The Cache Assignment Problem and its Application to Database Buffer Management

Hanoch Levy
Department of Computer Science
Tel-Aviv University
Tel-Aviv, Israel
hanoch@math.tau.ac.il *

Ted G. Messinger
Almaden Research Center
I.B.M. Research Division
San Jose, CA 95120
mess@almaden.ibm.com

Robert J.T. Morris
IBM Watson Research Center
Yorktown Heights, NY
rjtm@watson.ibm.com

November 5, 1996

Abstract

Given N request streams and $L \leq N$ LRU caches, the *cache assignment problem* asks to which cache each stream should be assigned in order to minimize the overall miss rate. An efficient solution to this problem is provided, based on characterizing each stream using the *stack reference model* and characterizing the interaction of the streams using a *bursty stream model*. It is shown that for Bernoulli (purely random) mixing of streams, the optimal cache assignment is to have one cache per stream. But in practice streams are mixed in a way that is much "burstier" than can be represented by the Bernoulli model. Therefore a method is presented for superposition of bursty streams. The performance of the methods developed for bursty stream superposition and cache assignment are tested using trace data obtained from the database system DB2. The resulting cache assignment recommendations are then applied to the DB2 system, and considerable performance improvement is found to result.

1 Introduction

Caches continue to be one of the most pervasive structures found in computer systems. They are used at every layer of the memory hierarchy: several levels are usually present in the processor; primary memory management is traditionally based on a cache model; and, buffer pools in a database management system are managed using cache principles. While many variations have been developed over the years, the predominant principle in the management policies for these caches is a *least recently used replacement* rule, typically applied uniformly across all elements in the cache. Sometimes, data elements of

*This work was written while the author was with RUTCOR and the School of Business, Rutgers University, New Brunswick, NJ.

separate types are kept segregated in separate caches, e.g., separate caches in a processor for instruction and data words, or separate buffer pools in a database management system for index and data (or complex query and transaction) pages. The general question as to when it is better to segregate caches, and when it is better to combine them, has had no simple solution.

We will discuss a more general problem. A stream is a time sequence of requests for elements or pages. Given a set of streams of requests and a set of caches, we ask which streams should be assigned to which caches to maximize performance. Several intuitive arguments help us to understand the tradeoffs. Suppose there are two streams X and Y. It is plausibly advantageous to share a system resource such as cache memory as much as possible, an argument that would favor the use of the single shared cache. Certainly, if X and Y rarely require use of the cache at the same time, then we would like to reuse the cache resource for Y when X is not using it. But supposing the X stream has much more locality of reference than Y, then it would seem preferable to reserve some cache exclusively for the use of X, free from the "polluting" requests of Y which generates much fewer hits. Or, it may turn out that X and Y might coexist quite well in a shared cache, but neither may coexist with two other streams A and B, which may in turn coexist quite well together sharing a second cache. This paper will provide a general method for analyzing and optimizing the assignment of arbitrary streams to caches, and the appropriate sizing of those caches. We will apply it to a problem of current interest in database buffer management in which we are required to assign the table and index pagespaces of the database to a number of buffer pools.

The solution to this problem requires several components. First, we must decide how to characterize request streams, using a method more compact than a complete trace. We will do this using the *stack reference model* or SRM. This model is one of several simple models for reference streams and it has been quite successful in capturing temporal locality of references within a trace. Second, we need to capture the interaction between streams, and we will do this by examining a matrix which measures, for the combined stream of X and Y, the mean burst length of X before it is interrupted by Y, for all pairs X and Y. Third, we must compute the performance of a cache when it is shared by the accesses of both streams X and Y. The latter problem, sometimes referred to as the problem of superposition of streams into a cache, has been addressed by a number of approximation approaches [4] [14] [11]. Recently the exact solution has been obtained under the assumption of Bernoulli (purely random) mixing of the streams [8]. However, as illustrated above, it is precisely the interaction of streams and the non-random mixing, which can also be viewed as "burst characteristics" of individual streams, that will govern the decision whether to share caches. Thus we must solve a more difficult problem, involving *non-Bernoulli* mixing of streams. A fast approximate solution to this problem will be described in this paper. Fourth, with this superposition procedure in hand, we can then provide an optimization procedure which starts with the characterization of the streams and their interaction and provides a good assignment of streams to caches and an optimum sizing of the individual caches under these assignments.

After developing and and testing the effectiveness of these four components of the solution to our problem, we will then illustrate the application of the technique to the database buffer management problem. Our experiments demonstrate the benefit of the technique by a significant reduction of the physical I/O rate of a large database system running with the recommended buffer pool assignment.

There has been quite a bit of previous work on database access pattern characterization and cache or buffer hit prediction. Much of this work builds on the original virtual memory modeling work which was already in textbook form over 20 years ago [3]. There are two broad categories of models for access streams. One category is called the *independent reference model* (IRM) and assumes independent accesses to different parts of a memory (or database). The other category is called the stack reference model (SRM) and assumes access to data according to the "stack depth" at which the previous reference is found. Whereas IRM focuses on modeling spatial locality of reference, SRM focuses on modeling temporal locality of access. Each model has its own strengths and weaknesses, for example simple IRM can provide effective modeling of a hot spot but cannot easily model cyclic access to data (it requires a concave hit ratio curve) - SRM easily models cyclic access (represents any hit ratio curve) but cannot directly model hot spots. The IRM model has been applied to database stream characterization and hit prediction in papers such as [4], with separate accommodation for cyclic behavior. In our work, we are focussed on general purpose methodologies to carry out optimal assignment and allocation of database buffers. This requires us to carry out a number of calculations that go beyond buffer hit prediction, and in doing that have found the SRM model much easier to work with. In fact, as we explain above, temporal locality of reference is a major factor in determining whether or not caches should be combined, so the SRM seems well suited for the task. The problem of optimal assignment and allocation of buffer pool space to different classes of work has had little prior work. Brown [2] develops a technique called *class fencing* which dynamically allocates buffer space to classes, in order to meet a hit rate or response time goal. The method is heuristic and operates on-line, and it was shown by simulation that it is able to meet a set of pre-determined goals, although no claims of optimality are made. Brown's method offers promise for implementation as a buffer manager that incorporates these policies. The method we describe here was designed to work with a partitioned buffer manager, such as that found in DB2 for MVS [5]. As such it also forces us to consider the combinatorial optimization problem of buffer pool assignment, as defined and discussed below.

In other recent work several variations on LRU have been proposed for database buffer pool management. The LRU/2 [12] and the 2Q [7] schemes prioritize the pages according to their *second most recent* access (as opposed to most recent access in the case of LRU). The advantage of these schemes, in comparison to LRU, is in their tendency to remove from the cache pages which are accessed only once in a long period. The question of how to allocate and assign buffers to streams in an environment managed by LRU/2 or 2Q has not been addressed.

2 Model Description and Preliminaries

We consider a fully associative cache operating with the Least Recently Used (LRU) replacement policy (see, e.g, Coffman and Denning [3], Smith [13]). We will adopt the traditional terminology of referring to the entities in the cache as *pages*. To describe the cache operation consider a finite size cache consisting of K page frames or locations, with *positions* or *depths* denoted $1, 2, ..., K$. If a request is made for page p and it is found in position j, p is moved to position 1, and the pages formerly in positions $1, 2, ..., j-1$ are pushed one position "deeper" into the cache, i.e., the position of each of them is increased by 1. If p is not found in the cache it is brought in, and stored in position 1; in this case all the pages formerly in positions $1, 2, ..., K-1$ are being pushed one position deeper and the page at position K is removed from the cache.

The event that page p is requested and found in the cache is called a *hit*; the event that p is requested and not found in the cache is called a *miss*. The fraction of page requests which are hits (resp., misses) is called the *hit ratio* (resp., *miss ratio*). Since the I/O rate to the level of storage below the cache in the hierarchy is directly proportional to the miss ratio, it is desired to decrease the miss ratio as much as possible, or conversely to increase the hit ratio as much as possible.

For modeling purposes we will assume that the streams obey the *stack reference model* (and will be called *stack reference streams*), that is, they are stochastic processes which at steady state choose their next reference according to an independent sampling of the *stack depth distribution*.

Consider N page request streams, $P^1, ..., P^N$, having stack depth distributions $d^1(n), ..., d^N(n)$, respectively. That is when stream i is applied to a LRU-managed cache, the probability that the current request finds the page it references at depth n is denoted $d^i(n)$. Note that $d^i(n)$ is a "defective" distribution in that it may not sum to unity: pages that have never previously been accessed will be assumed to be found at an infinite stack depth. Thus we define $d^i(\infty)$ to be the probability of such an event. Similarly, define $D^i(n)$ to be the cumulative depth distribution of stream i , i.e., $D^i(n) = \sum_{m=1}^{n} d^i(m), n \leq \infty$. Note that if a sample trace of the process is available, then an empirical estimation of the depth distribution of a trace of page-requests can be computed with a single pass through the trace ([9]).

2.1 The buffer allocation and cache assignment problem

We consider the above N page request streams, $P^1, ..., P^N$, L LRU caches, where $1 \leq L \leq N$, and total space to be assigned of K buffers or page frames. Let cache l, $1 \leq l \leq L$, be allocated K_l buffers such that $\sum_{l=1}^{L} K_l = K$. The stream set $\{1, ..., N\}$ is partitioned into L subsets, each corresponding to a single cache and the page requests are now directed to the cache corresponding to their stream.

For example, if there are 4 streams, a possible partition to 3 caches is $\{\{1, 3, \}, \{2\}, \{4\}\}$ in which requests of streams 1 and 3 are directed to cache 1, requests of stream 2 to cache 2, and those of stream 4 to cache 3.

Two problems can now be formulated:

1. **Buffer allocation problem**: Given a partition of the N streams to L caches, find an optimal allocation of the K buffers $K_1^*, ..., K_L^*$, $\sum_{l=1}^{L} K_l^* = K$, which minimizes the overall miss ratio.

2. **Cache assignment problem**: Given L caches, find an assignment of the N streams to the L caches which, under the optimal solution to the buffer allocation problem, minimizes the miss ratio.

Among the various cache assignments, two are of special interest:

- **Fully split assignment**: Under this assignment the number of caches is N and each cache is dedicated to exactly one stream.
- **Fully shared assignment**: Under this assignment the number of caches is 1, to which all streams are assigned.

The optimal allocation of buffers under the fully split assignment is called *the optimal fully split buffer allocation.*

3 Superposition of Request Streams: Properties and Modeling

When two streams are superposed, or combined together, in a single cache, the manner in which they are combined plays an important part in determining the resulting performance.

To illustrate the importance of how the superposition occurs, consider two mutually exclusive streams P_a and P_b, the first requesting pages $a_1, a_2, ..., a_K$ in cyclic order and the second requesting pages $b_1, b_2, ..., b_K$ in cyclic order. Each of the streams separately will achieve 100% hit ratio with a cache of size K or greater, and 0% hit ratio with any smaller cache.

If we obtain the combined stream by pure interleaving, i.e., alternately choosing from each stream, resulting in the sequence $a_1, b_1, a_2, b_2, ..., a_K, b_K$ repeated cyclically, then a shared cache of size $2K$ or more will achieve 100% hit ratio but any smaller size cache will achieve 0% hit ratio. On the other hand, if the method of combining allows long runs or bursts of one stream, then the other, then arbitrarily large hit ratio can be achieved with a cache of just size K. To see this, we borrow the *Kleene* notation and exhibit the sequence $((a_1, ..., a_K)^M, (b_1, ..., b_K)^M)^\infty$, which for a value of M, results in a hit ratio of $(M-1)/M$ for a cache of size K. It is clear that the nature of the combining, or superposition, process is critical in determining the end performance, and in fact in deciding how to assign and allocate the caches.

The most simple assumption about how streams might be combined is obtained by simply switching between them according to random independent trials, i.e., a Bernoulli process. Thus, with probability λ_i, the next page requested is drawn from stream i. Note that if the streams are considered to be "arrival processes" on the time line (as in queueing theory), then the superposition of two arrival processes which are Poisson would result in this method of combining of the streams. The Bernoulli assumption has been successfully applied in a number of contexts, and the resulting cache performance can be analyzed exactly in quadratic (in the cache size) time, or approximately in linear time [8] [11]. In [11] it was shown that the Bernoulli assumption is satisfactory when combining together index and data streams in a database management system, and a rule of thumb was given that as long as the burst lengths of either stream are small in comparison to the cache size, then the Bernoulli assumption and resulting models are satisfactory.

However, if one is seeking the optimal assignment of caches, we will now show that a Bernoulli assumption always directs the designer towards a fully split arrangement! We define a cache management algorithm for multiple streams as having the *per-stream LRU property* if it satisfies the following: if k pages of stream i $(i = 1, ..., n)$ are retained, then they are the most recently accessed k pages of stream i. Note that a scheme that combines a number of streams into a smaller number of LRU caches has this property.

Proposition 1 (Optimality of fully split assignment) *Consider N stack reference request streams, and a cache management algorithm having the per-stream LRU property. If we assume that stream superposition occurs according to the Bernoulli switching model, then within this class of cache algorithms, the cache miss ratio is minimized by the fully split arrangement in which each stream is given its own LRU cache and a fixed optimal buffer allocation.*

Proof: Recall that $D^i()$ denotes the cumulative depth distribution of stream i, $i = 1, ..., N$, and let λ_i be the probability that a request of the combined stream belongs to stream i.

For the class of cache algorithms considered (having the per-stream LRU property), if just prior to the t'th request there are $K^1(t), ..., K^N(t)$ pages of types 1, ..., N present in the entire cache, then by the Bernoulli and stack reference stream assumptions, the probability that the t'th request misses is

$$Pr[MISS(t)] = \sum_{i=1}^{N} \lambda_i (1 - D^i(K^i(t))). \tag{1}$$

In view of the form of this equation, the minimization of $(1/T) \sum_{t=1}^{T} Pr[MISS(t)]$ for arbitrary integer $T \geq 1$ reduces to the minimization of $\sum_{i=1}^{N} \lambda_i (1 - D^i(K_i(t)))$ for each t, subject to the common constraints $\sum_{i=1}^{N} K_i(t) \leq K$ and $0 \leq K_i \leq K, i = 1, ..., N$.

The optimal solution is thus the parameters $K_i^*, i = 1, ..., N$ which minimize $\sum_{i=1}^{N} \lambda_i (1 - D^i(K_i^*))$, i.e., the static fully split optimal assignment. □

But when the stream superposition is not Bernoulli, and contains long bursts from one stream, poor sharing performance which is far from optimal can result. This has already been illustrated by the cyclic stream $((a_1, ..., a_K)^M, (b_1, ..., b_K)^M)^\infty$ that was discussed at the beginning of this section. For this stream, if a cache of size K was available the optimal split cache allocation would result in a miss ratio of .5, whereas a fully shared cache would result in a miss ratio of just $1/M$.

The above result and illustration show that if no "memory" or burst characteristics are present in the superposition process, then we can proceed with separately optimized buffers. However, if there is an underlying burst process, then one needs to exploit this in the *time sharing* of the buffer resources. Thus, in order to properly solve the cache assignment problem, we must characterize this burst process, if present, and use it in our optimization procedure. In this paper, we will use a very simple characterization, namely we will measure the mean lengths of the alternate bursts of each type when two streams are considered superposed together. The next section is devoted to analyzing cache behavior under this and further assumptions.

4 Superposition of Two "Bursty" Streams

We will consider two stack reference streams P_1 and P_2, and evaluate the miss ratio of the combined stream when applied to a shared cache under assumptions that the superposed stream contains alternate bursts of each stream. To model the burstiness in the combining of the two streams, we will first transform the depth distribution characterization of a stack reference process to an inter-reference distribution. It will then be relatively simple to calculate the impact of the interruption of one stream by another as an increase in inter-reference distance, and then we can transform back to the depth characterization of the combined stream. To carry out this analysis, we first must derive the conditional distribution of the inter-reference distribution of a single stream in isolation.

4.1 Analysis of Conditional Inter-reference Distance

Consider a single stack reference stream applied to an infinite LRU cache with the stream having depth density function and cumulative distribution function of $d()$ and $D()$, respectively. Let I_i be a random variable representing for an arbitrary request the number of requests in the stream since the same page was last requested, conditioned on the requested page being found at depth $i \leq \infty$. I_i is called the *conditional inter-reference distance*. Let $I_i(z)$ and $E[I_i]$ be its generating function and expected value respectively. These variables can be computed by tracing the trajectory of a tagged page p within the cache. For simplicity of presentation assume that each request takes one unit of time; thus measuring distances within a stream is identical to measuring time.

Let R_i be a random variable denoting the duration for which the *tagged page* stays at position i; also let $R_i(z)$ and $E[R_i]$ be the generating function and expected value of R_i. Define $\overline{x} \stackrel{\text{def}}{=} 1 - x$ and $D(0) = 0$. By the properties of the stack reference stream, for the page to stay at position i for exactly k time units it must be that $k-1$ accesses occur to elements more recently accessed than p, followed by an access to p or a page less recently accessed than p. Thus the distribution of R_i is given by:

$$P[R_i = k] = \overline{D}(i-1)D(i-1)^{k-1} \qquad k \geq 1; i \geq 1 \tag{2}$$

and its generating function and expected value are given by:

$$R_i(z) = \frac{z\overline{D}(i-1)}{1 - zD(i-1)} \qquad E[R_i] = \frac{1}{\overline{D}(i-1)} \qquad i \geq 1;\ \overline{D}(i-1) > 0, \tag{3}$$

.

These equations are only given for i such that $\overline{D}(i-1) > 0$, since the tagged page can never reach locations indexed by other values of i.

After staying at position i, a page may either move to position $i+1$ (with probability $\overline{D}(i)$) or to position 1 (with probability $d(i)$). Since these events are independent of the duration for which the page stayed at position i, we have:

$$P[R_i = k|\text{page moves to } i+1] = P[R_i = k|\text{page moves to } 1] = P[R_i = k] \tag{4}$$

I_i can now be computed by noting that a page that is being retrieved from location i must have followed the cache trajectory $1, 2, ..., i$. Thus, we have:

$$I_i = R_1 + R_2 + ... + R_i$$

where the sum represents a sum of *independent* random variables. This leads to:

Proposition 2 (Conditional Inter-reference Distribution of a Stack Reference Stream) *In a stack reference stream with cumulative depth distribution $D()$ and complement $\overline{D}()$, the conditional inter-reference distribution of any page, given that it is found at depth i, has generating function*

$$I_i(z) = \prod_{j=1}^{i} R_j(z) = \prod_{j=1}^{i} \frac{z\overline{D}(j-1)}{1 - zD(j-1)} \qquad i \geq 1;\ \overline{D}(i-1) > 0, \tag{5}$$

and mean

$$E[I_i] = \sum_{j=1}^{i} \frac{1}{\overline{D}(j-1)} \qquad i \geq 1;\ \overline{D}(i-1) > 0. \tag{6}$$

.

It is interesting to check that if $D(K-1) < 1$ and $D(K) = 1$, then

$$\sum_{j=1}^{K} E[I_i]d(i) = K, \tag{7}$$

i.e., the expected inter-reference distance for any page is K, which is a well known (unconditional) property for a stack reference stream [3].

4.2 Conditional Depth and Distance in Merged Streams

We can now compute the effect of stream superposition on the inter-reference distribution of an arbitrary page from one of the streams. We take into consideration both the depth characterization of the individual streams and switching process between them [1]. Since our final objective is to solve the cache assignment problem, this analysis will be used many times as a component of an optimization procedure. Thus our emphasis is on a computationally very efficient technique. Our method for doing this will be to replace several random variables by deterministic versions with the correct mean.

Consider two stack reference streams P_1 and P_2, with depth distribution functions $d^1()$ and $d^2()$ and cumulative distribution functions $D^1()$ and $D^2()$. Assume that P_1 and P_2 are superposed to form a single stream P. This combined stream will consist of alternating bursts of each type of page. Let b_1 and b_2 be the expected size of type 1 and type 2 bursts in P.

Consider an arbitrary type 1 page, denoted p, whose depth in P_1 is a random variable $\mathcal{D}_1$. Let $\mathcal{R}_1$ be a random variable denoting the inter-reference distance of p in P_1. Similarly, let $\mathcal{D}$ and $\mathcal{R}$ be random variables denoting the depth and inter-reference distance of p in the combined stream P. Given that $\mathcal{D}_1 = \delta_1 < \infty$, we would like to find both $\mathcal{D}$ and $\mathcal{R}$, from which we then compute the statistics of the superposed stream.

First, from the depth distribution $d^1()$ and the analysis of the previous section we know the inter-reference distribution of p within P_1, i.e., $\mathcal{R}_1$. To reduce the computational complexity we will take $\mathcal{R}_1$ as a deterministic random variable with its correct mean. Thus we will take

$$\mathcal{R}_1 = E[I_{\delta_1}], \qquad \textit{with probability } 1. \tag{8}$$

[1] We are not aware of any previous attempts to solve the bursty stream superposition problem. A distantly related reference [6] models geometrically distributed task switch intervals, but at the end of each interval a cache flush is considered to occur.

Next, to derive the depth and inter-reference distance of p within P we need to find the number of type 2 requests (denoted $\mathcal{N}$) which have been made between the two successive requests for p. Again here, we will assume that the burst lengths are deterministic, i.e., all type 1 bursts are of the same size and all type 2 bursts are of the same size (given by b_1 and b_2 respectively) [2]. Since p is an arbitrarily chosen page, its location within the (type-1) burst is uniformly distributed.

Under the above assumptions, the value of $\mathcal{N}$ is either $b_2\lfloor\frac{\mathcal{R}_1}{b_1}\rfloor$ (with probability $1 - \frac{\mathcal{R}_1}{b_1} + \lfloor\frac{\mathcal{R}_1}{b_1}\rfloor$) or $b_2(1+\lfloor\frac{\mathcal{R}_1}{b_1}\rfloor)$ (with probability $\frac{\mathcal{R}_1}{b_1} - \lfloor\frac{\mathcal{R}_1}{b_1}\rfloor$). Thus, the distance of p within the combined stream P is approximated by:

$$\begin{aligned} Pr[\mathcal{R} = \mathcal{R}_1 + b_2\lfloor\tfrac{\mathcal{R}_1}{b_1}\rfloor] &= 1 - \tfrac{\mathcal{R}_1}{b_1} + \lfloor\tfrac{\mathcal{R}_1}{b_1}\rfloor \\ Pr[\mathcal{R} = \mathcal{R}_1 + b_2(1+\lfloor\tfrac{\mathcal{R}_1}{b_1}\rfloor)] &= \tfrac{\mathcal{R}_1}{b_1} - \lfloor\tfrac{\mathcal{R}_1}{b_1}\rfloor \end{aligned} \tag{9}$$

The evaluation of $\mathcal{D}$ is similar to that of $\mathcal{R}$. The distinction is that in this derivation we must account for the number of *distinct* type-2 pages that have been requested between the previous request and the most recent request of p (denote this number by $\mathcal{N}'$). In Appendix A we derive a simple recursion for computing the probability that a sequence of $\mathcal{N}$ requests contains $\mathcal{N}'$ ($\mathcal{N}' \leq \mathcal{N}$) distinct requests. These recursions can be used to either derive the exact distribution of $\mathcal{N}'$ (given the value of $\mathcal{N}$) or its expected value. An *approximated* value of $\mathcal{N}'$ can be derived by noting that the ratio between $\mathcal{N}'$ and $\mathcal{N}$ tends to (for large $\mathcal{N}$) the probability that a type-2 page is found at depth infinity at P_2, i.e., $d^2(\infty)$. Thus we have:

$$\mathcal{N}' \approx \mathcal{N} d^2(\infty) \tag{10}$$

This limit can be understood by realizing that $d^2(\infty)$ is the rate of introduction of new type 2 elements into the cache.

From Equation 10 the depth of p within P, is thus approximated by:

$$\begin{aligned} Pr[\mathcal{D} = \delta_1 + b_2\lfloor\tfrac{\mathcal{R}_1}{b_1}\rfloor d^2(\infty) | \mathcal{D}_1 = \delta_1] &= 1 - \tfrac{\mathcal{R}_1}{b_1} + \lfloor\tfrac{\mathcal{R}_1}{b_1}\rfloor \\ Pr[\mathcal{D} = \delta_1 + b_2(1+\lfloor\tfrac{\mathcal{R}_1}{b_1}\rfloor) d^2(\infty) | \mathcal{D}_1 = \delta_1] &= \tfrac{\mathcal{R}_1}{b_1} - \lfloor\tfrac{\mathcal{R}_1}{b_1}\rfloor \end{aligned} \tag{11}$$

4.3 Algorithm COMBINE: Superposing Two Bursty Stack Depth Processes

The analysis in Section 4.2 yields a simple algorithm for approximating the depth distribution of the superposition of two stack depth streams from their individual depth distributions and their burst sizes. Assume that $d^i(n)$ $(i = 1, 2)$ is defined for $1 \leq n \leq K$ and for $n = \infty$. The process for computing the depth distribution of stream 1 requests within the superposed process is:

Algorithm COMBINE

[2] Note that these deterministic assumptions are somewhat well motivated in database systems because we often see cyclic references to all pages within a table (known as *tablescans*), e.g., in the nested-loop join procedure of relational database. However, we will evaluate the effect of these assumptions in the section on experimental results.

1. Use Equation 6 for computing $E[I_i]$, $i = 1, ..., K$ (replace $d()$ and $D()$ in those equations by $d^1()$ and $D^1()$ respectively).
2. For $\delta_1 = 1, ..., K$ use Equation (11) (in which $\mathcal{R}_1$ is substituted by $E[I_{\delta_1}]$ (Equation (8)) to compute the depth distribution of stream 1 requests within the combined stream.

Similar steps are used to compute the depth distribution of stream 2 requests within the superposed stream.

It is clear that Algorithm COMBINE requires $O(K)$ operations. Note that the results of step 1 above can be used for superposing Stream 1 with any other stream. Thus, if one examines the superposition of Stream 1 with several alternative streams, step 1 for stream 1 needs to be executed only once.

5 Algorithms for Efficient Buffer Allocation and Cache Assignment

We now present an algorithm for efficient buffer allocation and cache assignment. The input of the algorithm is a trace of the combined page request stream, and the number of cache buffers to be allocated; the output is the assignment of streams to cache pools and allocation of buffers to the pools. The trace is an ordered sequence which provides for every request two identifiers: 1) the page identifier, and 2) the stream identifier[3]. Several procedures are used as building blocks for the algorithm; these are described in the following sub-sections (5.1, 5.2 and 5.3). Finally, using these building blocks we describe the complete algorithm in Section 5.4.

5.1 Algorithm 1: Computing Depth Distribution

A finite sequence of R page requests, $P = p_1, p_2, ..., p_R$, where $p_i = (t_i, m_i)$, $1 \leq t_i \leq N$ is the stream index ($1 \leq t_i \leq N$) and m_i is the page index, is given. It is assumed that the individual streams are mutually exclusive, namely if $t_i \neq t_j$ then $m_i \neq m_j$.

The *projection* P^t of P on t ($1 \leq t \leq N$) is defined as: $P^t = \{p_i | t_i = t\}$. In other words P^t is the tth stream. Let $T \subseteq \{1, ..., N\}$ be a subset of the streams, The *projection* of P on the *set* T , P^T is defined as: $P^T = \{p_i | t_i \in T\}$.

Given the stream P it is required to compute the depth statistics of P^t (the projection of P on t), for $1 \leq t \leq N$.

This computation can be done by simply simulating N LRU cache system. The simulation is done by constructing N data structures representing the N caches and inserting p_i into cache t_i according to the LRU scheme, while recording the depth at which p_i is found (or recording ∞ if p_i is not found). The size of each of the caches used in the simulation can be bounded from above by K, the number of buffers to be allocated by the optimization algorithm (note that using a larger cache is redundant). To efficiently implement a cache, note that the following operations must be performed:

[3] In a database application a stream identifier might reflect the identification of a table or index; in a processor application it might reflect a process identifier and whether the request is for instruction or data.

1. Finding p_i in the cache, if present.

2. Inserting p_i at position 1 while shifting a group of pages (those that where less deep than p_i when found) one position deeper.

The following structures can be used to achieve an efficient simulation:

1. Construct a 2-3 tree (or AVL tree) (see, e.g., [1]) in which the leaves represent the pages in the cache, ordered according to the same order as in the cache. The keys (internal nodes) in this tree are the page positions within the cache.

2. Construct a hash table (see, e.g., [1]) in which the keys are the page identifiers, and each of whose elements points to the corresponding leaf in the 2-3 tree.

A search for a page on these structures is done via the hash table, while the restructuring of the cache is done using the 2-3 tree. The time complexity of simulating a trace of length R requests is therefore $O(RlogK)$.

5.2 Algorithm 2: Computing Expected Burst Size

Given the finite page request sequence P of length R, consisting of a combining of the sequences P^1, P^2, ..., P^N, it is required to find for every two streams t_1 and t_2 ($1 \leq t_1, t_2 \leq N$, $t_1 \neq t_2$), the mean burst size of each of the two page types within the sequence consisting of only the requests from streams P^{t_1} and P^{t_2}. This computation can be done for *all pairs* t_1 and t_2 using a *single run* over P. The time complexity of this step is $O(RN)$. The resulting matrix of expected burst sizes for all pairs effectively summarizes the "interaction" between streams.

5.3 Algorithm 3: Optimal Buffer Allocation

Consider a system consisting of L caches and L page request streams. Let the depth distribution of process $1 \leq i \leq L$ be $D^i()$, and λ_i be the proportion of requests coming from stream i. Assume that cache i is assigned to process i and that the total number of buffers in the system is $K > 0$. It is required to find an optimal buffer allocation $K_1^*, ..., K_L^*$ ($\sum_{i=1}^{L} K_i^* \leq K$, $K_i^* \geq 0$) which maximizes the hit ratio. In other words, $K_1^*, ..., K_L^*$ is an optimal allocation if for any other allocation $K_1, ..., K_L$ ($\sum_{i=1}^{L} K_i \leq K$, $K_i \geq 0$), $\sum_{i=1}^{L} \lambda_i D^i(K_i^*) \geq \sum_{i=1}^{L} \lambda_i D^i(K_i)$.

The optimal solution of this problem can be found using dynamic programming ([10]). The idea is to advance the number of caches ℓ from 1 to L. If the optimal allocation is obtained for $\ell < L$ caches using k buffers, for all k with $0 < k \leq K$, then the optimal allocation for $\ell+1$ caches is the allocation resulting in the lowest miss ratio of any allocation in which k^* buffers are allocated to the first ℓ caches and $K - k^*$ buffers added to the remaining cache. The time complexity of this algorithm is $O(K^2L)$.

5.4 The Complete Algorithm: Efficient Cache Assignment

Given the sequence P consisting of page requests originating from N different streams, the total number of buffers K, and the number of caches, $N_C \leq N$, it is required to find an efficient assignment of each of the tables to one of the caches and an allocation $K_1, ..., K_{N_C}$ of the buffers to the caches.

The algorithm proposed below is a greedy algorithm that starts from a completely split architecture (each of the N streams is directed to a separate cache) and repeatedly applies a selective superposition of two streams until the number of streams and caches reaches N_C.

An outline of the algorithm is as follows: We start with a representative trace from the system on which we wish to solve the cache assignment problem. Algorithm 1 is used to efficiently find the depth distributions of each type from this trace. Algorithm 2 is used to obtain the burst lengths of any pair of streams when superposed. Then Algorithm 3 is applied to provide the optimal assignment and buffer allocation for $N_C = N$. Algorithm COMBINE (Section 4) can be used to estimate the depth distribution resulting from the superposition of any two of the N streams. A comparison of the hit ratio of the superposed stream with that of the two separate streams can now be used as a measure for evaluating the payoff resulting from a potential combined of the two. We do this for all $N(N-1)/2$ pairs to find the best two candidates for merging. Note the value of doing this via the fast approximate scheme embodied in COMBINE, rather than processing the whole trace again. After merging these two streams, the whole process can now repeat with the resulting $N-1$ streams.

The complete cache assignment algorithm is thus:

CACHE ASSIGNMENT Algorithm

1. Apply Algorithm 1 to P, and compute the cumulative depth distribution ($D^i()$) of each of the N streams.

2. Apply Algorithm 2 to compute the expected burst size of each stream within every pair of streams.

3. Using the $D^i()$, $i = 1, ..., N$, apply Algorithm 3 to compute the optimal fully split buffer allocation; let K_i^* denote the allocation to stream i.

4. Using Algorithm COMBINE, compute the (approximated) depth distribution of all streams resulting from merging every pair of streams. Let $D^{i,j}()$ denote the cumulative depth distribution of the resulting stream when streams i and j are superposed.

5. For every $1 \leq i, j \leq N$ $(i \neq j)$ compute $\lambda_i D^i(K_i) + \lambda_j D^j(K_j) - (\lambda_i + \lambda_j) D^{i,j}(K_i + K_j)$. This is an approximation for the decrease in number of misses if streams i and j are superposed. Let i^* and j^* be the values of i, j which maximize the difference.

6. Merge i^* and j^* into one stream, say i^*; update all required data structures (depth distributions and burst lengths), reapply Algorithm 3, set $N := N - 1$ and rename stream indices. If $N = N_C$ stop; otherwise go to step (5).

A variation on the above algorithm in step 6 is to recalculate the depth distributions and burst lengths, and reapply Algorithm 3, only occasionally (i.e., after several merge steps). This will speed up the algorithm at the possible expense of some reduced performance.

6 Experimental Results

To examine the performance of the algorithms developed in this paper, we conducted an experiment in which the algorithm was used to determine an efficient buffer allocation for the database system DB2 (see [5]). DB2 allows multiple buffer pools and allows the user to specify which objects (tables or indexes) are assigned to which buffer pools. Thus within the context of this paper, we will consider all the requests coming to one object to be one stream, and ask to which buffer pool it should be assigned.

It should be noted that DB2, and other database systems, do not use a pure-LRU strategy. They depart from it in several key ways[5]. Perhaps the most important departure is in the use of prefetch strategies that prefetch pages, either sequentially or from a list, before the pages are requested. Prefetch schemes are designed to bring in pages just before they are requested, and so improve the throughput and response time of a query. Pages whose requests are not known in advance are still read in on demand (synchronous reads). ¿From the point of view of pages entering the buffer pool, prefetches can be considered to simply replace misses. The only difference is that they might enter the buffer pool slightly earlier: when the prefetch scheme is working properly that time difference should be much smaller than the mean residency time of a page in the buffer pool. A second departure is the use of sequential steal thresholds to prevent sequential accesses from flooding the buffer pool. To counteract this effect in the DBMS, sequential pages are also managed on an additional list called the SLRU list, and pages on this list are stolen preferentially when a threshold in population of sequential pages is observed [15]. One of the purposes of separately managed and sized buffers, as provided by the assignment/allocation technique of this paper, is to prevent the activation of these thresholds. Where it is attractive to separate the treatment of sequential and non-sequential accesses, our methods are intended to do that. Thus we did not explicitly account for this effect. Thus, while buffer managers are not pure-LRU, LRU remains the single most dominant dynamic in managing existing commercial database buffer pools, and it is that characteristic on which we have decided to base our modeling. In fact the confirmation in the lab of the effectiveness of the present technique tends to support this approach.

In the experiments, we started with a large database installation running a mix of simple transaction and complex queries. Initially, the system was configured so all tables and indexes were applied to a single large buffer pool. A low overhead tracing facility was used to collect every logical I/O made to this buffer pool resulting in a trace of 458,213 page requests directed to 47 objects. This trace was then fed into the cache assignment algorithm to determine an efficient object to buffer pool assignment and an optimal buffer pool sizing. We then reconfigured the buffer pool in the DB2 system (following the algorithm's recommendation) and applied the same workload to this configuration.

We use this experiment to examine two questions: (1) The accuracy of the superposition approximation in predicting the depth distribution (hit ratio function) for mixed streams, and, (2) The quality of the cache assignment (and allocation) proposed by the cache assignment algorithm. The results of these examinations are reported in Sections 6.1 and 6.2 respectively.

6.1 Evaluation of the Superposition Approximation

Of the 47 individual streams that constitute the combined page request stream we selected various pairs and examined the quality of the superposition analysis. We have chosen three representative pairs to report in this paper. For each pair we compute the cumulative depth distribution function, namely the hit ratio as a function of the number of buffers, when the two streams combined are applied to a single cache. We derive this measure using three procedures:

1. **Exact:** The exact cumulative depth distribution, computed by the expensive process of simulating the application of the two table superposed stream to a joint cache buffer. Note that the precise superposition of the requests of these two streams can be extracted by projection from the 47 table streams.

2. **Bursty Stream Approximation:** The approximation of the two table superposed stream depth distribution as proposed in Section 4.

3. **Bernoulli Approximation:** The approximation of the two table superposed stream depth distribution using a model which assumes Bernoulli switching between the streams ([8], [11]). This derivation would be exact if the streams were combined according to a Bernoulli process. The Bernoulli assumption has been shown to be acceptable if we superpose, for example, the index and the data streams, and in general when the length of bursts is small compared to buffer size [11]. However, it may not be accurate when considering objects such as an individual table that may only be used once in a while.

Figure 1 depicts the results for merging streams 28 and 41. The burstiness of these streams is extremely large with respect to reasonable buffer sizes, in fact the sequence consists of 10410 requests for stream 28 followed by 3203 requests for stream 41 (thus there is no variation of the burst size). The deterministic superposition approximation performs extremely well at this case, while the Bernoulli approximation performs poorly.

Figures 2 and 3 depict the results for merging streams 1 and 30. Stream 1 contains 4332 requests while stream 30 contains 99695 requests. The amount of intermixing here is high and the expected burst sizes are about 5 for stream 1, and 114 for stream 30; the coefficient of variation of the burst size is about 3.0 for stream 1 and 1.63 for stream 30 (thus the burst sizes are quite variable). Both predictions deviate somewhat from the exact results in some regions but predict the "jumps" in the curve properly. As might be expected, the Bernoulli approximations tend to round off the "jumps" somewhat and therefore will perform less well in the buffer allocation application, since they predict delivery of hits from a buffer allocation prior to the point at which it occurs [4].

Note that the Bernoulli approximation is quite poor for small buffer sizes (less than 120): this results from not accounting for the burst size of stream 30 which tends to "flush" a small cache. Also, although the burst lengths has reasonably high coefficients of variation, this did not adversely affect the approximations.

[4] In a practical allocation of buffers in the presence of jump behavior, it would always be advisable to add a small amount of buffer space (safety factor) to account for modeling errors and prevent optimizing to a point "on the edge of a ravine".

The third case considered is that of streams 16 and 32. Figures 4 and 5 depict the results for superposing these streams. The expected burst sizes are about 12 and 11 (the number of page requests in the streams are 3624 and 3100). Both the deterministic and the Bernoulli approximations perform quite effectively, except at very small cache sizes where the method of Section 4 is superior.

Note that the variability of the burst sizes in this case are even higher (the coefficient of variation for the burst size is 4.9 for stream 16 and 3.65 for stream 32), yet the deterministic approximation still performs well.

We may conclude from these examples that the bursty stream superposition approximation predicts the hit ratio in the superposed stream quite effectively, and it serves the optimization procedure well by properly predicting the major characteristics (e.g., jumps) of the hit ratio curve. The Bernoulli approximation is shown not to have adequate performance for this particular application; indeed, for the reason of Proposition 1 it would not be applicable to the cache assignment optimization procedure.

6.2 Evaluation of the Cache Assignment Algorithm

We test the quality of the cache assignment algorithm in two ways. First, we apply it to the data at hand and see if it is able to obtain an improvement over the fully split arrangement. To see if the resulting recommendations are useful, we also compare it to a procedure by which we blindly choose random streams to be combined. Second, after obtaining the recommended cache assignments, we reconfigure the DB2 system to use these assignments and run the workload again to see if any performance improvement results.

Simulation results: Out of the 458,213 page requests in this experiment, 230,620 were found to be "compulsory misses", or "cold start misses" [13], in the sense that regardless of the cache assignment strategy these requests will end up with a miss, i.e., these requests consist of those for which this is the first time the page is requested in the duration of the trace. The number of compulsory misses could possibly be reduced by using a longer trace. The evaluation of the algorithm is therefore based on the number of misses it achieves out of the remaining 227,593 requests. The number of such misses when applying the page request stream to a simulation of a single shared cache with 5000 pages was 30,927 (13.6%). Then the cache assignment algorithm was run to determine an efficient assignment of the tables to 3 buffer pools, still using a total buffer size of 5000 pages. The number of non-compulsory misses under this assignment was 4,909 (2.16%).

The algorithm was also run to determine an assignment for a cache consisting of $n = 2, 3, \ldots, 47$ buffer pools. Figure 6 depicts the miss ratio (as fraction of the non-compulsory misses) for the cache assignments derived by the algorithm as function of the number of buffer pools. The rightmost point represents the fully split assignment while the left most point represents the fully shared assignment. Also depicted in this figure are the same measures for an algorithm which combined the cache pairs at random. The figure represents a case in which the total number of buffers allocated is 5000.

These figures show that splitting the cache into a small number of pools (about three), leads to a significant performance improvement. Proper cache assignment plays a major role in this improvement: the number of non-compulsory misses under the random assignment configuration is up to four times higher than that under the assignment of the algorithm, although in practice this improvement will be diluted by the effect of compulsory misses.

It is doubtful whether an administrator would want to use more than two or three buffer pools, even though DB2 in its Version 3.1 does allow 60 buffer pools to be used. A larger number would allow only marginal improvement, but may introduce excessive dependence on the trace data input to the analysis - in other words we would expect a two or three cache solution to be more robust.

DBMS Lab Test of the Recommendations:

The trace we used in the previous section was obtained from the DB2 Database Management System running a realistic workload under realistic conditions. In order to see whether the recommendations of the previous section were practically useful, and would actually improve performance, we adopted the recommendations for the cache assignment by reconfiguring the DB2 buffer pools. Note that the DB2 buffer pools will correspond to what we have been calling caches in this article. We ran DB2 both with a single shared buffer pool and with three buffer pools with objects assigned to these buffer pools according to the results of the previous section.

Details of the setup and the raw results are provided in Appendix B. The net results are quite encouraging, although they are clouded somewhat by some differences between DB2 operation and our model, and some aspects of the experimental procedure. This clouding was acceptable to us because we felt we were balancing the two requirements of having a simple efficient cache assignment process, but one whose success was demonstrated in a test under realistic workload and operational conditions. As described above, the lab test environment consisted of a workload consisting of a mixture of transactions (with updates) together with a complex read-only queries. The experimental procedure was to run all the query work once, i.e., a "batch" operation, while running the transactions continuously in a "closed loop" until the query work completed. We also followed the common practice of giving the transactions CPU priority. One advantage of this procedure is the assurance that we were not "overfitting", i.e., optimizing artifices of the particular trace that we analyzed: in the second run, the transactions behaved quite differently and in fact achieved a considerably higher throughput. Thus, in the two experiments, the order of page requests would have been totally different.

The key result from our experiments was that after cache assignment, the miss rate for queries dropped by approximately 6% and the miss rate for transactions by 29%. Note that for the query work, there was a number of table scans which resulted in many compulsory misses: DB2 treats these as known sequential I/O's and the pages are prefetched accordingly, as described at the beginning of this section. But it should be remembered that miss rate is directly proportional to physical I/Os, so the miss rate reductions reported are quite significant. For the transaction work, the improved miss rate resulted in a considerably (6%) higher throughput.

It is interesting to compare the miss rate reduction predicted from the model (Figure 6) with that observed in the lab. Figure 6 shows that going from one buffer to three buffers, optimally allocated, should result in a *non-compulsory* miss rate reduction from 13.6% to 2.16%. Now the trace shows that for almost every I/O which is a compulsory miss (230620) there are an equal number of I/O's (458213-230620 = 227593) which do not have to be misses. Thus the model predicts that overall miss rate should decline from 230620 + 227593*.0216 to 230620 + 227593*.136, or a reduction of about 10%. Now the total number of physically read pages in the lab experiment dropped from 285423 to 263825, or a 8% drop. Note that we would expect the drop in the lab to be slightly smaller than the model since the structure of the lab experiment caused a slight increase in the total number of overall logical I/O requests, due to the higher number of transactions completed.

In interpreting Table 1, we remind the reader that in the experiment a single batch of query work was run until completion, while transactions continued to run. Since transactions were particular beneficiaries of the improved buffer pool performance, they required fewer physical I/O's, and so their CPU utilization, as well as the overall CPU utilization, increased. Since transactions had priority at the CPU, the CPU resource remaining for the query work decreased slightly and, despite the also improved efficiency of the query workload, the entire experiment ran for slightly longer. Nevertheless, the overall performance was considerably improved.

The net result from this experiment is encouraging: considerably improved I/O performance, particularly for time-critical transactions, can be obtained by properly splitting and allocating a fixed set of buffering resources. The question does arise as to whether workloads are sufficiently repeatable that such methodologies can be used to carry out realistic optimization in complex commercial environments. We have had some experience with the allocation part of the methodology and found that it can cause considerable improvement in performance when recommendations are based on analysis of traces that were collected weeks or months before in the same production environment. Nevertheless, if current workloads are completely unrelated to those from which traces were obtained, the recommendations based on those traces may not be useful. However the trace collection methods and optimization methods described here are sufficiently inexpensive that there is also the potential to apply them in an online tuning system. This is an area for further work.

7 Conclusion

The intuitive tradeoffs in cache assignment have become quite clear, and in many cases a designer will be content with either a fully split or fully shared cache. But in other cases, as we have shown, there can be considerable payoff in carrying out an analysis, such as the cache assignment algorithm we have described, to obtain improved performance with no increase in memory resource expenditure. There are many aspects to the complete solution of this problem that we have only just begun to address. For example, the designer needs to describe the granularity of the objects that are being assigned: in the database context, are they individual objects, or are they groups of objects? Other variations on the problem also suggest themselves, for example, if one wishes to use cache assignment to allow a certain class of work to be accorded preferential performance, then the algorithms need to be modified. Further experimentation, particularly in contexts other than database would undoubtedly yield interesting insights.

8 Acknowledgements

The authors acknowledge Dick Mattson who provided us with some trace processing software, and Irv Turkfeld who successfully applied parts of the present methodology at several large commercial database installations. We also acknowledge three careful reviews that considerably improved the clarity of the paper.

References

[1] A. V. Aho, J.E. Hopcroft and J.D. Ullman, "Data Structures and Algorithms," Addison Wesley, 1983.

[2] K. P. Brown, "Goal Oriented Memory Allocation in Database management Systems," Ph. D. Thesis, University of Wisconsin - Madison, 1995, *http://ww.cs.wic.edu/ brown/brown.html.*

[3] E.G. Coffman, Jr. and P. J. Denning, "Operating System Theory," Prentice-Hall, 1973.

[4] A. Dan, P. Yu, and J. Y. Chung, "Characterization of Database Access Pattern for Analytic Prediction of Buffer Hit Probability," VLDB Journal 4, 127-154, 1995.

[5] IBM Systems Journal, Vol 23, No. 2, 1984.

[6] I. J. Haikala, "Cache Hit Ratios with Geometric task Switch Intervals," Proc. 11th Annual Symposium on Computer Architectures, Ann Arbor, MI, June, 1984, 364-371.

[7] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm ," *Proceedings of the 20th International Conference on Very Large Data Bases, (VLDB'94)*, September 12-15, 1994, Santiago de Chile, Chile pp. 439-450.

[8] H. Levy and R. J. T. Morris, "Exact Analysis of Bernoulli Superposition of Streams into a Least Recently Used Cache," *IEEE Transactions on Software Engineering*, Vol. 21, No. 8, August 1995, pp. 682-688.

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

[10] R.L. Mattson, Personal communication.

[11] R.J.T. Morris, "Analysis of Superposition of Streams into a Cache Buffer," ACM Sigmetrics '93, Santa Clara, California, May 1993.

[12] E.J. O'Neil, P.E. O'Neil and G. Weikum, "The LRU-k page replacement algorithm for database disk buffering," in *Proceedings of the 1993 ACM Sigmod International Conference on Management of Data*," pp. 297-306, 1993.

[13] A. J. Smith, "Cache Memories," ACM Computing Surveys, Vol. 14, No. 3, Sept. 1982, 473-530.

[14] H. S. Stone, J. Turek and J. L. Wolf, "Optimal Partitioning of Cache Memory," *IEEE Transaction on Computers*," Vol. 41, No. 9, Sep. 1992, 1054-1068.

[15] J. Z. Teng and R. A. Gumaer, "Managing IBM Database 2 Buffers to Maximize Performance," *IBM Systems Journal,* Vol. 23, No. 2, 1984.

A Deriving The Number of Distinct Requests in a Stream

Consider a stack reference stream $p_1, p_2, \ldots$ whose depth density function and cumulative distribution functions are $d()$ and $D()$, respectively. Let k_{max} be the highest integer for which $d()$ has a non-zero value, namely $d(k) = 0$ for any $k > k_{max}$; Let $d(\infty)$ be the probability that a new request is found at depth infinity, i.e., it is not found in the cache), $d(\infty) = 1 - D(k_{max})$.

Consider a sequence of k, $k \geq 1$, consecutive requests from this stream, $p_j, p_{j+1}, \ldots p_{j+k-1}$, for some arbitrary $j \geq 1$. We would like to calculate $DIST_k$, the number of *distinct* references contained in this sequence. For example if the sequence is $1, 2, 1, 7$ then it contains three distinct references.

To derive this measure we simulate the operation of an LRU cache using the depth distribution. We let it operate for k requests and examine the probability that there are n distinct references, namely $Pr[DIST_k = n]$; we denote this probability by $P(n, k)$. The exact computation of $P(n, k)$ can be done recursively using the following arguments:

1. A sequence consisting of one request contains exactly one distinct reference.

2. A sequence of k requests ($k > 1$) contains a single distinct reference if its prefix (of length $k - 1$) contains a single distinct reference, and if the k'th request is made to the element that was requested on the *k-1*'st request; note that the latter event occurs if and only if the reference requested at the k'th request is at position 1 of the cache and thus the probability for this event is $D(1)$.

3. A sequence of k requests ($k > 1$) contains n distinct references ($n > 1$) only under either one of the following situations:

   - The prefix (consisting of $k - 1$ references) of the sequence contains exactly n distinct references (which are therefore at cache positions $1, \ldots, n$ prior to the k'th request) and the k'th request is made to one of those references (made with probability $D(n)$).

   - The prefix of the sequence contains exactly $n - 1$ distinct references (which are therefore at cache positions $1, \ldots, n - 1$ prior to the k'th request) and the k'th request is made to an element that *is not* one of those references (made with probability $1 - D(n - 1)$).

These are reflected in the following recursive equations:

$$
\begin{aligned}
P(1,1) &= 1 && (12)\\
P(1,k) &= P(1,k-1)D(1) \qquad k > 1 && (13)\\
P(n,k) &= P(n,k-1)D(n) + P(n-1,k-1)\overline{D}(n-1) \qquad n > 1;\ k \geq n && (14)
\end{aligned}
$$

where $\overline{x} \stackrel{\text{def}}{=} 1 - x$.

Note that for every $n \geq k_{max}$ $D(n) = 1 - d(\infty)$; this can be substituted into the third equation (for $n \geq k_{max}$) to simplify its structure.

Using these equations, one can easily derive $P(n, k)$ in a straightforward recursion. It is then straightforward to compute the expected value of $DIST_k$, $E[DIST_k]$.

For high values of k an approximate expression for $E[DIST_k]$ is given by:

$$E[DIST_k] \approx kd(\infty) \tag{15}$$

A proof of this approximation is next established.

Theorem 3 *When k approaches ∞, $E[DIST_k]$ approaches $kd(\infty)$.*

Proof: It is easy to see that $lim_{k\to\infty} P(1, k) = 0$; this convergence is at geometric rate. Similarly, and using induction on n one can show that for every $n \leq k_{max}$, $lim_{k\to\infty} P(n, k) = 0$. Thus, in calculating $E[DIST_k]$ we need to consider the values of $P(n, k)$ only for values of n obeying $n > k_{max}$. However for all these indices the recursive equations obey:

$$P(n, k) = P(n-1, k-1)d(\infty) \quad + \quad P(n, k-1)\overline{d}(\infty) \qquad n > 1;\ k \geq n \tag{16}$$

and thus for $k \to \infty$ we have

$$E[DIST_k] \quad = \quad E[DIST_{k-1}] + d(\infty) + \epsilon \tag{17}$$

where $\epsilon \to 0$. This implies that $lim_{k\to\infty} E[DIST_k] = kd(\infty)$. □

B Details of the Database Experiment

The DB2 laboratory experiments were conducted using an an IBM 4381 - T92 processor running DB2 Version 2.3, MVS/ESA 4.2, and IMS/ESA 3.1 as the Transaction Manager. The applications accessed 47 objects: 24 indexes, 19 tables, 3 catalog files and 1 DB2 workfile table. The total amount of data for the 43 objects (not including the workfile and catalogs) was approximately 1.5 gigabytes.

	Base 1 Pool	Model 3 Pools	% change
Elapsed run time (seconds)	1216	1240	
CPU utilization	89.7	91.8	
Transaction throughput (per second)	6.7	7.1	+6%
Physical I/O's per transaction	3.1	2.2	-29%
Synchronous (non-prefetch) I/O's per query	987	800	-19%
Average query elapsed time (seconds)	236	240	
Total number synchronous read I/O's	30602	23531	
Total number of pages physically read	285423	263825	

Table 1: Experimental Results

The workload for the experiments was a mix of transactions and complex queries. The transactions were 6 IMS applications accessing 26 objects, and the query applications were read-only and accessed 17 different objects. The application mix was such that there was a maximum of 7 concurrent application threads, 6 transaction and 1 query. The mix was designed so that CPU utilization during the base measurement was approximately 90%. The same mix was then used for subsequent runs. The priority of the transactions was higher than the query application. A base measurement was made with all DB2 objects assigned to a single 5000 buffer pool. Each buffer is 4K bytes and can hold 1 page. Several runs of the base measurement were made to verify repeatability. The results are summarized in Table 1, and interpreted in Section 6.2.

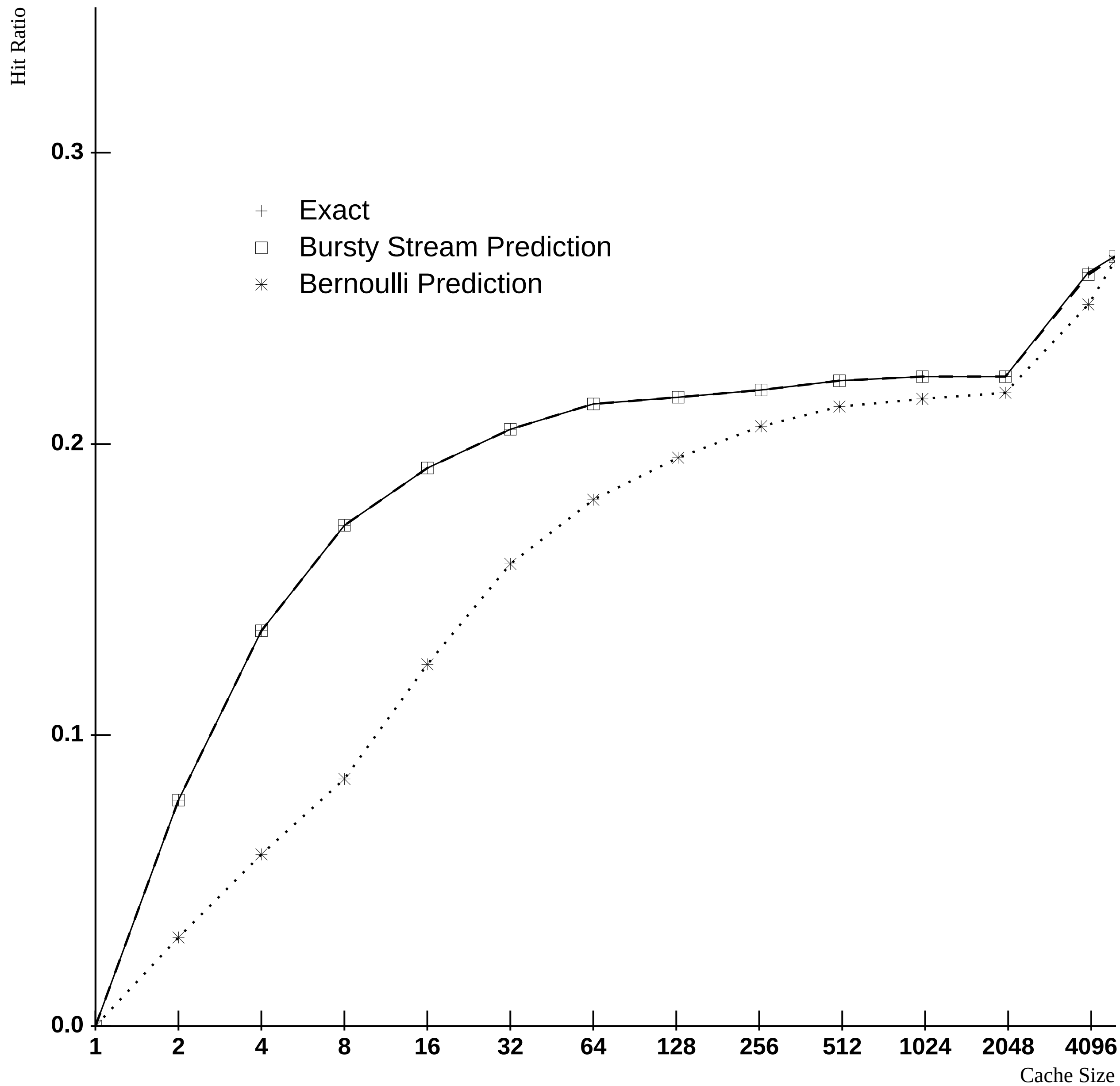

Figure 1: Hit ratio for superposed stream – very bursty streams, non-overlapping.

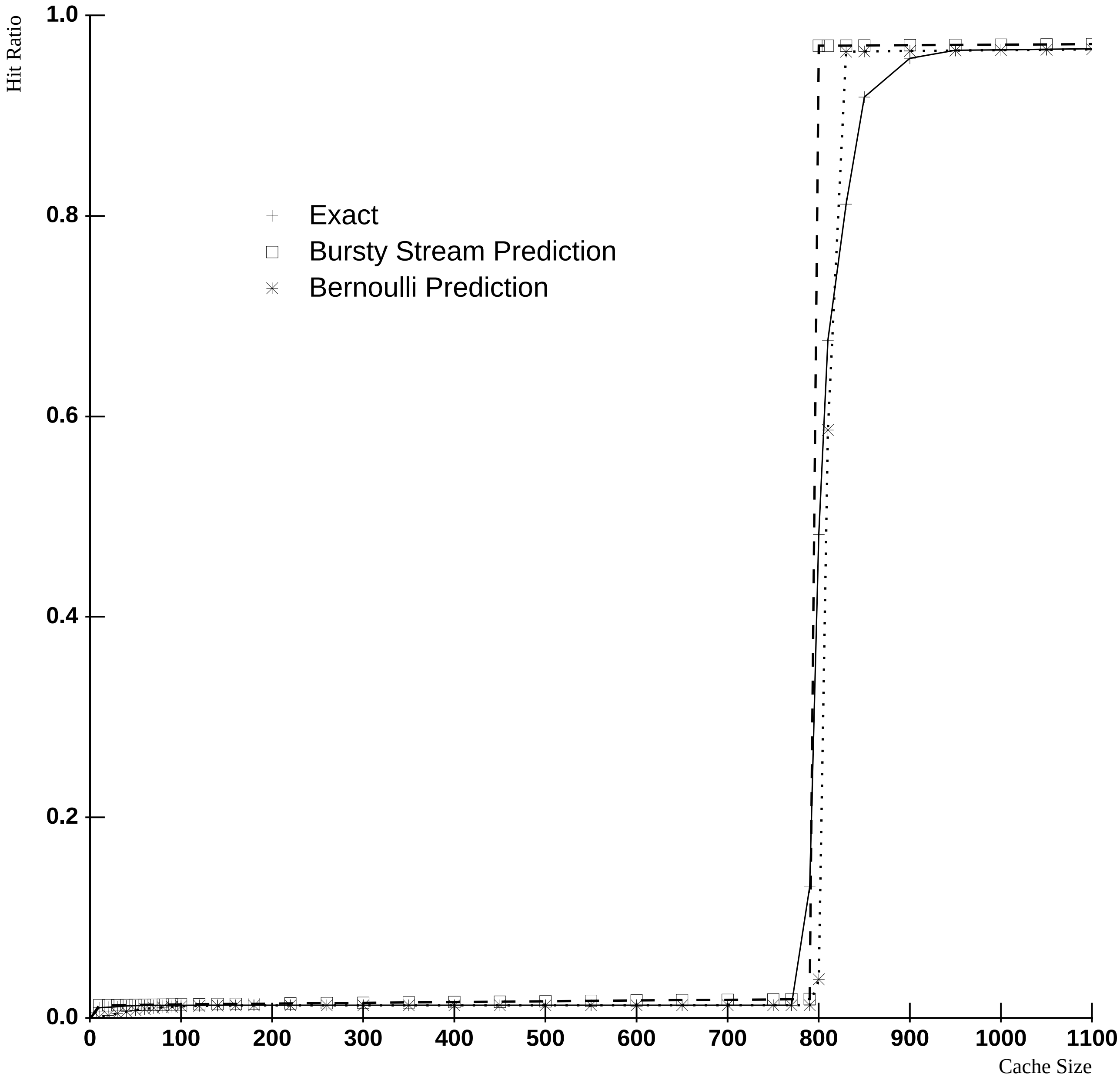

Figure 2: Hit ratio for superposed stream – very bursty streams, frequently alternating.

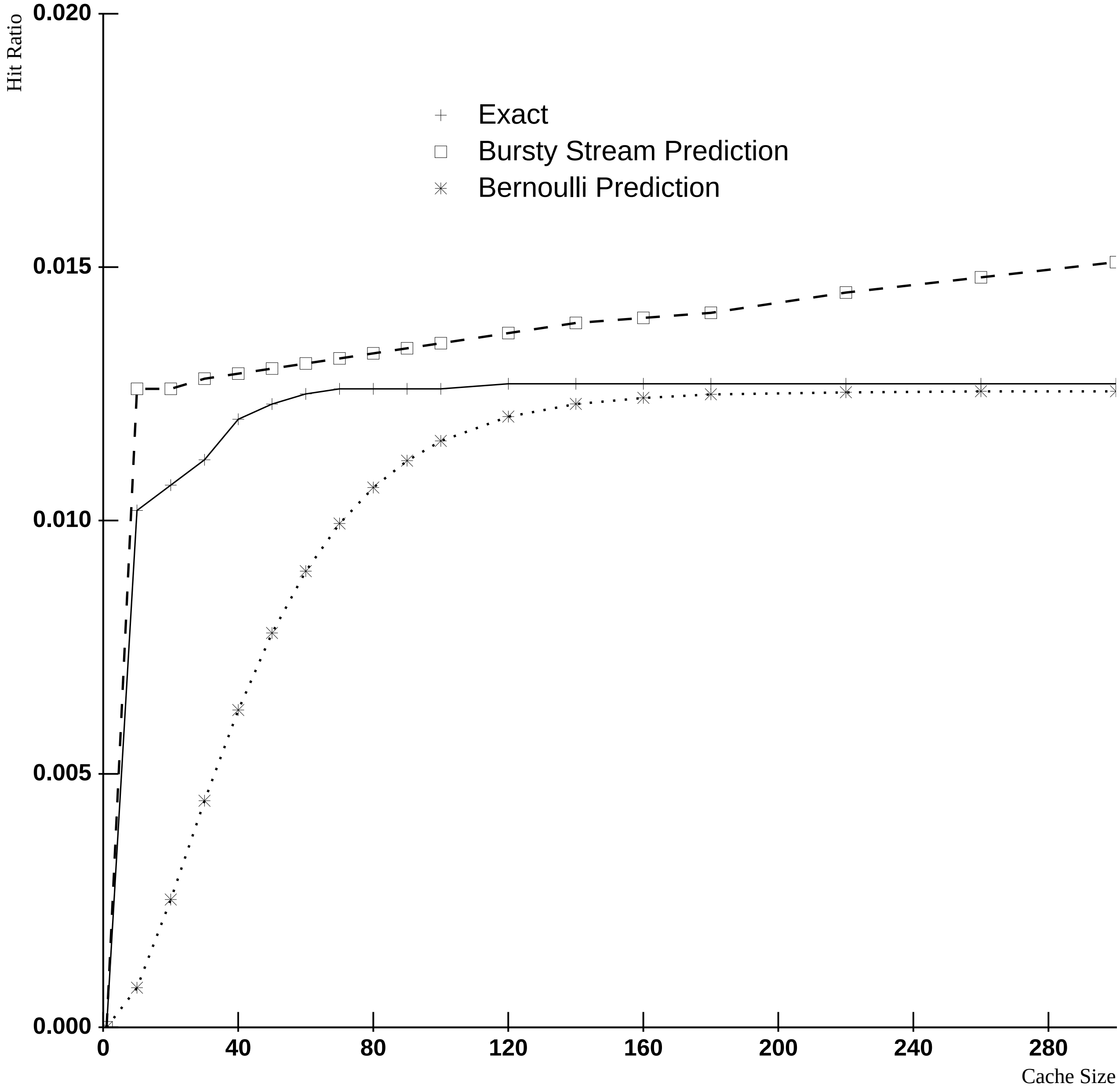

Figure 3: Closer Examination of Lower End of Figure 2: Hit ratio for superposed stream – very bursty streams, frequently alternating.

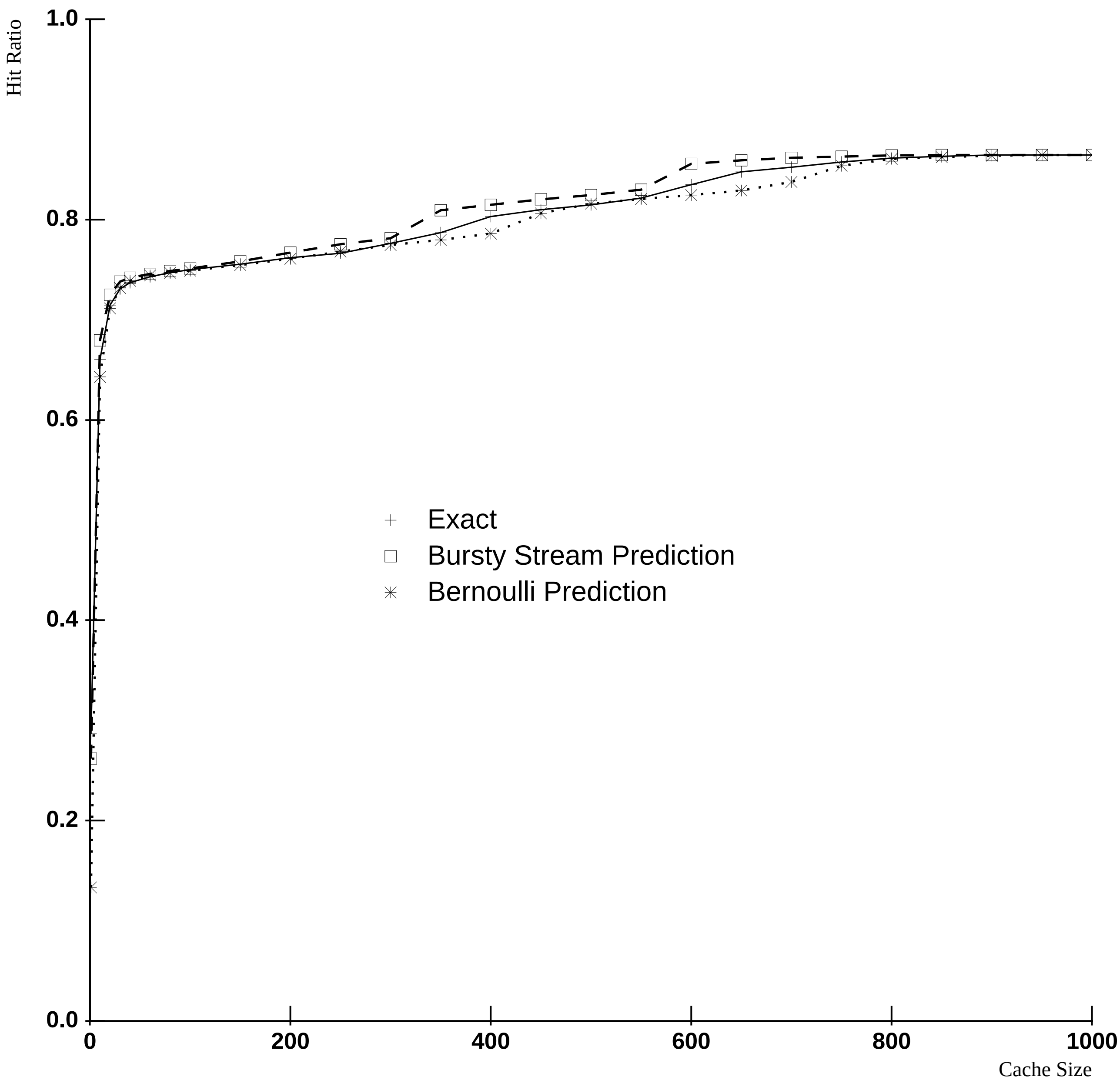

Figure 4: Hit ratio for superposed stream – less bursty streams.

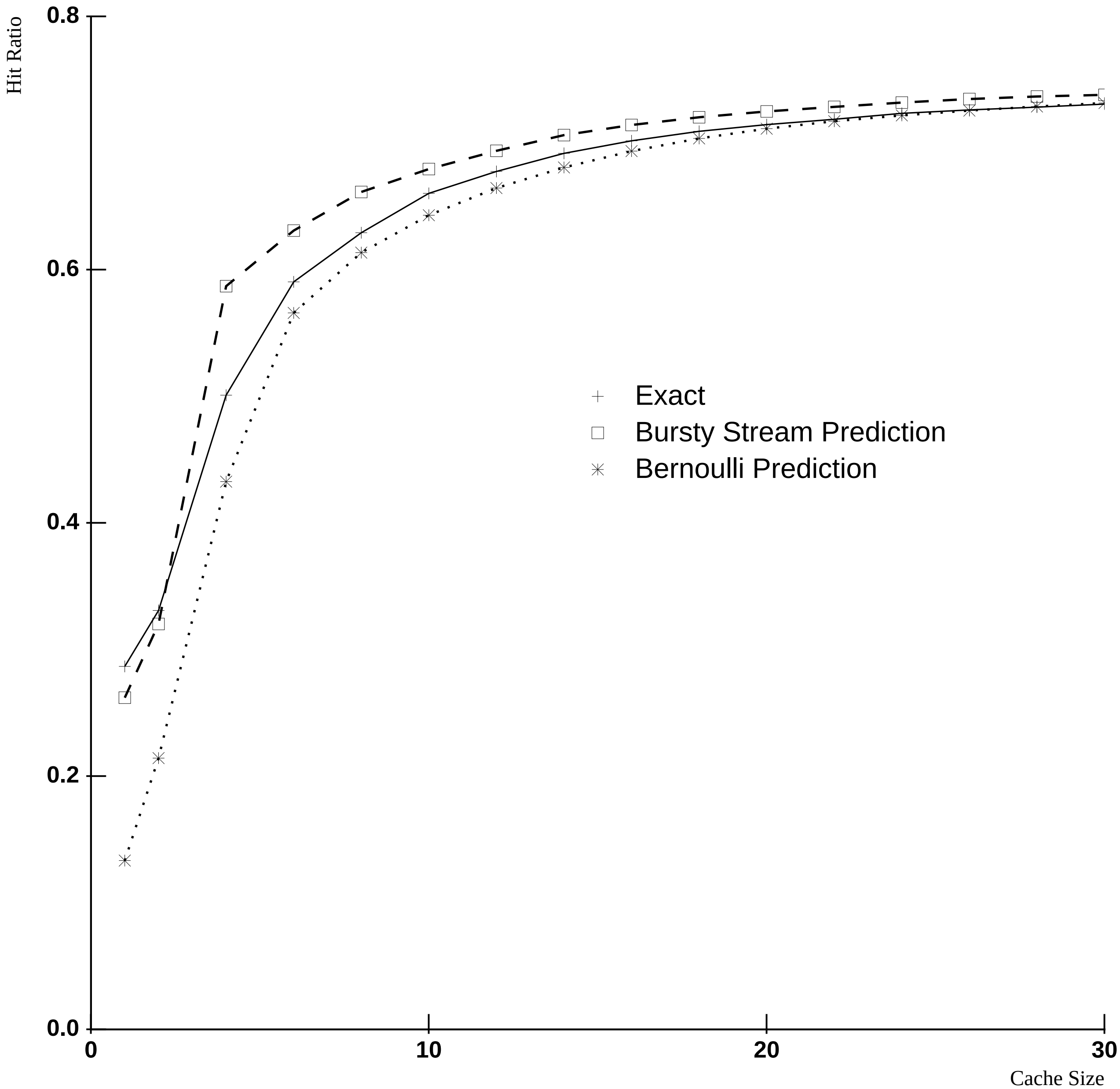

Figure 5: Closer Examination of Lower End of Figure 4: Hit ratio for superposed stream – less bursty streams.

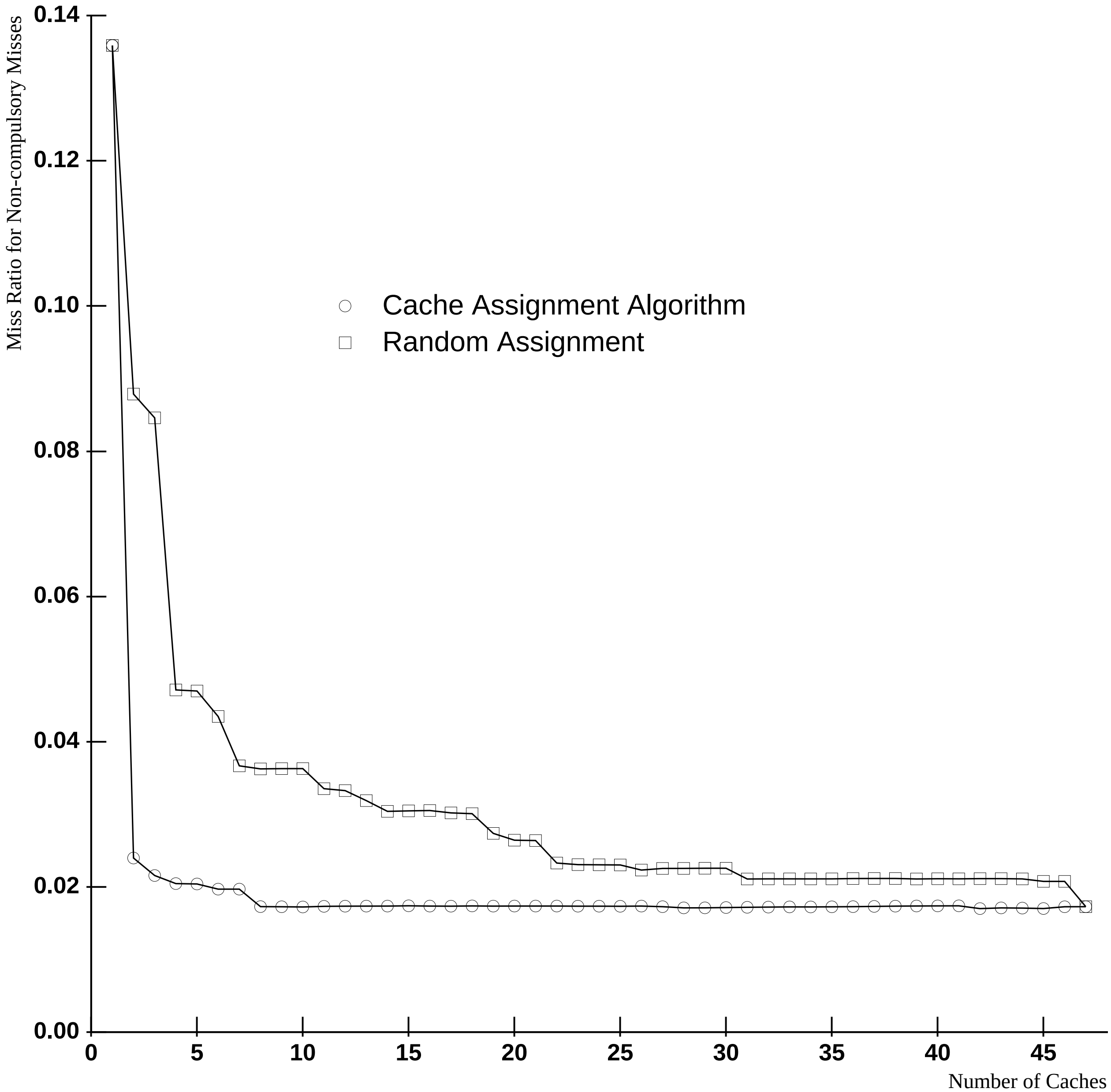

Figure 6: Miss Ratio of Cache Assignment Algorithm.