chapters, we have described the design and the evaluation of the QLS buffer management algorithm. In this chapter, we shall investigate issues related to the integration of different components of a database system. In particular, we shall examine how to interface a QLS-based buffer manager with two other major components of a database system, the query optimizer and the transaction manager.

5.1. Integration of Query Optimization and Buffer Management

A buffer manager based on the QLS algorithm requires more information than a conventional buffer manager does, including file sizes, blocking factors, expected patterns of accesses, etc.. In many database systems, such information is already available for the purpose of query optimization [Wong76] [Seli79]. In this section, we shall describe how to interface a QLS-based buffer manager with a query optimizer.

As shown in Figure 5.1, the buffer manager has two components, a **buffer predictor** and a **buffer management algorithm**. During access path selection for a query, the query optimizer consults the buffer predictor to determine the memory requirement of each access path. Given the intended usage of a file and other information pertinent to the access path, the buffer predictor returns two pieces of information, an estimation of locality set size and a replacement policy. Based on this information and the current availability of the buffers, the query optimizer generates an optimized access plan for the query. The locality set size and replacement policy associated with each selected access path are encoded in the access plan

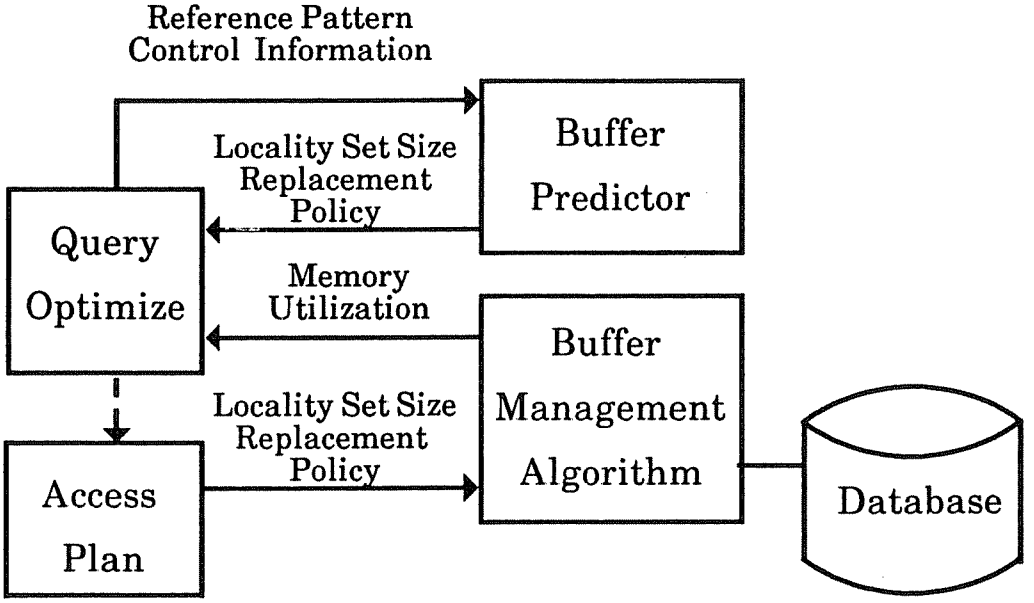

Figure 5.1 Integration of Buffer Management
and Query Optimization

and are given to the buffer management algorithm at query execution time. For compiled queries [Cham81], the query optimizer could generate alternative plans with different memory requirements, and delay the selection of the actual access plan until execution time.

Although the query locality set model covers the reference patterns of a wide range of algorithms for relational database operations, the design of new algorithms is still an active area of research. In particular, several new join algorithms, including the Grace-hash join algorithm and the hybrid-join algorithm, have recently been proposed for systems with a

large amounts of main memory [DeWi84a]. To explore the advantages of using a large buffer space, these algorithms, with the buffer size as an explicit parameter, carefully coordinate the use of buffers to minize disk I/O's. One approach for coping with the buffer management needs of these new algorithms is to extend the query locality set model. However, such an approach may be complicated and unnecessary, as the algorithms themselves are very efficient in utilizing the buffer space.

A simple and effective solution is to provide an interface that allows these algorithm to allocate the necessary space and do their own algorithm-specific buffer management. Using this approach, not only is the full efficiency of these new algorithms preserved, but the overhead associated with a general purpose buffer manager is also avoided. The only modification required to the buffer manager is the addition of a dummy "replacement policy" which does nothing. Under this scheme, no consultation with the buffer predictor is needed for a "space-conscious" algorithm. Instead, the query optimizer (or its run-time support) needs to know only the availability of the buffers and decides a proper buffer size for the algorithm. For a request with the dummy replacement policy, the buffer management algorithm simply allocates a buffer pool of the requested size and returns it to the requester. No further participation of the buffer manager is required until this buffer pool is released by the algorithm.

The preceding argument is also applicable to other "space-conscious" algorithms for database operations. The N-way merge sort operation discussed in chapter 2 is one such example. In addition, there is also a class of algorithms for statistical database operations that fall into this category [Khos84]. Like the hash join algorithms, these algorithms are aware of the size of the available buffer space and are very careful in scheduling disk I/O's and performing buffer overlays. Thus, there is no need to impose another layer of buffer management that is unlikely to improve the efficiency of these algorithms. However,

coordination between the query optimizer and the buffer manager is still necessary in a multi-user enironment to prevent possible monopoly of the buffer pool by a "greedy" algorithm.

5.2. Integration of Transaction Support and Buffer Management

Beside buffer management, transaction recovery is another important factor affecting the performance of a database system. Careful coordination between the buffer manager and the transaction manager is often necessary for recovery purposes. For example, the **Write Ahead Log Protocol** [Gray78] requires that enough information, such as an UNDO log record, be recorded on a nonvolatile storage device before a modified data page of an uncommitted transaction is written back to the database. The recovery mechanism can be simplified if there is more cooperation between the buffer manager and the transaction manager. One such scheme, called the **Database Cache** (DB-cache) management, has recently been proposed by Elhardt and Bayer [Elha84] (Figure 5.2).

A key concept in their approach is to forbid the modified pages of uncommitted transactions to be swapped out to the physical database, assuming that the cache (buffer pool) is large enough to hold the modified pages of all uncommitted transactions. Recovery from a transaction failure is fast since only the pages in the cache that have been modified by the transaction need to be invalidated. To guarantee the effects of a committed transaction, pages modified by the transaction are written sequentially to a nonvolatile storage device called the **safe** at commit time. Actual updates of a committed transaction to the database can then be done asynchronously after the transaction has been committed. The recovery procedure after the system crashes is fast and simple since only the relevant part of the cache (i.e. the set of pages modified by committed transactions which have not been flushed back

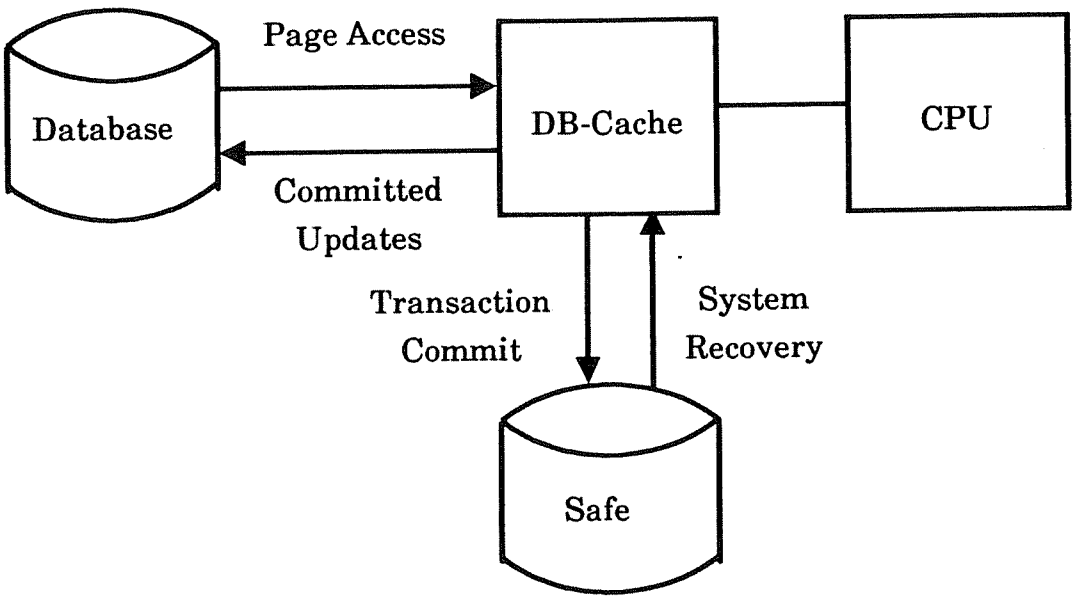

Figure 5.2 The DB-Cache Recovery Mechanism

to the physical database) need to be restored from the contents of the **safe**. No determination of winner and loser transactions [Gray78] and REDO/UNDO operations are necessary. Thus, the DB-cache mechanism provides a simple, elegant solution to the recovery of small and medium transactions[1].

Our QLS algorithm and the concept of the DB-cache are compatible for several reasons:

[1] Long transactions are also supported by additional mechanisms in DB-cache.

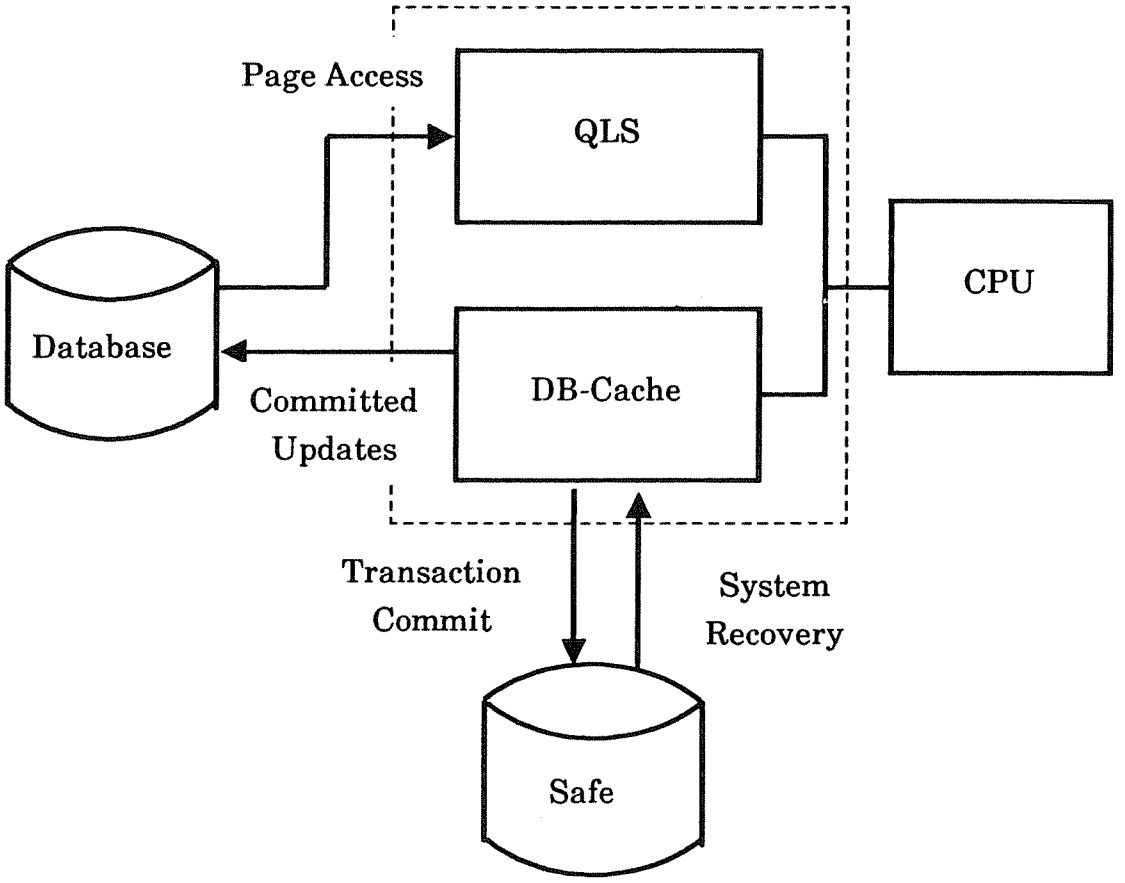

Figure 5.3 Integration of Buffer Management
and Transaction Management

(1) The separation of buffer management for different queries, i.e. localized management of buffers, is useful for identifying the pages, especially the modified pages, that are associated with a transaction. Such an identification is necessary for the commit and abort protocols described above.

(2) QLS is a space-efficient buffer management algorithm in the sense that each query is given the minimal amount of buffer space that is required for the efficient execution of the query. The conservation of buffer space by the buffer manager allows the transaction manager to have more space for transaction support.

One view toward the integration of QLS and the DB-cache concept is that read-only pages are efficiently managed by the QLS algorithm where as updated pages are under the control of the DB-cache mechanism for recovery purposes.

The QLS algorithm and the DB-cache concept can be integrated as follows (Figure 5.3): Each active transaction is given an **update set** to hold all the pages modified by the transaction. In addition, there is a **global update list** to keep track of all the pages that have been modified by committed transactions but have not been written back to the physical database. When a modified page of a transaction leaves a locality set, instead of releasing it back to the global free list, the page is added to the update set of the transaction. If a transaction aborts or is aborted, buffers associated with its update set are invalided and released back to the global free list. Otherwise, pages in the update set of a committing transaction are written to the "safe" and added to the global update list. A page in the global update list is written asynchronously to the physical database, and joins the global free list afterwards. Note that the global update list corresponds to the portion of the DB-cache that needs to be reconstructed from the "safe" after system crashes.

CHAPTER 6

CONCLUSIONS AND DIRECTIONS FOR FUTURE RESEARCH

Motivated by the need to find a buffer management algorithm that is better suited to database systems than conventional virtual memory policies [Ston81], and encouraged by the results of the hot set algorithm for single user queries [Sacc82], we initiated a study of the problem of database buffer management. Our study was guided by the three steps of a typical problem-solving life-cycle: modeling, algorithm design, and evaluation. To benefit from the results of previous virtual memory studies, we conducted a review of the literature in this area. This examination of the previous studies of virtual memory management, as well as recent advances in the study of database buffer management, led to the introduction of a new query behavior model and the design of a new buffer management algorithm. In this chapter, we shall summarize the results of our study.

In Chapter 2, we reviewed some important results on modeling the reference behavior of programs and database queries, and presented a new reference behavior model, the query locality set model (QLSM), for relational database systems. Using a classification of page reference patterns, we have shown how the reference behavior of common database operations can be described as a composition of a set of simple and regular reference patterns. Like the hot set model, the QLSM has an advantage over conventional stochastic models due to its ability to predict future reference behavior. However, the QLSM avoids the potential problems of the hot set model by separating the modeling of reference behavior from any particular buffer management algorithm.

In Chapter 3, we reviewed important results from the studies of main memory management, focusing on three important issues: load control, memory partitioning and page replacement. In particular, we showed how to systematically classify a memory policy according to each of the three facets of memory management. Based on this three-dimensional classification scheme, we are able to better understand the inherent features and potential efficiency (or inefficiency) of each particular memory policy. By applying the same classification scheme, we also examined and characterized a number of buffer management algorithms for database systems. We then proposed a new buffer management algorithm for database systems, the Query Locality Set (QLS) algorithm, which is based on the query locality set model. Using the file instance as the basic unit for buffer allocation and management, the QLS algorithm attends to the individual memory needs of the queries. In addition, the QLS algorithm also provides an integrated solution to the problems of load control and memory partitioning based on its predictive power.

An evaluation of buffer management algorithms in a multi-user environment was presented in Chapter 4. Using a combination of trace-driven and distribution-driven techniques for simulation experiments, we have compared the performance of six buffer management algorithms: RAND, FIFO, CLOCK, WS (Working Set algorithm), HOT (Hot Set algorithm), and QLS. Severe thrashing was observed for the three simple algorithms: RAND, FIFO, and CLOCK. Although the introduction of a feedback load controller alleviated the problem, it also created new potential problems. WS did not perform well in many cases since it was unable to capture the main loops in the larger queries. Although using a larger window may improve the performance, the selection of a proper window size is difficult. Furthermore, if a large window size is chosen, memory tends to be under-utilized during phases which contain only small loops. HOT generally performed better than the previ-

ous four algorithms. For loaded systems, however, the results of our experiment indicate that further performance improvement by 10% in throughput is possible by using QLS.

In Chapter 5, we investigated issues related to the integration of a QLS-based buffer manager and two other major components of a database system: the query optimizer and the transaction manager. Cooperation between the buffer manager and the query optimizer is necessary for two reasons:

(1) To allow the buffer manager to obtain necessary information (on the access paths) for the prediction of locality set sizes and the selection of proper replacement policies.

(2) To extend the characterization of a query's resource demands, which in turn enables the query optimizer to better select an access plan for a query.

Through the description of one possible interface organization, we demonstrated how such cooperation can be achieved.

Close interaction between the buffer manager and the transaction manager is necessary for recovery purposes and is beneficial for performance reasons. Proper coordination between the two components can lead to a simplification of the recovery process and a reduction in the overhead of the recovery mechanism. We showed how such goals can be achieved through an interface design that integrates a QLS-based buffer management and a previously proposed cache-based recovery mechanism, called the DB-cache.

For performance reasons, direct manipulation of the data in buffer is desirable for database systems. Unlike virtual memory systems, which have hardware support for dynamic address translation, a database system usually has to explicitly "fix" a buffer to prevent replacement during its use and "unfix" the buffer when there are no active memory pointers to the buffer [Effe84]. However, the need for such a fix-unfix mechanism can

easily be avoided under the QLS algorithm. For an active query, QLS guarantees that all the pages the query needs are fixed according to its locality sets. An active buffer pointer of a query can only be invalidated while the query is suspended, which can only occur when a query opens a file instance whose locality set size exceeds the number of available buffers. If each query opens all the files it needs in each query phase (or sub-query) before it actually operates on the data, there should be no buffer points active when a query is suspended. Thus, a QLS-based buffer manager can avoid not only the need for a fix-unfix mechanism, but also the potential deadlocks associated with fixing buffers.

While we presented a new solution to the problem of database buffer management, new problems also arose that may merit further investigation. The first issue is how the memory demands of different access paths can be taken into consideration during access path selection. This issue has been largely ignored in existing query optimization techniques [Wong76] [Seli79]. For example, the cost formula used by the query optimizer of System R [Seli79] is a weighted measure of I/O and CPU utilization. However, the number of disk I/O's is usually a function of the size of the allocated buffer space and the buffer management policy. Thus, buffer management can have a significant impact on the actual cost of a query. Integrating the memory demands of an access path into the cost formula seems beneficial and viable, especially since we already have the necessary tools for predicting these resource demands.

In our study, no update queries were included in the experiments. One reason is that buffer management for update queries is better evaluated in conjunction with transaction management. We have shown in Chapter 5 how to interface a QLS-based algorithm and a cache-based recovery scheme. In particular, we have suggested an elegant solution in which read-only pages are managed by the QLS algorithm, whereas updated pages are controlled

by the transaction manager. An interesting area for future study is the comparison of the performance of update queries under this organization with that of other alternative system organizations.

APPENDIX A

WORKING SET ANALYSIS OF THE TEST QUERIES

To better understand the behavior of the test queries and to determine the proper window sizes for the working set algorithm, we have analyzed some working set properties of the queries. In particular, we have calculated the inter-reference interval histogram $\mathbf{h(t)}$, the average missing page rate $\mathbf{m}(\tau)$, and the average working set size function $\mathbf{s}(\tau)$. As we mentioned in Chapter 2, the working set functions can be calculated from the formulas [Denn72a]:

$$\mathbf{s}(\tau) = \sum_{z=0}^{\tau-1} \mathbf{m}(z) = \sum_{z=0}^{\tau-1} \sum_{t>z} \mathbf{f}(t).$$

where $\mathbf{f(t)} = \dfrac{\mathbf{h(t)}}{\sum_{t} \mathbf{h(t)}}$ is the inter-reference interval density function. However, there are two reasons why these formulas are inappropriate for the page reference strings of database queries:

(1) These formulas were obtained from asymptotic derivations. Although they provide good approximations for long reference strings, they may provide inaccurate results for the buffer reference string of a database query which is usually several orders of magnitude shorter than a typical virtual address string.

(2) The formulas were derived by treating each page reference as one basic time unit. This assumption is reasonable for address trace strings of programs since the instruction execution times of a machine are usually similar. However, intervals between successive page references (to the buffer pool) of a database query can vary

significantly.

Considering these factors, we have derived a new set of formulas for calculating working set functions of database reference strings. Let $r_0 r_1 \cdots r_k$ represent the page reference string of a database query, and t_i be the time instance when page reference r_i is made. Our formulas for the working set functions are:

$$h(t) = \sum_j h_j(t) \tag{1}$$

$$m(\tau) = \frac{1}{t_k - t_0} \cdot \sum_{t > \tau} h(t) \tag{2}$$

$$s(\tau) = \frac{1}{t_k - t_0} \cdot \sum_{i=1}^{k} (t_i - t_{i-1}) \cdot w(t_i, \tau) \tag{3}$$

where $h_j(t)$ is the number of times the inter-reference interval of page j is t, and $w(t_i,\tau)$ is the working set size at t_i under window size τ. The missing page rate $m(\tau)$ was derived by dividing the total number of page faults of the query by the execution time of the query, whereas the average working set size is the weighted average of the working set sizes at page reference times.

The results are plotted in Figures A.1, A.2 and A.3. As shown in Figure A.1, there are large clusters of inter-reference intervals near or below 10 ms. Figure A.2 shows that the missing page rates drop significantly at similar window sizes. The curves for the working set sizes show that queries which include a non-clustered index as an access path (e.g. queries I and III) generally have a steeper increase in working set size as the window size increases. For queries with high memory demands, e.g. query VI, the increase in working set size is also rapid for window sizes that cover only part of the main loop.

Now we shall discuss how we selected the window sizes for the working set algorithm tested in Chapter 4. Based on a page residency argument, Denning suggested in his early

paper on the working set model [Denn68a] that the selection of the window size should be $\tau = 2\mathbf{T}$, where **T** is the swap time of the paging device. Prieve [Prie73] later showed that the residency argument was incorrect and provided little information on the selection of τ. Although the issue of parameter selection has been addressed in several implementations of the working set algorithm [Dohe70] [Rodr71], it is still unclear what the best solution is. We decided to base our selection on observations of the inter-reference intervals and the missing page rate function. As we mentioned earlier, 10 ms seems to be an appropriate choice for some of the queries. We also selected another window size, 15 ms, for other queries which have larger inter-reference intervals. Although the choices are somewhat arbitrary, these two window sizes seem large enough to cover most localities and yet small enough to avoid wasting buffers on unneeded pages.

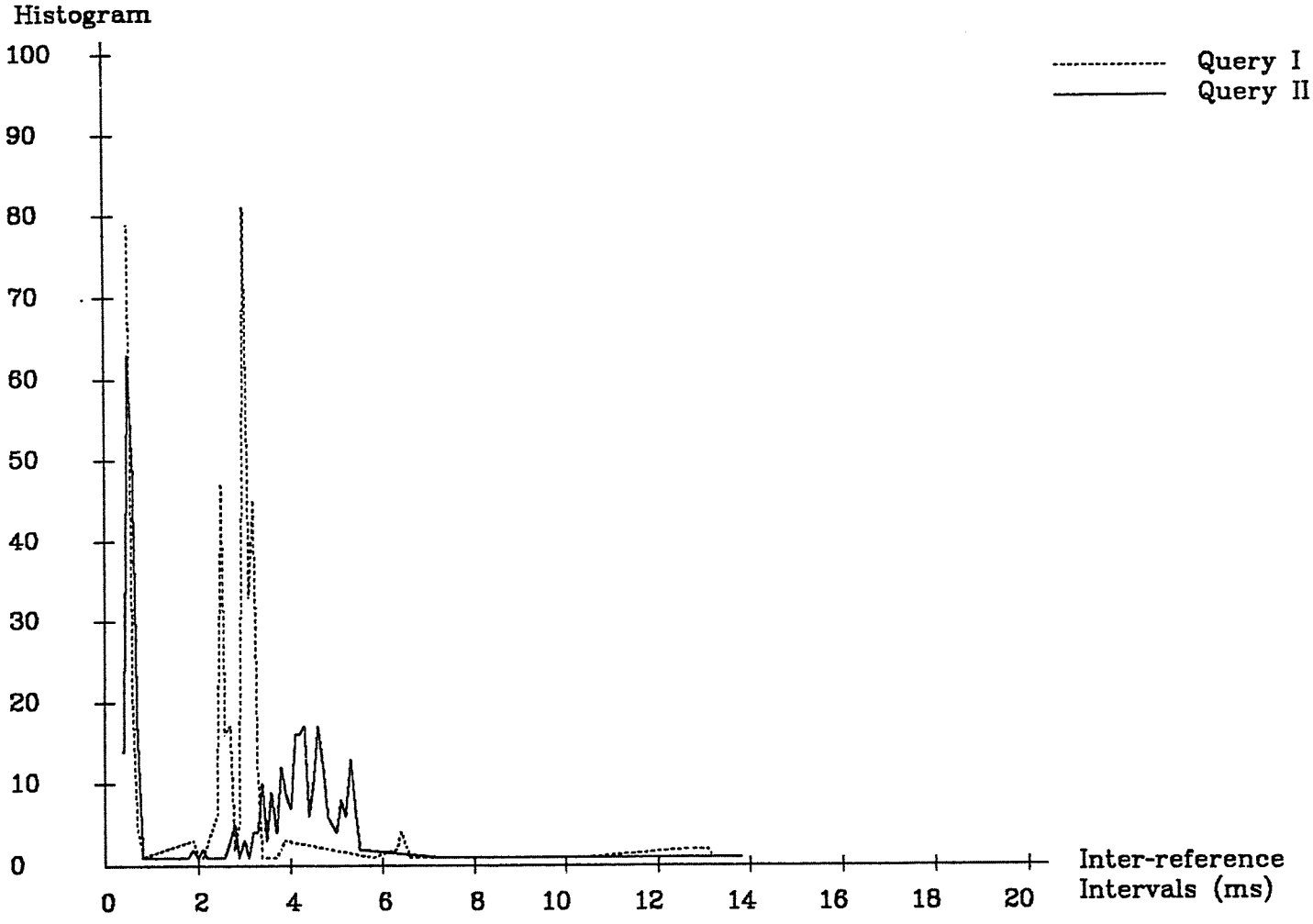

Figure A.1 (a) Analysis of Inter-reference Intervals

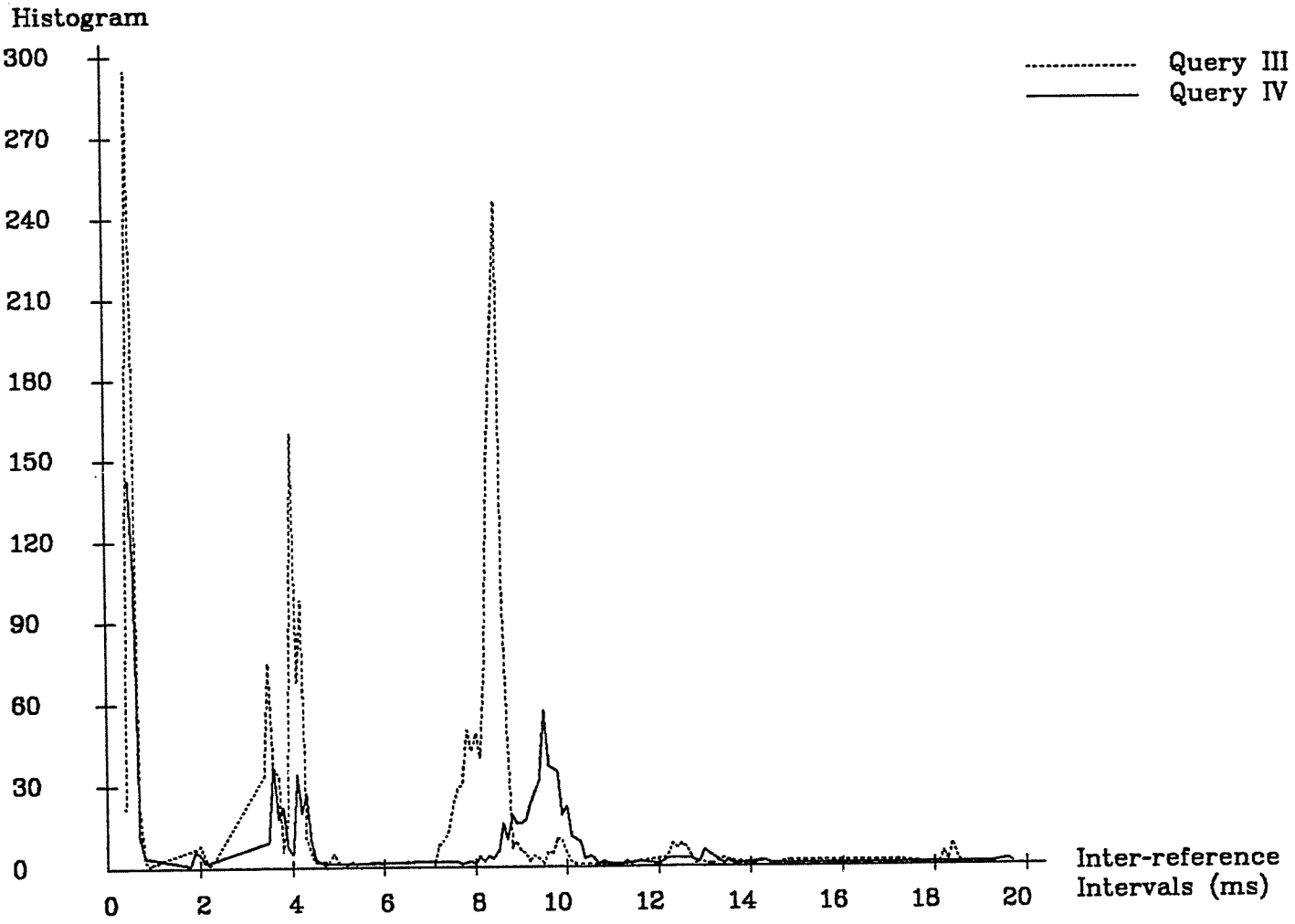

Figure A.1 (b) Analysis of Inter-reference Intervals

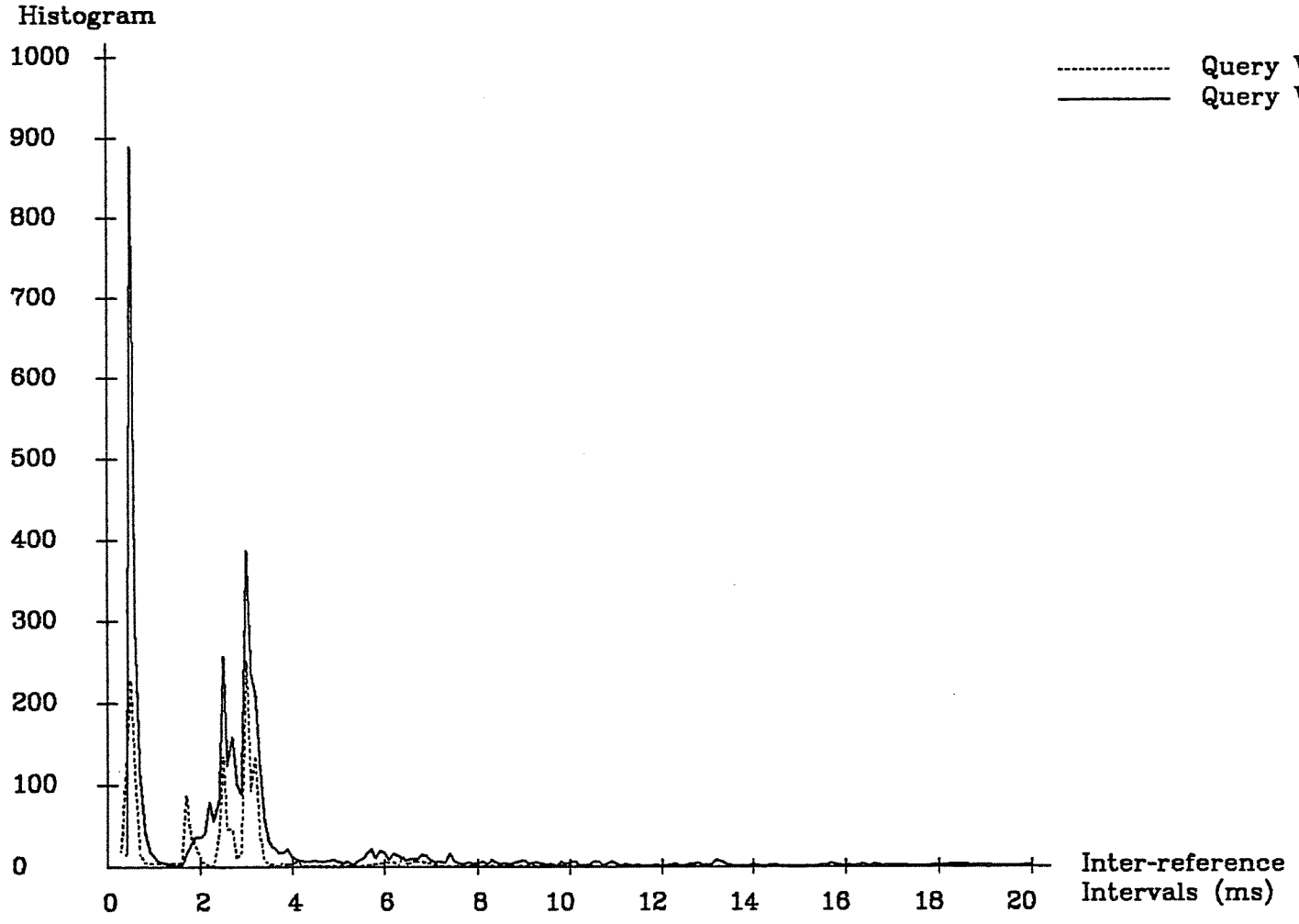

Figure A.1 (c) Analysis of Inter-reference Intervals

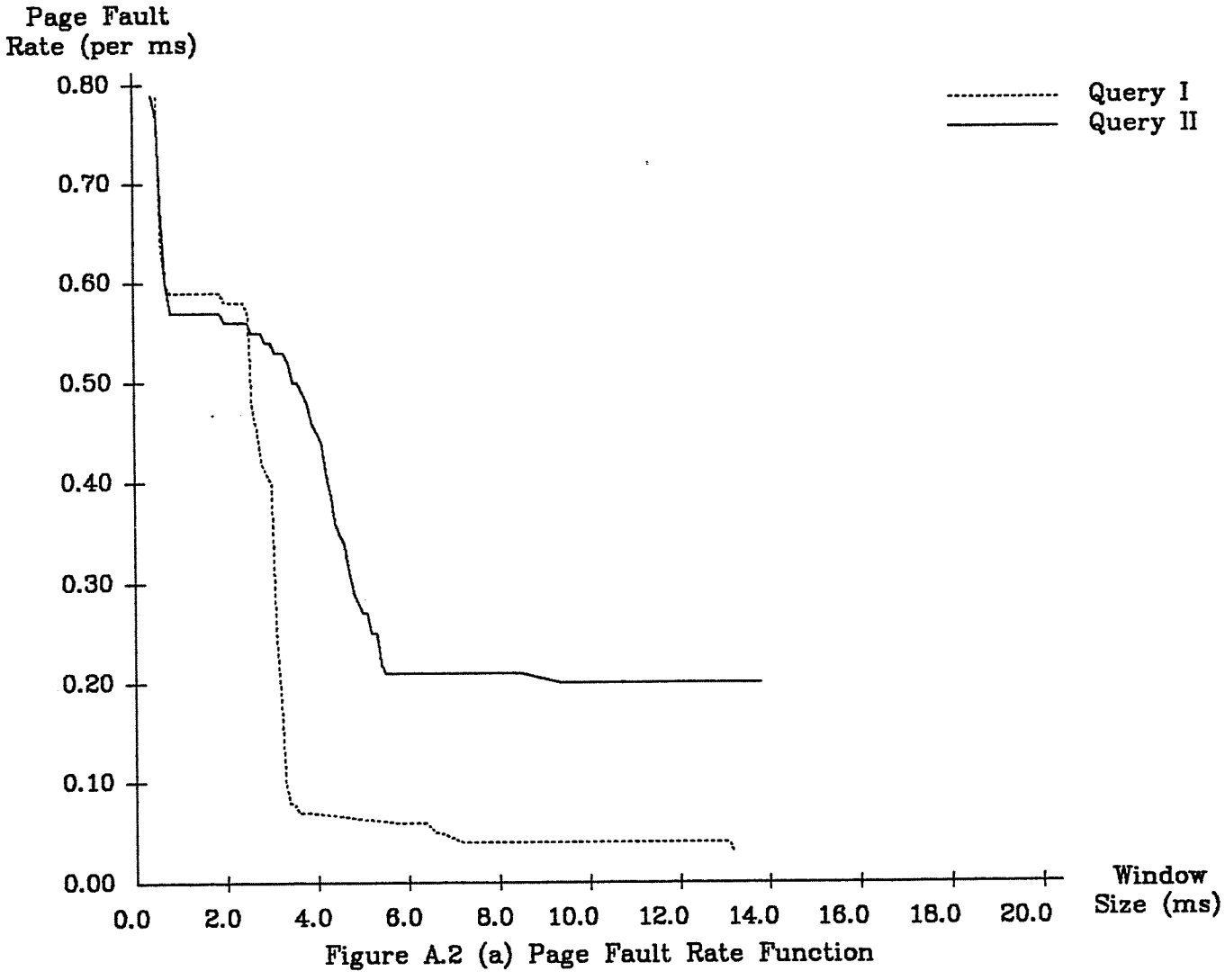

Figure A.2 (a) Page Fault Rate Function

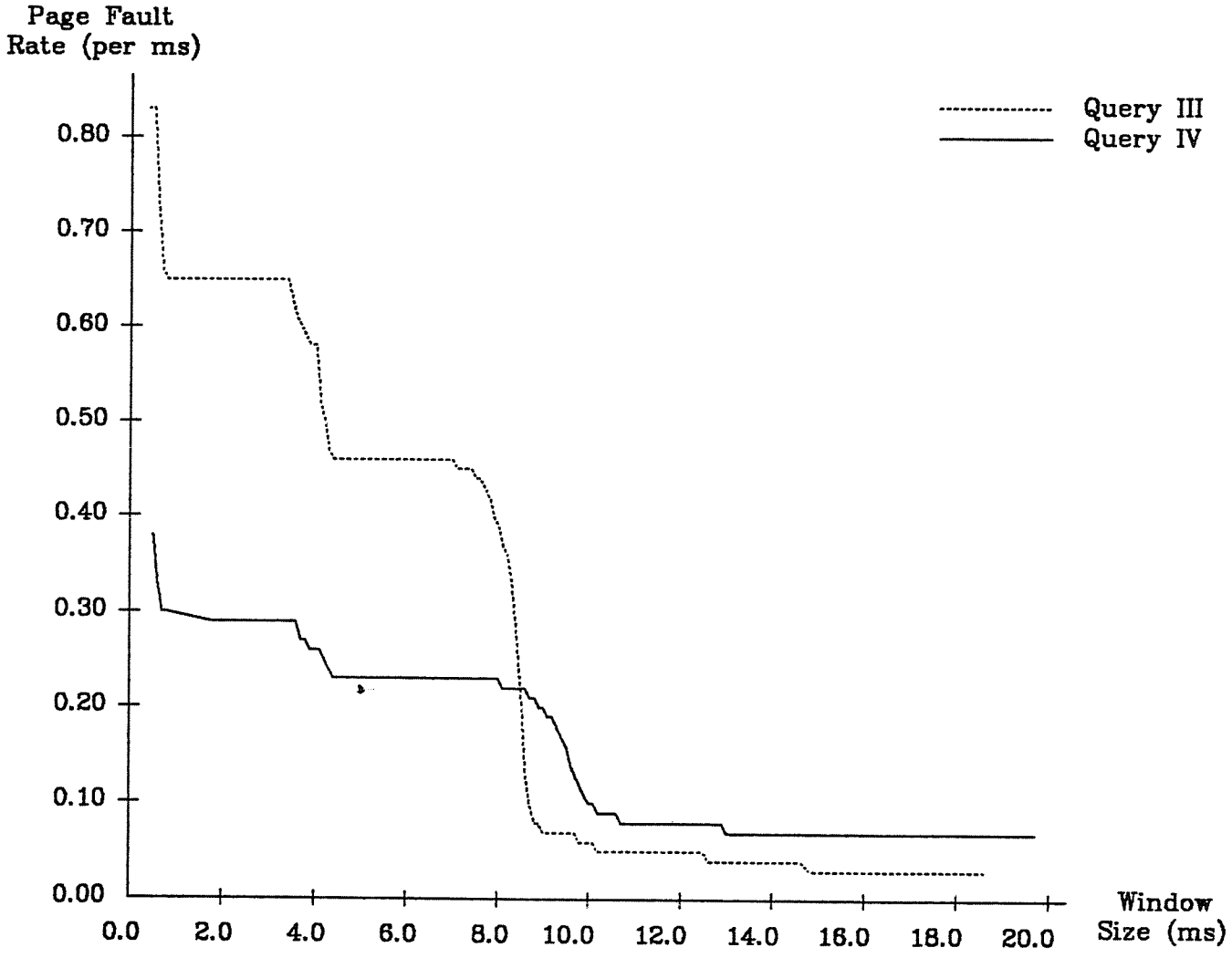

Figure A.2 (b) Page Fault Rate Function

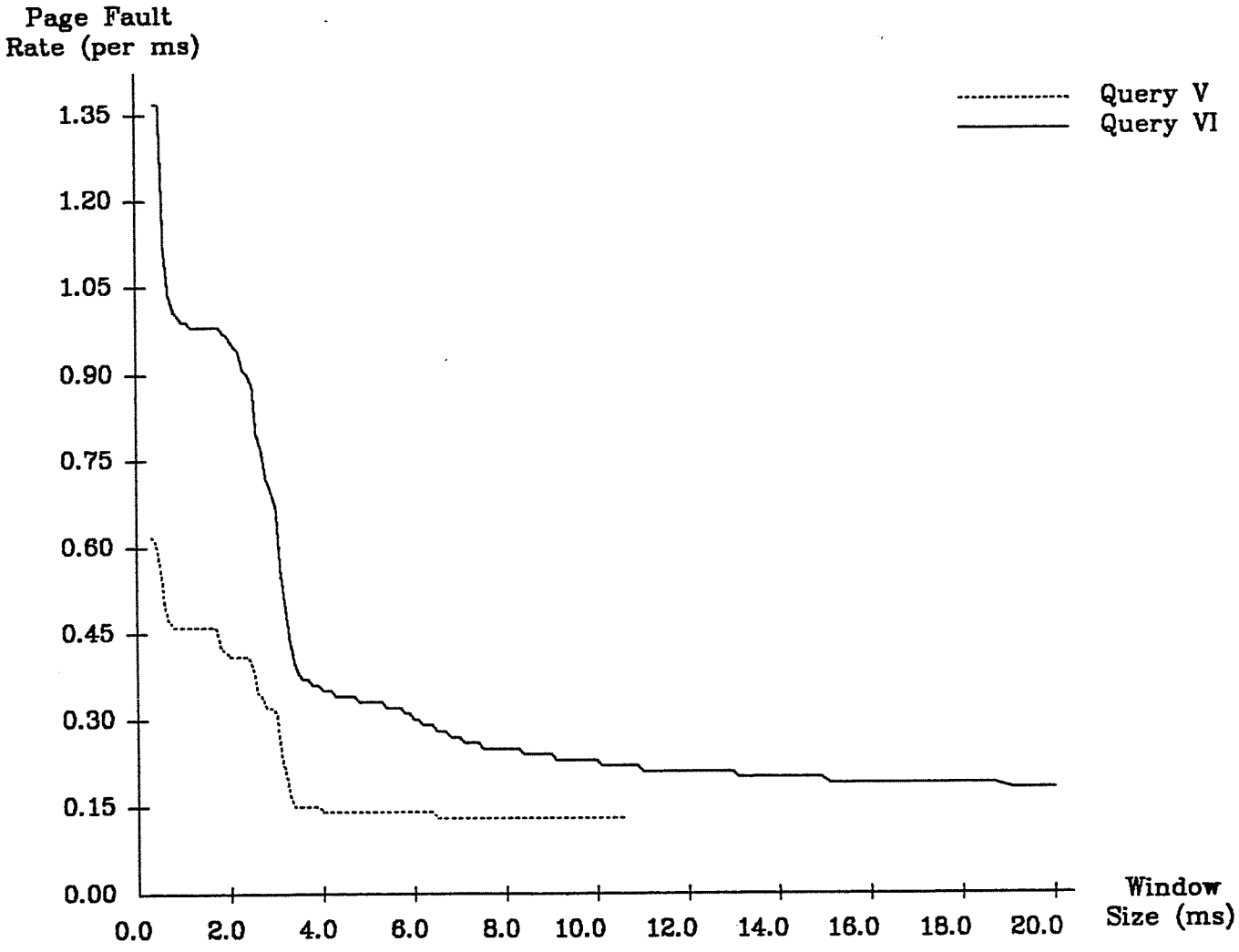

Figure A.2 (c) Page Fault Rate Function

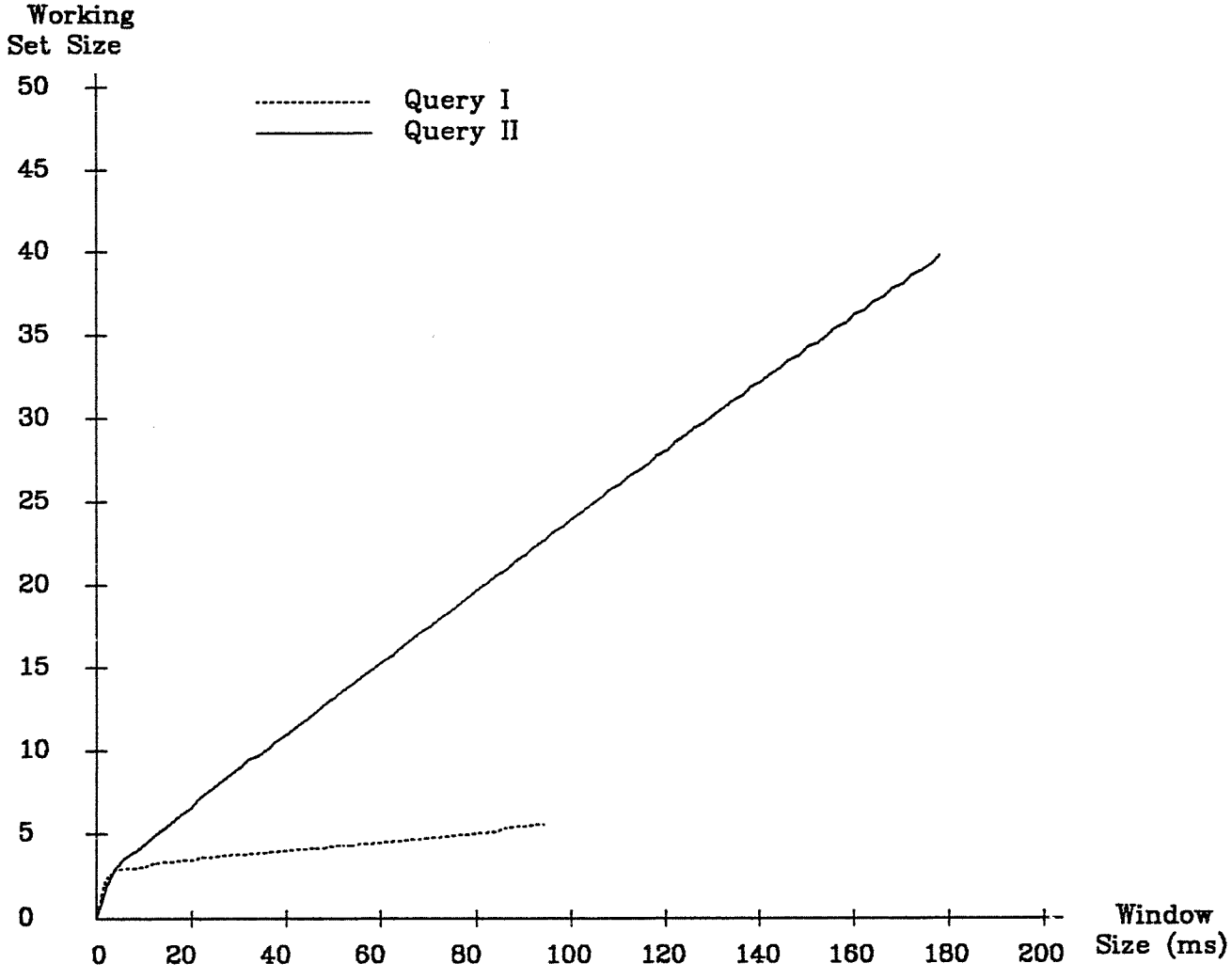

Figure A.3 (a) Working Set Size Analysis

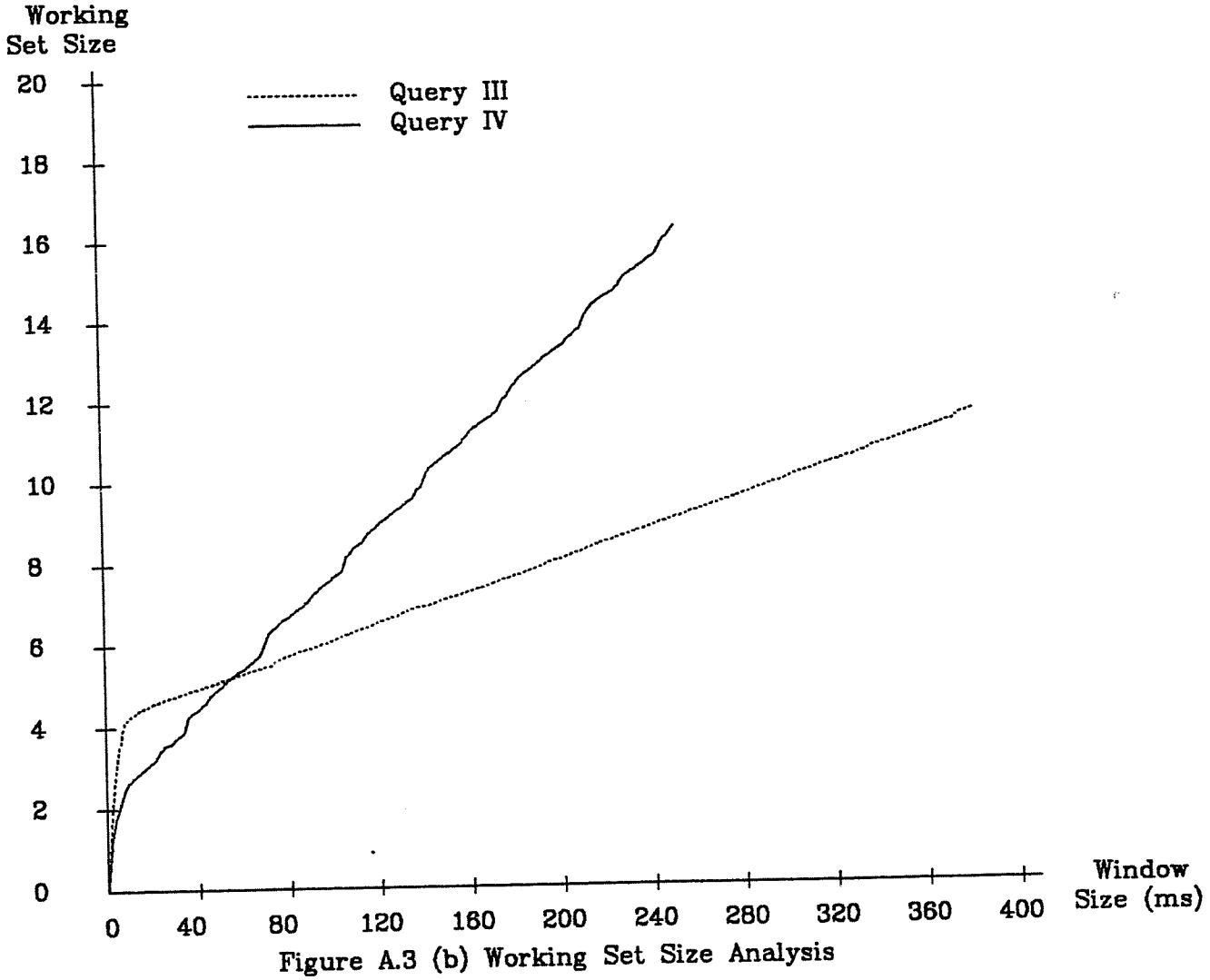

Figure A.3 (b) Working Set Size Analysis

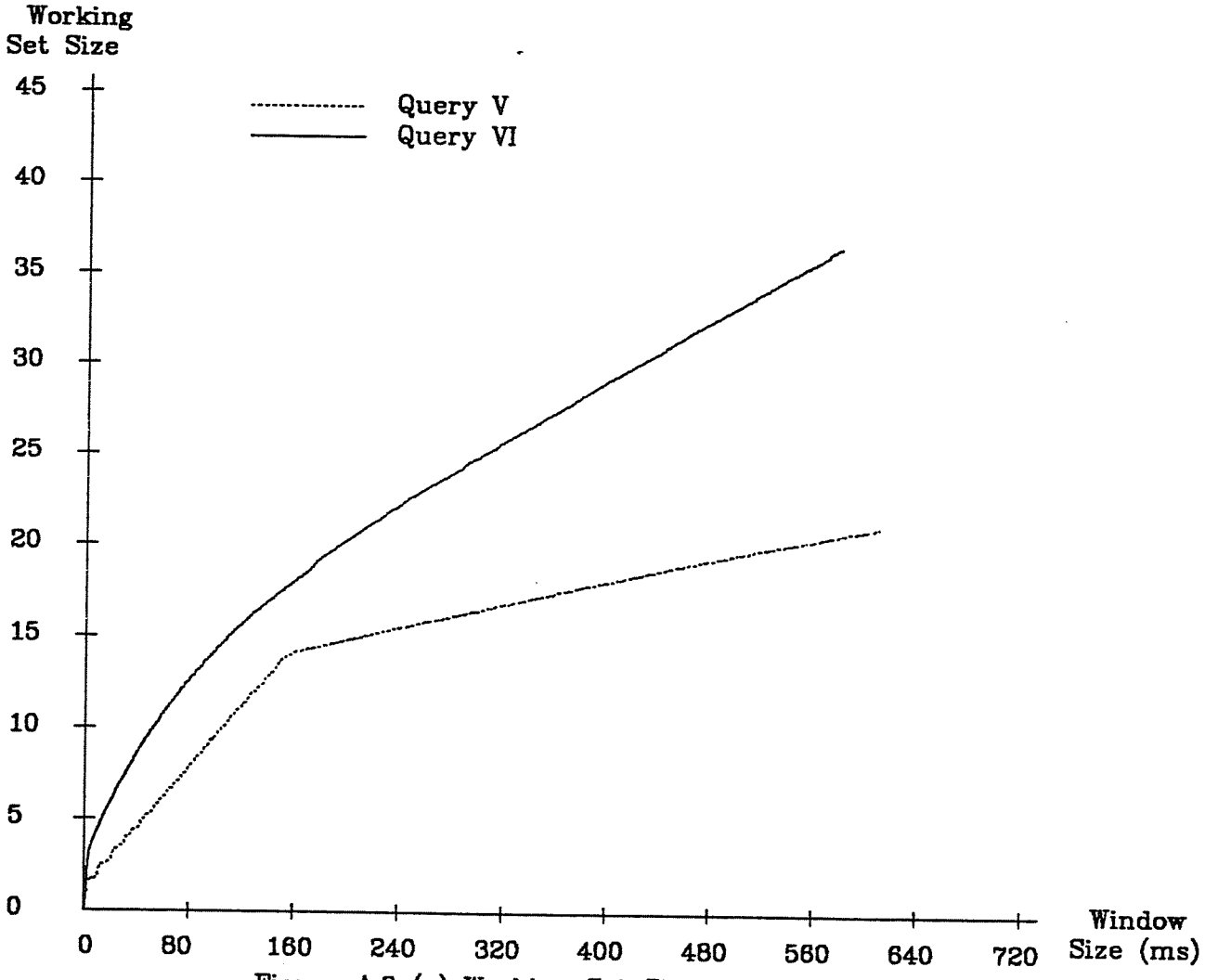

Figure A.3 (c) Working Set Size Analysis

REFERENCES

[Aho71] Aho, Alfred V., Denning, Peter J., and Ullman, Jeffery D., "Principles of Optimal Page Replacement," *Journal of the ACM*, Vol. 18, No. 1, pp. 80-93, January 1971.

[Aror73] Arora, Sant R. and Kachhal, Swatantra K., "Optimization of Design Parameters in a Virtual Memory System," *Proceedings of Computer Science and Statistics 7th Annual Symposium on the interface*, pp. 92-99, October 1973.

[Arvi73] Arvind,, Kain, R.Y., and Sadeh, E., "On Reference String Generation Processes," *Proceedings of the 4th Symposium on Operating Systems Principles*, pp. 80-87, October 1973.

[Astr76] Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J. N., Griffiths, P. P., King, W. F., Lorie, R. A., McJones, P. R., Mehl, J. W., Putzolu, G. R., Traiger, I. L., Wade, B. W., and Watson, V., "System R: A Relational Approach to Database Management," *ACM Transactions on Database Systems*, Vol. 1, No. 2, pp. 97-137, June 1976.

[Baba82] Babaoglu, Ozalp, "Hierarchical Replacement Decisions in Hierarchical Stores," *Proceedings of the 1982 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 11-19, August 1982.

[Bade75] Badel, Marc, Gelenbe, Erol, Leroudier, Jacques, and Potier, Dominique, "Adaptive Optimization of a Time-Sharing System's Performance," *Proceedings of the IEEE*, Vol. 63, No. 6, pp. 958-965, June 1975.

[Bask73] Baskett, Forest, *Proceedings of Computer Science and Statistics 7th Annual Symposium on the interface*, pp. 58-64, October 1973.

[Bayl68] Baylis, M.H., Fletcher, D.G., and Howarth, D.J., "Paging Studies Made on the I.C.T. Atlas Computer," in *Proceedings of IFIP Congress (Information Processing 68)*, pp. 831-837, North Holland Publishing Company, Amsterdam, August 1968.

[Bela66] Belady, L.A., "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal*, Vol. 5, No. 2, pp. 78-101, 1966 .

[Bela69] Belady, L.A. and Kuehner, C.J., "Dynamic Space-Sharing in Computer Systems," *Communications of the ACM* , Vol. 12, No. 5, pp. 282-288, May 1969.

[Bela73] Belady, L.A. and Tsao, R.F., "Memory Allocation and Program Behavior under Multiprogramming," *Proceedings of Computer Science and Statistics 7th Annual Symposium on the interface*, pp. 72-78, October 1973.

[Bens72] Bensoussan, A., Clingen, C.T., and Daley, R.C., "The Multics Virtual Memory : Concepts and Design," *Communications of the ACM* , Vol. 15, No. 5, pp. 308-318, May 1972.

[Bitt83] Bitton, Dina, DeWitt, David J., and Turbyfill, Carolyn, "Benchmarking Database Systems: A Systematic Approach," *Proceedings of the Ninth International Conference on Very Large Data Bases*, November 1983.

[Blak82] Blake, Russ, "Optimal Control of Thrashing," *Proceedings of the 1982 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 1-10, August 1982.

[Blas77] Blasgen, M. W. and Eswaran, K. P., "Storage and Access in Relational Data Base," *IBM System Journals*, No. 4, pp. 363-377, 1977.

[Bora84] Boral, Haran and DeWitt, David J., "A Methodology For Database System Performance Evaluation," *Proceedings of the International Conference on Management of Data*, pp. 176-185, ACM, Boston, June 1984.

[Bran74] Brandwain, A., Buzen, J., Gelenbe, E., and Potier, D., "A Model of Performance for Virtual Memory Systems (Abstract)," *Proceedings of the 1974 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, p. 9, October 1974.

[Braw68] Brawn, Barbara S. and , Frances G. Gustavson, "Program Behavior in a Paging Environment ," in *Proceedings of the 1968 AFIPS Fall Joint Computer Conference*, Vol. 33, pp. 1019-1032, AFIPS Press, Montvale, N.J., December 1968.

[Braw70] Brawn, Barbara S., , Frances G. Gustavson, and Mankin, Efrem S., "Sorting in a Paging Environment," *Communications of the ACM* , Vol. 13, No. 8, pp. 483-494, August 1970.

[Burk63] Burks, A. W., Goldstine, H., and Neumann, J. Von, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," in *Collected Works of John Von Neumann*, ed. A. H. Taub, Vol. 5, The Macmillan Company, New York, 1963.

[Carr81] Carr, Richard W. and Hennessy, John L., "WSClock - A Simple and Effective Algorithm for Virtual Memory Management," *Proceedings of the 8th Symposium on Operating Systems Principles.*, pp. 87-95, September, 1981.

[Cham73] Chamberlin, D.D., Fuller, S.H., and Liu, L.Y., "An Analysis of Page Allocation Strategies for Multiprogramming Systems with Virtual Memory," *IBM Journal of Research and Development*, Vol. 17, No. 5, pp. 404-412, September 1973.

[Cham81] Chamberlin, D. D., Astrahan, M. M., King, W. F., Lorie, R. A., Mehl, J. W., Price, T. G., Schkolnick, M., Selinger, P. Griffiths, Slutz, D. R., Wade, B. W., and Yost, R. A., "Support for Repetitive Transactions and Ad Hoc Queries in System R," *ACM Transactions on Database Systems*, Vol. 6, No. 1, pp. 70-94, March 1981.

[Chou83] Chou, Hong-Tai, DeWitt, David J., Katz, Randy H., and Klug, Anthony C., "Design and Implementation of the Wisconsin Storage System," Computer Sciences Technical Report #524, Department of Computer Sciences, University of Wisconsin, Madison, November 1983.

[Chu72] Chu, Wesley W. and Opderbeck, Holger, "The Page Fault Frequency Replacement Algorithm," in *Proceedings of the 1972 AFIPS Fall Joint Computer Conference*, Vol. 41, pp. 579-609, AFIPS Press, Montvale, N.J., December 1972.

[Coff68] Coffman, E.G. and Varian, L.C., "Further Experimental Data on the Behavior of Programs in a Paging Environment," *Communications of the ACM* , Vol. 11, No. 7, pp. 471-474, July 1968.

[Coff72] Coffman, E.G. Jr. and Jr., Thomas A. Ryan, "A Study of Storage Partitioning Using a Mathematical Model of Locality," *Communications of the ACM* , Vol. 15, No. 3, pp. 185-190, March 1972.

[Coff73] Coffman, Edward F. Jr. and Denning, Peter J., in *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, 1973.

[Come79] Comer, Douglas, "The Ubiquitous B-Tree," *ACM Computing Surveys*, Vol. 11, No. 2, pp. 121-137, June 1979.

[Corb68] Corbato, F.J., "A Paging Experiment with the Multics System," MIT Project MAC Report MAC-M-384, May 1968.

[Cour75] Courtois, P.J., "Decomposability, Instabilities, and Saturation in Multiprogramming Systems," *Communications of the ACM* , Vol. 18, No. 7, pp. 371-377, July 1975.

[Dale68] Daley, Robert C. and Dennis, Jack B., "Virtual Memory, Processes, and Sharing in MULTICS," *Communications of the ACM* , Vol. 11, No. 5, pp. 306-312, May 1968.

[Denn68a] Denning, Peter J., "The Working Set Model for Program Behavior," *Communications of the ACM* , Vol. 11, No. 5, pp. 323-333, May 1968.

[Denn68b] Denning, Peter J., "Thrashing: Its Causes and Prevention," in *Proceedings of the 1968 AFIPS Fall Joint Computer Conference*, Vol. 33, pp. 915-922, AFIPS Press, Montvale, N.J., December 1968.

[Denn70] Denning, Peter J., "Virtual Memory," *ACM Computing Surveys*, pp. 154-189, September 1970.

[Denn72a] Denning, Peter J. and Schwartz, Stuart C., "Properties of the Working-Set Model," *Communications of the ACM* , Vol. 15, No. 3, pp. 191-198, March 1972.

[Denn72b] Denning, Peter J., Spirn, Jeffery R., and Savage, John E., "Some Thoughts about Locality in Program Behavior," *Proceedings of the Symposium on Computer-Communications Networks and Teletraffic*, pp. 101-112, Polytechnic Institute of Brooklyn, April 1972.

[Denn73] Denning, Peter J., "Dynamic Storage Partitioning," *Proceedings of the 4th Symposium on Operating Systems Principles*, pp. 73-79, New York, October 1973.

[Denn75a] Denning, Peter J. and Graham, G. Scott, "Multiprogrammed Memory Management," *Proceedings of the IEEE*, Vol. 63, No. 6, pp. 924-939, June 1975.

[Denn75b] Denning, Peter J. and Kahn, Kevin C., "A Study of Program Locality and Lifetime Functions," *Proceedings of the 5th Symposium on Operating Systems Principles*, pp. 207-216, Austin, Taxis, November 1975.

[Denn76a] Denning, Peter J. and Kahn, Kevin C., "An L=S Criterion for Optimal Multi-Programming," *Proceedings of the international Symposium on Computer Performance Modeling, Measurement, and Evaluation. ACM SIGMETRICS (IFIP WG. 7.3)*, pp. 219-229, Cambridge, March 1976.

[Denn76b] Denning, Peter J., Kahn, Kevin C., Leroudier, Jacques, Potier, Dominique, and Suri, Rajan, "Optimal Multiprogramming," *Acta Informatica*, Vol. 7, No. 2, pp. 197-216, 1976.

[Denn78a] Denning, Peter J., "Optimal Multiprogrammed Memory Management," in *Current Trends in Programming Methodology. Vol.III Software Modeling*, ed. Raymond T. Yeh, pp. 298-322, Prentice-Hall, Englewood Cliffs, 1978.

[Denn78b] Denning, Peter J. and Slutz, Donald R., "Generalized Working Sets for Segment Reference Strings," *Communications of the ACM* , Vol. 21, No. 9, pp. 750-759, September 1978.

[Denn80] Denning, Peter J., "Working Sets Past and Present," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 1, pp. 64-84, January 1980.

[DeWi84a] DeWitt, David J., Katz, Randy H., Olken, Frank, Shapiro, Leonard D., Stonebraker, Michael R., and Wood, David, "Implementation Techniques for Main Memory Database Systems," *Proceedings of the International Conference on Management of Data*, pp. 1-8, ACM, Boston, June 1984.

[DeWi84b] DeWitt, David J., Finkel, Raphael, and Solomon, Marvin, "The CRYSTAL Multicomputer: Design and Implementation Experience," Computer Sciences Technical Report #553, Department of Computer Sciences, University of Wisconsin, Madison, September 1984.

[Digi80] Digital Equipment Corporation, *VAX Hardware Handbook*, 1980.

[Digi81] Digital Equipment Corporation, *VAX Architecture*, 1981.

[Dohe70] Doherty, Walter J., "Scheduling TSS/360 for Responsiveness," in *Proceedings of the 1970 AFIPS Fall Joint Computer Conference*, Vol. 37, pp. 97-111, AFIPS Press, Montvale, N.J., November 1970.

[East75] Easton, M.C., "Model for Interactive Data Base Reference String," *IBM Journal of Research and Development*, pp. 550-556, November 1975.

[East77] Easton, M.C. and Bennett, B.T., "Transient-Free Working-Set Statistics," *Communications of the ACM* , Vol. 20, No. 2, pp. 93-99, February 1977.

[East78] Easton, Malcolm C., "Model for Data Base Reference Strings Based on Behavior of Reference Clusters," *IBM Journal of Research and Development*, Vol. 22, No. 2, pp. 197-202, March 1978.

[East79] Easton, Malcolm C. and Franaszek, Peter A., "Use Bit Scanning in Replacement Decisions," *Transactions on Computers*, Vol. c-28, No. 2, pp. 133-141, IEEE, February 1979.

[Effe84] Effelsberg, Wolfgang and Haerder, Theo, "Principles of Database Buffer Management," *ACM Transactions on Database Systems*, Vol. 9, No. 4, pp. 560-595, December 1984.

[Elha84] Elhardt, Klaus and Bayer, Rudolf, "A Database Cache For High Performance and Fast Restart in Database Systems," *ACM Transactions on Database Systems*, Vol. 9, No. 4, pp. 503-525, December 1984.

[Fagi79] Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R., "Extendible Hashing - A fast Access Method for Dynamic Files," *ACM Transactions on Database Systems*, Vol. 4, No. 3, pp. 315-344, September 1979.

[Fern78] Fernandez, E.B., Lang, T., and Wood, C., "Effect of Replacement Algorithms on a Paged Buffer Database System," *IBM Journal of Research and Development*, Vol. 22, No. 2, pp. 185-196, March 1978.

[Fine66] Fine, Gerald H., Jackson, Calvin W., and McIssac, Paul V., "Dynamic Program Behavior Under Paging," in *Proceedings of 21st National Conference of the Association for Computing Machinery*, pp. 223-228, Thompson Book Company, Washington D.C., 1966.

[Foge74] Fogel, Marc H., "The VMOS Paging Algorithm, a Practical Implementation of the Working Set Model," *ACM Operating System Review*, Vol. 8, January 1974.

[Foth61] Fotheringham, John, "Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store," *Communications of the ACM* , Vol. 4, No. 10, pp. 435-436, October 1961.

[Fuji82] Fujitsu Limited, *M2351A/AF Mini-Disk Drive CE manual*, 1982.

[Gele73] Gelenbe, Erol, "A Unified Approach to the Evaluation of a Class of Replacement Algorithms," *IEEE Transactions on Computers*, Vol. C-22, No. 6, pp. 611-618, June 1973.

[Ghan75a] Ghanem, M.Z., "Dynamic Partitioning for of the Main Memory Using the Working Set Concept," *IBM Journal of Research and Development*, Vol. 19, No. 5, pp. 445-450, September 1975.

[Ghan75b] Ghanem, M.Z., "Study of Memory Partitioning for Multiprogramming Systems with Virtual Memory," *IBM Journal of Research and Development*, Vol. 19, No. 5, pp. 451-457, September 1975.

[Gold74] Goldberg, Robert R. and Hassinger, Robert, "The Double Paging Anomaly," in *Proceedings of the National Computer Conference, 1974*, pp. 195-199, AFIPS Press, Montvale, N.J., May 1974.

[Grah74] Graham, G. Scott and Denning, Peter J., *Proceedings of the 1974 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 1-8, October 1974.

[Gray78] Gray, James N., "Notes on Database Operating Systems," IBM Research Report RJ2188, San Jose, Ca, Feburary 1978.

[Grit75] Grit, D.H. and Kain, R.Y., "An Analysis of a Use Bit Page Replacement Algorithm," *Proceedings of ACM Annual Conference*, pp. 187-192, Minneapolis, MN, October 1975.

[Jose70] Joseph, M., "An Analysis of Paging and Program Behaviour," *The Computer Journal*, Vol. 13, No. 1, pp. 48-54, February 1970.

[Kapl80] Kaplan, Julio A., "Buffer Management Policies in a Database Environment," Master Report, UC Berkeley, 1980.

[Kern78] Kernighan, Brian W. and Ritchie, Dennis M., in *The C Programming Language*, Prentice-Hall, Inc., New Jersy, 1978.

[Khos84] Khoshafian, Setrag Nishan, "A building Blocks Approach to Statistical Databases," Ph.D Thesis, Computer Sciences Department, University of Wisconsin, Madison, May 1984.

[Kilb62] Kilburn, T., Edwards, D.B.G., Lanigan, M.J., and Sumner, F.H., "One Level Storage System," *IRE Transactions on Electronic Computers*, Vol. EC-11, No. 2, pp. 223-235, April 1962.

[King71] King, W. F. III, "Analysis of Demand Paging Algorithms ," in *Proceedings of IFIP Congress (Information Processing 71)*, pp. 485-490, North Holland Publishing Company, Amsterdam, August 1971.

[Knut73] Knuth, Donald E., in *The Art of Computer Programming Vol.3: Sorting and Searching*, Addison Wesley, Reading, Mass., 1973.

[Lang77] Lang, Tomas, Wood, Christopher, and Fernandez, Ieduardo B., "Database Buffer Paging in Virtual Storage Systems," *ACM Transactions on Database Systems*, Vol. 2, No. 4, pp. 339-351, December, 1977.

[Lenf75] Lenfant, J. and Burgevin, P., "Empirical Data on Program Behaviour," in *Proceedings of the International Computing Symposium*, ed. E. Gelenbe and D. Potier, pp. 163-169, North-Holland Publishing Company, June 1975.

[Lero76] Leroudier, J. and Potier, D., "Principles of Optimality for Multi-Programming," *Proceedings of the international Symposium on Computer Performance Modeling, Measurement, and Evaluation, ACM SIGMETRICS (IFIP WG. 7.3)*, pp. 211-218, Cambridge, March 1976.

[Lew76] Lew, Art, "Optimal Control of Demand-Paging Systems," *Information Sciences*, Vol. 10, No. 4, pp. 319-330, May 1976.

[Lewi73] Lewis, P.A. W. and Shedler, G.S., "Empirically Derived Micromodels for Sequences of Page Exceptions ," *IBM Journal of Research and Development*, Vol. 17, pp. 86-100, March 1973.

[Litw80] Litwin, Witold, "Linear Virtual Hashing: A New Tool for Files and Tables Implementation," *Proceedings of the Sixth International Conference on Very Large Data Bases*, pp. 212-223, Montreal, 1980.

[Lome83] Lomet, David B., "A High Performance, Universal, Key Associative Access Method," *Proceedings of the International Conference on Management of Data*, pp. 120-133, ACM, San Jose, May 1983.

[Madi76] Madison, Wayne A. and Batson, Alan P., "Characteristics of Program Localities," *Communications of the ACM* , Vol. 19, No. 5, pp. 285-294, May 1976.

[Madn74] Madnick, Stuart E. and Donovan, John J., in *Operating Systems*, McGraw-Hill Inc., 1974.

[Marc81] March, Salvatore T., Severance, Dennis G., and Wilens, Michael, "Frame Memory: A Storage Architecture to Support Rapid Design and Implementation of Efficient Databases," *ACM Transactions on Database Systems*, Vol. 6, No. 3, pp. 441-463, September 1981.

[Mart75] Martin, James, in *Computer Data-Base Organization*, Prentice-Hall, Inc., 1975.

[Matt68] Mattson, R.L., Jacob, J.P., and Howarth, D.J., "Optimization Studies for Computer Systems with virtual Memory," in *Proceedings of IFIP Congress (Information Processing 68)*, pp. 846-852, North Holland Publishing Company, Amsterdam, August 1968.

[Matt70] Mattson, R.L., Gecsei, J., Slutz, D.R., and Traiger, I.L., "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal*, Vol. 9, No. 2, pp. 78-117, 1970.

[Nybe84] Nyberg, Chris, "Disk Scheduling and Cache Replacement for a Database Machine," Master Report, UC Berkeley, July, 1984.

[Oden72] Oden, P.H. and Shedler, G.S., "A Model of Memory Contention in a Paging Machine," *Communications of the ACM* , Vol. 15, No. 8, pp. 761-771, August 1972.

[Oliv74] Oliver, N. A., "Experimental Data on Page Replacement Algorithm ," in *Proceedings of the National Computer Conference, 1974*, pp. 179-184, AFIPS Press, Montvale, N.J., May 1974.

[Opde74] Opderbeck, Holger and Chu, Wesley W., "Performance of the Page Fault Frequency Replacement Algorithm in a Multiprogramming Environment," in *Proceedings of IFIP Congress, Information Processing 74*, pp. 235-241, North Holland Publishing Company, Amsterdam, August 1974.

[Opde75] Opderbeck, Holger and Chu, Wesley W., "The Renewal Model for Program Behavior," *SIAM Journal of Computing*, Vol. 4, No. 3, pp. 356-374, September 1975.

[Pome71] Pomeranz, John E., "Paging with Fewest Expected Replacements," in *Proceedings of IFIP Congress (Information Processing 71)*, pp. 491-493, North Holland Publishing Company, Amsterdam, August 1971.

[Poti77] Potier, Dominique, "Analysis of Demand Paging Policies with Swapped Working Sets," in *Measuring, Modelling and Evaluating Computer Systems*, ed. H. Beilner and E. Gelenbe, pp. 233-237, North-Holland Publishing Company, 1977.

[Prie73] Prieve, B. G., "Using Page Residency to Select the Working Set Parameter," *Communications of the ACM* , Vol. 16, pp. 619-620, October 1973.

[Prie76] Prieve, Barton G. and Fabry, R.S., "VMIN-An Optimal Variable-Space Page Replacement Algorithm," *Communications of the ACM* , Vol. 19, No. 5, pp. 295-297, May 1976.

[Reit76] Reiter, Allen, "A Study of Buffer Management Policies For Data Management Systems," Technical Summary Report # 1619, Mathematics Research Center, University of Wisconsin-Madison, March, 1976.

[Robi81] Robinson, J. T., "The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proceedings of the International Conference on Management of Data*, pp. 10-18, ACM, April 1981.

[Rodr71] Rodriguez-Rosell, Juan, "Experimental Data on How Program Behavior Affects the Choice of Scheduler Parameters," *Proceedings of the 3rd Symposium on Operating Systems Principles*, pp. 156-163, October 1971.

[Rodr72] Rodriguez-Rosell, Juan and Dupuy, Jean-Pierre, "The Evaluation of a time-sharing page demand system," in *Proceedings of the 1972 AFIPS Spring Joint Computer Conference*, Vol. 40, pp. 759-765, AFIPS Press, Montvale, N.J., May 1972.

[Rodr73] Rodriguez-Rosell, Juan, "Empirical Working Set Behavior," *Communications of the ACM* , Vol. 16, No. 9, pp. 556-560, September 1973.

[Rodr75] Rodriguez-Rosell, Juan and Hildebrand, David, "A Framework for Evaluation of Data Base Systems," in *Proceedings of International Computing Symposium 1975*, ed. E. Gelenbe and D. Potier, pp. 77-81, North Holland Publishing Company, Amsterdam, June 1975.

[Rodr76] Rodriguez-Rosell, Juan, "Empirical Data Reference Behavior in Data Base Systems," *IEEE Computer*, pp. 9-13, November, 1976.

[Sacc82] Sacco, Giovanni Maria and Schkolnick, Mario, "A Mechanism For Managing the Buffer Pool In A Relational Database System Using the Hot Set Model," *Proceedings of the 8th International Conference on Very Large Data Bases*, pp. 257-262, Mexico City, September 1982.

[Sarg76] Sargent, Robert G., "Statistical Analysis of Simulation Output Data," *Proceedings of ACM Symposium on Simulation of Computer Systems*, pp. 39-50, August 1976.

[Saue81] Sauer, Charles H. and Chandy, K. Mani, in *Computer Systems Performance Modeling*, Prentice-Hall, Inc., 1981.

[Seli79] Selinger, P. Griffiths, Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G., "Access Path Selection in a Relational Database System," *Proceedings of the International Conference on Management of Data*, pp. 23-34, ACM, Boston, 1979.

[Shed72] Shedler, G.S. and Tung, C., "Locality in Page Reference Strings," *SIAM Journal of Computing*, Vol. 1, No. 3, pp. 218-241, September 1972.

[Shem66] Shemer, J.E. and Shippey, G.A., "Statistical Analysis of Paged and Segmented Computer Systems," *IEEE Transactions on Electronic Computers*, Vol. EC-15, No. 6, pp. 855-863, December 1966.

[Sher73] Sherman, Stephen W. and Browne, J.C., "Trace Driven Modeling: Review and Overview," *Proceedings of ACM Symposium on Simulation of Computer Systems*, pp. 201-207, June 1973.

[Sher76a] Sherman, Stephen W. and Brice, Richard S., "I/O Buffer Performance in a Virtual Memory System," *Proceedings of ACM Symposium on Simulation of Computer Systems*, pp. 25-35, August, 1976.

[Sher76b] Sherman, Stephen W. and Brice, Richard S., "Performance of a Database Manager in a Virtual Memory System," *ACM Transactions on Database Systems*, Vol. 1, No. 4, pp. 317-343, December 1976.

[Smit76a] Smith, Alan J., "A Modified Working Set Paging Algorithm," *IEEE Transactions on Computers*, Vol. C-25, No. 9, pp. 907-914, September 1976.

[Smit76b] Smith, Alan Jay, "Analysis of the Optimal, Look-Ahead Demand Paging Algorithms," *SIAM Journal of Computing*, Vol. 5, No. 4, pp. 743-757, December 1976.

[Smit80] Smith, Alan Jay, "Multiprogramming and Memory Contention," *Software Practice and Experience*, Vol. 10, No. 7, pp. 531-552, July 1980.

[Spir72] Spirn, Jeffery R. and Denning, Peter J., "Experiments with Program Locality," in *Proceedings of the 1972 AFIPS Fall Joint Computer Conference*, Vol. 41, pp. 611-621, AFIPS Press, Montvale, N.J., December 1972.

[Ston76] Stonebraker, Michael, Wong, Eugene, and Kreps, Peter, "The Design and Implementation of INGRES," *ACM Transactions on Database Systems*, Vol. 1, No. 3, pp. 189-222, September 1976.

[Ston81] Stonebraker, Michael, "Operating System Support for Database Management," *Communications of the ACM* , Vol. 24, No. 7 , pp. 412-418, July 1981.

[Ston82] Stonebraker, Michael, Woodfill, John, Ranstrom, Jeff, Murphy, Marguerite, Meyer, Marc, and Allman, Eric, "Performance Enhancements to a Relational Database System ," Initial draft of a paper which appeared in TODS, Vol. 8, No. 2, June, 1983, 1982.

[Ston83] Stonebraker, Michael, Woodfill, John, Ranstrom, Jeff, Murphy, Marguerite, Meyer, Marc, and Allman, Eric, "Performance Enhancements to a Relational Database System," *ACM Transactions on Database Systems*, Vol. 8, No. 2, pp. 167-185, June 1983.

[Thor72] Thorington, John M. Jr. and IRWIN, David J., "An Adaptive Replacement Algorithm for Paged Memory Computer Systems," *IEEE Transactions on Computers*, Vol. C-21, No. 10, pp. 1053-1061, October 1972.

[Tuel76] Tuel, W. G. Jr., "An Analysis of Buffer Paging in Virtual Storage Systems," *IBM Journal of Research and Development*, pp. 518-520, September, 1976.

[Turn81] Turner, Rollins and Levy, Henry, "Segmented FIFO Page Replacement," *Proceedings of the 1981 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 48-51, September, 1981.

[Weiz69] Weizer, Norman and Oppenheimer, G., "Virtual Memory Management in a Paging Environment," in *Proceedings of the 1969 AFIPS Spring Joint Computer Conference*, Vol. 34, pp. 249-256, AFIPS Press, Montvale, N.J., May 1969.

[Whan83] Whang, Kyu-Young and Wiedlerhold, Gio, "Estimating Block Accesses in Database Organizations: A Closed Noniterative Formula," *Communications of the ACM* , Vol. 26, No. 11, pp. 940-947, November 1983.

[Wong76] Wong, Eugene and Youssefi,, "Decomposition - A Strategy for Query Processing," *ACM Transactions on Database Systems*, Vol. 1, No. 3, pp. 223-241, September 1976.

[Yao77] Yao, S.B, "Approximating Block Accesses in Database Organizations," *Communications of the ACM* , Vol. 20, No. 4, pp. 260-261, April 1977.