University of Alberta

Performance Evaluation of Optimistic Cache Consistency Algorithms for Web-based Electronic Commerce Applications

by

Justin Vimal Amalraj

Technical Report TR 00-15
May 2000

DEPARTMENT OF COMPUTING SCIENCE
University of Alberta
Edmonton, Alberta, Canada

Abstract

Caching is a simple and effective way to provide faster access to information on the Internet, and it has been used effectively to improve the performance of E-commerce servers. However, there is a growing concern by today's Web service providers to ensure that the data cached at a remote client is up to date with the current value in the server. A unique aspect of cache consistency for E-commerce applications is that most of the applications on the Web can tolerate some degree of inconsistency between the clients and the server.

In this work, we propose an Optimistic Web Cache Consistency algorithm which exploits the tolerance exhibited by E-commerce applications, and which is suitable for E-commerce transactions on the Internet. The optimistic algorithm relaxes consistency constraints but, at the same time, bounds the staleness of data viewed by transactions, and brings about a generalized correctness criteria for the execution of transactions on the Web.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Recent growth in Web-based commerce has led to increasingly active research in the design of high-performance electronic commerce Web sites. Today there is a growing interest in providing caching support for E-commerce applications on the Web - as caching is a simple and effective way to provide faster access to information on the Internet. Caching refers to storing Web objects accessed by clients at places through which the client's request passes. Hence, if the request is for an object that is stored in the cache, then the object is supplied from the cache rather than the server. A central issue of caching is *cache consistency*, which refers to the mechanism by which the server ensures that the data cached at a remote client is up-to-date with the current value in the server. The main objective of our work is to develop a cache consistency algorithm for Web-based electronic commerce (E-commerce) applications, and to evaluate its performance for Web workloads. Specifically, we focus on developing a relaxed cache consistency scheme for E-commerce types of applications, as many of the Web applications are able to tolerate some degree of inconsistency.

1.2 Problem Definition

Over the years Web caching has become a topic of increasing importance. The use of Internet caches has become a cheap and effective way to improve performance for Internet users. Caching attempts to improve performance in three ways [RCG98]: First, caching attempts to reduce the user-perceived latency associated with obtaining Web objects. Latency can be reduced because the caches are typically much closer to the client than the server, and hence cache hits result in faster delivery of data when compared to obtaining it from the server. Second, caching attempts to lower the network traffic from the Web servers. Network load can be lowered because some objects are served from the cache rather than from the server, thus preventing repeated trips to the server for data. Finally, caching can reduce the service demands on the server since cache hits prevent repeated requests for cached data, thus reducing the amount of processing that has to be done at the server. Caching is currently one of the main ways to reduce latency, server load, and bandwidth for delivering Web contents. Caching helps E-commerce servers achieve better performance, which is considered to be vital for the success of an E-commerce company.

Caching for E-commerce effectively causes migration of copies of objects from servers to the clients or to intermediate caching servers. This form of replication brings about the challenge of maintaining consistency of the data cached between the clients and the server. Generally, Web cache consistency schemes are referred to as *weak consistency* schemes as most of the schemes focus primarily on providing some degree of consistency for the users. They do not provide any guarantees on the freshness of data viewed by clients. Therefore, Web cache consistency schemes are not suitable for E-commerce transactions as they do not satisfy the usually strong consistency requirements of E-commerce transaction processing.

The goals of E-commerce transactions on the Web create a different set of demands for a cache consistency scheme. One of the primary concerns is that a cache consistency scheme should preserve correct execution of transactions and prevent the corruption of data at the server. Moreover, an E-commerce transaction requires transactional guarantees for correct execution as it involves transactions that could update the value of a data item at the server which may have been cached by other clients. Cache consistency algorithms for client-server databases provide strong guarantees on consistency and correct execution of applications, and can provide the correctness guarantee required by E-commerce applications. However, database cache consistency schemes are considered to be too restrictive, even for many database applications, and the transactional guarantees provided by them for correct execution of applications are considered to be very strong for the Internet.

Many Internet services do not need the strong consistency and guarantees of correctness provided by the database schemes for transaction processing. Moreover, it is widely acknowledged that strong consistency is not the primary requirement for Web applications [Fox98]. The general notion in the design of Web applications is to sacrifice some consistency to achieve greater availability and performance. A Web server is said to be available, if all users request are satisfied by the Web server. However, it can be seen that databases ensure strict consistency by sacrificing availability - which is contradictory to the requirements of the Internet. Hence, database cache consistency schemes are too restrictive for many Internet services.

A unique aspect of cache consistency for E-commerce applications is that most of the applications on the Web can tolerate some amount of inconsistency between the clients and the server [ZSJ00]. Hence, a suitable cache consistency scheme that can exploit the tolerance exhibited by E-commerce applications is required. In this report, we address this issue of tolerance exhibited by Web applications for relaxing database consistency schemes, and we attempt to provide application-specific strong consistency guarantees for E-commerce applications on the Web. By relaxing consistency to some tolerable bounds, we attempt to overcome some of the performance problems faced by E-commerce applications on the Web.

1.3 Motivating Examples

Today, there is a growing recognition of the need for E-commerce sites to provide better service to online customers. Cache consistency is one of the features that is desired by many customers on the Internet. Our work was motivated by observing the consistency requirements of a number of popular Internet applications. In this section we look at some of the most popular of those applications and the consistency requirements they require.

Online Shopping Store

In this application, the online customer receives a catalog of all the items s/he may need and adds the item s/he is interested in to a shopping cart. When s/he wants to check out, the client submits an order for all the items placed in the shopping cart. However, not all orders are accepted by the server; there are cases in which a requested item has been sold out by the time the customer checks out. Moreover, the customer usually spends a considerable amount of time making their decision to purchase an item and, hence, most of the transactions are lengthy. Since it is of some concern to the customer if their transaction does not materialize, there is a need for the server to send notifications regarding the availability of the item, and any price changes, to the client before s/he checks out. This minimizes the number of aborted transactions and helps better satisfy the demands of customers. The server can also provide hints about the depletion rate of an item from its inventory so that customers can make an immediate decision to reserve or purchase an item.

Reservation of Travel Tickets

The semantics of this application are similar to purchasing a product from an online store. In this case the customer purchases travel tickets. The customer profile may suggest that s/he likes to travel on certain airlines and wishes to monitor the prices and availability of tickets for those carriers. The customer might want to collect a large amount of data regarding all available options before making a decision, so any changes in price and availability need to be indicated to the customer. It is common for customers to book some tickets when they are available, and free them later when they are no longer needed or when they get a better deal. Immediate notification of cancelation of booked tickets to customers who are interested in those tickets is therefore important, and therefore there is a great demand for accessing the latest information, as reflected at the server.

Stock Quotes

Today's stock quote tickers periodically pull or receive data from the server. Some clients receive updates at intervals of several seconds, while other clients receive updates at intervals of several minutes. Up-to-date information about quotes is critical, especially during periods of heavy server load, because that signifies an important market fluctuation. Many shares will be traded and the demand to get the latest information will be high. Each client has their own customized list of quotes which can be in the order of tens to hundreds, and not all of the quotes will change at the same time. The customer makes a decision to purchase/trade shares based on the quotes s/he views for each share. Hence there is a concern for freshness in the data viewed by the clients for each share.

Online Auction

Online auction stores, such as *www.egghead.com*, auction items which come with a current bid, a closing time, and bid increments. Any customer can place a bid before the closing time, and the highest bid gets the item. A customer may bid for a number of items, and would like to monitor all the bids for the items s/he is interested in. Monitoring bids is critical - especially during the closing time when a large number of bids is typically received by the server, and all interested clients would like to get the most up-to-date information about those bids. When placing their bid for an item, a client does not want it to be rejected because they accessed stale data. If the client is tracking a large number of items, not all items will have the same closing time, so it is vital to send immediate notification to the customer of changes in bidding prices.

In all of these applications we observe that the consistency of data viewed by clients is a fundamental requirement, but not all of them require the same degree of consistency. Each application has its own tolerance in consistency for correct execution of the application. Hence, performance enhancement can be achieved by

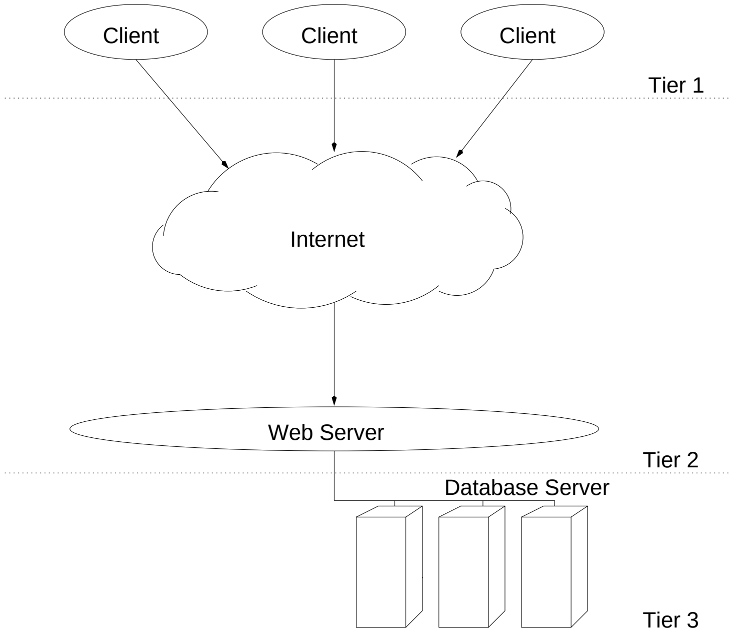

Figure 1.1: E-commerce System Architecture (adapted from [Men99])

providing application-specific consistency guarantees.

E-commerce System Architecture

In this section we look at a generic model for E-commerce that would suit the online shopping store applications described earlier.

An E-business site has a three-tiered architecture, as shown in Figure 1.1. The first tier comprises the clients who initiate the transaction. The transaction involves requests for data and updates on certain data items. The client consists of browser software running on the client workstation. The browser maintains a client cache (browser cache), which caches the data used by the transaction. All requests pass through this cache, and requests which are not satisfied by the cache are passed on over the Internet to the server.

The second tier consists of the Web server. The Web server is responsible for maintaining session information as well as maintaining the user profile and the state of the client's connection. It is also responsible for generating a dynamically featured and/or formatted document from the data repository, based on the user's request and profile, as well as user authentication to enforce secure access to information.

The third tier of the architecture is the back-end data repository - usually the database application server. The application running in this tier returns data to the Web server, which in turn generates a dynamic page and returns it back to the requested user.

E-commerce Transaction Model

An E-commerce transaction can be characterized based on the delays associated with the underlying network and the protocols. In this case we look at the model of an interaction that an E-commerce transaction has with the server, and identify potential sources of delays associated with it. E-commerce transactions use the HTTP protocol to interact with servers, and the HTTP protocol has a request-response model of interaction with Web servers. We adhere to the E-commerce system architecture we described in the previous section. The anatomy of an E-business transaction [MA98, Men99] is shown in Figure 1.2. The main components are the browser, the Web server, and the database application server, which form the three tiers of an E-commerce service. In Figure 1.2, time is shown advancing from top to bottom; the thick bars represent processing intervals at each component, as explained below:

- *Browser processing time (t2-t0)+(t11-t9)*: The user requests a data item and the browser looks in its local cache to see if the requested data is cached.

  - If the data item is cached at the client, then the response time is given by Ct.
  - In case of a cache miss:
    - The browser first asks the domain name server (DNS) to map the server host-name to an IP address; usually the client side component avoids this step by storing the information locally.
    - The client opens a TCP connection to the Web server.
    - The client sends an HTTP request to the Web server.

  Upon receiving a reply, the browser formats the data and displays it to the user.

- *Wide area network time (t3-t2)+(t9-t8)*: This is the network-imposed delay associated with the traversal of the information from the clients to the server, and vice versa. These delays depend on the router, switches, bridges, relays, network congestion, and the number of hops along the path between the client and the server.

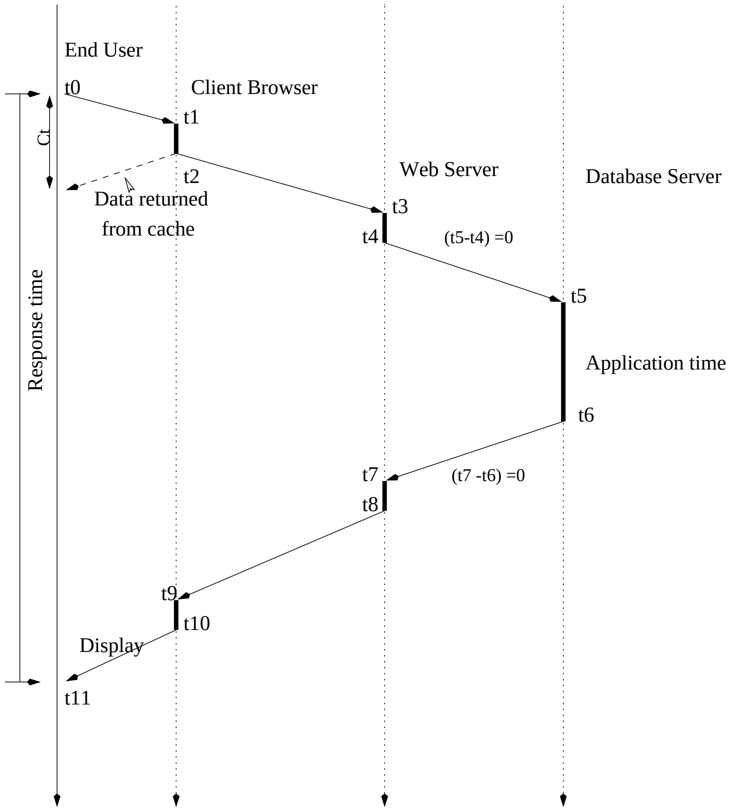

Figure 1.2: Anatomy of E-commerce Transaction Model (adapted from [Men99])

- *Web Server processing time (t4-t3)+(t8-t7)*: This delay is associated with the Web server parsing the HTTP request and sending a request to the database server. The server also receives the result from the database and generates a dynamic Web page which it sends to the user.
- *Database server processing time (t6-t5)*: This time represents the execution time of the client's transaction on the database.

Since in our system model we have reduced the three-tier architecture to a two-tier one, we assume that the delay *t5-t4* and *t7-t6* to be negligible.

1.4 Research Scope

This report explores Web cache consistency schemes and database transactional cache consistency schemes for designing a generic application-specific cache consistency algorithm that can more efficiently handle transactions similar to E-commerce transactions on the Web. In particular we focus on providing application-specific strong consistency guarantees, and target the scheme for a generic model of E-commerce applications on the Web. We consider scalability and availability as primary requirements for the design of our algorithm.

1.4.1 Cache Consistency Algorithm

Current cache consistency algorithms for the Web are considered to be *weak* as they do not provide any strong guarantees of consistency, as is required for E-commerce transactions. On the other hand, as argued earlier, database cache consistency algorithms provide strong consistency for transactions, but are not suitable for the Web environment. Availability is a primary concern for Web services, but most of the database consistency schemes sacrifice availability for strong consistency.

Our work basically focuses on developing a relaxed consistency scheme, as most Web applications are able to tolerate some degree of inconsistency. In particular, we try to bound the staleness of data viewed by transactions while at the same time bringing about a generalized correctness criteria for correct execution of transactions for applications that can tolerate some kind of inconsistency. We try to alleviate the bottlenecks that are inherent to transaction processing in order to achieve greater availability and concurrency. Staleness of a data item refers to the amount by which the value of a data item viewed by a transaction deviates from the current value at the server.

In this report, we propose an *optimistic* cache consistency algorithm for E-commerce applications that relaxes consistency and at the same time bounds the inconsistency of data viewed by clients to tolerable limits set by the application designer. Optimistic schemes use fewer synchronous round-trip requests than lock-based schemes [Gru97], and they are less sensitive to latency involved in message transmission. Optimistic schemes are also found to scale better when the clients and servers are widely separated as is usually the case with the Internet. These are the primary reasons for our choice of an optimistic scheme for cache consistency. Moreover, it can also be seen that all of the Web-cache consistency schemes are optimistic as they are less sensitive to the nature of the network. An optimistic algorithm makes the system very available. Since an optimistic algorithm does not have *read/write* or *write/write* conflicts, the server can readily satisfy a client's request. Lock-based schemes tend to block transactions upon detecting a conflict with some on-going update transaction. This kind of blocking reduces the availability of the system, and is also injurious to a Web site as denial of service is not an acceptable option for most Internet services. Denial of service refers to the server rejecting the users request for data upon detecting conflicts with on-going concurrent transactions.

The proposed optimistic scheme defers all update action until commit time as clients usually quit a transaction halfway if they experience bad service. In some cases, if part of the transaction is committed and the client decides to quit its transaction halfway, then in a lock-based scheme this scenario potentially affects other clients' transactions. Hence, compensating transactions or special rollback schemes may be needed in order to recover from the client's unexpected exit. There is also the case of a client being blocked because of another client's conflicting transaction, but if the conflicting transaction terminates before committing, then the blocking of other transactions was in vain. Hence deferring updates until commit time appears to be a good design property for the cache consistency algorithm.

All consistency actions are piggy-backed, thus reducing the overhead associated with message transmission. The optimistic scheme proposed in this report is a reasonable algorithm for E-commerce transactions and it better suits the requirements of the Web. We study the performance of three possible variants of the algorithm based on the update delivery mechanism involved-namely *invalidation*, *propagation*, and dynamically (*hybrid*) choosing between the two. The main performance study evaluates the scalability of these three strains; the secondary study evaluates of the impact of client cache size. This report also makes a detailed investigation of cache consistency schemes for both the Web and database domains.

1.4.2 Workload Characterization

We also consider workload characterization for E-commerce transactions. E-commerce workload is continuously changing, so it is difficult to forecast the intensity of the workload accurately. We evaluate Web and database workloads to come up with a workload characterization for E-commerce transactions. We evaluate two popular Web workload benchmarks, namely SPECweb [spe99] and SURGE [BC98], and borrow some of their characteristics that are relevant for our E-commerce transaction workload. In particular, the main features of a Web workload that capture the demands placed by clients on a Web server are object, *popularity, contention level* for an object and the *temporal locality* (i.e., repeated references to the same object). It is widely accepted that the popularity of objects on the Web follow the Zipf Law [BC98, spe99] and the contention for different objects is also believed to be non-uniform. In our work, we model the temporal locality of Web objects based on the SURGE model. In order to capture the other database parameters we use the 007-benchmark [CDN93], which was designed for multi-user object DBMSs. The values for the different parameters of the 007 workload were chosen to best reflect the demands generated by clients on the Web, and are described in detail in Chapter 4.

1.4.3 System Model

The system model consists of a client-server environment for the E-commerce architecture under consideration. In this environment, the clients and the server interact with each other at the level of physical database pages, or at the level of individual objects. We assume that the Web server and the database server (see Figure 1.1) are located on the same system, thus reducing the three-tiered architecture to two-tiers. The system architecture is such that the server dynamically generates a Web page from the database, based on the user request. Whenever there is a request for an object in a database page, the server sends the entire page that contains the object back to the client. At the client, the dynamically created Web page is cached in the client's cache. Although the caching of data is at the page level, conflict detection and lock management are performed at the object level. The updates initiated by the client are performed against the cached data and do not involve interaction with the server. When a transaction is ready to commit, it sends the modified objects back to the server along with a commit request.

The system environment is modeled to reflect the nature of the Internet. The main aspect of the Web that influences the behavior of cache consistency and, hence, transaction processing, is client variability, unpredictable delays associated with message transmission, and the error prone nature of the Internet [Pax97, FFF99, FGHW99]. Our system model accounts for these characteristics.

An accurate characterization of the nature of Internet is still an open research problem. Most of the work on modeling the Internet is not based on a generic model. Therefore, the delays associated with message transmission are mainly a result of the path taken by a packet, the geographical location of clients and server, and traffic volume. A generic model for characterizing delays associated with message transmission is still unknown and is beyond the scope of this report. For our purpose we consider that the delays show high variability and utilize a Poisson model to describe delays experienced by packets on the Internet. Client variation is modeled based on the survey conducted by [GVC98] and we use a mixed model for client characteristics, which is described in detail in Chapter 4.

1.5 Report Organization

The structure of the report is as follows.

In Chapter 2 we give the necessary background and related work. In this chapter we also provide a survey of the existing cache consistency schemes for both the Web and the database domain.

In Chapter 3 we describe in detail the design of our *optimistic* cache consistency algorithms.

In Chapter 4 we describe our simulation model for an E-commerce system and the workload characterization for E-commerce applications. We also describe the network model used and the experimental setup for our simulation study.

In Chapter 5 we explain the results of our experiments in detail.

The report concludes with Chapter 6, where we provide our contributions and comment on appropriate directions for future work.

Chapter 2

Background and Related Work

2.1 Reference Data-Shipping Architecture

A traditional client-server database system has two alternative architectures: *query-shipping* and *data-shipping*. In the query-shipping approach, query processing is done at the server. Most of the relational DBMS are based on this model. In contrast, in the data-shipping approach, the client's request is processed at the client, and the DBMS software component resides in both the clients and the server. All *read* and *write* requests to objects in the database are submitted to the client side DBMS component. The client DBMS software component processes the client's requests, and determines which data items are needed to satisfy them; these are then accessed from the server, if they are not found locally. Many object DBMSs are based on the data-shipping architecture.

The general architecture of a data-shipping client-server DBMS is shown in Figure 2.1. The system consists of a database server connected to a number of clients via the Internet. The server is connected to one or more disks on which the database is stored, as well as a log disk. Both the server and the clients have cache buffers, which serve as storage units for storing data that is repeatedly accessed. Caching at the client reduces repeated requests to the server; while caching at the server reduces repeated access for data from the disk.

In the data-shipping environment, the client DBMS component is responsible for taking the requests of the client and fetching the data items from the server to the client's memory. Some or all of these items may be cached at the client's buffer. Both the client and server DBMS components are responsible for preserving transactional semantics and maintaining the consistency of the cached data. The database client component communicates with the server, both synchronously and asynchronously, to handle reads, updates, and cache consistency. The clients are

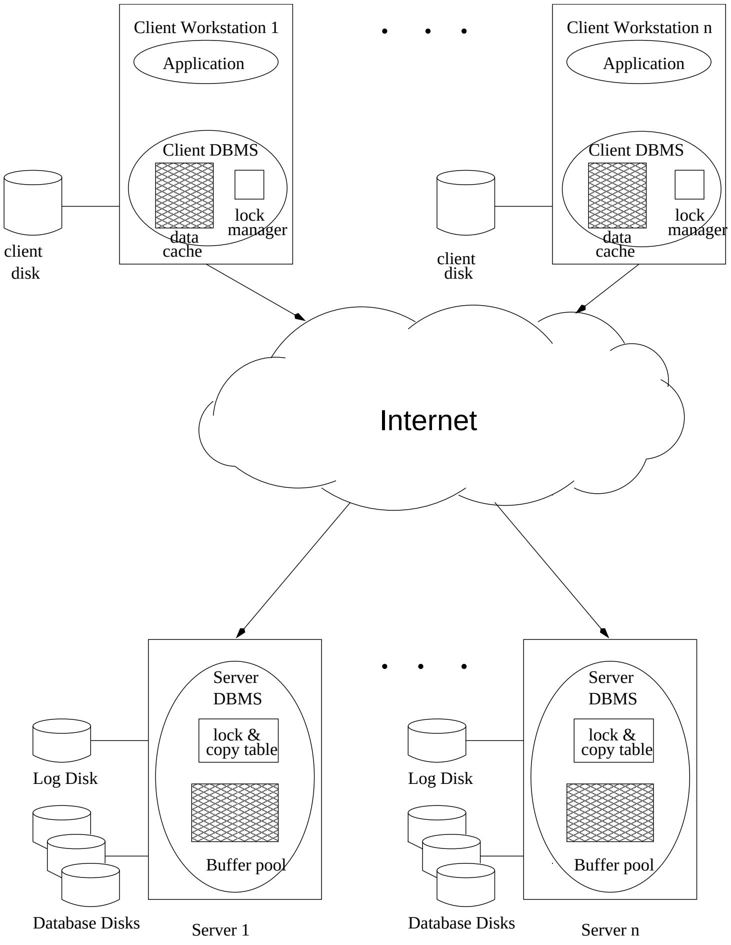

Figure 2.1: Reference Data Shipping Architecture

able to cache data across transaction boundaries as long as cache consistency is maintained.

In these systems, the server can operate either as a *page server* or as an *object server*. In a page server system, clients and servers interact using fixed-length units of data (known as database pages), while in an object server system, they interact at the level of groups of objects. Thus, a data page or object is the basis of interaction in a data-shipping architecture, and client requests are for objects or for pages.

2.2 Caching Dynamic Content on the Web

The content on the Web can be classified as *static* or *dynamic*. Static content refers to the pre-created Web pages stored at the server. Dynamic content refers to Web pages that are created by the server upon receiving a client's request from the data stored at the server repository. Most of the data referenced by E-commerce clients are dynamically created based on the client's query. Often these data are considered difficult to cache as they change with time and differ based on an individual user's request string. In this section, we discuss some of the existing techniques that aim at caching dynamically-created Web pages.

2.2.1 Delta Encoding

In the *optimistic delta* approach [BDR97], when a cache hit occurs, the cache manager optimistically returns the (possibly stale) data to the user, and sends a request for the current version of the document to the server. The server computes the *delta*, which is the difference between the current version and the old version that the client has accessed, and sends it to the requesting client. The client updates its version of the page by merging the delta with the cached version of the page. Although deltas save bandwidth, delta computation increases the server load as it must be done on the fly at the server, or at any intermediate network component. The server is also expected to keep track of the version that is cached by each client since computation of the delta is client-specific. The server, therefore, maintains a copy of the page that the client has accessed in order to calculate the delta. This technique might be reasonable for static pages where updates are minimal, but for dynamically-created pages — given the variation in the query strings and the possible ways a dynamic document can be generated — the number of versions to be maintained for each client increases enormously.

2.2.2 HTML Macro-Preprocessing (HPP)

The HTML macro preprocessing technique (HPP) [DHR97] is an application-specific approach that aims at separating the static and dynamic parts of a document. The static part, known as the *template*, can be cached, while the dynamic portion has to be accessed from the server for each request. The notion of static and dynamic parts comes from observing patterns in the way the dynamic page is created. For example, consider a page that is generated dynamically by the server in response to a query. The document contains a banner identifying the content provider, followed by the header information specifying fonts, formatting style, etc., and then the actual result for the query. On analysis of different queries we see that the static part can be cached freely. In [DHR97] Douglis et al. propose a macro encoding language which they refer to as HPP (HTML Pre-Processing). HPP is basically an extension to HTML that allows the separation of static and dynamic portions. The static portion – which can be cached – contains macro-instructions to insert dynamic information, while the dynamic portion has to be accessed from the server upon every request. The client expands the document before displaying it to the user, hence this method permits the partial caching of dynamic content. The macro preprocessing is done at the client side.

2.2.3 Active Cache

In [CZB98] Pei Cao et al. propose a scheme known as *Active Cache* for caching dynamic content. According to this technique, the server supplies cache applets that can be cached along with every document. The cache manager then invokes these applets upon cache hits, before contacting the server. Thus, a cache applet is attached to each URL or collection of URLs. The cache applets reside at the client cache or at any intermediate cache servers. Upon receiving a user request, the cache manager invokes the cache applet, which then decides what data needs to be sent to the user and whether it has to request data from the server or reply with the cached copy. Cache applets can do a variety of jobs, such as logging user accesses, rotating advertisement banners, checking for access permissions and constructing client-specific Web pages. The cache applet provides an on-demand migration of server functionalities to the cache.

The active cache is a technique proposed for caching dynamic pages for proxy servers. It has CPU overhead but results in significant bandwidth savings. As the number of clients and servers increases, the overhead experienced by this technique is also expected to increase. This is because it associates an applet for each document, which may increase the number of processes in the system, and therefore heavily load the CPU. Initial performance studies [CZB98] conclude that cache applets increase the latency experienced by the client by a factor of 1.5 to 4, and this degradation in latency is mainly due to the CPU overhead. The cache manager has the freedom to disable the cache applet if the CPU is heavily loaded.

2.2.4 Dynamic Content Caching Protocol (DCCP)

In [SAYZ99] the authors evaluate the similarity of dynamic documents generated by Web servers and present a technique called *Dynamic Document Caching Protocol* (DCCP) that allows applications to exploit query semantics and specify how a dynamic document should be cached and reused. According to this scheme the similarity exhibited by dynamically-created documents can be classified into three types: *identical requests*, *equivalent requests*, and *partially equivalent requests*.

- Identical requests have identical URLs and result in the generation of identical contents.
- Equivalent requests are not syntactically identical but result in the generation of identical content.
- Partially equivalent requests are not identical but result in the generation of partially identical contents, i.e., some fragment of the result can be satisfied from the cached content.

The Web server specifies the equivalence or partial equivalence measure for documents it generates. Hits for identical or equivalent URLs can be supplied from the cache without contacting the server, while those for partially equivalent requests are redirected to the server. An approximate solution is presented to the user while the actual content is being retrieved from the server.

2.3 Concurrency Control/Cache Consistency

Concurrency control refers mainly to the *consistency* and *isolation* properties of a transaction. Consistency refers to the validity of a transaction, i.e., the transaction manager must ensure that a transaction satisfies all the integrity constraints assumed by a database. The isolation property refers to the separation of one transaction from the effects of another transaction when they are executed concurrently. That is, a transaction T perceives the operations of another transaction as if that transaction was executed before T or after T. In a distributed multi-user environment there might be a number of concurrent transactions in the system that execute in an interleaved fashion – i.e., the operation of one transaction might execute between two operations of another transaction. This interleaved execution of transactions might sometimes interfere with the correct execution of programs, and might corrupt the database. Hence, the concurrent execution of transactions has to be controlled in order to avoid interference between transactions. A concurrency control algorithm ensures the correct execution of programs and prevents transactions from interfering with each other.

2.4 Serializability Theory

Serializability is considered to be the standard notion of correctness for transaction processing [RP95]. Intuitively, serializability can be defined as follows. A *serial* execution of all transactions prevents the interference problem mentioned earlier. Serial executions of transactions are correct since each individual transaction is correct, and their execution, one after another, leaves the database in a consistent state. However, since serial execution restricts system throughput, database systems allow concurrent transaction execution as long as the final database state is equivalent to one that would be obtained by serial execution. Such concurrent executions are called *"serializable"*. Since a serializable execution produces the same effect as a serial execution of the set of transactions, the serializable execution is also correct.

More formal definition of serializability refers to the notion of a *history*, which captures the execution order of the operations of a set of concurrent transactions. A serial history is one where there is no interleaving of operations belonging to different transactions - all the operations of one transaction are executed before the operations of the next transaction start their execution. A serializable history is one which is *conflict equivalent* to a serial history. Two operations from two different transactions conflict if they operate on the same data item and one of them is a write operation. Thus, two read operations do not conflict with each other, whereas a write conflicts with a read or a write.

Two histories H and H' are said to be *conflict equivalent* if they are defined over the same set of transactions, and if they order conflicting operations of committed transactions in the same way; that is, for any conflicting operations p_i and q_j belonging to transactions T_i and T_j $(i \neq j)$, if $p_i <_H q_j$ in H then $p_i <_{H'} q_j$ in H' for all p_i, q_j (where $<$ represents the order of operations). Hence the effect of concurrent execution of transactions depends only on the relative ordering of conflicting operations.

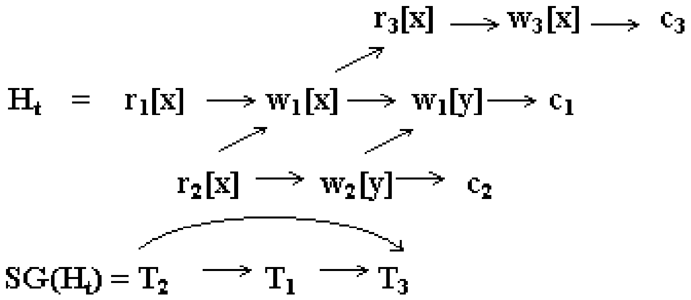

Figure 2.2: Example history (taken from [BHG87])

Serializability Theorem

A formal characterization of the serializability of a history is given by the Serializability Theorem: *A history H is serializable iff the serialization graph $SG(H)$ of the history H is acyclic*. The proof of this can be found in [BHG87].

The serializability of a history can be found by analyzing a graph derived from the history called a *serialization graph* $SG(H)$. Consider a history H made up of transactions T=$T_1, T_2, ..., T_n$. The serialization graph $SG(H)$ is represented as a directed graph whose nodes are the transactions in T that are committed in H, and whose edges are $T_i \rightarrow T_j$ where $i \neq j$ and one of the operations of T_i precedes and conflicts with one of T_j's operations in H.

Considering the example shown in Figure 2.2, we can observe from the serialization graph $SG(H_t)$ that $T_1 \rightarrow T_3$ and $T_2 \rightarrow T_3$ because of the conflicting operations $w_1[x] < r_3[x]$ and $r_2[x] < w_3[x]$, respectively, and that both $r_2[x] < w_1[x]$ and $w_2[y] < w_1[y]$ are responsible for the edge $T_2 \rightarrow T_1$. If the operations of T_i precede and conflict with operations of T_j, then in any serial history that is equivalent to H_t, T_i should precede T_j. For the history H_t, if there exists a serial history H_s with transaction ordering similar to the edges in the serialization graph $SG(H_t)$, then we can conclude that H_s is equivalent to H_t ($H_s \equiv H_t$) and therefore H_t is serializable. This is true as long as there are no cycles present in $SG(H_t)$.

The serializability notion is widely used in the database field to control concurrent execution of transactions. But for many applications, serializability is not the appropriate approach to control concurrent transactions. There are many applications in which concurrent execution of transactions is correct even if they are non-serializable. There are many extensions to serializability. In this work we consider one such technique, known as *epsilon serializability* [RP95], which aims at relaxing some of the bottlenecks that are inherent to serializability. Epsilon serializability is described in detail in Chapter 3.

2.5 Web Cache Consistency Algorithms

Web cache consistency algorithms are generally called weak consistency schemes because they can provide less-than-serializable levels of consistency, based on application needs. In this section, we discuss some of the popular Web cache consistency algorithms.

2.5.1 HTTP 1.1 Cache Consistency Model

The cache consistency model provided by the HTTP 1.1 protocol can be summarized as an expiration/validation scheme. Although the protocol does not include a fully defined (explicit) mechanism for cache consistency, it provides a number of message headers that allow the clients and server to specify the tolerance in consistency that they can accept. The most commonly used mechanism that the protocol supports is based on the *Expires* headers. The Expires header has the syntax *Expires: max-age*, where max-age represents the document's expiration time, i.e., the time until which the cached data is valid, rather than a time relative to the time at which the page was returned from the server. The server decides the max-age of any page and returns it by adding a *Cache-Control: max-age* header to the page. The cache manager has the freedom to heuristically determine the expiration time of a page by combining the max-age specified by the server, and the constraints specified by the client. When the client requests a page, if the request arrives before the expiration of max-age, the document is sent to the requesting client from the cache; otherwise the cache manager requests the data from the server. If max-age is not provided for a document, then the cache manager heuristically determines its max-age. If the server provides the *Last-modified* header, then the cache manager uses this value to determine the max-age value.

The HTTP 1.1 protocol also offers a cache validation mechanism that is used to validate the freshness of the cached data. The cache manager carries out this validation by including an *If-Modified-Since* (IMS) header along with a GET request for a page. The cache manager then places the Last-Modified date of the cached copy in the IMS header. Upon receiving the request, the server checks to see if the document has been modified since the time specified in the IMS request. If the document has not been modified, the server sends a 304 status code with the reply "Not-Modified", which states that the document was not modified; otherwise, it sends a 200 status code and the new page, along with the reply, if the document has been updated. A Not-Modified message can also include other cache consistency headers, such as Expires. One of the main disadvantages of the HTTP 1.1 protocol's cache consistency model is that it does not provide any strong guarantees about the staleness of documents. It is often difficult to specify the degree of staleness a user can tolerate, or the lifetime of documents in an interactive environment.

2.5.2 Time-To-Live (TTL)

Time-to-live is the most popular technique for cache consistency on the Web, and it uses the mechanism provided by the HTTP protocol. According to this algorithm, each cached document is associated with a TTL value, which denotes the time for which the document is expected to be valid. When there is a request for the document after the TTL value expires, the cache manager sends an IMS message to the server to determine if the cached copy is still valid. The difficulty lies in choosing the right TTL value for a document. Having a small TTL provides tight consistency guarantees, but at the cost of redundant IMS in case the cached copy is still valid. On the other hand, having a large TTL value increases the chances of accessing stale data. TTL is useful when the object's lifetime is predictable, as in the case of online newspapers, magazines, etc., which have fixed lifetime periods. The TTL value is mainly determined by heuristic functions. In [GS96], the authors use the notion that the longer the file is unmodified, the longer it will remain unmodified; i.e., the older the file is, the less likely it is to be modified. They propose an adaptive TTL scheme in which the TTL value is set to be a function of the lifetime of the document. When the TTL is set to zero, then an IMS request is generated for every request to the cached data. This prevents the client from accessing stale data, and is termed as *polling* every time. The client can also periodically poll to validate the freshness of the cached data. But polling can suffer from redundant IMS messages being sent to the server.

2.5.3 Invalidation Scheme

In the invalidation scheme, the server sends notification to the client whenever a page is modified. The server keeps track of all the clients that have cached a page, and sends an invalidation message to all the clients when the page is updated. The main overhead of this technique, compared to the others, is that the server has to keep track of the clients that have cached each document. The HTTP protocol does not provide any mechanism for supporting server-generated invalidation messages. In [CL98] the invalidation scheme is implemented as a HTTP accelerator to the existing server. The accelerator maintains an invalidation table for documents on the server, which contains the clients that have accessed the documents since the last invalidation. The accelerator detects changes to documents and sends an invalidation message to all the clients that have cached the document. Upon receiving the invalidation message, the client checks whether the document is cached; if so it deletes the copy from its cache. Once the invalidation message is received by the client, the server deletes the entry for the client from the server's invalidation table. This scheme prevents caches from holding stale data at the expense of increased message traffic.

2.5.4 Lease-based Consistency

In this method, whenever a client caches a page, it negotiates a (contract) lease with the server. When the page gets updated, the server notifies the client of the update, as long as the lease is valid. If the cache receives a request for a document for which the lease has expired, the cache manager sends an IMS header along with the request to the server and renews the lease for that document with the server. The consistency mechanism involved while the lease is valid is similar to that of the invalidation method. When a document is modified, the server sends invalidation messages to the clients holding the lease. The lease negotiation is similar to the TTL approach, as the length of the lease is basically decided by constraints specified by the client and the server which, in terms of the TTL scheme, is the expiration time for a document.

2.6 Transactional Client Server Cache Consistency Schemes

Caching is a dynamic form of replicating data at the client sites. This replication of data may jeopardize the correctness of the application execution and can corrupt the database. Database systems require that ACID (Atomicity, Consistency, Isolation, Durability) semantics be guaranteed for concurrent execution of transactions in the presence of failures. A transactional cache consistency maintenance algorithm ensures that caching does not result in the violation of transaction semantics. It guarantees that transactions which accessed stale data are not allowed to commit. Cache consistency algorithms can be classified as *avoidance-based* and *detection-based* [FCL97], depending on whether the algorithm avoids or detects transactions accessing stale objects. The main criterion used for classifying the different algorithms in this taxonomy depends on the mechanism adopted by algorithms for preventing transactions from accessing stale data.

Avoidance-based algorithms prevent access to stale cache data and ensure that invalid data is quickly removed from client caches. These algorithms ensure that caches do not contain stale data and they employ a read-one/write-all (ROWA) approach for replication management. A ROWA protocol ensures that all existing copies of an updated item have the same value when an updating transaction commits. In a ROWA scheme, a transaction is allowed to read any copy of a data item; however updates must go through a process of updating all the copies existing in the system, such that no remote cache holds a stale copy of the data beyond the updating transaction's commit point. Detection-based algorithms allow caches to contain stale data that can be accessed by transactions, but detect and resolve them at transaction commit time. The detection-based schemes require transactions to explicitly check for the validity of data accessed by transactions. The choice between the avoidance-based and detection-based approach has a great impact on the performance of the system [FCL97]. The design of a cache consistency algorithm, and hence the performance, is influenced by other metrics like correctness criteria, granularity of caching, inherent cost tradeoffs, and workload characteristics. In the remainder of this chapter, we explain both of these taxonomy classes in detail, and discuss some of the popular cache consistency algorithms for a data-shipping environment.

2.6.1 Detection-based Algorithms

Detection-based algorithms [FCL97] allow stale data copies to reside in a client's cache. Hence, a client can access stale data but a transaction goes through a validation phase, involving checking the validity of any cached page that it accesses, before it can be allowed to commit. The server is responsible for maintaining information that will enable it to perform the validity checking. Detection-based schemes are so named because access to stale data is explicitly checked for and detected. An advantage of the detection-based approach is simplicity: the cache management software on the clients is greatly simplified for the detection-based scheme, when compared to the ROWA-based approach employed by the avoidance-based scheme. However, their disadvantage is the greater dependence on the server. The detection-based algorithms are differentiated based on three key aspects of their design: *validity check initiation*, *change notification hints*, and *remote update action*.

Validity check initiation refers to the point during transaction execution at which the validity of accessed data is checked. The validity of any accessed data must be determined before a transaction can be allowed to commit. Validity checking can be performed in three ways: *synchronous, asynchronous,* and *deferred*. All these three methods ensure that once the validity of a client's copy of a data item is checked, then that copy is guaranteed to remain valid for the duration of the transaction. To implement this guarantee, the server must not allow other transactions to commit updates to such items until a transaction that has received a validity guarantee finishes (commits or aborts).

In synchronous validity check initiation, the transactions are not allowed to access the item on the first access to the data item, before its validity has been verified with the server. When the server has checked the validity of the client's copy of the data item, then the copy is guaranteed to remain valid at least until the transaction completes. In asynchronous validity checking, the transaction does not wait for the result of the validity check from the server. The client optimistically proceeds with accessing the local copy of the item under the assumption that the validity check will succeed. But if the server finds any conflicts then the transaction aborts and restarts.

The deferred validity checking technique is even more optimistic than the asynchronous validity checking method. The validity check for the cached data is not sent to the server until the transaction has completed its execution phase and has entered its commit phase. The main disadvantage of deferred validity checking is that the conflicts are detected late and this can result in aborts. On the other hand, the deferred method reduces the amount of work that has to be done to maintain consistency. The asynchronous method reduces the abort rate when compared to the deferred method, and reduces the overhead involved for earlier discovery of conflicts when compared to the synchronous method.

The change notification is another aspect by which the cache consistency algorithms can be differentiated. This notification refers to the action sent to a remote client as a result of concurrent update transactions or impending update transactions that may impact the validity of an item cached at the client. Purging or updating the stale copy are the methods used by most of the cache consistency algorithms. These techniques prevent subsequent transactions from accessing the stale data, and avoid transactions from being forced to abort as a result of accessing stale data. The notifications to affected clients can be sent asynchronously at any time, either during the execution of an update transaction or even after the update transaction commits. Early notification aids in the early removal of stale data from a client's cache, and results in reducing the possibility of clients accessing stale data, thereby reducing the number of transaction aborts. Sending notifications, before commit, that actually update the remote copies, rather than purging them, can sometimes be dangerous. That is, if the transaction on whose behalf the notification was sent eventually aborts, then the remote updates will have to be undone, adding significant complexity to the algorithm. This may also result in cascading aborts.

The final design option for the detection-based algorithm is concerned with the action taken when a notification arrives at a remote client. The algorithm can choose between *propagation*, *invalidation*, and dynamically choosing between the two (*hybrid*). Propagation results in the server sending the updated data along with the notification, and the stale copy is replaced by the new updated value at the remote client cache. Invalidation, on the other hand, simply purges the stale copy from the remote cache so that any subsequent transactions will be prevented from accessing it. Dynamic algorithms choose between invalidation and propagation heuristically; the heuristics are often designed for better performance and to handle varying workloads.

2.6.2 Avoidance-based Algorithms

The dynamics of avoidance-based algorithms [FCL97] are such that they ensure consistency by preventing the local cache from ever holding stale data; thus they prevent transactions from accessing stale data. Such algorithms use a read-one/write-all (ROWA) approach for replica management. Avoidance-based algorithms thus need to maintain extra information at the server that keeps track of the location of all copies in order to satisfy the "write all" requirement of ROWA protocol. The design of an avoidance-based cache consistency scheme can be differentiated based on the following key aspects: *write intention declaration, write permission duration, remote conflict priority,* and *remote update action*.

When a transaction wishes to update a cached page copy, the client must inform its write intention sometime prior to transaction commit so that the server can implement the ROWA protocol. When the server grants write permission on a page to a client, the permission is guaranteed to be valid for a certain period of time — known as the write permission duration - and the client can update the page without again having to contact the server. The write intention declaration can be performed in three ways: *synchronous*, *asynchronous*, and *deferred*. In synchronous algorithms, the client sends a lock escalation message at the time it wants to perform a write operation, and it blocks until the server responds. In asynchronous algorithms, the client sends a lock escalation message at the time of its write operation, but does not block waiting for a server response. Rather, it proceeds optimistically to access the local copy under the assumption that the write permission will succeed. In deferred algorithms, the client optimistically defers informing the server about its write operation until commit time.

These three classes provide a range from pessimistic (synchronous) to optimistic (deferred) techniques; therefore, they represent different tradeoffs between checking overhead and possible transaction aborts. Deferring consistency actions can have two advantages. First and most significantly, consistency actions can be bundled together in order to reduce consistency maintenance overhead. Second, the consistency maintenance work performed for a transaction that ultimately aborts is wasted; deferred consistency actions can avoid some of this work. However, there are also some potential disadvantages to deferring consistency actions. The main one is that deferral can result in the late detection of data conflicts, which will cause the abort of one or more transactions. However, blocking transactions – as in the synchronous approach – causes the transaction deadlock rate to be very high. The asynchronous approach is a compromise; it aims to minimize the cost of interaction with the server by removing it from the critical path of transaction execution, while at the same time lowering the abort rate, deadlock rate, and cost for earlier discovery of conflicts.

Write permission duration refers to how long a client is allowed to hold the write permission. There are two choices: write permissions can be retained only for the duration of a particular transaction, or they can span multiple transactions of a given client. In the first case, transactions start with no write permissions, so they must eventually declare write intentions for all pages that they wish to update and, at the end of the transaction, all write permissions are automatically revoked by the server. In the second case, the client retains the write permission as long as he wishes to do so, or until the server asks the client to drop the permission.

Remote conflict priority indicates the priority that is given to consistency actions when they are received at remote clients. The algorithms have the choice of either waiting or preempting. In the wait policy, the consistency actions at a remote client must wait until the ongoing conflicting transaction at another client completes. In contrast, under a preemption policy, the incoming consistency action can abort the ongoing transactions.

Remote update actions are based on how the updates are implemented. They are similar to the detection-based schemes, namely, invalidation, propagation, and choosing dynamically between the two (*hybrid*). However, an important difference between the remote update actions under the avoidance-based and detection-based schemes is that, in the avoidance-based schemes, the remote operations are initiated and must be completed on behalf of a committing transaction, and the committing transaction remains blocked until this is done. This mechanism is necessary in order to maintain the ROWA semantic guarantees that avoidance-based algorithms provide.

2.7 Database Cache Consistency Algorithms

The taxonomy presented in the previous section provides the various options available for the design of a transactional cache consistency maintenance algorithm. In this section, we study and critique some popular transactional database cache consistency algorithms for a data-shipping environment.

2.7.1 Caching Two-Phase Locking (C2PL)

C2PL is a detection-based algorithm [FCL97] that uses a “check-on-access” policy, i.e., the client validates cached pages synchronously on a transaction’s initial access to the page. C2PL is so called because the cached data are retained across transaction boundaries. The server takes care of all locking and deadlock detection duties. In C2PL, all page copies are tagged with the *log sequence number* (LSN) of the page, if the page is cached at the client. Whenever there is a lock request for a page that is resident at the client cache, the client includes the LSN of the page along with the lock request message. The server uses the LSN to check if the client’s copy is still valid. If the client’s copy is stale, then the server piggy-backs a valid copy of the page onto the lock response message returned to the client. The server maintains a the LSN table containing LSNs of all pages cached by all client workstations. The clients notify the server about page discards, and usually they are piggy-backed with other messages to the server. If any transaction holds conflicting locks, then the lock request blocks at the server until those locks are released. C2PL uses strict two-phase locking, in which all locks are held until transaction commit or abort. Deadlock detection is performed by the server, and is resolved by aborting the youngest transaction involved in the deadlock. The main disadvantage of this algorithm is that it is blocking and has a high transaction abort rate.

2.7.2 Callback Locking (CBL)

CBL [WR91] is an extension of the two-phase locking that supports inter-transaction page caching. However, callback locking algorithms are avoidance-based. They guarantee that pages cached at the clients are always valid. Whenever there is a request for data, the client first checks its cache. If the data item is cached at the client, then the client’s transaction is given access to the cached copy without contacting the server. But on a cache miss, the client sends a page request to the server and the server then returns a valid copy of the requested page if it does not detect any conflicts. Callback locking is a synchronous algorithm as the write intentions are declared synchronously. Hence a client must gain write permission on a page before a local write lock is granted to a transaction. The client obtains this write permission on a data item during the course of the transaction execution, and can commit a transaction without performing any additional consistency maintenance actions. There are two callback locking variants: Callback-Read (CB-R) and Callback-All (CBA). In the former the write permissions are granted only for the duration of a single transaction, while for the latter the write permissions are retained at the clients until being called back, or until the corresponding page is dropped from the cache.

The server maintains lock tables and keeps track of the state of all the clients' caches. The clients notify the server about page discard by piggy-backing that information onto messages they send to the server. The client's site contains a local lock manager from which a client's transactions obtain locks. Requests for read or write locks can readily be granted to the client without contacting the server if the client obtained write permission on that data item earlier. However, a request for a write lock on pages for which the write permission has not yet been granted causes the client to send a write intention request to the server and remain blocked until the server responds with permission for writing on that data item. When a request for a write lock arrives at the server, the server issues callback requests to all client sites that hold a cached copy of the requested page. The client, upon receiving a callback request, checks to see whether the data are being used and, if so, sends a reply conveying a conflict; or else sends a positive callback reply message conveying that there is no conflict with the requesting client's transaction. If the server does not detect any conflict, then it grants write permission immediately. If there are any conflicts, the requesting client will remain blocked. When the server grants write permission, subsequent read or write requests for the page, caused by transactions from other clients, remain blocked at the server until the write permission is released by the holding client, or else permission is revoked by the server. The advantage of the CBL scheme is that clients never access stale data, which results in a low transaction abort rate. However, they suffer from deadlock-related aborts as in some cases, one or more transactions being blocked in order for another transaction to complete can create a deadlock situation. Also, commit processing is less expensive since the validity of the item need not be checked at commit time.

2.7.3 Optimistic Two-Phase Locking (O2PL)

The O2PL algorithm is an avoidance-based cache consistency algorithm [CL91]; however it is considered to be more "optimistic" than callback locking as it defers notification of all write intentions until the end of a transaction. O2PL algorithms have two variants: O2PL-Invalidate and O2PL-Propagate, which are based on how remote updates are handled. In this algorithm, the clients are required to maintain a local lock manager, and all requests for locks are obtained from the client lock manager. There is no explicit request for a lock to be sent to the server during the execution of a transaction except when there is a request for a data page from the server. All updates on pages are against the cached local copy of the data item, and these updated pages are retained at the client until the transaction enters its commit time. At commit time, the client sends a message to the server containing the new copies of the updated pages. The server then acquires exclusive locks on these pages, on behalf of the committing transaction, which are held until the transaction completes. Once the locks are obtained at the server, the new updated values of the pages are safely installed in the database. The server, upon obtaining the required locks, sends a message to each client that has cached copies of any of the updated pages. If any of the remote client's transactions currently holds a read lock on its local copy, then the update transaction will have to wait for the read transaction to complete before it can continue commit processing. The remote client site performs consistency actions on its copies of the updated pages once all of the required locks have been obtained from the local lock manager. Since O2PL is locking-based, it suffers from deadlock-related aborts. However, the advantage of this algorithm is that it has low transaction-related aborts.

2.7.4 Adaptive Callback Locking

The adaptive callback locking scheme (ACBL) is a synchronous, avoidance-based cache consistency algorithm [CFZ94]. In this algorithm, the clients cache both data and read locks across transaction boundaries. ACBL is an adaptive version of the page level CBL algorithm, as it dynamically acquires either page level or object level locks. Any write operations on a data item require the client to obtain write permission from the server before they can proceed with the write operations. Clients try to acquire page level write locks; failing that, they try to acquire object level write locks on shared pages. If the page is cached at other clients, the server sends a callback message to these other clients asking them to downgrade or relinquish their locks. Since ACBL is an avoidance-based scheme, it assures that transactions never access stale data and therefore never have stale cache aborts. However, ACBL suffers from deadlock-related aborts.

In ACBL, if a client requests a write lock on an object for which it possesses a read lock, it sends its lock request to the server and remains blocked until the server responds. The server then checks if any other clients have cached the object, failing that it immediately grants the write lock to the requesting client. If any other client has cached the same object, then the server issues a callback message to that client. If the client is not operating on that object, it invalidates it and sends a callback reply to the server. The server then sends a response to the requesting client granting it an exclusive lock on that object. If the client is operating on that object, it indicates to the server that it cannot comply with the request. The requesting client thus remains blocked until the other client commits and releases the object. ACBL has low transactional abort rate. However, it is blocking whenever a read/write conflict arises and it has deadlock-related aborts.

2.7.5 Adaptive Optimistic Concurrency Control

AOCC [Gru97] is a deferred, detection-based cache consistency algorithm. Hence it does not prevent the access of stale data by clients. Since the work in this report is based on AOCC, the technique is discussed in detail in Chapter 3. In this section we provide a summary of the mechanism of this algorithm.

In this scheme the clients implicitly hold read permissions on cached data, but if they subsequently update cached data, like any other deferred scheme they defer all write notification messages until commit time. At commit time, the server performs commit time validation on every object that has been accessed by the client's transaction. The server checks whether the client accessed the most recent committed version of the object. If not, the transaction is aborted. Upon a successful update, the server sends invalidation messages to all the clients whose cache holds the updated pages. These invalidations are piggy-backed onto other regular messages to those clients. If the client that receives object invalidation has accessed the corresponding object, then it performs a stale cache abort and purges the page/object from its cache. Since AOCC is an optimistic algorithm, there is no locking involved. Clients do not encounter read/write or write/write blocking. AOCC does not encounter deadlocks, but does suffer from the problem of cascading aborts since it is susceptible to starvation, i.e., a transaction may abort repeatedly. The advantage of this scheme is that it has a low messaging overhead and, since the transactions are never blocked, there are no deadlock-related aborts. The disadvantages are that there is a high transaction abort rate, and also high overhead for commit time validation.

2.7.6 Asynchronous Avoidance-Based Cache Consistency Algorithm

AACC [ÖVU98] is an asynchronous, avoidance-based cache consistency algorithm. This algorithm is a locking-based scheme and is similar to ACBL. However, it overcomes the high message transmission and message blocking seen in ACBL, while retaining a low abort rate. AACC accomplishes high performance by applying a number of performance enhancement techniques as well as by adopting various features of the ACBL and AOCC algorithms. A detailed study of the algorithm's performance for a client-server DBMS is presented in [ÖVU98]. In the AACC algorithm, the client site contains a lock manager that regulates all access to pages on the client site. Since AACC is a lock-based scheme it has read/write and write/write conflicts. Similar to ACBL, the clients retain their locks across transaction boundaries, enabling inter-transaction caching. Both the server and clients play a role in lock management, and deal with page-level as well as object-level locks.

	Opt/ Pess	Validity chk ini.	Updates	Blocking	Dead locks	Abort rate	Message overhead
C2PL	Pess	Sync	Prop	yes	High	Low	High
O2PL	Opt	Deff	Inv/Prop	yes	High	High	Medium
CBL	Pess	Sync	Inv	yes	High	Low	High
ACBL	Pess	Sync	Inv	yes	High	Low	High
AACC	Pess	Async	Inv	yes	High	Low	Medium
AOCC	Opt	Deff	Inv	no	no	High	Low
HTTP 1.1	Opt	N/A	N/A	no	N/A	N/A	Low
TTL	Opt	N/A	N/A	no	N/A	N/A	Low
Inv.Schm	Opt	N/A	Inv	no	N/A	N/A	Low
Leases	Opt	N/A	Inv/Prop	no	N/A	N/A	Low

Table 2.1: Summary of Algorithms

In the AACC algorithm, the client must obtain appropriate locks from the client lock manager before a client transaction can access an object or a page. If a client requests a read lock on a page that is cached at the client site, the cache manager checks to see if there are any conflicts and, if there are none, the transaction is given access to the object. Upon a cache miss, the request is passed to the server and the clients implicitly obtain object read locks when the object is brought into the client's cache. All updates are against the data in the cache and require the client transaction to obtain a write lock before proceeding with the update. On obtaining a write lock, the client notifies the server of its write intention asynchronously and the server sends callback messages to all the clients holding that object in their cache. Each client which receives this callback message in turn checks to see if their client's transaction is using that object. If there are no conflicts then the clients send a positive callback message; otherwise, they convey that there is a conflict. Upon receiving a positive callback message from all the clients, the server grants a write lock to the requesting client. In the case of a conflict, the requesting client remains blocked until the read transaction of the conflicting client ends. The server performs deadlock processing when there are lock conflicts. The clients do not block the transaction at the time they perform the write operation; instead a client blocks at the commit time if its updates will cause a remote client's cache to contain stale objects.

Summary of Algorithms

In this section we evaluate some of the tradeoffs of the cache consistency algorithms for Web and database domain. The comparison of the tradeoffs are based on the study presented in [FCL97] and [ÖVU98]. The first study discusses the performance of O2PL, C2PL and CBL algorithms, while the later compares the performance of ACBL, AOCC and AACC algorithms. The performance of the different algorithms are largely dependent on the workloads used and the tradeoffs varies with different workloads. The values presented in Table 2.1 represent general characteristics that are dominant for each algorithm. It can be observed that most of the database cache consistency algorithms are pessimistic in nature (Pess), while all the Web cache consistency algorithms are optimistic in nature (Opt). All of the database schemes except AOCC are lock-based, therefore they involve transaction blocking whenever lock conflicts are detected. All of the lock-based schemes suffer from dead-lock related aborts, as there can be dead-lock situation in which two or more transaction can block requesting locks possessed by each other. The optimistic deferred schemes (AOCC and O2PL) suffer from high transactional aborts which are due to deferring all updates until commit time. The comparison of the messaging overhead of the different algorithms are with respect to the messaging overhead of AOCC algorithm and depends on the contention for a data item. The typical message counts for different algorithms are such that optimistic algorithms have low messaging overhead when compared to the pessimistic schemes. AACC algorithm reduces the messaging overhead by using asynchronous messages and achieves better performance when compared to CBL and ACBL. The tradeoffs presented for Web cache consistency algorithms are basically with respect to that of the database algorithms.

2.8 Update Propagation Techniques

In this section, we look at different techniques for propagating updated data from the server to the clients. There are five possible strategies for data propagation [DR98, DR94]. The main distinction among the different algorithms is based on whether the server keeps track of the data cached by clients, which is known as the *data binding information*.

On Demand Strategy (ODM): This policy refers to the clients requesting the data from the server on an on demand basis. The server does not have to do any book-keeping to keep track of the client cache status. Whenever there is a request, the client presents the server with the binding information. The server uses the binding information to filter the portions of the server logs that need to be sent to the clients.

Broadcasting with No Catalog Binding (BNC): This strategy uses broadcasting techniques. The server pushes updates or data modifications to all clients upon the commit of an update transaction. This method does not maintain any binding information regarding the client's cache status and hence does not keep track of which pages are cached at which client; it pushes the updates to all clients in the system, regardless of whether a client has cached the page or not. This system is not scalable in view of a large number of clients. The advantage of this technique is that the server avoids some of the overhead, such as look-up log operations and the computation needed to determine the destination of updates. Upon arrival of an update from the server, the client checks to see if the update affects its local operation; if so, it aborts.

Broadcasting with Catalog Binding (BWC): In this scheme, the server keeps track of the status of client caches. Upon an update, the server decides on the clients that must be notified, and propagates updates based on the binding information. This technique reduces the number of updates to be propagated at the cost of maintaining binding information for all the clients. The other two strategies (*Periodic broadcasting with catalog binding (PWC)* and *Periodic broadcasting with no catalog binding (PNC)*) incorporate the idea of periodic update broadcasting. In these techniques, the server collects the changes not seen by the client at some regular interval of time and initiates the propagation of the updates to the client. If the server maintains book-keeping information about the client's cache status, then the server sends only a portion of the updates to the client (PWC). On the other hand, if the server does not maintain any binding information, then all the updates are propagated to all the clients (PNC).

Chapter 3

Optimistic Web Cache Consistency Algorithm

In this chapter, we look at the design details of our proposed Optimistic Web Cache Consistency (OWCC) algorithm, considering a data-shipping environment. The design of the algorithm is tailored to meet the performance requirements of the Web and to exploit the tolerance in consistency exhibited by E-commerce applications. We use epsilon serializability [RP95, PC93, Pu93, Pu91] for correctness criteria to relax the consistency and to achieve a greater level of concurrency.

Relaxing ACID properties of databases is not new; there have been many proposed schemes. However, most of them are not compatible with serializability and this incompatibility restricts their use with the database consistency schemes. Epsilon serializability (ESR) is a generalization of classic database serializability. It achieves a higher level of concurrency by allowing a bounded amount of inconsistency to exist in applications that can tolerate such inconsistency. The main advantage of ESR is that it is a general framework, it can be tailored to fit to a wide range of application semantics, and it is compatible with classic database serializability (SR). Epsilon serializability is described in detail below, therefore a discussion is useful for understanding the relaxation mechanism used in OWCC.

3.1 Overview of Epsilon Serializability

Epsilon serializability is characterized based on the assumption that database state space exhibits a regular geometry, and that a distance function can be defined over this state space. A database state space is defined as the set of all possible database states, which basically refers to the set of all data values for the set of data items in the database. ESR is mainly applicable in systems that exhibit a metric space. A state space is considered to be a metric space if a distance function can be defined over every pair of states, and it exhibits triangle inequality and symmetry properties (a detailed definition for metric space is provided in [RP95]). In our work, we assume Web data state space to be a metric space because most of the operations involved in an E-commerce application are over numeric data. For example, consider an airline reservation application: the consistency calculation can be based on the availability of tickets for a particular plane. Hence, the distance function can be the difference between the currently available tickets and the data (tickets availability) viewed by a client. We can observe similar state spaces for most of the E-commerce applications.

ESR allows us to explicitly set bounds on the amount of inconsistency an application can accept. That is, a threshold is set by the application designer on the tolerable distance between any two conflicting states. The basic idea is that a query imports inconsistency if it reads stale data, and an update transaction exports inconsistency to all other concurrent transactions when it writes on a data item that has been read by other transactions. A query can accumulate inconsistency, but care must be taken to ensure that the inconsistency is within certain bounds - defined with respect to an *import-limit* and *export-limit*. The *import-limit* represents the maximum allowable inconsistency imported by a query transaction, and *export-limit* represents the allowable amount of inconsistency exported by an update transaction. Let *import-inconsistency* and *export-inconsistency* be the accumulated inconsistency imported and exported by a transaction *t* on a data item *x*. The system must ensure the following constraints in order to guarantee epsilon correctness criteria.

$import - inconsistency_{t,x} \leq import - limit_{t,x}$ ...(1)

$export - inconsistency_{t,x} \leq export - limit_{t,x}$ ...(2)

Hence, for each operation of a transaction, the above condition - known as the *safety criteria* - must be satisfied; otherwise the transaction's operation is said to conflict with operations of other transactions. A formal characterization of the correctness criterion used by epsilon serializability uses the same notation as the Serializability Theorem; the only difference is that ESR tolerates cycles appearing in the serialization graph as long as the conflicting edges of the graph do not violate the safety criteria described above. Thus a history H is (conflict preserving) epsilon serializable iff, in the serialization graph produced by the operations of the concurrent transactions, there is no cycle that has an epsilon-conflicting edge. A epsilon conflicting edge refers to the edge found in the serialization graph, which does not adhere to the safety criteria described above.

One of the problems of introducing inconsistency in a transaction is bounding it within certain limits. A divergence control algorithm (DC) is used to overcome this problem and to preserve epsilon serializability. An initial study on three DC algorithms, namely, two-phase locking DC, time-stamp ordering DC and optimistic DC, can be found in [WYP92]. The two-phase locking DC algorithm is based on relaxed lock management, and defines a distance function for tolerance in detecting a conflicting operation. Hence, two operations with conflicting locks are compatible as long as they are epsilon serializable. The timestamp-ordering DC method aims at relaxing non-serializable read and write conflicts. It also checks for the accumulated inconsistency of a transaction whenever there is a conflict, and allows the transaction to proceed if it satisfies the safety criteria. In the optimistic DC scheme, the requests are allowed to proceed immediately, but the transaction goes through a validation at commit time. This last method uses the notion of *weak-locks* and *strong-locks*; the strong-locks always have priority over weak-locks and conflict with each other, while two weak-locks are compatible, and do not conflict. At commit time, a transaction is marked for abort only if it has conflicting locks and its accumulated inconsistency exceeds certain bounds. Usually the transaction with a weak-lock is marked for abort. In OWCC we have the same notion of a commit time validation, but it does not involve any locks and lock conflicts.

3.2 Optimistic Web Cache Consistency Algorithm

The Optimistic Web Cache Consistency algorithm (OWCC) is an optimistic detection-based cache consistency algorithm, that is based on the AOCC algorithm [AGLM95, Gru97]. Current cache consistency algorithms for the Web are mainly based on the Time-To-Live (TTL) values for Web pages, and are not suitable for E-commerce transactions since they do not provide the needed consistency requirements. Availability is a primary concern for Web services and so OWCC is designed not to block any request for data. OWCC basically focuses on relaxing consistency as most of the Web applications can tolerate some degree of inconsistency. In particular, OWCC bounds the staleness of data viewed by transactions while, at the same time, bringing about a generalized correctness criteria for correct execution of transactions for applications that can tolerate some kind of inconsistency.

OWCC employs epsilon serializability (ESR) against the traditional serializability for correctness criteria for transaction processing. Through epsilon serializability, OWCC overcomes some of the bottlenecks that are inherent to transaction processing and provides application-specific transaction guarantees. Epsilon-serializability provides an easy means for allowing a tolerable inconsistency to exist in transaction processing for applications that can tolerate some degree of inconsistency. Hence, epsilon serializability relaxes consistency and bounds inconsistency to some tolerable limit specified by the application designer. One of the main features of this algorithm is that it defers all update notification until commit time. This scheme does not prevent the client from accessing stale data; rather it optimistically assumes that the data stored in the client cache is up-to-date. In OWCC, the clients communicate with the server only when there is a cache miss and when the transaction commits. All consistency actions are piggy-backed onto requests and replies. The usage of a minimum number of messages required to execute a transaction, along with early abort detection, and an almost up-to-date client cache, contributes to the good performance of OWCC. A transaction passes validation if it can be epsilon serializable with other concurrent transactions in the system.

Detailed Description

In order to describe the mechanism of OWCC, we describe in detail the mechanism of the client followed by that of the server. We assume a single server system and a system architecture as depicted in Figure 2.1.

Client Description

The client side mechanism of the OWCC algorithm consists of the fetch, update, commit, and abort/restart actions. The client data structure includes a table that stores all the objects used by the client's transaction (ROT). All the pages cached at the client cache are entered in another table (RPT), and the *modified-object-buffer* (MOB) stores all the objects updated by the client's transaction. The client also maintains an *undo-log-buffer* (ULB), which stores the original version of the updated objects.

Step 1: Upon receiving a request,

- if the request is for an object, check RPT to see if the page and, therefore, the object, is cached locally. If the object is found locally, proceed to Step 3, otherwise continue with Step 2.
- if the request is for transaction commit, proceed to Step 4.

Step 2: Upon a cache miss, generate a *fetch* request for a page and send it to the server. Piggy-back any page discard notifications to be sent to the server onto the fetch request. The fetch request results in the server sending the corresponding page from the database. On receiving the fetch reply, cache the page and make an entry in the RPT for the cached page. If the page was cached as a result of replacing any older page, then place a discard-notification to be piggy-backed onto the next request to the server, and mark the page discarded from the cache in RPT. Check for piggy-backed invalidation messages sent from the server along with the fetch reply. If there are invalidation messages, proceed to Step 5, otherwise continue with Step 3.

Step 3: If the object is present locally, then enter a read or write lock in the ROT table for the requested object. If the request is a write operation, then place the original version of the object in the ULB and the modified version of the object in the MOB; continue with Step 1.

Step 4: Send the elements of ROT and MOB to the server, along with a commit request; wait for the server's reply:

- if the server replies with a successful commit then clear the MOB, ULB, and ROT, and begin the next transaction.
- if the commit fails, then abort the transaction. Use the ULB for restoring the state of the objects that are present in the client's cache, and clear the MOB, ULB, and ROT. If there are any piggy-backed invalidations, then proceed to Step 5.

Step 5: Whenever there is a piggy-back invalidation for objects, along with replies from the server, the client does the following:

- if the object to be invalidated from a page P does not hold any lock in the ROT, then discard the object from the ROT. If the ROT contains no lock entries for any object of the page P, then discard the page from RPT. Place a discard notification to be piggy-backed onto the next request to the server, and continue with Step 1.
- if the object to be invalidated is present in the ROT, then abort the transaction. Use the ULB for restoring the states of the objects that are present in the client's cache, and clear the MOB, ULB, and ROT. Restart the transaction; continue with Step 1.

Server Description

The server side mechanism of the algorithm consists of the request processing, commit time validation, and the sending of invalidation notices to remote clients. The server data structure includes a table that stores all the pages cached by different clients' transactions (RPT) and the *modified-object-buffer* (MOB), which stores all the objects updated by these transactions. CLIENTS(P) stores the list of clients caching a particular page P, and Invalid(C) stores the list of invalidation messages for objects to be sent to a client C. Whenever the server sends a message to client C, the server piggy-backs the contents of Invalid(C) onto the message.

Step 1: Upon receiving a request from a client C, if the request is a fetch request proceed to Step 2; if it is a commit message, then proceed to Step 3.

On receipt of any message from a client C, first process any piggy-backed messages. For a discard notification for a page P, remove the entry for C from CLIENTS(P). For an invalidation acknowledgment, remove the invalidation message from the Invalid(C) that has the same sequence number as the sequence number on the acknowledgment.

Step 2: Upon a fetch request from a client, check to see if a page P is cached in the server cache. If the page P is cached, then access the page from the cache or else read it from the disk. Send page P to client C and add C to CLIENTS(P). Generate invalidation messages to be sent to C, from Invalid(C), and piggy-back them onto the fetch reply; proceed to Step 1.

Step 3: Upon a commit request from client C, perform commit time validation to check for epsilon serializability and validate the data accessed by the client's transaction.

- If the validation is successful, then commit the transaction and send a commit reply to client C. For each page updated by the client C's transaction, place an invalidation message in the invalidation queue of all the clients that have cached that page, i.e., place invalidation notices only for clients who cached the page updated by the committed transaction, and proceed to Step 1.
- If the validation fails, abort the transaction and send an abort reply to client C. Generate invalidation messages and piggy-back them onto the abort reply and proceed to Step 1.

Inconsistency Calculation

In order to find the inconsistency accumulated by a client for an object, the server maintains an instance of the page cached by a client in its RPT. The server uses this version along with the current version of an object to calculate the inconsistency of an object for a data item. Since the nature of the data item is assumed to be numeric, the inconsistency measure is the difference in value between the data viewed by the transaction and the current version of the data item at the server. Hence the distance function is the absolute difference between two versions of a data item.

In this work, we assume the limits for maximum acceptable inconsistency to be dynamic as most of the E-commerce applications can tolerate a greater amount of inconsistency when a commodity is aplenty than when the commodity is sparse. Therefore, we specify the bounds for maximum allowable inconsistency to be a percentage of the current value of a data item over which the distance function is defined.

Generating Invalidation Messages

Whenever a reply message is sent to a client C, the server calculates the list of objects cached by C for which it has to send an invalidation message. The invalidation to be sent is a subset of Invalid(C). In order to generate the invalidation list from Invalid(C), we check the epsilon serializability criteria for each object invalidation message in Invalid(C). Specifically, we check the safety criteria for epsilon serializability. The safety criteria involve the calculation of the inconsistency accumulated by a client for an object, which is basically the difference between the current value of the object and the value of the object in RPT (the value accessed by the client's transaction). If the accumulated inconsistency goes beyond the thresholds import-limit and export-limit, then the safety criteria are considered to be violated. If the safety criteria are satisfied for an object in Invalid(C), then that object's value accessed by C is considered to be valid; otherwise an invalidation message for that object is added to the piggy-back list to be sent along with the reply message.

Commit-time Validation

Upon receiving a commit request from a client C, the server validates the data accessed by the client's transaction. The server validation also involves calculating the inconsistency accumulated by the client for each object accessed by the client's transaction. The server checks to see if all the objects accessed by the transaction have their accumulated inconsistency within the specified bounds. If so, then the transaction validation is successful; otherwise the validation fails and the transaction aborts.

Variant of OWCC

In this report, we consider three variants of OWCC for a performance study. The three algorithms differ in how they implement update actions, namely: invalidation, propagation and dynamically choosing between the two (hybrid). In the invalidation variant of OWCC, the update notification message results in the client purging the stale page/object from the client's cache. The OWCC algorithm provided in the previous section belongs to the invalidation variant of OWCC. In the propagation scheme, the server piggy-backs the updated objects onto the regular reply message sent to the client. Hence, when the client receives the piggy-backed updates, it installs the new version of the objects in its cache.

The hybrid scheme chooses dynamically between invalidation and propagation. The heuristic used for this scheme is based on the popularity of the objects at the server. The objects in the database are characterized as *HOT* and *COLD*, based on the popularity and demand for the object. Since the popularity of Web objects follows Zipf Law [BC98], we use a similar characterization for characterizing HOT and COLD objects. We use a coarse classification that 70% of the requests are for 30% of the objects, which is also found to be true for client's access pattern on the Web and follows a Zipf-like distribution [BCF$^+$99]. Hence, 30 percent of the objects in the database are characterized as HOT and the remaining 70 percent as COLD. The heuristic for this scheme uses propagation for HOT objects and invalidation for COLD objects.

Chapter 4

Simulation Model and Experimental Setup

This chapter presents the experimental setup for the performance evaluation of the algorithms described in Chapter 3. First, the E-commerce simulation model is described and then some of the vital parameters and costs involved in our experimental setup are discussed. The basic architecture of E-commerce services is similar to that of a traditional transactional client-server system. Hence, the core of the E-commerce simulation model is that of a client-server system. However, the environment exhibits a number of other characteristics that are based on its dependence on the Internet, mainly due to the characteristics of the underlying network infrastructure, protocols used, and the access pattern of users (workloads). The simulation model addresses all of these characteristics while the parameters/costs involved are tailored to best suit caching on the Web.

The system model and the simulation work are based in part on previous work [ÖVU98]. The simulator was developed using SIDE [Gbu96] (Sensors In Distributed Environment), a programming language for expressing protocols and network configurations. SIDE provides many features for simulation and is a realistic simulation package for modeling reactive systems. The systems implemented using SIDE resemble event-driven finite-state machines.

4.1 System Model and Settings

In order to study the performance of caching for E-commerce transactions, we have constructed a detailed simulation model of a client server DBMS that captures the nature of the Web. The basic system model is similar to the E-commerce architecture described in Figure 1.1, and consists of an underlying data-shipping client-server DBMS as shown in Figure 2.1. The basic client server DBMS setup is similar to previous client server performance studies [ÖVU98, Gru97].

The structure of the client and server simulation model is as shown in Figure 4.1. A transaction operation is basically a read or write operation on objects in the database. The system model includes the clients, the server, disks, CPU, and the database, all of which are described in detail below.

4.1.1 Client

The clients consist of a group of independent parallel processes that communicate with each other in an interrupt-driven fashion. The client processes include: a client manager that manages the transaction execution at the client workstation; a resource manager that manages the physical resources of the client workstation; a buffer manager that manages the client buffer pool and is also responsible for cache consistency (i.e, locking); and the network process that is responsible for managing the transmission and receipt of messages. Clients send object requests to the server and cache the pages that are returned by the server. The clients use a page-level data buffer for caching pages at the client workstation buffer pool; the objects used by the clients' transactions (logs) are stored in an object-level log buffer. All updates are executed against the data cached at the client workstation. The page buffer uses an LRU-like (second chance) buffer replacement policy. The client CPUs have a high priority and a low priority queue. The high priority queue is used for dealing with system requests while the low priority queue deals with user requests. The high priority queue is modeled as a *first-in-first-out* (FIFO) queue, while the low priority queue is shared equally by all the processes at the client and is also modeled as a *first-in-first-out* (FIFO) queue.

4.1.2 Server

The server components are similar to client components except that disks are modeled at the server. The input to the server comes from the network manager, which passes the requests of the clients to the server. The server includes a server process, resource manager, buffer manager, network manager, and one or more disk managers.

The server models a buffer manager in detail; it caches the pages it returns to the clients and uses a LRU-like (second chance) buffer replacement policy similar to the client's. The server maintains a table for storing the clients' cache status for maintaining cache consistency. The server is responsible for consistency/concurrency control and is also responsible for sending and receiving messages. The server encounters two CPU costs associated with message transmission,

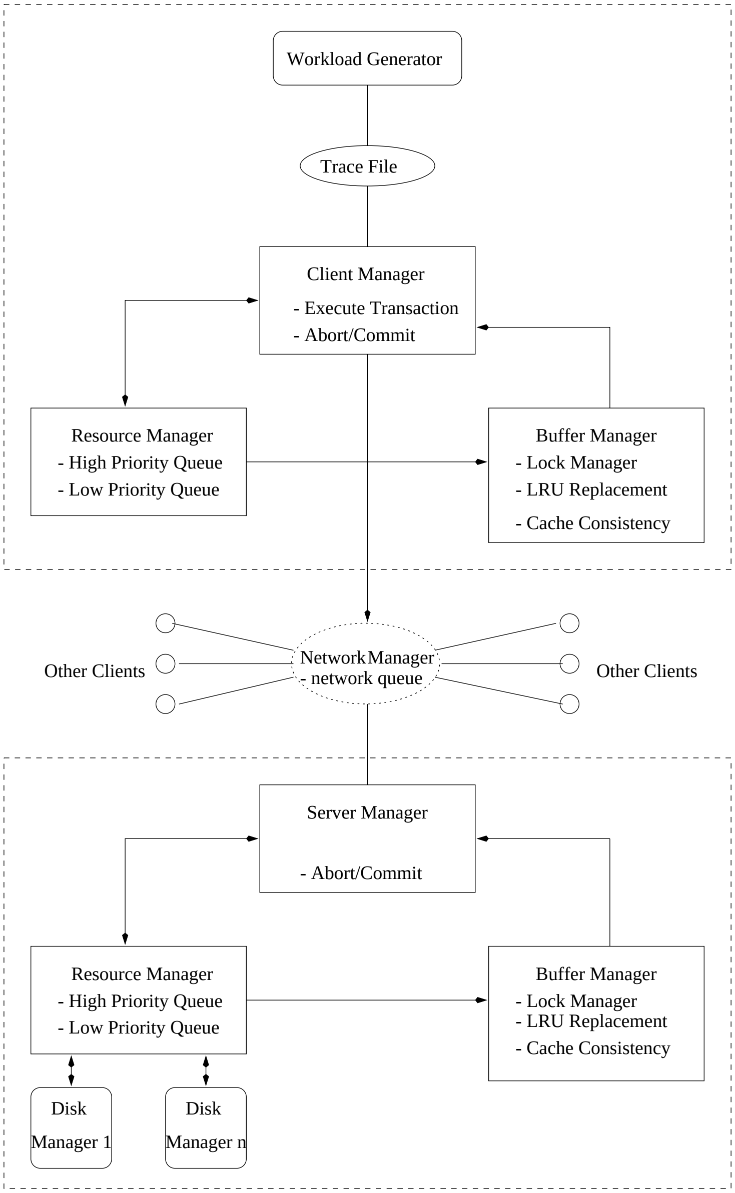

Figure 4.1: System Model

Cost Type	Value
Client CPU Speed	50 MIPS
Server CPU Speed	150MIPS
Client Buffer Size	25% of DB size
Client Log Buffer Size	2.5% of DB size
Server Buffer Size	50% of DB size
MOB	50% of DB size
Server Disks	4 disks
Fetch Disk Access Time	1600 microsecs/Kbyte
Inst Disk Access Time	1000 microsecs/Kbyte
Fixed N/W Cost	6000 cycles
Variable N/W Cost	7.17 cycles/byte
Disk setup Cost	5000 cycles
CacheLookup/Locking	300 cycles
Register/Unregister	300 cycles
Validation Time	300 cycles/object
DeadlockDetection	300 cycles
CopyMerge Instr	300 cycles/object
Database Size	1000 pages
PageSize	4 K
ObjectSize	100 bytes
Number of Clients	200 to 1000

Table 4.1: List of Costs for Different System Parameters

namely, variable network cost and fixed network cost. These reflect the processing overhead due to message transmission. The server contains a high priority and a low priority CPU input queue, which are similar in functionality to the client CPU input queues. The server models three buffers, namely, data buffer, log buffer, and modified object buffer (MOB). The server incurs CacheLookUp cost while accessing data from the cache or modified object buffer (MOB) buffers.

4.1.3 Disk

Upon a server cache miss, the server places a request for data to be retrieved from the disk. The disk consists of a disk manager process that manages I/O operations. The disk manager models a FIFO input queue for each disk, and uniformly selects one of the disks from the existing set, placing the request on that disk's input queue. For the disk I/O operations, we assume a slow disk bandwidth for read operations, and a fast disk bandwidth for installation of read/write operations for data stored in the MOB. The disk bandwidth was calculated based on *Seagate Barracuda disk* drives whose average seek time is 8.75 ms, rotation time is 8.33 ms, and average transfer rate is 0.37 ms for 4 Kbytes of data.

4.1.4 CPU

The client and server CPUs are modeled with a clock at microsecond granularity. The clock is advanced based on the time taken by an event to execute at the processor, and time taken for executing an operation is based on the costs associated with it. The costs associated with various operations are expressed as instruction cycles and are converted to microseconds based on the CPU speed. We use a client and server processor speed of 50 MIPS and 150 MIPS, respectively. Events queued at the CPU's input queues are executed based on the timestamp on the events. For messages sent, the network module computes an arrival time for that message, and places an event in the CPU queue with the calculated start-time for receiving that message. An event starts execution only at its start time, and can sometimes start late if the processor is heavily utilized.

4.1.5 Database Model

The database is represented as a set of pages. A page is a physical unit of storage which contains a number of objects stored in it. In this work, we consider the page and object sizes to be fixed at 4KB and 100bytes, respectively. The database consists of 1000 pages and each page consists of 40 objects.

In order to calculate the consistency of data accessed by clients, a virtual database was simulated that keeps track of the depletion of items as exhibited by an online store. The virtual database consists of a table, which holds the following fields: page ID, object ID, and quantity of items available. The table keeps track of the availability of items for each individual commodity through an update transaction which changes the availability field of an item. This table, along with other client-specific caching information, is used to calculate the consistency of data accessed by the clients.

4.2 Network Model

The simulator is modeled to best capture the characteristics of the Internet. We consider a generic model, rather than a particular Internet topology. The main characteristics that can be observed on the Internet are variations in client population and unpredictable delays associated with message transmission [Pax97, FFF99, FGHW99]. The network manager consists of a network queue and is modeled as a FIFO queue.

4.2.1 Client Population

The client's connection to the Internet shows high variation, so, in order to effectively model the Internet we need to address client variation. We, therefore, use a mixed population by defining classes of clients based on their connection to the Internet. Although the average connection speed of clients to the Internet is 56 Kbps, because of the continuous growth of the Internet, it cannot be clearly determined what percent of the client population is currently connected to the Internet and at what bandwidth. A reported survey [GVC98] indicates that 70% of the clients connect to the Internet with less than or equal to 56 Kbps, approximately 20% at 1Mbps, and less than 10% at higher rates. We have used a futuristic approach as this study was conducted in 1998 and there have been significant changes in the growth of bandwidth over these two years. The simulation parameters for the different classes of the user population are shown in Table 4.2. Although the percentage of user population for each class is the same as the GVU's survey [GVC98], we assume that their connection to the Internet changes as per Table 4.2. The bandwidth associated with each class is not of importance since our simulation model uses only the delay associated with each class of users.

User Class	% of population	Bandwidth	Avg. delay(msec)
Class 0	70 %	Less than or equal to 56 Kbps	200
Class 1	20 %	Less than or equal to 10 Mbps	150
Class 2	10 %	Less than or equal to 100 Mbps	100

Table 4.2: Client Population Model

4.2.2 Network Costs

The network costs consist of *fixed transmission cost, variable transmission cost*, and *wire-line propagation cost*. Every message has a fixed sending and receiving cost associated with it, which is assumed to be 6000 cycles. The variable cost component of the message is determined by its size and is assumed to be 7.17 cycles/byte [ÖVU98, Gru97]. One of the vital characteristics of the Internet is that it encounters unpredictable delays with message transmission [SCH$^+$99, Pax97]. The delays depend on the geographic location of clients and server, the underlying network path, the background traffic at the time of day, and the client's connection speed. The delays along a particular path can be modeled by a shifted gamma distribution but the parameters for the distribution are path-specific and do not scale according to a generic model [Muk94, Pax97]. The message delay is affected by many network parameters, such as, data packet loss, acknowledgment loss, packet corruption, the geographical location of the clients and servers, the number of nodes along a path, and the available bandwidth along a path. There is no empirical measure for any of these parameters that could best fit the nature of the generic Internet topology. Hence we were not able to come to any conclusions regarding the dynamics of packet delay. The network model simulates a switched network, where the clients use a point-to-point connection to a network switch and the network switch uses a point-to-point connection to the server. The message passing through the switch is modeled to experience an unpredictable delay associated with traveling between the switch and the server, and vice-versa. The delay refers to the wire line propagation cost and is modeled to be Poisson-distributed, with a mean as shown in Table 4.2.

4.3 Workload Model

Understanding the nature of the workload and system demands created by users of the World Wide Web is crucial for performance evaluation. Due to the continuous growth of the Internet, a benchmark for E-commerce database workloads is still an open problem. One of the earliest works to characterize Web workloads was done by Arlitt et al. [AW96]. They analyzed six different Web traces, and charac-

Workload Class	File Size	Access %
Class 0	Less than 1 K	35 %
Class 1	1 to 10 K	50 %
Class 2	10 to 100 K	14 %
Class 3	100 to 1000 K	1 %

Table 4.3: SPECweb99 Workload Model (taken from [spe99])

Requests	Percentage (%)
Static GET	70
Standard Dynamic GET	12.45
Standard Dynamic GET(CGI)	0.15
Customized Dynamic GET	12.6
Dynamic POST	4.8
Total	100

Table 4.4: SPECweb99 Request Type Distribution (taken from [spe99])

terized some commonly observed invariants that apply to all the traces. In this section, we look at two popular benchmarks that characterize Web workload, namely, SPECweb [spe99] and SURGE [BC98]. We then attempt to model an E-commerce database workload by putting together various observations regarding the characteristics of Internet server workloads and client-server DBMS workloads.

4.3.1 SPECweb

SPECweb is a benchmark developed by SPEC (Standard Performance Evaluation Corporation) to measure a system's ability to act as a Web server serving static and dynamic Web pages. SPECweb99 is the latest Web workload characterization by SPEC which defines classes of files based on their size and popularity. Table 4.3 shows the relative access percentage distribution, based on the file size.

It should be noted that the relationship between file size and access percentage follows the heavy tailed distribution observed in the Web, while the access pattern of files in each class follows the Zipf law. The benchmark also accounts for dynamic pages, which comprise 30% of the total requests. Table 4.4 summarizes the different types of requests and their respective percentage in the workload mix. The static GET refers to static URL requests to a Web server, while both standard dynamic GET, and standard dynamic GET(CGI) refer to the requests to scripts that simulate and model simple advertisement rotations by commercial Web servers. The customized dynamic GET models the server tracking the user's preferences

Components	Distribution Model
File Size	Lognormal + Pareto
Popularity	Zipf
Temporal Locality	Lognormal
Request Size	Pareto
Active OFF times	Weibull
Inactive OFF times	Pareto
Embedded Reference	Pareto

Table 4.5: SURGE Workload Model (taken from [BC98])

to provide customized ads for the user. The dynamic POST request models user registration at an ISP site, and refers to posting logfiles to the ISP sites.

4.3.2 SURGE

SURGE (Scalable Url Reference GEnerator) [BC98] is a comprehensive Web workload generator which captures most of the essential characteristics of the demands placed on the Web server by end users, and brings about an empirical distribution on the nature of Web workload characteristic for static Web pages. Table 4.5 shows the distribution used by SURGE for various components of Web workloads.

In SURGE, users are modeled as an ON/OFF process; the ON period refers to periods during which files are transferred while the OFF period refers to idle periods. Hence, the users are modeled as bursty processes and the resultant traffic exhibits self-similarity at high workloads [BC98]. Typically, a user's request for a single Web page will result in requests for multiple embedded objects that form that Web page. This property of the Web shows high variability and is modeled as a Pareto distribution. Since one request results in further multiple requests, SURGE models two types of OFF time - active and inactive. An active OFF time represents the think time between two requests for Web objects, while inactive OFF time represents the think time between two user requests. The popularity of files at the server is modeled to follow Zipf law. The temporal locality of Web objects refers to the likelihood of repeated requests for the same Web object, and is modeled to follow an, similar to that exhibited by most Web servers. The request size has an impact on network usage, and refers to the size of the set of files transferred in response to a request from the client. It also exhibits high variability, and so a heavy tailed distribution like Pareto distribution is used.

The SURGE model captures the high variability in demands experienced by Web servers, and also addresses the self-similarity in network traffic generated by Web requests [BC98]. A comparison of the performance of SURGE with

Parameters	Settings
Transaction Size	200 objects
Cluster Size	5 obj/page
Work Allocation at client	50%
Object Write probability	10%
Read access think time	50 cycles/byte
Write access think time	100 cycles/byte
Think time between trans	0
Hot pages	30% of DB pages
Abort Variance	100%

Table 4.6: List of Workload Parameters

SPECweb96, an earlier version of the SPEC benchmark, concludes that most Web workloads do not adhere to self-similar traffic and perform poorly at higher loads, while SURGE is scalable to high loads [BC98].

4.4 E-Commerce Workload Model

In our workload characterization for E-commerce services, we model the effects of high variability in workloads, as exhibited in the Web, and use SURGE for capturing these characteristics. The database object popularity, and temporal locality of requests, are modeled such that they exhibit the characteristics of Web workloads. These properties are crucial for a cache consistency performance study as the popularity of a data item in turn dictates the relative contention for the data item; moreover, popular objects tend to remain cached when compared to unpopular objects. The temporal locality directly influences the cache hit rate as it represents the number of repeated requests for a Web object. In order to capture the database workload features, we use the multi-user 007 benchmark [CDN93]. The 007 benchmark has been developed to study the performance of object DBMSs. An E-commerce transaction consists of many operations, and each operation accesses many objects from a database page. The workload parameters are listed in Table 4.6. Each parameter is explained in detail below.

Transaction Size and Object/Page Size

Transaction size refers to the number of objects accessed by a transaction. The transaction size, object size, and page size are similar to previous cache consistency studies [ÖVU98], and conform to object DBMS applications. Since there is no measurement available on these parameters for E-commerce applications, we will

User Class	% of population	# of items
Class 0	50 %	between 1 and 3
Class 1	30 %	between 3 and 6
Class 2	20 %	between 6 and 10

Table 4.7: Client Purchasing Model

adhere to values proposed for object DBMSs as most Web objects have the same characteristics as objects in an object DBMS. The transaction think time is the time interval between two consecutive transactions at the clients. The transaction think time, database size, and buffer sizes are similar to previous client server cache consistency studies. These values are specified in Tables 4.1 and 4.6.

Clustering

Clustering refers to the grouping of objects that are likely to be accessed together, and the storing of those groups in a database page. The cluster size represents the number of objects accessed per page by an operation. Under our workload, a cluster can be revisited by a transaction and is greatly dependent on the distribution used for temporal locality and object popularity.

Write Probability

The write probability determines whether any of the objects in an operation will be updated. It can be observed from [per99] that the percentage of Web transactions which actually start an update transaction is about 5.75% (about 8.6% for some of the most popular sites). Our simulation uses a write probability of 10 percent that is uniformly distributed over all the clients access pattern.

Client Purchasing Habits

The quantity of an item that each client purchases is different for different clients as the clients' purchasing habits are not uniform. This value has a potential impact on the performance of the consistency algorithms as update transactions of a client can cause the caches of other clients to contain stale data. We use a mixed distribution for modeling this behavior. The parameters used are shown in Table 4.7. The client population is divided into three classes, and the buying pattern of clients in each class is uniformly distributed between the min and max values. The percentage distribution is modeled to show non-uniform characteristics, similar to the Zipf distribution.

Data Sharing Pattern

The data-sharing pattern dictates the number of read/write and write/write conflicts. The data contention level is an important component for a cache consistency/concurrency control performance study. Although it is still not characterized in detail for E-commerce and the Web, it is mainly dependent on object/page popularity (Zipf Law). Hence, the most popular object/page is likely to be accessed by most of the clients. In this work, the data-sharing pattern is implicitly brought about by the uniform distribution for write probability over the access pattern of clients.

4.5 Generating a Transaction

We use SURGE for generating a sequence of references to database pages and objects, which follows the characteristics of the Web workload. In particular, we are interested only in generating a sequence of requests that captures the page/object popularity (Zipf law) and temporal locality of references to the same objects (log-normal distribution). The resulting sequence, produced by SURGE, is our sequence of transactions and is stored in a master file. The clients access these transactions from this master file on a first-come-first-served basis, and there are no repeated accesses to the same transaction from the sequence. Each client takes this transaction operation, which is basically a reference to the page involved in the transaction, and accesses a number of objects from that page. The number of referenced objects depends on the cluster size, and the objects accessed from a page are assumed to be uniformly distributed. Updates on the objects are based on the object write probability, which is also considered to be uniformly distributed over all the clients' requests. Upon an update operation, the client process generates the quantity of items to be purchased, for a particular data item. The clients repeat the process of generating a transaction till they reach the end of their transaction. The client internally stores the trace of all references to objects it makes. When the transaction aborts, the client uses this trace to restart the transaction. The abort variance determines if an aborted transaction accesses the same set of objects as the original transaction, or if it accesses a different set of objects. We use an abort variance of 100 percent, which means that the restarted transaction accesses the same objects as before. The work allocation scenario is similar to previous studies and we use a 50-50 split between client and servers.

Chapter 5

Experiments and Results

In this chapter, we present our main experiments and results for different variations of the OWCC algorithm for Web workloads. The three versions of OWCC algorithms under study are Invalidation, Propagation, and Hybrid. The study compares the performance of these three base algorithms with different levels of relaxation of consistency. The system and parameter settings are as described in Chapter 4. The main goals of the performance study are to investigate the performance of the proposed relaxed cache consistency algorithm for Web workloads and to demonstrate the fact that relaxing consistency reduces server load and, therefore, response time. We also study the performance of the OWCC algorithm for varying system parameter settings. The system throughput (commits/sec) is the primary measure of performance and is used to evaluate the response time of the system as it scales to a large number of clients. The secondary measures include cache hit/miss ratio, number of messages, and number of aborts, for varying numbers of clients and cache sizes.

Our simulator was not designed to predict the performance of a specific database system but rather to explore the relative performance of different algorithms. Hence, we are more interested in the relative positioning of the curves for different algorithms than the specific values they exhibit. The primary work involves the performance study of the impact of a large number of clients on the system, while the secondary one involves the study of the impact of cache size on the performance of different algorithms. The experiments were repeated several times in order to check for variance in the values of the results obtained. The results were found to be stable for all runs of the experiments and there was no variance exhibited in the results.

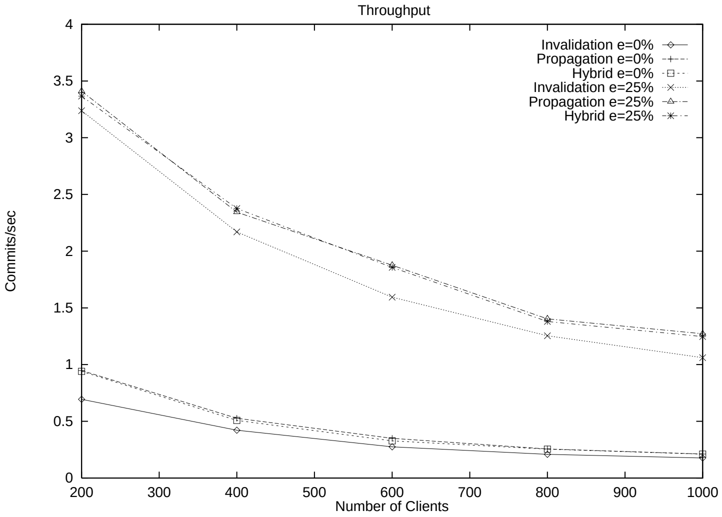

Figure 5.1: Throughput
write prob=10%, client cache size=25%

5.1 Impact of Large Number of Clients

In this section, we study the impact of a large number of clients on the system throughput, and present the performance results from testing the scalability of the different algorithms. The number of clients was varied from 200 to 1000, while the cache size was fixed at 25% of the size of the database and 10% of all operations were update operations (write probability). Two sets of readings for the three algorithms are shown in Figures 5.1 to 5.6. One refers to the case where epsilon is 0%, and hence, represents the result assuming no tolerance in consistency. The other represents the case where epsilon is 25% and, therefore, illustrates a relaxation in consistency by an absolute value of 25% of the quantity for a commodity available in the database. We can see from Figure 5.2 that relaxing consistency achieves better performance, which is mainly due to the reduction in the number of aborts that is brought about by having an epsilon value of 25%. The response time between the case when epsilon is 0% and 25% increases drastically when the number of clients goes beyond 600. When the number of clients is 1000, the performance gain of the relaxed scheme is about five-fold when compared to strict serializability, i.e., epsilon of 0%. Thus the performance of OWCC with a small epsilon value increases with an increasing number of clients. Hence relaxing consistency reduces the number of aborts and therefore reduces the server load, making the system more scalable.

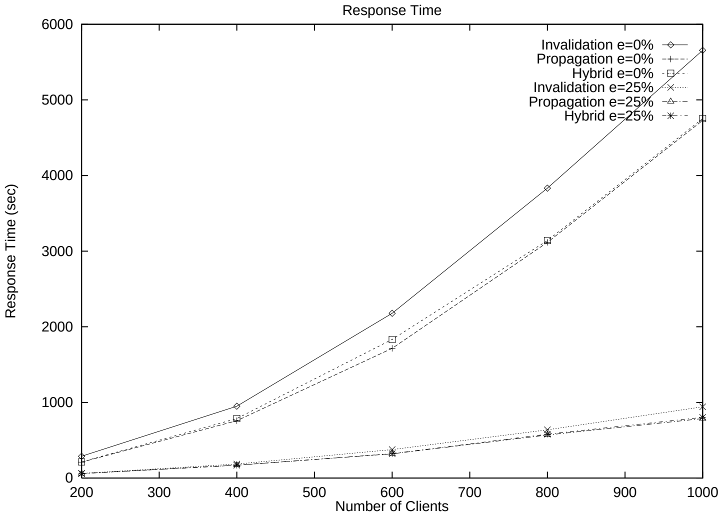

Figure 5.2: Response Time
write prob=10%, client cache size=25%

We can also observe from this experiment that the performance of the propagation and hybrid schemes is slightly better than that of the invalidation-based scheme. This is mainly because of the restart behavior. When a transaction aborts and restarts we assume that the transaction accesses the same objects as before and, therefore, the propagation and hybrid schemes have higher cache hit rates than the invalidation scheme. The performance difference between the propagation and the invalidation schemes increases gradually with an increasing number of clients when epsilon is 25% over that when epsilon is 0%. Hence, for a larger number of clients the performance gap between invalidation and the other two methods is expected to rise. We can also observe this phenomenon in Figures 5.2, 5.3, and 5.4 with respect to the number of aborts, number of messages, and response time.

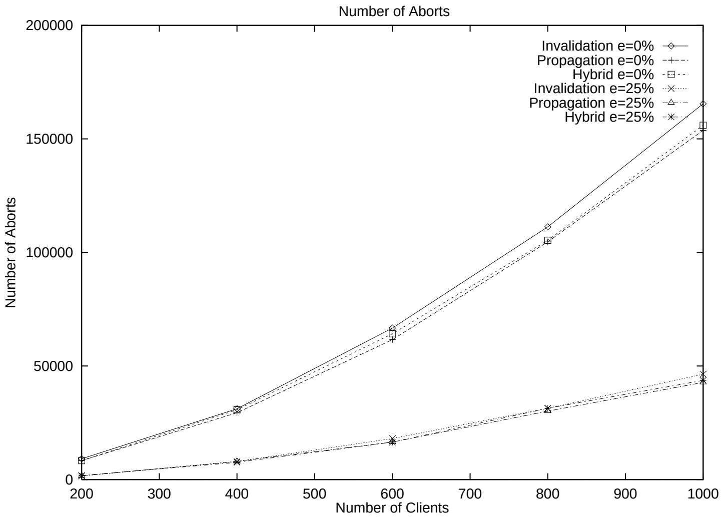

Figure 5.3: Number of Aborts
write prob=10%, client cache size=25%

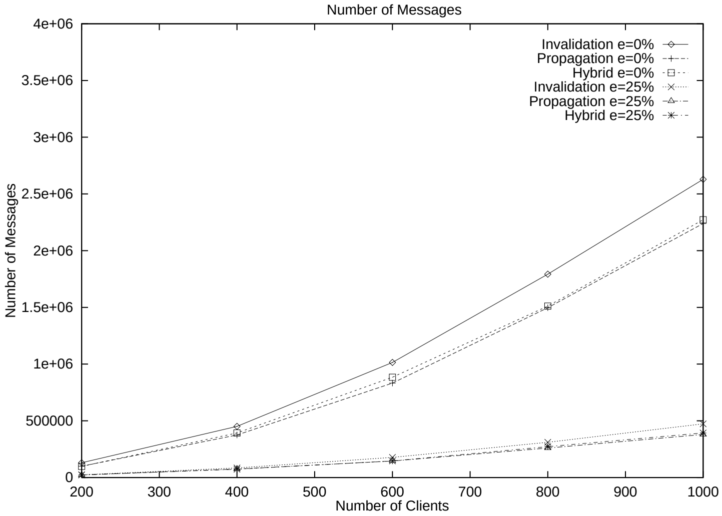

Figure 5.4: Number of Messages
write prob=10%, client cache size=25%

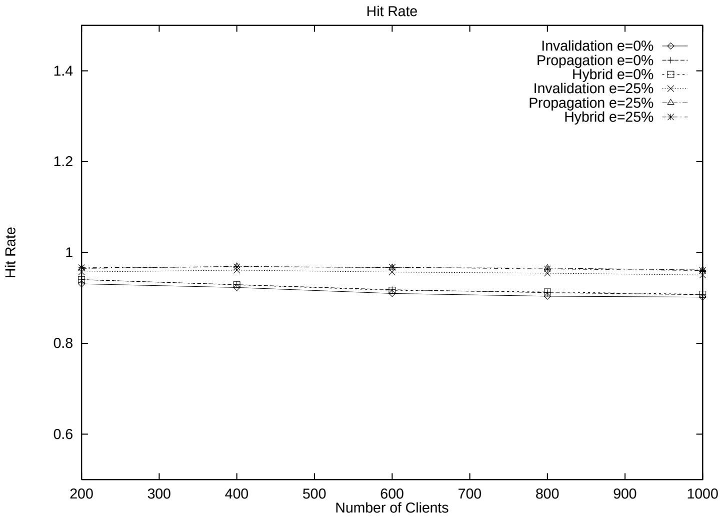

Figure 5.5: Hit Rate
write prob=10%, client cache size=25%

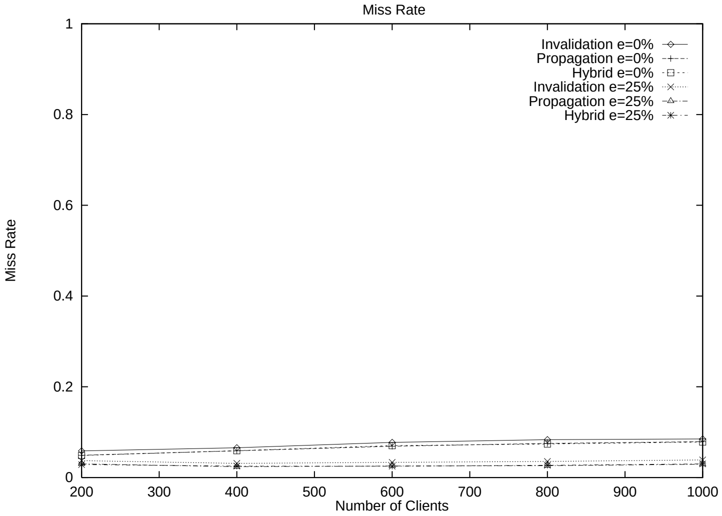

Figure 5.6: Miss Rate
write prob=10%, client cache size=25%

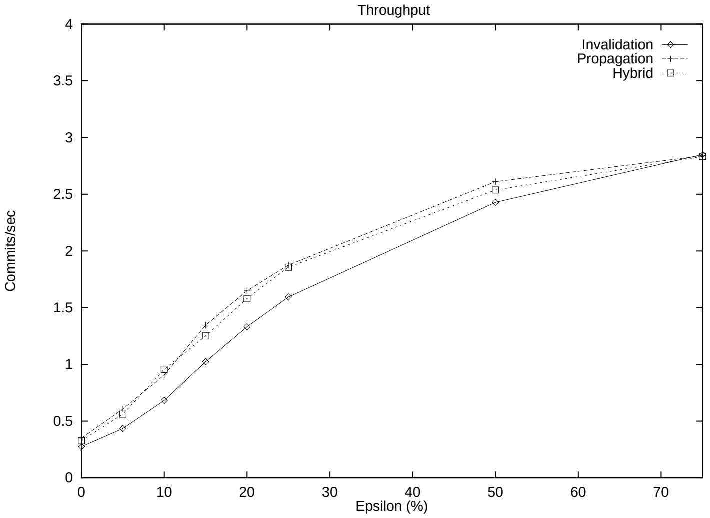

Figure 5.7: Throughput
write prob=10%, # of clients=600, client cache size=25%

5.2 Impact of Epsilon Value

This experiment captures the sensitivity of the algorithms' performance for varying epsilon values. The experiment was conducted for 600 clients where the write probability and cache size were fixed at 10% of the transaction length and 25% of the database size, respectively. In our implementation, we consider epsilon to be a percentage of the availability of a commodity in the database, so we use a single epsilon value to specify the maximum tolerable inconsistency for both read and update transactions. We can observe from Figures 5.7 to 5.10 that all the algorithms become less sensitive to epsilon value as it goes beyond 15%. This behavior is mainly because of the contention for a data item and the window of staleness produced by the epsilon value for a data item. The window of staleness represents the maximum inconsistency allowable for a data item. Since the quantity of a commodity changes with time, the window size also changes with time. Hence the window size depends on the inventory for a particular data item. For example, let the inventory for a commodity be 10 items (quantity) and epsilon be 25%; then the window of staleness is 10 * 25% = 2.5. When the quantity of a commodity reduces, the window size also decreases. The window size and popularity of a commodity and, therefore, the resulting contention must match in order to maintain a low abort rate. In our experiments, the values for the quantity of the commodities in the database and, hence, the window size and contention produced by the Web

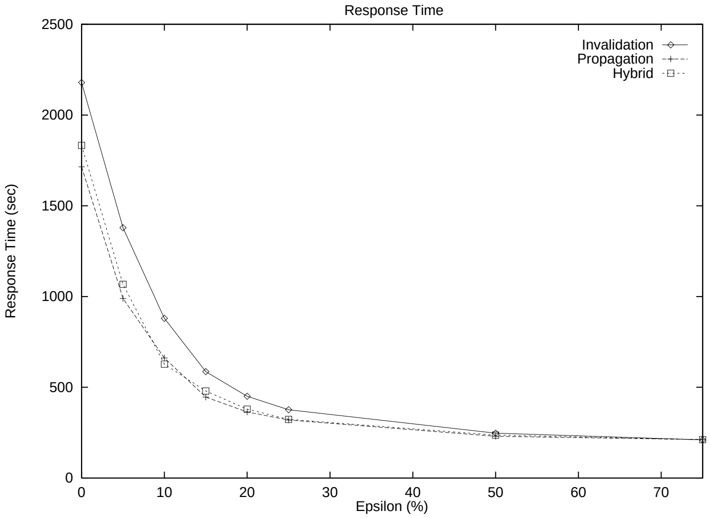

Figure 5.8: Response Time
write prob=10%, # of clients=600, client cache size=25%

workload, is such that the number of update transactions for a commodity matches with the allowable window of staleness. This matching occurs when epsilon is greater than or equal to 15%, and so there are fewer aborts. Therefore, the maximum benefit of having a relaxed consistency for our system configuration can be reaped when epsilon is at 15%. For larger values of epsilon, the window size is much larger than the contention level and so the values accessed by other clients are valid for a longer time; therefore there are very few aborts. For smaller values of epsilon, the window size is small and, if the contention level for a data item is high, then this may result in more aborts. We can also observe from Figures 5.7 to 5.10 that the performance of the propagation and hybrid schemes are better than the invalidation scheme but, for large epsilon values, all algorithms demonstrate a similar performance. This is mainly due to the restart behavior, as discussed in the previous section, which shows that propagation and hybrid schemes have an advantage over invalidation due to greater cache hits.

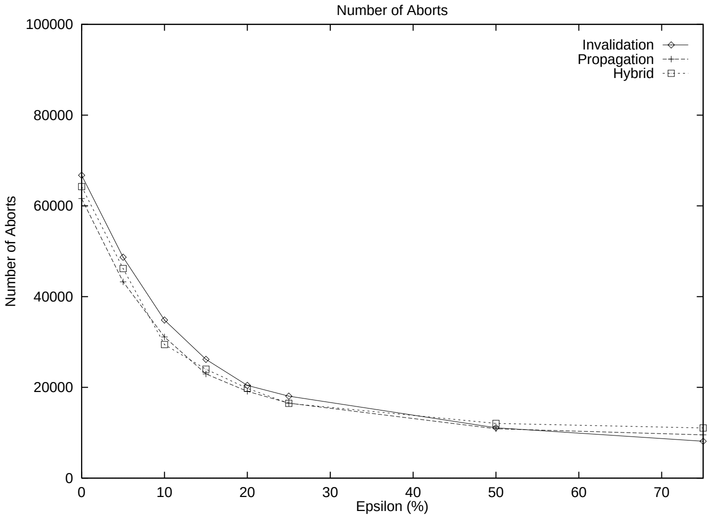

Figure 5.9: Number of Aborts
write prob=10%, # of clients=600, client cache size=25%

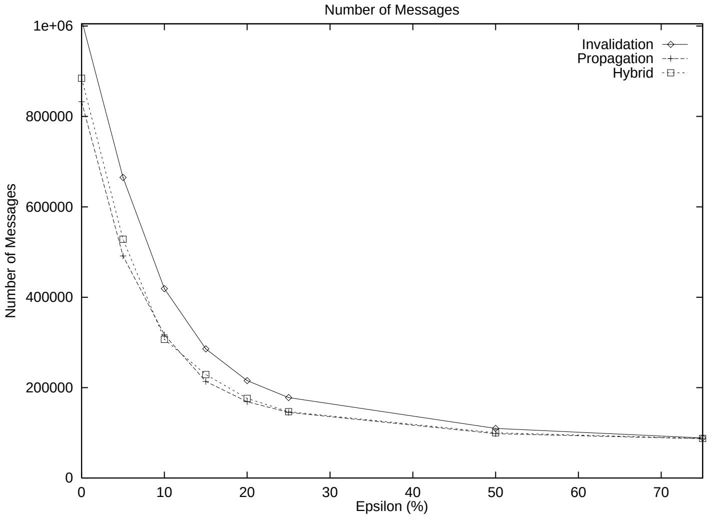

Figure 5.10: Number of Messages
write prob=10%, # of clients=600, client cache size=25%

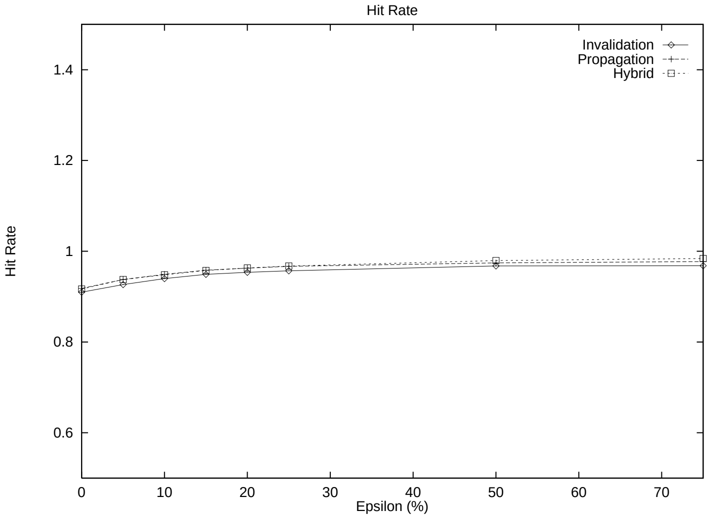

Figure 5.11: Hit Rate
write prob=10%, # of clients=600, client cache size=25%

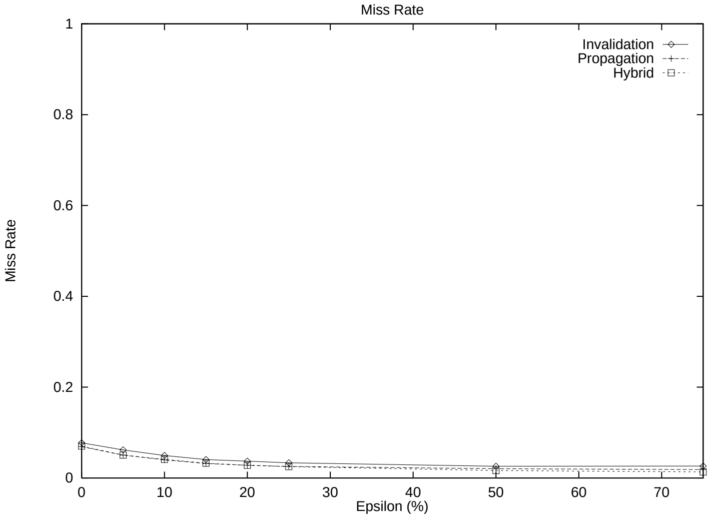

Figure 5.12: Miss Rate
write prob=10%, # of clients=600, client cache size=25%

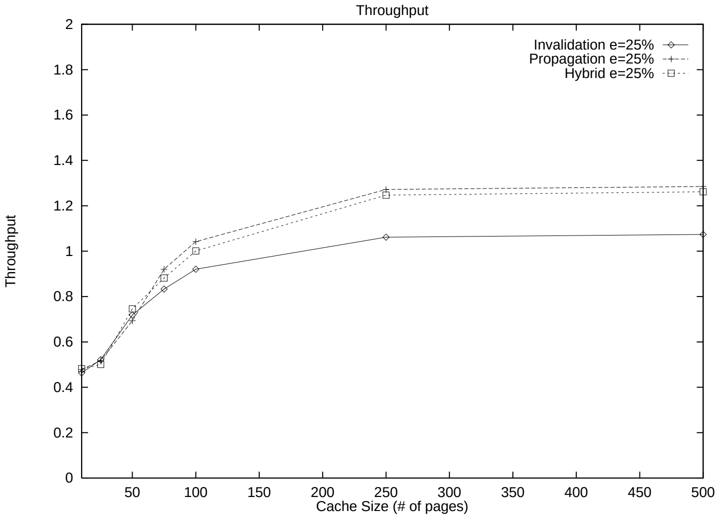

Figure 5.13: Throughput
write prob=10%, # of clients=1000, epsilon=25%

5.3 Impact of Client Buffer Size

In this experiment, we evaluate the impact of cache size on the performance of the algorithms. Although the client cache size is steadily increasing, application demand usually outstrips the available resources [ÖVU98]. Most of today's Internet applications have large transaction sizes and deal with large multimedia objects, such as images, video, and audio. Therefore, cache size influences the performance of a cache consistency algorithm. Here we vary the client cache size by keeping the number of clients, database size, and transaction size a constant. The performance graphs involved for this study vary the cache size along the x-axis; and plot the resulting system throughput, number of messages, number of aborts, and hit ratio along the y-axis. The experiment was carried out for epsilon value of 25% and 1000 clients. The client cache size was varied from 1% to 50% of database size, shown by the graphs with respect to the number of pages. From Figures 5.13 to 5.16, we can see that performance shoots up as the cache size increases up to 10% of the database size, and that it stabilizes once the cache size goes beyond 25%. This behavior of the algorithm is mainly due to the length of the clients' transactions. In our simulation, we assume that each transaction accesses approximately 40 pages. Hence, the performance gain is dramatic during the initial phase of the curve. We also observe that the algorithms are sensitive to cache size and that the performance of the propagation and hybrid schemes is superior to the invalidation

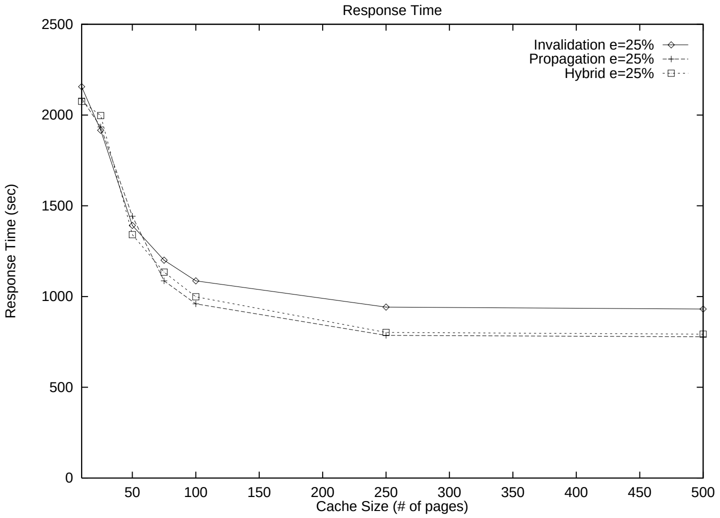

Figure 5.14: Response Time
write prob=10%, # of clients=1000, epsilon=25%

scheme when the cache size becomes larger than the number of pages accessed by the client's transaction. The main reason that the performance of hybrid and propagation techniques is better than invalidation is because, upon an abort, the client restarts the transaction and accesses the same objects and pages as before. Propagation, therefore, reduces the need for clients to contact the server for page requests, and thereby reduces the server load, response time, and the number of messages sent.

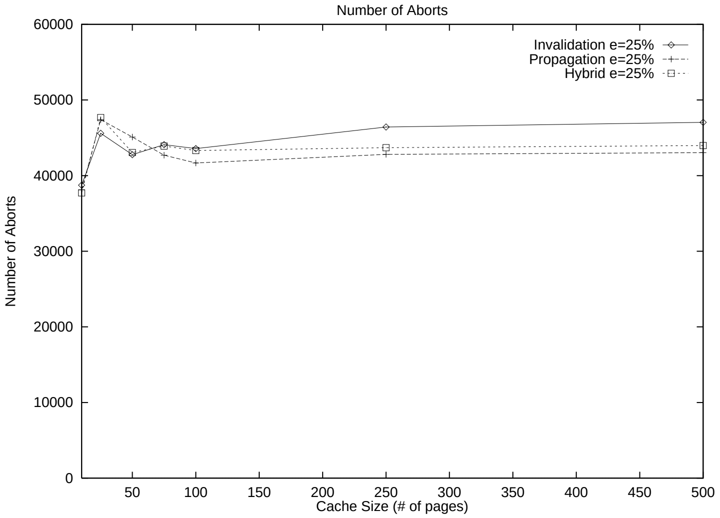

Figure 5.15: Number of Aborts
write prob=10%, # of clients=1000, epsilon=25%

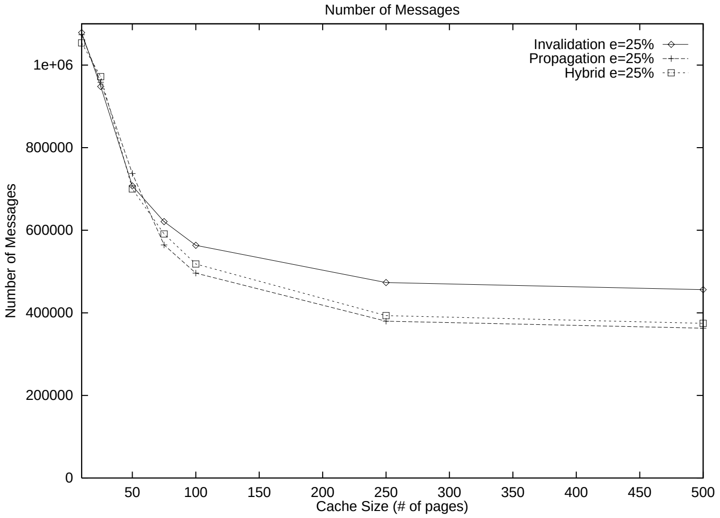

Figure 5.16: Number of Messages
write prob=10%, # of clients=1000, epsilon=25%

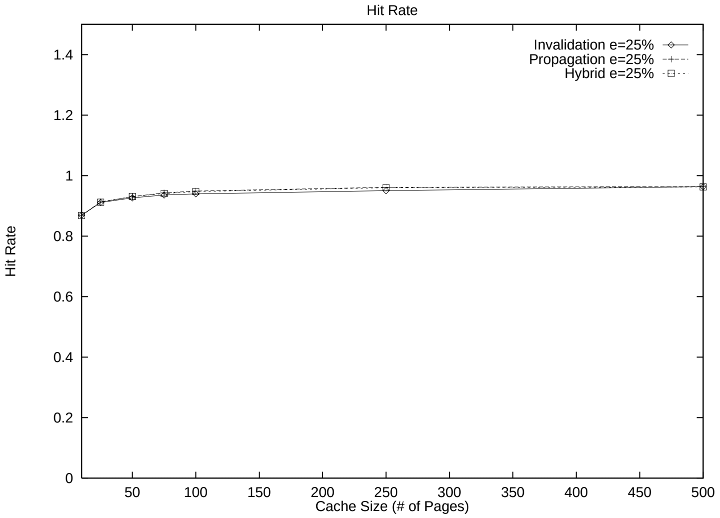

Figure 5.17: Hit Rate
write prob=10%, # of clients=1000, epsilon=25%

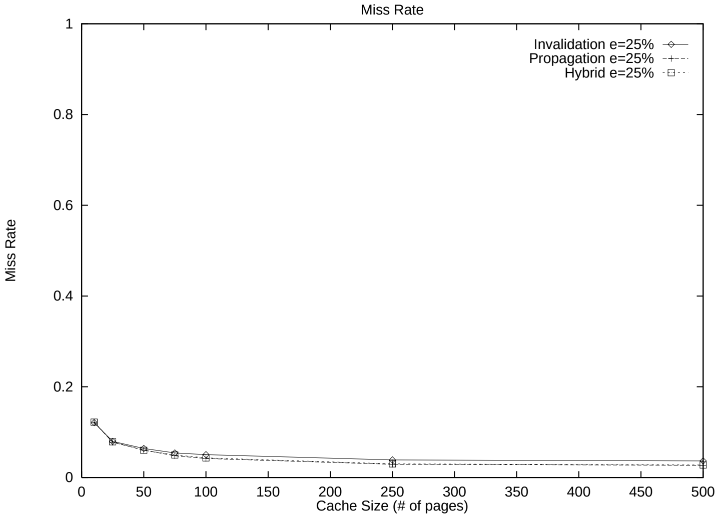

Figure 5.18: Miss Rate
write prob=10%, # of clients=1000, epsilon=25%

Chapter 6

Conclusion

6.1 Contributions

The main objective of our work was to design a Web cache consistency algorithm for Internet-based electronic commerce applications. We also evaluated the performance of our proposed OWCC algorithm and described its potential benefits for E-commerce applications. The main contributions of our work are as follows:

- We propose an optimistic cache consistency algorithm - OWCC - that exploits the tolerance exhibited by E-commerce applications with respect to consistency. In particular, OWCC relaxes the bottlenecks that are inherent to transaction processing by bounding the staleness of the data viewed by transactions. At the same time, OWCC brings about a generalized correctness criteria for correct execution of transactions.
- We present a thorough survey both of Web cache consistency algorithms and client-server database cache consistency algorithms.
- We evaluate the performance of the OWCC algorithm for Web workloads, and show that relaxing consistency reduces the response time and, hence, the server load and hence improves the throughput of the system.
- We present results of our sensitivity analysis study for three variations of the OWCC algorithm, which are differentiated based largely on how they implement updates. We study the impact of a large number of clients, client cache size, and epsilon value for Web workloads.
- The sensitivity analysis experiments demonstrate that propagation and hybrid techniques perform better than an invalidation scheme, but we show

that this is mainly because of the restart behavior we assume. It is generally agreed that, in the absence of detailed information about the clients' access patterns, invalidation is the safest choice for most situations [FCL97]. However, since we assume that when a transaction is restarted it accesses the same object as before, propagation appears to be a better option than invalidation. The hybrid scheme reduces the overhead due to propagation by selectively propagating objects. The advantage of this scheme depends mainly on the characterization of *HOT* and *COLD* objects. We also show that the contention level and the epsilon value for a commodity should match in order to maintain a low abort rate. The cache size and the transaction size have significant impact on the three algorithms; the cache size must be large enough to hold the pages accessed by the client's transaction. For a small client cache the performance of the three algorithms is similar.

- Another contribution of this work is the characterization of an E-commerce workload. We evaluate Web and client-server database workloads, and combine key aspects of Web workloads, along with database workloads, to come up with an E-commerce workload.

6.2 Future Work

The choice of avoidance-based or detection-based schemes for cache consistency has a significant impact on the performance of the system. Although we have determined that an optimistic algorithm best suits the nature of the Internet, there are two other algorithms whose performance is similar to that of our algorithm. In an earlier study [ÖVU98], the authors propose an asynchronous avoidance-based cache consistency scheme (AACC), and show that it performs similarly to the AOCC algorithm. In another study [Gru97], the author compares the results obtained for AOCC and ACBL and shows that, although optimism is more robust with an increase in the number of clients, frequency of updates, and data conflict patterns, there are situations for which ACBL is more suitable than an optimistic scheme, such as AOCC. He also shows that, if transactions are shorter, and for a workload mix of 98% read-only transactions, the throughput of ACBL equals AOCC and it has a clear advantage over AOCC: for a fully read-only workload, ACBL outperforms AOCC by 25.2%. Since a majority of Web services are read-intensive, the performance of ACBL and AACC is of great interest for Web workloads. Moreover, the ACBL and AACC algorithms are sensitive to latency [ÖVU98, Gru97] but, considering the growth of network speed on the Internet, the latency is expected to be reduced for future systems. Hence, the performance of AACC and ACBL is expected to improve.

AACC and ACBL can be extended by incorporating epsilon serializability and adapted for E-commerce applications. We call these relaxed versions of the AACC and ACBL algorithm the Asynchronous Web Cache Consistency algorithm (AWCC) and the Synchronous Web Cache Consistency algorithm (SWCC). Their performance for Web workloads is still an open area of research. Moreover, there are situations in which each of them performs better than the other. Hence a comparative study of AWCC, SWCC, and OWCC for Web workload is an interesting area for future work. The mechanism of SWCC and AWCC algorithms is described below:

Synchronous Web Cache Consistency Algorithm

The Synchronous Web Cache Consistency algorithm (SWCC) is an avoidance-based cache consistency algorithm. The mechanism of the algorithm is similar to the ACBL algorithm, and the extension done is with respect to how conflicts are detected and resolved. SWCC uses epsilon serializability for correctness criteria and, hence, the conflicts are dictated mainly by the relaxation provided by epsilon serializability. Therefore, read/write conflicts do not occur as long as the safety criteria (presented in Chapter 3) for epsilon serializability is maintained. Similar to ACBL, SWCC dynamically obtains page-level or object-level locks from the server, and prevents the client cache from holding stale data, thereby ensuring that the clients do not access stale data.

Request Processing

Whenever a client requests a lock on an object,

- If the object is present in the client's cache then, if it is a read request, the request is processed immediately since the client implicitly holds a read lock for pages in its cache. However, if the request is a write request, then the client sends a write lock request to the server. The server processes the request and grants a write lock on the page or object.
- If the page is not present in the client cache, then the client sends a read or write fetch request to the server, based on the lock requested by the client. Upon a read fetch, the server does not block the read transaction unless the epsilon value for that object equals zero. For a write fetch the server grants a write lock, along with the page returned, but - before doing so - it goes through the callback processing phase, which is described below. The client holds the write lock until the transaction commit time, or until it is asked to

revoke its write lock. Hence, repeated requests for a write on that page do not have to involve the server.

When epsilon is zero for an object, then the operation of the algorithm is similar to that of ACBL. The server maintains a table and keeps track of the information about each client's cache status and the lock they hold on cached pages/objects. An entry is made in the table whenever a page is sent to a client, and an entry is removed from the table when the client sends notification about page discards from its cache. SWCC also uses adaptive-granularity for locking in order to avoid unnecessary blocking since two clients can be in conflict over a page even if their transactions are not accessing the same objects.

Callback Processing

Upon a write request, the client sends a *delta* value, which represents the amount by which an object is updated. The server uses this value to compute the list of clients affected by this update (clients whose cache becomes stale due to the update by the write-requesting client), and sends a synchronous callback message to all affected clients. Each client, upon receiving the callback message, does the following:

- If the client's current transaction has not accessed the page previously, then the page is discarded and an acknowledgment is sent back to the server.
- If the client has read the page, then the callback is blocked at that client until the local transaction completes. When the transaction completes, the page is discarded and an acknowledgment is sent back to the server. Once all callbacks are acknowledged, then the server grants the requesting client a write lock. Note that the requesting client is the only client that holds the page in its cache at this point.

When a callback is blocked at a client, a *callback blocked* message is sent to the server. The server, on receiving all such messages, checks for deadlocks and, upon detecting a deadlock, the youngest transaction is aborted; all pages necessary to resolve the deadlock are discarded from the respective client's cache. Cache discards are also performed in an adaptive way. When a client receives a callback message, it checks to see if the client's transaction has read the page. If the page was not read by the client, then the page is discarded from the client's cache. If the page was read by the client, but the particular object which is called back was not read, then the client discards only the object while the pages still remain cached. The client acknowledges the callback, informing the server about the discard. Based on the type of discard performed at a remote client, the server decides to grant the lock-requesting client a page-level or an object-level write lock. Therefore, when an object level write lock is granted, then only the lock requesting client holds that object in its cache.

Asynchronous Web Cache Consistency Algorithm

The mechanism of the Asynchronous Web Cache Consistency algorithm is similar to the AACC algorithm, and the extension done is with respect to how conflicts are detected and resolved. AWCC also uses epsilon serializability for correctness criteria. Similar to AACC and ACBL, AWCC dynamically obtains page-level or object-level locks from the server and prevents the client cache from holding stale data, ensuring that the clients do not access stale data. The granularity of locking in similar to that of ACBL, and is detailed along with the description of the SWCC algorithm, above.

In AWCC, the clients implicitly obtain read locks for objects that are brought into the client's cache. When the client wants to obtain a write lock on an object, then the client checks for conflicts and, if it does not find any conflicts, it obtains a write lock and asynchronously sends a lock escalation message to the server. All clients retain their locks across transaction boundaries, enabling inter-transaction caching. The server and clients both play a role in lock management, and deal with both page and object-level locks. The server performs deadlock processing when there are lock conflicts. The clients do not block the transaction at the time they perform the write operation, but instead, a client blocks at the commit time if its updates will cause a remote clients cache to contain stale objects.

Data Request

When a client wants to read an object that is in its cache, then the client accesses the object from the cache and obtains a read lock on it. If the object is not present in its cache, the client sends an object read request to the server. When the server receives the request it does the following:

- If the object is cached at other clients with only a read lock, then the server gives the object to the new requesting client with a read lock. The server then updates its directory, indicating that the requesting client has the page in read mode.
- If the object is cached at another client in write mode, then the server checks to see if epsilon is equal to zero

  - If epsilon is greater than zero, then the object is returned to the client with a read lock. Hence, there is no blocking involved with read requests for most cases.
  - If epsilon is equal to zero, then the server checks to see if the client which has the write lock is using the object, by sending a callback message. If the client is not using it, then it changes its lock to read lock, and informs the server about its action; the server then gives the object to the new requesting client, with a read lock. Updates are made to the server's directory, indicating that the requesting client has also cached the object in read mode. However, if the client is operating on that object, then the server blocks the new requesting client.

Update Processing

When a client wants to update a read locked object, the client obtains a write lock on that object if that page is cached at the client's cache. The client then informs the server about this update by a piggy-backed or asynchronous *lock escalation* message to the server, and continues with its processing. The client also sends the delta value, which represents the amount by which the client transaction has updated the object. Upon receiving the lock escalation message along with the *delta* value (the amount by which the client updates a object), the server does the following:

- If the object resides at other clients in read lock mode, then the server sends invalidation message to all the affected clients (clients whose inconsistency exceeds the current epsilon value), and blocks till it receives a positive reply from all the affected clients. If the client receiving the callback message is not using the object, it simply invalidates the object and informs the server via a piggy-backed callback acknowledgment message. If the client is using the object, then it sends an asynchronous callback response indicating that there is a conflict. Upon receiving a positive callback reply from all clients, the server proceeds with this update or the client's transaction remains blocked until the other conflicting client's read transaction is completed.

- If none of the clients have the object, then the server table is modified to indicate that the requesting client has obtained a write lock on the object, and it proceeds with the update.

- If the page is not present in the client's cache then this results in a write fetch. The client sends a write fetch request to the server and the server, upon receiving this request, sends a callback message to all affected clients. The client's request remains blocked till all callback messages are acknowledged.

Callback Processing

When the server receives a callback response indicating that there is a conflict, then the server performs deadlock processing; and, if there are no deadlocks, the client that has performed the initial update cannot commit before the client that is reading the object. Here, server deadlock processing involves a check to see whether clients have updated objects that have been read by other clients, resulting in neither client being able to commit their respective transactions. If the server receives piggyback callback messages from all relevant clients indicating that they have invalidated the object, the server then sends an asynchronous message to the updating client asking it to upgrade its lock from read mode to write mode.

Commit Processing

At commit time, the client sends the logs to the server. The server then checks to see whether the particular client can go ahead with its commit, or whether it should remain blocked. The server also changes object write locks held by the committing client to read locks in its lock table. If a client has performed updates to an object read by many clients, then the server checks to make sure that no other client has that object in its cache for which the epsilon consistency condition is not satisfied, before allowing the update to proceed. If any clients do have the object violating the consistency constraints, then the server sends a callback message to those clients. The server allows the commit to proceed only after receiving all the pending callback messages from the necessary clients. The server then moves the logs to a persistent storage area, and also activates the other transactions that are waiting for this client to commit.

Our characterization of the workload for E-commerce services is very coarse and considers mainly the temporal locality, popularity of objects, and write probability. There are other parameters which influence the performance of a cache consistency algorithm - namely transaction length, think time between transactions, and restart behavior. The transaction length for an E-commerce transaction is not well defined. The contention level for a data item varies with the square of transaction length [Gru97]. In the case of OWCC, the longer the transaction, the more likely it will be aborted and restarted. We do not consider a think time between transactions in our current work, but the general notion is that E-commerce transactions have large think-times. Restart behavior has an impact on the performance of the algorithm, and knowledge about client access patterns is required to correctly characterize the restart behavior. New data collection is necessary to gain insights into the nature of E-commerce transactions and client behavior for correctly characterizing E-commerce workloads. We consider a detailed characterization of client's access patterns and benchmarking E-commerce workload as an important areas of future work.

Bibliography

[AGLM95] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 23–34, San Jose, California, May 1995.

[AW96] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 126–137, Philadelphia, PA, May 1996.

[BC98] P. Bradford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 151–160, Madison, Wisconsin, 1998.

[BCF$^+$99] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of IEEE INFOCOM Conference*, 1999.

[BDR97] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proceedings of 1997 USENIX Technical Conference*, pages 289–304, 1997.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.

[CDN93] M. Carey, D. DeWitt, and J. Naughton. The 007 Benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington, D.C., May 1993.

[CFZ94] M. Carey, M. Franklin, and M. Zaharioudakis. Fine Grained Sharing in a Page Server OODBMS. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 359–370, Minneapolis, Minnesota, May 1994.

[CL91] M. Carey and M. Livny. Conflict Detection Tradeoffs for Replicated Data. *ACM Transactions on Database Systems*, 16(4):703–746, December 1991.

[CL98] P. Cao and C. Liu. Maintaining Strong Cache Consistency in the World-Wide Web. *IEEE Transactions on Computers*, 47(4):445–457, 1998.

[CZB98] P. Cao, J. Zhang, and K. Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, pages 373–388, 1998.

[DHR97] F. Douglis, A. Haro, and M. Rabinovich. HPP:HTML Macro-Preprocessing to Support Dynamic Document Caching. In *Proceedings of 1997 USENIX Symposium on Internet Technologies and Systems*, pages 83–94, December 1997.

[DR94] A. Delis and N. Roussopoulos. Management of Updates in the Enhanced Client-Server DBMS. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, pages 326–334, 1994.

[DR98] A. Delis and N. Roussopoulos. Techniques for Update Handling in the Enhanced Client-Server DBMS. *IEEE Transaction on Knowledge and Data Engineering*, 10(3):458–476, May/June 1998.

[FCL97] M. J. Franklin, M. J. Carey, and M. Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. *ACM Transactions on Database systems*, 22(3):315–363, September 1997.

[FFF99] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-Law Relationships of the Internet Topology. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architecture and Protocols for Computer Communications*, pages 251–262, 1999.

[FGHW99] A. Feldmann, A. C. Gilbert, P. Huang, and W. Willinger. Dynamics of IP traffic: A Study of the Variability and the Impact of Control.

In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architecture and Protocols for Computer Communications*, pages 301–313, September 1999.

[Fox98] A. Fox. *A Framework for Separating Server Scalability and Availability from Internet Application Functionality*. PhD thesis, University of California at Berkeley, 1998.

[Gbu96] P. Gburzynski. *Protocol Design for Local and Metropolitan Area Networks*. Prentice Hall Inc., 1996.

[Gru97] R. E. Gruber. *Optimism Vs Locking: A Study of Concurrency Control for Client Server Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, 545 Technology Square; Cambridge, Massachusetts 02139, 1997.

[GS96] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proceedings of 1996 USENIX Annual Technical Transactions*, pages 141–152, San Diego, California, 1996.

[GVC98] Graphics, Visualization, and Usability Center. GVU's 10th WWW User Survey. http://www.cc.gatech/gvu/user_surveys/, October 1998.

[MA98] D. A. Menasce and V. A. F. Almeida. *Capacity Planning for Web Performance Metrics, Models and Methods*. Prentice Hall Inc., 1998.

[Men99] D. A. Menasce. Application Performance Management for E-Business. Technical report, CPT Software, http://www.cptsoftware.com/wp.htm, 1999.

[Muk94] A. Mukerjee. On the Dynamics and Significance of Low Frequency Components of Internet Load. *Internetworking : Research and Experience*, 5:163–205, December 1994.

[ÖVU98] M. T. Özsu, K. Voruganti, and R. C. Unrau. An Asynchronous Avoidance-based Cache Consistency Algorithm for client caching DBMSs. In *Proceedings of the 24th VLDB Conference*, pages 440–451, 1998.

[Pax97] V. Paxon. End-to-End Internet Packet Dynamics. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architecture and Protocols for Computer Communications*, pages 139–152, 1997.

[PC93] C. Pu and S. Chen. ACID Properties Need Fast Relief: Relaxing Consistency Using Epsilon Serializability. In *Proceedings of Fifth International Workshop on High Performance Transaction Systems*, Asilomar, California, September 1993.

[per99] Dynamic Caching for E-commerce. Technical report, Persistence Software Inc., http://www.persistence.com/Developers/papers/caching.html, 1999.

[Pu91] C. Pu. Generalized Transaction Processing with Epsilon-Serializability. In *Proceedings of Fourth International Workshop on High Performance Transaction Systems*, September 1991.

[Pu93] C. Pu. Relaxing the Limitations of Serializable Transactions in Distributed Systems. *Operating Systems Review*, 27(2):66–71, April 1993.

[RCG98] M. Rabinovich, J. Chase, and S. Gadde. Not All Hits are Created Equal: Cooperative Proxy Caching Over a Wide-Area Network. *Computer Networks*, 30:2253–2259, 1998.

[RP95] K. Ramamritham and C. Pu. A Formal Characterization of Epsilon Serializability. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):997–1007, December 1995.

[SAYZ99] B. Smith, A. Acharya, T. Yang, and H. Zhu. Caching Equivalent and Partial Results for Dynamic Web Content. In *Proceedings of 1999 USENIX Symposium on Internet Technologies and Systems*, 1999.

[SCH$^+$99] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The End-to-End Effects of Internet Path Selection. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architecture and Protocols for Computer Communications*, pages 289–299, 1999.

[spe99] SPECweb99 Benchmark. Technical report, Standard Performance Evaluation Corporation, http://www.spec.org/osg/web99/, 1999.

[WR91] Y. Wang and L. Rowe. Cache Consistency and Concurrency Control in Client/Server DBMS Architecture. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 367–376, Denver, Colorado, May 1991.

[WYP92] K. Wu, P. S. Yu, and C. Pu. Divergence Control Algorithm for Epsilon-Serializability. In *Proceedings of Eighth International Conference on Data Engineering*, pages 506–515, February 1992.

[ZSJ00] H. Zou, N. Soparkar, and F. Jahanian. Probabilistic Data Consistency for Wide-Area Applications. In *Proceedings of 16th International Conference on Data Engineering*, page 85, San Diego, California, 2000.

Appendix A

Glossary

AACC – Asynchronous Avoidance-Based Cache Consistency algorithm

ACBL – Adaptive CallBack Locking scheme

ACID – Atomicity, Consistency, Isolation and Durability

AOCC – Adaptive Optimistic Concurrency Control algorithm

AWCC – Asynchronous Web Cache Consistency algorithm

C2PL – Caching Two-Phase Locking scheme

CBL – CallBack Locking scheme

DBMS – DataBase Management System

DC – Divergence Control method

ESR – Epsilon Serializability

FIFO – First In First Out

GVU – Graphics, Visualization and Usability center

HPP – HTML Macro-PreProcessing

HTTP – HyperText Transfer Protocol

IMS – If-Modified-Since

IP – Internet Protocol

ISP – Internet Service Provider

LRU – Least Recently Used

O2PL – Optimistic Two-Phase Locking

ODBMS – Object DataBase Management System

OWCC – Optimistic Web Cache Consistency algorithm

MOB – Modified Object Buffer

ROT – Object Table

RPT – Page Table

ROWA – Read One Write All

SIDE – Sensors In Distributed Environment

SR – Serializability

SWCC – Synchronous Web Cache Consistency algorithm

TCP – Transmission Control Protocol

TTL – Time-To-Live

ULB – Undo Log Buffer