Design Issues for Large-Scale Distributed Systems:
Data and Resource Managements

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA, TWIN CITIES

By

Keqiang Wu

IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

David J. Lilja, Adviser
May, 2005
© Keqiang Wu (吴克强) 2005
This is to certify that I have examined this copy of a doctoral dissertation by

Keqiang Wu

and have found that it is complete and satisfactory in all respects, and that any and all revisions required by the final examining committee have been made.

David J. Lilja

Name of Faculty Adviser(s)

Signature of Faculty Adviser(s)

May 19, 2005

Date

GRADUATE SCHOOL
ACKNOWLEDGMENT

The results described in this dissertation would not have been possible without the help and support of a number of people. First and foremost, I would like to thank my advisor, Professor David J. Lilja, for his continuing guidance, encouragement, and support during the past four and half years. I would also like to thank Professor Pen-Chung Yew, Professor Wei-Chung Hsu, Professor Ahmed Tewfik, and Professor Gerald Sobelman, for serving on my Ph.D. examination committee and for their constructive suggestions to make this dissertation better. I acknowledge with thanks to Peng-fei Chuang for his contributions on the first part of this dissertation. Special thanks go to Chris Hescott and Joshua J. Yi for their valuable comments on the first publication directly related to this dissertation.

I would like to thank the members of ARCTiC Labs, Haowei Bai, Ying Chen, Peng-fei Chuang, Chris Hescott, Baris Kazar, Sreekumar Kodakara, Keith Osowski, Professor Resit Sendag, and Dr. Joshua Yi, for providing me with such a friendly working environment. They have been an invaluable resource for encouragement and inspiration.

My wife, Zhanxue Song (宋占雪), deserves greater thanks than I can possibly give. For more than six years from Champaign-Urbana, Chicago-Naperville, to Minneapolis-Saint Paul, she has stayed with me, patiently allowing me to pursue my dreams. Without her being there for me, I probably would have quit. I will always be grateful to her. I would also like to thank my parents Ping Wu (吴屏) and Ngan Ling Ho (何银铃), as well as my brother Keqin Wu (吴克勤), sisters Yeuk Tan Ng (吴若丹) and Yeuk Na Ho (何若娜), and their families for their constant love and support. This dissertation is dedicated to all of them.
ABSTRACT

Current cache consistency approaches in data-shipping DBMS architectures rely completely on a centralized server or servers to provide concurrency control, which imposes a limitation on the scalability and performance of these systems. In addition, traditional asynchronous and deferred protocols are “blindly” optimistic on the cached data and do not exploit data sharing information. Moreover, due to the increasing complexity of generic large-scale distributed systems, the Quality of Service (QoS) design becomes challenging. This dissertation proposes two protocols and a framework to address these issues of data and resource managements, respectively.

This dissertation designs a protocol, Active Data-aware Cache Consistency (ADCC), for data-shipping DBMS. Compared with Callback Locking (CBL), ADCC uses peer-to-peer (P2P) communication to reduce the latency for discovering data conflicts by 50%, while increasing message overhead by about 8% only. In addition, ADCC improves scalability by partially offloading the concurrency control function from the server to the clients.

Second, this dissertation designs a protocol, Self-tuning Active Data-aware Cache Consistency (SADCC), for data-shipping DBMS. By statistically quantifying the speculation cost, clients can self-tune between optimistic and pessimistic consistency control. Compared with Asynchronous Avoidance-based Cache Consistency (AACC) and ADCC, SADCC significantly reduces the speculation cost under high contention environment. In a non-contention environment, both SADCC and ADCC display a slight reduction (an average of
2.3%) in performance compared to AACC with a high-speed network environment. With high contention, however, SADCC has an average of 14% higher throughput than AACC and 6% higher throughput than ADCC.

Finally, this dissertation demonstrates the limitations of adaptive control theory on QoS design for generic distributed systems, and proposes an *adaptive dual control framework* for mitigating those limitations. By incorporating the existing uncertainty of the on-line prediction into the control strategy and accelerating the parameter estimation process, the dual adaptive control framework optimizes the tradeoff between the control goal and the uncertainty, and displays robust and cautious behavior. In particular, under the medium uncertainty, the average hit-rate ratio provided by the adaptive dual control system and the conventional adaptive control system deviate from the desired hit-rate ratio by about 13% and 40%, respectively.
**TABLE OF CONTENTS**

CHAPTER 1: INTRODUCTION .................................................................................................................... 1

1.1 OVERVIEW ........................................................................................................................................ 1

   1.1.1 Client/server architecture ............................................................................................................ 1
   1.1.2 Client/server DBMS ....................................................................................................................... 3
   1.1.3 Client Cache Consistency ............................................................................................................. 5
   1.1.4 Quality of Service Guarantee ..................................................................................................... 7

1.2 CONTRIBUTIONS OF THIS DISSERTATION ................................................................................. 8

1.3 DISSERTATION ORGANIZATION ................................................................................................... 9

CHAPTER 2: ACTIVE DATA-AWARE CACHE CONSISTENCY .......................................................... 11

2.1 BACKGROUND AND MOTIVATIONS .............................................................................................. 11

2.2 AN ACTIVE DATA-AWARE CACHE CONSISTENCY ALGORITHM .................................................. 13

   2.2.1 Client States and Protocol Descriptions ....................................................................................... 14
   2.2.1.1 Handling A’s read requests ....................................................................................................... 15
   2.2.1.2 Handling A’s write requests ...................................................................................................... 15
   2.2.1.3 Handling data requests from other clients ............................................................................... 16
   2.2.1.4 Handling invalidation requests ............................................................................................... 18

   2.2.2 Server States and Protocol Descriptions ...................................................................................... 18
   2.2.2.1 Handling data requests for read operations ............................................................................. 19
   2.2.2.2 Handling data requests for write operations ............................................................................ 20

   2.2.3 Scenario Descriptions .................................................................................................................. 21

   2.2.4 Deadlock Solution ......................................................................................................................... 27

   2.2.5 Committing Process ...................................................................................................................... 28

   2.2.6 Lazy Updates of the Client Directory .......................................................................................... 29

   2.2.7 Quantitative Comparison with CBL ............................................................................................. 30
2.3  A PAGE SERVER DBMS MODEL FOR PERFORMANCE EVALUATION .......................................................... 32

2.3.1  The System Model .............................................................................................................................. 32

2.3.2  The Workload Model .......................................................................................................................... 35

2.4  SIMULATION STUDY .............................................................................................................................. 36

2.4.1  The UNIFORM Workload .................................................................................................................... 37

2.4.2  The HOTCOLD Workload .................................................................................................................. 42

2.4.3  The PRIVATE Workload ..................................................................................................................... 45

2.4.4  The HICON Workload ......................................................................................................................... 47

2.4.5  Discussion ........................................................................................................................................... 49

2.5  SUMMARY ............................................................................................................................................. 50

CHAPTER 3: SELF-TUNING ACTIVE DATA-AWARE CACHE CONSISTENCY ................................................... 52

3.1  BACKGROUND AND MOTIVATIONS ...................................................................................................... 52

3.2  REVISITING ASYNCHRONOUS AVOIDANCE-BASED CACHE CONSISTENCE ................................. 53

3.3  AN SELF-TUNING ACTIVE DATA-AWARE CACHE CONSISTENCY ALGORITHM ............................ 55

3.3.1  Self-tuning speculation ....................................................................................................................... 56

3.3.2  Scenario Descriptions ...................................................................................................................... 62

3.4  EXPERIMENTAL SETUP ........................................................................................................................ 67

3.5  RESULTS AND DISCUSSION .................................................................................................................. 70

3.5.1  The UNIFORM Workload .................................................................................................................... 70

3.5.1.1  Impact of contention/data-sharing ................................................................................................. 71

3.5.1.2  Impact of transaction granularity .................................................................................................... 75

3.5.2  The HOTCOLD Workload .................................................................................................................. 77

3.5.2.1  Impact of a fast CPU .......................................................................................................................... 77

3.5.2.2  Impact of a fast network .................................................................................................................... 77

3.5.3  The HICON Workload ......................................................................................................................... 80

3.5.4  The PRIVATE Workload ..................................................................................................................... 83

3.5.5  Discussion ........................................................................................................................................... 84
LIST OF FIGURES

Figure 1.1. An example of two tier client/server architecture. ................................................................. 2
Figure 1.2. Query-shipping (a) vs. data-shipping (b) architectures for DBMS. ........................................ 4
Figure 1.3. A simplified example of incorrect balance due to incorrect cache consistency enforcement. ...... 6
Figure 1.4. The traditional feedback control theoretical design of a storage system with QoS guarantees based
on: pre-run-time system identification and analysis, off-line controller design, and integration of
the controller with the storage system .................................................................................................. 8
Figure 2.1. Schematic of the client and server directories. Page $i$ is cached at both A and B, but not C. Page $j$ is
cached only at C .................................................................................................................................. 14
Figure 2.2. State transition diagram for a page in client A’s buffer. To show the use of busy states, interactions
among two clients and the server are shown ............................................................................................ 17
Figure 2.3. Schematic state transition diagram for a page in the server buffer. To show the use of the busy
states, interactions from three clients (A, B, C) and the server are shown .............................................. 20
Figure 2.4. ADCC cache consistency scenarios that involve the server (srv) and three clients (A, B and C) .... 26
Figure 2.5. Model of a page-server client-server database management system ........................................... 33
Figure 2.6. Throughput (UNIFORM) ........................................................................................................ 39
Figure 2.7. Conflicts per commit (UNIFORM) ......................................................................................... 39
Figure 2.8. Aborts per commit (UNIFORM) ............................................................................................ 40
Figure 2.9. Messages per commit (UNIFORM) ......................................................................................... 40
Figure 2.10. Throughput (HOTCOLD) .................................................................................................... 44
Figure 2.11. Conflicts per commit (HOTCOLD) ..................................................................................... 44
Figure 2.12. Server CPU utilization (HOTCOLD) .................................................................................... 45
Figure 2.13. Throughput (PRIVATE) ...................................................................................................... 46
Figure 2.14. Throughput (HICON) .......................................................................................................... 48
Figure 2.15. Abort per commit (HICON) ................................................................................................... 48
Figure 3.1. A generic example of potential write/write conflict for a page cached by \( n \) clients, where (a) represents SADCC in its pessimistic mode, and \( T \) is the average one-way trip delay for a message between the client and the server or two clients...

Figure 3.2. Message overhead due to false speculation...

Figure 3.3. Abort overhead due to false speculation...

Figure 3.4. SADCC cache consistency scenarios that involve the server (srv) and three clients (A, B and C)...66

Figure 3.5. Throughput (UNIFORM)...

Figure 3.6. Aborts per commit (UNIFORM)...

Figure 3.7. Message volume (Kbyte) sent per commit...

Figure 3.8. Abort rate (UNIFORM). In notation \( x-y-z \), \( x \), \( y \) and \( z \) denote the protocol name, client population and transaction size, respectively...

Figure 3.9. Throughput (HOTCOLD)...

Figure 3.10. Server CPU utilization (HOTCOLD)...

Figure 3.11. Server CPU utilization (HOTCOLD)...

Figure 3.12. Throughput (HICON)...

Figure 3.13. Abort rate (HICON). In notation \( x-y-z \), \( x \), \( y \) and \( z \) denote the protocol name, client population and transaction size, respectively...

Figure 3.14. Throughput (PRIVATE)...

Figure 3.15. An example comparing the piggyback message approach in AACC to an asynchronous approach without using piggyback messages, where \( CB \) and \( LE \) denote callback and lock-escalation requests, respectively...

Figure 4.1. The adaptive QoS web cache control system structure proposed in [28]...

Figure 4.2. An on-line least square estimator for identifying a differentiated proxy caching system.

Figure 4.3. The dynamics of estimated parameters with the synthetic trace.

Figure 4.4. The dynamics of estimated parameters with the stochastic trace.

Figure 4.5. The dynamics of estimated parameters with different a priori knowledge.

Figure 4.6. The adaptive dual control framework for QoS web cache systems.
Figure 4.7. Comparison of two systems under the stochastic workload with the low uncertainty. ..................115
Figure 4.8. Comparison of two systems under the stochastic workload with medium uncertainty. ..................116
Figure 4.9. Comparison of two systems under the stochastic workload with high uncertainty. ......................118
LIST OF TABLES

Table 2.1. Comparing the Cost of ADCC with CBL for the Nine Scenarios in Figure 2.4. .................................31
Table 2.2. Parameters Used in the Simulation Study. ..........................................................................................34
Table 2.3. Workload Parameter for Client $i$. ....................................................................................................36
Table 3.1. Cost for SADCC, ADCC, and AACC for $(i+1)$ concurrent updates on a page which is cached by $n$ clients. The letters in parentheses, o and p, denote the optimistic and pessimistic modes, respectively. ..............................................................................................................................................59
Table 3.2. Workload Parameters for Client $i$. .....................................................................................................69
Table 4.1. Summary of the three workloads and the four excitation signals. ..........................................................100
Table 5.1. Comparison of Cache Consistency Algorithms. .........................................................................................121
Table 6.1. Example of two workloads sharing the system according to three throughput bands. The top row shows the total system throughput in each band; the two rows below show the ratio by which the two workloads share that additional throughput. Any available throughput beyond band 2 is shared fairly (50-50) between the workloads [23]. .................................................................................................................132
Chapter 1: Introduction

In general, distributed computing can be defined as any computing that involves multiple computers remote from each other that each has a role in a computation problem or information processing. With the significant advance of the price-performance characteristic for desktop workstations and the communications bandwidth, the distributed computing has become a more desirable and practical idea. However, when the scale of distributed computing increases, some issues on system management, such as data sharing, quality of service, etc, give rise to significant challenges and opportunities for designing a distributed system. This dissertation proposes and designs two techniques for maintaining client cache consistency and a framework for quality of service. The common theme of the techniques and the framework is how to improve the performance in a large-scale distributed environment.

1.1 Overview

1.1.1 Client/server architecture

The client/server architecture describes the relationship between two computer programs in which one program, the client, makes a service request to another program, the server, which fulfills the request. Although the client/server idea can be used by programs within a single computer, it is a more important idea in a distributed computing environment. In a distributed computing system, the client/server model provides a convenient way to
interconnect programs that are distributed efficiently across different locations. In general, the client/server architecture can have more than two tiers. In this dissertation, we consider the two-tier client/server architecture – the techniques presented in this dissertation can be easily extended to multi-tier client/server architecture.

Computer transactions using the client/server model are very common. For example (Figure 1.1), to check your bank account from your computer, a client program in your computer forwards your request to a server program at the bank. That server program retrieves your account balance. The balance is returned back to the bank data client, which in turn serves it back to the client in your personal computer, which displays the information for you.

![Two Tier Client/Server](image)

**Figure 1.1. An example of two tier client/server architecture.**

The client/server architecture has become one of the central ideas of distributed computing. Most business applications being written today use the client/server model. This
model can provide sophisticated system-wide services and management, and is configurable for maximum security.

1.1.2 Client/server DBMS

In client/server DBMS architectures, there are two categories of systems, query-shipping and data-shipping. In query-shipping systems, such as a relational client/server DBMS, clients send a query to the server, which processes the query and sends the results back to the clients. In contrast, most commercial object-oriented database management systems (ODBMS) have adopted the data-shipping technique [19]. When a client wants to access data, it sends a request for the specific data items (e.g. objects or pages) to the server. The data items are shipped from the server to the clients, so that the clients can run applications and perform operations on cached data. Figure 1.2 shows the difference between these two DBMS architectures.

The data-shipping architecture is inspired by the dramatic improvements in computer price-performance and in the performance and availability of network communication. Typically, data-shipping systems can be structured either as page servers, in which clients and servers interact using physical units of data (e.g. pages or groups of pages), or object servers, which interact using logical units of data (e.g. objects). The granularity of the data transferred from the server to the clients is the fundamental difference between page and object servers.
Figure 1.2. Query-shipping (a) vs. data-shipping (b) architectures for DBMS.
Having the data available at the clients can reduce the number of client/server interactions to thereby free server resources (CPU and disks), thus decreasing client-transaction response time. Local caching allows copies of a database page to reside in multiple client caches. Moreover, when inter-transaction caching is used, a page may remain cached locally when the transaction that accessed it has completed. Concurrency control for cached pages must be enforced to ensure that all clients have consistent page copies in their caches and that they see a serializable view of the database.

1.1.3 Client Cache Consistency

In traditional client/server architecture, clients rely on servers for resources, such as files, devices, and even processing power. However, the technology advances in computer price-performance and in the performance and availability of network communication have made it desirable and practical to offload more functionality from the server to the client workstations [19]. As a result, many commercial systems, including object database management systems (ODBMS), distributed file-systems, mobile data management systems, and multi-tiered Web-server systems all use some variant of the data-shipping model, where data objects are retrieved from the server, and are cached and operated upon at the client nodes. Aggressive client-side caching becomes a widely held tenet in designing data-shipping DBMS and distributed file system [9].

Client caching is usually an effective approach for improving the system performance, since it can reduce the number of interactions between the server and clients, and free the server resources, such as disk, CPU, etc. However, since multi-copies of the same data object
co-exist in the system, to maintain the semantics correctness, it is necessary to maintain the consistent view on the same data. The consistency control is particularly important for mission-critical tasks, such as on-line transaction, high performance computing, etc. Figure 1.3 shows a simplified example of concurrent on-line transactions on a shared bank account. If the cache consistency is not enforced correctly, Figure 1.3 shows a possible total order of the related operations in the two transactions, which results in incorrect results.

Using the data-shipping DBMS architecture as a concrete application, this dissertation proposes, designs, and validates two techniques, peer-to-peer (P2P) communication and self-tuning speculation, for maintaining client cache consistency.

Figure 1.3. A simplified example of incorrect balance due to incorrect cache consistency enforcement.
1.1.4 Quality of Service Guarantee

The widespread deployment of the advanced computer technology in business and industries has demanded the high standard on quality of service (QoS). For example, many Internet applications, i.e. online trading, e-commerce, and real-time databases, etc., execute in an unpredictable general-purpose environment but require performance guarantees. Failure to meet performance specifications may result in losing business or liability violations. To design a distributed computing system (such as data center) with certain performance guarantees becomes increasingly important.

The traditional procedures to designing a storage system with QoS guarantees typically include: to quantify hardware capability, software execution requirements, resource demands, and workload characteristic, then apply an appropriate combination of pre-run-time analysis, admission control, and resource allocation algorithms to ensure that the system is not overloaded and that the desired performance is achieved (Figure 1.4).

The system performance typically depends on the interaction of multiple components. To isolate the impact on performance of individual components becomes difficult in a complex computer system. In addition, the behavior of large-scale distributed systems is typically dynamically changing. These issues make the traditional approaches based on the pre-run-time analysis impractical. The ability of on-line identification and auto-tuning of adaptive control systems has made the adaptive control theoretical design an attractive approach for QoS design. However, there is an inherent constraint in adaptive control systems, i.e. a conflict between asymptotically good control and asymptotically good parameter estimates. This constraint has not been addressed in previous research [23][29] for designing computing systems.
1.2 Contributions of this Dissertation

This dissertation makes the following contributions.

This dissertation develops an algorithm, Active Data-aware Cache Consistency (ADCC), which employs peer-to-peer communication for maintaining client cache consistency in data-shipping DBMS. Compared with Callback Locking (CBL) algorithm [20], ADCC displays two major advantages: (i) It decreases the latency for detecting data conflicts by 50%, while increasing message overhead by only about 8%. (ii) It relieves the server bottleneck by partially offloading some functionality for concurrency control (such as callback) from the server to the clients.
Based on ADCC, this dissertation designs another algorithm, Self-tuning Active Data-aware Cache Consistency (SADCC), which employs not only peer-to-peer communication but also self-tuning speculation for maintaining client cache consistency in data-shipping DBMS. SADCC self-tunes between optimistic and pessimistic consistency control to improve performance. Compared with ADCC and Asynchronous Avoidance-based Cache Consistence (AACC) [35], SADCC improves throughput by 6% compared to ADCC and by 14% compared to AACC in a high contention environment.

This dissertation also addresses the limitations of adaptive control theoretical on designing generic distributed computing systems. This dissertation proposes an adaptive dual control framework for mitigating these limitation by optimizing the tradeoff between the control goal and the system identification. In particular, when the uncertainty increases, the average hit-rate ratio provided by the adaptive dual control system and the conventional adaptive control system deviate from the desired hit-rate ratio by about 13% and 40%, respectively.

1.3 Dissertation Organization

The remaining chapters are organized as follows:

Chapter 2 presents the algorithm, Active Data-aware Cache Consistency. First, the background and motivation are described and evaluated. The proposed ADCC algorithm is described in detailed. The experimental setup and the simulation results are presented and discussed.
Chapter 3 presents the algorithm, Self-tuning Active Data-aware Cache Consistency. First, the motivation is evaluated. Then, the proposed SADCC algorithm is described in detailed. The experimental setup and the simulation results are presented and discussed.

Chapter 4 presents the adaptive dual control framework. First, it briefly reviews the conventional adaptive control framework. The sensitivity analysis of on-line system identification is discussed in detailed. Then, the dual adaptive control framework is presented and the simulation results are presented.

Chapter 5 overviews recent work on client cache consistency protocols and the application of adaptive control theory to designing computing systems.

Chapter 6 outlines the future work.

Finally, Chapter 7 concludes.
Chapter 2: Active Data-aware Cache Consistency

This chapter presents in detail the algorithm, Active Data-aware Cache Consistency (ADCC) for data-shipping DBMS architectures. ADCC addresses a disadvantage of the conventional cache consistency algorithms, server-based communication, by employing peer-to-peer (P2P) communication.

2.1 Background and Motivations

Research on cache consistency protocols for data-shipping database systems has been motivated by the inherently distributed nature of the advanced applications that DBMSs support, e.g. electronic commerce, distributed multimedia applications, etc. The primary disadvantage of current protocols [4][10][20][35][39][41] is the server-based communication path. In other words, whenever a client needs permission to update a cached data item, it must always send a request to the server before the transaction commits. Before granting the request, the server must issue callback requests to all sites (except the requester) that hold a cached copy of the requested data. Since the server is the only source for enforcing cache consistency, two potential drawbacks arise:

1) It is well known that the earlier discovery of data conflicts can improve performance and lower the abort rate/cost [20]. However, since the server is always on the critical path, this unavoidably increases the communication latency for detecting data conflicts.
2) Excessive demands for pages from the server often limit the performance and scalability of traditional caching algorithms, such as CBL [19][20][37]. As the power of client workstations has increased dramatically, it is more desirable to transfer additional processing to the client machines [34]. The centralized server design limits this transition.

To address these problems, we propose an efficient client cache consistency algorithm, Active Data-aware Cache Consistency (ADCC), which employs both client-client (P2P) and server-based communications. Via a two-tier directory, ADCC allows not only the server but also the clients to track the global state of cached data. An optimization, Lazy Update, is introduced to reduce the message overhead for maintaining the client directory consistency.

The primary contributions of this chapter are:

1) It proposes an algorithm, Active Data-aware Cache Consistency [43], that decreases the latency for detecting data conflicts by 50%, while increasing message overhead by only about 8%.

2) This algorithm significantly reduces the potential interval for write/read and write/write conflicts.

3) This algorithm relieves the server bottleneck by partially offloading some functionality for concurrency control (such as callback) from the server to the clients.

We compare ADCC with the leading cache consistency algorithm, Callback Locking (CBL), which is the most widely implemented algorithm in commercial DBMSs. This study focuses on a page-server architecture with page level consistency, since non-adaptive callback schemes are the best overall choices of existing schemes [19] and a page-server architecture usually provides superior performance over an object-server architecture [34].
While security and robustness are important issues, they are beyond the scope of this work, which focuses primarily on performance.

### 2.2 An Active Data-aware Cache Consistency Algorithm

Before describing ADCC, we give a brief review of CBL [19][20]. In CBL, clients can cache data across transaction boundaries. Locally cached page copies are always guaranteed to be valid, so clients can read them without contacting the server. However, clients need to obtain write permission from the server before they can proceed with a write operation. A server typically maintains a table to keep track of which clients have locks (read or write) for a page. If the page is cached at other clients, the server sends callback messages to other clients asking them to downgrade or relinquish their locks. Before receiving the acknowledgement from all related clients for its callback request, the server cannot grant write permission to the requesting client.

There are two major CBL variants, CBL-R and CBL-A. In CBL-R, write permissions are granted only for the duration of a single transaction. In CBL-A, in contrast, write permissions are retained at the clients until being called back, or until the corresponding page is dropped from the cache.

In contrast to CBL, ADCC allows not only the server but also the clients to maintain a directory for each cached page. The directory for a page is organized as a bit vector of presence flags ($p$ is the total number of clients that cache the same page), indicating which client has a copy of that page in its cache, together with the state information. The related
directory information is tagged with the data page and sent to the requester. Figure 2.1 shows a schematic of the directory structures.

![Schematic of the client and server directories](image)

Figure 2.1. Schematic of the client and server directories. Page $i$ is cached at both A and B, but not C. Page $j$ is cached only at C.

### 2.2.1 Client States and Protocol Descriptions

A page in a client (e.g. client A) buffer may be in one of five valid states. Three of these are stable states: (1) exclusive-clean ($excc$) - only one unmodified (read-only) cached copy in the clients; (2) shared ($shrd$) - more than one read-only cached copy whose whereabouts are indicated by the presence vector; (3) modified ($mdfd$) - only one modified page is cached. The transition states are: (4) busy ($bsy$) - the transaction is in the process of updating the data;
(5) *speculative-busy* (*sbsy*) - which is similar to the *busy* state except that it implies the update is speculative.

When client A updates a cached page, it changes the page state to one of the transition states and will not change the page state to a stable state until the transaction has committed or aborted. When a page is in one of the transition states, the client ignores any invalidation requests for this page from peers. The *speculative-busy* state converts to the *busy* state if the server grants the speculation, or the transaction aborts if the server finds that the speculation is incorrect. Figure 2.2 shows the simplified state transition diagram for a page in client A’s buffer.

### 2.2.1.1 Handling A’s read requests

If the desired page is in its local cache, client A accesses it without contacting the server. If the page is not in its cache (represented by *inv* in Figure 2.2), it sends a read request for the page to the server (shown in Figure 2.3). When the page first arrives at A, depending on the tagged directory information of the previous state of this page, A sets the state as *exclusive-clean* or *shared* (the state transits from *inv* to *excc* or *shrd* in Figure 2.2).

### 2.2.1.2 Handling A’s write requests

If a page is in A’s cache in the *exclusive-clean* or *modified* states, A changes the client page state to *busy* and updates the page without contacting the server. If the cached page is in the *shared* state, A continues the update, changes the state to *speculative-busy*, and sends a
speculation request to the server and invalidation requests to the other sharing clients. When A receives an acknowledgement message (ack) from the server, A changes the page state to busy. A is not allowed to commit until it has received an ack message from all necessary clients. If the speculation is incorrect, A aborts the transaction. If the state is busy or speculative-busy, client A ignores any invalidation requests for that page from the other clients. Any race between multiple speculative updates is resolved at the server. If the page is not in A’s cache, client A sends a write request for the page to the server (shown in Figure 2.3). When the page first arrives at A, A sets the state as busy and continues the update (the state transits from inv to bsy with Sdata/Awr in Figure 2.2).

2.2.1.3 Handling data requests from other clients

If client A is the exclusive-owner of a page, A may receive a data request which requires A to provide the data page to client B. If B’s intended operation is read and A is not updating the data, client A forwards the page to B, and changes the page state to shared. If B’s intended operation is a write, then A forwards the page to B and invalidates its copy if there is no conflict. Otherwise, A forwards the page to B and informs the server of the conflict if A has read the page, or blocks the request and informs the server of the conflict if A has updated the page. If there is a conflict, B is not allowed to commit before A. These conflict scenarios are not shown in Figure 2.2, but are discussed in Section 2.3.
Figure 2.2. State transition diagram for a page in client A’s buffer. To show the use of busy states, interactions among two clients and the server are shown. For simplicity, scenarios with conflict transactions are not shown. The abbreviations inside the circles denote a page state in A’s buffer. The notation X/Y means if message X is received, then in addition to the state change, action Y is generated. “-” means no action. If multiple X/Y pairs are associated with an arc, it means that multiple inputs can cause the same state transition. “x2y” means that client A sends x to y. “A”, “B” and “S” denote client A, client B and server, respectively. “swr”, “abt” and “cmt” denote speculative write, abort, and commit, respectively. For example, “Brd/data2B” denotes that, when A receives a read request from client B for a page in the exclusive-clean or modified states, A forwards the data message to B and changes the page state to shared; “Aswr/req2S; inv2B” denotes that, when A wants to update a locally cached page which is also cached by B, A speculatively continues the update, changes the page state from shared to speculative-busy, sends a speculation request to the server, and sends an invalidation request to B. “Sdata/Ard” denotes that, when a data page which was requested by A arrives at client A from the server, depending on the original page state in the server buffer, A changes the page state to exclusive-clean or shared and conducts the rd operation.
2.2.1.4 Handling invalidation requests

If client A is one of the sharing clients of a page, A may receive an invalidation request from another sharing client B or the server because of a speculative or non-speculative update. If A’s transaction has read the page but not committed yet, A blocks the request and informs the server of the conflict. The remote client is not allowed to commit before A.

2.2.2 Server States and Protocol Descriptions

A page in the server buffer may be in one of five states. Three of these are stable states: (1) unowned - no cached copies in the clients; (2) exclusive - one read or write cached copy is in a client, indicated by the corresponding presence flag; (3) shared - two or more read-only cached copies whose whereabouts are indicated by the presence flags. Two of the states are transition states: (4) busy-shared and (5) busy-exclusive, which correspond to read or write requests that might still be in progress, or the related transaction has not committed yet. The server directory also maintains an entry indicating which client is busy if the page is in a transition state. The transition states are used to avoid race conditions and to provide serialization. If a request for a page in the transition state arrives at the server, the server either blocks the request or informs the requesting client to abort if blocking would lead to a deadlock.

An exclusive directory state in the server means that the page may be in either the modified or exclusive-clean states in a client buffer. If a client requires a page which is cached exclusively by another client, the server forwards the request to the exclusive owner. On the other hand, the exclusive owner is allowed to modify the page without contacting the
server. A simplified state transition diagram for pages in a server’s buffer is shown in Figure 2.3.

2.2.2.1 Handling data requests for read operations

When the server receives a page read request from client C, it checks the page state and owner information and responds to these requests as follows, given these possible states.

- **unowned**: the server returns the page and sets the state to exclusive.
- **exclusive-B**: the server forwards the request to client B, changes the state to busy-shared, and sets C to the busy client (we use the shorthand, busyShared-C). This shorthand means that the server thinks client C is in the process of receiving the page which is being provided directly by the previous exclusive owner. After receiving an ack message (via a piggyback message) from C indicating that it has received the data, the server changes the state to shared-C-B.
- **shared-A-B**: the server returns the page to C and adds C into the directory. The server informs A and B by piggybacking the directory update on subsequent messages to these clients. This scenario is not shown in Figure 2.3, but is discussed in Section 2.3.
2.2.2.2 Handling data requests for write operations

When the server receives a page write request from client C, it checks the page state and owner information and responds to these requests as follows, given these possible states.

- **unowned**: the server returns the page and sets the state to busyExcl-C. The server keeps this busy state until C commits or aborts.
• exclusive-B: the server forwards the request to B, and changes the state to busyExcl-C.

• shared-A-B: the server changes the state to busyExcl-C, informs A and B to invalidate their copies, returns the page to C, and informs C to wait for ack messages for invalidation from A and B before C can commit (the state transits from shared-A-B to busyExcl-C in Figure 2.3). If a speculative write request from A or B arrives at the server while the page state is still in transition (busyExcl-C), the server sends a nack message to the requester informing it of the unsuccessful speculation.

2.2.3 Scenario Descriptions

To assist in understanding the algorithm, we present several scenarios in Figure 2.4 that compare ADCC with CBL. These scenarios represent most write-related operations among the server and two or three clients, Scenarios 1, 2, 5 and 8 for a write without conflict, and Scenarios 3, 4, 6 and 7 for a write with write/read conflicts. Scenarios 9 and 10 represent a read. Scenario 9 is used to show an optimization in ADCC to remove the message overhead for maintaining client directory consistency. While we focus on the interaction among the server and two or three clients, the discussion is valid for any number of clients.

Scenario 1: Client A wants to update page m which is only cached by itself. In ADCC, A continues the update without contacting the others.

In contrast, CBL requires A to send a message (wr) to the server to obtain a write lock if A has only a read lock on the page. Since no other clients cache page m, the server grants the
write lock immediately to A. While A is waiting for a response from the server, A has to block, i.e. it must stop.

**Scenario 2:** Client A wants to update page m which is cached by both A and B. B does not use that page. In ADCC, A speculatively conducts the update. At the same time, A sends speculation (spec) and invalidation (inv) requests to the server and B, respectively. The server grants the speculation (ack), and B responds (ack) and removes its cached copy.

In CBL, when the server receives the write request from A, the server sends a callback message (CB) to B. Since B is not using page m, B invalidates its copy and informs the server (ack). Then, the server grants a write lock (ack) to A. A must block until the server responds. The sending and receiving of the four messages are done sequentially.

**Scenario 3:** Page m is cached by both A and B. A wants to update it. B has read it, but the transaction has not committed yet. In ADCC, A speculatively conducts the update, and sends speculation and invalidation requests to the server and B, respectively. The server grants (ack) the speculation. B blocks the request and informs the server of the conflict (nack). After B commits (cmt) its transaction, the server piggybacks (pgybk ack) this information in a subsequent message to A. A is not allowed to commit before B commits.

When the callback message (CB) from the server arrives at B in CBL, B informs the server of the conflict (nack). The server does not grant a write lock to A until B commits (cmt) and relinquishes the page. A blocks while waiting for the server to respond.
**Scenario 4:** Page m is cached by both A and B. Both want to update it. In ADCC, both A and B speculatively update the page, inform the server of the speculation (spec), and send an invalidation request (inv) to the other side. However, as A and B are both in the speculative-busy state, they ignore the incoming invalidation request. The race between the speculation requests is resolved by the server. If the request from A arrives at the server before B’s request, the server grants A’s speculation (ack). When B’s request arrives, the server informs B of the incorrect speculation (nack). After B aborts (abort), the server piggybacks this information in a subsequent message (pgybk ack) to A. A is not allowed to commit before B invalidates its cached copy.

In CBL, if the write request (wr) from A arrives at the server earlier than B’s request, the server sends a callback (CB) message to B. After B relinquishes its read lock and informs the server (ack), the server grants (ack) a write lock to A. A and B have to block until the server responds. The server will not grant a write-lock to B until A commits.

**Scenario 5:** Page m is cached only by B, but B is not using it. A wants to update it. In ADCC, A sends a request (wr) to the server. The server forwards the request (fwd req) to the exclusive owner, B, which forwards the data (fwd data) directly to A and invalidates its own copy.

In CBL, the server behavior is similar to Scenario 2, except that the server forwards the page (fwd data) to A after B responds (ack). Again, A blocks before the server responds.

**Scenario 6:** Page m is cached only by B. B has read it but the transaction has not committed. A wants to update it. In ADCC, A sends a request (wr) to the server, which is forwarded to B
(fwd req). B informs the server (nack) and A (rd-conf1) of the read conflict, but still forwards the page (fwd data) to A immediately. However, A is not allowed to commit until the commit acknowledgement (ack) from B arrives.

In CBL, the server behavior is similar to Scenario 3, except that the server forwards the page to A after B commits. Again, A blocks while waiting for the server to respond.

**Scenario 7:** A wants to update page m which is exclusively owned by B. B has updated the page, but has not committed it yet. In ADCC, this scenario is similar to Scenario 6, except B does not forward the data to A until B has committed.

In CBL, the server behaves the same as in Scenario 6.

**Scenario 8:** Page m is cached by both B and C, but neither uses it. A wants to update it. In ADCC, when the server receives A’s request (wr), it provides the data (fwd data) to A immediately and sends invalidation requests (inv) to B and C. A unblocks after the page arrives. However, A is not allowed to commit until the invalidation acknowledgements (ack) from both B and C arrive.

In CBL, the server does not grant a write lock to A until both B and C invalidate their copies. A blocks until the server responds.

**Scenario 9:** Page m is cached by both B and C, but neither is using it. A wants to read it. In ADCC, when the server receives A’s request (rd), it provides the data (fwd data) to A immediately and piggybacks the acknowledgement (pgybk ack) of having A as a new sharer on subsequent messages to B and C.
In CBL, the server responds to A immediately.

Scenario 10: Page $m$ is cached only by $B$, but $B$ is not using it. $A$ wants to read it. In ADCC, $A$ sends a request ($\text{rd}$) to the server. The server forwards the request ($\text{fwd req}$) to the exclusive owner, $B$, which forwards the data ($\text{fwd data}$) directly to $A$ and invalidates its own copy.

In CBL, depending on if $B$ has a read or write lock on $m$, the server may respond differently.

Case 1: If $B$ has only a read lock, the server provides the data ($\text{fwd data}$) directly to $A$.

Case 2: If $B$ has a write lock, the server sends a callback message ($\text{CB}$) to $B$. As $B$ is not using $m$, $B$ downgrades its write lock to read lock, and informs the server ($\text{ack}$). Then, server grants a read lock ($\text{ack}$) to $A$. $A$ must block until the server responds.
Figure 2.4. ADCC cache consistency scenarios that involve the server (srv) and three clients (A, B and C). Each arc denotes a message. A dashed arc indicates that the related request will be ignored by the recipient and a dotted arc denotes that the related message is piggybacked with a subsequent message. Each scenario is compared with CBL. Two possibilities exist for CBL in scenario 10, where case 1 represents B has only a read lock for the page, and case 2 B has a
write lock for the page, respectively. Note that more messages are handled in parallel in ADCC than in CBL.

### 2.2.4 Deadlock Solution

Similar to CBL, we use a centralized approach for handling deadlocks. The server maintains a Conflict Log Table (CLT) to keep track of conflicts. When a client commits or aborts, the server removes all entries in the CLT related to that client. When the server receives a Nack response from a client, or when the server receives a read or write request for a page whose server page state is busy-shared or busy-exclusive, the server performs deadlock detection.

When the server receives a Nack response (for example, Scenarios 3, 6 and 7 in Figure 2.4) from client B indicating that there is a conflict on a request of page \( m \) from client A, it checks the CLT. If there are no deadlocks, the server adds an entry into the CLT, indicating that A is blocked by B for the request of page \( m \). Consequently, A cannot commit before B.

The busy-shared state occurs under the following scenario. When the server receives a read request from client A for page \( m \) which is only cached by client B, the server forwards the request to B, changes the directory state to busy-shared and sets the busy-client to A. We use the shorthand, busy-shared-A, to indicate that the server thinks A is in the process of receiving the page. The page state remains in busy-shared-A until A acknowledges (via a piggyback message) the server that it has received the requested page from the original exclusive owner. If the server receives a read or write request for page \( m \) from another client, say C, before the acknowledgement from A arrives, the server first checks in the CLT if A (the busy client) is blocked by B (the original exclusive owner). If A is not currently blocked
by B, the server simply defers the request from C until the server page state turns to stable. Otherwise, the server conducts deadlock detection for C being blocked by B. If there are no deadlocks, the server updates the CLT with a new entry, i.e. C’s request on page \( m \) is blocked by B. The server will not handle this request from C until B has committed.

The *busy-exclusive* state occurs under all scenarios in Figure 2.4 except Scenarios 1 and 9. When the server receives a write request from A, the server changes the page state to *busy-exclusive* and sets A as the busy client. The server keeps this busy state until A commits or aborts. If the server receives a read or write request from client D for page \( m \), the server checks the CLT to see whether D being blocked by A leads to a deadlock. If there are no deadlocks, the server updates the CLT with a new entry, indicating that D is blocked by A on the request of page \( m \). The server will not handle this request from D until A has committed.

If the server detects a deadlock (a cycle in the wait-for graph), it informs the requester to abort. To avoid starvation, we ensure that clients can be picked for abortion only a small finite number of times [37].

### 2.2.5 Committing Process

At commit time, the client sends the logs to the server, changes the state of updated pages from *busy* to *modified*, and then activates the other client requests that are waiting for this client to commit.

When the server receives a commit message, it removes those entries in the CLT related to the committing transaction, changes the *busy-exclusive* state of those updated pages (with
busy-client as the committing client) to exclusive, moves the logs on to a persistent storage area, and activates the other client transactions that are waiting for this client to commit.

2.2.6 Lazy Updates of the Client Directory

A potential drawback of ADCC is the additional messages used to maintain client directory consistency. The server needs to inform the necessary clients to update their directories when a client has removed its cached copy or has just cached a copy. An optimization used in ADCC is to have the server inform the related clients via a piggybacked message. The inherent message delay due to the piggybacking may cause situations where some clients have an inconsistent directory relative to the server for the same page. As a result, those clients may have outdated presence flags in their directories. However, the client page states are still shared, i.e. clients B and C in Scenario 9 may not know that A has cached page \( m \). Such directory inconsistency causes a problem only when those clients want to update the page. However, as described in Section 2.3 (Scenarios 3 and 4), such updates are speculative. When the server receives a speculative update request for a shared page, the server compares its directory with that of the client. If the server detects that the client directory is outdated, it grants the speculation, but at the same time it informs the client of the discrepancy. If there are some new sharing clients, the speculative client is not allowed to commit before the new sharing clients invalidate their copies.
2.2.7 Quantitative Comparison with CBL

Table 2.1 compares ADCC with CBL for the nine scenarios in Figure 2.4. In this table, \textit{Par} denotes “partially”, \# \textit{ttl msg} means the total number of messages generated, \textit{blocked} denotes whether the client has to wait when it wants to perform an operation, and \# \textit{srv msg} denotes the total number of messages sent and received by the server. The number in parentheses denotes the number of sequential steps in the longest communication path. In Scenario 3, for example, the longest path is between A and B in ADCC. A sends an \texttt{inv} to B (1st step). After receiving the \texttt{inv}, B sends a \texttt{nack} to the server (2nd step). After B commits, it sends a message to the server (3rd step). The server piggybacks this information on a subsequent message to A (4th step). Piggybacking does not generate an additional message. In CBL, in contrast, all of the five messages have to be sent and received in that particular order. The sending/receiving of the messages constitutes the five sequential steps shown in the table for CBL. Note that the number of sequential communication steps for ADCC is always less than or equal to the number of communication steps needed for CBL.

Compared with CBL, ADCC provides several advantages: (1) ADCC reduces the number of sequential communication steps in seven scenarios; (2) it removes blocking for four scenarios, and reduces the blocking time for two scenarios; and (3) it typically reduces the total number of messages that are sent and received by the server (except in Scenarios 4 and 9), as the server is partially removed from the critical path.

In Scenario 6, ADCC requires an additional control message compared to CBL but it reduces the blocking time.
Table 2.1. Comparing the Cost of ADCC with CBL for the Nine Scenarios in Figure 2.4.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADCC # ttl msg</td>
<td>0(0)</td>
<td>4(2)</td>
<td>5(4)</td>
<td>7(4)</td>
<td>3(3)</td>
<td>6(4)</td>
<td>5(4)</td>
<td>6(3)</td>
<td>2(2)</td>
<td>3(3)</td>
</tr>
<tr>
<td>blocked</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Par</td>
<td>Yes</td>
<td>Par</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td># srv msg</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CBL # ttl msg</td>
<td>2(2)</td>
<td>4(4)</td>
<td>5(5)</td>
<td>4(4)</td>
<td>4(4)</td>
<td>5(5)</td>
<td>5(5)</td>
<td>6(4)</td>
<td>2(2)</td>
<td>2(2) or 4(4)</td>
</tr>
<tr>
<td>blocked</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td># srv msg</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>2 or 4</td>
<td></td>
</tr>
</tbody>
</table>

If $T$ is the average one-way trip delay for a message between the client and the server or two clients, then in Scenario 4, it takes time $T$ in ADCC, but $2T$ in CBL, for B to be aware of A’s write intention. Although ADCC requires two additional messages, it reduces the latency for invalidation requests by 50%. As a result, ADCC reduces the potential conflict interval by approximately 50%. In addition, the two additional messages are small control messages. More importantly, most messages in ADCC are handled in parallel so that the critical path in ADCC is no longer than in CBL. ADCC requires more messages than CBL as the number of clients involved increases. However, the analysis in Section 4 shows that the expected message cost in ADCC is actually similar or lower than CBL for generic write/write conflicts.

When client A wants to read page $m$ which is exclusively owned by client B but not in use by B (Scenario 10), in ADCC the server forwards A’s request to B, then B forwards the page to A and changes the page state to *shared*. ADCC requires three messages (2 control + 1 data). In contrast, CBL performs differently. If B has a read only for page $m$, the server forward the page to A immediately after receiving A’s request – CBL generates two messages (1 control + 1 data). If B has a write lock for page $m$, then the server sends a callback (CB) message to B. After B downgrades its write lock to a read lock and informs the
server (ack), the server grants (ack) a read lock to A – CBL requires four messages (3 control + 1 data).

For the first case, CBL might outperform ADCC for that read operation. However, when the page is cached by more than one client, this advantage disappears. Similar to the design of CBL-R and CBL-A, whether to allow the exclusive owner to keep the write permission is a design choice. Since we focus on concurrency control under data sharing, the design in ADCC brings significant benefits for write requests, such as in Scenario 1. This approach also partially offloads the server from serving data, which is similar to the data forwarding used in global memory management [20].

2.3 A Page Server DBMS Model for Performance Evaluation

2.3.1 The System Model

We use a simulation model similar to those used in previous client cache consistency performance studies [4][10][19][20][35][39][41], as depicted in Figure 2.5. The model consists of a single server and a varying number of client workstations, which are connected via a network. The number of clients is a parameter of the model. Each client node consists of: (1) a transaction generator, which submits transactions to the client one after another, (2) a buffer manager, which manages the buffer pool using an LRU page replacement policy, (3) a transaction manager, which coordinates the execution of client transactions, (4) a concurrency control manager, which implements consistency management functions and is
algorithm dependent, and (5) a resource manager, which models CPU activity and provides access to the network. Transactions themselves are each modeled as a string of page references (i.e. page reads and writes) with a unique transaction ID. If a transaction aborts, it restarts the same transaction with a new ID. After a transaction commits, a new transaction is submitted after a specified thinking period.

![Figure 2.5. Model of a page-server client-server database management system.](image)

The server model is similar to that of the clients except that the work for the server always arrives via the network. The resource manager models disk activity as well as CPU activity and network access. The server’s transaction manager coordinates the server’s operation based on the stream of incoming client requests.

The network manager models communication among the clients and server as a FIFO server with a specified bandwidth. The communication latency consists of a fixed CPU overhead for protocol processing at both the sending and receiving sites per message and a variable transmission delay on the network. To avoid network saturation, we simulate a
network with the actual load at 80% of the nominal bandwidth. Similar to [5], the unpredictable network delay is modeled by making the message provider wait for a specified time before sending the message. The delay probability and time are specified similar to [35].

### Table 2.2. Parameters Used in the Simulation Study.

<table>
<thead>
<tr>
<th>System Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pageSize</td>
<td>Size of a page</td>
<td>4 Kbyte</td>
</tr>
<tr>
<td>databaseSize</td>
<td>Size of database in pages</td>
<td>2000</td>
</tr>
<tr>
<td>numClients</td>
<td>Client workstations</td>
<td>1 to 40</td>
</tr>
<tr>
<td>client CPU speed</td>
<td>Instruction rate of client CPU</td>
<td>50 MIPS</td>
</tr>
<tr>
<td>server CPU speed</td>
<td>Instruction rate of server CPU</td>
<td>200/400 MIPS</td>
</tr>
<tr>
<td>clientBufSize</td>
<td>Per-client buffer size</td>
<td>5% of DB size</td>
</tr>
<tr>
<td>serverBufSize</td>
<td>Server buffer size</td>
<td>50% of DB size</td>
</tr>
<tr>
<td>serverDisks</td>
<td>Number of disks at server</td>
<td>4 disks</td>
</tr>
<tr>
<td>minDiskAccessTime</td>
<td>Minimum disk access time</td>
<td>4 millisecond</td>
</tr>
<tr>
<td>maxDiskAccessTime</td>
<td>Maximum disk access time</td>
<td>12 millisecond</td>
</tr>
<tr>
<td></td>
<td>Overhead Parameters</td>
<td></td>
</tr>
<tr>
<td>fixedMsgInstr</td>
<td>Fixed num of instr per msg</td>
<td>20000</td>
</tr>
<tr>
<td>perByteMsgInstr</td>
<td>Num of addl instr per msg byte</td>
<td>4</td>
</tr>
<tr>
<td>networkBandwidth</td>
<td>Network bandwidth</td>
<td>80/800 Mbps</td>
</tr>
<tr>
<td>networkDelayProb</td>
<td>Probability of delaying msg</td>
<td>10%</td>
</tr>
<tr>
<td>networkDelayTime</td>
<td>Average time a msg is delayed</td>
<td>1 msec</td>
</tr>
<tr>
<td>lockInst (CBL)</td>
<td>Instr per lock/unlock pair</td>
<td>300 instr</td>
</tr>
<tr>
<td>controlMsgSize</td>
<td>Size in bytes of a control msg</td>
<td>256</td>
</tr>
<tr>
<td>dirLookupInst</td>
<td>Instr per directory lookup/setup</td>
<td>600 instr</td>
</tr>
<tr>
<td>readAccessTime</td>
<td>CPU instr cost for RD operation</td>
<td>50 instr/byte</td>
</tr>
<tr>
<td>writeAccessTime</td>
<td>CPU instr cost for WR operation</td>
<td>100 instr/byte</td>
</tr>
<tr>
<td>diskOverheadInstr</td>
<td>CPU overhead to perform I/O</td>
<td>5000 instr</td>
</tr>
<tr>
<td>thinkTime</td>
<td>Delay for submitting a new trans</td>
<td>0</td>
</tr>
</tbody>
</table>

The simulated CPUs employ a two-level priority scheme for input queues. System requests, such as disk I/O and packaging of network messages, are given priority over client transaction requests. The high priority queue is managed as a FIFO queue and the low priority queue is managed using processor sharing among the user requests. Each disk has a FIFO queue of I/O requests, and the disk for each request is chosen uniformly from all of the
server’s disks. Disk access times are drawn from a uniform distribution between a specified minimum and maximum.

Table 2.2 describes the parameters that are used to specify the costs of the different operations and the system configuration. These parameters are similar to the ones used in previous performance studies [20].

2.3.2 The Workload Model

The multi-user OO7 benchmark has been developed for object DBMS performance studies [11]. However, this benchmark is under-specified for cache consistency studies because it does not include the necessary data sharing patterns or transaction sizes for determining the data contention level of the system [12][20]. Similar to [19][20], we examine UNIFORM, HOTCOLD, PRIVATE, and HICON data sharing patterns. These cover a wide spectrum of data contention levels and are useful in assessing the robustness of the cache consistency algorithm [19][20]. Table 2.3 summarizes the workloads that are examined in this paper.

In these workloads, transactions are represented as a string of page reference requests in which some are for reads and the others are for writes. There is a CPU instruction cost when a client performs a read or write operation. The database consists of a set of hot regions (one for each client), and a cold region. The hot region for a client is also treated as a private region for the client. The probability of an access to a page in the hot region is specified; the remainder of the accesses are directed to cold region pages. For both regions, the probability that an access to a page in the region will involve a write (in addition to a read) is specified.
Table 2.3. Workload Parameter for Client \(i\).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>UNIFORM</th>
<th>HOTCOLD</th>
<th>PRIVATE</th>
<th>HICON</th>
</tr>
</thead>
<tbody>
<tr>
<td>transSize</td>
<td>20 pages</td>
<td>20 pages</td>
<td>16 pages</td>
<td>20 pages</td>
</tr>
<tr>
<td>hotBounds</td>
<td>-</td>
<td>p to p+49, p=50(i-1)+1</td>
<td>p to p+24, p=25(i-1)+1</td>
<td>1 to 400</td>
</tr>
<tr>
<td>coldBounds</td>
<td>All of DB</td>
<td>Rest of DB</td>
<td>1001 ~ 2000</td>
<td>Rest of DB</td>
</tr>
<tr>
<td>hotAccProb</td>
<td>-</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>coldAccProb</td>
<td>1.0</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>hotWrProb</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.25</td>
</tr>
<tr>
<td>coldWrProb</td>
<td>0.1/0.2/0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>perPageInstr</td>
<td>30,000</td>
<td>30,000</td>
<td>30,000</td>
<td>30,000</td>
</tr>
<tr>
<td>thinkTime</td>
<td>0</td>
<td>0</td>
<td>30,000</td>
<td>0</td>
</tr>
</tbody>
</table>

The UNIFORM workload has no per-client locality and a high level of data contention. Each client accesses the data uniformly throughout the whole database. The HOTCOLD workload has a high degree of locality per client and a moderate amount of sharing and data contention among clients. Each client accesses the data from its private region (80% of the time) and the cold region (20% of the time). The clients can update pages in both regions. PRIVATE is a CAD-like workload with the highest per-client data locality and no contention. The only inter-client sharing in the workload involves read-only data. HICON is a skewed workload, which is unlikely in client-server DBMS environment. Nevertheless, it is typically used to expose the performance tradeoffs in a very high contention environment.

2.4 Simulation Study

We use the cost and workload settings described in Table 2.2 and Table 2.3 to obtain the following results. This paper uses CBL-A for the PRIVATE workload comparison since CBL-A performs slightly better than CBL-R for this workload [20]. For the other three
workloads, we use CBL-R as the basis. The large client cache size assumption is not realistic in situations where the transaction size is very large or the client workstation buffer is shared by multiple transactions [35]. Consequently, we use a small client cache (5% of the active database size). We examine the impact of the relative gap among client/server CPU performance, server disk I/O performance, and network bandwidth under different workloads on the overall performance and scalability. Similar to previous studies [4][10][19][20][35][39][41], the system throughput (transactions per second) and abort rate (aborts per commit) are the major performance metrics in this paper. To ensure the statistical validity of the results, the 90 percent confidence intervals for system throughput in commits/second were calculated using batched means. The confidence intervals were within a few percent of the mean. Each experiment was run ten times using ten different random number seeds.

2.4.1 The UNIFORM Workload

The UNIFORM workload has no per-client locality. Consequently, it does not benefit much from caching. As the fundamental difference between ADCC and CBL or the other protocols is P2P communication under the write/read and write/write data sharing, we first examine the impact of P2P communication under different levels of contention with the server CPU speed and network bandwidth as 400 MIPS and 80 Mbps, respectively. This configuration prevents the server/client CPU and network bandwidth from becoming a bottleneck as the client population increases. Therefore, we can focus on the impact of different levels of contention.
Figure 2.6 shows that ADCC outperforms CBL for all configurations for this workload. When the contention increases, the throughputs of both protocols drop, but the gap between ADCC and CBL increases. As the client population increases from 8 to 40, the throughput gap ranges from 5% to 17% under low data contention (coldWrProb=0.10), changes from 11% to 21% under medium data contention (coldWrProb=0.20), and varies from 15% to 54% under high data contention (coldWrProb=0.40).

This performance trend can be explained from several aspects. First of all, when write/read and write/write sharing exists, the increasing client population intensifies the data contention, since the number of cached copies for any given page increases. This contention increases the overhead of client caching, i.e., the communication latency for callback when a client wants to update a page which is also cached at other sites. Compared with the traditional server-based communication path in CBL, ADCC reduces this overhead using the direct P2P communication. Secondly, CBL relies on the server for callback handling, while ADCC partially offloads this function from the server to the clients. When the contention increases, the server in CBL has to handle more lock callbacks, while in ADCC, many more invalidations occur directly between the clients. Therefore, the server is more heavily loaded in CBL than in ADCC. When the contention is high, increasing the client population causes the performance to degrade further.
Figure 2.6. Throughput (UNIFORM).

Figure 2.7. Conflicts per commit (UNIFORM).
Figure 2.8. Aborts per commit (UNIFORM).

Figure 2.9. Messages per commit (UNIFORM).
The P2P communication path in ADCC also reduces the potential conflict interval. Consider the example of Scenario 2 in Figure 2.4, assuming that page $m$ is not in use until A wants to update it at time $t$. On average, the invalidation request from A in ADCC reaches B at $(t+T)$, while the corresponding callback message in CBL arrives at B at $(t+2T)$. Therefore, in ADCC if B accesses the data during the interval $[t, t+T]$, A’s update will be blocked. However, this interval becomes $[t, t+2T]$ in CBL. Figure 2.7 and Figure 2.8 show the block rate and abort rate for the two protocols, which confirms that the reduction in the potential blocking window by ADCC produces less contention than CBL.

Figure 2.4 shows that a significant portion of messages are handled in parallel in ADCC but sequentially in CBL. Due to the different overheads associated with the sequential and parallel message sending, the total number of messages sent is not an accurate metric for comparing the performance of these two schemes. Figure 2.9 shows that the highest contentious level of the UNIFORM workload (coldWrProb=0.4) results in ADCC sending approximately 8% more control messages (256 byte) than CBL across the range of the client population. Compared to the typical message size of a data page ($\geq$4 Kbyte), this message overhead is low. Despite of this overhead, the shorter communication path make ADCC outperform CBL.

From the above discussion, it can be seen that ADCC improves both performance and scalability compared to CBL. The increased data contention in the UNIFORM workload makes P2P communication an important aspect of producing these improvements.
2.4.2 The HOTCOLD Workload

The HOTCOLD workload has high per-client locality and moderate write/read and write/write sharing among the clients. Due to the presence of data contention, deadlock aborts are possible. By adjusting the server CPU and network bandwidth, we examine the impact of the relative gap among server/client CPU performance, network bandwidth, and disk I/O performance.

For both ADCC and CBL, the throughput increases and then gradually drops as the number of clients varies from 1 to 40 (Figure 2.10). The overhead for concurrency control gradually catches up with the gain due to client caching as the number of clients increases. Clearly, there exists an optimal number of clients to achieve the highest overall system throughput. This trade-off is more obvious in HOTCOLD than in UNIFORM because the per-client data locality in the HOTCOLD workload is reduced as the data contention increases.

P2P communication shortens the network latency when a client updates a locally cached page which is also cached on other sites. The asynchronous behavior reduces the message transmission overhead, which also helps ADCC outperform CBL. The shorter communication path for detecting data conflicts in ADCC also reduces the block rate (Figure 2.11). In addition, ADCC increases the number of clients needed to obtain the highest overall throughput by offloading the concurrency control functions partially from the server to the clients. Therefore, ADCC scales better than CBL.

Impact of a fast CPU: When the server CPU speed increases from 200 to 400 MIPS, both ADCC and CBL show improvements in throughput. The reduction in transaction execution time reduces the write/read conflict blocking time in both algorithms. Figure 2.12 shows that
for 4 to 40 clients, doubling the server CPU speed increases the throughput in CBL around 20% on average, while the improvement in ADCC is only about 10%. With slow CPUs, the server CPU’s utilization in CBL saturates when the number of clients exceeds 8 (Figure 2.12), which explains why the fast server CPU helps CBL more than ADCC.

*Impact of a fast network:* When the network bandwidth increases from 80 to 800 Mbps, both ADCC and CBL improve the throughput due to the reduced network latency. However, in CBL the server quickly saturates again with only 8 clients. On the other hand, the increased bandwidth improves the server CPU utilization in ADCC (Figure 2.12). As a result, it displays significant improvements in both throughput and scalability by moving the optimum number of clients to 16 (Figure 2.12).

The server is often eventually the bottleneck for performance and scalability due to the excess demands for pages [19][37]. However, because the server is always on the critical path for enforcing concurrency control in CBL, this server-based communication makes the situation even worse. From the simulation results, it can be seen that, due to the server-based communication path, ADCC will benefit more from expected technology advancements than CBL.
Figure 2.10. Throughput (HOTCOLD).

Figure 2.11. Conflicts per commit (HOTCOLD).
2.4.3 The PRIVATE Workload

The PRIVATE workload has the highest per-client locality among the four workloads. With this workload, the clients perform writes only on their private hot regions and there is no write/read or write/write data sharing. The lack of data contention leads to no transaction aborts.

Figure 2.13 shows that ADCC outperforms the pessimistic, synchronous CBL for this workload. For the fast server CPU and the high-speed network, the network and processing latencies are small. Therefore, the two algorithms perform very similarly when the client population is small.
As no write/read or write/write data sharing exists, P2P communication does not produce a direct benefit for ADCC in this situation. However, the non-blocking behavior and lower message transmission overhead allow ADCC to slightly outperform CBL-A. ADCC allows a client to continue its update on its cached pages without delay, even if this page was brought into the cache due to a read operation in a previous transaction. The client does not need to contact the server if it is the exclusive owner. On the other hand, CBL-A requires a client to send a message to the server on every initial page update and blocks until the server responds, if the client does not have a cached copy or only has a read lock for the cached copy. Compared with CBL-A, as there is no data contention in the PRIVATE workload, ADCC has slightly lower overhead for updates on locally cached data.

Figure 2.13. Throughput (PRIVATE).
2.4.4 The HiCON Workload

The HiCON workload displays a skewed data access pattern which is not usually present in data-shipping applications [20]. We include this workload primarily to examine the robustness of ADCC under extreme data contention situations.

Figure 2.14 and Figure 2.15 show that in the range of 1 to 2 clients, ADCC and CBL behave similarly. At 4 clients and beyond, the effects of increased data contention becomes apparent and ADCC outperforms CBL significantly since CBL suffers more from blocking. The shorter communication path in ADCC also reduces the blocking time. Both ADCC and CBL suffer from increased conflict rates as clients are added.

An increase in data contention leads to high block and abort rates. Consequently, both schemes exhibit thrashing behavior when the number of clients increases beyond 2 for CBL and 4 for ADCC. For example, at 40 clients, CBL produces about 0.8 aborts per commit. The blocking overhead due to write/read conflicts dominates the other overhead. ADCC has a slightly lower abort rate than CBL. This is due to the efficient P2P communication that reduces the latency for detecting data conflicts by about 50%. Earlier discovery of data conflicts can lower the abort rate and improve the performance [19][20].
Figure 2.14. Throughput (HICON).

Figure 2.15. Abort per commit (HICON).
2.4.5 Discussion

A key strength of ADCC is the use of P2P communication for detecting data conflicts. P2P communication reduces the communication path under read/write and write/write sharing workloads. Consequently, it reduces the potential blocking window due to write/read and write/write conflict. It is also important for scalability as an increasing client population leads to higher overheads with client caching. Similar to CBL, ADCC encounters deadlock related aborts. The shorter communication path leads to fewer deadlock related aborts and higher throughput in ADCC in a high contention environment.

The second strength of ADCC is that the functionality of concurrency control is partially offloaded from the server to the clients. As the power of client workstations is increasing rapidly, ADCC can better exploit client resources. In CBL, the server is the only source for enforcing cache consistency. ADCC removes the server partially from the critical path, which makes it scale much better than CBL.

Another feature of ADCC is that, similar to the detection-based No-Wait Locking protocol [39], it also has asynchronous behavior. In CBL, clients that are conducting update operations must remain blocked until the lock escalation message and the necessary callback messages have been processed at both the server and the clients. This message blocking delay increases in a heavily utilized server and network. The asynchronous behavior of ADCC, however, allows it to outperform CBL under the low contention environments.

Nevertheless, the advantages of ADCC are not free. ADCC tags some directory information into the data and speculation request messages. However, the total number of affected messages is limited. The size of related information depends on how many clients have cached the data. To ensure the performance gain due to client caching, the number of
sharing clients is usually limited. Compared to the typical message size of a data page (≥4 Kbyte), the memory and bandwidth overhead for tagging the related directory information is low. For extreme situations which require a large number of clients to cache data, ADCC can reduce the overhead by using a coarse directory representation, i.e. using a flag to represent a group of clients [26].

Additional control messages have to be generated to maintain the directory consistency. These messages are typically small (256 byte). An optimization, Lazy Update of the client directory, has been designed in ADCC in order to remove this overhead. Messages for directory updates are piggybacked with other messages to reduce communication costs.

In general, ADCC reduces the network latency significantly but consumes slightly more network bandwidth due to additional control information (256 byte) and related directory information tagged with data messages. Compared to the typical message size of a data page (≥4 Kbyte), the bandwidth overhead is generally low. In addition, network technology has evolved from 10 Mbps in the early 1990s to Gigabits today. The bandwidth utilization in today’s Gigabit networks is typically low [22].

2.5 Summary

In order to reduce the latency for detecting data conflicts and to relieve the server bottleneck, an efficient cache consistency protocol, Active Data-aware Cache Consistency, has been proposed for highly-scalable data-shipping DBMS architectures. By allowing clients to be aware of the global state of their cached data, the clients are actively involved in maintaining the cache consistency. An optimization for the client directory consistency, Lazy
Update, has been designed to reduce the message overhead for maintaining client directory consistency. Using P2P communication, ADCC reduces network latency for invalidation messages for write/read and write/write sharing by 50% compared to the server-based communication scheme, while increasing message overhead by only around 8%. Shortening the communication path not only improves throughput but also reduces the abort rate. By partially removing the server from the critical path for cache consistency, ADCC scales better than CBL. Both the simulation results and the analysis indicate that the overhead of ADCC is low. The experimental study shows that ADCC outperforms CBL under the four workloads tested. In particular, without per-client data locality, the increasing level of data contention leads to a higher performance gap between ADCC and CBL.
Chapter 3: Self-tuning Active Data-aware Cache Consistency

This chapter presents in detail the algorithm, Self-tuning Active Data-aware Cache Consistency (SADCC) for data-shipping DBMS architectures. SADCC addresses a disadvantage of the conventional cache consistency algorithms, blind optimism under high contention environment, by employing self-tuning speculation.

3.1 Background and Motivations

The primary disadvantage of traditional protocols [4][10][20][35][39][41] is the server-based communication path, i.e. whenever a client needs data or permission to perform an operation on cached data, it must always send a request to the server sometime prior to transaction commit. Before granting the request, the server must issue callback requests to all sites (except the requester) that hold a cached copy of the requested data. Putting the server on the critical path unavoidably increases the communication latency. This problem can be resolved by employing P2P communication, which has been presented in Chapter 2.

However, in current optimistic schemes (asynchronous [35][43] or deferred [4]), clients always proceed to write to locally-cached copies under the assumption that no other clients are currently using the data. If this optimism turns out to be wrong, then the transaction has to abort. This simple assumption makes current asynchronous and deferred schemes blindly optimistic in a high contention environment, which can result in significant lost work for
users in the highly interactive OODBMS environments for which page servers are often intended. For example, the high abort rate in high contention workloads make optimistic approaches, such as Adaptive Optimistic Concurrency Control (AOCC) [4], unsuitable for interactive application domains [20][35].

In this chapter, we propose an efficient client cache consistency algorithm which employs both parallel communication (client-client and server-based) and self-tuning speculation. The primary contributions of this chapter are:

1) A new algorithm, Self-tuning Active Data-aware Cache Consistency (SADCC) [44], that self-tunes between optimistic and pessimistic consistency control to improve performance.

2) A demonstration that the integration of self-tuning speculation and parallel communication in SADCC improves throughput by 6% compared to ADCC [43] and by 14% compared to Asynchronous Avoidance-based Cache Consistence (AACC) [35] in a high contention environment.

3) A demonstration that, with high contention and transaction-level temporal locality, write/write conflicts can dominate other overheads when the transaction granularity (size) is reduced.

3.2 Revisiting Asynchronous Avoidance-based Cache Consistence

In Asynchronous Avoidance-based Cache Consistence (AACC) [35], clients totally rely on the server to enforce cache consistency. It displays asynchronous write behavior, i.e.,
clients always proceed to write locally cached copies under the assumption that no other clients are currently using the data. In addition to the deadlock abort, AACC encounters the mis-speculation abort due to false speculations. AACC was originally proposed for adaptive locking, which switches locking between the page and object levels. Since this dissertation focuses on page level consistency, we study AACC with page level locking only. In this section, we briefly review AACC.

Similar to traditional protocols [4][10][20][39][41], AACC [35] allows only the server to keep track of data cached by clients. There are three modes for cached pages, private-read, shared-read, and write. Locally cached page copies are always guaranteed to be valid, so clients can read them without contacting the server.

When a client (e.g. client A) wants to read a page which is not in its cache, it sends a read request to the server. (1) If the page is not cached anywhere else, the server returns the page to client A in private-read mode. (2) If the page is exclusively cached by client B in private-read mode, the server returns the page to client A in shared-read mode. The server also informs client B to change the page lock to shared-read mode via a piggyback message. The inherent message delay may cause situations where B has the page in private-read mode but A has the same page in shared-read mode. The potential problems are resolved by commit-time validation. (3) If the page is cached by clients B and C in shared-read mode, the server returns the page to A in shared-read mode. (4) If the page is cached at client B in write mode, the server blocks the requesting client, A, until client B has relinquished the write lock.

When A wants to write a cached page, it proceeds to update speculatively. At the same time, A takes some additional actions depending on the page state. (1) If the cached page is in private-read mode, A informs the server about this update by piggybacking the
information on a subsequent message. Upon receiving this piggyback message, the server sends callback requests to related clients if this page is residing at other clients in shared-read mode. If no other clients cache the page, the server updates its lock table to indicate that A has a write lock for the page. (2) If the cached page is in share-read mode, A informs the server about this update via an explicit lock escalation message. Upon receiving this message, the server sends callback requests to related clients. The server cannot grant a write lock to A before receiving positive response from all related clients indicating that they have invalidated the page. If the server receives a negative callback response indicating that there is a conflict, it performs deadlock processing. If there are no deadlocks, the client that has performed the initial update cannot commit before the client that is reading the data.

It is noted that, due to the potential inconsistent states held by different clients (e.g. A and B) on the same page, A might update a page that B has read or written. Consequently, when a client finishes the execution phase, it must conduct commit-time validation. Before the server responds, the client has to block. If the client receives a negative response from the server, it must abort the transaction.

3.3 An Self-tuning Active Data-aware Cache Consistency Algorithm

SADCC is an extension of ADCC, which also employs both direct client-client (P2P) and server-based communication. Consequently, SADCC shares two important advantages with ADCC: (1) a shorter path for detecting read/write and write/write conflict; and (2) better scalability via partially offloading the server from the critical path. In addition, compared to
ADCC, SADCC reduces the message overhead in several scenarios by using piggyback messages. Most importantly, SADCC employs a self-tuning speculation technique to improve performance and reduce the abort rate in a high contention environment.

### 3.3.1 Self-tuning speculation

Current asynchronous schemes, including ADCC and AACC, proceed to write a locally-cached copy under the assumption that the write intention declaration will succeed. If this assumption turns out to be incorrect, then the transaction must abort. In contrast to these previous protocols, SADCC statistically quantifies the risk of this assumption to self-tune between optimistic and pessimistic consistency control.

Figure 3.1 shows a generic example of a potential write/write conflict with $n$ clients who have cached the same page $m$. Only client $A_1$ wants to write the cached page $m$ at time $t_1$. To simplify the explanation, we only show the actions of client $A_1$. In SADCC, $A_1$ can be in either optimistic or pessimistic mode, depending on the total number of clients sharing a page. If $n$ exceeds a certain threshold, $n_t$, $A_1$ switches to the pessimistic mode. While in the pessimistic mode (Figure 3.1-a), $A_1$ informs the server ($\text{spec}$) of its write request and the related mode. At the same time, $A_1$ sends an invalidation request ($\text{inv}$) to each of the other ($n-1$) clients. Before the server responds, $A_1$ has to block (i.e. stops). If it is in the optimistic mode, $A_1$ behaves similarly to the pessimistic mode, except that $A_1$ proceeds with the update before the server responds.
Client $A_1$ in ADCC (Figure 3.1-b) takes similar actions as SADCC in the optimistic mode. However, the server in ADCC may take different actions from the server in SADCC. In ADCC, when the server receives a speculation write request, it always sends an *explicit* message to the requesting client. In contrast, in SADCC the server responds to the requesting client using a *piggyback* message if the speculation succeeds, or an *explicit* message if the speculation is false or the requesting client is in pessimistic mode. Compared to SADCC, ADCC has higher message overhead.

In AACC (Figure 3.1-c), $A_1$ always proceeds to update ($\text{Spec wr}$) and informs the server of the speculative update ($\text{spec}$), and it is the server that callbacks all copies cached by the other clients. The server cannot grant a *write* lock to $A_1$ before receiving positive response from $A_2, \ldots, A_n$.

If $T$ is the average one-way trip delay for a message between the client and the server or two clients, it takes an average of time $T$ in SADCC and ADCC, but $2T$ in AACC, for the destinations to be aware of $A_1$’s write intention. If there are $i$ clients among the other $(n-1)$ clients want to update page $m$, but the other $(n-1-i)$ do not use the same page before being aware of $A_1$’s write intention, this leads to $i$ conflicts in all three schemes. The unsuccessful speculative clients in ADCC and AACC have to abort. However, there is no speculation-related abort in SADCC when in the pessimistic mode. The overheads of messages, aborts due to mis-speculation, and conflicts are shown on Table 3.1.
Figure 3.1. A generic example of potential write/write conflict for a page cached by $n$ clients, where (a) represents SADCC in its pessimistic mode, and $T$ is the average one-way trip delay for a message between the client and the server or two clients.

We use the commonly-used HOTCOLD workload [19][20] as a concrete example to illustrate the self-tuning speculation technique. Similar analysis can be applied on UNIFORM and HICON workloads. Consider the HOTCOLD workload in Table 3.1, each
client has a hotBound (e.g. 50) page range from which hotAccProb (e.g. 80%) of its accesses are drawn. The remaining coldAccProb (i.e., 1- hotAccProb) of the accesses are distributed uniformly among the remaining pages in the database. All pages are accessed with a certain write probability, hotWrProb (e.g. 20%) and coldWrProb (e.g. 20%). The hot ranges of the clients do not overlap. However, it is noted that the hot range pages of each client are also in the cold ranges of all of the other clients.

Table 3.1. Cost for SADCC, ADCC, and AACC for (i+1) concurrent updates on a page which is cached by n clients. The letters in parantheses, o and p, denote the optimistic and pessimistic modes, respectively.

<table>
<thead>
<tr>
<th>Cost</th>
<th>SADCC (o)</th>
<th>SADCC (p)</th>
<th>ADCC</th>
<th>AACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Msg #</td>
<td>i(n+1)+2n-1</td>
<td>i(n+1)+2n</td>
<td>i(n+1)+2n</td>
<td>3i+n+2</td>
</tr>
<tr>
<td>Speculation-Abort #</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Conflict #</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Assume page m shown in Figure 3.1 is in the hot range of A_n. Then the scenario for i concurrent updates consists of the following two possible cases. The probability for this scenario to occur, P_i, is estimated as follows, where p is the probability for A_2, ..., A_{n-1} to write the locally cached page m before being aware of A_1’s write intention, and ΔT is equal to T for SADCC and ADCC, and 2T for AACC.

\[
p = (\text{coldAccProb}) \times \frac{(\text{coldWrProb})}{(\text{databaseSize} - \text{hotRange})} (\text{throughput})(\text{transSize}) (\Delta T)
\]

Case 1. i clients among A_2, ..., A_{n-1} want to update page m, but A_n does not.

\[
P_i(1) = [1 - (\text{hotAccProb})] \times \frac{(\text{hotWrProb})}{(\text{hotRange})} (\text{throughput})(\text{transSize})p^i(1 - p)^{n-2-i} \frac{(n-2)!}{i!(n-2-i)!} (\Delta T)
\]
Case 2. $A_n$ and the other $(i-1)$ clients among $A_2, \ldots, A_{n-1}$ want to update page $m$.

\[
P_i(2) = \left(\frac{\text{hotAccProb}}{\text{hotRange}}\right)^V (\text{transSize})^p (1-p)^{n-2-(i-1)} \frac{(n-2)!}{(i-1)!(n-2-(i-1))!} \Delta T
\]

\[
P_i = P_i(1) + P_i(2)
\]

Therefore, the expected cost is:

\[
\exp[\text{overhead}] = \sum_{i=1}^{n-1} (\text{overhead} \times P_i) \ldots (3.1)
\]

Recent measurements in 2003 have shown that the one-way network latency $T$ is typically less than 28ms for communication within North America [31]. Assuming the same configuration and throughput for SADCC, ADCC, and AACC, and the threshold for SADCC is $n_t=6$, Figure 3.2 and Figure 3.3 show the expected overheads, which are computed using equation (3.1) and the following system parameters.

- $\text{databaseSize}=2000\text{pages}$, $T=28\text{ms}$, $\text{throughput}=25\text{TPS}$,
- $\text{transSize}=20\text{pages}$, $\text{hotRange}=50\text{pages}$, $\text{hotAccProb}=0.8$,
- $\text{hotWrProb}=0.2$, $\text{coldAccProb}=0.2$, $\text{coldWrProb}=0.2$.

For a small number of sharing clients, SADCC has lowest message overhead, although the message overhead in ADCC is close to SADCC. When the number of sharing clients is larger than 9, the message overhead for SADCC and ADCC gradually exceeds AACC. However, ADCC consistently displays a lower abort overhead than AACC. SADCC has a similar abort overhead as ADCC under the optimistic mode, i.e. the number of sharing clients is less than 6, but it has no aborts due to mis-speculation under the pessimistic mode.
The primary goal of the asynchronous protocols, such as ADCC and AACC, is to achieve a compromise between the synchronous and deferred techniques. They aim to mitigate the cost of interaction with the server while at the same time lowering the abort rate. However, if the asynchronous update leads to a wrong speculation, to block a write operation before receiving permission is a better choice than non-blocking and then aborting the whole transaction. Considering that an unsuccessful non-blocking update operation results in an abort of the whole transaction, SADCC uses a heuristic criterion to adaptively switch between optimistic (non-blocking) and pessimistic (blocking) for update on a shared cached page based on the following restriction, equation (3.2). This criterion determines the threshold of number of sharing clients, $n_t$, at which SADCC switches between optimistic and pessimistic modes.

$$\text{Expected number of aborts due to false speculation} < \frac{1}{\text{Transaction Size}} \quad \ldots \quad (3.2)$$
3.3.2 Scenario Descriptions

To assist in understanding the algorithm, we present four scenarios in Figure 3.4 to illustrate the speculation and blocking behaviors in the three schemes. These scenarios are similar to those used in previous studies [35] in client-server DBMS cache consistency algorithms. While we focus on the interaction among the server and two or three clients, the discussion is valid for any number of clients. To simplify the illustration, we only consider SADCC with optimistic mode in scenario 1.

Scenario 1: Client A wants to update page m which is cached by both A and B (the state is shared for SADCC and ADCC and shared-read for AACC). B does not use that page. In SADCC, A sends speculation (spec) and invalidation requests (inv) to the server and B,
respectively. At the same time, assuming the optimistic mode, A speculatively conducts the update. (If the mode is pessimistic, A blocks before the server responds.) The server grants the speculation (\texttt{pgybk ack}) via a piggyback message, and B invalidates its cached copy and informs A (\texttt{ack}) directly.

ADCC behave similarly except that the server grants the speculation (\texttt{ack}) via an explicit message.

In AACC, when the server receives the lock-escalation request (\texttt{LE}) from A, the server sends a callback message (\texttt{CB}) to B. Since B is not using page \textit{m}, B invalidates its copy and informs the server (\texttt{pgybk ack}) via a piggyback message. It is noted that, when all operations in A’s transaction have been finished, AACC requires commit-time validation. A has to send a commit-validation request (\texttt{CV}) to the server. Before the server response, A has to block (i.e. stop). The server does not allow A to commit until the server has received all positive responses (\texttt{ack}) to its callback requests.

Before being aware of A’s write intention, the other two scenarios are possible. (1) If B has read the data, then A is not allowed to commit before B. (2) If B has proceeded to update the data, then either A or B has to abort. The race for write/write conflict is resolved at the server. The client whose speculation request arrives at the server earlier is the winner. The server in ADCC always responds to A using an explicit message; while in SADCC, the server responds to A using a piggyback message if A’s speculation succeeds, or an explicit message if A’s speculation is false or A is in pessimistic mode. As piggybacking does not generate additional messages, compared to ADCC, SADCC typically saves one message. In average, it takes time $T$ in SADCC and ADCC, but $2T$ in AACC, for B to be aware of A’s
write intention. As a result, compared to AACC, SADCC and ADCC reduce the potential conflict interval by approximately 50% and partially remove the server from issuing callback message (CB) requests to B.

Scenario 2: Page m is cached only by B (the state is exclusive for SADCC and ADCC, and private-read for AACC), but B is not using it. A wants to update it. In both SADCC and ADCC, A sends a request (wr) to the server. The server forwards the request (fwd req) to the exclusive owner, B, which provides the data (fwd data) directly to A and invalidates its own copy.

In AACC, when the server receives the lock-escalation request from A, the server sends a callback message (CB) to B. Since B is not using page m, B invalidates its copy and informs the server (ack). Upon receiving the positive response from B, the server provides the page (fwd data) to A.

Consider the other two possible scenarios before B is aware of A’s write intention. (1) If B has read the data, then the server in AACC blocks A’s write request and will not provide the data to A until B has committed; in contrast, client B in SADCC and ADCC still forwards the data to A but will also inform A of the read conflict. Therefore, A is not allowed to commit before B. (2) If B has updated the data, then A will be blocked in all three schemes. However, in AACC, the server will not provide the data to A until B has committed. In contrast, in SADCC and ADCC, client B directly forwards the data to A after B commits. In general, compared with SADCC and ADCC, AACC makes A block at least $3T$ (including $2T$
due to B’s commit-time validation) time longer. Similar to Scenario 1, SADCC and ADCC partially remove the server from the critical path.

*Scenario 3*: Page m is cached by both B and C (the state is shared for SADCC and ADCC and shared-read for AACC), but neither uses it. A wants to update it. In SADCC, when the server receives A’s request (wr), it provides the data (fwd data) to A immediately and sends invalidation requests (inv) to B and C. A unblocks after the page arrives. However, A is not allowed to commit until both B and C invalidate their copies. After invalidating the cached copies, B and C informs the server using piggyback messages (pgybk ack).

ADCC performs similar to SADCC, except that B and C inform A of the invalidation directly using explicit messages (ack). Compared to SADCC, ADCC generates two more messages.

In AACC, the server does not grant a write lock to A until the invalidation acknowledgements (ack) from both B and C arrive. A blocks until the server responds. Compared to SADCC and ADCC, AACC makes A block at least $2T$ longer.
Figure 3.4. SADCC cache consistency scenarios that involve the server (srv) and three clients (A, B and C). Each arc denotes a message. A dashed arc denotes that the related message is
piggybacked with a subsequent message. Each scenario is compared with ADCC and AACC, where CB, LE, and CV denote requests for callback, lock-escalation, and commit-time validation, respectively. More messages are handled in parallel in SADCC and ADCC than in AACC.

Scenario 4: Page m is cached by both B and C (the state is shared for SADCC and ADCC and shared-read for AACC), but neither is using it. A wants to read it. In SADCC, when the server receives A’s request (rd), it provides the data (fwd data) to A immediately and piggybacks the acknowledgement (pgybk ack) of having A as a new sharer on subsequent messages to B and C.

ADCC requires the client to take similar actions as SADCC.

In AACC, the server returns the page in shared-read mode to A immediately.

Compared to AACC, SADCC and ADCC increase the bandwidth consumption slightly due to the two piggybacked control messages.

3.4 Experimental Setup

We implemented the algorithms, SADCC and AACC, based on the simulation model which was described in Chapter 2. The other important difference from the experimental setup as described in Chapter 2 is we introduce a new workload.

The UNIFORM, HOTCOLD, HICON and PRIVATE workloads have been widely used for client caching consistency study [4][19][20]. These cover a wide spectrum of data contention levels and spatial (per-client) data locality. However, these workloads do not
embed sufficient temporal locality at transaction level, i.e. a write operation is likely to update a page which has been read by the same transaction, which is common in reality [37]. For example, a transaction process for purchasing an airplane ticket typically consists of following steps: (1) A customer is queried for a desired flight time and cities. Information about the desired flight is located in database pages $A$, $B$ and $C$. (2) The customer is told about the options, and selects a flight whose data, including the number of reservations for that flight is in A. A reservation on that flight is made for customer. (3) The customer selects a seat for the flight; seat data for the flight is in database page $D$, and so forth. This transaction can be represented as a sequence of read and write actions as follows.

$$\text{RD}(A); \text{RD}(B); \text{RD}(C); \text{WR}(A); \text{RD}(D); \text{WR}(D); \ldots$$

In this chapter, we extend the UNIFORM, HOTCOLD, HICON and PRIVATE data sharing patterns by embedding the transaction-level temporal locality, e.g. “$\text{RD}(A); \text{RD}(B); \text{RD}(C); \text{WR}(A)$” shown in the above example. In order to embed such temporal locality, we randomly pair $x\%$ of write operations with read operations within the same transaction. Each pair performs read/write operations on the same page. The level of temporal locality can be adjusted by varying the value $x$. If $x=0$, the workloads are the same as those used in previous study [19][20][43]; if $x=100$, the workloads have the highest level of read/write temporal locality among read and write operations. In this study, we set $x=50$. Adding such temporal locality might slightly change the overall distribution of spatial locality but are equal for all protocols. Table 3.2 summarizes the workloads that are examined in this chapter. For completeness, we give a brief review of the workloads below.
Table 3.2. Workload Parameters for Client $i$.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>UNIFORM</th>
<th>HOTCOLD</th>
<th>HICON</th>
<th>PRIVATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>transSize</td>
<td>20/10/5 pages</td>
<td>20 pages</td>
<td>20 pages</td>
<td>16 pages</td>
</tr>
<tr>
<td>hotBounds</td>
<td>-</td>
<td>p to p+49, p=50(i-1)+1</td>
<td>1 to 400</td>
<td>p to p+24, p=25(i-1)+1</td>
</tr>
<tr>
<td>coldBounds</td>
<td>All of DB</td>
<td>Rest of DB</td>
<td>Rest of DB</td>
<td>1001 ~ 2000</td>
</tr>
<tr>
<td>hotAccProb</td>
<td>-</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>coldAccProb</td>
<td>1.0</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>hotWrProb</td>
<td>-</td>
<td>0.2</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>coldWrProb</td>
<td>0.2/0.4/0.6</td>
<td>0.2</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>perPageInstr</td>
<td>30,000</td>
<td>30,000</td>
<td>30,000</td>
<td>30,000</td>
</tr>
<tr>
<td>thinkTime</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Transactions are represented as a string of page reference requests in which some are for reads and the others are for writes. There is a CPU instruction cost when a client performs a read or write operation. The database consists of a set of hot regions (one for each client), and a cold region. The hot region for a client is also considered as a private region for the client. The probability of an access to a page in the hot region is specified; the remainder of the accesses are directed to cold region pages. For both regions, the probability that an access to a page in the region will involve a write (in addition to a read) is specified.

The UNIFORM workload has no spatial locality and a high level of data contention. Each client accesses the data uniformly throughout the whole database. The HOTCOLD workload has a high degree of spatial locality and a moderate amount of sharing and data contention among clients. Each client accesses the data from its hot region (80% of the time) and the cold region (20% of the time). The clients can update pages in both regions. HICON is a skewed workload, which is unlikely in client-server DBMS environment. Nevertheless, it is typically used to expose the performance tradeoffs in a very high contention environment.
PRIVATE is a CAD-like workload with the highest per-client data locality and no contention. The only inter-client sharing in the workload involves read-only data.

### 3.5 Results and Discussion

We use the cost and workload settings described in Table 2.2 and Table 3.2 to obtain the following results. Similar to previous studies [4][10][19][20][35][39][41][43], the overall system throughput (transactions per second) and abort rate are the major performance metric in this paper. The large client cache size assumption is not realistic in situations where the transaction size is very large or the client workstation buffer is shared by multiple transactions [35]. Consequently, we use a small client cache (5% of the active database size). We examine the impact of the relative gap among client/server CPU performance, server disk I/O performance, and network bandwidth under different workloads on the overall performance, and impact of transaction size on abort rate. To ensure the statistical validity of the results, the 90 percent confidence intervals for system throughput in commits/second were calculated using batched means. The confidence intervals were within a few percent of the mean. Each experiment was run ten times using ten different random number seeds.

#### 3.5.1 The UNIFORM Workload

The UNIFORM workload has no spatial (per-client) locality. The accesses are distributed uniformly throughout the whole database. All pages are accessed with the same write probability (\textit{coldWrProb}). As SADCC is fundamentally distinguished from ADCC and
AACC in self-tuning speculation, we first examine the impact of self-tuning speculation under different levels of contention with the transaction size as 20, and the server CPU speed and network bandwidth as 400 MIPS and 80 Mbps, respectively. This configuration prevents the server/client CPU and network bandwidth from becoming a bottleneck as the client population increases. Therefore, we can focus on the impact of different levels of contention. Finally, we study the impact of transaction size on the abort rate with $\text{coldWrProb}=0.2$.

3.5.1.1 Impact of contention/data-sharing

Figure 3.5 shows that at $\text{coldWrProb}=0.2$, as the client population increases from 10 to 40, the throughputs of all protocols drop. At the client number as 10 and 20, the three protocols display similar behavior. As the client number increases from 20 to 30 and 40, AADCC gradually outperforms both ADCC and AACC. When $\text{coldWrProb}$ varies from 0.2, 0.4 to 0.6, the throughput gap between SADCC and ADCC ranges from 0.9% to 4.2% under low data contention ($\text{coldWrProb}=0.20$), changes from 2% to 8.6% under medium data contention ($\text{coldWrProb}=0.40$), and varies from 2.8% to 8.9% under high data contention ($\text{coldWrProb}=0.60$); while the gap between SADCC and AACC corresponds to 0.7% to 6.9%, 6.5% to 16%, and 9.6% to 18.4% respectively.

This performance trend can be explained from several aspects. First of all, the increasing client population intensifies the data contention, since the number of cached copies for any given page increases. This contention increases the overhead of client caching, i.e., the communication latency for callback when a client wants to update a page which is also cached at other sites. Compared to the traditional server-based communication path, both
SADCC and ADCC reduce this latency using direct client-client (P2P) communication. Earlier discovery of data conflict can improve the performance [20].

Second, AACC relies on the server for callback handling and data forwarding, while SADCC and ADCC partially offloads this function from the server to the clients. When the contention increases, the server in AACC has to handle more lock callbacks, while in SADCC and ADCC, many more invalidations occur directly between the clients. Therefore, the server is more heavily loaded in AACC than in SADCC and ADCC. When the contention is high, increasing the client population causes the performance to degrade further.

Third, the P2P communication path also reduces the potential conflict interval. Consider the example in Figure 3.1, assuming that page \( m \) is not in use until A wants to update it at time \( t_1 \). On average, the invalidation request from A in SADCC and ADCC reaches A at \( (t_1+T) \), while the corresponding callback message in AACC arrives at A at \( (t_1+2T) \). Therefore, in SADCC and ADCC if A accesses or update the data during the interval \([t_1, t_1+T]\), A’s update will be blocked or aborted. However, this interval becomes \([t_1, t_1+2T]\) in AACC. The reduction in the potential blocking/abort window by SADCC and ADCC produces less contention than AACC (Figure 3.6).

Fourth, compared to ADCC, SADCC has lower message overhead by using piggyback messages, e.g. scenarios 1 and 3 in Figure 3.4.

Finally, when the contention increases, both ADCC and SADCC become blindly optimistic. In contrast, SADCC adaptively switches to pessimistic if the speculation risk is high, which reduces the abort rate due to mis-speculation. The integration of self-tuning
speculation and P2P communication makes SADCC have the highest throughput (Figure 3.5) and lowest abort rate (Figure 3.6) among the three protocols.

Compared to AACC, both SADCC and ADCC need to piggyback additional control information (256 bytes) between the server and clients for maintaining the client caching directory consistency, e.g. scenario 4 in Figure 3.4. This piggyback information does not generate additional messages but does consume additional network bandwidth. However, compared to the typical message size of a data page (≥4 Kbyte), this overhead is generally low. For example, Figure 3.7 shows that, in average, SADCC and ADCC have about 4.8% bandwidth overhead compared to AACC. Consequently, with contention, this small overhead does not prevent SADCC and ADCC from outperforming ADCC in a high-speed network environment.

Figure 3.5. Throughput (UNIFORM).
Figure 3.6. Aborts per commit (UNIFORM).

Figure 3.7. Message volume (Kbyte) sent per commit.
3.5.1.2 Impact of transaction granularity

The aborts in asynchronous schemes, such as SADCC, ADCC and AACC, typically are resulted from either deadlock or mis-speculation. Due to temporal locality between read/write operations within the same transaction, the write/write conflict becomes important as the contention increases. Similar to AACC, SADCC and ADCC also employ two optimization techniques, Sneak-Through and Blocking Reversal [35], to reduce deadlock-abort. However, due to lack of global data-sharing information, the self-tuning speculation and P2P communication used in SADCC for reducing abort rate cannot be applied on AACC without substantial changes.

Figure 3.8 shows that, with the write probability as 0.2, when the transaction size varies from 20 to 5, the mis-speculation abort becomes dominant over the deadlock abort. For example, when the transaction size is 5, the abort is almost purely due to mis-speculation. This trend follows the intuition. The shorter is the transaction size, the faster does a transaction finish. In other words, in average, a transaction with smaller size has shorter blocking time. Therefore, the lower probability is the deadlock to occur.

The direct P2P communication in SADCC and ADCC helps reduce the abort rate because of the earlier discovery of read/write and write/write conflicts. Using the self-tuning speculation optimization, SADCC further reduces the abort rate due to mis-speculation by more than 30% in average (Figure 3.8). It is noted that, when SADCC switches from optimistic to pessimistic, it reduces the abort rate due to mis-speculation but might generate additional blocks. A small portion of these blocks might lead to additional deadlock-aborts.
For example, when client population is 30 and transaction size is 20, SADCC has the highest deadlock-abort rate among the three protocols. However, compared to the large reduction in mis-speculation aborts, these additional deadlock-aborts are insignificant. Figure 3.8 shows that SADCC has the lowest overall abort rate among the three protocols.

![Figure 3.8. Abort rate (UNIFORM). In notation $x$-$y$-$z$, $x$, $y$ and $z$ denote the protocol name, client population and transaction size, respectively.](image)

From the above discussion, it can be seen that SADCC improves throughput and reduces abort rate compared to ADCC and AACC. The increased data contention in the UNIFORM workload makes the integration of self-tuning speculation and P2P communication an important aspect for producing these improvements.
3.5.2 The HOTCOLD Workload

The HOTCOLD workload has high spatial locality and moderate write/read and write/write sharing among the clients. By adjusting the server CPU and network bandwidth, we examine the impact of the relative gap among server/client CPU performance, network bandwidth, and disk I/O performance.

3.5.2.1 Impact of a fast CPU

When the server CPU speed increases from 200 to 400 MIPS, SADCC, ADCC and AACC show improvements in throughput. The reduction in transaction execution time reduces the write/read conflict blocking time in all algorithms. Figure 3.9 shows that for 10 to 40 clients, doubling the server CPU speed increases the throughput in AACC around 28% on average, while the improvement in ADCC is only about 20%. With slow CPUs, the server in AACC is heavily used, i.e. the server CPU’s utilization increases rapidly and exceeds 86% when the number of clients reaches 20 (Figure 3.10), which explains why the fast server CPU helps AACC more than SADCC and ADCC.

3.5.2.2 Impact of a fast network

It has been shown that there exists an optimal number of clients to achieve the highest overall system throughput, because the overhead for concurrency control gradually catches up with the gain due to client caching as the number of clients increases [43]. When the network bandwidth increases from 80 to 800 Mbps, SADCC and ADCC not only display
significant increase for all client population (an average of 62%), but also improve the
scalability by moving the optimal number of client population to 20. However, AACC
increases the throughput by 43%, and more important, the overall throughput consistently
drops as the number of clients increases from 10 to 40. It is noted that, with fast network, the
server CPU’s utilization in AACC quickly reaches 91% at the client number as 20 and the
server almost saturates with the utilization as 97% for client number as 30 (Figure 3.10). On
the other hand, the increased bandwidth improves both the server and average clients CPU
utilization in SADCC and ADCC (Figure 3.10 and Figure 3.11). Consequently, SADCC and
ADCC display significant improvements in throughput (Figure 3.9).

The server is often eventually the bottleneck for performance and scalability due to the
excess demands for pages [19][37]. However, because the server is always on the critical
path for enforcing concurrency control in AACC, this server-based communication makes
the situation even worse. From the simulation results, it can be seen that, due to the server-
based communication path, SADCC and ADCC will benefit more from expected technology
advancements than AACC. At the same time, SADCC performs better (an average of 8%)
than ADCC at the large number (i.e. 40) of clients, since the contention increases as the
client population increases and the self-tuning speculation helps SADCC reduce wasted
work.
Figure 3.9. Throughput (HOTCOLD).

Figure 3.10. Server CPU utilization (HOTCOLD).
3.5.3 The HICON Workload

In HICON workload, all clients access the shared data region 80% of the time and the rest of the database 20% of the time. All write operations are conducted on the shared data. It displays a skewed data access pattern. This workload is not usually present in data-shipping applications [19]. We include this workload primarily to examine the robustness of SADCC under extreme data contention situations.

SADCC, ADCC and AACC suffer from increased conflict rates as clients are added. Figure 3.12 shows that SADCC and ADCC outperforms AACC consistently since AACC suffers more from data conflicts. The shorter communication path in SADCC and ADCC also reduces the blocking time. SADCC displays slightly higher throughput than ADCC.
because of the self-tuning speculation and lower message overhead (e.g. scenarios 1 and 3 in Figure 3.4).

Increasing the data contention leads to increases in the block and abort rates (Figure 3.13). Consequently, all three schemes exhibit thrashing behavior. For example, for 40 clients, SADCC, ADCC and AACC produce about 0.32, 0.33 and 0.42 aborts per commit, respectively. SADCC and ADCC have lower abort rates than AACC. This is due to the efficient P2P communication that reduces the latency for detecting data conflicts by about 50%. Earlier discovery of data conflicts can lower the abort rate and improve the performance [20]. More importantly, the deadlock abort overhead dominates the other overheads (Figure 3.13). Since all write operations in the HICON workload can be conducted on a small set of shared data only, the results (Figure 3.13) imply that, although most updates are frequently occurred on a small set of data items, aborts due to false speculation are not common. This indicates that using the number of write operations or conflicts for a unit time [14] is not an accurate criterion to identify the “hot/cold” data spots for asynchronous/synchronous cache consistency algorithms.
Figure 3.12. Throughput (HICON).

Figure 3.13. Abort rate (HICON). In notation $x$-$y$-$z$, $x$, $y$ and $z$ denote the protocol name, client population and transaction size, respectively.
3.5.4 The PRIVATE Workload

The PRIVATE workload has the highest spatial locality among the four workloads. With this workload, the clients perform writes only on their private hot regions and there is no write/read or write/write data sharing. The lack of data contention leads to no transaction aborts.

Figure 3.14 shows that, while SADCC and ADCC perform similarly, AACC slightly outperform ADCC at large number of population. As no write/read or write/write data sharing exists, no direct client-client communication occurs. SADCC and ADCC essentially downgrade to the server-based communication. The primary overhead of SADCC and ADCC over AACC is the additional network bandwidth for small control message (256 bytes) for maintaining client directory consistency. This overhead increases as the client population increases. However, compared with typical size of a data page (4K bytes), this overhead is low. Therefore, SADCC and ADCC do not exhibit significant downgrade. In average, the performance gap is 5.3% when the network bandwidth is 80 Mbps. This gap drops to 2.3% when the network bandwidth increases to 800 Mbps.
3.5.5 Discussion

An important feature of AACC is that a client uses piggyback message to inform the server of updates on private-read locked pages. The objective was to reduce the message overhead [35]. However, it may lead to late discovery of conflict under high contention environment. For example, Figure 3.15-a shows an example of late-abort due to this piggyback approach. Client A proceeds to update (Spec wr) a locally cached page which is under private-read lock at time $t_0$ and informs the server via a piggyback message (pgyb) at the same time, client B sends a read request (rd) which arrives at the server at $(t_0+15)$. Upon receiving the read request from B, the server forwards the data to B (data) but informs A to change the page state from private-read to share-read via a piggyback message (to simplify the illustration, this message is not shown on Figure 3.15-a). If A’s
piggyback message (pgybk LE) arrives at the server later than \( (t_0+15) \), then A’s speculation is not allowed to commit before B. Furthermore, if B also wants to update the page after caching that page, B proceeds the update and informs the server using an explicit message (LE) which arrives at the server at \( (t_0+45) \). As a result, at least one of the two clients has to abort (if the message \( \text{pgybk LE} \) arrives at the server before \( (t_0+45) \), B has to abort; otherwise, A has to abort). In contrast (Figure 3.15-b), if A informs its speculation to the server using an explicit message (LE), then A’s update will not be blocked by B’s read request (rd) and no additional aborts occur.

Another important hallmark is that AACC requires commit-time validation. When client A finishes the execution phase, it informs the server of the intention to commit. Before receiving the acknowledgement from the server, A has to stop. The server checks whether A can go ahead with its commit or not. If A has updated a page that has been read by another client, the server blocks A. If A has updated a page that has been read and then updated by another client (e.g. Figure 3.15-a), the server informs A to abort. The necessity for commit-time validation makes AACC more like the deferred schemes, such as AOCC [4].

Compared to AACC, ADCC uses parallel communication with simultaneous client-server and client-client communication, which may generate more messages. However, a significant portion of messages is handled in parallel in ADCC but in sequential in AACC. Note that due to the different overheads associated with the sequential and parallel message, the total number of messages sent is not an accurate metric for comparing the performance of these two schemes.
Figure 3.15. An example comparing the piggyback message approach in AACC to an asynchronous approach without using piggyback messages, where CB and LE denote callback and lock-escalation requests, respectively.

SADCC is developed based on ADCC but advances further. SADCC self-tunes between optimistic and pessimistic by statistically evaluating the speculation risk. To apply the self-tuning technique in a real-world workload, the information of access pattern can be obtained by dynamically monitoring the database. For example, by counting the accessing frequency within a certain time window, the hot/cold regions and write/read probabilities can be profiled.

In addition to the self-tuning speculation, piggyback messages are also employed in SADCC to reduce the message overhead. However, the design on using piggyback messages in SADCC is fundamentally different from AACC. Using piggyback messages reduces the total number of messages but may lead to significant communication delay, even if the network traffic is low. As discussed in Figure 3.15, the piggyback approach used in AACC requires commit-time validation and may result in late abort. In contrast, SADCC piggybacks acknowledgements for successful speculative updates but sends explicit messages for
speculation requests and unsuccessful speculative updates. If the network traffic is not the bottleneck, this design has little impact on performance, since the successful speculative client does not block when it wants to update a cached data. The network technology has evolved from 10Mbps in the early 1990s to Gigabits today. The bandwidth utilization in today’s Gigabit networks is typically low [22][36]. The design of using explicit messages for speculation request, invalidation, and negative responses in SADCC is due to the fact that earlier discovery of conflicts can lower the abort rate and cost [20] and the substantial advance in network technology.

To summarize, the key strength that distinguishes SADCC from ADCC and AACC is the self-tuning speculation technique. By adaptively switching between optimistic and pessimistic consistency control, SADCC reduces the speculation cost and improves the performance in a high contention environment.

The second strength of SADCC is the use of P2P communication for detecting data conflicts, which is similar to ADCC. P2P communication reduces the communication path under read/write and write/write sharing workloads. Consequently, it reduces the potential blocking window due to write/read and write/write conflict. It is also important for scalability as an increasing client population leads to higher overheads with client caching. Compared to AACC, the shorter communication path leads to fewer aborts and higher throughput in SADCC in a high contention environment.

The third strength of SADCC is that the functionality of concurrency control is partially offloaded from the server to the clients. As the power of client workstations is increasing rapidly, SADCC can better exploit client resources. In AACC, the server is the only source
for enforcing cache consistency and providing data. *SADCC removes the server partially from the critical path, which makes SADCC perform better for large number of clients.*

Nevertheless, compared to AACC, the advantages of SADCC do not come for free. Similar to ADCC, SADCC tags some directory information into the data and speculation request messages. However, the total number of affected messages is limited. The size of related information depends on how many clients have cached the data. To ensure the performance gain due to client caching, the number of sharing clients is usually limited. For extreme situations which require a large number of clients to cache data, SADCC and ADCC can reduce the overhead by using a coarse directory representation, i.e. using a flag to represent a group of clients [26]. In addition, both SADCC and ADCC piggyback control information (256 byte) to maintain the directory consistency, which consumes additional network bandwidth. Compared to the typical message size of a data page (≥4 Kbytes), the bandwidth overhead is low. Moreover, there has been significant advance in network technology during the past decade. A study conducted in 2003 shows that, with 2GB networks, typically less than 20% of the peak bandwidth is used [36].

### 3.6 Summary

An efficient cache consistency protocol, Self-tuning Active Data-aware Cache Consistency (SADCC), has been proposed for data-shipping DBMS architectures. Using P2P communication, SADCC reduces network latency for invalidation messages for read/write and write/write sharing by 50% compared to the server-based communication scheme, while only increasing the network bandwidth overhead by around 4.8%. By statistically evaluating
the speculation risk, SADCC adaptively switches between optimistic and pessimistic. By integrating self-tuning speculation and parallel communication (client-client and server-client), SADCC not only improves the throughput but also reduces the abort rate. The experimental study shows that SADCC has an average of 14% higher throughput than AACC and 6% higher throughput than ADCC under the high contention workloads; while under the non-contention workload, both SADCC and ADCC display a slight reduction (2.3%) in throughput compared to AACC with a high-speed network environment.
Chapter 4: An Adaptive Dual Control Framework for QoS Guarantee

The widespread deployment of the advanced computer technology in business and industries has demanded the high standard on quality of service (QoS). For example, many Internet applications, i.e. online trading, e-commerce, and real-time databases, etc., execute in an unpredictable general-purpose environment but require performance guarantees. Failure to meet performance specifications may result in losing business or liability violations. To design a computer system with certain performance guarantees becomes increasingly important.

While the previous two chapters focus on client cache consistency for data-shipping DBMS architecture, this chapter addresses quality of service (QoS) for generic distributed systems. A QoS-aware web-caching system is used as a concrete example for validation study.

4.1 Background and Motivations

The traditional approach to designing a computing system with QoS guarantees has been to quantify hardware capability, software execution requirements, resource demands, and workload characteristic, then apply an appropriate combination of pre-run-time analysis, admission control, and resource allocation algorithms to ensure that the system is not overloaded and that the desired performance is achieved.
The system performance typically depends on the interaction of multiple components. To isolate the impact on performance of individual components becomes difficult in a complex computer system. An alternative approach, which was originated from the control theory, is to treat the whole system as a “black box” and identify the “black box” behavior by running standard workloads. During recent years, there have been several designs based on the conventional feedback control theory for QoS guarantee [2][30].

In order to meet the design requirements, the control-theoretic approaches rely on an accurate model for the system behavior. The fixed parameter controllers based on the conventional feedback theory can be designed to be somehow insensitive against process variations, noise, or disturbance. However, such controllers must, by nature, be conservative in the sense that the bandwidth of the closed loop system has to be decreased to reduce the influence of the variation in the process. Substantial changes of process behavior can significantly degrade control loop performance [6].

Due to the increasing scale and complexity of distributed computing systems, their behavior becomes more difficult to predict. For example [23], enterprise-scale storage systems are large (with capacities often in the order of 100s of TBs), distributed, and increasing heterogeneous, with constantly evolving hardware and software. The model parameters oftentimes depend on the hardware and software configuration. It is impractical to repeat system profiling every time a system upgrades. Moreover, the system behavior oftentimes depends on workload. Their workloads are complex consisting of multiple overlapping I/O streams with unpredictable request patterns. The dynamic changing behavior of the distributed computing systems imposes hurdles on fixed parameters designs for QoS guarantee.
In the area of automation and control, adaptive control is an important way to handle system uncertainties. For example, the dynamics of an airplane change significantly with speed, altitude, angle of attack, and so on. The traditional constant gain, linear feedback can work well in one operating condition. However, difficulties can be encountered when operating conditions change. An adaptive control system integrates the controller design with on-line recursive parameter estimation (system identification). As a result, the system controller can automatically adjust its parameters in response to changes in process and disturbance dynamics.

The parameter-varying characteristic of large computing systems makes the self-tuning adaptive control theory an attractive technique. Recently, the adaptive control theoretic design has been advocated for providing the QoS guarantees in large distributed environments [29][23]. A case study was conducted in designing an adaptive controller for managing cache resources in QoS-aware servers, and proposed the adaptive control framework as a general approach for other types of QoS guarantees [29]. An adaptive controller based on an on-line recursive least-square estimator was designed to control access requests to a shared storage, and suggested that an adaptive control law is the only possible generic way to control a large-scale storage system [23].

Undoubtedly, the adaptive control theory is a powerful technique and has been widely used for controlling non-linear systems with unknown and changing behavior in avionics industry, automobile control, etc. Traditionally, adaptive controllers are based on the separation of system identification, i.e. parameter estimation, and controller design. The uncertainty of estimation is not taken into consideration for the controller design, and the parameter estimates are used for designing a controller as if they were the real values of the
unknown parameters. As a result, for the controller to be effective, the system identification process must be accurate and timely.

Unfortunately, there is an inherent conflict between identification and control in adaptive control systems – a conflict between asymptotically good control and asymptotically good parameter estimates [25]. To obtain good system information for identification it is necessary to perturb the process. Normally, the information about the system increases with the level of perturbation. On the other hand, the specifications of the control system are such that the output normally should vary as little as possible. There has been increasing interest in applying adaptive control theory to design computing systems in both academic and industry research recently [29][23], but this constraint has not been sufficiently addressed.

This work focuses on the applicability, limitations, and improvements of adaptive control theoretical design on computer systems. Based on the adaptive control framework [29] and the experimental results for QoS guarantees in a proxy cache, we conduct a simulation study. This chapter makes the following contributions.

1) Demonstrate that the adaptive control theoretical design is excitation-dependent. If a workload does not possess sufficient excitation levels, the model prediction for a computing system does not converge to actual values.

2) Demonstrate that there is no guarantee on the accuracy and convergence of on-line prediction under an uncertain environment. When the uncertainty is high, the prediction does not capture the system behavior.

3) Demonstrate that the adaptive control theoretic design is dependent on the initial “guess” on the parameters for an unknown system. If the “guess” deviates the real system significantly, the model prediction does not converge.
Most importantly, this work proposes an adaptive dual control framework that optimizes the tradeoff between the control goal and the system identification. In particular, when the uncertainty increases, the average hit-rate ratio provided by the adaptive dual control system and the conventional adaptive control system deviate from the desired hit-rate ratio by about 13% and 40%, respectively.

### 4.2 Revisiting Adaptive Control Framework

The aim of adaptive control is to automatically adjust the controller parameters in the case of both unknown and time-varying process parameters such that a desired degree of the performance index (such as cache hit rate, response time, etc.) is met. Adaptive control systems are characterized by their ability to tune the controller parameters in real-time from the measurable information in the closed-loop system. Most of the adaptive control schemes are based on the separation of parameter estimation and controller design. This means that the identified parameters are used in the controller as if they were the real values of the unknown parameters.

A challenge on applying the adaptive control theory on computer system designs is to represent a computer system using “control language”. Previous researches [29][23] typically assumed the system as a “black box”, whose current performance depends on a finite history of past measurement. Therefore, the system behavior can be described as a difference equation, whose parameters need to be identified. This section briefly reviews a previous framework for QoS guarantee [29].
4.2.1 The Framework Overview

The structure of an adaptive control framework for QoS guarantee in a web cache system [29] is shown in Figure 4.1. This model represents an application of adaptive control to provide proportional differentiation on relative average hit-rate ratio of different content classes. In general, there are $N \geq 2$ content classes in the system (for example, WML and regular HTML content). In a proportional differentiated caching service [29], the cache space is partitioned among classes, and assigning more storage space to a traffic class will increase its hit-rate and vice versa. Consequently, the quality spacing between classes is guaranteed by imposing constraints of the following form (equation 4.1) on successive pairs of classes, where $H_i$ denotes the measured average hit-rate of class $i$ content, and $S_i$ the allocated storage space for class $i$.

$$\frac{H_i}{H_j} = \frac{S_i}{S_j} \quad (i, j = 1, \ldots, N) \quad (4.1)$$

In the adaptive control web-caching system (Figure 4.1), an automatic model estimator (System Identification in Figure 4.1) periodically monitors actual system performance (relative hit-rate ratio) and current resource allocation (storage space ratio). A mathematical model (i.e. estimated plant parameters) is derived based on the input-output relationship. This mathematical model is dynamically fed to a controller design block which in-turn design a controller based on the estimated model. It is worthwhile pointing out that the estimated parameters are used for designing a controller by assuming that they were the real parameters.

A system with two content classes ($N=2$) was studied in [29]. An adaptive pole-placement controller was design to make the system output $y(k)$ (which represents the
measured relative hit-rate ratio \( \frac{H_i}{H_{js1}} \) at \( k \)th sampling time) track a reference trajectory \( y_m(k) \).

The task of the controller is to compute and provide the input \( u(k) = \frac{S_i}{S_{js1}} \) to the proxy cache such that the control goal is achieved.

\[ y(k) = -p_1 y(k-1) - p_2 y(k-2) + r_1 u(k-1) + r_2 u(k-2) \ldots (4.2) \]

**4.2.2 System Identification**

The web cache system was modeled as a second order linear and time-invariant system (equation 4.2), where the parameters \( p_1, p_2, r_1, \) and \( r_2 \) are unknown and need to be estimated. These set of parameters represent the web cache system behavior. Once the system behavior changes, these parameters changes too. At every sampling time (i.e. 30 second), the input \( u(k) \) and output \( y(k) \) measurements are used for identifying these parameters, which is based on the on-line regressive least square estimation.
Two workloads, synthetic and empirical workloads, were used in their study. Their system identification results revealed that the change of traffic leads to the change of parameters, which implies that fixed parameter controller does not work well for all traffic patterns. Based on the on-line system identification, the system models under the synthetic and empirical workloads are described in equations (4.3) and (4.4), respectively.

\[ y(k) = 1.13y(k-1) - 0.22y(k-2) + 0.0027u(k-1) + 0.0177u(k-2) \ldots (4.3) \]

\[ y(k) = 1.26y(k-1) - 0.34y(k-2) - 0.0082u(k-1) + 0.0517u(k-2) \ldots (4.4) \]

### 4.2.3 Controller Design

Based on the on-line estimation of the web cache parameters, an adaptive controller was designed based on the pole-placement strategy. Pole-placement is a standard and widely-use rule for designing controllers. The pole location determines the system transient-response, such as speed, damping ratio, or bandwidth. The basic ideal was to determine a controller that gives desired closed-loop poles. Therefore, the system follows command signal in a specified manner. The adaptive pole-placement controller applied to the QoS proxy cache system can be represented in equation 4.5, where \( l \) is the controller order, \( a_{l,j}(k) \) and \( b_{l,j}(k) \), \( j=1, \ldots, l \) are controller parameters that are adjusted automatically online. The detailed procedures for designing a pole-placement controller can be found in [6].

\[ u(k) = \sum_{j=1}^{l} a_{l,j}(k)u(k-j) + \sum_{j=1}^{l} b_{l,j}(k)[y_m(k-j) - y(k-j)] \ldots (4.5) \]

At every sampling time, the adaptive controller is fed with the output \( y \), the reference \( y_m \), and the plant input \( u \). The controller computes and produces the new plant input \( u \) for the
next sampling time. If the estimated parameters are accurate, the system output \( y \) should asymptotically track the reference \( y_m \). However, the uncertainty of the parameter estimate is not incorporated into the controller design.

### 4.3 System Identification Sensitivity

To design an effective controller that can track variations in system behavior, an accurate system model is essential. Therefore, the on-line system identification becomes crucial. Two criterion determine the effectiveness of the on-line system identification, i.e. whether the parameter estimates converge the actual values, and how fast the estimate converges. There are several practical issues that have a significant impact on these two criteria. While the adaptive control theory has been applied on QoS design in previous study [29][23], these issues have not been addressed.

In this section, through a series of sensitivity experimental study on the on-line system identification technique, we demonstrate the limitations of the adaptive control technique.

#### 4.3.1 Experimental Setup

We develop a digital model (Figure 4.2) for the differentiated proxy caching service [29]. This model consists of system identification only, and the plant dynamics is based on the results reported in [29]. We also implement the same on-line system identification algorithm, i.e. Regressive Least Square Estimation [6]. For on-line estimation, an initial ‘guess’ on the system order and the system parameters \([p_1, p_2, \ldots, r_1, r_2, \ldots]\) is required. If not indicate explicitly in the following experiments, the system behavior is assumed to be a second order
system and the estimates for the system parameters \([p_1, p_2, r_1, r_2]\) are initialized to some arbitrary values within \([-1, +1]\). These set-ups are similar to [29].

![Diagram](image)

**Figure 4.2. An on-line least square estimator for identifying a differentiated proxy caching system.**

In addition to the two workloads (synthetic trace and empirical trace) used in [29], we introduce a workload with uncertainty, stochastic workload. Under stochastic workload, the web cache displays uncertain behavior as described by equation 4.6, where the parameter drift is represented by \(\varepsilon(k)\), a white noise drift vector with zero mean and a small variance, \(var\). This workload represents the system dynamics transition between synthetic and empirical workloads but with some uncertainty. The magnitude of \(var\) determines the significance of uncertainty of the web cache behavior.

\[
y(k) = [1.195 + \varepsilon_1(k-1)]y(k-1) - [0.28 + \varepsilon_2(k-1)]y(k-2) - [0.00545 + \varepsilon_3(k-1)]u(k-1) + [0.0347 + \varepsilon_4(k-1)]u(k-2)
\]

\[\text{...}(4.6)\]

We compare the estimator performance using four types of excitation signals. Table 4.1 summarizes the three workloads and four excitation signals, i.e. storage space ratio.
Table 4.1. Summary of the three workloads and the four excitation signals.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Synthetic trace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Empirical trace</td>
</tr>
<tr>
<td></td>
<td>Stochastic trace with variance of $\epsilon(k)$ as 0.0002, 0.002, 0.02, and 0.2.</td>
</tr>
<tr>
<td>Excitation $u$ (storage space ratio signal)</td>
<td>Ex-1: $4\cos(2t)$</td>
</tr>
<tr>
<td></td>
<td>Ex-2: $\cos(t) + \sin(2t) + \cos(3t) + \sin(5t)$</td>
</tr>
<tr>
<td></td>
<td>Ex-3: white noise with variance 0.1</td>
</tr>
<tr>
<td></td>
<td>Ex-4: white noise with variance 1.0</td>
</tr>
</tbody>
</table>

4.3.2 Sensitivity Analysis

In this section, we discuss the impact of excitation signal, workload uncertainty, and a prior knowledge on the system behavior.

4.3.2.1 Impact of Excitation

Whether the on-line estimator can predict the web cache behavior is the key to the adaptive controller performance. Therefore, the on-line parameter estimation must be accurate and on time. However, whether the on-line estimation can converge relies on the excitation signal, i.e. the storage space ratio $u$, to the web cache.

We conduct two sets of experiments with the workloads as synthetic trace and empirical trace, respectively. For each set of experiments, we use four different excitations as shown on Table 4.1. Figure 4.3 (a to d) shows the estimated parameters for the synthetic trace workload under the four excitations. For Ex-1 and Ex-2, the estimations for all parameters significantly deviate from the actual values. For example, the estimate for $r_2$ asymptotically approaches to 50 and 18 for Ex-1 and Ex-2 respectively, which are more than 1000 times for than the actual value, 0.0177 (Figure 4.3-d). For Ex-3, the estimated parameters changes slowly, and after
5000 seconds they all converge to the plant parameters as reported in [29]. When the variance of the white noise (represent the signal power) is increased, the estimated parameters in Exp-4 converge at a faster rate, i.e. all estimates converge after 3000 seconds. Similar patterns are observed for the estimated parameters under the empirical trace workload.

Obviously, it is impossible to design an effective controller based on the estimated parameters under the excitation 1 and 2. The comparison indicates that: (i) whether the estimated parameters converge is excitation-dependent; (ii) the frequency contents of an excitation are more important than the magnitude of the excitation.

While previous studies [29][23] show success on on-line system identifications, it is important to realize that the excitation signal must be persistently exciting or sufficiently rich [7][8]. Unfortunately, there is an inherent conflict between identification and control in adaptive control – a conflict between asymptotically good control and asymptotically good parameter estimates [25]. To obtain good system information for identification it is necessary to perturb the process. Normally, the information about the system will increase with the level of perturbation. On the other hand the specifications of the closed loop system are such that the output normally should vary as little as possible. Since the excitation signal to the web cache is generated by the controller which takes the feedback as input, good control may lead to a lack of identifiability due to a poor excitation. As a result, there is no guarantee that the web system will be properly excited.
Figure 4.3. The dynamics of estimated parameters with the synthetic trace.
4.3.2.2 Impact of Workload Uncertainty.

The workload trace behavior in reality consists of multiple overlapping I/O streams and is
dynamically changing with complex request pattern [29][23]. We model this uncertainty by
using the stochastic workload (Table 4.1). The variance represents the significance of the
uncertainty. Since under the excitation of Ex-4, the on-line parameter estimator displays the
best performance. In this section, we examine the on-line estimator performance under the
stochastic workload with Ex-4 as the excitation (Figure 4.4 a-d).

With the low uncertainty (lo uncert, var=0.0002), it takes about 500 seconds for $p_1$, $p_2$, and
and $r_2$, and 1600 seconds for $r_1$ to converge to their actual values, respectively. However,
with the medium uncertainty (me uncert, var=0.002), the convergence times for $p_1$, $p_2$, $r_1$ and
$r_2$ increase to about 1600, 1500, 2000, and 1900 seconds, respectively. With the high
uncertainty (hi uncert, var=0.02), the four estimated parameters finally converge after 3000s,
but the converging values significantly deviate from the actual values (>40%). When the
uncertainty further increases (unpred, var=0.2), there are considerable oscillations on the
estimated parameters even after 6000 seconds – the estimator is not able to predict the
dynamics of system behavior.

These results indicate that, it takes certain time for the on-line estimator to learn and
predict the system dynamics. The convergence and accuracy of the parameter estimates
depend on the system uncertainty. The on-line prediction works well when the uncertainty is
low. When the uncertainty is high, the prediction may not be able to capture the system
behavior.
Figure 4.4. The dynamics of estimated parameters with the stochastic trace.
4.3.2.3 Impact of Initial Guess on System Behavior.

Similar to [29], we have initialized the parameters $[p_1, p_2, r_1, r_2]$ to be some random values within $[-1, +1]$, which do not deviate too much from the actual parameters. In this section, we relax these constraints and examine the impact that if a priori knowledge is not available. We denote good guess, fair guess, and poor guess as the initialization parameter ranges as $[-1, +1]$, $[-10, +10]$, and $[-100, +100]$, respectively. Figure 4.5 (a – d) shows the prediction using Ex-4 as the excitation input.

Under the synthetic trace with good guess, the estimated parameters converge to the actual values after 500 to 1600 seconds. With fair guess, two parameter estimates (i.e. $p_1$ and $p_2$) significantly deviate (>100%) from the actual values even after 6000 seconds. With poor guess, all parameter estimates deviate from the actual values considerably – the estimator is not able to predict the system dynamics.

Under a large-scale distributed environment, it is more practical to assume that we do not have close a priori knowledge of the system behavior. This constraint reduces the prediction accuracy and increases the time for prediction to converge.

Under a large-scale distributed environment, it is more practical to assume that we do not have close a priori knowledge of the system behavior. This constraint reduces the prediction accuracy and increases the time for prediction to converge.
Figure 4.5. The dynamics of estimated parameters with different a prior knowledge.
4.3.3 **Summary**

The original motivation of applying adaptive control theoretical design to computing systems is to automate the performance tuning process. One of the critical factors is whether the on-line parameter estimation can be accurate and on time. The sensitivity analysis in this section indicates that sufficient excitation, low uncertainty, and good *a priori* knowledge of the system behavior and workload characteristic are important for achieving accurate and on-time estimate. However, there is a dilemma between asymptotically good control and asymptotically good parameter estimate. In addition, for a large-scale distributed computing system, the applications execute in an uncertain general-purpose environment, and the workload, as well as the system behavior, is dynamically changing. Moreover, good a priori knowledge might not be available.

As a result, to design a control strategy for a computing system that operates under uncertainty conditions should incorporate the existing uncertainty. The control signal should have the following properties: (i) it cautiously follows the control goal, which means, in the case of uncertainty parameters of the system, the control signal should be smaller (cautious) than the control signal in the system with known parameters; and (ii) after adaptation, it excites the plant to improve the estimation.

4.4 **An Adaptive Dual Control Framework**

In this section, we introduce a technique, adaptive dual control theory [16][18][42], for mitigating the inherent constraint of adaptive control theory, the dilemma between asymptotically good control and asymptotically good parameter estimates. The basic idea of
dual control theory is to incorporate the existing uncertainty in the control strategy with the control signal. First the controller must control the process as well as possible. Second, the controller must inject a probing signal or perturbation to get more information about the process. By gaining more process information, better control can be achieved in future time.

4.4.1 Overview

The structure of the adaptive dual control framework [45][46] for the QoS web cache system is shown in Figure 4.6. The main differences between an adaptive dual control system and a conventional adaptive control system (Figure 4.1) are the transmission of the accuracy of the parameter estimates from the estimation to the control design algorithm, and the combination of caution control and excitation. The utilization of the accuracy of the estimation for the controller design allows generating the optimal excitation and cautious control signal for an adaptive dual controller.

While the web cache system was modeled as a linear time-invariant system [29], we model the system as a discrete system with time-varying parameters (equation 1). Considering an enterprise-scale web cache system is larger, distributed, and increasing heterogeneous, with constantly evolving hardware and software, and its behavior also depends on workloads that consists of multiple overlapping I/O streams with uncertain request pattern, it is more reasonable to model the system with time-varying parameters.

\[ y(k + 1) = -a_1(k)y(k) + ... - a_n(k)y(k - n + 1) + b_1(k)u(k) + ... + b_m(k)u(k - m + 1) \ldots (4.7) \]
where \( y(k) \) is the actual ratio of hit rate, \( u(k) \) the control signal for adjusting storage space ratio, \( k \) is the discrete time index; \( a_i(k) \) and \( b_j(k) \) for \( i=1,...,n \), and \( j=1,...,m \) are the unknown time-varying system parameters. Equation 4.7 can be written in vector form as
\[
y(k + 1) = p^T(k)m(k) \ldots (4.8)
\]
where
\[
p(k) = [-a_1(k)\ldots-a_n(k)|b_1(k)\ldots b_m(k)]^T \ldots (4.9)
\]
\[
m(k) = [y(k)\ldots y(k-n+1)|u(k)\ldots u(k-n+1)]^T \ldots (4.10)
\]

We model the uncertainty by using an additional stochastic parameter drift (equation 4.11).
\[
p(k + 1) = p(k) + \varepsilon(k) \ldots (4.11)
\]
where the noise vector \( \varepsilon(k) \) is a white noise drift vector with zero mean and a small variance, \( var \). The parameter vector \( p(k) \) is estimated using the standard technique for on-line system identification.

There are two major issues in designing an adaptive dual controller, i.e. (1) selecting an appropriate performance index for control optimization; (2) describing the uncertainty of the controller parameters in the adaptive pole-placement control system as well as defining a measure for the uncertainty. For easy understanding, we omit the detail process of mathematical derivation. Readers can refer [16][18] for the detail.
We define the nominal output as the system response to the controller that provides the desired system dynamics when no uncertainty exists. It is clear that the control performance would be improved if, in case of disturbances and parameter uncertainties, the controller tried to bring the system output as close as possible to the nominal output after complete noise compensation. The following two cost functions to be minimized are introduced in order to derive the control law, where $\mathcal{X}_k$ is the set of input and output values available at time $k$.

$$J^u_k = -E\{\beta^2 [y_u(k+1) - y(k+1)]^2 | \mathcal{X}_k \} \ldots (4.12)$$

$$J^e_k = -E\{[y(k+1) + \sum_{i=1}^m c_i y(k-i+1) - \hat{p}^T \mathbf{m}(k)]^2 | \mathcal{X}_k \} \ldots (4.13)$$

The first cost function, equation 4.12, is used for control purposes to minimize the deviation of the system output $y(k+1)$, from the unknown nominal output $y_u(k+1)$, which would be obtained by the adjusted unknown regulator. The coefficient $\beta^2$ is introduced for...
the simplification of further algebraic manipulations. The second cost function, equation 4.13, is used for the acceleration of the parameter estimation process by increasing the predictive error value, where \( \hat{p} \) is an estimate of \( p \), and \( c_j \) is determined the desired pole values. These two criteria correspond to the two goals of adaptive dual control: to control the system output and to accelerate the estimation for future control improvement. The adaptive dual controller is designed by solving this optimization problem (minimization of equations 4.12 and 4.13).

### 4.4.2 Experimental Results and Discussion

We have integrated the adaptive dual controller with the digit simulator (Figure 4.2) for the differentiated proxy caching service [29]. We have conducted a series of experiments by comparing the adaptive dual control framework with the adaptive control framework [29] under the three workloads (Table 4.1). In this section, we present the performance results under the stochastic workload with the low, medium, and high uncertainties (i.e. \( var=0.0002 \), \( var=0.002 \), and \( var=0.02 \)). Since the design goal is to provide the guaranteed QoS (i.e. the ratio of hit-rate between the two content classes), the performance index is the deviation of actual hit-rate ratio from the guaranteed hit-rate ratio NOT the absolute value of the hit-rate ratio.

When the uncertainty is low, the two controllers display similar behavior (Figure 4.7) at the stable state. As the control signal in the adaptive dual control system is intentionally smaller (cautious) than the control signal in the conventional adaptive control system with known parameters, the adaptive dual system is inherently conservative. As a result, the
adaptive dual control system displays sluggish behavior, while the conventional adaptive control system quickly gives a large overshoot at the beginning of the process. After obtaining the enough information (>1000 seconds), the adaptive dual control system provides an average hit-rate ratio as 65.3% with a standard deviation as 10.5%, while the conventional adaptive control system provides an average hit-rate ratio as 65.9% with a standard deviation as 10.1%. Compared with the desired hit-rate ratio 67%, both systems provide good service at the steady state.

![Figure 4.7](image)

Figure 4.7. Comparison of two systems under the stochastic workload with the low uncertainty.

The conservative behavior of the adaptive dual control system demonstrates significant advantages when the uncertainty increases. The conventional adaptive controller is designed by assuming the prediction is accurate and on-line. However, there is an inherent conflict between good control and good prediction. In addition, the existing uncertainty further
reduces the prediction accuracy. Based on inaccurate information, it is difficult to design a control strategy that can achieve the control goal effectively. Simply assuming the parameter estimates as if they were the real values of the unknown parameters makes the conventional adaptive control system more sensitive to the uncertainty. As a result, the conventional adaptive control system provides poor QoS with large oscillations (Figure 4.8).

![Figure 4.8. Comparison of two systems under the stochastic workload with medium uncertainty.](image)

On the contrast, by incorporating the prediction accuracy into the control strategy, and the combination of caution control and excitation, the adaptive dual control system provides QoS with much smaller oscillations. In particular, at the steady state (>1000 seconds), the adaptive dual control system provides an average hit-rate ratio as 80% with a standard deviation as 28%, while the conventional adaptive control system provides an average hit-
rate ratio as 127% with a standard deviation as 45%. Compared with the desired hit-rate ratio 67%, the adaptive dual control system displays much better performance than the conventional adaptive control system (Figure 4.8).

When the uncertainty is high, the system behavior changes more rapidly. Consequently, the on-line estimator cannot capture closely the dramatically changing behavior of the system. However, due to the caution behavior of the adaptive dual control system, the corresponding QoS displays smaller oscillations compared with the conventional adaptive control system (Figure 4.9). Without accurate and on-time information about the system characteristic, it is difficult to design a control strategy that can produce an effective control, no matter whether the control strategy is aggressive or conservative.

As shown in Figure 4.9, after 1000 seconds, the adaptive dual control system provides a relative hit-rate ratio in a range of [16%, 232%], while the conventional adaptive control system provides a relative hit-rate ratio in a range of [18%, 1100%]. Compared with the desired relative hit-rate ratio as 67%, neither the adaptive dual control system nor the conventional adaptive control system provides a reasonable QoS. This also verifies another point in the adaptive control theory – it may not be appropriate to apply the adaptive control techniques for systems with rapidly changing behavior [8]. Nevertheless, the adaptive dual control system is more stable and robust than the conventional adaptive control system.
Figure 4.9. Comparison of two systems under the stochastic workload with high uncertainty.

4.5 Summary

In this chapter, we have addressed the limitations of applying the adaptive control theory to design computing systems with QoS guarantee via sensitivity analysis. The effectiveness of the adaptive control technique on the computing system design relies on the excitation signal, a prior knowledge of the system behavior, and the environment uncertainty. We propose an adaptive dual control framework for QoS guarantee for mitigating these constraints. By incorporating the existing uncertainty of the on-line prediction into the control strategy, the dual adaptive control framework optimizes the tradeoff between the control goal and the uncertainty, and displays robust and cautious behavior. With the low uncertainty, the dual adaptive control system displays similar performance compared with the conventional adaptive control system. With the medium uncertainty, the dual adaptive
control system outperforms the adaptive control system significantly. When the uncertainty further increases and significantly affects the prediction accuracy, neither the dual adaptive control system nor the conventional adaptive control system provides reasonable QoS, but the dual adaptive control system displays more stable and robust performance.
Chapter 5: Related Work

The scope of previous work that is related to this dissertation extends from work that has focused on client cache consistency for data-shipping DBMS architectures to application of adaptive control theory to quality of service (QoS) design for distributed applications (i.e. differentiated caching services for proxy server, shared storage, etc.).

5.1 Client Caching Consistency

When a client wants to update a cached page copy in data-shipping systems, in order to maintain cache consistency, the server must be informed of this write intention sometime prior to the transaction commits. Table 5.1 compares ADCC and SADCC with the algorithms proposed during the last decade.

5.1.1 P2P Communication vs. Server-based Communication

The idea of ADCC originated with the memory coherency protocol used in SGI Origin multiprocessor systems [26]. ADCC is distinguished from Origin memory coherency protocol in several important aspects. ADCC is software-based and employs a two-tier directory, while the protocol in Origin systems is hardware-based and uses a single-tier directory only. Another important difference is that the coherency protocol in multiprocessor systems maintains a consistent view of memory for every processor on each memory operation. ADCC has a coarser granularity of atomicity, which requires a sequence of
operations to be executed as a whole. Therefore, ADCC must handle deadlocks and aborts at the transaction (a sequence of operations) level.

<table>
<thead>
<tr>
<th>Protocols</th>
<th>Invalid access prevention</th>
<th>Validity check initialization</th>
<th>Optimistic/ Pessimistic</th>
<th>Serialization mechanism</th>
<th>Consistency granularity</th>
<th>Communication path</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBL</td>
<td>Avoidance-based</td>
<td>Synchronous</td>
<td>Pessimistic</td>
<td>One-tier dir, Lock-based</td>
<td>Page</td>
<td>Srv-based</td>
</tr>
<tr>
<td>AACC</td>
<td>Avoidance-based</td>
<td>Asynchronous</td>
<td>Optimistic</td>
<td>One-tier dir, Lock-based</td>
<td>Page + object</td>
<td>Srv-based</td>
</tr>
<tr>
<td>ADCC</td>
<td>Avoidance-based</td>
<td>Asynchronous</td>
<td>Optimistic</td>
<td>Two-tier dir, State-based</td>
<td>Page</td>
<td>P2P + Srv-based</td>
</tr>
<tr>
<td>SADCC</td>
<td>Avoidance-based</td>
<td>Asynchronous</td>
<td>Self-tuning</td>
<td>Two-tier dir, State-based</td>
<td>Page</td>
<td>P2P + Srv-based</td>
</tr>
<tr>
<td>Notify Locks</td>
<td>Avoidance-based</td>
<td>Deferred</td>
<td>Optimistic</td>
<td>One-tier dir, Lock-based</td>
<td>Page</td>
<td>Srv-based</td>
</tr>
<tr>
<td>O2PL</td>
<td>Avoidance-based</td>
<td>Deferred</td>
<td>Optimistic</td>
<td>One-tier dir, Lock-based</td>
<td>Page</td>
<td>Srv-based</td>
</tr>
<tr>
<td>C2PL</td>
<td>Detection-based</td>
<td>Synchronous</td>
<td>Pessimistic</td>
<td>One-tier dir, Lock-based</td>
<td>Page</td>
<td>Srv-based</td>
</tr>
<tr>
<td>NWL</td>
<td>Detection-based</td>
<td>Asynchronous</td>
<td>Optimistic</td>
<td>One-tier dir, Lock-based</td>
<td>Page</td>
<td>Srv-based</td>
</tr>
<tr>
<td>AOCC</td>
<td>Detection-based</td>
<td>Deferred</td>
<td>Optimistic</td>
<td>One-tier dir, Time-stamp</td>
<td>Page + object</td>
<td>Srv-based</td>
</tr>
<tr>
<td>Cache Locks</td>
<td>Detection-based</td>
<td>Deferred</td>
<td>Optimistic</td>
<td>One-tier dir, Lock-based</td>
<td>Page</td>
<td>Srv-based</td>
</tr>
</tbody>
</table>

The algorithms can be classified into two categories according to their policy for invalid access prevention: avoidance-based and detection based [20]. All of the algorithms, except ADCC, completely rely on a centralized server for concurrency control. CBL is widely accepted as the leading algorithm due to its good performance and low abort rate [19]. In general, it has better performance than Caching Two-Phase Locking (C2PL) [39], No-Wait Locking (NWL) [39], Cache Locks [41] and Notify Locks [41]. Optimistic Two-Phase
Locking (O2PL) [10] and Adaptive Optimistic Concurrency Control (AOCC) [4] have similar or higher throughput. However, the major drawback of these two optimistic approaches is the deferred consistency check, which leads to high abort rates. The abort rate is a critical issue for users in the highly interactive environments that are common for page servers. Asynchronous Avoidance-based Cache Consistency (AACC) [35] can lower the abort rate while maintaining high throughput. However, both AOCC and AACC were proposed for adaptive locking, which switches locking between the page and the object level.

Franklin et al. [20] extended CBL with a global memory management concept that allows clients to obtain pages from other clients. Dahlin et al. [16] examined a similar concept, cooperative caching, in the context of distributed file system reads. ADCC differs from global memory management and cooperative caching in several important aspects. The most significant difference is that, in the global memory management or cooperative caching approaches, the server is still the only site with knowledge of where page copies are cached in the system and the only source for enforcing cache consistency. Therefore, clients are not able to detect data conflicts via the direct client-client (P2P) communication. In ADCC, in contrast, not only the server but also the clients each maintain a directory to track the global state of locally cached data. The direct P2P communication in ADCC decreases the communication latency for detecting data conflict. As a result, clients in ADCC are more actively involved in maintaining cache consistency and the server is partially removed from the critical path. By contrast, the primary goal of global memory management or cooperative caching is to reduce the server disk accesses by utilizing the remote client memory, while ADCC focuses on maintaining cache consistency with parallel communication (client-server
To integrate ADCC with the concept of global memory management or cooperative caching is an important direction for future work.

### 5.1.2 Self-tuning Speculation vs. Naive Optimism

Cache consistency algorithms can also be classified into two classes, ranging from pessimistic (synchronous) to optimistic (asynchronous and deferred). These techniques represent different tradeoffs between write intention declaration overhead and possible transaction aborts.

The pessimistic approaches, such as Callback Locking (CBL) [20] and Caching Two-Phase Locking (C2PL) [39], require clients to contact the server at the time that they first decide to update a page to which they do not currently possess write permission. The clients have to block before the server response. In the optimistic approaches, such as Adaptive Optimistic Concurrency Control (AOCC) [4], Cache Locks [41], Notify Locks [23], and Optimistic Two-Phase Locking (O2PL) [10], declaration of write intentions are deferred until the transaction finishes its execution phase. The asynchronous approaches are a compromise between the pessimistic and optimistic approaches. The asynchronous approaches, e.g. Active Data-aware Cache Consistency (ADCC) [43], Asynchronous Avoidance-based Cache Consistence (AACC) [35] and No-Wait Locking (NWL) [39], allow client to proceed to write the local copy under the assumption that the write intention declaration will succeed. If this optimism turns out to be incorrect, then the transaction must abort.

The simple assumption that no other clients want to update the same page makes traditional asynchronous and deferred approaches blindly optimistic in high contention.
environments. A possible approach to reduce the high abort rate in optimistic schemes was also proposed in [4], which assumed the system can automatically detect which data are under high contention, then the system switches to a more pessimistic protocol for such data. However, no techniques were provided for detecting or predicting “hot spot” data.

The most recent published hybrid concurrency control scheme we have found is [14], where the data contention “temperature” is maintained for each item. The scheme totally relies on the server to set the temperature state as hot or cold, depending on the number of write operations for a unit time, and inform relevant clients the temperature state conversion. Then clients dynamically switch between a synchronous scheme, C2PL [39], and a deferred scheme, Cache Locks [41]. However, SADCC self-tunes between synchronous and asynchronous consistency control. The fundamental difference between the pessimistic (synchronous) and optimistic (deferred and asynchronous) schemes is the speculative update. Due to the lack of speculation, the pessimistic approaches encounter deadlock-related aborts only. Because clients defer all of their write notification messages until commit time, the deferred approaches encounter stale cache aborts only.

By contrast, the asynchronous approaches encounter both deadlock and mis-speculation related aborts. SADCC focuses on reducing the mis-speculation aborts by self-tuning between synchronous and asynchronous consistency control. The mis-speculation aborts are mainly due to the simultaneous speculative update on locally cached data, which therefore leads to aborts of unsuccessful speculative clients.

It should be noted that a conflict between a write on a locally cached data and another remote write on non-cached data does not result in mis-speculation aborts. If the data-sharing
does not exist, a speculative update scheme is not likely to produce aborts due to false speculation. Consequently, the update frequency [14] does not accurately catch the “hot/cold spot” data which is more appropriate for synchronous/asynchronous schemes. For example, in our study, if a “hot” data is exclusively cached by a client, adopting a pessimistic approach can only deteriorate the performance. In addition, due to the passive nature of this scheme [14], it does not catch the changing user data access pattern accurately [35]. Without the global sharing information of cached data, it is difficult for clients to detect or predict “hot/cold spot” data accurately for synchronous/asynchronous consistency control.

5.1.3 Concurrency Control Granularity

The granularity of concurrency control (locking and callbacks) leads to two categories of algorithms, adaptive and non-adaptive. The non-adaptive schemes conduct locking and callbacks at single granularity, i.e. page or object, while the adaptive schemes adaptively switch between the page and object level. Franklin [19] demonstrated that non-adaptive schemes are the best overall choices. Carey et. al. [12] showed that adaptive schemes have better performance for workloads that exhibit fine-grained read/write sharing at the expense of additional complexity. It also showed that, for workloads with high page locality, the non-adaptive callback schemes perform as well as the adaptive schemes, or even better than the adaptive schemes, when the probability of a write is high. Whether a non-adaptive scheme or an adaptive scheme is better depends on the page sharing granularity in the workload [19].
5.2 Quality of Service Guarantee

The behavior of computer systems typically depends on the interaction of multiple components. As the systems are getting more complex, to isolate the impact of individual components becomes difficult. An alternative approach, which was originated from the control theory, is to treat the whole system as a “black box” and identify the “black box” behavior by running standard benchmarks.

During the past decade, there have been many feedback control-theoretic designs in different computer research areas, i.e. thermal management for microprocessor design [38], packet flow control in Internet routers [15], web server resource management based on web content adaptation [1][2][3], shared storage QoS [27], etc. All these designs used non-adaptive controllers that are designed off-line. Consequently, these designs rely on a pre-runtime model that describes the system behavior. If the system behavior changes, then the control loop performance degrades.

Due to the increasing scale and complexity of distributed computing systems, how to design a computing system with QoS guarantee becomes a challenge. The on-line system identification and auto-tuning properties of adaptive control technique have made the adaptive control an attractive direction for both academic and industry research. A case study was conducted in designing an adaptive controller for managing cache resources in QoS-aware servers, and an adaptive control framework was proposed as a general approach for other types of QoS guarantees [29]. More recently, an adaptive controller was designed based on an on-line recursive least-square estimator to control access requests to a shared storage, and an adaptive control law was suggested as the only possible generic way to control a large-scale storage system [23].
The objective of applying the adaptive control theoretical design is to automate the system identification and controller design. However, there is an inherent conflict between asymptotically good control and asymptotically good system identification [25]. This problem has not been sufficiently addressed in designing a computing system. In addition, the accuracy of the on-line estimation is not incorporated in designing the control strategy – this makes the conventional adaptive control more sensitive to the existing uncertainties.
Chapter 6: Future Work

This chapter describes future work related to cache consistency techniques, P2P communication and self-tuning speculation, and application of adaptive control theory to computing system design.

6.1 Consistency and Byzantine Fault Tolerance

We have designed two techniques, P2P communication and self-tuning speculation, for maintaining client cache consistency on the data-shipping DBMS architecture. Another potential application of these two techniques is distributed storage and file systems. In particular, when Byzantine fault tolerance is desired, these two techniques provide potential for performance and scalability.

Aggressive client-side caching has become a widely held tenet for improving performance, scalability and resilience of a large distributed storage system [9]. For client-cached copies to be useful, however, cache consistency must be maintained, that is, cached copies should be updated when the originals change. We define weak consistency as the consistency model in which a stale document might be returned to the user, i.e. session semantics; and strong consistency as the model in which after a write completes, no stale copy of the modified document will ever be returned to the user, i.e. UNIX semantics. The exact definition of the completion of a write varies by the consistency approaches.

On the other side, the growing reliance of industry and government on online information services makes malicious attacks more attractive and makes the consequences of successful
attacks more serious. In addition, the number of software errors is increasing due to the growth in size and complexity of software. These trends make Byzantine-fault-tolerance an increasingly important factor in the system design. Unfortunately, Byzantine agreement [13] typically requires a number of messages quadratic in the number of participants and the process for reaching agreement takes a significant amount of time.

Existing storage and file systems tend to provide weak consistency due to the cost of maintaining strong consistency. Consequently, a stale document might be returned to the user, though infrequently. More important, the design requirement for Byzantine-fault-tolerance increases the probability for inconsistent data concurrently existing in a system. Consider a Byzantine-fault-tolerant client/server storage system that provides weak consistency via the callback-based technique [24][33] and prevents faulty servers by replicating data at multiple server sites. Before providing data or giving write permission to a client, consensus must be established among all related servers. The cost (time) for reaching agreement typically increases rapidly with the desired level of fault-tolerance. This cost may significantly increase the occurrence of inconsistent data within the system. For example, when client A1 wants to write and client A2 wants to read the same data object that is in their local caches, to achieve Byzantine agreement in replicated servers can significantly delay the detection for this conflict. This long latency for detecting data conflict not only degrades the performance but also make the persistence of data update more unpredictable.

Current distributed file systems seem to emphasize the cost of achieving strong consistency. However, with weak notions of consistency users can observe confusing data, and some applications, such distributed databases, will likely to produce incorrect results. In
particular, longer is the latency for detecting write/read and write/write conflicts, more unpredictable is the persistence of data update. As the demand of data-sharing is increasing rapidly, the improved consistency will become increasingly desirable [39].

Choosing the consistency level according to which the system would allow a client to open, update, and close objects, without offering any guarantees about the persistence of those updates, is unfair to the user and undesirable from a system’s design point of view [31][39]. The embedment of Byzantine-fault-tolerance makes updates in a system with weak consistency guarantee further more unpredictable. From our view, a user-friendly distributed storage and file system should present a single-system view to applications. In other words, the distributed storage and file system should present the same, or as much as possible, interface and semantics to an application program as does a single-machine file system. Even an application distributed across many machines should see on a consistent storage system, not many slightly inconsistent ones. Existing applications should not require changes to work with the new storage system, and programmers should not need to learn a new set of skills to write new applications.

The techniques presented in this dissertation, P2P communication and self-tuning speculation, provide an opportunity of integrating strong consistency control with Byzantine fault tolerance for distributed file and storage systems. P2P communication can partially offload the expensive Byzantine agreement process from the critical path. Self-tuning speculation can reduce the probability of write/write conflicts. The integration of these two techniques can reduce the cost for achieving strong consistency on client-cached data in a large distributed Byzantine-fault-tolerant storage system. It should be noted that, different...
from DBMS architectures, the cache consistency in distributed file and storage systems is in the granularity of single operation level instead of transaction level.

6.2 Adaptive (Dual) Control Theory for Designing Computing Systems

We have demonstrated that the adaptive (dual) control theory can be applied to designing some distributed computing systems. However, how to generalize this technique to generic computing systems is still a big question.

Application of adaptive (dual) control theory typically requires a mathematical model in a form of differential or difference equations for describing system or application behavior. On the contrary, computing system and applications are oftentimes policy-based. It could be difficult to find a clean mathematical form which is suitable for applying adaptive control theory.

Stability, observability, and controllability are three fundamental system properties in control engineering. Stability, from a geometric point of view, is related to the properties of system trajectories around an equilibrium point. Controllability is another geometric property of a system, describing the ability to “drive” the system states to arbitrary values through the control input. Its dual notion of observability describes the ability to infer the system states given output measurements in an interval. It could be difficult or even impossible to prove that a computing system possesses these three properties. However, these three properties
should be dealt with if the adaptive control theory is to be extended to generic computing system design.

A good example is given in [23]. Two workloads compete for access to a shared storage infrastructure. A business critical workload W1 demands up to 350 IO/s, irrespective of other workload activities. Another workload W2 (e.g., one performing data mining) requires up to 550 IO/s. W2 is less important than W1, but it still requires at least 50 IO/s to make progress; otherwise the application breaks. So will W1, if it does not get 50 IO/s. To satisfy the combined throughput requirements of the two workloads, the three bands for throughput sharing is specified in Table 6.1 [23]. According to the specification, the first 100 IO/s in the system are shared equally between the two workloads, so that both can make progress. Any additional available throughput up to a total of 400 IO/s is reserved for W1. Thus, W1’s 350 IO/s are met first. Any additional available throughput is given to W2 until its 550 IO/s goal is met. Any further throughput in the system is shared equally between the two workloads.

Table 6.1. Example of two workloads sharing the system according to three throughput bands.

The top row shows the total system throughput in each band; the two rows below show the ratio by which the two workloads share that additional throughput. Any available throughput beyond band 2 is shared fairly (50-50) between the workloads [23].

<table>
<thead>
<tr>
<th>aggregated throughput (IO/s)</th>
<th>Band 0</th>
<th>Band 1</th>
<th>Band 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>workload 1</td>
<td>50%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>workload 2</td>
<td>50%</td>
<td>0%</td>
<td>100%</td>
</tr>
</tbody>
</table>

This is a typical example of policy-based systems. It is difficult to describe such a system in a clean mathematical form using control language. To assume the storage system as a
“black” box is a popular approach in current research for applying adaptive control theory in computing system design [23][29][45]. Adaptive control theory is usually effective for systems with slowly varying behavior. However, the system is likely to display quite different behavior at different bands, since the resource allocation policies at different bands are significantly different. When the storage system crosses over the band boundaries, how much impact it has on system stability, observability, and controllability is an important direction for future research.
Chapter 7: Conclusions

Client caching has become a widely-held tenet in designing modern client-server architectures. Typically, client caching can reduce the number of interactions between client and server, free server resources (i.e. CPU, disk), therefore improve the system performance and scalability. However, to allow multiple copies of the same data to coexist in the system, the cache consistency must be enforced.

This dissertation designs two protocols, Active Data-aware Cache Consistency (ADCC) and Self-tuning Active Data-aware Cache Consistency (SADCC), for improving performance and scalability on data-shipping DBMS architecture.

ADCC enables simultaneous communication between not only server and client but also client and client, by allowing clients to be aware of the global state of their cached data. Using P2P communication, ADCC reduces network latency for invalidation messages for write/read and write/write sharing by 50% in average compared to the server-based communication scheme, while increasing message overhead by only around 8%. Shortening the communication path not only improves throughput but also reduces the abort rate. By partially removing the server from the critical path for cache consistency, ADCC scales better than CBL. Both the simulation results and the analysis indicate that the overhead of ADCC is low. The experimental study shows that ADCC outperforms CBL under the four workloads tested. In particular, without per-client data locality, the increasing level of data contention leads to a higher performance gap between ADCC and CBL.
SADCC integrates self-tuning speculation with P2P communication. By statistically evaluating the speculation risk, SADCC adaptively switches between optimistic and pessimistic consistency control. The integration of self-tuning speculation and parallel communication (client-client and server-client) in SADCC not only improves the throughput but also reduces the abort rate. The experimental study shows that SADCC has an average of 14% higher throughput than AACC and 6% higher throughput than ADCC under the high contention workloads; while under the non-contention workload, both SADCC and ADCC display a slight reduction (2.3%) in throughput compared to AACC with a high-speed network environment.

This dissertation also addresses the resource management for large scale distributed systems. The self-tuning characteristic of adaptive control theory has made it an attractive approach for automating the profiling and performance tuning. However, the inherent limitations of adaptive control theory for designing computing systems have not been addressed. Via sensitivity analysis, this dissertation demonstrated these limitations of applying the adaptive control theory to design computing systems with QoS guarantee. The effectiveness of the adaptive control technique on the computing system design relies on the excitation signal, a prior knowledge of the system behavior, and the environment uncertainty. In addition, this dissertation also proposes an adaptive dual control framework for mitigating these limitations. By incorporating the existing uncertainty of the on-line prediction into the control strategy, the dual adaptive control framework optimizes the tradeoff between the control goal and the uncertainty, and displays robust and cautious behavior. With the low uncertainty, the dual adaptive control system displays similar performance compared with the conventional adaptive control system. With the medium
uncertainty, the dual adaptive control system outperforms the adaptive control system significantly. When the uncertainty further increases and significantly affects the prediction accuracy, neither the dual adaptive control system nor the conventional adaptive control system provides reasonable QoS, but the dual adaptive control system displays more stable and robust performance.
Chapter 8: Bibliography


