# Towards a Theory of Cost Management for Digital Libraries

A. Prasad Sistla
Univ. of Illinois, Chicago,
Ouri Wolfson
Univ. of Illinois, Chicago and NASA/CESDIS,
Yelena Yesha
NASA/CESDIS
and
Robert Sloan
Univ. of Illinois, Chicago

Wolfson's research was supported in part by NSF grants IRI-9408750 and IRI-9712967, DARPA grant N66001-97-2-8901, Army Research Labs grant DAAL01-96-2-0003, NATO grant CRG-960648.; Sistla's work was supported in part by NSF grant IRI-97-11925.

One of the features that distinguishes digital libraries from traditional databases is new cost models for client-access to intellectual property. Clients will pay for accessing data items in digital libraries, and we believe that optimizing these costs will be as important as optimizing performance in traditional databases. In this paper we discuss cost models and protocols for accessing digital libraries, with the objective of determining the minimum cost protocol for each model.

We expect that in the future information appliances will come equipped with a cost optimizer, in the same way that today computers come with a built-in operating system. This paper makes the initial steps towards a theory and practice of intellectual property cost management.

Categories and Subject Descriptors: H.2.m [**Database Management**]: Miscellaneous; H.3.5 [**Online Information Services**]: Commercial Services, Web based Services; H.3.7 [**Digital Libraries**]: Dissemination

General Terms: Cost Models, Protocols

Additional Key Words and Phrases: Average case analysis, caching, demand, on-line services, protocols, subscription, worst case analysis

Authors' addresses: A. Prasad Sistla , Ouri Wolfson and Robert Sloan, Electrical Engineering and Computer Science Department University of Illinois at Chicago, Chicago, Illinois 60607; Yelena Yesha, Center of Excellence in Space Data and Information Sciences at NASA, Goddard Space Flight Center, Greenbelt, MD.

## 1. INTRODUCTION

Digital libraries are repositories of data objects residing on servers, and accessed by clients through the Internet/WWW or other electronic networks. The server of an object stores the object, and automatically receives the object-updates. The clients read and/or subscribe to receive the updates to an object of interest from the server.

One of the features that distinguishes digital libraries from traditional databases is new cost models for client-access to intellectual property. Clients will pay for accessing data items in digital libraries, and we believe that optimizing these costs will be as important as optimizing performance in traditional databases. Indeed, in a recent workshop on strategic directions in computing research, cost management was identified as one of the six major grand challenges that have to be overcome in order to make digital libraries and electronic commerce widely accepted and used (see [N. Adam and et al. 1996]).

In this paper we address the problem of cost optimization for clients that access digital libraries. The cost optimization problem arises when a client has a choice of cost models and retrieval protocols. This choice may be available from a single vendor of information, or from multiple vendors that provide the same information. Thus, for example, a client may be able to make a choice between receiving the latest-price for a portfolio of stocks on Demand, or by Subscription. Furthermore, the Subscription may entitle the client, for a flat fee, to receive all the updates during a day; or the client may pay per update.

Our approach to cost optimization is to provide two basic cost models, and several complexity measures for each one of them. Then we analyze several algorithms according to these complexity measures.

We consider two cost models of information, the *request* cost model and the *time* cost model.

—In a request cost model, a client pays a read cost $rc$ for each read of an object submitted and satisfied by the server; if the client subscribes to receive the updates to the object, then the client pays a write cost $wc$ (which may be different than $rc$) for each propagated update. This is the cost model for newspapers and magazines, where the cost per copy is higher at the news-stand than for subscription delivery. One possible option that we consider in this cost model is Cache-invalidation; for an invalidation cost $ic$ which lower than $wc$, the server informs the client that the object was updated, but without providing the new value.

—In a time cost model the updates can be propagated from the server to the client using a time-based subscription. Namely, there is an initiation fee $if$, after which the client is entitled to receive from the server, for a flat fee $ff$, all the object-updates during a time period (say a month). This is the cost model for cable TV service. The client can still submit Demand requests to the server (Pay per view).

The two cost models lead to different problems and solutions in minimizing the cost, and thus to drastically different cost optimization algorithms. These algorithms use two basic static retrieval algorithms (or policies, or protocols) which are *Demand* and *Subscription*. We also introduce and analyze one hybrid static algorithm called *Divergence Caching*, and a set of dynamic algorithms called *Sliding-Window* algorithms. The above algorithms are defined as follows.

—In Demand the client propagates reads to the server, in response to which the server provides the latest version of the object. Cache-invalidation is an option that may be available with Demand; if so, when the object is updated the server notifies the client (without providing the new version). Thus the client can cache the object received from the server, and read the cached version until it receives an update notification.

—In Subscription the server propagates to the client each update of a cached object.

—Static Divergence Caching is a combination of Subscription and Demand. It takes advantage of reads that, in order to reduce access cost, can tolerate an out-of-date version of the object. The tolerance of a read is a number indicating the maximum divergence of the version returned in response to the read from the most up to date version. Divergence caching is based on a parameter called the *refresh rate*, which denotes the maximum divergence between the client and server versions of the object. Divergence caching uses Subscription to satisfy reads with a tolerance higher than the refresh rate, and Demand to satisfy the other reads. In this sense divergence caching is a combination of Subscription and Demand. Experimental results show that Divergence Caching reduces by at least half the expected cost of request compared to the optimum of Subscription and Demand (see section 4.4). We provide analytical and experimental evaluation of the Static Divergence Caching algorithm in the request cost model.

—The Sliding-Window algorithms are dynamic in the sense that they switch between Subscription and Demand. The principle behind the Sliding-Window algorithms is to predict the read-write distribution in the near future based on the recent past history. The recent past history is captured by the concept of a sliding window. The "incarnation" of this principle differs for the two cost models. In the request cost model, the sliding window is the latest $k$ requests, and this window is used to predict the probability of the next request being a write. The predicted value is used to decide between Subscription and Demand.
In the time cost model the sliding window is the latest $k$ time slots, and it is used to predict the expected number of reads in the subsequent time slots. Based on this prediction, using an algorithm *Opt* it is determined whether the next time slot should employ Subscription or Demand. The algorithm *Opt* is of independent interest. It can be used to allocate Subscription or Demand to each slot in a sequence of time-slots for which the expected numbers of reads are known; the allocation minimizes the total expected cost.
The Sliding-Window algorithms are also incorporated into Divergence Caching to obtain an algorithm that we call *Dynamic Divergence Caching*. In this algorithm the current sliding window is used to establish an optimal refresh rate, and this refresh rate may change dynamically as the distribution of reads and writes in the sliding window changes. Our experimental evaluation shows that dynamic divergence caching is superior to static divergence caching when the read-write

probabilities change dynamically.

We analyze the above algorithms using two types of complexity measures, namely *probabilistic* and *worst case*.

—For the probabilistic analysis we minimize the expected total cost under one of the following assumptions. For the request cost model we assume a probability $\theta$ for a request to be a write (i.e. an update), and a probability $1-\theta$ for it to be a read. For the time cost model we assume an expected number of requests per time unit.

—The worst case complexity measure considers performance of an algorithm for the worst case input. More specifically, we use the notion of competitiveness (see [A. Karlin and et al. 1988]) of online algorithms. An on-line algorithm is an algorithm that receives its input schedule one request at a time, and acts on each request before obtaining the next one; thus the algorithms that we consider in this paper are on-line. Roughly speaking, an on-line algorithm is competitive if, for every schedule $s$, its performance on $s$ is at most a constant times the performance of the optimal off-line algorithm on $s$ (in the above, performance may refer to any cost measure). We determine which retrieval algorithms are competitive. In general, the dynamic algorithms, i.e. ones that switch dynamically between Subscription and Demand, are competitive; whereas the static ones are not competitive. The worst-case analysis is appropriate for a chaotic read write pattern, i.e. one in which the pattern in the recent past is not indicative of the pattern in the near future.

The rest of the paper is organized as follows. In section 2 we analyze the Demand, Subscription, and Sliding Window algorithms in the request cost model. We derive formulas which establish the read and write costs and probabilities for which each one of the algorithms is optimal, i.e. has a lower expected cost than the others. We show that if the probabilities are unknown or they vary, then either the Sliding Window or Demand-with-cache-invalidation are optimal; we establish the costs for which each one of these two dominates the other. Our worst case analysis shows that, again, Sliding Window or Demand-with-cache-invalidation are competitive whereas the other algorithms are not competitive.

In section 3 we consider the time cost model. In this model Subscription has a flat fee (and possibly an initiation fee) per time slot, regardless of the number updates propagated to the client. We devise the algorithm *Opt* that selects for each time slot either Subscription or Demand, depending on the relative expected costs of the two. We also develop the Sliding Window variant for the time cost model.

In sections 2 and 3 we make the tacit assumption that each read must retrieve the latest version of the object. In section 4 we consider divergence caching in the request model. We show that the dynamic divergence caching algorithm is competitive, whereas the static algorithm is not. Experimental results for the static and dynamic divergence caching algorithms are also provided in this section. In section 5 we compare our work to relevant research, and in section 6 we provide a detailed summary of our results.

## 2. THE REQUEST COST MODEL

Consider some library object. The *relevant* requests for the object are writes that are issued by the server, and reads that are issued by the client. In other words, we ignore the reads at the server and the writes at the client since their cost is not affected by the retrieval algorithm. In this section we assume that reads are *nontolerant*, i.e. they require the latest version of the object. A *schedule* is a finite sequence of relevant requests to the data item $x$. For example, $w, r, r, r, w, r, w$ is a schedule. For the purpose of analysis we assume that the relevant requests are sequential. In practice they may occur concurrently, but then some concurrency control mechanism will serialize them.

In the request model, the cost of satisfying a read request sent from the client to the server is $rc$. The cost of propagating a write (or an update) of the object from the server to the client is $wc$.

Our analysis of protocols addresses both, the expected-case and the worst-case. The complexity measure for the worst-case is competitiveness, which was informally defined in the introduction, and is formally defined in subsection 2.2.3. The complexity measure for the expected case is the expected cost of a relevant request. To derive this expected cost we assume that the probability that a relevant request is a write is $\theta$ and the probability that a relevant request is a read is $1-\theta$.

Suppose that $A$ is a protocol, and the write-probability $\theta$ is fixed and known. Then we denote by $EXP_A(\theta)$ the *expected cost* of a relevant request. Suppose now that $\theta$ is unknown, or it varies over time, with equal probability of having any value between 0 and 1. Then we define a new complexity measure, the *average expected cost* per request, denoted $AVG_A$. It is the mean value of $EXP_A(\theta)$ for $\theta$ ranging from 0 to 1, namely

$$AVG_A = \int_0^1 EXP_A(\theta)d\theta \tag{1}$$

The average expected cost should be interpreted as follows. Suppose that time is subdivided into periods, where in the first period the probability that a relevant request is a write is $\theta_1$ and that it is a read is $1-\theta_1$; in the second period the probability that a relevant request is a write is $\theta_2$ and that it is a read is $1-\theta_2$; and so on. Suppose further that each $\theta_i$ has equal probability of having any value between 0 and 1. In other words, each $\theta_i$ is a random number between 0 and 1. Then, when using the protocol $A$, the expected cost of a relevant request over all the periods of time is the integral denoted $AVG_A$.

We analyze the Subscription, Demand (with and without cache invalidation) and the Weighted-Sliding-Window ($WSW$) protocols under the above complexity measures. The Demand with Cache Invalidation ($DCI$) protocol is basically the Demand protocol augmented such that consecutive reads from the same client (i.e. without intervening writes) can use the saved copy of the first read. To achieve this, the client may be in one of two states. It either has a copy of the object, or it does not have a copy. $DCI$ works as follows. If there is a copy of the object at the client, the client behaves as follows. A read (which returns the local copy) does not incur a cost, and it leaves the client in the same state. A write (at the server) causes an invalidation notification to be sent, incurring a cost of $ic$. In respose, the client discards its copy and changes to the 'no-copy' state. If there is no copy of the object at the client, then the client behaves as follows. A read request is propagated to the server. When the response is received, the client saves the copy of the object and it switches states. A write leaves the client in the same state, and it incurs no cost to the client (since it is not propagated to the client).

The Weighted-Sliding-Window ($WSW$) family of protocols are suggested by the need to dynamically change between Subscription and Demand based on the relative costs of executing the last $k$ requests under Demand and Subscription: if the cost of executing the last $k$ requests under Demand is lower than that of executing them under Subscription then we use Demand, otherwise we use Subscription. The following note indicates the rationale behind the above approach of selecting a policy for the next request.

*Note:.* Let $k_r$ be the number of reads among the last $k$ requests, and assume that the probability of the next request being a read is $k_r/k$. Then the expected cost of the next request under Demand (i.e. $rc \cdot k_r/k$ )is lower than that under Subscription (i.e. $wc \cdot (1 - k_r/k)$ ) if and only if the cost of executing the last $k$ requests under Demand (i.e. $rc \cdot k_r$) is lower than that of executing them under Subscription (i.e. $wc \cdot (k - k_r)$).

These protocols maintain a sliding window of the last $k$ requests, and make a decision for the next operation based on this window. The different protocols in the sliding window family differ based on the size of the window, $k$.

Subscription and Demand are *static* protocols, whereas the Demand-with-cache-invalidation and the sliding window protocols are *dynamic* in the sense that they dynamically switch between Subscription and Demand. The objective of our analysis of the protocols is to select the protocol that minimizes cost. If each information provider supports either Subscription or Demand but not both, then protocol selection implies provider selection. Thus a dynamic protocol may switch dynamically between providers.

### 2.1 The Weighted-Sliding-Window protocol description

The WSW protocol switches between Subscription and Demand retrieval of an object $x$. It does so by examining a window of the latest relevant read and write requests, and switching from Demand to Subscription if the cost of the reads in the window is higher than that of the writes, and from Subscription to Demand in the reverse case. Observe that on Demand the window needs to be examined only when a read is issued, and on Subscription it needs to be examined only when a write occurs. We denote by $WSW_k$ the protocol that uses a window of size $k$. In this section we describe a distributed and scalable implementation of the protocol $WSW_k$.

The simple case is when the client is on Subscription to $x$. Then all the reads issued at the client are satisfied locally, and all the writes issued at the server are propagated to the client; thus the client receives all the relevant requests. The window is tracked as a vector of $k$ bits (e.g. 0 represents a read and 1 represents a write). At the receipt of any relevant request, the client drops the oldest request in the window, and adds the current one. If there are $m$ reads and $n$ writes in the window, then it compares $m \cdot rc$ and $n \cdot wc$. If the first term is higher, then the client remains on Subscription, i.e. it simply waits for the next request. Otherwise it cancels the Subscription, and moves to Demand (i.e. sends every read to the server).

When the client is on Demand, then the server receives all the relevant requests, and can maintain the window for each client. Observe that different clients may have different windows, therefore, this solution will not scale as the number of clients increases. Thus we devised the following scheme, that enables the client to always maintain the window, i.e. even when the client is on Demand. In order to facilitate this, the server cooperates by maintaining a set $S$ of time-stamps of the latest $M$ writes on the object $x$, where $M$ is greater than or equal to $k$. Here $M$ is taken to be the maximum of the window sizes of all the clients accessing the object $x$. In its response to a read request, the server piggybacks the set $S$ on the object $x$ sent to the client. This set of values are used by the client to maintain the window during the Demand phase as described below.

The client maintains a sliding window of $k$ bits. This window reflects the latest $k$ requests preceding the last read that occurred during the current Demand phase; if no read occurred during the current Demand phase, then the window reflects the latest $k$ requests preceding the current Demand phase. Thus, the window does not capture write requests that may have occurred since the last read of the current Demand phase, since these writes are currently insignificant for the purpose of detecting when to switch to Subscription.

This window is maintained as follws. The client always remembers the timestamp of the last read in the current window. Whenever a new read is issued from the client, the server, with its response, piggybacks the set $S$ of time-stamps of write operations as described earlier. Using the current window, the timestamp of last read in the current window, the timestamp of the new read, and the set $S$, the client constructs a new window. The new window is constructed by adding to the current window the new read, and all the writes that occurred between the new read and the previous one (i.e. the last read in the current window).

Now we demonstrate by an example how the new window is constructed. Suppose that the size of the window is 5, and the current window is 10010 where 0 represents a read and 1 represents a write. Suppose further that the timestamp of the last read in the window is 7:01. Assume now that a new read is issued by the client at time 7:05, and the set $S$ in the response contains the timestamps: 6:58, 7:00, 7:02, 7:03, 7:04. Then the new window is 01110, where the timestamp of the last read in the new window is 7:05. The new window represents the reads at 7:01 and 7:05, and the three writes between them.

After constructing the new window, the client determines whether to switch to Subscription as before. Namely, if there are $m$ reads and $n$ writes in the new window, then the client compares $m \cdot rc$ and $n \cdot wc$. If the second term is higher, then the client remains on Demand, i.e. it does not take an action. Otherwise it initiates Subscription. This separate Subscription-initiation can be avoided. It can be incorporated into servicing reads, if the server can satisfy client read requests of the following type: "Send $x$ and $S$, and if the number of writes since time $t$ (a parameter of the request) is lower than $q$ (another parameter of the request), then initiate Subscription, i.e. propagate further writes. $t$ is the time of the last read request from the client; since the client has the latest $k$ timestamped relevant requests preceding time $t$, it can compute $q$, the number of writes below which the cost of reads in the window of latest $k$ relevant requests (preceding the current time) exceeds the cost of writes.

In summary, the client maintains a window $W$ of $k$ requests and it executes the following algorithm.

—If the client is on Subscription then it performs the following procedure: 1. When a write is propogated by the server, it is added to the window, and the oldest operation is deleted from the window. Denote by $m$ the number of reads in the window and by $n$ the number of writes in the window. If $m \cdot rc < n \cdot wc$, cancel the Subscription (and move to Demand). 2. When a read occurs, add it to the window and discard the oldest operation from the window.

—If the client is on Demand then it performs the following procedure whenever a read occurs: 1. Propogate the read request to the server. 2. When the server responds to a new read request, then construct the new window of the latest $k$ requests by using the old window, the set $S$ of write time-stamps, the timestamp of the new read, and the time-stamp of the previous read, i.e. the last read in the current window (or, if this last read occurred before the beginning of the Demand phase, then the timestamp of the beginning of the Demand phase is used). The current read becomes the latest request in the new window; save its timestamp. 3. Denote by $m$ the number of reads in the window and by $n$ the number of writes in the window. If $m \cdot rc \geq n \cdot wc$, then initiate Subscription.

The server executes the following algorithm.

—It maintains the set $S$ of timestamps of the latest $M$ writes.

—When a remote read is received from a client (the client must be on Demand, otherwise the read is satisfied locally) it piggybacks the set $S$ on the reply to the client.

In the following analysis we assume that $k > 1$, since $WSW_1$ is equivalent to Demand with cache invalidation.

### 2.2 Cost Analysis

In this section we analyze the expected cost, average expected cost and the worst case costs of the protocols Subscription (denoted S), Demand (denoted D), $DCI$ and $WSW_k$ (for $k > 1$).

The section is divided into 3 subsections. In the first subsection, we analyze and compare the expected costs of the protocols, and in the second subsection we analyze and compare the average (over all possible read-write ratios) of the expected costs of the protocols. Finally, in the third subsection we analyze the worst case performance of the protocols.

2.2.1 *Comparison of expected costs* . In this subsection we derive the expected costs of the protocols, and we show that for each $k$ and for each $\theta$, the $WSW_k$ protocol has a higher expected cost than one of the static protocols. Therefore, one of the protocols, D, S, and DCI has minimal cost. We determine the conditions for which each one of the protocols, D, S, and DCI is optimal.

In case of Subscription, read requests do not cost anything, whereas each write operation costs $wc$. Thus the expected cost of an operation for Subscription is equal to the probability that the operation is a write operation multiplied by $wc$, and thus is equal to $\theta \cdot wc$. In case of Demand, each read operation costs $rc$, whereas each write operation does not cost anything. Thus the expected cost of an operation for Demand is equal to $(1-\theta) \cdot rc$.

Now consider the $DCI$ protocol. The expected cost of an operation is the sum of the following: (i) (the probability that the client has a copy) times (the probability that the next request is a write) times $ic$; (ii) (the probability that the client does not have a copy) times (the probability that the next request is a read) times $rc$. Note that the probability that the client has a copy (resp. does not have a copy) is same as the probability that the previous request was a read request (resp. write request). Thus the expected cost of a request is $(1-\theta) \cdot \theta \cdot ic \,+\, \theta \cdot (1-\theta) \cdot rc$.

Thus,

$$EXP_D = (1-\theta) \cdot rc \qquad and \qquad EXP_S = \theta \cdot wc$$
$$and \qquad EXP_{DCI} = (1-\theta) \cdot \theta \cdot ic \,+\, \theta \cdot (1-\theta) \cdot rc \tag{2}$$

Now we derive the expected cost of the $WSW_k$ protocols. Consider the $WSW_k$ protocol for some given $k$, and consider any instant of time during the operation of the protocol. Let $l$ and $k-l$, respectively, be the number of read and write requests among the last $k$ requests. Then the probability that currently the client is on Subscription is equal to the probability that $l \cdot rc$ is greater than or equal to $(k-l) \cdot wc$, and this is same as the probability that $l \geq k \cdot wc/(rc+wc)$. This is equal to the probability that the number of writes among the last $k$ requests is less than or equal to $k \cdot rc/(rc+wc)$. Let $\alpha_k$ denote this probability. Also let $L = \lfloor k \cdot rc/(rc+wc) \rfloor$. Based on the Bernoulli trial, it can be shown that

$$\alpha_k = \sum_{j=0}^{L} \binom{k}{j} \cdot \theta^j \cdot (1-\theta)^{k-j} \tag{3}$$

The next theorem gives an expression for the expected cost of $WSW_k$ algorithm. Since the protocol switches between Subscription and Demand, this expression is equal to the sum of two subexpressions— the first one is the contribution of Subscription and the second is that of Demand.

THEOREM 1. *For every $k$ and for every $\theta$, the expected cost of the $WSW_k$ algorithm is*

$$EXP_{WSW_k}(\theta) = \theta \cdot \alpha_k \cdot wc + (1-\theta) \cdot (1-\alpha_k) \cdot rc \tag{4}$$

PROOF. Let us consider a single request, $q$. When the client is on Subscription, then the expected cost of $q$ is equal to $\theta \cdot wc$ (which is $wc$ times the probability that $q$ is a write operation). When the client is on Demand, the expected cost of $q$ is $(1-\theta) \cdot rc$. The actual expected cost of $q$ is the probability that the client is on Subscription times the expected cost of $q$ when the client is on Subscription, plus the probability that the client is on Demand times the expected cost of $q$ when the client is on Demand. Thus, we conclude the theorem.

□

The next theorem compares the expected costs of the $WSW_k$ and the static protocols.

THEOREM 2. *For every $k$ and every $\theta$, $EXP_{WSW_k} \geq \min\{EXP_S, EXP_D\}$*

PROOF. The following can be shown from equations 2 and 4.
$EXP_{WSW_k} - EXP_S = (1-\alpha_k)\cdot(rc\cdot(1-\theta) - wc\cdot\theta)$,
$EXP_{WSW_k} - EXP_D = \alpha_k\cdot(wc\cdot\theta - rc\cdot(1-\theta))$.
From the above two equations, we see that $EXP_{WSW_k} \geq EXP_S$ when $(rc\cdot(1-\theta) \geq wc\cdot\theta$, and $EXP_{WSW_k} \geq EXP_D$ when $rc\cdot(1-\theta) \leq wc\cdot\theta$. Putting the above observations together, we see that for every $k$ and every $\theta$, $EXP_{WSW_k} \geq \min\{EXP_S, EXP_D\}$. □

The above theorem indicates that the expected cost of the $WSW$ protocols is never lower than the minimum of the other protocols. In other words, either Subscription or Demand is better than the WSW protocol. Therefore, one of the protocols Demand, Subscription and DCI has minimum expected cost for any value of $\theta$. The following corollary specifies the exact ranges of $\theta$ for which each of the above three protocols has the minimum expected cost.

COROLLARY 1. *If $\theta > \max\{\frac{rc}{rc+wc}, \frac{rc}{rc+ic}\}$, then Demand has the minimum (among all the protocols) expected cost. If $\theta < \min\{\frac{rc}{rc+wc}, 1-\frac{wc}{rc+ic}\}$ then Subscription has the minimum expected cost. Otherwise, i.e. when $\theta$ lies between the above values, [1] Demand with Cache Invalidation has the minimum expected cost.*

PROOF. From equation 2 we see the following results.
(a) $EXP_D < EXP_S$ when $\theta\cdot(rc+wc) > rc$, i.e. when $\theta > \frac{rc}{rc+wc}$;
(b) $EXP_D < EXP_{DCI}$ when $rc < \theta\cdot(rc+ic)$, i.e. when $\theta > \frac{rc}{rc+ic}$.
From (a) and (b), we get that Demand has minimum expected cost when $\theta > \max\{\frac{rc}{rc+wc}, \frac{rc}{rc+ic}\}$. The other parts of the corollary are proved along similar lines. □

2.2.2 *Comparison of average expected costs* . In this subsection we derive the average expected costs the protocols, and we show that at least one of the dynamic protocols has a lower average expected cost than both static protocols. We also show how to determine which dynamic protocol has the lowest average expected cost for a given value of the parameters $ic$, $rc$, and $wc$.

By equations 1 and 2 we obtain in a straightforward fashion that:

$$AVG_D = \int_0^1 EXP_D d\theta = \frac{rc}{2}\ , \qquad AVG_S = \int_0^1 EXP_S d\theta = \frac{wc}{2}\ ,$$
$$AVG_{DCI} = \int_0^1 EXP_{DCI} d\theta = \frac{ic+rc}{6} \qquad (5)$$

Note that, roughly speaking, the average expect cost is computed by assuming that $\theta$ is equally liekly to take any value between 0 and 1; this means that on the average $\theta = \frac{1}{2}$, and since on Demand writes do not incur a cost, the average expected cost of a request in this case (i.e. $AVG_D$) is $\frac{rc}{2}$. A similar intuition applies for the expression $AVG_S$. The expression for $AVG_{DCI}$ is obtained by observing that $\int_0^1 \theta\cdot(1-\theta)d\theta = \frac{1}{6}$.

The following technical theorem gives the expression for $AVG_{WSW_k}$.

[1]Observe that it must be the case that $\max\{\frac{rc}{rc+wc}, \frac{rc}{rc+ic}\} \geq \min\{\frac{rc}{rc+wc}, 1-\frac{wc}{rc+ic}\}$.

THEOREM 3. *For the $WSW_k$ protocol, the average expected cost per request is given by the following equation:*

$$AVG_{WSW_k} = \frac{wc(L+1)(L+2)\ +\ rc(k-L)(k-L+1)}{2(k+1)(k+2)} \tag{6}$$

*(recall that $L = \lfloor k \cdot rc/(rc+wc) \rfloor$).*

PROOF. The derivation of the above equation is obtained as follows. First, from equation 4 we see that

$$EXP_{WSW_k}(\theta) = (1-\theta) \cdot rc - \alpha_k \cdot rc + (wc + rc) \cdot \theta \cdot \alpha_k \tag{7}$$

From the above equation and using $AVG_{WSW_k} = \int_0^1 EXP_{WSW_k} d\theta$, we get the following equation after some simplification.

$$AVG_{WSW_k} = \frac{rc}{2} + (wc + rc) \cdot \int_0^1 \alpha_k \cdot \theta d\theta - rc \cdot \int_0^1 \alpha_k d\theta \tag{8}$$

Using the equation 3, we see that

$$\int_0^1 \alpha_k d\theta = \sum_{j=0}^{L} \binom{k}{j} \cdot \int_0^1 \theta^j \cdot (1-\theta)^{k-j} d\theta \tag{9}$$

Using the following identity, for non-negative integers $a$ and $b$,

$$\int_0^1 x^a \cdot (1-x)^b dx = \frac{a! \cdot b!}{(a+b+1)!} \tag{10}$$

and after some simplification, we get

$$\int_0^1 \alpha_k d\theta = \frac{L+1}{k+1} \tag{11}$$

Similarly, we can show that

$$\int_0^1 \alpha_k \cdot \theta d\theta = \frac{(L+1)(L+2)}{2(k+1)(k+2)} \tag{12}$$

Using equations 11 and 12, and substituting for $\int_0^1 \alpha_k d\theta$ and $\int_0^1 \alpha_k \cdot \theta d\theta$ in equation 8, we get 6 after some simplification. □

Remember that the $WSW_k$ protocol operates as follows: if there are too few writes in the window, more specifically less than or equal to $L$, then the client will be put on Subscription for the next request; if there are too few reads in the window, more specifically less than or equal to $k - L - 1$, then the client will be put on Demand for the next request. In this sense the above expression is symmetric with respect to read/write costs.

Thus, we have now obtained the expressions for the average expected costs of all the protocols. For the rest of this subsection we will concentrate on comparing these costs.

The next theorem shows that $AVG_{WSW_k}$ converges as $k$ goes to $\infty$. Furthermore, the theorem indicates that starting from a certain value of $k$, $AVG_{WSW_k}$ is bigger than its value in the limit.

THEOREM 4. $\lim_{k\to\infty} AVG_{WSW_k} = \frac{wc \cdot rc}{2(wc+rc)}$. *Furthermore, for all* $k \geq \frac{1}{4}(\frac{wc}{rc} + \frac{rc}{wc} - 6)$, $AVG_{WSW_k}$ *is bigger than* $\frac{wc \cdot rc}{2(wc+rc)}$.

PROOF. We modify equation 6 as follows. Let us define $l$ to be $\frac{rc}{wc+rc}$. We first observe that $L = \lfloor k \cdot l \rfloor$ and hence $L$ can be written as $k \cdot l - \delta$ where $0 \leq \delta < 1$, more precisely $\delta = k \cdot l - L$. Replacing $L$ by $k \cdot l - \delta$ in equation 6, after a series of simplifications we get the following equation:

$$AVG_{WSW_k} = \frac{wc \cdot l}{2} + X + Y \tag{13}$$

where

$$X = \frac{k \cdot wc \cdot l}{2(k+1)(k+2)} + \frac{2wc^2}{2(wc+rc)(k+1)(k+2)} \tag{14}$$

and

$$Y = \frac{\delta(rc - 3wc) + \delta^2(wc+rc)}{2(k+1)(k+2)}. \tag{15}$$

From the above equation we see that the limits of $X$ and $Y$ both go to zero as $k$ goes to $\infty$. Hence, we get $\lim_{k\to\infty} AVG_{WSW_k} = \frac{wc \cdot l}{2} = \frac{wc \cdot rc}{2(wc+rc)}$.

To prove the second part of the theorem, we proceed as follows. First, observe that in the above equation it is possible for $Y$ (and hence $X + Y$) to become negative; this may cause $AVG_{WSW_k}$ to be less than the asymptotic value for some $k$. However, when $k$ is greater than or equal to some value, $X + Y$ will be positive. This happens when $X + Y \geq 0$, i.e. when

$$k \cdot wc \cdot l + \frac{2wc^2}{(wc+rc)} + \delta(rc - 3wc) + \delta^2(wc+rc) \geq 0. \tag{16}$$

If $rc > 3wc$ then all the terms on the left hand side of the above inequality are positive and hence the inequality is satisfied for all $k \geq 0$. Now assume that $rc < 3wc$. Now consider the sum $\delta(rc - 3wc) + \delta^2(wc + rc)$ in the above inequality. It can be shown that the minimum value of this sum is $-\frac{(3wc-rc)^2}{4(rc+wc}$ (to see this, take the derivative of the sum with respect to $\delta$; this derivative is zero when $\delta = \frac{(3wc-rc)}{2(rc+wc)}$; substituting this $\delta$ in the sum we get the above minimum value). Substituting this minimum value for $\delta(rc - 3wc) + \delta^2(wc + rc)$ in the inequality 16, and after some simplification, we see that $X + Y \geq 0$ when $k \geq \frac{1}{4}(\frac{wc}{rc} + \frac{rc}{wc} - 6)$. This proves the second part of the theorem. □

The last theorem, combined with equation 5, indicates that for each $rc$ and $wc$ there exists a value $k_0$, such that for all $k \geq k_0$ $AVG_{WSW_k}$ is smaller than both $AVG_S$ and $AVG_D$. (The reason is that the asymptotic value of $AVG_{WSW_k}$, which is $\frac{wc \cdot rc}{2(wc+rc)}$, is smaller than both $AVG_S$ which is $wc/2$ and $AVG_D$ which is $rc/2$). In other words, Subscription and Demand are suboptimal in terms of average expected cost. Thus, the minimum average expected cost is obtained by a dynamic protocol, i.e. $DCI$ or $WSW_k$. The last theorem also indicates that if there is a window size $k$ for which $AVG_{WSW_k}$ is smaller than the value in the limit (i.e. $\frac{wc \cdot rc}{2(wc+rc)}$), then this $k$ is smaller than $\frac{1}{4}(\frac{wc}{rc} + \frac{rc}{wc} - 6)$.

Therefore, the protocol with the lowest average expected cost can be selected by the following procedure. Compute the values for

(1) $AVG_{DCI}$ (using equation 5),
(2) $AVG_{WSW_k}$ for each $k < \frac{1}{4}(\frac{wc}{rc} + \frac{rc}{wc} - 6)$, (using equation 6),
(3) $\frac{wc \cdot rc}{2(wc+rc)}$.

If (1) is the smallest, then select DCI. If some value in (2) is the smallest, then select $WSW_k$ for the window size $k$ that has the minimum $AVG_{WSW_k}$. If (3) is the smallest, then select $WSW_k$ with a window size that is arbitrarily close to $\frac{wc \cdot rc}{2(wc+rc)}$. For any given $\epsilon$, a window size for which $AVG_{WSW_k}$ is within $\epsilon$ of $\frac{wc \cdot rc}{2(wc+rc)}$ can be found using equation 13 given in the appendix (the value of $k$ is chosen by setting $X + Y < \epsilon$).

In some cases the above procedure can be bypassed. For example, it can be shown that if $rc \geq 3wc$, then $AVG_{WSW_k}$ is smaller than $AVG_{DCI}$ for every window size. It can also be shown that if $ic \leq \frac{(6rc \cdot wc - 3rc^2 - wc^2)}{4(wc+rc)}$ and $1 - \frac{\sqrt{6}}{3} \leq \frac{rc}{wc} \leq 1 + \frac{\sqrt{6}}{3}$ then $AVG_{DCI} \leq AVG_{WSW_k}$ for all $k \geq 2$. For brevity, the proof of these statements are omitted.

2.2.3 *Worst case analysis.* In this subsection we show that the static algorithms, $S$ and $D$, perform poorly in the worst case, whereas the dynamic algorithms DCI and WSW (i.e. the ones that switch dynamically between Subscription and Demand) perform well.

We take competitiveness as the worst-case performance measure of a protocol. We use the notion of competitiveness as defined in [Motwani and Raghavan 1991]. Formally, a c-competitive protocol $A$ is defined as follows. Suppose that $M$ is the perfect protocol that has complete knowledge of all the past and future requests. Protocol $A$ is *c-competitive* if there exist two finite numbers $c$ ($\geq 1$), and $b$ ($\geq 0$), such that for any schedule $\psi$, $COST_A(\psi) \leq c \cdot COST_M(\psi) + b$. We call $c$ the *competitiveness factor* of the protocol $A$. Competitiveness bounds the worst-case cost of protocol to be within a constant factor of the minimum cost. We say a protocol $A$ is *tightly c-competitive* if $A$ is $c$-competitive, and for any number $d < c$, $A$ is not $d$-competitive.

First, let's consider the two static protocols. For Demand, we can pick a long schedule which consists of only reads. Then the cost of the Demand protocol is un-boundedly higher than the cost of the optimal protocol on this schedule (which is 0 if we keep a copy at the client).

For Subscription, we can also pick a long schedule which consists of only writes. Then the cost of the Subscription protocol on this schedule is also un-boundedly higher than the optimal cost (which is 0 if we do not keep a copy at the client). Therefore, the static algorithms Demand and Subscription are not competitive.

THEOREM 5. *The sliding-window algorithm* $WSW_k$ *is tightly* $(\frac{\epsilon}{\min\{wc,rc\}})$-*competitive where* $\epsilon = (k+1)wc + \lceil \frac{k \cdot wc}{rc+wc} \rceil (rc - wc)$.

PROOF. Let $\epsilon$ be as given in the statement of the theorem. Let $\psi$ be a schedule of requests. Consider the subsequence $S$ of $\psi$ consisting of read requests, each of which occurs immediately after a write request. Let $N_\psi$ be the number of reads in $S$. The cost of an optimal off-line algorithm on the schedule $\psi$ is $N_\psi \cdot \min\{wc, rc\}$, since for each read $r$ in $S$ either the immediately preceding write is sent to the client, or $r$ is sent to the server; all other requests incur a zero cost. We prove the theorem by showing (1) $COST_{WSW_k}(\psi) \leq N_\psi \cdot \epsilon + c$ where $c$ is a constant, and (2) there exists a schedule $\psi_0$ for which $COST_{WSW_k}(\psi_0) = N_{\psi_0} \cdot \epsilon + c$. It follows that $WSW_k$ is tightly $(\frac{\epsilon}{\min\{rc,wc\}})$-competitive.

Now we prove the first postulate (1) above. We divide the schedule $\psi$ into maximal blocks consisting of similar requests. Formally, let $B_1, B_2, ..., B_r$ be the division of $\psi$ into blocks such that the requests in any block are all reads or they are all writes, and successive blocks have different requests. It is easy to see that the total number of read blocks in $\psi$, i.e. blocks that only contain read requests, is either $N_\psi$ or $(N_\psi + 1)$. Similarly, the total number of write blocks in $\psi$ is either $N_\psi$ or $(N_\psi + 1)$. Now, we analyze the cost of read and write requests separately. Consider any read block $B_i$. Let $L = \lceil \frac{k \cdot wc}{rc+wc} \rceil$. It should be easy to see that only the first $L$ reads in $B_i$ may each incur a cost of $rc$. After the first $L$ reads the algorithm will definitely be switching to Subscription and further reads in the block will not incur any cost. Thus the cost of executing all the reads in $B_i$ is bounded by $L \cdot rc$. Hence the cost of all the reads in $\psi$ is bounded above by $(N_\psi + 1) \cdot L \cdot rc$. By a similar argument, it can be shown that the cost of all the writes in a write block is bounded by $(k + 1 - L)wc$. As a consequence, the cost of all the writes in $\psi$ is bounded by $(N_\psi + 1) \cdot (k + 1 - L) \cdot wc$. Hence, $COST_{WSW_k}(\psi) \leq (N_\psi + 1) \cdot (L \cdot rc + (k + 1 - L) \cdot wc)$. Substituting for $L$ and after some simplification we get $COST_{WSW_k}(\psi) \leq N_\psi \cdot \epsilon + c$ for an appropriate constant $c$.

To show that the above bound is tight (postulate (2) above), consider a schedule $\psi_0$ that starts with a block that contains $k$ read requests, and in which each subsequent block is also of length $k$. It should be easy to see that $COST_{WSW_k}(\psi_0) = N_\psi \epsilon + c'$ for an appropriate constant $c'$. □

THEOREM 6. *The algorithm DCI (i.e. Demand with Cache Invalidation) is tightly* $(\frac{rc+ic}{\min\{rc,wc\}})$*-competitive.*

PROOF. Similarly to the proof of theorem 5, we let $N_\psi$ be the number of reads in $\psi$ that occur immediately after a write, where $\psi$ is an arbitrary schedule of requests. It is easy to see that $N_\psi \cdot min(rc + wc)$ is the minimum cost to satisfy all the requests in $\psi$ by an off-line algorithm. Let $B_1, B_2, ..., B_r$ be the division of $\psi$ into blocks such that the requests in any block are all reads or they are all writes, and successive blocks have different requests.

As in the proof of theorem 5, the total number of read blocks in $\psi$ is less than or equal to $(N_\psi + 1)$, and a read block costs at most $rc$ since after the first read the client will have a copy and will be on Subscription. Thus the total cost of reads is bounded by $(N_\psi + 1) \cdot rc$. Similarly, the total number of write blocks in $\psi$ is less than or equal to $(N_\psi + 1)$. A write block costs only $ic$ since, at the first write in the block the server will send an invalidate message and during the remainder of the block the client will have a copy of the object. Thus, the total cost of writes in $\psi$ is bounded by $(N_\psi + 1) \cdot ic$, and $COST_{DCI}(\psi) \leq N_\psi(rc + ic) + (rc + ic)$.

To show that the above bound is tight, consider a schedule $\psi_0$ that starts with a read request, and in which each subsequent block is also of length 1. It should be easy to see that $COST_{DCI}(\psi_0) = (rc + ic) \cdot N_{\psi_0} + c$ where $c$ is some constant. □

## 3. THE TIME COST MODEL

In this section, we consider the time cost model. In this model, if the client chooses subscription then he/she will be charged an initiation fee ($if$), and thereafter the client will be charged a flat fee ($ff$) for each time slot. During this time, every write to the data item will be propagated to the client. On the other hand, if the client chooses Demand then each read request will be charged $rc$. It is possible for the client to change the access policy dynamically after each time slot. However, the client will be charged the initiation fee whenever he/she switches from Demand to Subscription; a switch from Subscription to Demand will not incur an initiation fee. The complexity measure that we propose for this cost model is the *average cost* per time slot (rather than the cost per request as in the previous section).

Note that if we choose the Subscription policy for a time slot, then the cost incurred for that time slot is fixed irrespective of the number of write requests or read requests that occur during that slot. On the other hand, if we chose Demand for a time slot then the access cost for that slot will be proportional to the number of read requests in that slot. Thus, the number of writes in each time slot has no bearing on the cost when we choose either Subscription or Demand. The optimal strategy that we devise will switch between Demand and Subscription so as to minimize the expected access cost over a given period of time.

In the time cost model it is insufficient to assume a probability for each one of the two types of relevant requests, as we did in the previous section. These probabilities do not provide an indication as to the number of requests per time slot. Thus, in the first subsection we assume an expected read-pattern, i.e. an expected number of read requests for each time slot. We provide an algorithm that assigns Subscription or Demand to each slot. In the second subsection we assume that the expected number of reads per time slot is unknown. However, the read pattern is estimated, i.e. the expected number of reads for a time slot can be estimated based on the number of reads in a sliding window consisting of the latest $k$ slots. Then we consider the equivalent of the Sliding Window algorithm for the time cost model. It assigns Subscription or Demand to time slots for estimated read patterns. In the third subsection we extend the results of the first two subsections to the case where Demand is available with cache invalidation.

### 3.1 Expected read pattern

In this subsection we assume that we are given a finite sequence of time slots and we are given a read-pattern, i.e. the expected number of read requests in each time slot. We present an algorithm, called *Opt*, for selecting the optimal retrieval protocol in each time slot. The objective of the *Opt* algorithm is to produce an *allocation pattern* i.e. assign Subscription or Demand to each time slot. This determines the retrieval protocol used for the slot. We will show that for the given read pattern, the allocation pattern gives the minimum total cost over the sequence of time slots. Consequently, the average cost per time slot is minimized.

The algorithm *Opt* operates as follows. It initially computes the demand fees $df_i$ for each slot $i$. Here, $df_i$ is computed as the product of the number of reads ($nr_i$) in the $i^{th}$ slot and the demand access cost for a single read request ($rc$). After this, it examines the slots one after another. After examining each slot, it either

$$\underbrace{\underline{SDD\cdots S}}_{Decided\ Slots}\ \underbrace{\overbrace{-+\cdots+}^{cs}}_{Undecided\ Slots}\ \underbrace{2r\ \mid 5r\ \mid \cdots}_{Unexamined\ Slots}$$

Fig. 1. Classification Of Slots

immediately decides a Retrieval Protocol for the slot, or it keeps the decision for the slot pending until it examines sufficient number of future slots. Thus, at any point in the execution of the algorithm, the sequence of time slots can be divided in to three types, (1) the sequence of *Decided* slots which occur in the beginning, followed by (2) the sequence of *Undecided* slots, i.e. slots that have been examined but for which a decision is pending, followed by (3) the sequence of *Unexamined* slots. The Decided slots are the slots for which Subscription or Demand protocol has been assigned. The algorithm uses two variables *status* and $FUS$ (first undecided slot). At any time the *status* variable denotes the retrieval protocol assigned to the last decided slot, while $FUS$ gives the index of the first undecided slot. After the termination of the algorithm, the array variable *decision* contains the decisions that have been taken for each of the slots. Figure 3 shows the classification of the different slots.

Initially, the *status* variable is assigned Demand. The algorithm *Opt* scans the slots one by one, starting from the first. For each time slot $i$, it computes $(df_i - ff)$, where $df_i$ is the demand fee of the $i^{th}$ slot as described above, and $ff$ is the flat fee per time unit. Assume that the value of the status variable is Demand. If the demand fee $(df_i)$ is less than or equal to flat fee $(ff)$, i.e. $(df_i - ff) \leq 0$, the algorithm *Opt* allocates Demand to the current $(i^{th})$ slot. If the demand fee is greater than the flat fee, i.e. $(df_i - ff) > 0$ , it means that Subscription is cheaper than Demand. However, we cannot assign Subscription whenever this condition is satisfied, because Subscription also involves the payment of the initiation fee $(if)$. So, we check whether the demand fee $(df_i)$ is greater than the flat fee $(ff)$ by $if$, i.e. $df_i - ff \geq if$. However, if the condition is not satisfied for a single slot, we cannot assign Demand to that slot. This is because it may be possible to distribute or amortize the initiation fees over a sequence of time slots so that Subscription is still better than Demand. More specifically, the cumulative sum of $(df_i - ff)$ for a sequence of consecutive slots may be greater than $if$, i.e. $\sum(df_i - ff) \geq if$, although for each individual slot $i$ the difference $(df_i - ff)$ may be positive or negative. So, algorithm *Opt* proceeds to the next slot and computes the cumulative sum $(cs)$, $cs = \sum(df_i - ff)$.

In summary, when the value of the *status* variable is Demand (i.e. the retrieval protocol assigned to the last decided slot is Demand), the Demand protocol is assigned to all the Undecided slots, i.e. from the First Undecided Slot to the current $(i^{th})$ slot, whenever $cs \leq 0$. If $cs \geq if$, then Subscription is assigned to all the Undecided slots and the status variable becomes Subscription. If neither of the above conditions is satisfied, the algorithm proceeds to the next slot and computes the cumulative sum and repeats the above operations.

If the status is Subscription, it means that the initiation fee has been paid, and we can allocate Subscription to all the Undecided slots whenever $cs \geq 0$. If $cs < 0$, it means that Demand is cheaper for the Undecided slots. However, we cannot choose Demand if $\mid cs \mid < if$, because if we have Subscription in an Unexamined slot, we would be paying more in switching back to Subscription. Thus the algorithm *Opt* checks if $\mid cs \mid \geq if$, i.e. it checks if initiation fee is recoverable. If so, the algorithm chooses Demand for the Undecided slots.

In summary, when the status is Subscription, the Subscription protocol is allocated to all the Undecided slots whenever $cs \geq 0$. If $cs < 0$ and $\mid cs \mid \geq if$, then Demand is assigned to all the Undecided slots and the status becomes Demand. If neither of the above conditions is satisfied, the algorithm *Opt* proceeds to the next slot and computes the $cs$ and repeats the above operations.

If we are unable to arrive at a decision based on the cumulative sum and we reach the end of the pattern, there are two possible cases, $cs < 0$ and $0 < cs < if$. In either case, the algorithm *Opt* chooses Demand for all the Undecided slots. This is justified because, if $cs < 0$, Demand is cheaper and if $0 < cs < if$, it means that the initiation fee is not recoverable.

A formal presentation of the algorithm is given below. In this algorithm, $n$ is the total number of time slots over which we are performing the optimization.

ALGORITHM 1. *Algorithm Opt*

```
{ For i = 1 to n df_i = nr_i · rc
i:=1
last:=n
status:=Demand
while i ≤ last
    { cs := 0
    FUS:=i
    If status=Demand
        { repeat
            cs := cs + (df_i − ff)
            i := i + 1
        until ((cs ≤ 0) ⋁ (cs ≥ if) ⋁ (i > last))
        if (cs ≥ if)
        status:=Subscription
        }
    else
        { repeat
            cs := cs + (df_i − ff)
            i := i + 1
        until ((cs ≥ 0) ⋁ (cs ≥ −if) ⋁ (i > last))
        if ((cs ≤ −if) ⋁ (i > last))
        status:=Demand
        }
    for j = FUS up to i − 1
    decision[j]=status
    }
}
```

Now, we prove the optimality of the above algorithm. For this, we need the following definitions. We assume that the number of time slots $n$, and the expected number of reads ($nr_i$) in each time slot $i$ (for $i = 1, ..., n$) are given. As indicated above, we define an allocation pattern $p$ to be a mapping that associates a policy $p(i)$, which is either Demand or Subscription, with each time slot $i$. For the allocation pattern $p$, we define the number of initiations of subscription, denoted $nis(p)$, to be the number of times subscription is initiated in $p$; formally, $nis(p)$ is the number of $i$ such that $p(i) = Subscription$ and either $i = 1$ or $p(i-1) = Demand$. Now, we define the cost of a single slot $i$ as follows. If $p(i) = Subscription$ then cost of $i$ is $ff$, otherwise it is $df_i$ (where $df_i = nr_i \cdot rc$). Now, we define the cost of $p$, denoted $cost(p)$, to be the sum of the cost of all the slots plus $nis(p) \cdot if$; note that we add the initiation fees corresponding to the number of initiations of subscription. We say that an allocation pattern is optimal if it has the least cost among all the allocation patterns.

THEOREM 7. *The above algorithm computes an optimal allocation pattern, i.e. after termination of the above algorithm, the allocation pattern given by the array decision is optimal.*

PROOF. We prove the theorem by showing that the following property, called $INV$, holds at the beginning of each iteration of the while loop, i.e. it is loop invariant.

($INV$) There exists an optimal allocation pattern (for all $n$ time slots) which is an extension of the allocation pattern defined by the array *decision* up to the first $i-1$ slots, i.e. there exists an optimal allocation pattern $q$ such that $q(j) = decision[j]$ for $j = 1, ..., i-1$.

We prove that the property $INV$ is loop invariant by induction on the number of iterations of the while loop. The theorem would automatically follow from this since at termination $i = last + 1$.

Clearly, in the beginning, i.e. when we first enter the while loop, $INV$ vacuously holds. Now consider any particular iteration. Assume that $i_0, i_1$ are the values of $i$ before and after execution of this iteration of the while loop. Assume that $INV$ holds before the execution of the body of the while loop, i.e. for $i = i_0$. Now, assume that $p$ is such a pattern, i.e. $p$ is an optimal allocation pattern and $p(j) = decision[j]$ for $j = 1, ..., i_0 - 1$. We will show that $INV$ holds after the execution of this iteration when $i = i_1$. Clearly, $i_1 > i_0$. If $p$ satisfies $INV$ for $i = i_1$ then we are done. So assume that $p$ does not satisfy $INV$ for $i = i_1$. Now we have the following cases. In each case, we show the existence of an allocation pattern whose cost is smaller than $p$ and thus contradicting the assumption that $p$ does not satisfy $INV$ for $i = i_1$, or we exhibit another optimal pattern which satisfies $INV$ for $i = i_1$.

(Case A): The value of the *status* variable is Demand before execution of the iteration. In this case, the first repeat statement is executed. We have the following two sub-cases.

(1) The repeat statement terminates because $cs \leq 0$ or $i > last$. In this case, the algorithm assigns Demand for all the time slots from $i_0$ up to $i_1 - 1$. Since we assumed that $p$ does not satisfy $INV$ for $i = i_1$, there exits at least one value of $j$ such that $i_0 \leq j < i_1$ and $p(j) = Subscription$. Let $j_0$ be the smallest value such

that $i_0 \leq j_0 < i_1$ and $p(j_0) = Subscription$. Now, let $p'$ be another allocation pattern such that $p'(j) = Subscription$ for $j = i_0, ..., j_0$ and $p'(j) = p(j)$ for all other $j$. It should be fairly easy to see that $Cost(p') \leq Cost(p)$; this is because $(\Sigma_{j=i_0}{}^{(j_0-1)}(df_j - ff)) > 0$ (if this were not the case the repeat loop would have terminated earlier). Now, we define another allocation pattern $p''$ whose cost is strictly smaller than $p'$, and hence that of $p$, contradicting the optimality of $p$. We have the following two sub-cases.

(a) $p'$ changes the allocation policy back to Demand after $j_0$ and before $i_1$, i.e. there exists $j_1$ such that $j_0 < j_1 < i_1$ and $p'(j_1) = Demand$. In this case, let $j_1$ be the smallest such value. Define $p''(j) = Demand$ for $j = i_0, ..., j_1$ and $p''(j) = p'(j)$ for all other values. Now, $Cost(p'') - Cost(p') = (\Sigma_{j=i_0}{}^{j_1-1}(df_j - ff)) - if$. The second term on the right hand side accounts for the extra initiation fees in $p'$; the first term is the difference in the sum of costs of each time slot. It should be obvious that the first term, although positive, is strictly less than $if$ (otherwise the repeat loop would have terminated earlier). Thus, $Cost(p'') < Cost(p')$ and hence $Cost(p'') < Cost(p)$ which contradicts the optimality of $p$.

(b) No $j_1$ as specified in the previous sub-case exists, i.e. $p'(j) = Subscription$ for all $j = i_0, ..., i_1$. In this case, define $p''(j) = Demand$ for all $j = i_0, ..., i_1$ and $p''(j) = p'(j)$ for all other values of $j$. Now, it is easy to see that $Cost(p'') \leq Cost(p')$, and hence $Cost(p'') \leq Cost(p)$ and $p''$ is optimal. Clearly $p''$ satisfies $INV$ for $i = i_1$.

(2) The first repeat statement terminates because $cs \geq if$. In this case, the algorithm sets $decision[j] = Subscription$ for $j = i_0, ..., (i_1 - 1)$. Since $p$ does not satisfy $INV$ for $i = i_1$, there exists at least one value of $j$ such that $i_0 \leq j < i_1$ and $p(j) = Demand$. Let $j_0$ be the smallest $j$ satisfying the above condition. Let $p'$ be an allocation policy such that $p'(j) = Demand$ for $j = i_0, ...j_0$ and $p'(j) = p(j)$ for all other $j$. If $j_0 = i_0$ then $p'$ and $p$ are identical; otherwise, $Cost(p) - Cost(p') = if - (\Sigma_{j=i_0}^{(j_0-1)}(df_j - ff))$ which is greater than zero as the second term on the right hand side is less than $if$ (otherwise, the repeat loop would have terminated earlier). Thus $Cost(p') \leq Cost(p)$ and hence $p'$ is also optimal. Now we have the following sub-cases.

(a) The allocation pattern $p'$ switches to Subscription some time before $i_1$, i.e. there exists a $j$ such that $j_0 < j < i_1$ and $p'(j) = Subscription$. Let $j_1$ be the smallest such $j$. Now, define $p''$ such that $p''(j) = Subscription$ for $j = i_0, ..., j_1$ and $p''(j) = p'(j)$ for all other values of $j$. It is easy to see that $Cost(p') - Cost(p'') = \Sigma_{j=i_0}^{(j_1-1)}(df_j - ff)$ and this value is strictly positive, otherwise the repeat loop would have terminated earlier. Hence the cost of $p''$ is lower than that of $p'$ and $p$; this contradicts the optimality of $p$.

(b) The previous sub-case does not hold, i.e. $p'(j) = Demand$ for $j = i_0, ..., (i_1 - 1)$. In this case, let $p''$ be an allocation policy such that $p''(j) = Subscription$ for $j = 1, ..., (i_1 - 1)$ and $p''(j) = p'(j)$ for all other values of $j$. Now, $Cost(p') - Cost(p'') \geq (\Sigma_{j=i_0}^{j=(i_1-1)}(df_j - ff)) - if$. The first term on the right hand side of the above inequality is greater than or equal to $if$ (due to the termination condition of the repeat loop) and hence $Cost(p') \geq Cost(p'')$ and hence $p''$ is also optimal. Clearly $p''$ satisfies $INV$

for $i = i_1$.

(Case B): The value of the *status* variable is Subscription before the iteration. The proof for this case is similar to case (A). □

*Complexity:* In the algorithm Opt, the total number of times the body of either of the inner repeat statements is executed is at most $n$. This is seen from the following analysis. The variable $i$ is initialized to 1 and is incremented in the body of the "repeat" loops; these are the only places $i$ is updated. Hence the total number of times the body of each "repeat" loop is executed, over all iterations of the outer "while" loop, is $n$. Hence the total number of iterations of the "while" loop is also $n$. It is also easy to see that the total number of iterations of the inner "for" loop is at most $n$. Hence the over all complexity of the algorithm is $O(n)$, i.e. linear in the number of time slots.

### 3.2 Estimated read pattern

This subsection considers estimated read patterns. The sequence of time slots may be infinite. In an estimated read (write) pattern the expected number of reads in the next time slot is estimated to be the floor average of the numbers of reads (writes) in the last $k$ time slots. This approach is an adaptation of the Sliding Window Protocol to the time cost model. A side-effect of this approach is the extension of the *Opt* algorithm to infinite sequences of time slots.

To deal with estimated read patterns we introduce the *E_Opt* algorithm. In the *Opt* algorithm, in order to make an optimal policy decision for the next time slot we needed the expected number of reads not only for the next time slot but also for subsequent time slots; the reason is that we wanted to know whether the initiation fee in case of Subscription can be amortized over many time slots. Thus, in the *E_Opt* algorithm we estimate the expected number of reads in the next time slot and also in subsequent time slots.

The allocation algorithm *E_Opt* is on-line and operates as follows. At the beginning of each time slot it computes the estimated value of the number of reads for the next one or more time slots, and uses these estimated values and runs the *Opt* algorithm for selecting Demand or Subscription for the next time slot.

More specifically, at run time, at the end of each time slot, *E_Opt* computes the estimated number of reads for the next time slot. Then, it uses the *Opt* algorithm to check if a decision can be reached; if so, we use the corresponding selection. Otherwise, i.e. if the slot is Undecided, *E_Opt* computes the estimated values for the time slot after the next, and repeats the above procedure until a decision about the next time slot (which is the First Undecided Slot) can be made, or until the sequence of estimated number of reads converges. By convergence we mean that $k$ successive estimated values are identical; when this occurs all further estimated values will be equal to these values. Later, we show that the sequence of estimated values will eventually converge after a finite number of steps.

Now, assume that the *E_Opt* algorithm has not reached a protocol decision for the current time slot, until convergence of the sequence is detected. Also assume that the convergence occurs exactly after $j$ iterations of the above procedure. Let $av$ be the value to which the sequence converged. Assume the policy in the last time slot is Demand. The fact that the algorithm *E_Opt* could not make a decision even after $j$ iterations indicates that the value of the variable $cs$ at this time lies strictly between 0 and $if$. If $av \cdot rc > ff$, then if we repeat the above procedure for sufficient number times eventually the value of $cs$ will be greater than $if$; hence, in this case, we switch to Subscription for the next time slot based on the estimated numbers of reads. On the other hand, if $av \cdot rc < ff$ then it is easy to see that after sufficient number of iterations of the above procedure eventually $cs$ will equal to 0; hence, in this case, we retain Demand for the next time slot. If $av \cdot rc = ff$ then the value of $cs$ will remain unchanged even after any number of iterations and will never cross $if$; hence, in this case also we retain Demand. On the other hand, if the policy in the last time slot is Subscription the $E\_Opt$ algorithm proceeds as follows. Since no decision has been reached until convergence, it follows that the value of the variable $cs$ at that time lies strictly between 0 and $-if$. In this case, it switches to Demand if $av \cdot rc \leq ff$; otherwise, it retains Subscription.

It is fairly straightforward to prove that the $E\_Opt$ algorithm is optimal if the numbers of reads in all the future time slots are equal to the corresponding estimated values.

Now, we show that the sequence of estimated values converges.

LEMMA 1. *Let $a_0, a_1, ..., a_{k-1}, a_k, ..a_j, ...$ be a sequence of positive integers such that for each $i \geq 0$, $a_{i+k} = \lfloor (a_i + ... + a_{i+k-1})/k \rfloor$. Further assume that max, min denote the maximum and the minimum of $a_0, ..., a_{k-1}$. The sequence converges within $(max - min) \cdot k$ steps.*

PROOF. It is easy to see that, for any $i \geq 0$, $a_{i+k} < \max\{a_i, a_{i+1}, ..., a_{i+k-1}\}$ and $a_{i+k} \geq \min\{a_i, ..., a_{i+k-1}\}$. Now, we divide the sequence into groups of $k$ successive numbers. Consider the $j^{th}$ group for some $j > 1$. The first member of this group is less than the maximum of the elements in group $(j-1)$, and greater than or equal to the minimum of the elements in group $(j-1)$. Now consider the second element of the $j^{th}$ group. Clearly it is less than the maximum of the $k$ elements that appear before it. Since the first member of the $j^{th}$ group is less than the maximum of the elements in group $(j-1)$, it follows that the second member of the $j^{th}$ group is also less than the maximum of the elements in group $j-1$. By repeating this argument inductively, it is easy to see that all members of group $j$, and hence the maximum of these, are less than the maximum of the members of group $j-1$. Similarly, it is easy to see that every member of group $j$ is greater than or equal to the minimum of the elements in group $j-1$. Thus, we see that, in successive groups, the difference between the maximum and the minimum decreases by at least 1. Hence the sequence $a_0, ..., a_i, ...$ converges within $k(max - min)$ elements. □

The following theorem shows that the sequence actually converges even faster than indicated by Lemma 1. Lemma 1 is used in the proof of this theorem.

THEOREM 8. *Let $a_0, ..., a_i, ...,$ referred to as the a-sequence, be a sequence of positive integers such that for all $i \geq 0$, $a_{i+k} = \lfloor (a_i + a_{i+1} + ... + a_{i+k-1})/k \rfloor$, and let max,min be the maximum and minimum of the first $k$ numbers in the sequence. This sequence converges within $(\log(max - min) + k)k$ steps, i.e. for all $i \geq (\log(max - min) + k)k$, the values of $a_i, a_{i+1}, ...$ are all equal.*

PROOF. In order to prove the theorem, we define another sequence of integers $d_0, ..., d_i, ...$, called the $d$-sequence, where $d_i = a_i - a_{i+1}$. Note that the integers $d_i$ can be either positive or negative. Since, for each $i \geq 0$, $a_{i+k} = \lfloor (a_i + a_{i+1} + ... + a_{i+k-1})/k \rfloor$, it follows that there exists a sequence of integers $c_k, c_{k+1}, ..., c_{i+k}, ...$ such that $0 \leq c_{i+k} < k$ and the following equation holds.
(A) $a_i + a_{i+1} + ... + a_{i+k-1} = ka_{i+k} + c_{i+k}$
Similarly, we have
(B) $a_{i+1} + a_{i+2} + ... + a_{i+k} = ka_{i+k+1} + c_{i+k+1}$
Subtracting the left hand sides and right hand sides of (A) and (B), we get (C) given below.
(C) $d_i + d_{i+1} + ... + d_{i+k-1} = kd_{i+k} + (c_{i+k} - c_{i+k+1})$
Now we will prove the following claim.
**Claim:** For every $i \geq 0$, the following equality given by (D) holds.
(D) $d_i + 2d_{i+1} ... + jd_{i+j-1} + ... + kd_{i+k-1} = c_{i+k}$

Proof of the claim: First we observe that the following equality (E) given below holds.
(E) $a_{i+1} + 2a_{i+2} ... + ja_{i+j} + ... + ka_{i+k} = a_i + 2a_{i+1} ... + ja_{i+j-1} + ... + ka_{i+k-1} - c_{i+k}$
The validity of equation (E) is seen by substituting $(a_i + a_{i+1} ... + a_{i+k-1} - c_{i+k})$ in place of $ka_{i+k}$ in the left hand side of the equation (E) (this is a sound substitution due to equation (A)). Equation (D) follows directly from (E) by moving $c_{i+k}$ to one side and all other terms to the other side of the equality, and replacing each $a_i - a_{i+1}$ by $d_i$ etc. □

From (D), we have the following.
(F) $d_{i+k-1} = \frac{1}{k}(c_{i+k} - d_i - 2d_{i+1} ... - jd_{i+j-1} ... - kd_{i+k-2})$
By substituting the expression given by (F) for $d_{i+k-1}$ in the equation (C), and by simple algebraic manipulation, we get the following equation.
(G) $d_{i+k} = x + \frac{1}{k^2}(d_i(k-1) + d_{i+1}(k-2) + ... + d_{i+j}(k-j-1) + ... + d_{i+k-2})$
where $x = (\frac{c_{i+k+1}}{k} - \frac{c_{i+k}}{k} + \frac{c_{i+k}}{k^2})$. For any number $y$, let $|y|$ denote the absolute value of $y$. From equation (G), we see that the following inequality holds.
(H) $|d_{i+k}| \leq |x| + \frac{1}{k^2} \max\{|d_i|, ..., |d_{i+k-2}|\}(k - 1 + k - 2 + ... + 1)$
Since $c_{i+k+1}$ and $c_{i+k}$ are positive integers less than $k$, it follows that $|x|$ is less than 1. Using this observation and by replacing the sum $(k - 1 + k - 2 + ... + 1)$ by $k(k-1)/2$ in (H) , we get the following inequality.
(I) $|d_{i+k}| \leq \lceil \frac{1}{2} \max\{|d_i|, |d_{i+1}|, ..., |d_{i+k-1}|\} \rceil$
The inequality (I) indicates that the absolute value of each $d_{i+k}$ is less than or equal to half of the maximum of the absolute values of the previous $k$ numbers. Now, we divide the $d$-sequence $d_0, ...d_{i+k}, ...$ into groups of $k$ consecutive numbers. Now consider the $j^{th}$ group. Using (I), we see that the absolute value of any number $z$ in a group $j$ is less than or equal to half of the maximum of the absolute values of any of previous $k$ numbers; of these $k$ numbers some of them belong to group $j-1$ and others belong to group $j$; now, by a simple induction on the number of group $j$ elements that appear before $z$, it can easily be shown that the absolute value of each one of them, including that of $z$, is less than or equal to half of the maximum absolute value of elements in group $j-1$. Since $max$, $min$ are the maximum and

minimum of $a_0, a_1, ..., a_{k-1}$, it is easy to see that the maximum absolute value of any number in $d_0, ..., d_{k-1}$ is bounded by $(max - min)$. It should be easy to see that for $j = \log(max - min)$, the absolute value of all elements in group $j$ is less than or equal to 1.

In order to prove the theorem, we also divide the $a$-sequence $a_0, ..., a_i, ...$ into groups consisting of successive $k$ elements. From the above argument, we see that for some $j \leq \log(max - min)$, the successive elements in group $j$ do not differ by more than one (this is because the $d$ values in this group have an absolute value less than or equal to 1). Hence the maximum and minimum of the elements in group $j$ differ by at most $k$. From lemma 1, we see that from group $j$ onwards the $a$-sequence converges with in $k^2$ elements. Hence from the beginning, the $a$-sequence converges within $(\log(max - min) + k)k$ elements. □

In many practical situations the $a$-sequence converges faster than the bound given by theorem 8.

### 3.3 Demand with Cache invalidation

In this subsection we assume that the Demand policy can be combined with cache invalidation. Consider first the expected read-write pattern. The algorithm *Opt* of subsection 3.1 can be applied verbatim, using a revised demand fee $df$. $df_i$ is the minimum of the Demand cost for slot $i$ as defined in subsection 3.1 (this corresponds to the case where cache invalidation is not used), and the Demand with cache invalidation cost. The Demand with cache invalidation cost, denoted $dci_i$, is the cost of the read requests and invalidation notifications in the $i$'th slot. Each read in the slot that is immediately preceded by a write, namely a critical read, costs $rc$. Each write in the slot that is immediately preceded by a read, namely a critical write, costs $ic$. The expected number of critical reads $n_c$, assuming that there are $n_r$ expected reads and $n_w$ expected writes in a time slot and assuming that they are uniformly distributed, is given as $n_c = \frac{n_r \cdot n_w}{n_r + n_w}$. Similarly, the expected number of critical writes is also $n_c$. Then $dci_i$ is $n_c \cdot (ic + rc)$. With this revised demand fee, the algorithm *Opt* can be applied verbatim for access cost optimization.

The algorithm that optimizes access cost for the estimated patterns can also be applied verbatim using the so revised demand fee.

## 4. DIVERGENCE CACHING

In this section we assume that some reads need not obtain the most recent version of the object. We use the request cost model and the complexity measures introduced in section 2. Namely, the cost of satisfying a read request sent from the client to the server is $rc$. The cost of propagating a write (or an update) of the object from the server to the client is $wc$.

The complexity measure for the worst-case is competitiveness. If the probabilities of the relevant requests are fixed and known, then the complexity measure for the expected case is the expected cost of a relevant request. Otherwise, as in section 2, we use the average expected cost of a request. The analysis of this case is performed experimentally.

Since reads need not obtain the most recent version of the object, the algorithms that analyze are based on a new mechanism called *divergence caching*. The mechanism is a hybrid between Subscription and Demand in a sense that will become clear shortly. Divergence caching uses the following two techniques for each object stored at the server.

—The first technique uses *tolerant reads*. In order to reduce access charges, each read issued by a client is associated with a natural number representing the divergence-tolerance for the read. For example, read(IBM,3) represents a request to read IBM's stock price (i.e., the object) with a tolerance of 3. This read can be satisfied by any of the three latest versions of IBM's stock price; in other words, it can be satisfied by a version that is up to two updates behind the most recent version.
  Formally, each write request creates a new version of the object. Each read request has a *tolerance* $t$, specifying how recent a version of the object is required. We denote such a read by $r(t)$. We assume that the read tolerance is an integer in the range $1, \ldots, M$, where $M$ is known in advance. A read tolerance of 1 indicates that the most recent version of the object is requested; a read tolerance of $k$ indicates that any of the last $k$ versions will do. A *schedule* is a finite sequence of requests; for example, $w, w, r(1), w, r(3), r(2), w$.

—The second technique is *automatic refresh*. It means that the server has a refresh rate for every client. For a natural number $\ell$, a *refresh rate* of $\ell$ means that the version of the object cached at the client is at most $\ell - 1$ updates behind the version at the server. The server automatically propagates to the client the $\ell$th version since the last transmission of the object to the client. The client saves the last version of the object it receives from the server. Thus, those reads at the client with a tolerance greater than $\ell$ can be satisfied locally; that is, without access to the on-line database (so there is no charge). Therefore, charges are incurred only for automatic refresh, and for each read with a tolerance that is lower than $\ell$.
  The refresh rate can have any value between 1 and infinity. A refresh rate of 1 means that the client has a regular copy of the object (i.e., the client is on Subscription), and each update of the object is propagated to the client. A refresh rate of infinity means that the client is on Demand, and each read, regardless of its tolerance, will require a transmission from the server to the client (even when the object has not changed since the last read). For a finite refresh rate $\ell > 1$, the client behaves as if it is on Subscription (i.e., it reads the local version) for reads with tolerance $\ell$ and higher; the client behaves as if it is on Demand for reads with a tolerance lower than $\ell$. The optimal refresh rate, that is, the refresh rate that minimizes the total read and write costs paid, depends on the ratio between the frequency of updates at the server on one hand, and the frequency and tolerance of reads at the client on the other hand.

In this model, a read with tolerance less than the refresh rate costs $rc$, and a read with tolerance greater than or equal to the refresh rate has zero cost. A write costs $wc$ if it is propagated to the client, otherwise it costs zero.

To contrast this scenario with the ones in previous sections observe the following. In the previous sections, at any point in time the client is either on Subscription (and pays for the writes) or Demand (and pays for the reads); it may switch between Subscription and Demand periodically. In contrast, in the current scenario, at any point in time the client is on Subscription for some reads (the ones with a tolerance higher than the refresh rate) and on Demand for other reads. The client pays for the Demand-reads, and for some of the writes (i.e., the ones that are transferred to the client).

We assume that $p_{r_i}$ is the probability that a relevant request is a read with a tolerance $i$, $1 \leq i \leq M$. The probability that a relevant request is a write is $p_w = 1 - \sum_{i=1}^{M} p_{r_i}$. (That is, $p_w$ is the analogue of $\theta$ in the preceding sections.)

In this section we propose and analyze two divergence caching algorithms, Static Divergence Caching (SDC) and Dynamic Divergence Caching (DDC). The SDC algorithm works as described above, and it has a fixed refresh rate. A refresh rate of $\ell$ means that the object is automatically transmitted to the client every time the object has been updated by $\ell$ writes, without having been sent to the client in the meantime. Thus, when the refresh rate is $\ell$, the client always has one of the $\ell$ most recent versions of the object in its local memory, and can satisfy any read request with a tolerance of $\ell$ or more with that local version.

One of the problems that we solve in this section is to determine the optimal refresh rate for given probabilities of writes and reads of each tolerance. This would have been easy if the server propagated to the client exactly one in every $\ell$ writes. However, a refresh rate of $\ell$ does not necessitate that the server does so. For example, if the client solicits a refresh (to satisfy a read with a low tolerance) after $\ell/2$ writes, then that refresh will reinitialize the refresh counter. In other words, an automatic, or solicited refresh will occur after $\ell$ writes only if in the meantime there was no solicited refresh.

The DDC algorithm is similar to the static one, except that the refresh rate varies over time. It does so since we assume that the probability of each type of request is unknown or it varies over time. Our DDC algorithm learns these probabilities by "watching" a sliding window of read-write requests, and based on it the algorithm continuously adapts the refresh rate to the current request probabilities.

We analyzed the DDC algorithm experimentally. Details of the experimental setup are given in subsection 4.4. The experiments indicate that the appropriate window size is approximately 23, in the sense that for a higher window size the cost-improvement is marginal. They also indicate that if the request probabilities are fixed, then the DDC algorithm comes within 15–45% of the optimal SDC algorithm, i.e. the static algorithm with the optimal refresh rate for the given request probabilities. If the request probabilities are fixed, then the DDC algorithm should be used when these probabilities are unknown a priori; when the parameters are known and fixed one should use the optimal SDC algorithm. The experiments also indicate that if the request probabilities vary over time, then the DDC algorithm with a window size of 23 or higher has a cost that is approximately 30% lower than that of any static algorithm.

In addition to the expected case, we also analyze the worst case for both algorithms, and we show that the DDC algorithm is superior to the SDC algorithm in the sense that the DDC algorithm is competitive, whereas the SDC is not competitive.

The rest of this section is organized as follows. In subsection 4.1, we make a detailed mathematical analysis of the Static Divergence Caching algorithm; particularly, we determine the optimal refresh rate, i.e. the rate that minimizes the expected cost of a request, for the case where the probability distribution of the read and write requests is known and fixed over time. Subsection 4.2 presents the Dynamic Divergence Caching algorithm, which is appropriate when the probability distributions change over time or are simply unknown. Subsection 4.3 gives a theoretical analysis of the worst case complexity of the SDC and DDC algorithms. In subsection 4.4 we give experimental analysis of the average expected cost complexity measure.

### 4.1 Fixed and known distributions

In this subsection we assume that the $p$'s are fixed and known a priori, and we are using the Static Divergence Caching algorithm. We develop the expected cost of a request in the schedule as a function of the refresh rate, and we show how to find the minimum of that function. This minimum is the optimal refresh rate.

Our first goal is to compute the expected cost for any given fixed refresh rate $\ell$. The case $\ell = \infty$ is straightforward. The client pays $rc$ for every read, and nothing for any of the writes. Thus the expected cost is $rc \sum_{t=1}^{M} p_{r_t}$. The case $\ell = 1$ is also straightforward; the client pays $wc$ for each write and nothing for any reads, so the expected cost is $wc \cdot p_w$.

Otherwise, for fixed integer $1 < \ell < \infty$, we define a *significant* request to be either a write or a read with a tolerance less than $\ell$. Reads with a tolerance at least $\ell$ are called *insignificant* requests. Notice that the insignificant requests have a zero cost, and some of the writes also have a zero cost. However, as explained previously, the number of nonzero cost writes is not $(1/\ell)$th of all the writes (that would make the derivation of the optimal refresh rate easier).

The probability that a request is significant is $p_w + \sum_{t=1}^{\ell-1} p_{r_t}$. Let $R_\ell$ be the conditional probability that a request is a write, given that it is significant. Using the formula for a conditional probability [2] we get: $R_\ell = p_w/(p_w + \sum_{t=1}^{\ell-1} p_{r_t})$. (Notice that $R_1 = 1$ and $R_{M+1} = p_w$.)

Now, from here on in we condition all probabilities and expectations on the event that the request being considered is significant. (At the end, we will need to multiply through by the probability of this event, which is $p_w + \sum_{t=1}^{\ell-1} p_{r_t}$.)

The probability that an arbitrary request has a nonzero cost is: the sum of the probabilities that the request is a significant read (which is clearly $1 - R_\ell$), plus the probability that the request is a write which has a nonzero cost. Hence we need to calculate the probability that a significant request is a write which has a nonzero cost. Observe that a write in a schedule has a nonzero cost only when the sequence of significant requests leading up to and including that request is either a significant read followed by $\ell$ writes, or a read followed by $2\ell$ writes, or a read followed by $3\ell$ writes, etc. (otherwise the write has a zero cost). These events are all disjoint, so the probability of any of them occurring is just the sum of the probabilities of each. This sum in the limit is

$$\Pr[\text{Request is write which has a nonzero cost}] = \sum_{n=1}^{\infty} (1 - R_\ell)(R_\ell)^{n\ell}$$

[2]this formula is $p(A|B) = p(A \cap B)/p(B)$; in this case, since a write is a significant request, $p(A) = p(A \cap B)$.

$$= \frac{(1 - R_\ell)R_\ell^\ell}{1 - R_\ell^\ell}.$$

Thus the expected cost of an arbitrary significant request is

$$E(\text{cost}) = (1 - R_\ell)\left(rc + wc\frac{R_\ell^\ell}{1 - R_\ell^\ell}\right). \tag{17}$$

Recall that this was conditioned on the request being significant, so the actual expected cost of one arbitrary request is

$$\left(\sum_{t=1}^{\ell-1} p_{r_t} + p_w\right)(1 - R_\ell)\left(rc + wc\frac{R_\ell^\ell}{1 - R_\ell^\ell}\right) = \sum_{t=1}^{\ell-1} p_{r_t}\left(rc + wc\frac{R_\ell^\ell}{1 - R_\ell^\ell}\right) \tag{18}$$

since the first two multiplicands on the left-hand side are the probability that a request is significant and the probability that a request is a read with tolerance at most $\ell$ given that it is significant.

Putting Equation 18 together with the extreme values for the refresh rate we obtain that:

THEOREM 9. *For the SDC algorithm with refresh rate $\ell$, the expected cost of a request is:*

$$E(cost) = \begin{cases} wc \cdot p_w & \text{for } \ell = 1 \\ \sum_{t=1}^{\ell-1} p_{r_t}\left(rc + wc\frac{R_\ell^\ell}{1-R_\ell^\ell}\right) & \text{for } 1 < \ell < \infty \\ rc \cdot \sum_{j=1}^{M} p_{r_j} & \text{for } \ell = \infty \end{cases} \tag{19}$$

COROLLARY 2. *In the Static Divergence Caching algorithm the minimum cost per request can never be achieved for any finite refresh rate greater than $M$.*

PROOF. Straightforward based on the cost function in Equation 19 that for any $\ell > M$ the second line is bigger than the third line. □

Thus, assuming that all the $p$'s are fixed and known, the algorithm for finding the optimal refresh rate is trivial. All one must do is compute the $M + 1$ different costs associated with the refresh rates $1, 2, \ldots, M$ and $\infty$ according to Equation 19, and then choose the refresh rate which gives the minimum cost.

### 4.2 The dynamic divergence caching algorithm

The Dynamic Divergence Caching algorithm works for probabilities ($p$'s) that are unknown and that may vary over time. The algorithm varies the refresh rate of an object $x$ at the client. It does so by computing the $p$'s based on a window of the $k$ latest relevant read and write requests, using Equation 19 to recompute the optimal refresh rate, and establishing it as the new refresh rate.

Now we explain the algorithm in detail. Recall, the relevant reads are issued at the client, and the relevant writes are issued at the server. At any point in time there is a refresh rate, $r$. Each read at the client with a tolerance higher than $r$ is satisfied locally, and each read at the client with a tolerance lower than $r$ results in a refresh solicitation to be sent to the server; the server responds by refreshing $x$ (i.e., sending the latest version of $x$). The server also performs an unsolicited refresh of $x$ when it receives $r$ consecutive write requests since the last refresh of $x$ (this last refresh may be either solicited or unsolicited).

Adaptation of the refresh rate occurs at each refresh point (solicited or not), as follows. At every point in time, the server maintains the *write-sliding-window*, i.e. the set of time-stamps of the last $k$ write requests. Each time a new write is received at the server its time-stamp is added to the write-sliding-window, and the smallest time-stamp in the window is deleted. At every point in time, the client maintains the *read-sliding-window*, i.e. the time-stamp and tolerance of the $k$ latest reads. Specifically, the client maintains a set of $k$ pairs $(i, t)$ where $i$ is a tolerance and $t$ is a time-stamp.

When the client solicits a refresh, it piggybacks the read-sliding-window on the refresh request. Before refreshing $x$, the server computes the new refresh rate as follows. It uses the time-stamps in the read-sliding-window and the write-sliding window in order to compute the *request-numbers-window*; it is the number of writes denoted $w$, the number of reads with tolerance 0 denoted $r_0$, the number of reads with tolerance 1 denoted $r_1$, the number of reads with tolerance 2 denoted $r_2$, etc., for the last $k$ read-write requests. Then $p_w$ is taken as $w/k$, $p_{r_1}$ is taken as $r_1/k$, $p_{r_2}$ is taken as $r_2/k$, etc. Then the server uses Equation 19 in order to compute the optimal refresh rate; this will become the new refresh rate. Then the server responds to the refresh request, and it informs the client of the new refresh rate by piggybacking this rate on the refresh response.

When the server performs an unsolicited refresh, it piggybacks the write-sliding-window on the message. Using the write-sliding-window the client computes a new refresh rate as follows. The client computes the request-numbers-window and the $p_w$ and $p_r$'s as explained above for the server. The client uses Equation 19 in order to compute the optimal refresh rate; this becomes the new refresh rate.

In subsection 4.4, we analyze experimentally the DDC algorithm and compare it with the SDC algorithm.

### 4.3 Worst case analysis

In this section we analyze the worst case behavior of the Dynamic Divergence Caching and the Static Divergence Caching algorithms. We show that the DDC algorithm is competitive, whereas the SDC algorithm is not competitive.

We begin by presenting the optimal off-line algorithm, called $O$. The algorithm is optimal in the sense that its cost on any schedule of requests is lower than the cost of any other algorithm. Let $\sigma$ be an arbitrary schedule. Assume that at the beginning of $\sigma$ the client has the latest version of the data object.

Off-line algorithm $O$ marks reads in the schedule as follows. The first read that is marked is the first read $r_1$ with a tolerance that is less than or equal to the total number of writes that precede $r_1$. The next read that is marked, $r_2$, is the first read following $r_1$ that satisfies the following property: the total number of writes between $r_1$ and $r_2$ is bigger than or equal to the tolerance of $r_2$. In general, the $r_i$th read that is marked, is the first read following $r_{i-1}$ that satisfies the following property: the total number of writes between $r_{i-1}$ and $r_i$ is bigger than or equal to the tolerance of $r_i$.

After completing the marking, algorithm $O$ determines the reads or writes that will be transferred between client and server. This is done as follows. If $rc <$

$wc$, then the client sends to the server every marked read. Otherwise, the server propagates to the client the last write that precedes a marked read. All the other reads and writes are local, and consequently incur zero cost. Thus, the cost of $O$ is $\min(rc, wc) \cdot N$, where $N$ is the total number of marked reads.

For example, consider the schedule $w, w, r(1), w, r(3), r(2), w, w, r(2)$. In this schedule the third and ninth requests are the marked reads, and the cost of $O$ on the schedule is $2\min(rc, wc)$.

LEMMA 2. *Algorithm $O$ is an optimal off-line algorithm.*

PROOF. Consider a schedule $\sigma$ and an algorithm $A$, and suppose by way of contradiction that $A$ has a lower cost on $\sigma$ than $O$. Assume first that $rc < wc$. If $A$ propagates some write $w$ in $\sigma$ from the server to the client, then the cost of $A$ on $\sigma$ can be reduced by as follows. Instead of incurring the cost for $w$, incur the cost on the first read that succeeds $w$, i.e. transfer that read from the client to the server. Thus, let us assume without loss of generality that $A$ incurs a cost only for reads. The first read for which $A$ incurs a cost cannot come after $r_1$, namely the first read on which $O$ incurs a cost; otherwise $r_1$ will read a version for which it is intolerant. By induction it can be shown that the $i$th read for which $A$ incurs a cost cannot come after $r_i$, namely the $i$th read on which $O$ incurs a cost; otherwise $r_i$ will read a version for which it is intolerant. Thus the cost of $A$ cannot be lower than the cost of $O$ on $\sigma$.

Assume now that $wc < rc$. Then we can assume without loss of generality that $A$ incurs a cost only for writes. As in the case $rc < wc$, it can be shown by induction that the cost of $A$ cannot be lower than the cost of $O$ on $\sigma$. If $wc = rc$ we can also assume without loss of generality that $A$ incurs a cost only for writes, and the previous argument holds. □

THEOREM 10. *The Static Divergence Caching algorithm is not c-competitive for any $c > 0$.*

PROOF. We will demonstrate that there exist schedules for which the ratio cost(A)/cost(O) is unbounded. Assume first the fixed refresh rate of the SDC algorithm is infinity. Construct a schedule that contains only reads with tolerance 1. The cost of the SDC algorithm is equal to the number of reads in the schedule; and the cost of Algorithm $O$ on the schedule is 0. If the fixed refresh rate is $\ell < \infty$, then construct a schedule that contains only writes. Again, the cost of Algorithm $O$ on the schedule is 0, but the SDC algorithm will pay for every $\ell$th request in the schedule. □

In order to prove that DDC algorithm is competitive we need the following lemma.

LEMMA 3. *Let $\sigma$ be a schedule. Let $B_1$,...,$B_m$ be the blocks of consecutive reads and writes in $\sigma$, such that each block consists of either only reads or only writes, and adjacent blocks contain different kind of requests. Then, between any two read requests marked by Algorithm $O$ there are at most $2(M+1)$ blocks.*

PROOF. Consider two read requests marked by Algorithm $O$, $r_1$ and $r_2$. Assume that there are more than $2(M+1)$ blocks between $r_1$ and $r_2$, and consider the last read-block $B_r$ that precedes the block of $r_2$. However, there are more than $M$ write blocks between $r_1$ and $B_r$, thus there are more than $M$ writes. The maximum tolerance of a read in $B_r$ is $M$, and this contradicts the correctness of Algorithm $O$. □

Theorem 11. *The Dynamic Divergence Caching algorithm is c-competitive for* $c = 2(M+2)k \max(wc, rc)/\min(wc, rc)$ .

Proof. Assume that the window size of the DDC algorithm is $k$. Consider any schedule $\sigma$ and break it into blocks as described above. Assume that there are $l$ blocks in $\sigma$. By Lemma 3 above, the cost of the algorithm $O$ on $\sigma$ is at least $\min(wc, rc) \cdot l/(2M+2)$.

Now consider the cost of the DDC algorithm. For one read block the DDC algorithm incurs a cost of at most $k \cdot rc$, because after that the window will contain entirely reads, so the algorithm's estimate of $p_w$ will be 0 and it will set its refresh rate to 1. Similarly, the DDC algorithm incurs a cost of at most the $k \cdot wc$ for any one write block, because after $k$ writes it will have set its refresh rate to $\infty$. Thus, by Lemma 3 above, the competitiveness factor of DDC is at most $2(M+2)k \max(wc, rc)/\min(wc, rc)$ . □

### 4.4 Experimental results

Mathematical analysis of the expected and average expected costs for the DDC algorithm is not tractable, i.e. it is difficult to obtain closed form solutions and it may require additional assumptions. For these reasons, we take an experimental approach to this problem. The goal of the experiments is to compare the the perfomance of the DDC algorithm with that of the optimal SDC algorithm, i.e. the SDC algorithm with a refresh rate which is optimal for each schedule. Clearly, the optimal SDC algorithm is unobtainable if the request-probabilities are unknown.

We have conducted many experiments, where we randomly generated schedules of requests, and compared the algorithms' performance on those schedules. In this section, we summarize the results. In both subsections we compare the DDC and SDC algorithms on schedules of requests generated by Poisson processes. For all our experiments, we fixed the maximum tolerance of any read, denoted $M$, at 20. We denote by $\lambda_w$ the intensity of the process that generated writes, and by $\lambda_{r_j}$ (for $1 \leq j \leq 20$) the intensity of the process that generated reads with tolerance $j$. This model is obviously equivalent to taking $p_w = \lambda_w/(\lambda_w + \lambda_{r_1} + \cdots \lambda_{r_M})$ and $p_j = \lambda_{r_j}/(\lambda_w + \lambda_{r_1} + \cdots \lambda_{r_M})$.

We took $wc$ to be 1, and $rc$ to be $1+\omega$ for various nonnegative values of $\omega$.

The difference between the subsections lies in the selection of the input schedules. In the first subsection the input schedules have fixed distribution parameters ($\lambda$'s), and in the second subsection they vary over time.

4.4.1 *Fixed distribution parameters.* We generated schedules of 1600 requests, for each of 84 different values of the 21 parameters $\lambda_w$ and $\lambda_{r_j}$ for $1 \leq j \leq 20$. For each schedule $\sigma$ we used Equation 19 to compute the optimal refresh rate, $k_\sigma$. Then we ran on $\sigma$ the DDC algorithm with various window sizes, and the SDC($k_\sigma$) algorithm.[3]

In Figure 2 we show a grand summary of the data —the average of all 84 runs, for the case where $\omega = 0$. In particular, we plot the ratio of the average (over all

[3]SDC($t$) is the SDC algorithm with refresh rate $t$.

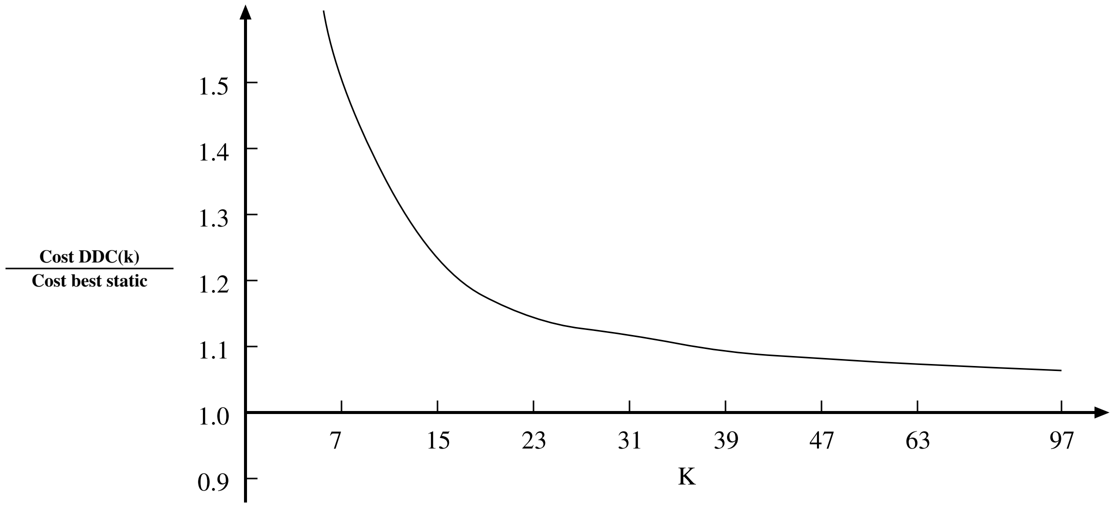


Fig. 2. Experimental results for fixed $\lambda$'s for $\omega = 0$ case. Graph shows ratio of costs paid by DDC($k$) to costs paid by the optimal static algorithm.

runs) cost of the DDC algorithm with window size $k$ (hereinafter DDC($k$)) to the average cost of the SDC algorithm, as a function of $k$. Notice that the threshold of the SDC algorithm differs from schedule to schedule. The solid line at 1 represents the average cost of the optimal static algorithm for each case.

The main result of these experiments is that the performance of $DDC(k)$ improves sharply as $k$ increases to about 23, and for $k \geq 23$ the cost of DDC($k$) is only 10–15% greater than the cost of the best static algorithm. In fact, this pattern was observed for almost every schedule of the experiment, although the graph in Figure 2 shows only data for all the runs averaged together.

Two other important quantities are *not* shown in the graph. One is the performance of the optimal off-line algorithm. The average cost of the optimal static algorithm was typically about 2.25 times the average cost of the optimal off-line algorithm. Recall that the off-line algorithm has the advantage of seeing all the requests in advance, something that is impossible in real life.

We also compared the optimal static algorithm to the better (on the particular run) of the static algorithms with refresh rates 1 and infinity. This corresponds to traditional methods that do not allow for divergence caching, but only for caching (refresh rate 1) or not caching (refresh rate infinity). The optimal static algorithm showed, on average, a factor of two improvement. This demonstrates the power of divergence caching. In other word, being able to satisfy reads by out of date versions reduces the access cost by a factor of two.

Note that the factor of two improvement is an average over 84 different settings of the $\lambda$s. For certain values of the $\lambda$s a refresh rate of 1 or infinity was the optimal value (typically with writes being either a very high or very low percentage of all requests), and for certain values of the $\lambda$s the improvement with divergence caching was much greater than fourfold.

We also computed the performance of our algorithms for values of $\omega$ ranging

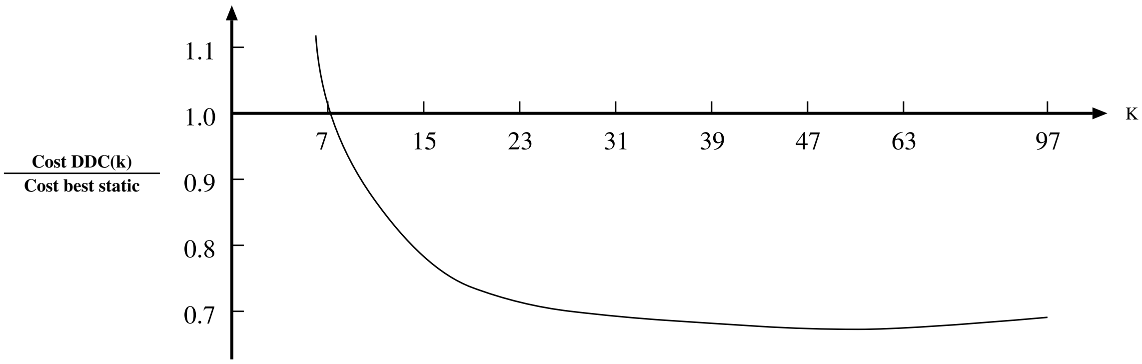


Fig. 3. Experimental results for time-varying $\lambda$'s for $\omega = 0$ ca se. Graph shows ratio of costs paid by DDC($k$) to costs paid by the optimal static algorithm.

from 0.1 to 0.9. We performed the same experiments as for the case of $\omega = 0$. The results were broadly similar.

4.4.2 *Time-varying distribution parameters.* As we describe above in Section 4.3, theory leads us to believe that if the $\lambda$'s change over time, then the dynamic divergence caching algorithms will outperform all static algorithms. We ran several experiments where we varied the $\lambda$s over time, and we summarize the results here. They confirm our expectation.

We begin with the case of $\omega = 0$. A typical experiment is presented in Figure 3, which reports the average of 40 runs of 1600 requests each. In each run, we picked each of the $\lambda_{r_j}$s uniformly at random from 1 to 100, and then uniformly at random picked the fraction $p$ of all requests that would be writes. (Thus $\lambda_w$ was set to be $p$ times the sum of the $\lambda_r$'s.) Every 157 requests, the $\lambda$'s were randomly assigned new values according to the same rules.

We determined empirically which refresh rate gave the best performance for the SDC algorithm over the forty runs. In figure 3 we plot the ratio of the average (over all 40 runs) cost of DDC($k$) to the average cost of the SDC algorithm with optimal refresh rate. The main result is that the cost of DDC($k$) improves with $k$, with considerable improvement up to around $k = 23$, and slight improvement thereafter. For $k \geq 23$, the average cost of DDC($k$) is roughly 70% of the cost of the best static algorithm.

The cost-advantage of the dynamic divergence caching algorithms varied with the method of changing the $\lambda$s over time. If the ratio of writes to reads remained constant over time, while the individual $\lambda_{r_j}$s varied, then the dynamic divergence algorithms were only slightly better than the best static algorithm. On the other hand, much larger improvements than those shown in Figure 3 were found when we alternated periods where writes were at most 30% of all requests with periods where writes were at least 70% of all requests. In practice, alternating phases of read-intensive and write-intensive patterns might be common.

## 5. RELEVANT WORK

This paper does not fit neatly into an existing research discipline. However, one relevant research area is caching in various contexts, e.g. networking and the World-Wide Web (see for example [Liu and Cao 1997; K. Obraczka and et al. 1996; Gwertzman and Seltzer 1996; J. Gwertzman 1995; A. Chankhunthod and et al. 1996; Crovella and Carter 1995; J. D. Guyton 1995]), Client/Server databases (e.g. [Zahariousdakis and Carey 1997; Franklin 1996]), distributed file systems (e.g. [Levy and Silbershatz 1990; J. Kistler 1991; G. Popek and et al. 1990; Gray and Cheriton 1989]), and distributed shared memory(e.g. [Li and Hudak 1989; Nitzber and Lo: 1991]). However, our approach has some important aspects whose combination is unique. First, we study cost models as an independent concept, unrestricted by the limitations of a particular system, protocol, or application. In existing studies, a particular environment often restricts the caching options. Second, we study the cost of caching in both, weak-consistency and strong-consistency environments. Third, we take a mostly analytical approach and we study both average case and worst case complexities. As far as we know, these aspects have not been studied in combination in any of the existing works.

Another related research area is query processing in parallel and distributed databases (see for example [M.J. Franklin and Kossman 1996; Carey and Lu 1986; Srivastava and Elsesser 1993]). The relevance is in the sense that the query shipping versus data shipping versus hybrid approaches in query processing bear a resemblence to our subscription, demand and sliding window protocols. However, there are important differences. First, these papers deal with query plans for relational operators, mainly joins, that can be parallelized and and pipelined. Second, they do not consider sequences of database updates and queries, as we do in this paper. Third, these papers take an experimental approach, and thus they do not provide cost formulas. And although there are quite a few mathematical cost analysis works in the database literature (e.g. [Christodulakis 1983; Christodulakis 1984] in the area of performance evaluation of a centralized physical database design) our problem has not been considered previously using a similar approach to ours.

Regarding our worst case analysis of the protocols, the closest problem addressed in the theoretical computer science community is called rent-to-buy, or the ski-rental problem (see [Irani and Karlin 1997]) as described below: assuming that the cost of renting a pair of skis is $r$ and the cost of buying it is $b$, after how many rentals should one buy the skis (assuming that the number of future skiing events are unknown). There it is shown that the following algorithm is 2-competitive, and this competitiveness coefficient is optimal. The algorithm rents the pair of skis $b/r$ times, and then it buys it. The difference between the rent-to-buy and subscription-demand is that after buying, no additional cost is incurred; on the other hand, in switching between subscription and demand one switches between paying for writes and paying for reads, but in no case are all future requests (events) free of charge.

In the AI literature, work that is close to our approach appears under the subject of negotiation (see [Rosenschein and Zlotkin 1994] for a good introduction to the subject and a survey of the relevant literature). The difference is that we are concerned with the problem of choosing a minimal-cost retrieval protocol (e.g. Subscription or Demand) from a fixed set of candidates, whereas the AI work is concerned with the protocol of arriving at a mutually agreed price. In other words, we assume that the object owner has fixed several pricing options, one of which is selected by the buyer. The AI work assumes that these options are subject to negotiations. We feel that our approach is closer to the existing commercial model of business. For example, the phone companies have several cost plans, among which the customer selects one, without negotiation. In the database literature, an approach similar to ours was proposed in [J. Sidel and et al. 1996]. That paper has also put price-negotiation at the heart of its method.

Finally, some of our results are based on preliminary work published in [Y. Huang and Wolfson 1994a; P. Sistla and et al. 1996; Y. Huang and Wolfson 1994b], although the cost models in the present paper are different.

## 6. DISCUSSION AND CONCLUSION

In general there are two basic business models for information providers. One is advertiser paid and the other is customer paid. In existing media both models coexist, e.g. newspapers and cable tv. Based on this we predict that a similar coexistence will occur in the future digital libraries. In this paper we addressed the issue of cost management/optimization in the customer paid business model.

Some specific digital library applications of our work include various forms of electronic news services, such as stock trading and electronic mail services. In the case of stock trading, an object is information (including price) of a particular stock or a group of stocks. In other cases the object may be a more complex data structure such as a view or a queue. For example, the object may be a queue of news-items that satisfy a particular filter, or it may be an electronic mail box.

Other applications of our work include data warehousing and cache management on the web. In data warehouses, which maintain views on data from various sources, the views need to be kept up to date by getting new data. A new version here is not a complete new copy, but an incremental change from the previous one. Subscription versus Demand correspond to the terminology of "push" versus "pull" used in this context. The results also apply for cache management on the Web.

We introduced complexity measures and analyzed retrieval protocols for two cost models, the request cost model and the time cost model. In these cost models we considered the Subscription, Demand, Demand with cache invalidation, and Sliding Window protocols. These protocols can be employed by a client to access an object at the digital library server. The first two protocols are static in the sense that an object is either cached or it isn't; the last two protocols are dynamic in the sense that an object may be cached at some time, and not cached at another time. The protocols are different in the two cost models, and they also vary depending on whether or not each read of the object must be consistent, i.e. access the latest version of the object.

It is important to emphasize that the set of cost models and protocols considered in this paper is far from being exhaustive. Many other scenarios are conceivable, and this paper should be regarded as a demonstration of our proposed approach to the problem of cost management in accessing digital libraries. For the rest of this section we summarize the results of our analysis.

First consider an object accessed in the request model using consistent reads. Assume that at any point in time the probability is $\theta$ that the next relevant access of the object is a write at the server (thus the probability is $1-\theta$ that the next relevant access of the object is a read at the client). If $\theta$ is fixed and known a priori, then the protocol that has the optimal expected cost depends on the costs of a read $rc$, a write $wc$, and an invalidation notification $ic$. These results are summarized in Corollary 1. If $\theta$ is unknown or it varies over time, then the Demand and Subscription static protocols are suboptimal. For the dynamic protocols, the average expected cost results are summarized in subsection 2.2.2. If the relevant requests are chaotic (i.e. do not follow a probabilistic pattern) and the objective is to reduce the worst-case cost, then again one of the dynamic protocols is optimal; the one with the lowest competitive ratio can be computed based on $rc$, $wc$ and $ic$ using Theorems 5 and 6.

Now consider an object accessed in the time cost model using consistent reads. Here the problem is to select between Subscription and Demand (possibly with cache invalidation) for each time slot; in contrast to the request cost model, the switch between the two protocols cannot occur in the middle of a slot, only at time-slot boundaries. This gives rise to a totally new set of concerns. The first problem is to determine the protocol with minimum expected cost for each time slot, assuming that we are given the number of expected relevant requests in a time slot. In subsection 3.1, we devise an efficient algorithm that determines the optimal policy for each time slot, such that the average cost per time slot is minimized. In subsection 3.2, we devise the Sliding Window algorithm for this model. Cache invalidation is combined with the Demand protocol in a straightforward manner (see subsection 3.3). Again, the issues are totally different than in the request cost model.

Finally, we consider an object in the request cost model using tolerant (or inconsistent) reads, i.e. reads that can tolerate an out-of-date version of the object. It turns out that straightforward use of Subscription and Demand cannot take advantage of such reads in order to reduce cost. Therefore, for this environment we proposed a hybrid mechanism between Subscription and Demand. In the previous scenarios, at any point in time the client is either on Subscription (and pays for the writes) or Demand (and pays for the reads); it may switch between Subscription and Demand periodically. In contrast, in the request cost model using tolerant reads, at any point in time the client is on Subscription for some reads and on Demand for other reads, depending on the tolerance of the read. The client pays for the Demand-reads, and for some of the writes. We called this the Static Divergence Caching (SDC). The first problem that we solved in this model is determining (for given probabilities of the relevant requests) the optimal refresh rate of SDC, i.e. optimal lower bound on the tolerance of the Subscription reads (equation 19). For the case where the probabilities of the relevant requests are unknown or they vary over time, we devised the Sliding Window algorithm for this model, called the Dynamic Divergence Caching (DDC) (subsection 4.2). We showed that for optimizing cost in the worst case the DDC algorithm is better than SDC (theorems 10 and 11). Finally we analyzed the DDC and SDC algorithms by simulations. We showed that tolerant reads improve the cost compared with nontolerant reads by a factor of two. We also showed that when the relevant probabilities are fixed but unknown, the cost of the DDC algorithm is almost as good as that of SDC with the optimal refresh rate (subsection 4.4.1). On the other hand, when the relevant probabilities vary over time, the cost of the DDC algorithm is 70% of the cost of the SDC algorithm having the optimal refresh rate. (subsection 4.4.2).

We believe that in the future information appliances will come equipped with a cost optimizer, in the same way that today computers come with a built-in operating system. Similarly customer agents searching for information may be equipped with similar optimizers. This paper makes the initial steps towards a theory and practice of cost management and optimization in accessing information. Such a theory and its implications may become critical for the information economies of the future. Much remains to be done in terms of new cost models, protocols, a unifying theory, and the development of cost management systems.

## REFERENCES

A. Chankhunthod, P. B. D. and et al. 1996. A hierarchical internet object cache. *Proc. of USENIX*.

A. Karlin, M. M. and et al. 1988. Competitive snoopy caching. *Algorithmica 3*, 1.

Carey, M. and Lu, H. 1986. Load balancing in locally distributed database system. *Proceedings of the ACM-SIGMOD , International Conference on Management of Data*. May.

Christodulakis, S. 1983. Estimating block transfers and join sizes. *Proceedings of the ACM-SIGMOD, International Conference on Management of Data*. May.

Christodulakis, S. 1984. Implications of certain assumptions in database performance evaluation. *ACM Transactions on Database Systems (TODS) 9*, 2 (June).

Crovella, M. E. and Carter, R. L. 1995. Dynamic server selection in the internet. *Proc. of the 3rd Workshop on the Architecture and Implementation of High Performance Communication Subsystems*.

Franklin, M. 1996. Client data caching. *Kluwer Academic Publishers*.

G. Popek, R. G. and et al. 1990. Replication in ficus distributed file systems. *Proceedings of the Workshop on Management of Replicated Data*. IEEE, Houston, Nov.

Gray, C. and Cheriton, D. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *Proceedings of the 12th International Symposium on Operating System Principles*.

Gwertzman, J. and Seltzer, M. 1996. World-wide web cache consistency. *Proceedings of the 1996 Usenix Conference*. January.

Irani, S. and Karlin, A. 1997. On online computation in approximation algorithms for np-hard problems. *D. Hochbaum ed.* PWS Publishing Company.

J. D. Guyton, M. F. S. 1995. Locating nearby copies of replicated internet servers. *TR CU-CS-762-95 Department of Computer Science*. Univ. of Colorado, Boulder, Feb.

J. Gwertzman, M. S. 1995. The case for geographical push caching. *Workshop on Hot Operating Systems*.

J. Kistler, M. S. 1991. Disconnected operation inn the coda file system. *Proceedings of the 13th International Symposium on Operating System Principles*. Pacific Grove, CA, Oct.

J. Sidel, P. A. and et al. 1996. Data replication in mariposa. *12th Int. Conf. on Data Engineering*. Feb.

K. Obraczka, P. D. and et al. 1996. A tool for massively replicating internet archives: Design, implementation, and experience. *Proc. of the International Conf. on Distributed Computing Systems*.

Levy, E. and Silbershatz, A. 1990. Distributed file systems: Concepts and examples. *ACM Computing Surveys 22*, 4 (Dec.).

Li, K. and Hudak, P. 1989. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Surveys 7*, 4 (Nov.).

Liu, C. and Cao, P. 1997. Maintaining strong cache consistency in the world-wide web. *Proc. of the International Conference on Distributed Computing Systems*.

M.J. Franklin, B. J. and Kossman, D. 1996. Performance tradeoffs for client-server query processing. *Proceedings of the ACM-SIGMOD, International Conference on Management of Data*. May.

Motwani, R. and Raghavan, P. 1991. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer 24*, 8 (August).

N. Adam, Y. Y. and et al. 1996. Strategic directions in electronic commerce and digital libraries: Towards a digital agora. *ACM Computing Surveys 28*, 4 (Dec), 818–835.

Nitzber, B. and Lo:, V. 1991. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer 24*, 8 (August).

P. Sistla, O. W. and et al. 1996. An architecture for consumer-oriented online database services. *Proceedings of the 6th International Workshop on Research Issues in Data Engineering: Interoperability of Nontraditional Database Systems (RIDE-NDS'96)*, 50–60. New Orleans, LA, Feb.

Rosenschein, J. and Zlotkin, G. 1994. Rules of encounter: Designing conventions for automated negotiation among computers. The MIT Press.

Srivastava, J. and Elsesser, G. 1993. Optimizing multi-join queries in parallel relational databases. *Proceedings of the Parallel and Distributed Information Systems Conference*. Jan.

Y. Huang, P. S. and Wolfson, O. 1994a. Data replication for mobile computers. *Proceedings of the ACM-SIGMOD, International Conference on Management of Data*, 13–24. Minneapolis, MN, May.

Y. Huang, R. S. and Wolfson, O. 1994b. Divergence caching in client-server architectures. *Proceedings of the third International Conference on Parallel and Distributed Information Systems (PDIS)*, 131–139. Austin, TX, Sept.

Zahariousdakis, M. and Carey, M. J. 1997. Hierarchical, adaptive cache consistency in a page server oodbms. *Proc. of the International Conference on Distributed Computing Systems*.