Information Systems Vol. 18, No. 4, pp. 197–213, 1993
Printed in Great Britain.

0306-4379/93 $6.00 + 0.00

CACHE MANAGEMENT FOR SHARED SEQUENTIAL DATA ACCESS

ERHARD RAHM[1] and DONALD FERGUSON[2]

[1]University of Kaiserslautern, Department of Computer Science, 67653 Kaiserslautern, Germany

[2]IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, U.S.A.

(Received 5 November 1992; in final revised form 10 April 1993)

Abstract—This paper presents a new set of cache management algorithms for shared data objects that are accessed sequentially. I/O delays on sequentially accessed data is a dominant performance factor in many application domains, in particular for batch processing. Our algorithms fall into three classes: replacement, prefetching and scheduling strategies. Our replacement algorithms empirically estimate the rate at which the jobs are proceeding through the data. These velocity estimates are used to project the next reference times for cached data objects and our algorithms replace data with the longest time to re-use. The second type of algorithm performs asynchronous prefetching. This algorithm uses the velocity estimations to predict future cache misses and attempts to preload data to avoid these misses. Finally, we present a simple job scheduling strategy that increases locality of reference between jobs. Our new algorithms are evaluated through a detailed simulation study. Our experiments show that the algorithms substantially improve performance compared to traditional algorithms for cache management.

Key words: Caching, buffer management, replacement strategies, prefetching, sequential data access, batch processing, workload scheduling, performance evaluation

1. INTRODUCTION

This study presents and evaluates new caching algorithms for sequentially accessed data that is being concurrently used by several processes or jobs. This type of access is common in many data processing environments. In this paper, we focus on *batch processing*, which is often dominated by concurrent sequential access to data [1, 2]. Several of our algorithms are applicable to other domains in which sequential scanning of data is common such as query processing or long running transactions in database systems. In this section, we provide a motivation for our algorithms, discuss related work, highlight the new concepts incorporated in our algorithms and outline the organization of the paper.

1.1. Motivation

Batch processing constitutes a major part of most large computer complexes and data centers. Typical batch jobs include daily and monthly report generation, payroll processing and performing bulk updates against databases. These jobs are usually executed at night since they have high resource requirements (CPU, I/O) which would seriously reduce performance for interactive applications. Batch processing, however, faces the problem that more and more computations have to be performed every night since applications and databases grow permanently. Furthermore, the "batch window" (i.e. the time period in which resources are mainly reserved for batch processing) should be as small as possible to allow for increased periods of interactive processing. Optimizing batch performance means minimizing the *elapsed time* required to process a fixed set of jobs.

In order to reduce elapsed time, it is essential to minimize I/O delays. Batch processing offers two characteristics which can be exploited for decreasing I/O delays: (1) dominance of sequential access patterns and (2) *a priori* knowledge of the jobs' resource requirements. These characteristics are only partially utilized in current systems. In this paper, we propose new algorithms which exploit sequentiality of accesses and *a priori* knowledge about which datasets (files, relations, databases) are going to be referenced by each job to decrease the elapsed processing time of batch workloads.

1.2. Related work

Prefetching (prepaging) as a means to utilize spatial locality of reference (sequential access) in order to reduce the number of I/Os has been analyzed in many previous studies. Studies for virtual memory systems often concentrated on preloading of program portions for which sequentiality is anticipated [3–8]. Prefetching means that in the event of a page fault multiple physically adjacent pages are fetched together in addition to the page for which the fault occurred. Critical control parameters are the number of pages to be read together, as well as the page size. Simple prefetching schemes for programs have generally been found to be ineffective [3–5] since pages are often unnecessarily prefetched. More sophisticated strategies which use *a priori* knowledge obtained by analyzing program traces [7], accept user advice or dynamically analyze the program reference behavior can significantly improve performance [5].

Stronger sequentiality and thus more effective prefetching has been observed for database accesses [9, 10]. Sequentiality of access is often a predictable consequence of database organization (clustering of record types) and operations (e.g. table scan). In [11], a static preanalysis of canned transactions is proposed to determine which pages should be prefetched when the transaction is started. Prefetching can improve performance in two ways. Firstly, the I/O delay and thus response time of a query (transaction) can be reduced by caching data prior to the actual access. Secondly, the I/O overhead for fetching n physically clustered pages at once is usually much smaller than n times the cost of bringing in 1 page.

General caching strategies for main memory buffers have been studied primarily for database systems [12–16]. These schemes generally attempt to reduce the number of I/Os by exploiting temporal locality of reference within a transaction and between different transactions. The replacement algorithm used in virtually all existing database systems is the well-known LRU scheme (least recently used). LRU is not expected to be effective for sequential access patterns (spatial locality) because a transaction reads a particular page only once. The *Most Recently Used* (discard after use) policy performs well for sequential access when there is no concurrent data sharing among the jobs. However, this scheme cannot utilize locality between sequential jobs (inter-job locality) and is therefore not appropriate for our application domain. We are not aware of previous caching studies that consider inter-job locality for sequential data access.

1.3. New concepts

In this paper we will describe and evaluate three new cache replacement strategies for sequential data access and compare their performance with LRU and configurations with no data caching. The schemes adopt the following new concepts:

- —For every job and every dataset, the cache manager periodically determines the *velocity* with which the job proceeded through the dataset during the previous observation period.
- —The replacement schemes use the velocities to predict if and when cached data will be referenced in the future. This analysis determines the replacement order. Thus our schemes use the predictive information of references in the future for replacement decisions, while conventional algorithms like LRU solely rely on reference information of the past.
- —One of our schemes also utilizes *a priori* knowledge for replacement decisions. For every dataset, it requires information about how many jobs are going to process it sequentially. During execution of the jobs, the scheme tries to steal pages from datasets with the least number of outstanding (unstarted) jobs.

Apart from the new replacement schemes, two further methods are applied to improve performance:

- —*Prefetching* is employed in two ways. The standard form of prefetching which reads multiple blocks (pages) per I/O is always applied for sequentially accessed datasets. An additional, *asynchronous* form of prefetching is implemented by dedicated prefetch processes which read in non-cached pages expected to be referenced soon. The velocity estimates are used to prefetch pages in the order they are likely to be needed.
- —We also evaluate how performance is affected by improving locality of reference through a

simple *job scheduling* strategy. For this purpose, we employ a method called *Load Partitioning* which uses *a priori* knowledge to schedule jobs accessing the same datasets concurrently.

1.4. Organization

The remainder of this paper is organized as follows. In the next section, we introduce the system model assumed for all caching strategies. The three new replacement schemes as well as the methods used for velocity determination and prefetching are described in Section 3. Section 4 presents the simulation model which has been implemented to evaluate the new algorithms. Simulation results which compare our schemes with LRU and configurations without caching are analyzed in Section 5. We also discuss simulation experiments which study the performance impact of prefetching and load partitioning. Section 6 summarizes the main conclusions from this investigation.

2. PROBLEM STATEMENT AND MODEL

There are M datasets (files) in the system, which are denoted $D_1, D_2, \ldots, D_M$. The system processes a set of batch jobs denoted $J_1, J_2, \ldots, J_N$. All of the jobs are ready to execute when the batch window begins and we assume that they can be processed in any order†. A job J_i reads $M(i)$ datasets $D_1, D_2, \ldots, D_{M(i)}$. Job J_i reads every record in a dataset in order from the first to the last. The job may read all $M(i)$ datasets in parallel, one dataset at a time or any other interleaving. Our algorithms dynamically adapt to the job's usage pattern.

Since all jobs that access D_j are reading sequentially, performance can be imporved by reading *blocks* of physically adjacent records as opposed to reading one record at a time. We assume that the records are stored in their logical ordering and that there is an intrinsic *granule size* which defines the maximum number of adjacent records that can be read by a single I/O. An obvious example could be all of the records in a single cylinder. A *granule* is the unit of transfer between the disks and the main memory of the computer system.

Given this model, the problem is to minimize the time between starting the batch jobs and the completion of the last job. One way of decreasing the elapsed time is to cache granules from the datasets in main memory. There is a *cache manager* that controls a main memory dataset cache. All granule requests are processed through the cache manager. There are three possible outcomes from a request. The first is a *hit* and the job immediately begins processing the individual records/pages in the granule. The granule is "fixed" (not replaceable) until the job begins processing the next granule. The second outcome is a *miss*, and the job is suspended until the granule is read into the cache by the cache manager.

The final outcome is called an *in-transit*, which means an I/O is already in progress as a result of some other job's request. The second job is also suspended until the I/O completes. In-transit I/Os delay jobs (like misses) but decrease disk contention (like hits). One physical I/O satisfies multiple job I/O requests. This reduces disk contention and decreases the time required to service cache misses. Our experiments in Section 5 demonstrate the importance of decreased device utilization achieved by the in-transit I/Os. Previous caching studies have ignored the significance of the in-transit I/O state and have simply assumed cache hits when in-transits occur. This assumption greatly affects previous results for sequential access patterns.

To complete the system model, it is necessary to specify two algorithms. The first is the *cache replacement algorithm* that determines which cached granule is replaced when a new granule is read into the cahce and the cache is full. The second algorithm determines which granules are prefetched by asynchronous *prefetch jobs*. Due to the sequential nature of the jobs' access patterns, it is possible to predict which granule a job needs next. This granule can be read into the cache before the job submits the read request. Several algorithms for cache replacement and prefetching are described in the next section and their performance is studied in Section 5.

This model only deals with the datasets that are sequentially read by the batch jobs. Batch jobs

†In practice, there may be *precedence constraints* among the jobs. Optimal scheduling of precedence constrained jobs has been extensively studied [17]. We assume no precedence constraints to isolate the performance of our cache management algorithms from other extraneous effects on performance.

also perform random reads and writes to datasets, and sequentially write new datasets. In this paper, we are examining situations in which the sequential read processing delays due to I/Os and queuing delays on shared disks is a dominant factor in the elapsed time performance. Analysis of several batch workloads has shown this to be common for batch processing, and it may be common in other domains.

3. CACHE MANAGEMENT ALGORITHMS

This section presents our new cache replacement and prefetching algorithms. The three new cache replacement algorithms are the *Binary Use Count*, *Weighted Binary Use Count* and *Use Time* algorithms. In this section, we also present the *Deadline Prefetching* algorithm. The *Load Partitioning* strategy for scheduling is described in Section 4.

All of the cache replacement algorithms and the prefetch algorithm anticipate the set of granules that jobs will read in the near future. This set can be determined using the current positions of the active jobs in the datasets (i.e. granule ID being read), the fact that jobs read granules sequentially, and estimates of the *velocities* at which jobs are proceeding through the datasets. Our algorithm for estimating job velocities is presented in the first subsection.

3.1. Velocity estimation algorithm

The *velocity* of job J_i is defined to be the number of granule read requests that job J_i submits per unit of time. Clearly, the number of cache hits, misses and in-transits experienced affect the rate at which J_i submits requests. The velocity estimation algorithm isolates job J_i from the results of read requests and computes J_i's *attainable* velocity. The attainable velocity is the rate at which J_i would read granules if no cache misses or in-transits occur. This definition is similar in philosophy to the definition of working set in virtual memory systems [18], which isolates the reference behavior of a program from the page faults it incurs.

A job's attainable velocity is determined by many factors, such as CPU time per granule, CPU queuing delay, random I/Os, etc. These factors may not be constant over time. So, the estimates of the job's velocities are periodically recomputed. The velocity estimation algorithm is invoked every Δ seconds. Assume the algorithm is invoked at time t_0 and that J_i is reading D_j. The algorithm records the job's current position (granule) in the dataset, denoted p_i, and also sets a variable b_i to 0. The variable b_i records the amount of time J_i is blocked by cache misses or in-transits during the interval $(t_0, t_0 + \Delta)$. The cache manager can update this variable by recording the time it blocks J_i when a miss or in-transit occurs and the time it restarts J_i after the I/O completes.

The next invocation of the velocity estimation algorithm occurs at time $t_0 + \Delta$. Let c_i be job J_i's new position in D_j. The velocity of J_i is defined as

$$V_i = (c_i - p_i)/(\Delta - b_i).$$

The algorithm then sets p_i to c_i and resets b_i to 0.

Figure 1 depicts an example of the velocity estimation algorithm. Let $\Delta = 0.5\,s$. At time t_0, job J_i is at granule 10 ($p_i = 10$) and at time $t_0 + \Delta$ at granule 15 ($c_i = 15$). Assume that J_i encountered a miss when accessing granule 12 and an in-transit at granule 14 with blocked times of 70 and 30 ms,

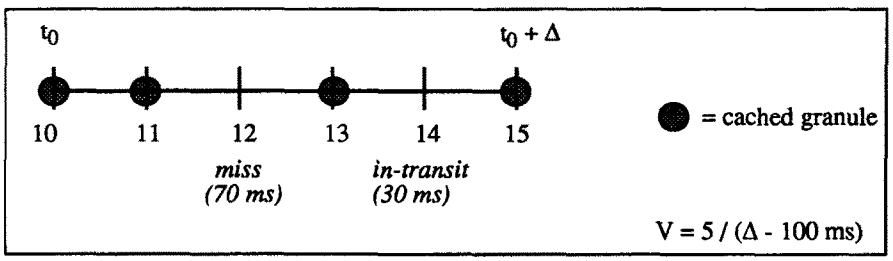

Fig. 1. Velocity estimation. The horizontal line represents a dataset. A circle indicates that the granule is cached when the job references it.

respectively. The velocity is simply

$$V_i = (15\text{–}10)/(0.5 - 0.1) = 12.5 \text{ granules/s.}$$

It is possible that a job starts reading a new dataset at some time t_1 in the interval $(t_0, t_0 + \Delta)$. The velocity estimation algorithm handles this case by setting p_i to $-n$ where n is the number of granules of the old dataset referenced by the job during the interval (t_0, t_1).

3.2. Cache replacement algorithms

The three cache replacement algorithms (Binary Use Count, Weighted Binary Use Count and Use Time) have two components. The first component is an *in-line* algorithm that is called to determine which stealable cached granule should be replaced when a new granule is to be added to a full cache. The second component is *asynchronous* with respect to job execution and granule replacement, and is triggered after the velocity estimation algorithm updates the jobs' velocities. This component is parameterized by a *look ahead time* L and determines which cached granules will be referenced in the next L s. Before presenting the replacement schemes, we first discuss how the look ahead is determined. At the end of this subsection, we estimate the overhead introduced by our algorithms.

Assume that job J_i is reading dataset D_j, its attainable velocity is V_i and its current position in D_j is p. In the next L s, J_i will at most read $L \cdot V_i$ granules. These granules are $p + 1$, $p + 2, \ldots, p + K_i$ with $K_i = \lceil L \cdot V_i \rceil$. These K_i granules are said to be in the look ahead of H_i. The look ahead is computed for all datasets that J_i is currently reading. It is possible that not all K_i granules are cached. In this case, some of the granules in the look ahead of J_i must be read from disk. Each read could be initiated by J_i, another job J_k or a prefetch job for dataset D_j. Let T be the average time required to read a granule from disk and insert it into the cache. The maximum number of granule transfers for a dataset in the next L s is $m = L/T$. The look ahead of J_i is pruned and does not include a granule k for which there are at least m non-cached granules in the set $\{p + 1, p + 2, \ldots, p + (k - 1)\}$. This pruning limits overhead by not examining granules that J_i cannot reach in the next L s.

Figure 2 presents an example illustrating the pruning of look aheads. There are two look ahead vectors associated with each job. The solid vector represents the look ahead based on the attainable velocity V_i containing K_i granules. The dashed vector represents the pruned look ahead when $T = L/2$. Job J_1 is at granule 3 and has $K_1 = 4$. Granule 7 would be reachable except for the fact that granules 4 and 6 are not cached and $m = 2$. Figure 2 will also be used to describe the cache replacement algorithms.

Binary Use Count (BUC). The BUC algorithm (as well as WBUC) associates a *use bit* with each granule in the cache. This bit is set of 0 each time the asynchrous component of the algorithm is invoked. For each job J_i, the algorithm computes the pruned look ahead in each dataset and examines granules $p + 1, p + 2, \ldots, p + k$. Each cached granule in the look ahead has its *use bit* set to 1, the use bits of other granules are set to 0. The use bits indicate which granules will be

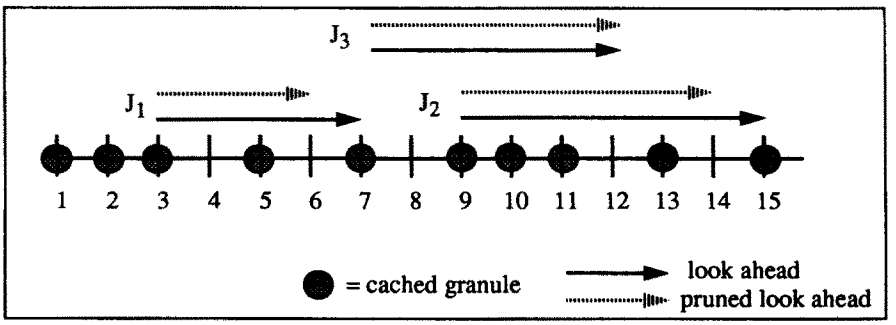

Fig. 2. Look ahead of three concurrent jobs. The horizontal line represents the dataset and circles represent cached granules.

referenced during the next L s and the in-line algorithm does not replace any granules with a use bit of 1. In the example of Fig. 2, the set of granules with use bits set to 1 is $\{5, 9, 10, 11, 13\}$.

The BUC in-line component is very simple. It randomly chooses a granule from the set of granules from all datasets what have use bits set to 0. The intuition behind this algorithm is that granules that are not in the look aheads will not be referenced for a long time, and are essentially indistinguishable.

Weighted Binary Use Count (WBUC). As for the BUC algorithm, the asynchrous component of WBUC sets the use bits of granules in the jobs' look ahead. The in-line component of WBUC is more sophisticated than for BUC. WBUC uses information obtained from a static analysis of the work load to *preselect* the dataset from which the granule is to be stolen. The following values are used to select the dataset.

—E_j: this is the total number of jobs that will read dataset D_j.
—R_j: this is the total number of jobs that have already read or are currently reading D_j.
—C_j: this is the number of granules from D_j currently in the cache.

The *weight* of dataset D_j is defined as the total number of outstanding jobs times the dataset size (denoted S_j) and is

$$W_j = (E_j - R_j) \times S_j.$$

WBUC dynamically partitions the cache into sub-caches dedicated to each dataset. For a cache of size S_C the size of D_j's allocation is proportional to its weight and is defined by

$$A_j = S_C \times \left\langle W_j \Big/ \left(\sum_{k=1}^{M} W_k \right) \right\rangle.$$

The WBUC algorithm steals the next granule from the dataset k that most exceeds its allocation (i.e. maximizes $C_k - A_k$). A dataset is allowed to exceed its allocation if some other dataset has not yet had enough requests to fill its allocation, e.g. no job has started reading it.

The idea behind the preselection is twofold. First, granules from datasets that have many expected future jobs should be replaced after granules from datasets with few expected jobs. This is a greedy heuristic for decreasing the number of cache misses. Secondly, a certain percentage of granules from all datasets should be kept in the cache, even those with few expected jobs. If this were not done, datasets with few expected jobs would not have any cached granules and the jobs reading them would proceed at disk I/O rates.

After preselecting a dataset D_j, WBUC selects the granule within the dataset to replace. Two heuristics are used. First, the granule with highest ID behind the last job is chosen. Second, if there are no granules behind the last job, the granule with highest ID and a use bit of 0 is chosen. If the WBUC algorithm selects the dataset in the example of Fig. 2, the order in which granules will be replaced is: 2, 1, 15, 7.

Use Time algorithm. The asynchronous component of the Use Time algorithm explicitly estimates the next reference time (use time) for each cached granule. Assuming that J_i's velocity is constant during the next L s and all granules in the look ahead are cached, J_i will read granule $p + j$ in j/V_i s. If, however, there are q non-cached granules in the set $\{p + 1, p + 2, \ldots, p + j - 1\}$, the estimated reference time for granule $p + j$ is

$$j/V_i + q \cdot T.$$

A cached granule may be in the look ahead of several jobs. In this case, the granule's use time is defined as the smallest reference time over all jobs.

There may be cached granules that are not in the pruned look ahead of any job. These granules fall into two classes. The first class are those granules that are behind the last active job in their dataset. In the example of Fig. 2, granules 1, 2 and 3 fall into this class. The second class contains graunles that are not behind the last job in the dataset. In the example, granules 7 and 15 are in this class.

The asynchronous component of the Use Time algorithm builds three sorted lists of cached granules from all datasets, which are the following:

—The *behind_last_job_list* containing granules behind the last job in their datasets. This list is sorted from largest to smallest granule ID. In the example, this list is {2, 1}.
—The *not_seen_list* containing granules not in the look ahead of any job and not behind the last job in their dataset. This list is also sorted from largest to smallest granule ID, and is {15, 7} in the example.
—The *use_time_list* containing granules in job look aheads. This list is sorted from largest to smallest use time. In the example, this list is {13, 5, 9, 11, 10}.

The in-line component of the Use Time algorithm steals the granule with the largest use time. This algorithm makes similar assumptions as WBUC and first replaces all granules in the behind_last_job_list followed by granules in the non_seen_list and finally granules in the use_time_list. The Use Time algorithm also dynamically updates the behind_last_job_list as the last job in a dataset advances.

There are two main differences between WBUC and Use Time. The first is that Use Time does not preselect a dataset and its lists contain granules from all datasets. The second difference is that Use Time algorithm differentiates between granules in job look aheads. This is especially important when the size of the cache is small relative to the total size of the datasets, when the jobs have high velocities or when there are many active jobs. In these situations it is likely that all cached granules are in the look ahead of a job and have non-zero use bits. BUC and WBUC cannot make an intelligent replacement decision in this situation. For this reason, a longer look ahead time L can be used with the Use Time algorithm.

Overhead considerations. The overhead of the in-line algorithms is comparable to the overhead of LRU. The Use Time algorithm simply replaces the granule at the head of one of its three lists. This is done in time $O(1)$ as for LRU. BUC randomly chooses a granule that has a 0 use bit. This may be implemented by maintaining a list of stealable granules from which the head is chosen for replacement by the in-line algorithm. While this policy is not truly random, it is sufficient since granules with 0 use bit are considered equally stealable. The complexity of BUC's in-line component is thus reduced to $O(1)$. WBUC selects the replacement victim from a dataset-specific list which can be done in time $O(1)$. Additional overhead is introduced to implement the dataset selection policy. Maintaining the variables C_j and R_j as well as adapting the weights W_j and allocations A_j incurs little overhead. The worst-case complexity of WBUC's in-line component is $O(A)$, where A is the number of concurrently active jobs.

The overhead of the asynchronous components of the BUC and WBUC algorithms is linear in the number of active jobs A and the cache size S_C. The complexity of examining the look aheads and setting use bits is $O(A \cdot S_C)$. The Use Time algorithm incurs a similar overhead for determining the use times in the look ahead of active jobs. However, it also incurs the complexity of building the three sorted lists which is $O(S_C \cdot \log S_C)$.

The overhead of the asynchronous components can be controlled by setting the parameters Δ and L. The parameter Δ determines how often the asynchronous component is invoked and L controls the complexity of processing the look aheads. Furthermore, the asynchronous components do not directly delay the processing of granule requests that jobs submit. They can be run as a background job so that their overhead only indirectly affects the performance of the batch jobs through CPU contention. If CPU utilization is high, Δ can be increased and L decreased to lower overhead. In time of high CPU utilization, optimal cache management is less imperative. For most cache sizes, the system is I/O bound and there is ample spare CPU capacity for the asynchronous components.

3.3. Deadline prefetch algorithm

The BUC, WBUC and Use Time algorithms use job look aheads to determine which cached granules will be referenced in the next L s. The examination of the look aheads also reveals which *non-cached* granules are expected to be referenced. For example, in Fig. 2, job J_1 is expected to read non-cached granule 4. Granules that are not cached and are in the look ahead of a job are called *prefetch candidates*. The determination of the prefetch candidates and the prefetch order is part of the algorithms' asynchronous component.

A control block is created for each prefetch candidate and this control block contains the

estimated use time of the granule. The use time is defined exactly as for cached granules and represents the deadline by which this granule must be cached to avoid a miss. Computing the deadlines of prefetch candidates is done for all cache replacement algorithms (including LRU), if prefetching is active.

After all look aheads have been processed, a prefetch list is built for each dataset. The list for D_j contains all prefetch candidates from this dataset and is sorted from smallest to largest deadline. In the example of Fig. 2, the prefetch list for the dataset is: 8, 4, 12, 6, 14.

A prefetch job for dataset D_j simply loops through the following steps:

(1) Remove granule p from the head of the prefetch list for D_j. The list will not contain prefetch candidates read into the cache by other jobs.
(2) Read granule p into the cache. The prefetch job is blocked until the granule is cached.
(3) Go to 1.

4. SIMULATION MODEL

We implemented a detailed simulation system for evaluating the performance of the newly developed caching strategies. The overall structure of the simulation system is shown in Fig. 3. The main parameters of our simulation system are summarized in Table 1 together with their settings used in the experiments. The four major components of the simulation system are CPU management, I/O management, job management and cache management/prefetching, and will now separately be discussed. The parameters in Table 1 are also grouped according to this distinction.

4.1. CPU management

We assume that the workload is executed on a tightly coupled multiprocessor with the number of CPUs provided as a simulation parameter. Each active batch job is executed by a job process. The execution cost of a job is modelled by requesting CPU service for every granule access. CPU service is also requested for every disk I/O to account for the I/O overhead. The number of instructions per CPU request is exponentially distributed over a mean specified as a parameter.

4.2. I/O management

It is assumed that every dataset resides on a separate disk. Each disk is modeled as a single server queue in order to capture I/O queuing delays. The disk I/O time per granule is the sum of disk latency and transfer time. With no queuing delays, the parameter settings from Table 1 result in an average I/O time of 45 ms per granule.

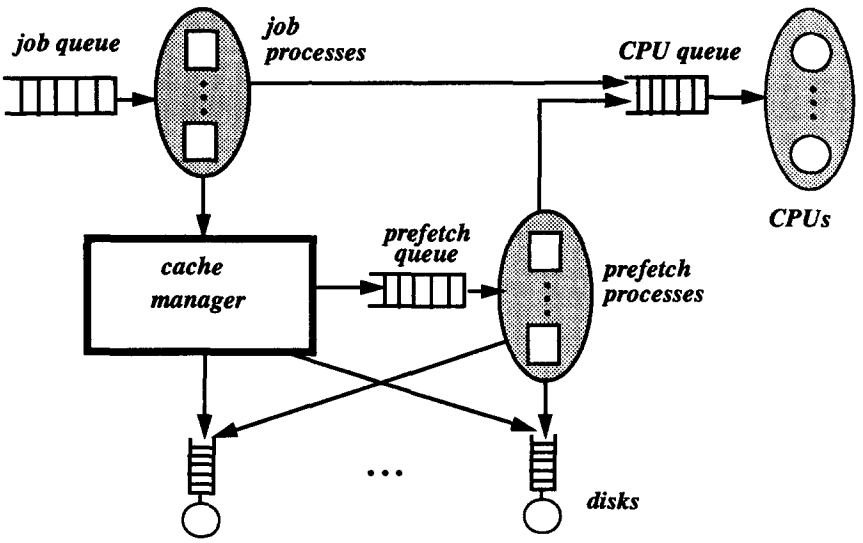

Fig. 3. Structure of the simulation model.

Table 1. Simulation parameters

Parameter	Parameter settings
#CPUs	6
MIPS per CPU	20
Instructions per granule access	50,000
#Instructions per I/O	5000
#Datasets	40
Dataset size (# granules)	1000
Granule size (# blocks)	25
Block size	4 KB
Average I/O latency	20 ms
I/O transfer rate	4 MB/s
#Jobs to be executed	1000
READMAX (see text)	3
Multiprogramming level	60
#Partitions	1–10
Caching	yes/no
Cach size	100 MB–2 GB
Replacement algorithm	LRU, BUC, WBUC, Use Time
Prefetching	yes/no
Buffer threshold	5% of cache size
Wake-up interval Δ	2 s
Look-ahead window L	20 s

4.3. Job management

In every simulation run, we simulate the processing of a fixed number of batch jobs. We assume that all batch jobs are ready to execute at the beginning of a run, and wait in a job queue until they are scheduled for execution. Like in real systems, we allow only a fixed number of batch jobs to be processed concurrently; this number is referred to as the *multiprogramming level (MPL)*. When an active job completes, another job is selected from the job queue. The MPL is a configuration-dependent parameter and should be chosen such that resources like main memory and CPUs can be well utilized without causing severe overload situations.

The concentration on batch jobs and sequential access allowed us to use a fairly simple, synthetic workload model. Every job reads a certain number of datasets sequentially. This number is a uniformly distributed random variable per job in the range 1 to READMAX. The datasets read by a job are also randomly selected with uniform probability thus representing a worst case scenario for caching (small locality of reference between different jobs). Update accesses are currently not supported by the simulation system although the replacement schemes can deal with read as well as write accesses†. With the parameter settings from Table 1, a job reads 2 datasets on average (2000 granule accesses) resulting in an average of 100 million instructions per job (without I/O overhead).

Locality of reference between jobs can be improved by a *job scheduling* strategy which tries to concurrently activate jobs accessing the same datasets. Such a strategy is feasible since the datasets a batch job needs to access are generally known in advance. In the simulation model, we implemented a simple (static) scheduling strategy called *Load Partitioning* that rearranges the order of jobs in the job queue befor batch processing starts. This policy is controlled by the parameter "# partitions" which must be smaller than the number of datasets. We use this parameter to group the datasets into the specified number of partitions such that every partition is referenced by about the same number of jobs. After this we use the dataset partitions to build load partitions where a job is assigned to load partition x when most of its granule accesses fall into dataset partition x. The scheduling order is then determined by these partitions, i.e. we first start all jobs belonging to load partition 1, followed by jobs of partition 2 and so on. Within a load partition, the scheduling order of the jobs is random. Note that there is generally more than one dataset partition in use since a job may reference more than one partition and because jobs of multiple load partitions may be active at the same time (e.g. in transition phases between load partitions or if the MPL is higher than the number of jobs per partition).

†The replacement of a modified granule in the cache has to be delayed until the modification is written back to disk. Response time deterioration due to such write delays can largely be avoided, however, by asynchronously writing out modified granules.

4.4. Cache management and prefetching

We have implemented a global LRU steal policy as well as the three new replacement algorithms described in Section 3 (BUC, WBUC and the Use Time algorithms). Caching can also be switched off; in this case every granule access causes a disk I/O. As pointed out in Section 3, some control functions are asynchronously executed by the cache manager. The execution of these functions is triggered either by a timer (wakeup interval Δ) or when the number of replaceable cache frames falls below a specified threshold (parameter "buffer threshold"). If prefetching is chosen (parameter), a prefetch queue is used to specify the granules to be read by the prefetch processes. In our model, there is one prefetch job per dataset (disk) to avoid disk contention among prefetch jobs.

5. SIMULATION RESULTS

In this section, we present the results of simulation experiments that measure and explain the performance of our new cache management algorithms. All simulation experiments were conducted with the parameter settings given in Table 1. We compare our algorithms to the traditional LRU algorithm and configurations with no caching. The impact of different cache sizes, the effect of prefetching (Section 5.2) and the influence of job scheduling strategies that try to improve inter-job locality (Section 5.3) are considered.

5.1. Cache replacement algorithms

Figure 4 shows the elapsed time to process the batch jobs versus cache size for the four replacement algorithms (LRU, BUV, WBUC, Use Time), without prefetching or load partitioning (# partitions = 1). The elaspsed time obtained without caching is also shown for comparison purposes. The x-axis in this figure represents the *relative* cache size, as a percentage of the total dataset size. In our experiments, the total dataset size is 4 GB and we vary cache size from 100 MB (2.5%) to 2 GB (50%).

The graph shows that all replacement schemes result in shorter elapsed time than with no caching, even for small cache sizes. Our replacement schemes outperform LRU for all cache sizes. The Use Time algorithm is clearly superior to the other schemes in particular for smaller cache sizes thus supporting high cost-effectiveness. For a relative cache size of 2.5% it cuts elapsed time already by 25% and almost by 30% for a cache size of 5% (compared to a maximal improvement of 10% for the other schemes). While the BUC scheme is only slightly better than LRU, the WBUC scheme gains most from increasing cache sizes and approaches the Use Time algorithm for very large caches (2 GB). LRU could not utilize the 2 GB cache; the elapsed time for this case is almost the same as for a 1.2 GB cache (30%) and substantially worse than with the other schemes.

The elapsed time results are mainly determined by the algorithms' I/O behavior. Without caching the average I/O delay per granule was about 62 ms including 17 ms queuing delays due to disk

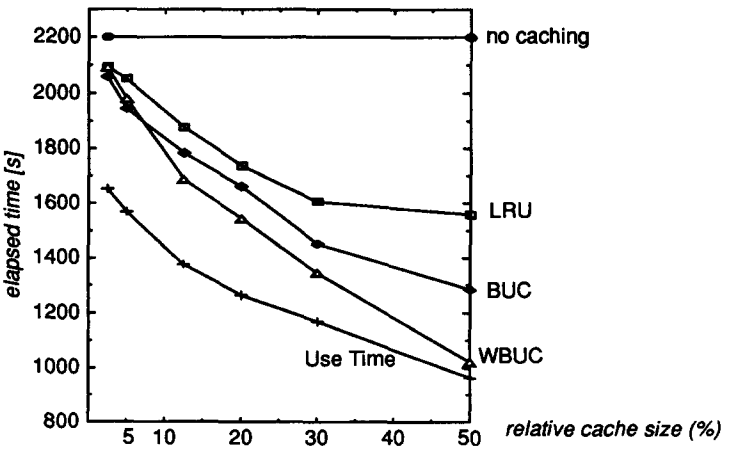

Fig. 4. Elapsed time.

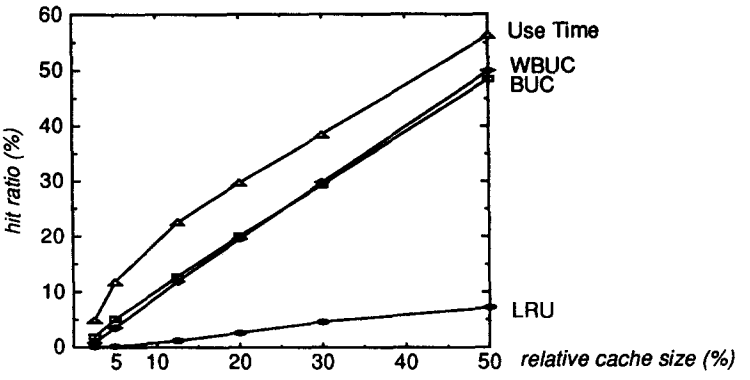

Fig. 5. Hit ratios.

contention. Caching reduced the I/O delay and therefore elapsed time in several ways. *Cache hits* reduce the total number of I/Os and thus the I/O delay. Fewer I/Os also mean reduced disk contention and thus shorter disk access times when misses occur. In addition, CPU overhead for disk I/O is reduced and thus CPU queuing delays are decreased. *In-transit granules* also contribute to an improved I/O performance. In this case, another job has already initiated the disk read for the granule and the requesting job is only delayed until the I/O completes. Although in-transit granules cause an I/O delay, this delay is on average shorter than a full disk access. In addition, the I/O overhead associated with a disk access is saved and disk contention is decreased thereby reducing the I/O delay for cache misses.

The effectiveness of these factors varies substantially for our replacement schemes thus giving rise to the differences in elapsed time. This is clarified by Fig. 5 showing the hit ratios for the four replacement schemes. We observe that the Use Time scheme consistently shows the best hit ratios. However, hit ratios are comparatively low even for this scheme when compared to the relative cache size (e.g. 56% hit ratio for a relative cache size of 50%). This is because accesses are uniformly distributed over all datasets resulting in a comparatively low locality of reference between jobs if no load partitioning is applied. LRU achieved very low hit ratios of less than 10% for all cache sizes, although a high number of in-transit I/Os (see below) helped to improve performance compared to the case with no caching. BUC and WBUC have almost identical hit ratios which increase linearly with the cache size. WBUC outperforms BUC because it required fewer I/Os, i.e. it had in-transit I/Os in many cases where BUC experienced a buffer miss.

With LRU the system remained I/O bound even for very large caches indicating that increased CPU power could not be used to reduce elapsed time. In contrast, our schemes could significantly reduce I/O delays with increasing cache size and were CPU bound for large cache sizes. Thus an additional advantage of our new algorithms compared to LRU is that they support vertical growth, i.e. elapsed time can be reduced by providing faster CPUs and larger caches.

The relationship between hits, misses and in-transit I/Os becomes clearer with Fig. 6 which shows their frequency for LRU and the Use Time algorithm. The number of hits, misses and in-transit I/Os add up to the total number of granule accesses by all jobs (2,000,000 = 100%). Figure 6 shows that a substantial amount of granule accesses are delayed because of in-transit I/Os (up to 40% for LRU) illustrating the importance of considering this case. If we had assumed a hit for every in-transit granule, as previous studies have usually done, results would be totally different and even LRU would have performed much better.

In-transit I/Os show that there is locality of reference between different jobs since more than one job wants to access the same granule concurrently. For the Use Time algorithm, the number of in-transit I/Os is proportional to the number of misses and decreases with larger caches. This scheme can use locality of reference to substantially increase the hit ratios for growing cache sizes. With LRU, on the other hand, hit ratios remain very low even for large cache sizes although the miss ratio can be reduced significantly. Larger caches improve performance for LRU mainly because more and more buffer misses are replaced by in-transit I/Os. This trend, however, flattens

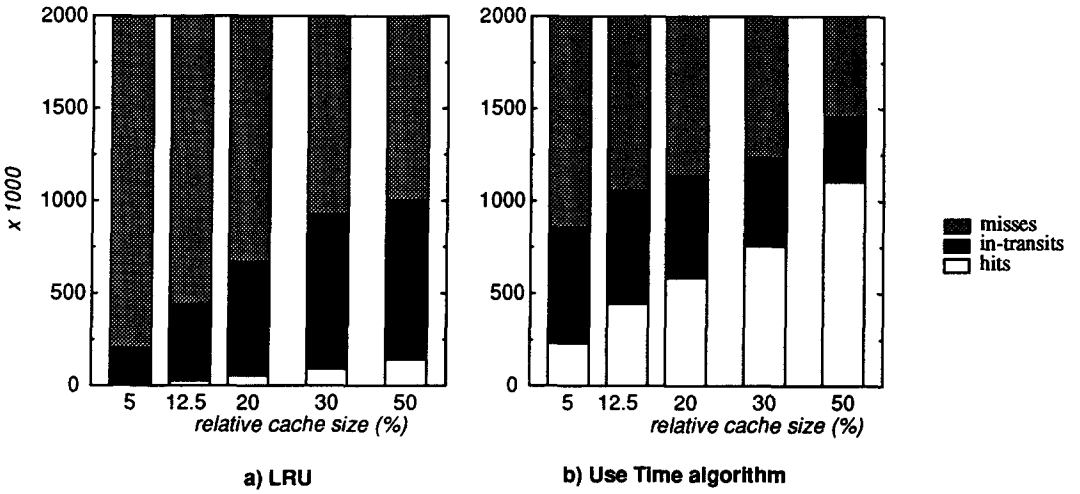

Fig. 6. Number of hits, misses and in-transits.

out for a relative cache size of more than 30% accounting for the fact that the elapsed time for 50% relative cache size is almost the same than with 30%.

In contrast to WBUC and the Use Time algorithm which try to steel directly behind the last job in a dataset, LRU steels the least recently referenced granule. Thus LRU frequently steals from the beginning of a dataset. As a consequence, in most cases newly arriving jobs cannot make use of already cached portions of the dataset, but have to read the entire dataset again thus explaining the very low hit ratios for LRU. Due to the stealing from the beginning of a dataset, this effect can only be avoided when (almost) the entire dataset can be kept in the cache. However, when the first portion of a dataset is still cached when a new job wants to access the dataset, this job can quickly catch up with the job further ahead in the dataset that has read in the granules still residing in the cache. From then on, the two (or more) jobs proceed together and very slowly through the dataset (one job does the I/O while the jobs clustered together with it find in-transit I/Os with delays close to the total access time). The new replacement strategies look ahead in the dataset to prevent the replacement of cached granules that will be referenced in the near future. This gives rise to much better hit ratios, which increase greatly with the cache size, and to a higher variance in job velocities so that they proceed less clustered than with LRU.

A limitation of the BUC scheme is that it randomly selects replacement victims from cached granules which are not in the look ahead of currently active jobs. The WBUC allows for a performance improvement by using several heuristics for finding a replacement candidate. In particular, WBUC uses an additional information to steal from datasets with the least number of outstanding jobs, and it tries to steal behing the last job in this dataset. The simulation results show that this approach brings a significant advantage mainly for large cache sizes when datasets with many outstanding jobs can completely be kept in the buffer, or when large percentages of the beginning of datasets can be kept in the cache to improve processing times of jobs when they start reading a dataset.

The improvement of the Use Time algorithm over WBUC comes from several factors which turn out to be more effective than using information about the number of outstanding jobs. One factor is that the Use Time scheme does not apply a preselection of a dataset from which to steal, but considers granules from all datasets for replacement. In this way, it can steal behind the last job as long as this is possible for any dataset. WBUC, in contrast, may preselect a dataset where it cannot steal behind the last job anymore, although this would be possible for another dataset. Another advantage of the Use Time scheme is that is uses the information on next time to reference for stealing granules in the current look ahead of jobs. The use of this information makes is superior to WBUC particularly for small caches (or large total dataset size) or high MPL, where it is usually the case that there are no granules behind the last job in a dataset and all granules have their use bit set. WBUC cannot perform much better than random replacement in this scenario since all cached granules appear identical.

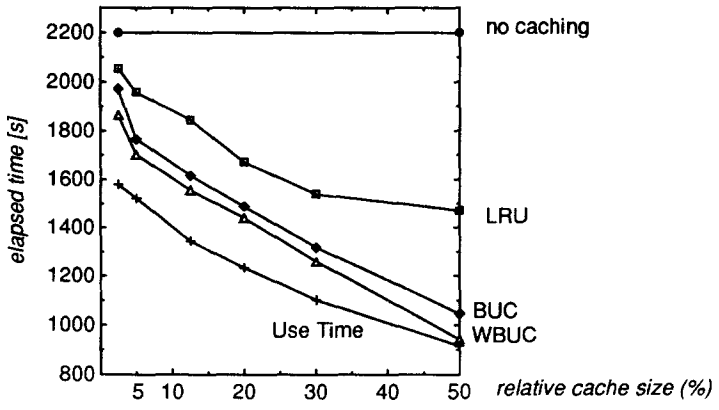

Fig. 7. Elapsed times with prefetching.

We conducted a series of experiments to find out the influence of different look aheads. It turns out that the BUC scheme depends mostly on a large look ahead since otherwise only few of the cached granules are considered useful. This, however, increases the probability of selecting unfavorable replacement victims, since BUC picks them randomly from the granules not considered useful. We observed performance degradations of up to 10% for a look ahead of 2 s compared to the results with 20 s. The WBUC and the Use Time algorithms are very insensitive to different look aheads as long as they are at least as large than the wake-up interval Δ.

5.2. Influence of prefetching

In this section we study the effects of asynchronous prefetching performed by dedicated prefetch processes. As outlined in the previous sections, we have one prefetch process per disk that uses deadline information based on velocity estimates to determine the prefetch order of granules. Figure 7 shows the elapsed times obtained with such a prefetching (and with no load partitioning). We observe very similar performance trends compared to the no prefetching experiments (Fig. 4), and the Use Time algorithm remains the best and LRU the worst replacement scheme. A notable change is that the BUC algorithm benefits the most from prefetching and becomes substantially better than LRU. The BUC scheme has the fewest in-transit I/Os without prefetching. Prefetching causes the highest increase of in-transit I/Os for this scheme leading to substantial performance improvements. For the other schemes, prefetching resulted in comparatively small improvements in elapsed time (less than 10%) for all cache sizes.

To explain the performance impact of prefetching, we analyze the frequencies of buffer hits, misses, in-transit and prefetch I/Os for the LRU and Use Time schemes (Fig. 8). As in Fig. 6 for

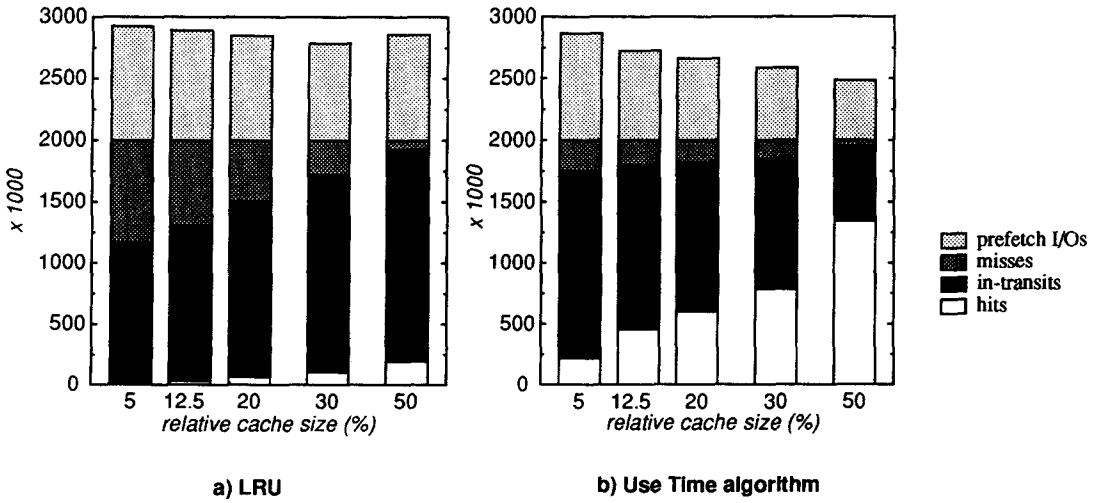

Fig. 8. Number of hits, misses, in-transits and prefetch I/Os.

no prefetching, the number of hits, misses and in-transit I/Os add up to the total number of granule accesses by all jobs (2,000,000). In addition, we have shown the number of prefetch I/Os on top of the buffer misses in Fig. 8. Prefetch I/Os and misses combined represent the total number of I/Os.

We observe that for LRU the hit ratios as well as the total number of I/Os are virtually the same with prefetching as without prefetching. The hit ratio for the Use Time algorithm is somewhat higher with prefetching, especially for large caches. For both algorithms, what changes significantly is the number of buffer misses and the number of in-transit I/Os. Basically, the number of misses is reduced by the number of prefetch I/Os and the number of in-transit I/Os is increased by the number of prefetch I/Os. With prefetching, "I/O in transit" is the dominant case for granule accesses in all schemes (for LRU up to 90% of all granule acceses!).

Though prefetching does not improve the hit ratios or number of I/Os, it saves the CPU time and overhead for initiating the I/O. Furthermore, the average I/O delay for an in-transit I/O is shorter than for a buffer miss (full disk access time). However, savings in I/O delays and thus in elapsed time are comparatively small since the prefetch jobs proceed slowly through a dataset (at I/O speed) so that the jobs are mostly close behind them. As a result, in-transit I/O delays were almost as high as the disk access times.

For prefetching to be more effective, the mean time between a job submitting granule access requests would have to be very high so that more benefit can be derived from overlapping I/O and other processing. In this case, however, not I/O but other resources would be the bottleneck which seems not typical for current batch processing applications. Another possibility is to restructure the storage of datasets on the disks to allow multiple concurrent I/Os per dataset (e.g. store the dataset on multiple disks, disk striping). This would allow prefetch I/Os and normal data access I/Os to overlap and may increase the benefits of prefetching.

5.3. Influence of Load Partitioning

Load Partitioning is a job scheduling strategy that uses information about the job reference behavior to concurrently schedule jobs accessing the same datasets. As outlined in Section 4, we use the parameter " # partitions" to control this kind of scheduling. The primary goal of load partitioning is to improve locality of reference between jobs and thus the hit ratios and elapsed time. A possible problem is increased disk contention since more jobs are now concurrently accessing the same datasets (disks) than without load partitioning.

Figure 9 shows the elapsed time results obtained for a load partitioning with 4 partitions and prefetching. Four partitions means that usually only one fourth of the datasets (disks) is accessed by active jobs. We observe that load partitioning leads to dramatic changes in elapsed time for the different cache sizes. With small cache sizes, all caching schemes but the Use Time algorithm have higher elapsed time than without caching! (We did not apply load partitioning without caching because it would obviously be a bad idea.) The Use Time algorithm's performance is slightly better with load partitioning for small cache sizes than without load partitioning. Increasing the cache

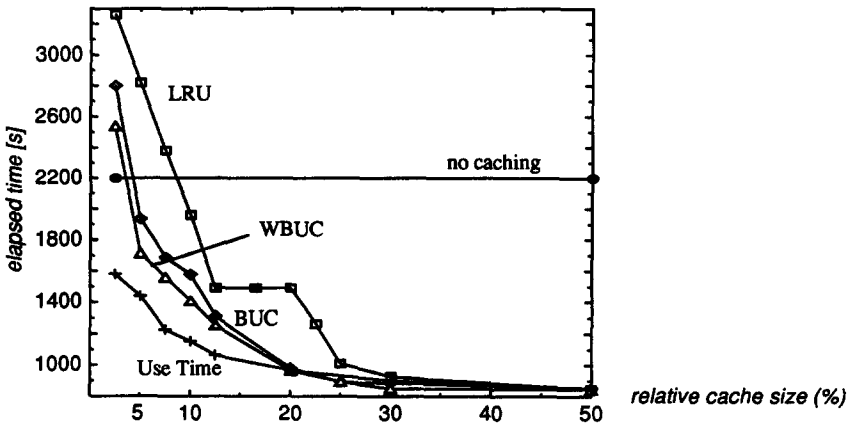

Fig. 9. Elapsed time for load partitioning (4 partitions, prefetching).

size leads to steep performance improvements until all schemes converge to an elapsed time close to the best possible. For a relative cache size of more than 25% all schemes achieve about the same elapsed time since almost all concurrently accessed datasets can be completely kept in main memory in this range.

For small cache sizes, however, the "working set" (granules with small use times) of active jobs is too large to be kept in the cache and only a few hits can be achieved. Since the high number of I/Os is directed only to a subset of the disk devices, disk contention increases drastically compared to the case without load partitioning. This increases total I/O delay for the jobs which in turn increases elapsed time. Increasing the cache size quickly reduces the number of I/Os because the higher locality of reference due to load partitioning improved hit ratios and/or the amount of in-transit I/O and therefore disk contention. As a result, the I/O delay per granule access and elapsed time improve greatly.

The Use Time algorithm shows a much stabler behavior than the other algorithms. In particular its effectiveness for small cache sizes is extremely valuable with load partitioning and prevents the thrashing effects observed for the other replacement schemes. Therefore, even with very small caches (2.5%) significantly better elapsed time than without caching and dramatically better results than with the other caching schemes are achieved.

LRU exhibits an especially unstable performance. This is also illustrated by Fig. 10 depicting the frequency of hits, misses, in-transit and prefetch I/Os for LRU. Figure 10 as well as Fig. 9 show that LRU experiences three phases with different performance characteristics. The first phase is associated with small cache sizes ($\leqslant 12.5\%$) and is characterized by a high number of I/Os (mostly regular I/Os) and high disk contention. Hit ratios are very low in this phase. The third phase is when almost all active datasets can be kept in the cache (relative cache size $\geqslant 25\%$). In this situation even LRU experiences very high hit ratios and few I/Os; elapsed times are therefore as good as with any of the other schemes. The second (perhaps most interesting) phase is observed for relative cache sizes between 12.5 and 25%. In this phase the cache size is high enough to reduce the number of buffer misses to a large extent, however withoug significantly improving the hit ratios. In the corresponding cache size range, elapsed time remains almost the same even when increasing the cache size. This corresponds to the observation made without load partitioning (Section 5.1), where LRU could not further improve performance after a relative cache size of 30%, and hit ratios were very low even for relative cache size of 50%. The underlying reason is that after a certain cache size, LRU cannot utilize larger caches until the cache is large enough to hold almost all datasets (with load partitioning all currently active datasets).

Figure 11 compares elapsed time results for LRU and the Use Time scheme varying the number of partitions from 1 (no load partitioning) to 10. The cache size for these experiments has been 500 MB (12.5% of the total dataset size). For this cache size, we see comparatively small changes in elapsed time for the different number of partitions. In all configurations, LRU is clearly outperformed by the Use Time algorithm. (The Use Time scheme achieved hit ratios of up to 66% compared to 6% for LRU.) With LRU, elapsed time deteriorates for 2 partitions compared to the

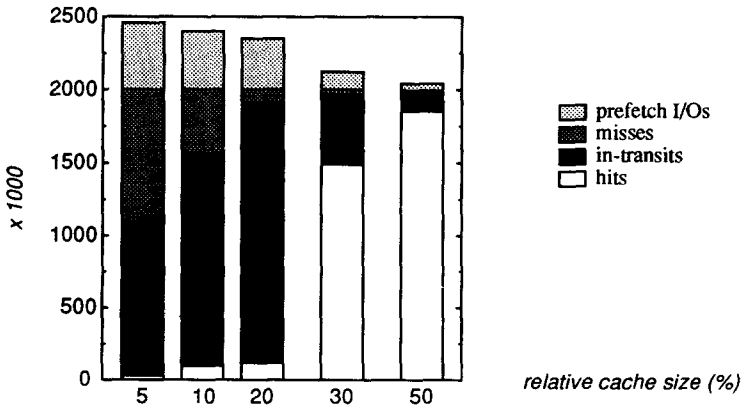

Fig. 10. Number of hits in-transits, misses and prefetch I/Os for LRU (4 partitions).

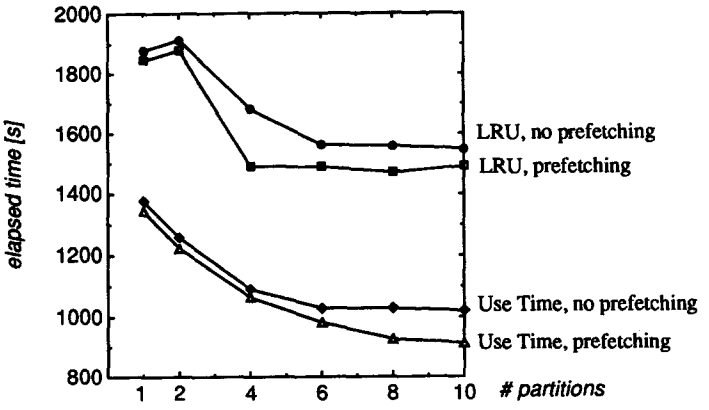

Fig. 11. Elapsed time vs # partitions (cache size 500 MB).

case without load partitioning due to increased disk contention, but improves significantly with 4 partitions. More than 4 partitions do not result in significant performance improvements for both replacement strategies. This was because locality could not significantly be increased any more since with a larger number of partitions the population per load partition becomes smaller, making is more likely that more than one load partition is concurrently in use (fixed multiprogramming level). A dynamic load scheduling scheme, not employed in the simulation, could have been more successful in improving locality further, although at the expense of increased complexity.

6. SUMMARY

In this paper, we have presented a set of new cache management algorithms for shared sequentially accessed data. Sequential access is common in many important problem domains. We have focused on overnight batch processing as an example of such an environment. Most of our algorithms apply to other problem domains in which sequential data access is performed.

Our cache replacement algorithms have an asynchronous component that is periodically executed. It estimates the velocities with which jobs proceed through datasets to determine which data objects will be referenced in the future. This information is used by the replacement schemes to avoid replacement of data that will be re-referenced soon. The velocity estimates are also used by our Deadline Prefetch algorithm to perform an asynchronous prefetching. Finally, we have presented a new job scheduling strategy called Load Partitioning to improve inter-job locality.

We outlined the results of simulation experiments to measure the performance of the new algorithms. Our experiments show that the Use Time and WBUC algorithms significantly outperform traditional cache replacement algorithms. For small caches ($\leqslant$ 5% of total database size), the Use Time algorithm is by far the best replacement algorithm studied and decreases elapsed processing time by 30% compared to all other algorithms. The Deadline Prefetch algorithm achieved approximately a 10% decrease in elapsed time for most cache replacement algorithms (including LRU) in our experiments. Load partitioning was shown to be very effective for a small number of load partitions. However, for small cache sizes it can increase disk contention thus lowering performance. Only the Use Time algorithm was able to avoid such an unstable behavior and take advantage of load partitioning for all cache sizes.

The Use Time algorithm has been implemented in the new version of the MVS/ESA Hiperbatch (High Performance Batch) product of IBM which is successfully being used at more than 300 commercial data centers worldwide [1, 2]. Hiperbatch allows caching to take place in Expanded Storage (page-addressable extended memory) rather than in main memory. This difference has littly impact on the replacement algorithms, but permits a more cost-effective caching since the storage cost of extended memory is lower than for main memory [19].

There are several possible areas for further work on the problems studied in this paper. These include enhancing the algorithms to deal with update and non-sequential I/O activity. Integrating

precedence based scheduling algorithms with our load partitioning algorithm is another possible avenue for further work.

REFERENCES

[1] C. K. Thayne. Batch processing pressures eased with Hiperbatch. *Mainframe J.* **5**(5), 52–55 (1990).
[2] C. Edwards and B. Rad. MVS/ESA's Hiperbatch. *Enterprise Syst. J.* **6**(9), 22–30 (1991).
[3] M. Joseph. An analysis of paging and program behavior. *Comput. J.* **13**(1), 48–54 (1970).
[4] J. L. Baer and G. R. Sager. Dynamic improvement of locality in virtual memory systems. *IEEE Trans. Software Engng* **2**(1), 54–62 (1976).
[5] A. J. Smith. Sequential program prefetching in memory hierarchies. *IEEE Comput.* **11**(12), 7–21 (1978).
[6] K. S. Trivedi. An analysis of prepaging. *Computing* **22,** 191–210 (1979).
[7] E. J. Lau. Improving page prefetching with prior knowledge. *Performance Eval.* **2,** 195–206 (1982).
[8] M. Martinez. Program behavior prediction and prefetching. *Acta inf.* **17,** 101–120 (1982).
[9] J. Rodriguez-Rosell. Empirical data reference behavior in data base systems. *IEEE Comput.* **9**(11), 9–13 (1976).
[10] A. J. Smith. Sequentiality and prefetching in database systems. ACM Trans. Database Syst. **3**(3), 223–247 (1978).
[11] H. Wedekind and G. Zoerntlein. Perfetching in realtime database applications. In *Proc. ACM SIGMOD Conf.*, pp. 215–225 (1986).
[12] E. B. Fernandez, T. Lang and C. Wood. Effect of replacement algorithms on a paged buffer database system. *IBM J. Res. Dev.* **22**(2), 185–196 (1978).
[13] W. Effelsberg and T. Härder. Principles of database buffer management. *ACM Trans. Database Syst.* **9**(4), 560–595 (1984).
[14] H. T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational databases. In *Proc. 11th Int. Conf. on Very Large Data Bases*, pp. 127–141 (1985).
[15] G. M. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Trans. Database Syst.* **11**(4), 473–498 (1986).
[16] M. E. C. Hull, F. F. Cai and D. A. Bell. Buffer management algorithms for relational database management systems. *Inf. Software Technol.* **30**(2), 66–80 (1988).
[17] E. G. Coffman (Ed.). *Computer and Jop Scheduling Theory*. Wiley, New York (1976).
[18] P. J. Denning. Working sets: past and present. *IEEE Trans. Software Engng* **6**(1) (1980).
[19] E. Rahm. Performance evaluation of extended storage architectures for transaction processing. In *Proc. ACM SIGMOD Conf.*, pp. 308–317 (1992).