TR TR-PDS-1998-009 June 1998

Parallel & Distributed Systems group

Department of Electrical & Computer Engineering

University of Texas at Austin

Austin, Texas 78712

Cooperative Caching for Financial Databases with Hot Spots

Aman Sinha Craig Chase* Munir Cochinwala

Parallel and Distributed Systems Laboratory
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712

Abstract

We measure the performance of several cooperative caching policies for a database with hot spots. The workload consists of queries and append-only update transactions, and is modeled after a financial database of stock (historical) trading information.

We show that cooperative caching is effective for this application. We show that selecting the correct set of peer servers when servicing a cache miss is crucial to achieving high performance, and we demonstrate a greedy algorithm that performs close to optimal for this workload. We also evaluate several cache replacement policies and show that a 2^{nd}-chance algorithm performs best. In a 2^{nd}-chance algorithm, replaced pages are transferred to a peer server rather than being discarded. When a page is selected for replacement a 2nd time, the page is discarded.

Our results can be applied in the design of "proxy" servers for databases or web servers where a layer of proxy servers are used to scale the system performance.

1 Introduction

We study the performance of cooperative caching policies in a tiered database architecture. In this architecture, a set of proxy servers is logically located between clients and a centralized database. Scalable performance is obtained by allowing the proxy servers to process queries in parallel.[1] Updates to the database are buffered at the proxy servers but must eventually be committed to the centralized database. Obviously, the performance of such a system cannot be scaled beyond the point where the update bandwidth exceeds the capacity of the centralized database. However, workloads that are dominated by query traffic (for example, web servers) are well suited to this design.

Figure 1 illustrates the tiered system model. Clients interact with proxy servers (hereafter simply "servers") and cannot directly access the centralized (hereafter "root") database. Updates must eventually be resolved at the root database. A query is processed by the server where it is first received. Data is cached at the servers in a cooperative fashion. When one server requires data to process a query, and that data is not cached locally, it may obtain that data from one of its peers. The root database is contacted for data only as a last resort.

* Supported by Bellcore, NSF and Tandem

[1] Individual queries are processed sequentially, different servers process different queries in parallel.

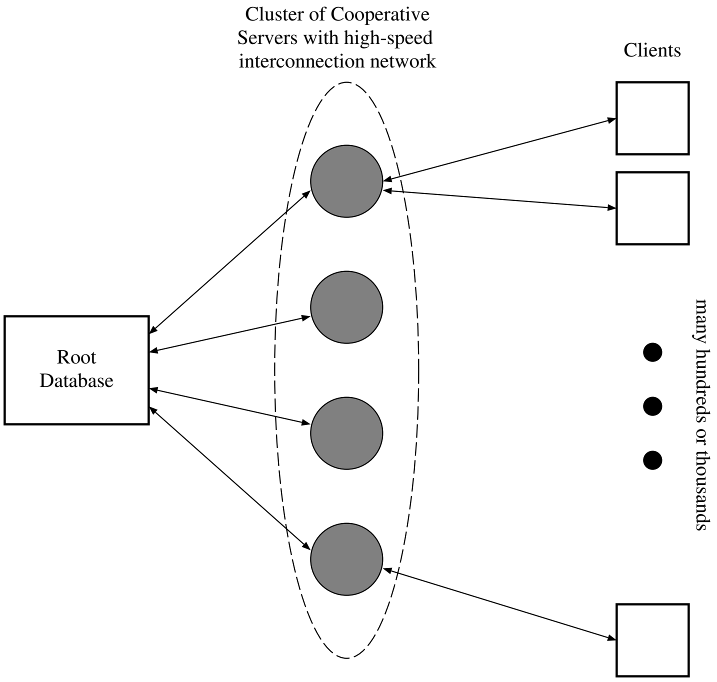

Figure 1: A Tiered Database Architecture

In this paper we identify three key issues for cooperative caching: 1) from which peer server(s) should data be fetched on a cache miss, 2) which pages should be selected for replacement, and 3) should replaced pages be transferred to a peer server? We describe and compare several cache management policies to study these issues. In each case, the server to uses only locally available information. Although better decisions can be made with global information, such policies can typically not be implemented in a scalable fashion — global information requires either a centralized directory, or constant broadcasts from each server as pages are cached and replaced.

We consider a workload based upon historical stock trading information, building upon the study described in [5]. Updates to the database consist of the symbol for the stock (*e.g.* IBM), the sales price, the number of shares exchanged and a time stamp. Note that this is an append-only database. Once a tuple is stored in the system it will not be modified. Queries in our workload request tuples based on a stock symbol and a time-stamp range. For example, "what is the average price of IBM over the last 20 minutes". Both queries and updates are subject to hot spots. We model hot spots by selecting roughly 10% of the stock symbols to be the source of 90% of the updates and queries. This distribution is typical of Wall Street trading data. Finally, the update rate for the system we model is several thousand updates per second (our experiments range from 8,000 to 10,000 updates per second). Query rate is an order of magnitude lower, with hundreds of queries performed per second (100-1000 queries per second). However, it must be noted that the data traffic and processing overhead is actually dominated by queries. An update adds only one tuple to the database and requires minimal processing. A query requests several hundred tuples and may require substantial computation to process. Under these assumptions, a tiered architecture is a reasonable design to achieve scalability without the complexity of distributed transactions.

Other examples of append-only databases with queries based on time-stamp ranges include network management databases. In network management databases (*e.g.* [2]), new data arrives periodically from various sources. Statistical measurements based on such data are made quite frequently. As in the stock market case, most queries are temporal in nature (*e.g.* "what was the average packet transfer rate through this router during the past 1 hour ?"), and hot spots are common.

The remainder of this paper is organized as follows: Section 2 discusses work related to memory management in both the operating systems and database domains. Section 3 describes the protocols used to manage the stock market data and servicing of a query. Section 4 discusses the policies for cache management. Our simulation methodology is described in Section 5 and the results of our simulation are presented in Section 6. Section 7 concludes the paper.

2 Related Work

Dahlin *et al* [6] performed a trace driven study of four cooperative caching algorithms. Their simulations showed that a coordinated approach, called N-Chance provided the best results and was also simple to implement. When a page to be replaced was the only copy in the global cache, it was forwarded to a random client. After N (a small constant) such transfers the page would finally be discarded. Dahlin found that for file system workloads, the best value for N was 2. We show that for our workload, the best performance results when N = 1 (*i.e.* pages can be transferred only once, and then must be evicted). In [9], Feeley *et al* provide a global memory management scheme based on *epochs*. At the beginning of each *epoch* (or time period), a centralized manager collects age information of the oldest pages in the client caches and distributes this information to every client. Using this information, the clients are able to choose a more appropriate target for eviction, based on an approximate global LRU replacement policy. A recent study on global memory management [14] explores the possibility of deviating from strict LRU policy. In that paper, Voelker *et al* found that if forwarding decisions take into account load information of the target clients, the global memory system performs better. Thus, forwarding to an idle client was more beneficial than forwarding to a randomly chosen client, or one chosen based purely on global LRU. The heuristics used in all of the above papers attempt to find out the exact details (facts) about a data page. In contrast to a fact based approach, a hint based cooperative caching scheme has been suggested in [12], wherein the heuristics minimize the overhead of finding out the exact details of a data page by propagating hints about it. The first copy of a page obtained from the file server is designated the *master* copy. Hints provide information about the *probable* location of the master copy and only when hints fail to locate the master copy, is it fetched from the file server. This approach substantially reduces the coherence overhead.

In the domain of client-server databases, global memory management of client memory has been explored. In a study by Franklin *et al* [10], clients forward data to each other to avoid disk accesses and an important objective is reducing replication between the clients' cache and the server cache. However, the server does not attempt to coordinate the contents of the client caches to reduce replication among the client caches. More recent work on memory management for a cluster of web servers has been reported in [13]. In that paper, Venkataraman *et al* model a read only database and propose heuristics to eliminate duplicate pages while making replacement decisions.

Simulation studies have been done for both object level and page level interactions in databases, see [3, 4]. In the Thor system [7], related objects are clustered in segments, with each segment consisting of subsegments of size k. When an object is requested, the entire subsegment is transferred. The value of k is determined adaptively.

Because of its high update rate, temporal queries and hot spots, the stock market application differs from many of the workloads used in the studies mentioned earlier in this section. In general, file systems display much more read traffic than write traffic. Both [6] and [12] use a write-through policy whenever a data page is modified. Client caches are kept consistent using a write-invalidate protocol. This methodology cannot be expected to perform well when there are a large number of updates (due to the need to multicast the updates to preserve cache coherence). Further, it is unclear whether the file system traces display significant hot spots. In [13], the authors model a read-only database, since web servers are primarily read-only. They use a zipfan distribution for modeling access to different files on the web server. In such a distribution, the 10th most popular data file receives 10 times fewer accesses than the most popular file. Although this has some similarity to the hot spots in the stock market application, the complete lack of update traffic makes it less interesting for our study. The workloads studied in [3] and [4] model object writes based on an object write probability. However, the ratio of write operations to read operations is relatively low. Their HotCold workload models hot spots, but in a limited manner. Essentially, each client has its own preferred hot region. There is no notion of a common hot and cold region spanning across all the clients. With a large number of clients, the "hot spots" will be evenly spread over the database, greatly mitigating any load imbalance problems.

3 Data Management and Query Servicing

The data in our database consists of stock trading information. Each stock has a canonical home server. All updates for a stock are forwarded to the home server for that stock. Stocks are assigned home servers using a hash function. The use of the hash function distributes the stocks across the servers and obviates the need for a global directory.

Tuples are clustered into fixed size pages (16 tuples per page) and replication is implemented at the granularity of a page to reduce the overhead of cache coherence. A page has a begin and end time stamp representing the oldest and youngest tuples respectively within the page. A directory maintained by the home server keeps track of the location of a page (or copies of it) in the global cache.

All queries in the stock market application are range queries, which contain the name of the stock and the low and high time stamps of a certain period of time (t_{low} and t_{high}). When a query for a stock S arrives at a server (called the querying server), the server first checks in its cache for pages that could satisfy the query. It then sends out this list of serviced pages (the set $serviced$) and the time stamp range t_{low}, t_{high} of the query to the the home server of that stock.

When a home server receives a query, it examines its directory to determine the set of all the pages in the given time stamp range. It immediately removes from this set any pages belonging to the set $serviced.$ It then generates a reply to the querying server containing any remaining pages that are available locally. The remaining pages must either be sent by a peer server where they are cached or retrieved from the root database.

The home server for a stock maintains knowledge (possibly stale) of which servers have cached copies of pages for that stock. When it cannot satisfy a data request locally, it uses this knowledge to construct a set of messages to send to peer servers, requesting them to forward the data to the querying server. The home server does not have exact knowledge of when the other servers evict cached copies of pages and thus may erroneously select a server to request for the data. In our protocol, this case is handled by having the peer respond to the home server with a NACK message. The home server then updates its local state and selects another server from which to fetch the data. As a last resort, the home server will contact the root database to retrieve any data that is not cached anywhere in the system.

It is possible that the same page is cached in multiple servers. By carefully selecting which server to contact when fetching pages, the total number of request messages can be reduced. For example, if a query requires two pages, A and B, and if server 1 has page A while server 2 has both page A and page B, then the optimal[2] strategy is to send a single request message to server 2 that requests both pages. Note that, in general, the problem of minimizing the number of request messages is NP-Hard (can be reduced to MINIMUM COVER [11])[3].

We consider three policies for fetching data from peer servers. The first policy, **Random**, simply selects any peer (among those that have copies of the required page) at random. The second policy, **Greedy** uses a heuristic to reduce the number of request-to-forward messages that are required. The Greedy policy is a simple algorithm with the following behavior:

1. For each server, construct a list of requested pages that are available in the server's cache.
2. Send a "request-to-forward" message to the server with the most requested pages. Request all pages available on that server. Remove this server from further consideration for this query.
3. Repeat step 2 until all pages have been requested.

We also consider an artificial policy, **OPT**. In OPT the minimum number of request messages are sent. We allow OPT to use exact global knowledge of the state of remote caches. This information is available trivially in a simulator but would require additional messages in an actual implementation. Thus OPT is not a realistic policy as it is modeled in our study.

For each of the three policies, at most one request message will be sent to each server. Requests are processed concurrently — *i.e.* we do not wait for a response to the first request before issuing additional requests.

4 Managing the Global Cache

In this section, we first describe the use of meta data for cache management. The cache coherence scheme for the pages is then described. Finally, we outline the cache replacement policies used.

4.1 Maintaining Page Details

In order to efficiently locate the pages without communicating frequently over the network, we use knowledge available locally at a server. Each page contains the following pieces of information:

- $clean/dirty$ – Recall that pages are created as a result of an update. Thus, all pages start out *dirty*. They are marked *clean* only when written to the root database.

[2] It should be noted that in this discussion we only consider the number of messages. The strategy that minimizes the number of messages will not necessarily produce the lowest response time in all cases. Load imbalances may produce situations where it is profitable to send more messages if that strategy also results in messages being sent to lightly loaded servers.

[3] The MINIMUM COVER problem states that given a collection C of subsets of a finite set S and a positive integer $K \leq\mid C\mid$, does there exist a subset $C' \subseteq C$ with $\mid C' \mid\leq K$ such that every element of S belongs to at least one member of C' ?

- is_master – Exactly one cached copy of a page is marked as the *master* copy. The master copy can never be discarded as a result of a cache replacement. Ultimately, this page must be committed to the root database. However, the master page can be transferred to a peer server.
- $evict\_count$ – this value gets incremented every time a page is transferred from one server's cache to another server.
- num_copies – this value represents the number of copies of the page in the cluster of servers.

The use of these flags is described in the following subsections.

4.1.1 Page Replication

When a page is first created (in the home server), $evict\_count$ is 0, num_copies is 1 and the page is in the $dirty$ state. This page is marked the *master* copy. Our policy is to replicate a page only as a result of a query. Assume that a query for a stock S arrives at a server (called the querying server). Then, as described in Section 3, the querying server will contact the home server first, and will eventually receive the relevant pages from one or more servers. Since the home server knows that the querying server is definitely going to cache these pages, it increments the value num_copies. Similarly, all the peer servers that forward a copy of the page to the querying server increment the value of num_copies in their own caches. Note that we do not guarantee (or require) that num_copies be an accurate count of the global number of copies. When selecting a page for replacement, a server may use the num_copies field as a hint to whether there are duplicates of this page at other servers.

4.1.2 Page Replacement

Recall that the smallest unit of replacement or data transfer is a page. Page replacement can occur in two cases:

1. When a new tuple arrives at the home server and there is no space left.
2. When a page arrives at any of the servers. This page could be the result of a query or can be the result of a page being transferred from some other server.

When a page is chosen for replacement (based on one of the cache replacement policies), its $evict\_count$ is incremented. If this value reaches a user-specified parameter $EVICT\_THRESHOLD$ and the page is the master copy, it means that the page is ready to be flushed to the database. If the value is below the threshold, the page is eligible to be transferred to another server in the cluster. The target server is chosen randomly.

4.1.3 Page Flushing

As described in the previous subsection, a master page is ready to be flushed (committed) to the root database when its $evict\_count$ reaches a threshold. At this point, the page is marked $clean$. Note that once a page is marked $clean$, it remains so forever, since a $clean$ page is immutable. If this page is fetched from the root database into the cache of any of the servers, this information assists in making decisions about future replacements of the page.

4.2 The Cache Replacement Policies

We first describe certain features that are common to all the cache replacement policies. In all of the following policies, whenever a page is chosen for replacement, we first check if it is the master copy. If it is not the master copy, it is always discarded. If it is the master copy, then one of 3 possible actions can be taken: 1) the page is discarded or 2) the page is transferred to another server or 3) the page is flushed to the root database. If $evict\_count$ is smaller than $EVICT\_THRESHOLD$, the master copy is transferred to another (randomly selected) server. Otherwise, clean master copies are discarded while dirty master copies are flushed to the root database.

The following cache replacement policies are compared and evaluated:

1. **Modified 2nd-Chance**: In [6], the best performing cooperative caching algorithm, N-Chance Forwarding, always chooses an LRU page in the client cache. Once the page is chosen, flags associated with the page are examined to see if the page is the last global copy. If it is the last copy, then the page is transferred to a random peer instead of being discarded. We call our version of this policy as the Modified 2nd-Chance because we found best performance occurred when N was 1 and because we never transfer a non-master copy of a page.

2. **Dup**: In this policy, whenever a replacement decision has to be made, we first check if there are any pages in the local cache that have duplicates elsewhere — *i.e.* if our num_copies hint has a value of 2 or greater. If more than one page in our cache is a duplicate copy, then we choose among them based on the standard LRU policy.

3. **Dup-Clean**: In this policy, we give first preference to replace pages that are duplicates *and* clean. LRU is used to break any ties.

4. **Clean**: A clean page is chosen, irrespective of how many duplicates it has. Again, the default policy to break ties is LRU.

As pointed out in [6], any algorithm where a page is replaced from one cache to another peer's cache could lead to a ripple effect, causing a chain of replacements. The Dup, Dup-Clean and Clean policies described above reduce the probability of selecting a ripple creating page because they all attempt to first find a page that can be discarded instead of transferred to another server. Our implementation of 2nd-Chance avoids this ripple effect by choosing the Dup-Clean policy for replacement whenever the incoming page has been transferred from another server.

Once a server has sent a page of data that was requested from it, it has two options: 1) it can update the access time of the page, so that this page moves to the bottom of the LRU stack (*i.e.* , it will take longer before the page is eligible for replacement), or 2) the access time of the page remains unchanged. The motivation for the first option is that it takes into account the global importance of the page, while the motivation for the second option is that it is in the spirit of a purely local LRU policy; if there have been no local references to the page, there is no need to move it anywhere in the LRU stack. Our simulation measurements demonstrated that the first option yielded lower overall response time compared to the second one. Hence, our simulator always updates the access time of a page, irrespective of whether it was accessed by a local query or a remote query.

5 Simulation Methodology

We study the performance of our cooperative caching policies through extensive simulation. We use an event driven simulator written in C++. The parameters of our simulation model are given in Section 5.1. We use a synthetic workload for our experiments (rather than a specific workload trace). Although not 100% representative of actual data, the synthetic workload permits us to vary key workload characteristics (*e.g.* update rate) and measure the resulting changes in performance.

In this study, we are primarily concerned with the efficacy of cooperative caching algorithms. Accordingly, we focus the simulations on modeling the cost to send and receive messages. The cost of processing the queries is limited to the cost of obtaining all the necessary data. The network interface at each server is modeled by a pair of FIFO queues. The input queue contains messages that were sent to this server, and the output queue contains messages waiting to be transmitted to other servers. Transmission costs are modeled with a fixed overhead and latency cost and a variable bandwidth cost per message. The bandwidth cost is proportional to the size of the message. This model of the network has been found to give reasonably accurate results [8]. Whenever the input queue is non-empty, the simulated server executes a task that models the overhead required to receive the message at the head of the queue. When sufficient processing time has passed, the message at the head of the queue is removed and processing can begin on the next message. Similarly, whenever the output queue is non-empty the server executes a task that models the overhead required to send the message at the head of the output queue. When the necessary processing has been completed, the message is removed from the output queue. If the message is being transmitted to another server, the message is inserted into the input queue of that server after an additional delay equal to the appropriate network transmission time (a function of the size of the message). Messages destined to clients are not modeled in our simulation.

In the stock market application, 90% of the updates and queries are for 10% of the stocks. To simulate this access pattern, we first create two separate lists — the hot list, which contains the symbols of those 10% of the stocks that are the most popular, and the cold list, which contains the symbols of the rest of the stocks. Every time an update or query is generated, the simulator randomly chooses one of the lists. With 90% probability it selects the hot list. Once a list is chosen, a stock from within that list is selected randomly (uniform distribution).

For updates, the only information in the message is the symbol for the stock and the current time stamp. The update message is then inserted into the input queue of one of the servers (selected randomly). When a new query is generated, the query message contains the symbol for the stock and a time stamp range. The higher time stamp of the query is always the current time, modeling a user requesting the most recent trade data for that stock. The lower time stamp is selected randomly with a uniform distribution such that the time range for this query is within a prescribed bound (specified as a simulator parameter). Note that the expected value of the time range is constant for a single simulation. Therefore, queries to cold stocks will have fewer tuples than queries to hot stocks. Note also that by appropriately selecting the time stamp range with respect to the update rate, we can control the expected value for the tuples per query.

Each server has a fixed amount of memory available as cache which we model explicitly. We do not model disk accesses on the servers. To initialize the cache of each of the servers, we follow a two step procedure. In the first step, only updates are generated and we directly insert these into the cache of the appropriate server until all the caches are filled (for 4 servers, each with 1MB cache, this amounts to about 32,000 updates). In the second step, both queries and updates are generated according to the prescribed arrival rates. Simulation proceeds in the normal fashion until 2,500 queries have been processed without collecting any performance statistics. At this point, the caches are considered warm. A simulation run consists of an additional 2,500 queries (and the associated number of updates) during which performance results are collected.

Parameter	Value	Comments
CPU MIPs	80	
Memory size	0.5-2MB	
Memory frame size	2K	
Network	60 MBits/s	Bidirectional FIFO

Table 1: Server parameters

5.1 Simulation Parameters

Table 1 gives the parameters of the servers. We chose a network bandwidth value that we believe is not unreasonable for the effective throughput using switched 100Mbits/second ethernet, or 155Mbits/second ATM networks. Our choice of the CPU MIPs values are based loosely upon the observation [1] that database applications are typically memory bound, effectively consuming about 3 cycles per instruction. Thus, for a 200 MHz superscalar processor, we are likely to get a speed of roughly 60-80 MIPs. Table 2 shows the miscellaneous parameters used and Table 3 shows the parameters of the workload.

Parameter	Value
Eviction Threshold	1
# Queries	5000
# Servers	4
Variable send cost	1000 instructions per page
Variable receive cost	1000 instructions per page
Fixed send cost	15000 instructions
Fixed receive cost	15000 instructions

Table 2: Miscellaneous Parameters

6 Experimental Results and Analysis

This section presents the results of our simulations and provides an analysis of the system behavior. In Section 6.1, we describe the experiments conducted to determine a convenient working set for our simulations. Section 6.2 describes the comparison between the three fetch policies, Random, Greedy, and OPT, exploring the sensitivity of these policies to various parameters. Section 6.3 provides an evaluation of the cache replacement policies. Finally, Section 6.4 presents the overall performance of the tiered architecture when used for the stock market workload described in this paper.

Parameter	Value	Comments
# Stocks	200	
Tuple size	128 bytes	
Page size	16 tuples/page	
Update rate	8000 to 10000	from Poisson distr.
Query rate	100 to 1000	from Poisson distr.
Percent hot set	10	

Table 3: Workload Parameters

Since one of our goals is to analyze the hot spot traffic, we first define our measure of popularity. The *popularity* of a stock is computed by taking the ratio of the number of queries processed for that stock over the total number of queries processed. There can be several stocks with approximately the same popularity value. To simplify and clarify the data, we combine stocks with similar popularities into the same data points. For example, if IBM has 5.9% popularity and GM has 6.1% popularity we'll report the data (in the graphs) from both of these stocks as having 6% popularity. We use 5 popularity "bins", spaced evenly at 0.4 percent intervals to represents the stocks between 4% and 6% popularity. The response time of queries to all cold stocks are are averaged and reported as a single data point in our graphs.

Recall that the number of tuples accessed per query is a function of the popularity of the stock (since the update rate scales with popularity while the time-stamp range is held constant). To minimize the bias this effect has on our simulation results we choose to report performance using *response time per tuple* rather than *response time per query*. The exception is for the overall system performance graph in Section 6.4 for which response time per query is the obvious metric of interest.

6.1 Working Set for the Simulations

The first experiment we conducted was designed to study the performance of the system in the presence of increasing query and update traffic. In particular, we wanted to determine at what load does a tiered system of four servers reach an unstable state. The cache size of each server was fixed at 1 MB.[4] The fetch policy considered was Greedy, and Modified 2^{nd}-Chance was used as the cache replacement policy. The time stamp range of each query was kept constant across different runs of the experiment. The query arrival rate was varied from 100 to 900 and the update arrival rate from 8500 to 10000 updates per second. We recorded the average response time per tuple over the range of query and update arrival rates. The results of this experiment are shown in Figure 2.

For higher update arrival rate (9500 and above), the system quickly reaches an unstable state, at relatively low query arrival rates. For these update arrival rates two effects are noteworthy. First, since the time-stamp range of each query is fixed, the average number of tuples per query for higher update rates is going to be large. The number of tuples per query is slightly less than one tenth of the update rate (expressed as updates per second) in this experiment. The increased volume of data transfer places greater load at each server in terms of higher processing costs for each page of data sent and received. Second, the higher rate of update arrival forces existing tuples out of the local cache much faster. This further implies that inter-query data

[4]This value of cache size is found adequate for our workload. An increase in cache size from 0.5M to 1M caused an increase in cache hit ratio from 45 % to 87 %. Further increase in cache size produced relatively small increase in cache hit ratios.

locality is going to be low (new queries cannot take advantage of data fetched by the older queries). The effect of these factors is that the cache hit ratio is lower at high update rates. The cache hit ratio at an update rate of 10000 was poor – from 64 % to 76 %. The cache hit ratio for an update rate of 9500 was in the range of 74 % to 87 %. For lower update arrival rates (9000 and 8500), the cache hit ratio was in the range of 80 % to 92 %.

The graph shows that the response time characteristics for update rates of 9000 and lower are significantly better than for update rates of 9500 and higher. For our experiments, we use an update rate of 9000, since this yields a reasonable value of tuples per query (about 840) and is stable for the desired range of query arrival rates ($\approx$500 queries / second).

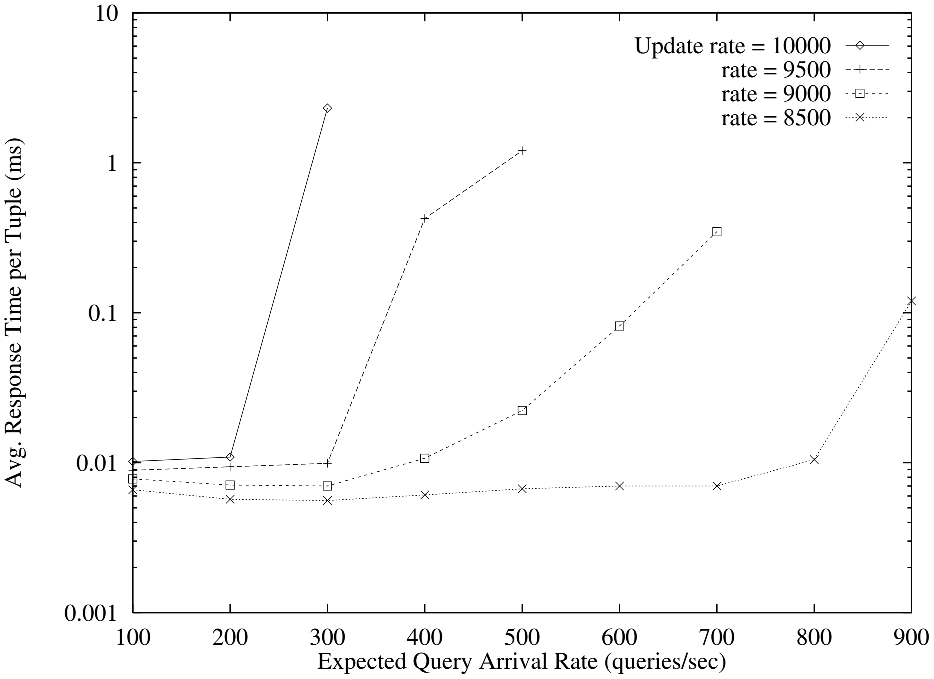

Figure 2: Effect of query arrival rate on the response time, for different values of update rates. Update rates are given in updates per second

6.2 Comparison of Fetch Policies

When a cache miss occurs for a page that is cached at two or more remote servers, the home server must decide which server to contact for forwarding the page. As stated in Section 3, the following two fetch policies are studied: **Greedy** and **Random**. For comparison purposes, we include an artificial third policy called **OPT**. This policy uses exact global information to determine the minimal number of messages that can be sent to fetch all of the required data. The parameters that are likely to affect the performance of these policies are: 1) popularity of a particular stock, 2) query arrival rate, and 3) size of a query. In the subsections that follow, we report experiments that measure the sensitivity of the response time performance to each of these factors.

6.2.1 Popularity

Popular stocks are more likely to be replicated at multiple servers. The degree of replication will impact the response time characteristics of queries under the different fetch policies. Figure 3 shows the variation of the average response time per tuple for different values of popularity. The update rate was 9000 updates per second and the query rate was 500 queries per second. The Greedy fetch policy dramatically outperforms Random for all values of popularity. The response time averaged across all values of popularity was 40 % better for Greedy as compared to Random. This large difference is attributable to the increased number of messages sent by Random. Not only must the requesting process send the message (and receive a response) but the target server must also process the request — taking processing time away from that server's normal query processing activities. Note that Greedy performed very close to the optimal policy for both hot and cold stocks.

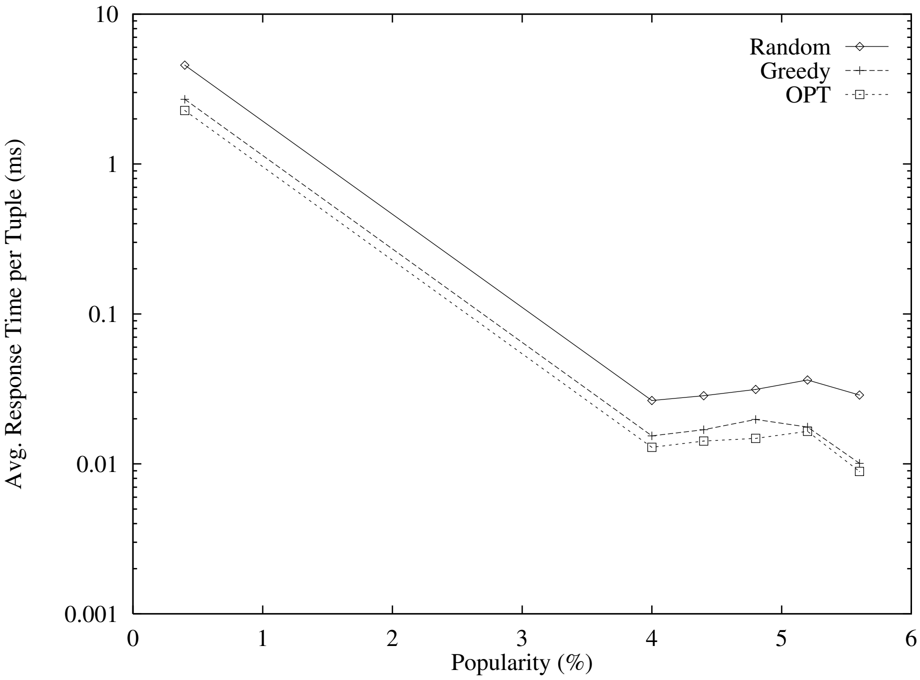

Figure 3: Performance of fetch policies for different values of popularity. Cache replacement used: Modified 2^{nd}-Chance

6.2.2 Query Arrival Rate

This subsection studies the effect of variation in query arrival rate on the average response time per tuple of the queries. The update arrival rate was fixed at 9000 updates per second. Increase in query arrival rate is likely to increase the load at each server, and thus will impact the response time behavior of the fetch policies. Figure 4 shows the results of this experiment. Note that with a low query arrival rate, all of the policies show similar performance. For higher query rates, the advantages of Greedy and OPT become very significant. There are two causes for this behavior. First, at low query arrival rates (with respect to the update rate) the probability that a single page is cached at multiple servers is low. Therefore, Random has fewer opportunities to make a "bad choice" when selecting a peer server from which to fetch data. Second, at low query arrival rates (with respect to the network speed and processing rates) there is surplus capacity in the cluster. Input queues at the servers remain small, and the additional delay caused by sending additional messages is relatively minor. However, as the system becomes more heavily loaded, input queues grow and the impact of unnecessary messages in the system becomes significant. Note that Greedy again performs fairly close to OPT in this study.

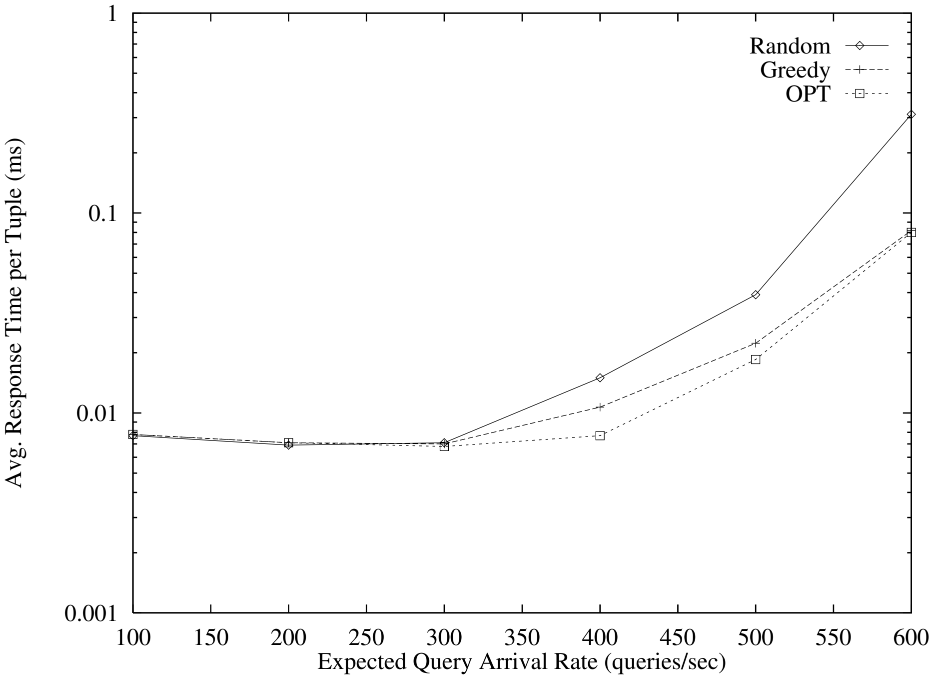

Figure 4: Performance of fetch policies with variation in expected query arrival rate

6.2.3 Size of Query

The next experiment that we conducted was designed to study the effect of increasing query size on the overall average response time per tuple. Keeping the update rate constant at 9000 updates per second and the query rate constant at 500 queries per second, we varied the number of tuples retrieved by each query. We control the number of tuples by increasing (or decreasing) the time stamp range for each query. The results of this experiment are shown in Figure 5. As was the case in Figure 4, at low loads the fetch policies show similar performance. As the load increases, Random begins to show poor performance while Greedy and OPT remain similar. However, at very high loads, the policies appear to converge. The reason for this behavior is that at very high loads, bandwidth costs begin to dominate the message communication costs. Even with the Random policy, the number of pages sent per message will increase, simply because the number of pages per query is increasing while the number of servers (four) remains constant. Since all three policies fetch the same total volume of data, the total cost of fetching the data becomes less dependent on the number of messages used.

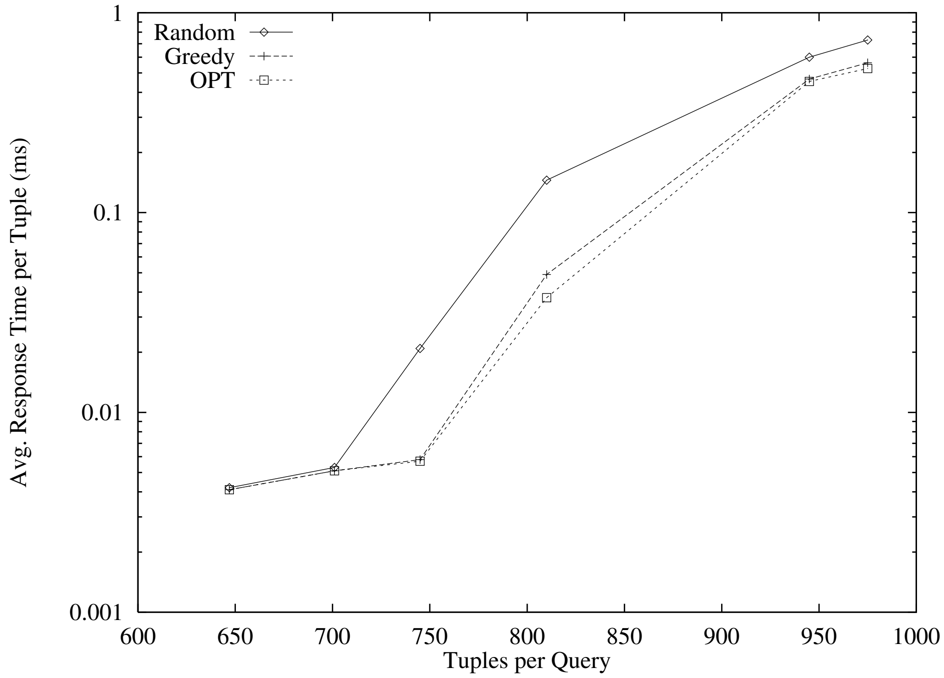

Figure 5: Response time as a function of query size

6.3 Comparison of Cache Replacement Policies

Figure 6 shows the performance of the cache replacement policies, for an update rate of 8000 updates per second. The cache policies compared were (see Section 4.2) Dup, Clean, Modified 2^{nd}-Chance and Dup-Clean. Dup performs the worst among all policies, followed by Clean. The Modified 2^{nd}-Chance and Dup-Clean policies perform almost equally well for most values of popularity and are the superior ones among all policies. Since Dup-Clean gives preference to duplicate and clean pages for replacement, it is most likely to choose a page that can be discarded. Dup, on the other hand, could end up choosing a master page for replacement, since the master page of popular stocks is likely to have been duplicated. Since the master page cannot be discarded, Dup incurs the extra overhead of forwarding or flushing the page.

For a higher update rate, Modified 2^{nd}-Chance outperforms all other cache polices. Figure 7 shows the performance of the policies for an update rate of 9000 updates per second. Since Dup performed poorly at this rate, it is not plotted on the graph. The reason both Dup-Clean and Clean are outperformed is that both these policies may discard a page that was fetched recently. The disadvantages of this approach are more pronounced at a higher update rate because the rate of replacement also increases at higher update rates.

6.3.1 Selecting the Value of N in N-Chance

We found that a second chance algorithm outperformed other values of N with our workload. Figure 8 shows the performance of two Modified N-chance configurations. Note that the value of N in the figure is the eviction threshold used in the experiment. Thus when N = 1, pages have a second chance (can be transferred once) before being replaced. A conventional LRU algorithm corresponds to the case where N = 0. Figure 8 clearly shows that performance is significantly worse when the threshold is raised to 2. The

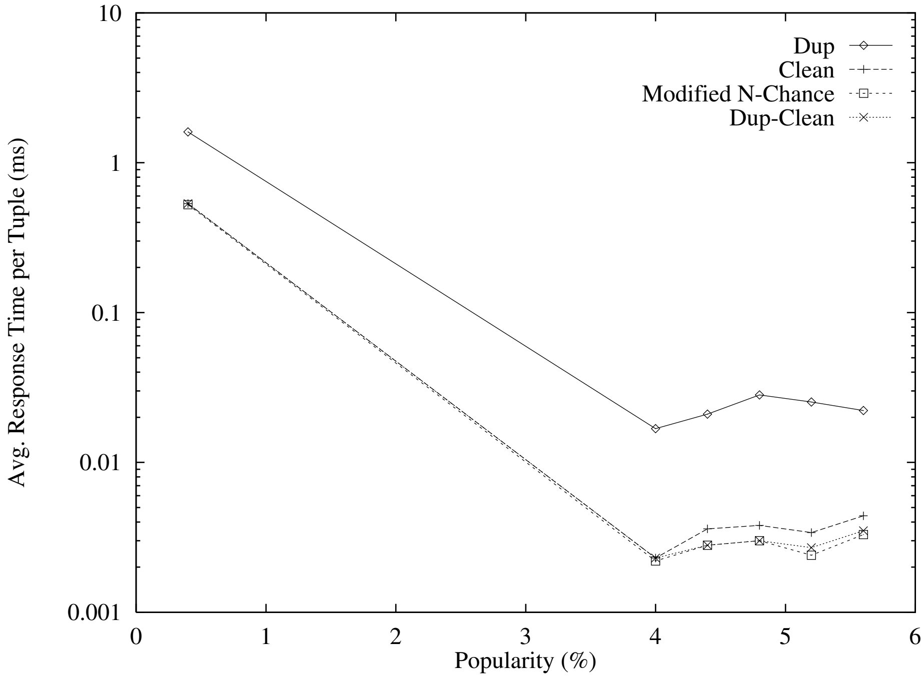

Figure 6: Comparison of Cache Replacement Policies. Update rate: 8000

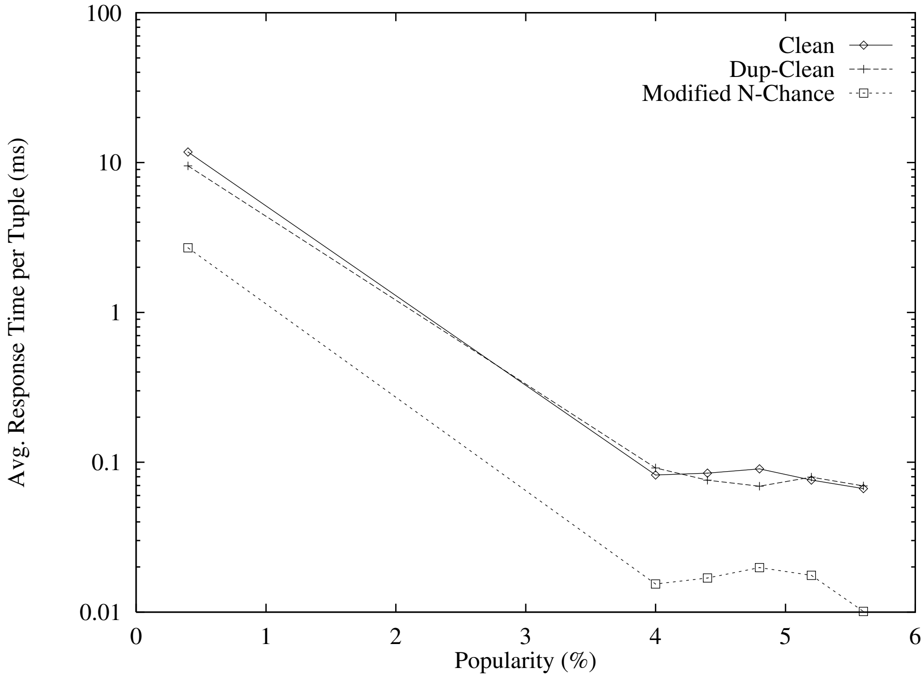

Figure 7: Comparison of Cache Replacement Policies. Update rate: 9000

reason for the lower performance is the increased number of ripples caused when an old page is replaced. Since as tuples age they naturally become less active in our workload (queries access the most recent data), particularly old pages have little value. A high eviction threshold gives old pages extra chances to remain cached, replacing other more useful pages, and lowering the overall cache hit rate.

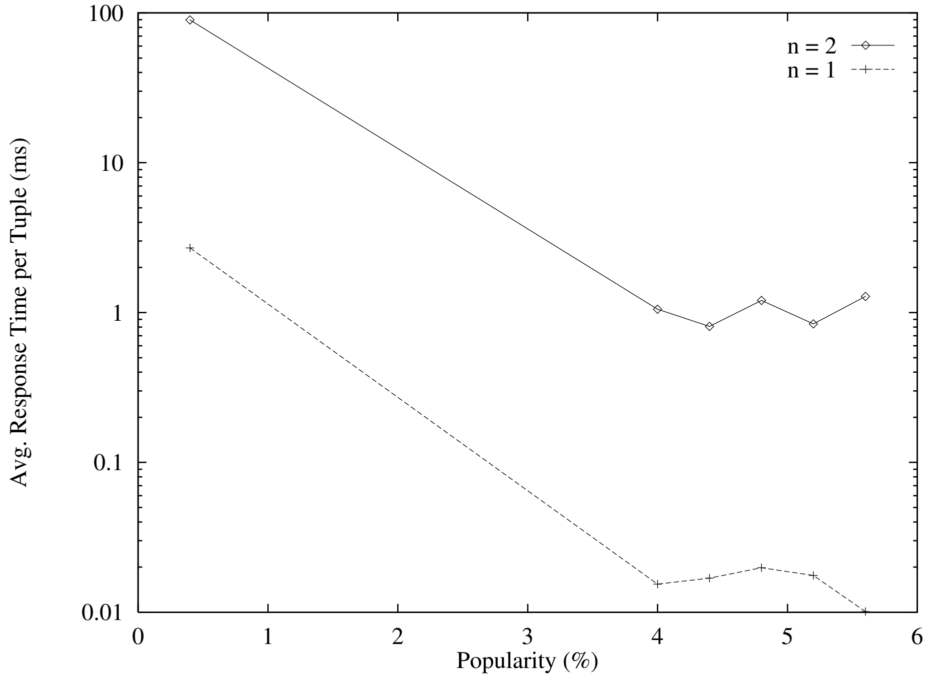

Figure 8: Comparison of N-Chance Policies for Different Eviction Thresholds (N). Update Rate: 9000, Query Rate: 500

6.4 Overall Performance and Quality of Service

Figure 9 shows the mean and 95% confidence interval for query response times plotted against popularity. The fetch policy used is Greedy and the cache replacement policy is Modified 2^{nd}-Chance. Note that we consider response time per query in this section (rather than response time per tuple). The 95% confidence interval is within a few percentage of the mean value in most cases, indicating fairly uniform quality of service under for these policies. Note also that the graph is relatively flat, showing that reasonable response time is provided to most stocks. Hot stocks actually experience lower response times, due to the higher cache hit rates for these stocks.

7 Conclusion

In this paper, we describe a database architecture where proxy servers engage in cooperative caching in an effort to reduce the global miss rate during query processing. We show this architecture is effective in the case of append-only transactions and a workload containing both significant query traffic and significant update traffic. We perform extensive simulation of this system based upon the traffic patterns of a stock

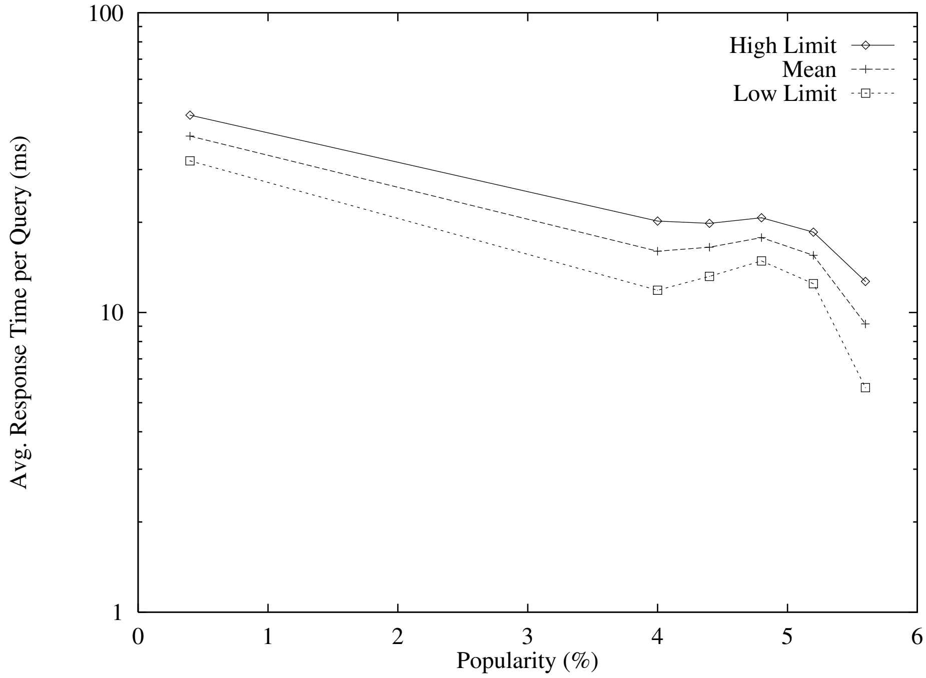

Figure 9: Response Time of Hot and Cold Stocks. The three lines represent the mean and the 95% confidence interval around the mean

trading database. Our study considers several cache management policies and measures the performance (query response time) of each policy. Our primary results are:

- **Selecting an appropriate set of peers to from which to fetch data during a cache miss is critical to achieving high performance.** Since the same data may be simultaneously cached at multiple peers, a server has a choice when it fetches data. We show that a simple greedy algorithm significantly out performs a naive random fetch policy. In fact, our greedy algorithm is shown to yield performance close to the optimal.

- **Page replacement policies must be cooperative.** The best performing page replacement policy was a "2^{nd}-chance" policy. Specifically, on the first replacement for a specific page it is beneficial to transfer the page to another server (chosen randomly) rather than discarding the page. When that page is subsequently selected (again) for replacement, discarding the page becomes the profitable choice.

- **Hot spots emphasize the need for cooperative caching algorithms.** When the database contains hot data, that data is likely to remain in the global cache even when it is replaced from local caches. Cooperative caching reduces the time needed to access this data.

Several areas remain to be explored in future work. Our study did not consider fault tolerance. A significant performance benefit can be achieved by allowing proxy servers to buffer transactions. However, failure of one or more proxies must not be allowed to corrupt the data base. We did not model the cost of logging updates to stable storage in our our simulations. Our study only considered append transactions. More general updates would introduce cache coherence complexity into the model. Future studies could evaluate the effectiveness of cooperative caching in the presence of arbitrary updates. Finally, we considered a very specific workload in this study, modeled after a stock trading database. More general workloads also should be studied, particularly in the context of internet web and proxy servers.

References

[1] Transaction processing council; http://www.tpc.org.

[2] Ilsoo Ahn. Database issues in telecommunications network management. In *SIGMOD*, pages 37–43, 1994.

[3] Michael J. Carey, Michael J. Franklin, Miron Livny, and Eugene J. Shekita. Data caching tradeoffs in client-server dbms architectures. In *ACM SIGMOD*, 1991.

[4] Michael J. Carey, Michael J. Franklin, and Markos Zaharioudakis. Fine grained sharing in a page server oodbms. In *ACM SIGMOD*, 1994.

[5] Munir Cochinwala and John Bradley. A multi-database system for tracking and retrieval of financial data. In *20th VLDB Conference*, pages 714–721, 1994.

[6] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *1st Symposium on OSDI*, 1994.

[7] B. Liskov et. al. Safe and efficient sharing of persistent objects in thor. In *ACM SIGMOD*, pages 318–329, 1996.

[8] Ed Lazowska et. al. File access performance of diskless workstations. *Transactions on Computer Systems*, August 1986.

[9] Michael J. Feeley, William E. Morgan, Frederic H Pighin, Anna R. Karlin, and Henry M. Levy. Implementing global memory management in a workstation cluster. In *15th SOSP*, pages 201–212, 1995.

[10] Michael J. Franklin, Michael J. Carey, and Miron Livny. Global memory management in client-server dbms architectures. In *18th VLDB*, pages 596–609, 1992.

[11] Michael R. Garey and David S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[12] Prasenjit Sarkar and John Hartman. Efficient cooperative caching using hints. In *OSDI*, 1996.

[13] Shivakumar Venkataraman, Miron Livny, and Jeffrey Naughton. Memory management for scalable web data servers. In *International Conference on Data Engineering*, pages 510–519, 1997.

[14] Geoffrey M. Voelker, Herve A. Jamrozik, Mary K. Vernon, Henry M. Levy, and Edward D. Lazowska. Managing server load in global memory systems. In *SIGMETRICS*, pages 127–138, 1997.