

## **INFORMATION TO USERS**

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

# **UMI**

**A Bell & Howell Information Company**  
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA  
313/761-4700 800/521-0600



Multiview Model for Protection and Access Control

by

Dawn N. Jutla

A Thesis Submitted to the  
School of Computer Science  
in Partial Fulfilment of the Requirements  
for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

APPROVED:


  
\_\_\_\_\_  
Dr. Peter Bodozik, Supervisor

  
\_\_\_\_\_  
Dr. Philip Cox, Computer Science

  
\_\_\_\_\_  
Dr. Allan Jost, Computer Science

  
\_\_\_\_\_  
Dr. William Phillips, Dept. of Applied Mathematics

  
\_\_\_\_\_  
Dr. William Robertson, Dept. Of Electrical Engineering

  
\_\_\_\_\_  
Dr. Michael Stumm, University of Toronto, External Examiner

TECHNICAL UNIVERSITY OF NOVA SCOTIA

Halifax, Nova Scotia

1996



National Library  
of Canada

Acquisitions and  
Bibliographic Services

395 Wellington Street  
Ottawa ON K1A 0N4  
Canada

Bibliothèque nationale  
du Canada

Acquisitions et  
services bibliographiques

395, rue Wellington  
Ottawa ON K1A 0N4  
Canada

*Your file / Votre référence*

*Our file / Notre référence*

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-31529-0

**Canada**

TECHNICAL UNIVERSITY OF NOVA SCOTIA LIBRARY

"AUTHORITY TO DISTRIBUTE MANUSCRIPT THESIS"

TITLE:

Multiview Model for Protection and Access Control

The above library may make available or authorize another library to make available individual photo/microfilm copies of this thesis without restrictions.

Full Name of Author:

DAWN NATALIE JUTLA

Signature of Author:



Date:

01/30/97

# TABLE OF CONTENTS

<b>LIST OF TABLES.....</b>	<b>IX</b>
<b>LIST OF FIGURES.....</b>	<b>XI</b>
<b>LIST OF ABBREVIATIONS.....</b>	<b>XII</b>
<b>ACKNOWLEDGEMENTS.....</b>	<b>XIV</b>
<b>ABSTRACT.....</b>	<b>XVII</b>
<b>1. INTRODUCTION.....</b>	<b>1</b>
1.1 Thread Synchronization and Address Space Isolation .....	1
1.2 Objectives .....	3
1.3 Outline of the Chapters .....	4
<b>2. RELATED WORK.....</b>	<b>6</b>
2.1 Synchronization Mechanisms.....	7
2.1.1 Semaphores.....	8
2.1.2 Monitors .....	8
2.1.3 Message Passing.....	9

2.1.4	Barriers.....	9
2.1.5	Virtual Memory Synchronization Mechanisms.....	9
2.2	Virtual Memory Protection.....	10
2.2.1	Software Based Virtual Protection Mechanisms.....	11
2.2.1.1	Software Virtual Memory Primitives Used For Synchronization.....	11
2.2.1.2	Address Space Isolation.....	12
2.2.2	Hardware-Based Schemes for Virtual Memory Protection.....	14
2.2.2.1	The PA-RISC Design.....	15
2.2.2.2	The Protection Lookaside Buffer Design.....	16
2.2.2.3	Sharing Within The PA-RISC And PLB Designs.....	17
2.3	Variable Sized Protection Units.....	19
2.3.1	Address Translation: Variable Sized System Pages.....	19
2.3.1.1	Simultaneous Support of Two Page Sizes.....	20
2.3.1.2	Architectural Support for Multiple Page Sizes: The MIPS R4000.....	21
2.3.1.3	Elastic Page Allocation.....	21
2.3.1.4	A Historical Perspective.....	22
2.3.2	Concurrency Control: Variable Sized Locking Units.....	22
2.3.2.1	Page Level Locking.....	23
2.3.2.2	Sub-Page Level Locking.....	24
2.3.3	Coherence Control: Variable Sized Coherence Units.....	25
2.3.4	Problems Faced By OSs Due To Architecture-Provided Variable Page Sizes.....	26
2.4	Customizable Operating System Services.....	27
2.4.1	Microkernel Operating System Support.....	28
2.4.2	Tempest And Typhoon: User Level Shared Memory.....	29

2.4.3. SPIN: An Extensible Microkernel for Application -Specific Operating System Services .....	29
2.4.4 Aegis: Lowering the OS Interface .....	30
2.5 Cache Functionalities .....	30
2.6 Conclusions.....	31
<b>3. MULTIVIEW MEMORY MODEL.....</b>	<b>33</b>
3.1 The Multiview Memory Model.....	33
3.1.1 A Conceptual Representation of The Multiview Memory Model .....	34
3.1.2 Flat Versus Segmented Address Space .....	37
3.1.3 Access Control State Model .....	37
3.1.4 Customizability .....	38
3.1.5 Kernel Primitives.....	39
3.1.6 Sharing of Memory Views In A Single Processor System.....	40
3.1.6.1 Sharing in Multiple Address Spaces .....	40
3.1.6.2 Address Space Isolation.....	42
3.2 Summary .....	42
<b>4. ARCHITECTURE.....</b>	<b>44</b>
4.1 Implementation Options.....	44
4.2 The Cache Protection Architecture.....	45
4.3 Variable Sized Protection Units and the Protection Lookaside Buffer .....	49



4.4 Miss Handling For The PLB Cache.....	51
4.5 Cache Entries.....	53
4.5.1 The Data Cache Entry .....	53
4.5.2 The PLB Entry .....	55
4.5.3 The FSM Cache Entry .....	55
4.5.4 ViewDefinition Cache Entry.....	56
4.6 Conclusions.....	57
<b>5. PROTOCOLS.....</b>	<b>59</b>
5.1 Virtual Cache Coherence Manager Protocol.....	59
5.2 Coherence Protocols.....	63
5.2.1 Invalidation-based Coherence Protocol.....	64
5.2.2 Update-based Coherence Protocol.....	67
5.2.3 Integration of Concurrency/Coherence Protocols.....	68
5.2.4 Comments.....	69
5.3 Support for Mach 3.0 Pager.....	69
5.4 Conclusions.....	70
<b>6. EVALUATION.....</b>	<b>72</b>
6.1 The Simulator.....	74
6.1.1 Simulation Input Parameters for the Multiview Memory Model Architecture.....	75
6.1.2 Memory Trace Input .....	75

6.1.3 Hardware Parameters Input .....	80
6.2 The Access Control Protocol.....	83
6.3 The Experiments.....	86
6.4 Multiview Lock Management Results.....	87
6.4.1 PCU Delay Results .....	87
6.4.2 Access Results for the Transaction Mixes .....	89
6.5 Conclusions of Results.....	92
<b>7. COMPARATIVE EVALUATION.....</b>	<b>93</b>
7.1 Software Implementation of a Lock Manager.....	93
7.1.1 Qualitative Comparison.....	95
7.1.2 Software Support for Multiview Lock Management.....	96
7.2 Statistical Results for Software Scheme .....	97
7.3 Summary and Conclusions .....	98
<b>8. CONCLUSIONS .....</b>	<b>100</b>
8.1 Contributions.....	100
8.2 Future Work .....	101
<b>REFERENCES.....</b>	<b>102</b>

<b>APPENDIX A</b> .....	<b>111</b>
<b>APPENDIX B</b> .....	<b>197</b>
<b>APPENDIX C</b> .....	<b>222</b>

# LIST OF TABLES

TABLE 2-1	CLASSIFICATION OF EXPLICIT AND IMPLICIT SYNCHRONIZATION MECHANISMS .....	8
TABLE 5-1	STATE TRANSITIONS FOR VIRTUAL CACHE COHERENCE MANAGEMENT [WHEELER 1992] .....	61
TABLE 5-2	MULTIVIEW FSM DEFINITION FOR [WHEELER 1992] VIRTUAL CACHE COHERENCE.....	62
TABLE 5-3	A MULTIVIEW FSM IMPLEMENTATION OF GOODMAN [1983] INVALIDATE PROTOCOL.....	66
TABLE 5-4	FIREFLY UPDATE PROTOCOL.....	67
TABLE 5-5	INTEGRATION OF CONCURRENCY AND COHERENCE PROTOCOLS.....	68
TABLE 5-6	KERNEL TRANSITIONS TO SUPPORT THE MACH 3.0 EXTERNAL PAGER.....	70
TABLE 6-1	THE NUMBER OF TUPLES TO BE INSERTED AND SIZES OF THE DATA TABLES USED BY THE TRANSACTIONS IN MIXES 1-4.....	78
TABLE 6-2	TUPLE SIZES FOR DATA TABLES.....	79
TABLE 6-3	NUMBER OF UNIQUE LOCK UNITS PER TRANSACTION MIX.....	79
TABLE 6-4	PERCENTAGES OF READ AND WRITE ACCESSES TO THE DATA TABLES FOR EACH APPLICATION.....	79
TABLE 6-5	OPERATIONS AND TYPICAL DELAY IN CYCLES.....	82
TABLE 6-6	STATE TRANSITION TABLE FOR LOCKING .....	86
TABLE 6-7	DELAY TO OBTAIN READ/WRITE LOCKS UNDER VARIOUS CACHE CONFIGURATIONS.....	88
TABLE 6-8	ACCESS CHARACTERISTICS OF TRANSACTION MIXES .....	90
TABLE 6-9	MISS RATES FOR TRANSACTION MIXES UNDER BASE CONFIGURATION (IN	

PERCENTAGES).....	91
TABLE 6-10 DATA ACCESSES FOR TRANSACTION MIXES.....	91
TABLE 7-1 STATISTICS FOR SETTING A LOCK FOR THE CONVENTIONAL LOCK MANAGER AND MULTIVIEW.....	97
TABLE 7-2 AVERAGE COST (DELAY) IN CYCLES FOR TRANSACTION MIX 1 .....	98
TABLE 7-3 DATA AND TLB ACCESSES FOR LOCK MANAGER AND MULTIVIEW FOR TRANSACTION MIX 1 .....	98

# LIST OF FIGURES

FIGURE 2-1	APPLICATIONS IN A SINGLE 32-BIT ADDRESS SPACE.....	13
FIGURE 2-2	PA-RISC PROTECTION DESIGN.....	16
FIGURE 2-3	THE PLB DESIGN.....	17
FIGURE 3-1	MULTIPLE VIEWS ON AN ADDRESS SPACE .....	35
FIGURE 3-2	SHARING VIEWS IN MULTIPLE ADDRESS SPACES.....	41
FIGURE 4-1	THE CACHES FUNCTIONALITIES.....	46
FIGURE 4-2	THE PCU UNIT.....	48
FIGURE 4-3	BIT ORDERING IN THE PHYSICAL ADDRESS.....	50
FIGURE 4-4	THE STATE STORAGE ENTRY VIRTUAL ADDRESS.....	53
FIGURE 5-1	STATE DIAGRAM FOR THE GOODMAN WRITE-ONCE PROTOCOL .....	64
FIGURE 6-1	BLOCK DIAGRAM OF HARDWARE.....	81
FIGURE 6-2	STATE TRANSITION DIAGRAM FOR LOCKING .....	84
FIGURE 7-1	STRUCTURES OF THE LOCK CONTROL AND LOCK REQUEST BLOCKS .....	94
FIGURE 7-2	LOCK TABLE IMPLEMENTATION.....	94
FIGURE 7-3	SOFTWARE SUPPORT FOR THE IMPLEMENTATION OF MULTIVIEW LOCK MANAGEMENT.....	96

# LIST OF ABBREVIATIONS

ALU	ARITHMETIC LOGIC UNIT
AU	ACCESS UNIT
AUID	ACCESS UNIT ID
BOT	BEGIN OF TRANSACTION
CPU	CENTRAL PROCESSING UNIT
DBMS	DATABASE MANAGEMENT SYSTEM
DC	DATA CACHE
EOT	END OF TRANSACTION
FIFO	FIRST IN FIRST OUT
FPU	FLOATING POINT UNIT
FSM	FINITE STATE MACHINE
FSMID	FSM IDENTIFIER
L1 DC	LEVEL 1 DATA CACHE (ON-CHIP)
L2 DC	LEVEL 2 DATA CACHE (OFF-CHIP)
LAPA	LOCAL AVAILABLE PAGE AREA
LCB	LOCK CONTROL BLOCK
LRB	LOCK REQUEST BLOCK
LRU	LEAST RECENTLY USED
LSB	LEAST SIGNIFICANT BIT
MMU	MEMORY MANAGEMENT UNIT
MRW	MULTIPLE READ-WRITE
MSB	MOST SIGNIFICANT BIT

OS	OPERATING SYSTEM
PCU	PROTECTION CONTROL UNIT
PDID	PROTECTION DOMAIN IDENTIFIER
PFN	PAGE FRAME NUMBER
PID	PAGE GROUP IDENTIFIER
PL	PRIVILEGE LEVEL
PLB	PROTECTION LOOKASIDE BUFFER
PTE	PAGE TABLE ENTRY
RISC	REDUCED INSTRUCTION SET COMPUTER
RPC	REMOTE PROCEDURE CALL
RTB	REVERSE TRANSLATION BUFFER
RW	READ-WRITE
SAPA	SYSTEM AVAILABLE PAGE AREA
SID	SUBJECT IDENTIFIER
TID	TRANSACTION IDENTIFIER
TLB	TRANSLATION LOOKASIDE BUFFER
TPC	TRANSACTION PROCESSING COUNCIL
UDP	UNIVERSAL DATA PACKET
VAC	VIRTUALLY ADDRESSED CACHE
VFN	VIRTUAL FRAME NUMBER
VIEWID	VIEW IDENTIFIER
VM	VIRTUAL MEMORY
WW	WRITE-WRITE



## ACKNOWLEDGEMENTS

Many people contributed to making the four years I spent working on my PhD degree and other things very enjoyable. It was a pleasure to work with my supervisor, Dr. Peter Bodorik, to whom I owe a great deal for my professional development. His knowledge of the research areas represented in this thesis was an invaluable resource for me. He has inexhaustibly contributed to and supported the final product presented here in many ways. I am very grateful for the time he took to straighten out a few of my operating system concepts and to hear me out even when we were disagreeing on some academic subject. Not only did he patiently wade through several drafts of the thesis but he made invaluable suggestions as to presentation and to the highlighting of the important contributions in this work.

Canadian Microelectronics Corporation (CMC) is acknowledged for providing the computer resources that were used for the quantitative analysis published in this thesis. The School of Computer Science at TUNS is also recognized for providing the equipment, academic training and scholarships that made this work possible.

During the years that I was registered in the PhD program, I was given the opportunity to teach at three Universities in Metro Halifax. I am compelled to mention the friends and colleagues that I made at these Universities because they have provided me with many fine moments. I especially appreciate the support of Dr. Francis Boabang, Chair, Dept. of Management Science and Finance of Saint Mary's University. My numerous reappointments for teaching positions were his responsibility and constituted a welcome subsidy to my scholarship. My friendship with Dr. Eric Lee of Saint Mary's University is especially treasured.

The all-female faculty whom I currently work with in the Information Management Department of Mount Saint Vincent University deserve mention for they provide me with a stimulating day-to-day working environment. Eve Rosenthal, Barbara Casey, Jean Mills, Glen Flemming, Paula Crouse, Dana Adams and Sonia Verabioff comprise a fabulous

group of enterprising individuals and I am very happy to be associated with their department.

The faculty at TUNS has been a constant in my academic life for the past six years. They have been instrumental in providing me with a good education and I respect their abilities and always enjoy seeing their familiar faces. I am also grateful to the administration for providing me with the opportunity to teach courses at TUNS. I have a lot of fun in the classroom when I am instructing the generally high-calibre TUNS students.

I thank the internal TUNS members of my examining committee, Drs. Peter Bodorik, Philip Cox, Allan Jost, Bill Phillips, and Bill Robertson for the work they did in reading and evaluating early sketches of the dissertation work, for both my PhD comprehensive presentation and for the progress presentation. I am also indebted to the gentlemen listed above, and to Dr. Michael Stumm for the work they did in preparing and conducting my oral PhD defence. In addition, I thank the External Examiner, Dr. Stumm, for his critical comments which further improved my work. Pam Griffin-Hody and the moderator, from the Graduate Studies office, are also warmly thanked for their input.

About treasured friendships and loved ones I leave for last because they are the most difficult to write. My grandparents, Arnold and Josephine (Tilda) Joseph are the reasons I do what I do. They have provided for me, guided me and loved me for all of my life; I cannot repay them. Serina Rambersingh and Karen Lank are very dear to me; Serina who lives in Trinidad has been my "best friend" since I was 14 years old and Karen for all the years I have lived in Halifax. They contribute greatly to my happiness. Marian and Cory Lee Qui are very special people in my life.

I would also like to thank Bill, Stella and Kate Lord for their friendship, and the invitations to join them at their very fun New Year's Eve, birthday and dinner parties. I would also like to acknowledge the Lords for their helpful participation in my wedding. Danyelle, Laura, Tasia, and Jason are warmly acknowledged for their kindness and friendship, and especially for all the hours of baby-sitting that they did for me. The community at Saint Paul's Anglican Church has contributed greatly to our life here, and though I have been remiss in attending services this year (since Logan was born), I look forward to rejoining them in the New Year.

My dearest husband, Eldon, and my son, Logan are the sources of my greatest delight. Eldon has shown me how easy it is to be successful in your professional life when you have a happy successful private life. He has helped me to create a wonderful environment for our baby son who in his turn has literally opened up an entirely new and remarkable world to us. I am indebted to Eldon's parents, Eugene and Joyce Olmstead for the many happy hours I spent at their home and particularly to Joyce for the warm and wonderful care she provides for Logan while I am at work.

Happy First Birthday, Logan!!

# ABSTRACT

In recent years, there has been an increasing number of applications that efficiently use virtual memory for access control purposes. In many cases, improved application performance has been attained when the protection unit size is different from that of a system page. Improved application performance has also been achieved through the provision of alternative access control protocols per application within a computer system. The advent of 64-bit architectures has caused the issue of maintaining isolation among threads working in a single address space to assume new importance.

This dissertation presents a novel virtual memory model which provides efficiencies to applications in terms of *variable-sized protection units*, customizability of access control service per region of memory, access control handling *without* the use of software handlers, synchronization at the level of threads, and fault isolation. These features are accomplished by imposing views on the regions of memory. Applications are provided with the flexibility of selecting from a large range of access control units' sizes (sizes must be powers of two). Protocols are implemented as fully as possible by table lookups. The potential reloading of the tables facilitates the customizability of the operating system control services. Synchronization at the level of threads and the maintenance of the views on the regions of memory facilitate fault isolation.

The widespread applicability of the model is demonstrated, in the thesis, by illustrating support for a number of access control protocols. The protocols are decomposed into state transitions, and these descriptions are supported by a cache-based protection architecture.

A qualitative analysis is performed to compare the performance of the proposed system with the performance of a traditional scheme. A quantitative performance study is conducted to compare the costs, in terms of machine cycles, of performing access control determination through the conventional method as opposed to using the virtual memory model with the chosen protection architecture and operating system support.

Results from the performance study show an overall improvement over other schemes in terms of cycle times. Much of this is due to a reduction in the frequency of faulting to software handlers, lesser complexity in maintaining status of shared data and non pollution of the TLB and primary data caches. The implementation of access control as shown in this thesis results in an increased hit rate in the TLB cache which is a critical factor to enhancing applications' overall performance.

# CHAPTER 1

## INTRODUCTION

### 1.1 Thread Synchronization and Address Space Isolation

This dissertation focuses on enforcement of protection upon access to virtual memory. Memory protection can take several forms: security, synchronization and isolation. Two aspects are considered in this thesis: access control in terms of thread synchronization and address space isolation.

Synchronization and address space isolation both refer to the means by which a system protects applications against illegal access to instructions or data stored in memory. Synchronization ensures correctness by providing controlled access (e.g. mutual exclusion, ordering of simultaneous multiple writers) to shared writable data and by the proper sequencing of processes. The latter requirement is known as conditional synchronization where threads (or processes) may need to wait until a set of variables is in a specific state before proceeding. For example in a shared buffer implementation, there would be a wait if the buffer was full and the next operation was an insertion. Mutual exclusion and conditional synchronization ensure correct execution of processes despite exposure to race conditions; the latter occurs when two or more threads (or processes) can access some shared data and the results depend on the order in which the threads (or processes) were run.

Address space isolation, in the context used throughout this thesis, refers to the prevention of access across non-shared *protection* spaces. A trend in recent operating system designs (e.g. Windows NT™) is to map operating system code, service managers' code and application code into one single process space. The advantage of such an organization is a saving in the number of context switches among the various code subsystems. However the boundaries among components begin to blur; hence the need for isolation among memory regions within one address space and the resulting distinction between a process space and a protection space. For instance, the operating system code will have its own protection space and so too would each individual service manager. Hence the protection space may be considered to be a subset of the process' virtual address space or may be the entire address space. The various subsystems mapped into a process'

address space besides its own code and data regions may also be referred to as extensions or untrusted modules.

Many issues from the latest advances in computer architecture, operating systems and database systems have motivated the design of the virtual memory model which is presented in this thesis. In many cases, improved application performance has been attained when the protection unit size is different from that of a system page [Stonebraker 1984, Kumar 1989, Appel 1991, Kagimasa 1991, Dubnicki 1992, Talluri 1992]. Hardware platforms currently provide for variable sized page sizes (e.g. MIPS R4000). However there is minimal OS support for the architecturally provided pages and the indexing of TLB entries becomes more complex.

Another issue motivating this dissertation is that current research indicates that improved application performance is achieved through the provision of alternative access control protocols per application within a computer system. Alternative access control protocols have traditionally been provided through provision of multiple user-level servers. This generates a substantial increase in context switching if the servers are in address spaces which are different from that of the client (e.g. in the Mach 3.0 microkernel organization) [Anderson 1991]. Context switching is avoided if servers and clients are mapped to a single address space, but protection boundaries must then be rigidly enforced through address space isolation. There is no current support for the enforcement of protection boundaries in many commercial applications (e.g. Microsoft's Object Linking and Embedding™, Xpress Quark™) where third-party code extensions are known to cause software crashes [Wahbe 1993].

The problem of address space isolation is even more critical when moving from 32-bit address spaces to the single space 64 bit-address spaces. Solutions appearing in recent 64-bit computer architectures (e.g. HP-PA RISC) feature the separation of address translation from protection. This facilitates the provision of different access rights by multiple subjects (e.g. threads) to the same object (e.g. a data item).

Specialized, dedicated hardware support has also been developed for memory management subsystems in order to support specific protocols. For example, Chang [1988] reported on one of the first complete attempts to incorporate transaction management functions in a Memory Management Unit (MMU) within IBM's 801 storage architecture. Locking is provided by a hardware mechanism which monitors the read and write references of individual transactions. The size of the lock unit is 128 bytes. A hardware locking scheme which operates concurrently with the address translation subsystem of the

MMU unit was described in [Bodorik, 1992b]. Hardware support for directory-based and snoopy cache protocols based on the write-invalidate and write-update mechanisms have been widely studied [Censier 1978, Archibald 1985, Agarwal 1988]. The hardware based solutions, although fast, are inflexible. This factor motivates features of the design for the architectural support of the memory model presented in this thesis.

The processor speed to memory access time ratio is increasing dramatically [Hennessey 1996]. Techniques to reduce the impact of the ratio, such as through the use of caches, are being explored. The importance of the use of caches as a means to provide increased performance is evidenced in [Horowitz 1987, Heinrich 1992, Schimmel 1994]. Caches are small high speed memories close to the processor. Data and instruction caches, on-chip and hierarchical caches, caches in single processor and multiple processor architectures are used for enhancing systems' performance. Caching is also an important technique for improving performance in distributed systems [Howard 1988; Nelson 1988]; it reduces network accesses and server loading. These motivate the use of caches in the implementation of the model proposed in this thesis since they are expected to enhance application performance.

## 1.2 Objectives

The introduction above provides a synopsis of the relevant developments in fields which motivated the objectives of this work. Chapter 2 gives a detailed literature survey on the various areas that impact this work.

The objectives of this dissertation include the design and support for a novel virtual memory model and protection architecture which package efficiencies to applications in terms of the following:

- variable-sized protection units
- customizability of applications through the choice of an access control protocol per region of memory
- flexible hardware to support access control handling *without* the use of software handlers in order to reduce kernel-user application communication
- on-the-fly access control which lowers access control overhead
- address space isolation within and across address spaces
- synchronization at the level of threads and/or tasks

- lower Instruction and Data TLB pollution
- increased concurrent access (e.g. multiple writers to the same system page)
- efficient support of different access rights to the same access unit by different subjects (fine grain protection)
- efficient hardware support for protection - specifically the separation of protection from address translation within a multiple address space where rights need not be checked on each and every memory access.

The thesis shows how all the above objectives are met. It demonstrates how operating systems can provide variable sized access control units without altering the underlying fixed sized paging implementation by the use of views. It provides a proposal for reduced context switching costs by placing decomposable access control protocols at the hardware and OS level. Another benefit of the latter is the reduction in data and instruction TLB pollution. Since the TLB is recognized as becoming the major bottleneck [Romer 1995] in future systems, this is an important advantage. The customizability of OS services can also be achieved on a per system basis. The thesis provides a costing of an implementation of the memory model and architecture in terms of CPU clock cycles. A performance comparison is made to a software implementation of a widely implemented conventional access control scheme.

### **1.3 Outline of the Chapters**

Chapter 2 provides further motivation for the objectives of this research along with the presentation of research related to this dissertation. It includes surveys of schemes which independently implement virtual memory access control, utilize variable sized units, and provide customizability of operating system and application services. Literature on the use of caches for improving performance is also presented in Chapter 2. Also included is a survey on synchronization mechanisms. These topics were sketched out in the Introduction and are thoroughly represented in Chapter 2 since my objectives include virtual memory hardware support designed for flexibility.

Chapter 3 presents the description of the Multiview Memory Model. How the model meets the objectives as outlined in Chapter 1 is discussed in detail.



Chapter 4 describes a design for the protection architecture required to support the memory model. The modifications to kernel data structures and to the entries within the data cache are described. Additional hardware structures are also defined.

Chapter 5 describes protocols which can be supported by the proposed model. The widespread applicability of the model is demonstrated qualitatively by illustrating support for various concurrency, coherence and paging protocols.

Chapter 6 presents the qualitative and quantitative evaluation of the Multiview model. The main metric for evaluation and comparative purposes is that of CPU clock cycles. The number of cycles of delay to determine whether a read or write access is allowed to proceed for an example protocol is measured for four applications under different cache configurations. The costs are given in terms of the number of accesses to the TLB, primary data cache, secondary data cache, other supporting caches, and main memory for a given application. The miss rates on the various caches are also provided in order to cost the memory hierarchy.

Chapter 7 provides detailed qualitative and quantitative comparative results of the software implementation of an example access control protocol with the Multiview implementation of the same protocol.

Chapter 8 provides a synopsis of the motivation of this thesis, a summary of thesis contributions, conclusions, and suggestions for future research on this topic.

# CHAPTER 2

## RELATED WORK

This chapter presents many details on the several areas of research related to this work. The work presented in this thesis overlaps with the fields of operating systems, computer architecture and with applications such as database systems. First, literature pertaining to conventional synchronization mechanisms is surveyed and problems are stated. Provision of efficient memory synchronization is the main thrust of this thesis, hence the important synchronization mechanisms and their associated advantages and disadvantages are covered in section 2.1.

Next, literature on virtual memory protection systems is presented in section 2.2. Many virtual memory applications make use of the computer's address translation hardware to facilitate access control. This lays a basis for an aspect of the work presented in this thesis, in that the hardware which supports virtual memory is explored to provide sophisticated access control. The work contributed in this thesis also features a solution to provide address space isolation.

Concurrency control and coherence control, which benefit from being serviced with *variable sized* control units, are also surveyed. A simple solution for the provision of access control to variable sized units is another contribution of this thesis. Literature pertaining to other solutions for enforcing access control on variable and fixed sized units is surveyed in section 2.3.

Various implementations of customizable operating system services are then overviewed in section 2.4. Research in many environments clearly demonstrates that application performance can be greatly improved by customizing operating system services, such as paging algorithms, concurrency control and coherence control algorithms, to suit the individual characteristics of an application. Customization of control services to applications is an additional contribution made in this thesis.

Applications of caches are presented in section 2.5. It is proposed in this thesis that tables, which are required to support implementation of access control to variable sized units, customizability, and the other objectives presented in Chapter 1, be stored in dedicated caches. Finally a summary is provided at the conclusion of the chapter.

## 2.1 Synchronization Mechanisms

This section first provides background on synchronization mechanisms and presents a further categorization for these mechanisms in terms of implicit and explicit invocations. The problems associated with explicit invocations of synchronization primitives will be discussed.

Synchronization may be accomplished through hardware, software or combination hardware/software mechanisms which regulate the sharing of data in memory. Hardware-based synchronization mechanisms may include the disabling or issuance of process interrupts, traps, and structural support for atomic read, write, and read-modify-write memory operations.

In this thesis, attention is restricted mainly to the uniprocessor environment, although synchronization support for policies in a shared memory multiple processor environment will be briefly examined. A further refinement in the categorization of synchronization mechanisms is introduced in this work: **explicit** and **implicit** synchronization. Table 2.1 presents the results of a survey, to follow, of standard mechanisms that identify the applicable category for the individual mechanism.

The definition of *implicit* synchronization in this work means that synchronization operations are induced as a by-product of the standard read and write memory operations or end of process execution. *Implicit* synchronization mechanisms are those which may potentially occur in parallel with some other activity such as address translation and data fetching. Synchronization is *explicit* when it is achieved through explicit use or activation of the synchronization mechanisms (e.g. signal, a call to lock). Features of selected synchronization mechanisms are examined below with a view to characterizing them as either implicit or explicit.

**Table 2-1 Classification of Explicit and Implicit Synchronization Mechanisms**

<b>Synchronization Mechanism</b>	<b>Explicit</b>	<b>Implicit</b>
Semaphore	x	
Monitor	x	
Message Passing	x	
Barriers	x	
Virtual Memory Protection Primitives	x	
Hardware Protection (e.g. dedicated protection caches, registers)		x

### 2.1.1 Semaphores

Semaphores are used for both access and sequence control. The proper sequencing of processes is known as conditional synchronization (sequence control) where threads (or processes) may need to wait until a set of variables is in a specific state before proceeding. The representation of a semaphore variable is typically an integer and a queue. The presence of the queue structure depends on whether busy waiting or positive-wakeup is used. The semaphore is classified as an explicit mechanism since operations, P and/or V, on the semaphore must be issued before access is allowed to data, and also on exit of the critical section. Operations on semaphores must be explicitly invoked for access control by the programmer who decides when to synchronize and upon what conditions; an inadvertent omission of a wait or signal operation can deadlock a program or result in incorrect execution. Generally, programs using semaphores are difficult to code, understand and prove correct. This is especially the case when multiple semaphores are used.

### 2.1.2 Monitors

The monitor is an encapsulation mechanism with a structure similar to an abstract data object. It provides mutual exclusion by allowing only one process to be executing within it at any instant. The monitor is classified as an explicit synchronization mechanism because the application must issue explicit calls to the monitor entry procedures. From the perspective of an application using a monitor, explicit synchronization occurs only when a monitor entry procedure is called. Everything else is implicit within the monitor.

There are several disadvantages associated with the use of monitors. The signal-wait complicates proving program correctness, since control of the monitor is relinquished to another process or thread on either the signal or wait operation. The state of the monitor can then significantly change between the times a process relinquishes the monitor and subsequently regains control. Additionally, outstanding signals on condition variables are not saved. Since the monitor is a programming language construct, the compiler must recognize it and arrange for mutual exclusion. However most programming languages (for example C and Pascal), do not support monitors and hence the compiler does not enforce mutual exclusion rules [Tanenbaum 1992].

### **2.1.3 Message Passing**

Monitors and semaphores are designed for solving the mutual exclusion problem on one or more CPUs that all have access to a common memory. The associated primitives are inapplicable to a distributed network of computers; they do not provide for an exchange of information among multi-computers. The message passing synchronization mechanism solves this problem.

Message passing, by definition, is an explicit synchronization mechanism. It is usually used across multiple address spaces in non-shared memory environments. Processes synchronize by the explicit sending and receiving of messages.

### **2.1.4 Barriers**

Parallel processes can be synchronized using barriers, another explicit synchronization mechanism. A barrier has a specific number of processes associated with it. Only when all participating processes have reached the barrier, can a process execute beyond it. A barrier can be implemented by a shared memory word which keeps counting the number of processes reaching the barrier [Hwang 1993]. An alternative hardware implementation appears in the form of wired barriers for fast synchronization. Synchronization using barriers is efficient when the synchronizing processes are short and/or the execution times are roughly the same. Otherwise, a subset of the processors executing the short-lasting participating processes will stall while waiting for some other longer process to synchronize on the barrier.

### 2.1.5 Virtual Memory Synchronization Mechanisms

Appel [1991] defines virtual memory primitives which manipulate the status bits in page table entries in order to effect synchronization. A process may modify the rights required to access the page and thus prevent other processes from accessing that page, thereby providing synchronization.

Use of the memory management primitives for synchronization can be more convenient and efficient than most standard synchronization mechanisms. One process may perform synchronization in accessing a page unilaterally without other processes actively participating in synchronization. This is used, for instance, in checkpointing [Appel 1991]. Note that with respect to the checkpointing example only one (active) process performs the synchronization without the other (passive) processes even being aware of the synchronization with the exception of potentially increased delays. Such synchronization is not only convenient but it is also efficient, in that it reduces the number of kernel calls when compared to user-level synchronization servers which utilize kernel calls.

The HP PA-RISC and the PLB designs, described later in Section 2.2.2, include mechanisms to provide extra hardware support for the automatic invocation of a check of access rights on access to virtual memory. These mechanisms are classified as *implicit*. Advantages include the following. The responsibility for synchronization is no longer a burden on the application programmer. The synchronization may be faster depending on the organization and speed of the hardware which supports it. There is an opportunity to reduce the number of context switches and system calls if the protection is provided in hardware as opposed through the use of user-level access control managers.

## 2.2 Virtual Memory Protection

The Virtual Memory (VM) system is being increasingly exploited to provide access control, in addition to its fundamental support of mapping functions among address spaces. In relatively recent proposals [Appel 1991, Eppinger 1989, Koldinger 1992, Li 1989, Wilkes 1992], the protection mechanism of the virtual memory system has been exploited by applications, kernels and operating systems for purposes such as the synchronization of processes. The adoption of these proposals in commercial operating and hardware systems (e.g. Windows NT™, Hewlett Packard PA-RISC) shows the rising emphasis on using virtual memory for efficiently implementing control mechanisms. Some example control applications are checkpointing and garbage collection.

The schemes which increase the functionality of the virtual memory mechanism can be categorized into software and hardware based and will be described in sections 2.2.1 and 2.2.2. A problem common to all the schemes is that the smallest unit of protection is the page.

## **2.2.1 Software Based Virtual Protection Mechanisms**

These schemes are further categorized according to whether they emphasize efficient support for synchronization as opposed to that for address isolation.

### **2.2.1.1 Software Virtual Memory Primitives Used For Synchronization**

Virtual memory primitives are used, by application programs, to manipulate the protection bits on page table entries for synchronization purposes. This is achieved either by the use of explicit OS primitives [Appel 1991] or through function calls in conjunction with exception handling as is done in Windows NT Version 3.1 [Kath 1992]. The page protection hardware can efficiently test simple predicates on addresses, subject to the restrictions discussed below, in order to determine whether access is allowed or not. This can provide substantial savings over the fetch and store operations used by applications which maintain lookup tables for control purposes (e.g. lock tables). For applications utilizing virtual memory primitives, the parts of the operating systems structures and code which support and define the primitives must be mapped into the user space of the calling application for efficiency purposes. Otherwise, the increase in context switching due to the crossover of user to kernel interface is too costly for this scheme to work efficiently.

Six Virtual Memory primitives were identified in [Appel 1991] as useful to many applications. They include the batched protection of  $n$  pages, removal of protection for one page at a time from a batch of protected pages, the protection of one page at a time, and the mapping of a page to two different virtual address ranges. The primitive which supports the dual mapping of a page facilitates the support of the rights of one thread to access the page via one virtual address, and enables the faulting of all other threads in the other virtual address space.

The Win32 subsystem in NT provides the *VirtualProtect()* and *VirtualProtectEx()* functions to change page protection. The Ex extension on the function name allows

changes in a page protection in processes other than the calling process, provided that the caller has appropriate privileges. Page protection states in the NT system are PAGE\_NOACCESS, PAGE\_READONLY and PAGE\_READWRITE. However 5 bits are reserved to represent page protection in each page table entry in the NT system, thereby allowing expansion of protection states to a maximum of 32, presumably for anticipated future use.

Windows NT uses conventional forward mapped page tables, but with an additional level of indirection for shared pages. To avoid having to update multiple Page Table Entries (PTEs) on the change of state of a shared page (i.e. on a write access), Windows NT implements another level in the page table for shared pages. A prototype PTE is introduced which references the shared page. Therefore, the previous level in the page table hierarchy points to the prototype PTE which in turn points to the page frame of the shared page. Only the prototype PTE entry needs to be changed.

Performance studies [Hosking 1993, Wahbe 1992] of page level access indicate that applications using software implementations of access control can outperform applications using virtual memory primitives for access control. For instance, a dedicated software locker can outperform a locker which uses VM primitives for locking pages. The problem is not the cost of the virtual memory primitives but the large page size on which they operate. Hosking [1993] states that sub-page protection and dirty bits, along with appropriate operating system interfaces should overcome the performance disadvantages that were observed. This further supports the motivation for providing access control on variable sized access units.

### **2.2.1.2 Address Space Isolation**

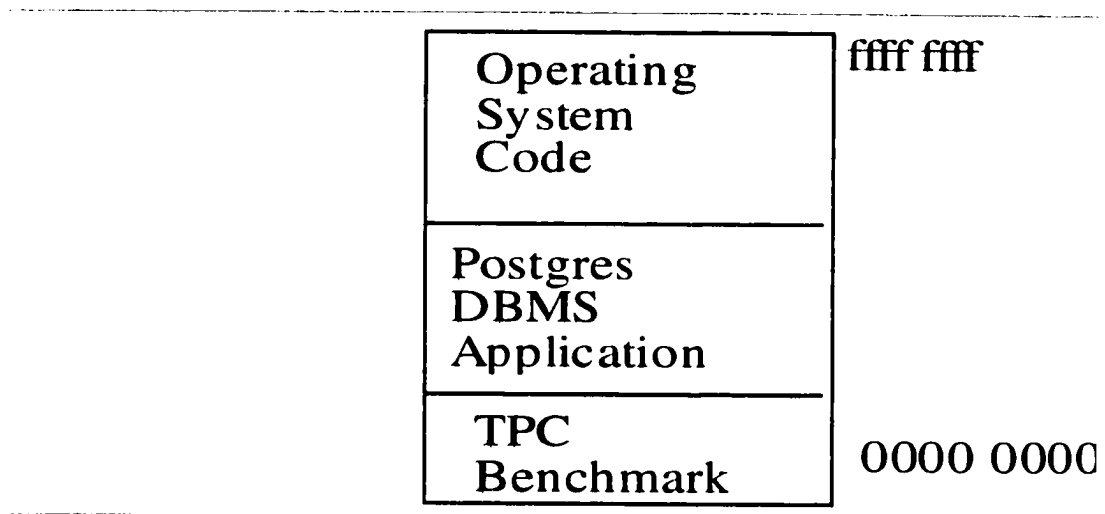
Operating systems such as Chorus [Rosier 1988] and Windows NT allow servers to be loaded in the same address space and protection domain as the kernel. This is an alternate to loading servers into their own address spaces.

Figure 2.1 illustrates Windows NT logical division of its address space. The Postgres DBMS and TPC Benchmark applications are not built-in to NT, but simply illustrate an example address space. Addresses of OS code and applications' code can be differentiated by examining the highest order bit. If it is set, the address references OS code or data. However no further support for distinction of addresses is made within the 2 Gigabyte user



code area. All servers or extensions mapped into the process's address space can potentially access any of the 2 GB area.

A software technique which imposes logical boundaries between protection spaces within a single address space is proposed in [Wahbe 1993]. Wahbe et al propose to divide the process' address space into segments such that addresses in each segment have a unique pattern of higher-order/upper bits. Protection spaces are then made up of these segments and are given unique segment identifiers. Each untrusted module or extension that is mapped into the process's address space is assigned at least two segments: one for its data and one for its code. Two segments are needed in order to prevent an untrusted module from modifying its code, while allowing it to modify its data.



**Figure 2-1: Applications in a single 32-bit address space**

Wahbe [1993] modifies the upper bits of addresses targeted within unsafe instructions using a software transformation technique called *sandboxing*. An unsafe instruction is one whose target address cannot be statically verified. Examples are jumps through registers which are commonly used to implement procedure returns, and stores that use registers to hold their target address. Sandboxing sets the upper bits of the addresses to the bit pattern of the protection identifier of the space in which the thread is currently executing. Hence, instructions issued by a process will only affect addresses in its own protection space. Sandboxing thus prevents access to memory using illegal addresses. Another technique known as segment matching can be used to identify and trap illegal references. Segment matching code, requiring approximately four instructions, is inserted before each unsafe

instruction. A trap may be generated if the segment identifier does not match the unique number of upper bits of the target address.

Support for data sharing within a sandboxing and/or segment matching scheme cannot be provided through manipulation of read/write or other state bits within page table entries. Conventional hardware schemes implement protection by placing each process, extension or untrusted module into its own address space. Hence shared data will have its own page table entry in every address space to which it is mapped, thereby allowing different protection rights to be associated with the individual mappings. The untrusted modules, in multiple address spaces, co-operate in the following way. A context switch occurs whenever an address in another process space is referenced (e.g. when a print server thread is invoked). The appropriate page tables for the new process (thread) are then made current and the access rights to the shared data for the new protection space are then enforced.

In Wahbe's [1993] software based scheme, the hardware page tables are modified in order to map shared data to the same segment offset of every protection space which participates in the sharing within the single process address space. This implies that the number of TLB entries will be increased for shared data with implementation of this method. TLB entries are a scarce resource which means that this is a highly undesirable side effect. This problem will also appear again within the discussion of customizing operating systems where user-level servers' target addresses pollute the TLB.

Another overhead is introduced in Wahbe's [1993] scheme in the requirement for translation of the unique shared addresses to addresses within the protection space of the currently executing thread. The steps for the translation are identical to that of hardware translation from virtual to physical addresses, except that it is being done through software. Thus the provision for sharing complicates the overall management of virtual addresses.

## **2.2.2 Hardware-Based Schemes for Virtual Memory**

### **Protection**

The next major method to enforce and utilize protection on virtual addresses is demonstrated in the PA-RISC virtual memory model [Wilkes 1992] and in the Protection Lookaside Buffer (PLB) model [Koldinger 1992]. Both the PA-RISC and PLB protection schemes use the concept of a protection domain to distinguish among process spaces in a 64-bit single space virtual memory system. Note that all addresses are globally addressable in the single space virtual memory system. Protection in this environment is provided not

through conventional address space boundaries, since they do not exist, but through protection domains that dictate which pages of a global address space a process can reference [Koldinger 1992]. Hardware protection occurs on lightweight context switches where the new protection domain identifier is loaded in a special register, and legal virtual memory references are recognized by matching the protection domain tag of the address space reference to the register's contents. This level of protection does not yet describe the type of allowed access (e.g. whether the access can be read-only or write or no-access etc.). Traditional hardware schemes maintain read/write bit information in the per page entry in the TLB.

### 2.2.2.1 The PA-RISC Design

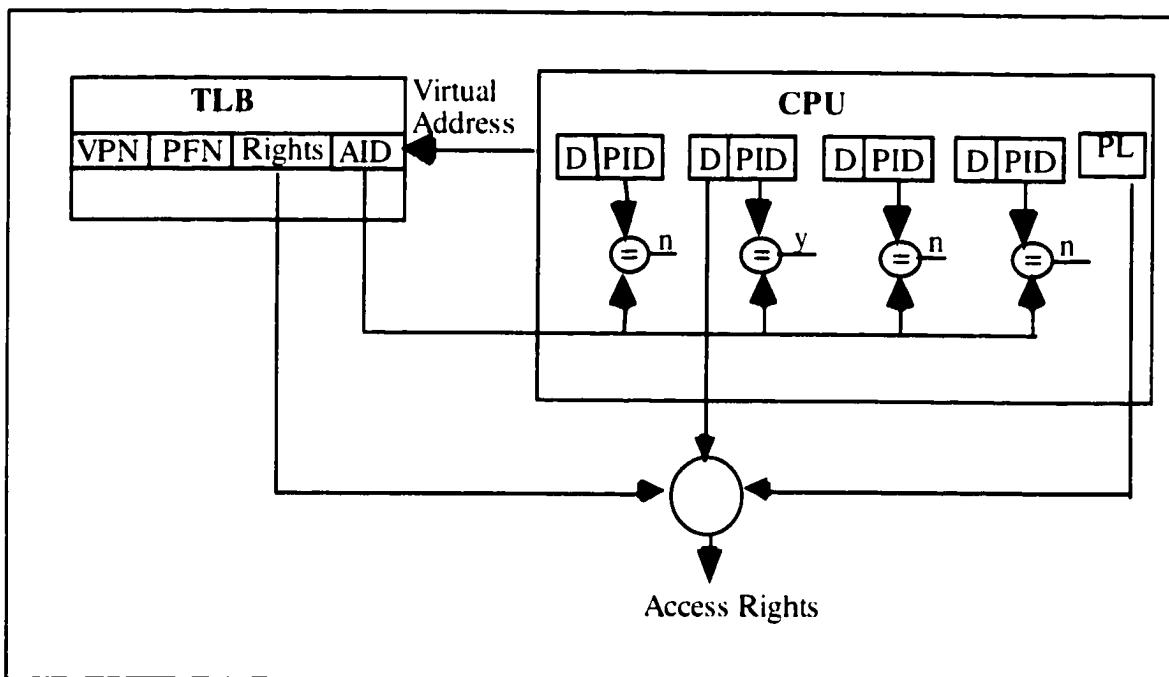
The PA-RISC scheme defines the protection domain to be the set of page-groups that an executing entity may access. Each virtual page is uniquely assigned to exactly one page-group by associating an Access Identifier (AID) with the page. The PA-RISC design extends the page table entries with Access Identifiers which, in turn, are matched to the contents of 4 special registers, containing Page-Groups Identifiers (PIDs), to determine access rights privileges. Access is allowed on an exact match of the PID within one of the registers and the AID stored in the TLB entry for the page. Access rights to a page may be altered by setting different Privilege Levels (PLs) while maintaining the same PID-AID pair. Thus only *one set of access rights* need be maintained on a per page basis. Note that the Privilege Level can only distinguish multiple access rights among page groups, not at the page granule level. That is, a PL change affects all pages within a page group. If the requirement is merely to change privileges for one page, then the page must be moved to another page group: an expensive operation since memory accesses will be involved. This restriction penalizes the PA-RISC support for sharing when synchronization is required on a per page basis.

Architectural support for the PA-RISC virtual memory system consists of a first level virtually addressed, physically tagged direct mapped cache (VAC) and a TLB where the latter contains the page's AID as well as translation mappings. When a memory reference is issued, the VAC is indexed with the lower order bits of the virtual address. Concurrently with the indexing of the cache, the TLB is accessed in order to obtain the physical frame number for comparison with the VAC's tag and also to obtain protection

information. The VAC must wait until the TLB access is complete before the matching of the tags step can start.

There are four control registers which hold the PIDs in the current PA-RISC architecture. This means that the process has access to at most four page groups at a time without generating a PID control register fault. A register fault takes approximately the same time as a TLB miss.

If the PID stored in one of the control registers matches the AID stored in the TLB entry (denoted by "y" in Figure 2.2; an "n" represents no match), then the access rights obtained from the TLB entry are modified with the Write disable bit (D) and the PL to obtain the exact access rights of the accessing entity to the memory location.



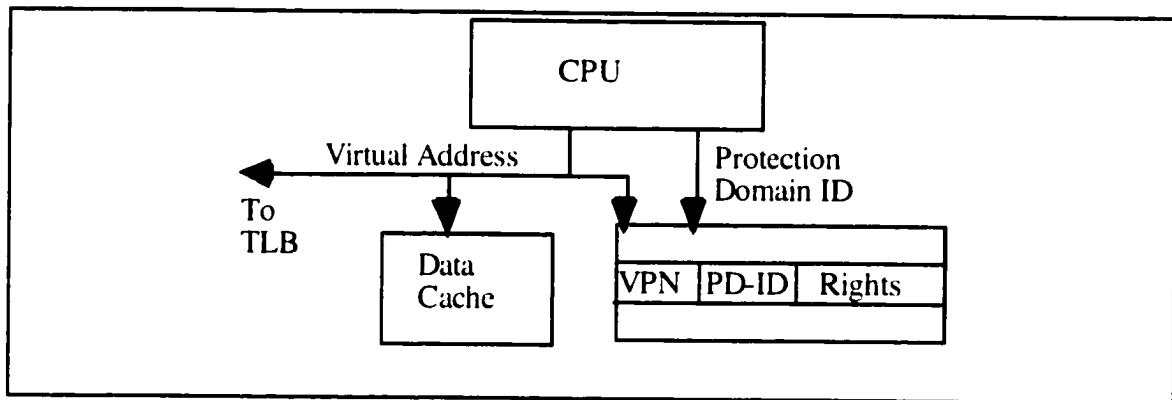
**Figure 2-2: PA-RISC Protection Design**

### 2.2.2.2. The Protection Lookaside Buffer Design

Koldinger [1992] proposes the use of a specialized cache called a Protection Lookaside Buffer (PLB) which contains protection information on pages on a per protection domain basis. The PLB model defines the protection domain as the entity defining the pages that can be accessed by the current execution [Koldinger 1991]. The threads associated with

different Protection Domain Identifiers (PDIDs) can have different access rights to the same page.

The scheme is based on a capability-style model where a process (analogously a task or thread) is given a list of pages that it may access. The PLB design supports a complete separation of protection from address translation. Koldinger proposes that the PLB and the data cache be situated on chip, and the TLB relocated off chip. This can be done since both the data cache and the PLB are virtually addressed and virtually tagged caches. However this means that additional space is required to store the wider tag information. Relocating the TLB off chip also means that the address translation cycle is slowed down: on cache misses, line replacements will generate a second access to the off-chip TLB. The PLB architecture is illustrated in Figure 2.3.



**Figure 2-3: The PLB Design**

In contrast to the PA-RISC scheme, the PLB maintains multiple entries in the PLB for each page in order for each to represent a different set of access rights to the page. In the PLB case, the entries are differentiated by the use of different PDIDs for the same virtual address. The index for the PLB cache is a concatenation of the PDID and the virtual address. Since the data cache entries do not contain protection information, *each* data cache access generates a PLB cache access.

### **2.2.2.3 Sharing Within The PA-RISC And PLB Designs**

Sharing is accomplished on the PA-RISC through the use of common PIDs. Thus the smallest granularity of sharing is the page-group. There are two sharing situations which generate complicated operations in the HP model: when the unit of sharing is a subset of a page-group and when a subset of a page-group's rights are to be modified. These two

issues are not the same, although the second can be the outcome of the first. Consider the first situation; it may arise if two or more processes require access to pages in a page-group such that for security reasons, one process is to be permitted access to only a subset of the pages in the page-group. The second scenario may emerge even if the entire page-group is to be shared on a long-term basis. Here regions of the page-group may be set as inaccessible or accessible throughout the processes' lifetimes.

If a domain should require the sharing of only a subset of pages of one page-group, the page-group will have to be split and the PIDs (re)assigned to the emergent page-groups. Cascades may be generated in that a change of PID through memory will also be necessary for all processes sharing the page-group. Loading of new PIDs and the purging of page-group PIDs from the page-group cache are involved.

When a page's access rights are to be modified such that the resultant access rights are distinct from those of the associated page-group then the page will have to be dissociated with that page group. The page is placed into another page group which has different access rights associated with it. This is an example case where support of the various PL levels is not sufficient to differentiate among multiple access rights.

Sharing can be done in two ways within the *PLB* scheme. One way is to share the same protection domain identifier (PDID) and the same virtual addresses. Processes have exactly the same access rights to shared pages. The second way to accomplish sharing is through the use of a common virtual address only; the PDIDs are different, leading to different simultaneous access rights to the same set of pages.

There is potential difficulty when PDIDs are shared by multiple processes since attachment or detachment of segments within a protection domain affects all sharing processes. It may not be desirable for all processes to share attaching segments, and similarly all processes that share through the same PDID will be affected by a detaching segment. Redefinition of the protection domains will result in some cases. This may be costly in terms of the (re)assignment of PDIDs in memory and for cache entries. The identification of the relevant PDIDs to be changed in the cache, on its own is an expensive task.

If only one domain has access rights to a page then a sole change is made to a *PLB* entry. If multiple domains share the same page, and access to all but one copy of the page should be the same, every *PLB* entry for that page would have to be updated; additionally any stored information for the PDID/virtual page pair in memory would need to be updated too.

The PLB scheme is the more flexible of the two in terms of providing for simultaneously different access rights to the same memory unit. The PLB model presents an opportunity for the support of the access units sizes which are different from that defined by the system for a page, that is, the access units can be variable-sized. This arises because the protection mechanism is separate from address translation. However the architecture defined by Koldingier is for system page sized protection units. Koldingier [1991] contends that fine grained protection should be provided by the language and compiler.

The PA-RISC model differs from the PLB model in that unique rights are associated with each page which implies only one protection entry is required per page. In contrast, explicitly different rights may be associated with each page in the PLB model thereby necessitating multiple protection entries for a page. However sharing is much less complicated and less expensive within the PLB model.

## **2.3 Variable Sized Protection Units**

The disadvantages of using a page of fixed size as the control unit for various purposes in a virtual memory system are numerous. This statement will be exemplified, in the following subsections 2.3.1 to 2.3.3, by detailed consideration of the following three areas: address translation, concurrency, and coherence.

### **2.3.1 Address Translation: Variable Sized System Pages**

The common virtual memory page size in modern operating systems (e.g. Windows NT) and architectures is 4 Kb. The page sizes are likely to grow larger as processor speeds improve and physical memories and applications' working sets grow. The working sets of sequential applications tend to be large which would imply that such applications will benefit from a larger page size. The working sets of random access applications tend to be much smaller leading to a bias toward a smaller system page size. A large page size can cause more unused program to be in memory than a smaller page size for some applications [Tanenbaum 1987]. Most systems' costs depend more strongly on the number of pages in a region than on the number of bytes in it [Fitzgerald 1986]. Small page sizes can cause more remapping of physical memory and more faulting operations than would

occur with larger pages, and thus lowers the effectiveness of address translation caches by reducing the size of the address range covered by a single cache entry. The small page size can also increase the costs of kernel data structures such as an Inverted Page table, whose size is the number of physical pages in the system. Small disk page size often implies large overhead to transfer a large amount of data. However some programs are tuned to a smaller page size and get more page faults with a larger page size due to internal fragmentation of real storage. Thus the appropriate page choice for applications is an important decision.

### **2.3.1.1 Simultaneous Support of Two Page Sizes**

Talluri et al [1992] examine the support of two page sizes within an address space and report on the increased complexity of architectural structures (e.g. TLB) and OS data structures (e.g. page tables), and other side effects introduced with the support of a second page size. The indexing of TLBs using a fixed number of bits, in prearranged positions (e.g. the lower order bits of the virtual frame number), no longer suffices. The performance of a TLB design is greatly affected by the efficiency of its miss handling. Supporting multiple page sizes complicates TLB miss handling. Talluri [1991] reports a 25% increase in execution time in the miss handling routine [Slater 1991] for a TLB which supports two page sizes. Address translation mappings must be aligned in both virtual and physical memory so that concatenation, rather than addition, can be used in the address translation process. Finally external fragmentation becomes a problem once more since page sizes are no longer fixed.

A series of experiments were carried out on applications from the SPEC benchmark suite to determine the effect on TLB performance by supporting either a 4Kb or 32 Kb page size within a single page size system, and also, for a system which simultaneously supports the two page sizes. The large page size is expected to increase TLB effectiveness due to the fact that the pages translated through TLB entries span a larger portion of memory. However results show that the simultaneous support of the two page sizes leads to a smaller increase (10%) in working set size than moving to a larger fixed page size where the working set increase was reported at 60%. It is surmised that the larger page size results in an increase in internal fragmentation. Talluri et al [1992] state that their results are not conclusive due to a lack of OS and multiprogramming behaviour within the



input address traces. The work represents a first effort in understanding the issues arising due to the support of multiple page sizes.

### **2.3.1.2. Architectural Support for Multiple Page Sizes: The MIPS R4000**

The MIPS R4000 [Mirapuri 1992] is a good example of an architecture designed with an understanding of applications' needs for variable sized pages and for multiple coherence protocols within the multiprocessor environment. The machine features an on-chip CPU, FPU, MMU, primary caches and secondary cache control logic. The MIPS R4000 supports seven page sizes (4Kb to 16Mb) using a 48 entry fully-associative TLB. Each entry in the TLB maps 2 consecutive pages. Each page size in the specified range must be a multiple of 4 Kb. A mask is stored within each TLB entry. These masks determine which bits of the virtual address and tag will participate in the comparison for determining a TLB hit. The data cache organization is direct mapped, virtually addressed, physically tagged, write-back, and supports a choice of 4-word or 8 word cache line.

A coherency attribute is stored with each page entry within the TLB. Each page can be marked as either uncached, noncoherent, coherent exclusive, coherent-write exclusive, or coherent -write update. Thus the processor supports write-invalidate and write-update protocols on a per page basis.

The R4000 supports a pair of instructions (load linked, store conditional) which work in conjunction with the cache coherency scheme to provide synchronization between processors on the system bus. The store conditional instruction fails if the location has been updated or invalidated since the preceding load linked instruction. The mechanism can implement bit-locks, semaphores and indivisible fetch-and-add instructions among other synchronization mechanisms.

### **2.3.1.3 Elastic Page Allocation**

Kagimasa [1991] proposed a multi-size paging architecture with elastic page allocation (EPA) to provide a solution to efficient storage management for very large virtual/real storage systems. In the EPA scheme the virtual address space and the physical address space are divided into two regions - one for large pages and one for small pages. EPA uses several methods to allocate real pages to virtual pages with different page sizes; all methods involve a lookup of either the System Available Page Area (SAPA) defined in real

memory for large or small pages, or a Local Available Page Area (LAPA), depending on whether the requested virtual page size is small or large. Virtual storage management overhead was measured in terms of the page faulting procedure while real storage management overhead was measured by page stealing and page measurement. Page measurement is the CPU overhead required to maintain unreferenced indexes of real pages for page replacement algorithms.

The experimental system supported four sizes of large virtual/real page: 16, 64, 256 and 1024 kilobytes, and the small page region of virtual/real storage is partitioned into 4 Kb pages. The Elastic Page allocation procedure was tested with Gaussian Elimination, Random Probe, Quicksort and Relational Database Retrieval Algorithms. In one scenario the virtual page size was set at 4Kb and the real page size at 64Kb and it was found that EPA dramatically reduced the CPU overhead. Then the effect on performance by varying the virtual page size was examined. It was found that the performance of Quicksort improved for larger page sizes, the performance of the random probe procedure improved for smaller page sizes, and that the virtual page size influenced performance more than the real page size.

#### **2.3.1.4 A Historical Perspective**

Randell [1969] proposed a partitioned segmentation scheme where different page sizes were supported. He investigated the phenomenon of storage fragmentation and found via a series of simulation experiments that using fixed sized pages results in more loss of storage, than allowing a number of different sizes of blocks to coexist in storage. That is, the increased internal fragmentation outweighed the decreased external fragmentation.

#### **2.3.2 Concurrency Control: Variable Sized Locking Units**

Although many applications require locking of data units in virtual memory, locking is generally implemented by applications themselves instead of using OS facilities. One of the reasons is that locking fixed-size pages, for the purposes of concurrency control, leads to the problem of unintentional locking (also known as false sharing) where other units are locked on a page besides the targeted unit. For example, in a system where the unit of locking is the same as the unit of address translation, say a page, if only one record on a

page were to be updated, then all other records would be locked also. This is unacceptable to many applications that access "hot spot" data, such as frequently accessed catalog or index information, because concurrency and hence performance are unnecessarily penalized. It is difficult to avoid this problem by data layout techniques.

### **2.3.2.1 Page Level Locking**

Stonebraker [1984] described a hardware scheme which supports OS virtual memory transaction management at a fixed-size page level. Concurrency control and logging for recovery purposes were both done at the page level. A known disadvantage of having fixed-size page level locks is that they are inefficient for B+-Tree index pages in instances where an index is split or underflow occurs. It has been shown [Kumar 1989] that Data Base Management Systems (DBMS) Transaction Managers (TMs) outperform TM services provided by an OS, because the OS provides control only on a page-level basis, whereas the DBMS's TM can provide control at finer granularities. Thus the OS is not generally used by the Transaction management system for locking services.

The hardware proposal [Stonebraker 1984] assumes that locking is the concurrency control method and that direct update is the technique used for the recovery procedure. Several implications of binding a file into a user's address space led to concern over consistent updates and the addressability of data. The binding scheme makes the entire file addressable by the client, thus implying a file lock. If the file is in a shared segment, a breach in concurrency control can occur since several clients can access the file without the intervention of the operating system. The problem is handled by providing addressability only after a page lock is obtained. The latter requirement is expensive, and in an effort to reduce it the lock manager is run in the user space and the lock table can be accessed from a shared segment.

The following fixed-page size level locking scheme [Stonebraker 1984] is included in this literature survey to show that the OS supported software and assistance provided by hardware for transaction management is a practical alternative to current software-only DBMS algorithms that run in user space. With the inclusion of variable unit sizes it should become a competitive alternative.

To cater for the page locking and addressability issues described above, four bits and a count field are associated with each page, acting as part of the address translation hardware. The title and use of each bit when set is shown below.

don't care bit ...	1 - activate hardware checking
write-lock bit...	1 - presence of write lock
access bit ...	1 - page has been read
update bit ...	1 - page has been written to
count field ...	value = # of readers who have referenced the page

When a transaction begins, the lock manager assigns two newly initialized bit vectors to it. One vector represents access bit values and the other write bit values; the bit values are on a per page basis. On the first read access to the page, the page's access bit is set and the count field is updated. The transaction accessing the page also sets its access bit for the page in the appropriate vector. On write access, the page's write bit is set and the transaction's write vector is updated.

When an interrupt occurs control is passed to the lock manager which saves the two vectors and the associated process id at each task switch. The lock manager is also responsible for loading the memory management hardware with the vectors of the new process. On commitment of a transaction, the lock manager first forces all the pages with a write bit of one to disk and resets the write bit to zero. The count field is decremented for each page that had its access bit set to one. Once the transaction is committed, the vectors can be deallocated and destroyed.

One drawback to this hardware proposal is the absence of fairness provision. A transaction wishing to write to a heavily accessed page may wait indefinitely [Stonebraker 1984]. Another hardware bit can be implemented along with the algorithm given to avoid the situation. However, the main drawback is that the granularity of locking is fixed at the page level.

### **2.3.2.2 Sub-Page Level Locking**

Chang [1988] reports on one of the first complete attempts to incorporate transaction management functions in a MMU unit within the IBM's 801 storage architecture. Described is a hardware locking mechanism which monitors the read and write references of individual transactions to 128 byte lines of storage. A page segmentation scheme was used; the page size was fixed at 2Kb. This scheme provides finer granularity locking and logging than Stonebraker's[1984] scheme. Also only a transaction id register must be changed on a process switch. As was already stated above, there are several reasons for

the specialized hardware. The 801 handles the problem by utilizing the line (128 bytes), which has a different size than the page as the unit of locking.

There are other reasons, however, for the specialized hardware. The hardware locks a unit for a transaction and not for a thread or a task. Thus any thread working on behalf of the transaction can access that unit. Furthermore, the access is without software intervention. If a unit is not locked by another transaction in an incompatible mode, the unit is locked in hardware and the access continues without software intervention. The mechanism provides for shared read locks on lines and exclusive locks on whole pages. In effect, when a thread accesses the unit, the state of access is checked and, potentially, changed in hardware. If there is a lock fault, that is the locks in the line are not for the current transaction, then software intervenes and searches a memory implemented lock table. The lock is granted if no conflict is determined, and is subsequently recorded in the lock table. The software (storage manager) then copies the current transaction's locks into the inverted page table entry.

Locking information is stored in two tables. One table is accessed by a hash of segment identifier and virtual page address. This lock table is used to determine lock conflicts and record locks granted and requested. The second table is accessed by the Transaction identifier and is used by the commit software to find all locks held by the committing transaction on all pages.

### **2.3.3 Coherence Control: Variable Sized Coherence Units**

In terms of coherence, the cost of I/O transfer for pages of a system-wide fixed size may be needlessly expensive. System designers are faced with the tradeoff between the requirement of a large page size for minimizing I/O transfer and a small page size for maximizing storage utilization, when determining the system's fixed page size. The issue of false sharing also arises here as in the concurrency control case [Bennett 1991, Dubnicki 1992]. Large coherence access unit sizes exploit processor and spatial locality, but cause false sharing. Small access units can reduce the number of invalidations or updates, but increases the bus or network traffic.

Dubnicki [1992] proposes a cache organization that dynamically adjusts the cache block size according to recently observed reference behaviour. A power-of-two buddy scheme is used to split and merge cache blocks across cache lines based on recent access patterns.

Each cache line is associated with size, LU (Lower/Upper bits) and split-merge counter fields. The size indicates the size of the data block in the cache line; the LU field contains bits for the upper half and lower half of the data block stored in a cache line; the split-merge counter records the number of processors that accessed only the lower or upper half of the cache block. The coherency protocol is responsible not only for the maintenance of coherence but must also distribute reference information and choose when to split and merge cache blocks.

Experiments were performed with input from C-Threads, the SPLASH suite and sample Presto applications. Results from the power-of-two buddy scheme showed that for every fixed block size, some programs suffer a 33% increase in average waiting time per reference and a factor of 2 increase in the average number of words transferred per reference, when compared to an adjustable block size cache. The Plus [Bisiani 1991] distributed shared memory system employs a hardware supported cache manager with a 4 Kb page as the unit of replication and a 32-bit word for its unit of memory access and coherence maintenance. Arguments for the required hardware support for coherence are similar to that of concurrency where synchronization is done on-the-fly with the recording of state transitions in hardware.

### **2.3.4 Problems Faced By OSs Due To Architecture-Provided Variable Page Sizes**

Current virtual memory (VM) systems are not suited to the support of multiple page sizes since the use of only one page size is assumed in most OS code. Adding OS support for multiple page sizes raises many new issues. Examples are how should physical memory be managed, how should common optimizations such as copy-on-write, read-ahead and clustering be implemented, and should virtual memory data structures be affected? A problem that is tied closely to the architectural design is that of modifying page table structures to support multiple page sizes.

The VM data structures will undoubtedly grow more complex. For example consider the situation when two mappings are made simultaneously to a page frame where one mapping is to a large page and the other to a small page. The traditional hash lists of page frame numbers kept and indexed on page frame identifier will need to be arranged and accessed differently. Also traditional page protection strategies will have to be modified.

Another problem that needs to be addressed is with regards to the *choosing* of a page size. Talluri [1993] considers the user application, the compiler and the virtual memory subsystem of the OS as candidates for the decision-making of what page size is suitable for a given mapping. He settles on the OS as being the most suitable since it maintains the information as to whether the system can satisfy requests for large mappings and can "suggest" a number of small mappings as a substitute. Therefore it is currently recommended that the notion of multiple page sizes should be exported to the user/compiler as a hint only [Talluri 1993].

Physical memory must also be managed on multiple page size architectures within the additional constraint that mappings must be aligned. The VM system will have to structure its "free" and "clean" page lists into several lists of the supported page sizes. A physical memory allocator must find a clean page of the required size when a mapping is requested. If a page of the appropriate size is not found, then the system may need to use smaller mappings or coalesce smaller pages into larger ones. Then another issue arises as to when and how should the allocator coalesce memory? Will it be worth it to copy memory around to create larger pages should physical memory become so fragmented that no large mapping requests can be accommodated? How will page replacement strategies be affected? Will one page size receive preferential treatment over other sizes? How is the page faulting rate affected by a system that manages free lists according to size?

In conclusion, the question arises as to whether the operating system should provide support for multiple page sizes in light of the additional complexity envisaged, or the hardware and operating system should be redesigned to support a single page size more efficiently. For reasons that will become clear as the reader progresses through this thesis, the author believes that the third option is the most straightforward, most beneficial and the simplest of the three given options.

## **2.4 Customizable Operating System Services**

Many researchers [Anderson 1992, Black 1993, Kiczales 1993] have arrived at the consensus that there is a need for providing customizable services to applications in order to improve their performance. To this end, some operating systems have been restructured into a group of servers communicating with each other via message passing (e.g. Mach 3.0, Chorus). Others define interfaces which expose low-level communication and

memory-system mechanisms to programmers and compilers [Reinhardt 1994, Bershad 1995]. The issues and problems of such schemes are presented below.

### **2.4.1 Microkernel Operating System Support**

Modern microkernels attempt to cater to the need for applications to be provided with the flexibility to choose their individual service protocols. This requirement has been manifested in the form of the movement of services such as paging out of the kernel and into user-level space, where, as an example, applications may choose among servers with FIFO, LRU or other replacement policies. The highly modular and thus flexible structure of microkernels, such as Mach 3.0, allows the prospective user to easily debug and modify code.

However, care must be taken with the writing of user level servers to prevent race conditions due to having more than one process looking after memory management and, also, to avoid untrusted intervention. Some operating system designers also consider Mach's upcalls to user level servers inelegant [Tanenbaum 1992] since it is in violation of the principle that services should only be provided by layers below. Changes to the bottom layers could then proliferate in changes to upper levels when upcalls are allowed. In addition, Mach 3.0 has not performed as well as the monolithic 2.5 version [Anderson 1991]. Because operating system calls are implemented by cross address space RPCs to operating system servers running at user level, each invocation of an OS service requires at least two system calls and two context switches to do the work of one system call in a monolithic system [Anderson 1991]. This effect is compounded on RISC architectures since the latter generate more instructions per OS primitive function in order to save pipeline and register window states.

Maeda [1992] has demonstrated that a Universal Data Packet (UDP) protocol user-level server implemented in Mach 3.0 gave comparable performance to an in-kernel implementation. It should be noted that the desired performance was obtained by the restructuring of the UDP protocol server with a smaller number of primitive locking operations and context switches.



### **2.4.2. Tempest And Typhoon: User Level Shared Memory**

Reinhardt's [1994] position is that the user-level software control of the shared address space provides more flexibility and improves performance over dedicated hardware support for coherence protocols.

In [Reinhardt 1994], a customized user-level software update-based protocol is shown to outperform (by +35%) a hardware Dir<sub>NB</sub>I invalidate-based protocol for a graph application running on parallel processors. This result does not separate the value of the customization from the gains in performance due to the fact that Reinhardt compared a should hardware invalidate-based protocol to an update-based software coherence protocol. It would have been fairer to compare an update-based hardware protocol to the update-based software protocol. But Reinhardt's study does point out that hardware protocols are inflexible and while they perform well for certain applications, flexibility to customize them is required so that they perform as well for other applications.

### **2.4.3. SPIN: An Extensible Microkernel for Application - Specific Operating System Services**

In SPIN [Bershad 1995], application-specific services are implemented with code sequences which are installed in the kernel at runtime: hence an extensible microkernel. The code sequences expose alternative interfaces to computer system resources. The operating system interface is defined by an actual programming language (instead of a set of primitives) through which applications can define and install new interfaces. Safety and performance factors are left to the language and its compiler. Type safety, object based methodology and explicit guards are used to limit the access of untrusted extensions, also known as spindles (SPIN's Dynamically Loaded Extensions). The performance is dependent on aggressive compiler technology: intraprocedural data flow analysis, symbolic evaluation, and inline expansion. Inline expansion of calls in spindles to kernel operations can result in direct data structure access and the use of the partial evaluation technique is claimed to reduce the cost of crossing from the spindle's execution domain to the kernel's.

#### **2.4.4 Aegis: Lowering the OS Interface**

Customizability is provided by an exokernel which exposes the machine primitives to the applications [Engler 1995]. The burden of use of these primitives lies directly with the programmers; they are responsible for the direct management of physical resources. No operating systems services which abstract the physical resources are used. This is to provide the maximum allowable flexibility in the customization of a system. Sandboxing is the method proposed to provide protection across boundaries [Wahbe 1993]. The disadvantages are that portability issues arise due to machine dependent implementations and that the programmer is burdened with formerly OS policy management and implementation.

#### **2.5 Cache Functionalities**

Two significant uses of caches are to improve throughput in (a) compute-bound systems where the speed of main memory is the limiting factor and in (b) multiprocessors to reduce demand on the bus bandwidth. However, caches have been used for many purposes, some of which are described below. The wide use of caches is examined here because caches will be used as a part of the architectural support for the memory model proposed in this thesis.

A common usage of a cache is for maintaining virtual to physical address translation mappings on Von Neumann machines, where it is known as a Translation Lookaside Buffer (TLB). The speedup of a system with an on-chip TLB as opposed to a system with no TLB is of several orders of magnitude [Hwang 1993]. Other traditional usages as data caches and instruction caches or unified caches are also widely found in computer systems' designs.

An early example of the use of dedicated cache is that of the recovery cache proposed in Lee [1980]. The recovery cache is used with programs whose code has been divided into recovery blocks. The recovery cache functions as follows. When a recovery block is entered, and an object is to be written to for the first time, the original object and its address is stored in the cache before it is written to. Backward error recovery merely involves restoring the values from the recovery cache.

An add-on recovery cache is used so as to be compatible with existing CPUs, memory and other peripheral devices. The kernel contains the routines to interface with the recovery cache and provides recovery services for the objects which cannot be cached in the recovery cache such as internal CPU registers. The kernel also generates the error log for the system on a peripheral device. New "instructions" are added to the CPU instruction set to interface with the recovery cache. As alterations to the CPU are to be avoided, the new instructions are obtained by making the recovery cache appear as a bus peripheral device which is controlled by writing to its status registers.

The recovery cache slows down the system in two ways: the time needed to interpret the recovery cache instructions and the potential interference of the memory cycles on the host system. Storing of data in the recovery cache needs to be done on read-modify-write and write cycles. However, typical applications show a much smaller percentage of writes than read memory accesses. An analysis of the performance gains of the recovery cache was not presented in the paper so we do not know how the hardware recovery cache would compare to other conventional recovery schemes.

## **2.6 Conclusions**

Section 2.1 reviewed existing synchronization mechanisms. The explicit mechanisms place the burden squarely on the application programmer for their invocations in all the right places. Even programmers experienced with working at the system level are prone to make errors in code that requires synchronization services. The programmers accustomed to working at the GUI level or at the application engine levels often have a difficult time with the correct invocation of synchronization primitives. For this reason, implicit and thus automatic synchronization is pursued in this work. The synchronization is transparent to the programmer.

Section 2.2 detailed existing state-of-the-art virtual memory models. It is efficient to provide protection and thus access control at this level. The common appearance of pipelines and floating point units in today's machines make the heavy duty operating system overheads such as context switching even more expensive. It may be desirable for OSs to provide support for the avoidance of context switching in as many situations as possible. Also, applications which result in the mapping of multiple service routines in a single address space are obtaining minimal or no support (e.g. Windows NT) from current OSs for protection space isolation.

From the review of variable sized system pages (section 2.3.1) it may be concluded that the operating system should either provide support for multiple page sizes, or the hardware should be redesigned to support a single page size more efficiently, or the operating system redesigned to support the single page size more efficiently. The unit of coherence control should no longer be tied to the size of the cache blocks nor the unit of concurrency control to the system page size. In general, the size of the protection unit for access control purposes should not be tied to the system page size.

Customizability of the services to applications is a desirable feature as shown in Section 2.4. Hardware solutions to provide access control have been provided by others [Stonebraker 1984, Chang 1988, Stenstrom 1990] but thus far they have been *inflexible*, in that all applications were forced to use a single protocol embedded in the hardware. Caches can be used to attain a *flexible hardware* implementation of access control policies and for storage of state information since their entries can be unloaded and reloaded. Section 2.5 showed examples of applications which used cache services for performance enhancements.

These factors motivate a single solution which would provide for efficient and automatic access control, address space isolation, synchronization at the thread level, access control to variable sized access units, application services customizability, and reduction in kernel-application context switching at no further cost to the TLB and primary cache systems.

# CHAPTER 3

## THE MULTIVIEW MEMORY MODEL

This and the subsequent chapters present a clean and elegant means of providing efficient and automatic access control, address space isolation, synchronization at the thread level, access control to variable sized access units, application services customizability, and reduction in kernel-application context switching at very minimal cost to the TLB and primary cache systems.

Section 3.1 presents the virtual memory model which supports the above features. Section 3.2 summarizes the contributions of the work presented in this chapter. Chapter 4 presents an architecture required to support the memory model.

### 3.1 The Multiview Memory Model

Many issues raised in Chapter 2 motivate the need for additional features to be incorporated in the virtual memory model for a computer system. The memory model is given the responsibility of providing features for purposes of increased efficiency and for the possible customization of access control protocols. I do not believe that the burden of providing synchronization should be placed squarely on the application programmer for several reasons:

- (1) mass duplication of effort will result across applications as occurred in database applications that were individually optimized by reimplementing OS-type routines instead of using the OS services.
- (2) lessening of the designers and programmers' productivity due to the need to address finer details;
- (3) individual programmer's efforts are more likely to be error-prone than well tested reusable services provided by the operating system.

The rest of this section synthesizes the features which are selected for incorporation into a virtual memory model. Reasons for the selections are also given.

Firstly, memory management should provide access control to memory units which may be of different size than the pages, the units of address translation. In this thesis, a more efficient way of utilizing a single system page size is advocated: variable sized

protection units are defined within or across pages. The option to provide virtual memory support for hardware-provided multiple page sizes is not pursued due to the numerous problems associated with such support (see Chapter 2, section 2.3.4).

Synchronization should be at the level of threads so that threads of the same task can have different access rights to access units within the address space of the task. This caters for isolation of extension code modules and their data areas within a single address space. Protection can thus be provided through lightweight context switches (see Chapter 2, Section 2.2).

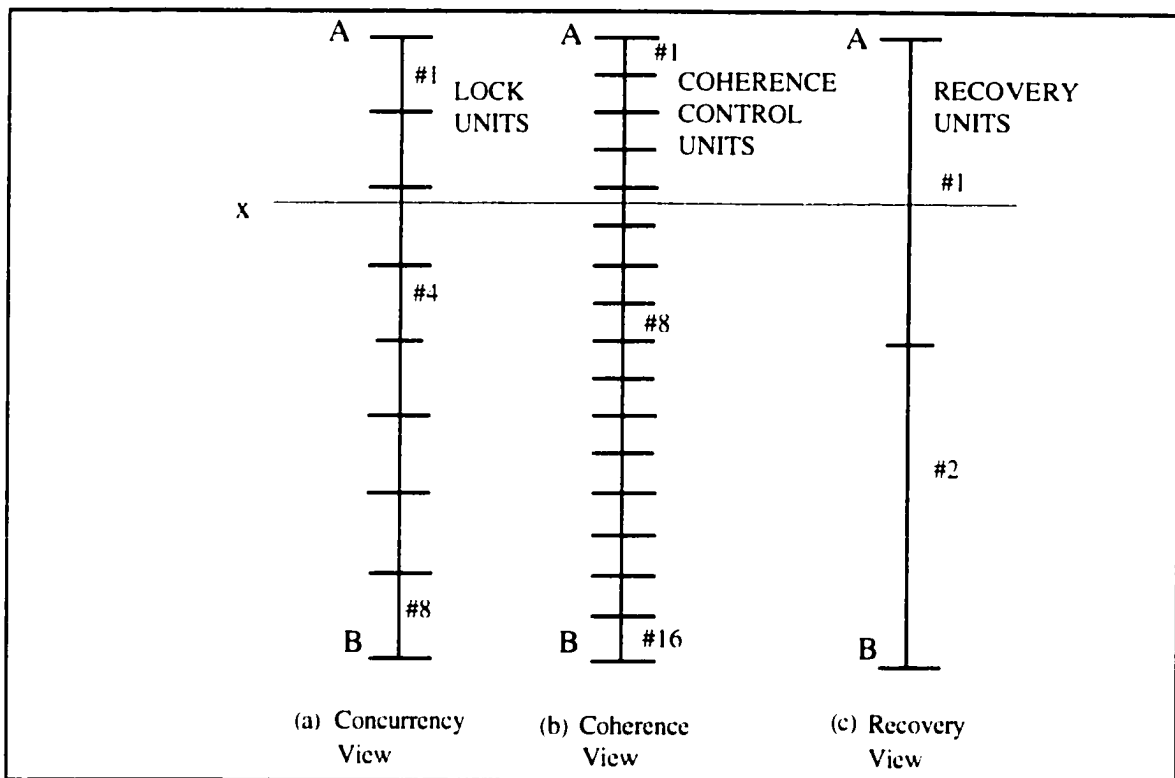
The memory management should provide, at least conceptually at the memory interface level, support to different memory access control protocols to the same region of memory. Multiple access control protocols defined on a memory unit must all be for different purposes. It is obvious why, for example, both a timestamping and locking method for the same purpose of concurrency control cannot be applied to the same memory unit. In a shared memory environment supporting transaction processing, it is desirable to apply several protocols on the same memory location. A byte can be part of an access unit on which a locking protocol should be applied. The same byte can be a part of a coherence access unit for the purposes of a coherence protocol. Even in a single processing environment, it may be useful to have the simultaneous application of both a checkpointing protocol and a locking protocol to the same memory unit.

In order to reduce the number of kernel calls, the access control should be on-the-fly; hardware support should be provided for not only checking the access rights but also for recording changes of state of access to the accessed data units. Furthermore, any supporting hardware mechanism should be flexible so that it could support various protocols to control access to data.

### **3.1.1 A Conceptual Representation of The Multiview Memory Model**

A conceptual representation of memory protection under the Multi-view memory model is given in Figure 3.1. Consider the memory region AB. A view may be imposed on a region of memory such that the region is logically divided up into equal sized memory access units and such that an association is made between the view and a single FSM definition of an access control policy. Several views may be imposed on memory such that each view can support a *different sized protection unit* and each view on the same region is

for a different access control purpose. For example, one view may be for concurrency control, another for coherence control and yet another for recovery purposes. A view can correspond to any memory control area which requires an access protection protocol. Hence different protocols may be applied for the same control purposes to virtual address spaces belonging to different applications within a single system. Each *access control unit*, or *access unit* for short, has the same size within a single view of the memory region AB, but the size can vary across views. An *access unit* is interchangeably referred to as a *protection unit* in this thesis. Applications can thus be serviced with various sizes of access units. A read or write memory access may only proceed if access is allowed in each of the views defined on the memory region.



**Figure 3-1 Multiple Views On An Address Space**

**EXAMPLE**

Consider a single memory location. It may "belong" to a number of access units, one access unit per view which may be defined for that location. An operation on that location causes the access control system to be invoked for each view, such that different policies are applied to the different views. In Figure 1, for instance, the memory location *x*

“belongs” to access unit #3 of the Concurrency view, access unit #5 of the Coherence view and access unit #1 of the Recovery view. For a read operation to be permitted on the memory location with the memory address  $x$ , the thread which issued the operation must have appropriate access rights in all three views.

Consider, for example, the Concurrency view and state of access to that view as represented by the standard access matrix  $A[S,O]$  as described in current Operating system texts [Singhal 1994, Nutt 1992]. Each row of the matrix corresponds to a subject. In a transaction processing environment each subject would correspond to a transaction. Threads of a task working on behalf of the same transaction would be bound to the same subject which represents that transaction. Each matrix element  $a[S,O]$  represents the state of access of the subject (transaction)  $S$  to the data item (locking access unit)  $O$ . Examples of state values might be “unlocked”, “locked by  $S$  in a shared mode”, “locked by  $S$  in an exclusive mode”, “locked in a shared mode by many transactions”, “locked by another transaction in an exclusive mode”. If a thread, bound to a subject  $S$  in the Concurrency view, issues a read of the memory location with the address  $x$ , the read may proceed only if the concurrency unit #3 is either “unlocked”, “locked by  $S$  in a shared mode” or “locked in a shared mode by other transactions”. If it is assumed that the memory read can proceed in all control views, then state transitions must be effected. For the Concurrency view this implies that the state of access to the control unit #3 must be changed, possibly for all subjects. If, for instance, the control unit #3 was originally unlocked, then, as a result of the read operation, the state of  $a[S,\#3]$  changes to “locked by  $S$ ”, while the state of access to the unit #3 by all other subjects changes to the state “locked in a shared mode by other transactions”.

The model avails itself of the regular address translation paging mechanism in order to avoid problems associated with variable sized system pages. Please note that there is a distinction between variable sized *system* pages and *variable sized protection units*. Variable sized protection units may be provided within regular system pages. For implementation efficiencies, restrictions are that views must be aligned (the starting view address should be on a power of two boundary), the views must be a multiple of the system page size and the protection units sizes within a page must be powers of two. The reasons for these efficiency constraints will become apparent in the following chapter. System page size is the page size defined by the particular operating system implementation.

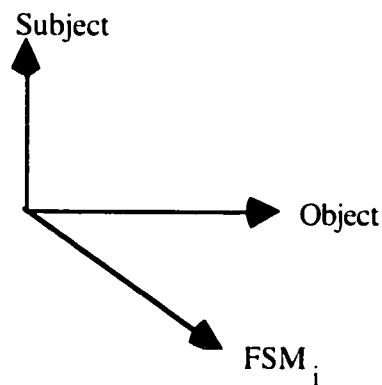


### 3.1.2 Flat Versus Segmented Address Space

In current RISC architectures, pure paged schemes for virtual memory management dominate the segmented counterparts due to an emphasis on the reduction of the cycle per instruction (CPI) count requirement, rather than on sharing and protection issues. In comparison to the flat paged address space, the segmented address space requires an additional level of lookup in the determination of physical addresses and it also requires explicit manipulation of the segment registers. It is for the same reasons that the multiview memory model also assumes a flat address space. Consequently, the following discussion assumes that virtual address spaces are flat and also that tasks, virtual address spaces and threads have the usual meanings: zero, one or more threads execute in an environment of a single task which has one virtual address space shared by all of the task's threads.

### 3.1.3 Access Control State Model

The standard access matrix to enforce access rights of subjects to objects has been adopted as a basis for the proposed memory model because it may represent access rights by Finite State Machines [Bodorik 1994a]. In the *access matrix* method [Nutt, 1992], each subject has a row while each object has a column. Each matrix entry  $A[S, O]$  is determined by a Finite State Machine (FSM) which defines the access rights of  $S$  to  $O$  depending on the current state and the desired operation/access. Since different FSMs used for various control purposes can define access rights on an access unit (an object), the access matrix can be conceptualized in three dimensions. That is, the same subject and object can have multiple FSMs associated with them.



More simply, however, the access state of a *single* view can be represented by an *access matrix*  $A$ . Access is in terms of memory access by the read, write and execute

operations. Objects are units of a view, i.e. they refer to data contained in memory locations and FSMs define the access rights for threads, or sets of threads which actually issue the memory operations. Subjects represent the executing entity such as a transaction or a thread working on behalf of a transaction or a procedure. More than one thread can associate to a single Subject ID. For example the subject ID can be associated with a transaction and the transaction can have several threads working on its behalf. If access control is defined with the transaction and not the individual thread then one subject ID would be associated with the transaction's threads. Synchronization can thus be provided at the level of threads or at the level of tasks, that is at *any granularity*.

The term "state" is used to mean "*access state*". Each view of a memory region has its own state which is independent from the states of the other views. Accessibility here implies accessibility "on-the-fly", i.e., without the operation causing a *fault* and possibly even without invoking kernel software. Consequently, for a memory operation to succeed, access must be permitted for each view defined on the referenced memory location. Once access is permitted for each view, the state changes independently for each one. Each view has a unique identifier (ViewID); views are unique across memory regions even if the same FSM identifier is associated with more than one view.

The concept of a *protection domain* is intrinsic to the multiview memory model. The protection domain defines the group of access units to which one or more subjects *may* have access; access is subject to further restrictions from the access control protocol(s) operating on the domain. A protection domain need not be one contiguous span of pages, unlike a view, but may consist of several groups of pages and/or a union of views. The protection domain is used to enforce fault isolation. A view can exist in several protection domains.

A subject is bound to exactly one protection domain. If a subject attempts to operate outside its protection domain, an illegal access will be captured.

### **3.1.4 Customizability**

Different FSMs, which can be defined on a single memory region or on different regions, satisfy the requirement of flexibility to support various protocols; a protocol is represented by an FSM definition. As long as FSM definitions, that is the outputs and rules for transitions are not hardcoded, defining an FSM to be used in determining the access rights defines a protocol.

The model is sufficiently flexible to support many protocols. For instance, various coherence protocols for distributed shared memories can be supported. More than one coherence handler can coexist in the system: one such handler may use an invalidation based policy for copy coherence while another may use a write-update method. Another example protocol is that of locking. Locking is performed on *concurrency control units*, not on address translation units. In addition to explicit manipulation of locks by invoking appropriate kernel primitives, granting of locks can also be initiated upon a transaction's first request for either read or write on a concurrency unit, if that unit is not locked by some other transaction. With appropriate hardware support this provides locking "on-the-fly", i.e., without delay due to software intervention.

Integration of protocols, such as for concurrence and coherence, is feasible within the model as well. Further discussion is deferred to section 3.2.

### 3.1.5 Kernel Primitives

This subsection deals with kernel primitives required to support the multiview memory model within a single address space (task), i.e., it deals with the creation of views, subjects and operations which are required to support general protocols on views.

Since access control is specified for protection domains while memory operations are issued by threads, a thread (subject) must be bound to (associated with) a protection domain. A thread is restricted to be bound to only one protection domain at any one time. Of course, many threads can be bound to a specific protection domain. The kernel provides the primitive needed to bind a thread to a protection domain. Multiple views can be associated with a protection domain. For a memory read/write/execute operation on a particular address to succeed, it must be permitted in each view defined for that location.

It should be noted that the definition of a view is such that it is defined on a region of consecutive memory locations. Because views are independent of each other, at least conceptually, there is no constraint on where these ranges are relative to each other within an address space. Sharing of views, however, does impose such constraints which will be discussed in the following section.

To support general protocols for views, there must be explicit operations which enable the forcing of the entries of the access matrix to a specified state. An example of an explicit operation can be found in transactional locking, when at the end of a transaction all locks acquired by the transaction must be released. Similarly, operations to examine the state or states of the access matrices must also be provided.

All of the following operations are provided as kernel functions; their implementation depends on the provided memory interface. To create a view within a task (address space) the following parameters need to be supplied: the size of access units and the definition of the FSM to be applied to the view. An FSM definition includes states, rules for transitions and a default initial state. This definition serves as a “template” to create instances of FSMs for the access matrix of the view. Minimally a view identifier is returned as a result of the invocation of the primitive to create a view.

When a subject is deleted, the operation may be rejected or a special action may be taken depending on the states of the whole row of the access matrix which corresponds to the subject. For instance, deletion of a subject may be rejected if the states in its access matrix row do not permit successful deletion of the subject, that is, when the states of the objects are such that the responsible subject cannot be deleted. Deletion of a view is constrained in a similar way.

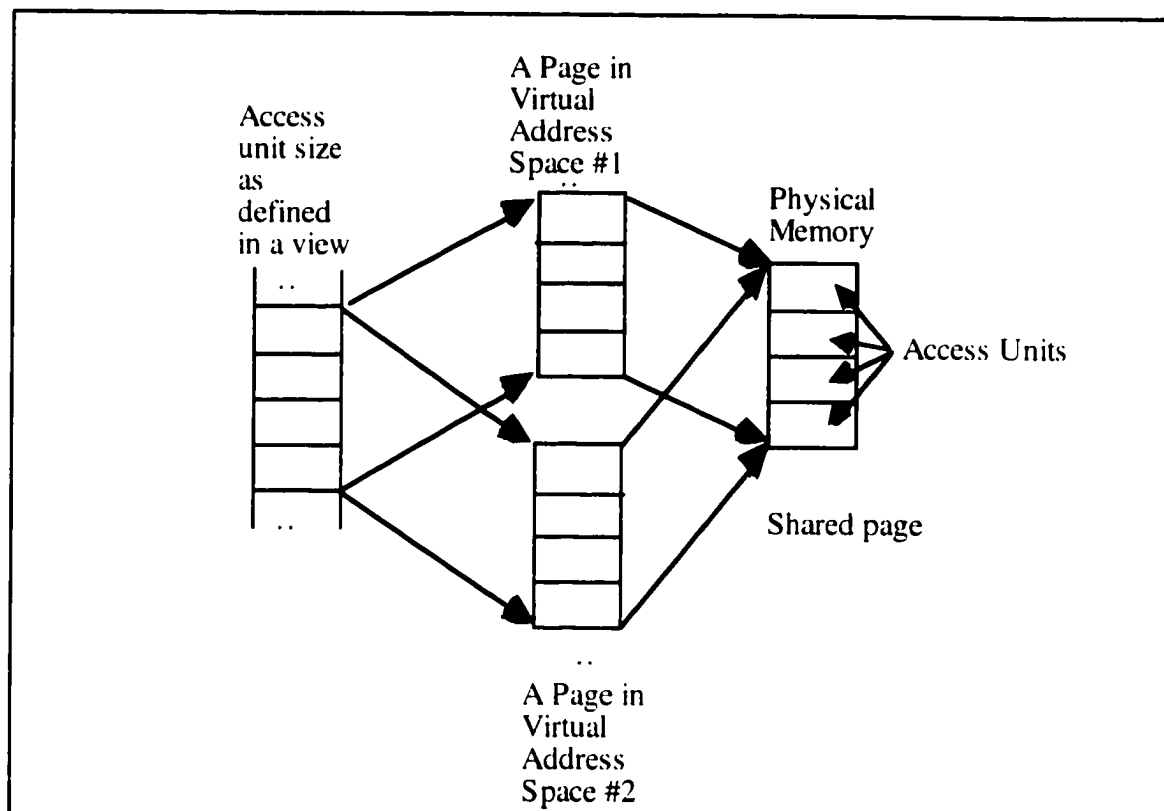
### **3.1.6 Sharing of Memory Views In A Single Processor System**

The multiview memory model supports present-day models of sharing: the traditional model where shared memory regions are mapped into multiple address spaces in a single system, and also sharing within a single address space.

#### **3.1.6.1 Sharing in Multiple Address Spaces**

To request the sharing of views among address spaces, parameters such as the view to be shared (e.g. viewID) and the starting address of the view in the local address space is passed to a kernel function responsible for the creation of the mapping of the appropriate view table entry. Figure 3.2 illustrates the mapping of a page in a region on which a view is defined to two address spaces and the sharing of a view. The view defines the access unit size for the page. Hence when the page is mapped into two or more different address spaces as shown in Figure 3.2, the view must also be shared in order to keep control information on the same units in both address spaces. The graduations shown on the view and the page in Figure 3.2 illustrate the access units and their sizes.

It should be noted that the memory model does not impose sharing of “all” views which may be defined on a region of memory. The kernel may, however, enforce sharing of all views defined on a memory region upon request of applications. The mapping information contains the size of the shared region, which of course must be the same for all of the address spaces sharing the region, and the starting address in each address space.



**Figure 3-2: Sharing Views in Multiple Address Spaces**

Sharing a view across tasks implies sharing the underlying mechanism enforcing access controls on the view, i.e., sharing the subjects, objects (access units) and also the FSMs which specify the access controls. This is necessary since shared protection units mapped to different protection spaces must comply to the same access control protocol for each purpose. That is, if locking is the concurrency control protocol defined on the shareable data within one address space, then the identical locking protocol must also be activated in any other region to which the data is mapped. Also the protection unit sizes cannot change across shared regions if inconsistency is to be avoided. Hence the view definitions are shared.

### 3.1.6.2 Address Space Isolation

In systems where the server runs in the *same* address space as the client, the Multiview memory model can enforce automatic address space isolation through the use of protection domains. Recall that the protection domain defines the group of access units which one or more subjects *may* access. To provide fault isolation, a view for that purpose can be set up on the memory regions(s) to be protected. The entire memory region, on which the view is defined, is considered the protection unit on which fault isolation will be enforced. A subject will incur an illegal access to a memory address if the fault isolation view for that memory region does not belong to its protection domain.

On an instruction or data cache miss, the requesting subject is matched to its protection domain. Consequently a check is made to ascertain that the address to be accessed is within a view in the protection domain. If it is not, the attempted illegal access will be captured.

Note that the multiview memory model allows for different access rights to exist for different threads operating within the same address space. Hence only one copy of a shared region need be present within an address space. This can be contrasted with Wahbe's [1993] proposal, presented in Chapter 2, Section 2.2.1.2, for sharing data where the data must be mapped into each participating protection space even if they are within the same address space.

## 3.2 Summary

The Multiview model supports *variable sized* access units through a novel *view* concept. Information on views is kept and can be used to trap illegal accesses thus providing for *address space isolation*. State information is conceptually maintained on the variable sized units through use of the access matrix method, thus providing for *fine grain synchronization* by the association of subject IDS to state information. Memory access triggers the access control mechanism which is implemented by an FSM, thus providing *automatic and implicit access control*. Different FSMs can be applied to different regions of memory within a single or multiple address spaces, thus allowing for the *customization* of access control services on a per application basis. Access control handling can be implemented by table lookups (through FSM definitions) without the use of software handlers thereby *removing additional TLB and data cache cost* inherent in software

implemented synchronization mechanisms. A *reduction in kernel calls* will be achieved since cross address space calls to synchronization software servers are reduced.

# CHAPTER 4

## ARCHITECTURE

This chapter presents a proposal for the architectural support required to implement the Multiview memory model. In the past, hardware support has been proposed to provide for memory synchronization (e.g. the IBM 801 system) but this support has been inflexible in that only one protocol is represented in the hardware. To provide maximum flexibility in the representation of multiple access control protocols and for enhanced performance, a cache based architecture is proposed in this thesis. The caches' content organizations are novel to kernel designs in terms of their functionalities. An important feature is that access control need **not** be invoked on each and every memory access. Other surveyed access control methods [Koldinger 1992, Wheeler 1992, Wilkes 1992] enforce protection on every memory access.

Section 4.1 briefly outlines the implementation options for the support of the model. Section 4.2 describes the chosen implementation for the protection architecture. Section 4.3 presents the alternatives for addressing a cache (Protection Lookaside Buffer cache) which contains information on variable sized memory units and the proposed schemes for PLB miss handling. Section 4.4 describes the entries within each of the caches. Section 4.5 discusses cache coherence issues associated with the protection architecture.

### 4.1 Implementation Options

There are many possible implementations of the protection control unit which supports the Multiview memory model. They can be classified broadly into software-only and hardware assisted implementations. A number of the proposed implementations are currently being investigated by others [Bodorik 1995]. Some are briefly outlined here. All implementations minimally require three tables: one to store view information, a table to store access control or state information and a table to store state transition (FSM) definitions.

These tables may be easily implemented in software along with access routines and routines for the logic defining the functions of the Protection Control Unit (PCU) which is



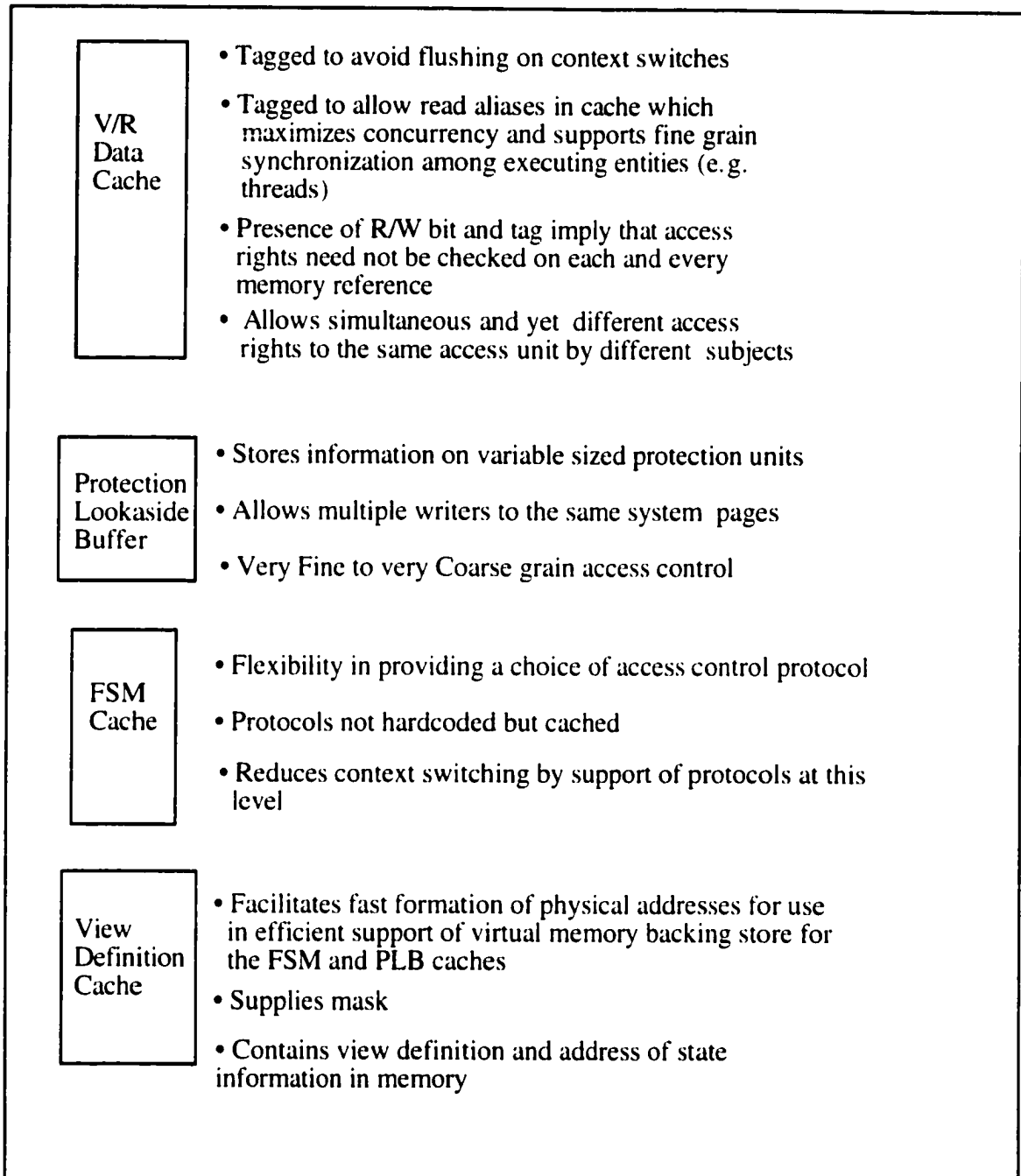
the entity that describes the support structure for the Multiview memory model. The software implementation would involve design decisions such as table organization (e.g. natural, ordered, inverted) and access methods (e.g. binary search, hashed).

Alternatively, a hardware controller can be implemented to retrieve information from the three tables stored in memory. The determination of where the required information is stored would involve simple operations such as shifting and comparisons. Simple logic would be used to decide whether the executing thread has sufficient access rights to accessed memory location.

Another implementation is the one pursued in this thesis, that of the use of specialized caches to contain the tables' contents. This implementation is expected to provide the highest performance with respect to the other two outlined above. Of course, it will also be the most expensive in terms of hardware cost. There are two reasons for pursuing this option in this thesis; one is that it is expected that the TLB will be the bottleneck [Homer 1995, Wulf 1995] in very near future computers, given the vast leaps in processor speed in the last year. The cache implementation of the protection architecture avoids excessively polluting the TLB with access control information, unlike any software scheme (either existing conventional or multiview memory model software implementations) which passes access control information through the data cache and will thus need translation information in the TLB. The table which stores the state information for the access units in memory is implemented in the virtual address space and only those translations that pertain to accesses to this table will affect the TLB entries. The second reason for use of specialized caches is that **flexible** hardware assistance for numerous protocols could be explored.

## 4.2 The Cache Protection Architecture

The cache-based protection architecture to support the Multiview memory model consists of three specialized caches: a Protection Lookaside Buffer (PLB) cache, a ViewDefinition cache and a cache labeled the FSM cache because it contains entries denoting state transition definitions. As a group these three caches and their controllers are referred to as the Protection Control Unit (PCU). The TLB entry is not affected in this architecture. In addition, the organization of the data cache allows simultaneous read synonyms to be mapped to it. The functionalities and characteristics of each cache are summarized in Figure 4.1.



**Figure 4-1 The Caches Functionalities**

The PCU is proposed to be off-chip, and is not on the critical path for data cache read hits, and for data cache write hits when the line's W bit is set. Note that the Protection

Control Unit is disabled if applications do not need views to be defined on them, thus not affecting the computer's normal CPI rating.

Figure 4.2 illustrates the flow of information among the three caches in the Protection Control Unit. The Multiview model assumes a virtually indexed, physically tagged set associative data cache which provides the first level of access control determination through its maintenance of a read/write bit. The first level of access rights determination is described in detail within Section 4.4.1.

A data cache hit upon a read request to an access unit allows read access to proceed automatically. That is, the PCU unit is not invoked. Write cache hits however will only proceed if the R/W bit is set. A *write access control fault* occurs when a write is requested and the R/W bit is not set. If a data cache miss or a write access control fault occurs then the PCU unit is invoked.

When the PCU unit is activated, the PLB and View definition caches are first accessed to obtain the current state of the access unit including the stored subjectID if any. The appropriate PLB entry is detected by addressing the PLB cache with the masked physical address of the access unit. This is to be discussed in detail in the following subsection. The physical address of the access unit is obtained from the TLB.

The information from the PLB is sent to the FSM cache which uses it to access the appropriate FSM state transition entry. The correct FSM entry is found upon a match of the requested operation (read or write), the result of the match of the requesting subject identifier with the subject identifier retrieved from the PLB and a match of current state retrieved from the PLB and that stored in the FSM entry. The result stored in the "Proceed/Fault" field of the FSM entry (see section 4.4.3 for details on the FSM entry description) indicates whether the subject has appropriate access rights. A successful memory operation may cause a state transition with respect to the accessed unit's current state. The "New State" to be transitioned to is stored in the FSM - the "New State" value is thus sent to the PLB to update its corresponding entry. The output of the PCU is a signal to the executing entity either to proceed, on a positive result, or to fault. The value of the PCU's output is that retrieved from the FSM entry's "Proceed/Fault" field.

The detailed description of each of the above described operations follows.

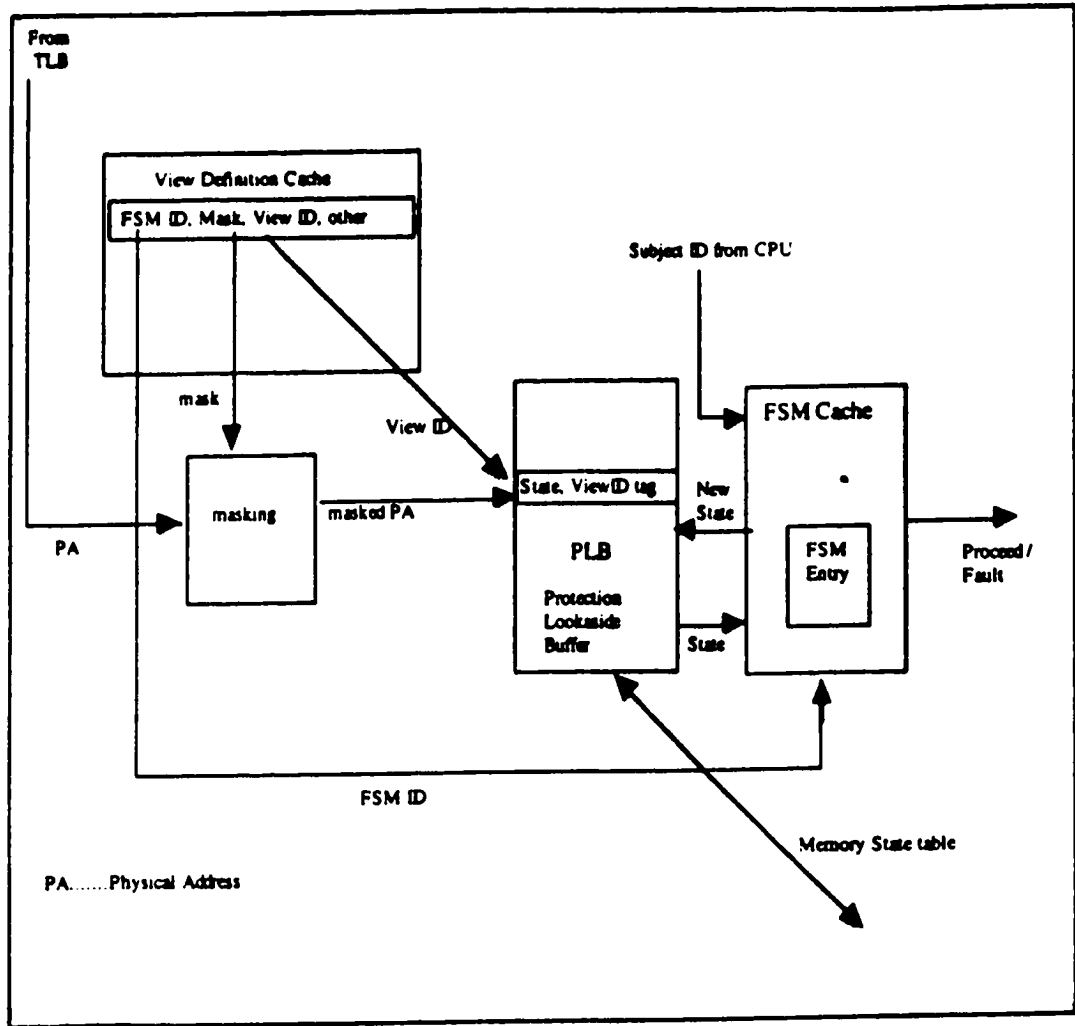


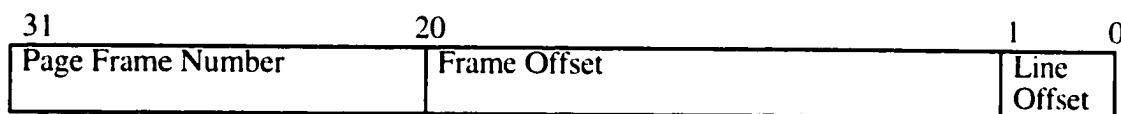
Figure 4-2 The PCU Unit

### 4.3 Variable Sized Protection Units and the Protection Lookaside Buffer

The problem with keeping state information on variable sized protection units is to efficiently find this information both in the PLB cache and in memory. This section addresses the locating of the information in the PLB cache. The range of virtual addresses within a protection unit must map to the same PLB entry. The number of virtual addresses in the protection unit is defined by its size. A bit mask may be used to mask out the  $\log_2(\text{size of protection unit})$  lowest bits within the offset to achieve mapping to a single cache entry. Hence a mask that represents the protection unit size is stored in the ViewDefinition entry for each view. The masked physical address represents the start address of the physical protection unit in memory. The constraints here are that the *access unit size is a power of two, and its size is less than or equal to the physical frame size*. Note that the location of the protection unit's *state* information in memory is quite different. Section 4.3 addresses the retrieval and storage of the state information in memory.

The PLB cache is physically indexed and physically tagged. The PLB is indexed using bits from a masked physical address. The choice of the set bits (that is, bits used for indexing the cache) has implications on the choice of a tag, the size of the cache, the degree of associativity of the cache and on the miss rate of the PLB. The implication on the latter only holds true for a masked address since it is possible for only a portion of the cache to be addressed with the masking action. The number of physical memory page frames to the number of cache sets ratio also has implications on the degree of associativity to be chosen for the cache. For organizations with large ratios, collision handling will benefit from a higher degree of associativity, since more addresses can be generated with identical set bits. For small ratios such as a factor of 2, 4, or 8 the set associativity can be small, i.e., 2 or 4-way set associativity has a good hit rate potential. The ratio cannot be considered alone, however, since the system's software reference pattern will determine which subset of addresses are generated most frequently. If the degree of associativity chosen by the designer is too large there may be unused entries in the cache.

Please refer to the following diagram (Figure 4.3) for explanation of the terms: line offset, frame offset and frame number. These terms will be used in considering the following alternatives for which bits should be used as the set bits for the indexing of the PLB. Assume for purposes of reading the diagram that only 8 bits are required as set bits to access the PLB cache. That is, assume that the number of sets to be indexed is 256.



**Figure 4-3 Bit Ordering in the Physical Address**

Little-endian ordering is applied to the bits within an address. The most significant bit (MSB) is numbered bit 31 and the least significant bit (LSB) is numbered 0. Recall that if the cache is to be byte addressable then the LSBs, (e.g. bits 0 through 7 if the line size is 128 bytes) will be assigned as the line offset.

There are several alternatives in choosing the index bits. As an example, bit range <28..21> may be used to index a cache with 256 sets. A cache with a high degree of associativity, e.g. 8, can be used to store entries whose addresses collide due to the identical <28..21> bit range settings. Thus there could be 8 entries (lines) within a set which may correspond to different access units on the same or different pages.

**Alternative 1:** If set bits are chosen from the masked frame offset part of the address, the PLB miss rate will increase since only a subset of the PLB cache entries will be accessed. For example, assume that the line size is 128 bytes, the access unit size is 2Kb and that the index bits are bits 19..9. Then masking of the address will result in zeroing bits 10..0. Bits 9 and 10 are part of the index bits. Therefore 2 bits of addressability are lost. Here we assume that the system page size is large enough that the offset bits within the frame are sufficient to address the PLB cache (i.e. the number of PLB lines is the equal or less than the number of bytes on a page) Since the set bits in Alternative 1 equate to the offset of an access unit within a page, then the tag in each PLB cache line must be the physical frame number component of the physical address. More collisions are likely to occur with this form of addressing in a cache when an application's read or write references are to many pages with the same access unit size. This may lead the designer to decide that a PLB cache with a larger degree of associativity may have a better hit rate due to a system's software mix that references many different pages where only a small number of differently sized units need be accessed. That is, reference may be made to many pages with access unit sizes within a small set, say {512, 1024 and 2048} bytes.

**Alternative 2:** The increased PLB miss rate due to the variable number of bits used to access the PLB cache may be resolved by right shifting, from the page frame component of the address, a number of bits equal to the number masked out from the index bits. For example, again assume that the line size is 128 bytes, the access unit size is 2Kb and that the index bits are bits 19..9. The masking of the address will result in zeroing bits 10..0. But right shifting two bits from the page frame component will allow the full 11 index bits to be utilized. The tag in each PLB cache line will be the physical frame number component of the physical address. A barrel shifter will need to be incorporated in the supporting hardware for the PLB cache. The number of set collisions here should be smaller here than in alternative 1, providing that the system's applications' reference patterns are the same.

**Alternative 3:** Use LSB bits from the page frame number component of the address as the set bits. That is, candidate bits are in the range 31..20 with respect to the above

diagram. There will still be more than one address resulting in the same index to the cache. To differentiate among them the tag must be the whole masked physical address. This means that the on-chip area of the cache is increased for a wider than normal tag. The tag usually consists of the PFN. With respect to 32-bit addresses, approximately 12 bits are used for the PFN. This option avoids the use of a barrel shifter and hence avoids increased delay due to shifting within the PCU unit. It also avoids the problem of using a variable number of bits to access the PLB cache.

Alternative 3 is chosen for implementation with the proposed cache-based design since it does not suffer from a higher PLB miss rate as in alternative 1 and it does not require additional hardware as in alternative 2.

#### 4.4 Miss Handling For The PLB Cache

Virtual memory is used as backing store for the PLB cache. The table that backs the information in the PLB cache is referred to as the State Storage table. One of the objectives of the design of the PCU is to locate state information in memory as quickly as possible. To do so a virtual address is formed which represents the address of the state storage table entry. This virtual address is sent to the TLB for translation. Figure 4.3 illustrates the mapping that is required, that is the logical connection between an access unit and its state storage entry in virtual memory.

The method of calculation requires a lookup to the ViewDefinitionCache. The PAddr attribute in the View Definition entry (described later in Section 4.4.4 ) is the virtual address of the start of the State Storage Table for a view. The start address of the State Storage Table must be combined with an offset to form the virtual address of the required state table entry. To calculate this offset and the resulting address of the State Storage Entry, the access unit number of the view must be determined and then modified by the size of the State Storage Entry.

The virtual address of the State Storage Entry is determined as follows:

(1) Right shift the original VA for the data by

$$\log_2 \left( \frac{\text{access unit size}}{\text{\# of bytes for State Storage Entry}} \right) \text{ bits}$$

(2) The  $\log_2$  (size of State Storage Entry) least significant bits must be zeroed.

(3) Next the remaining offset bits (after the shift) and the lower  $\log_2$  (# of frames in view) bits of the VFN are ORed with the virtual address of the start of the State Storage table.

For the above procedure to be correct, views must be aligned on locations that are powers of two such that addresses with their lower n bits zeroed can uniquely identify the

$2^n$  pages within a view. This is to avoid having to subtract the base address of the view from a virtual address in the process of forming the backing store address. All state storage unit sizes must be a power of two in order to facilitate the calculation of the above address.

The bits associated with the virtual page number are sent for translation to the TLB. Next the location in memory is accessed and the access unit state information is loaded in the PLB cache.

The following example illustrates the above process. Assume that a view is defined on a region consisting of 128 pages. Each access unit within the view has a size of 256 bytes and the PLB state storage unit is 8 bytes in size. Also assume that the system page size is 1Kb. Assume also that the virtual page number at the start of the view is 00000000000000001000000 and that the virtual page number at the start of the backing store region for the view is 0010000000000000000000. Let us say that the virtual address that caused the miss on the PLB cache is 00000000000000001111111010000000.

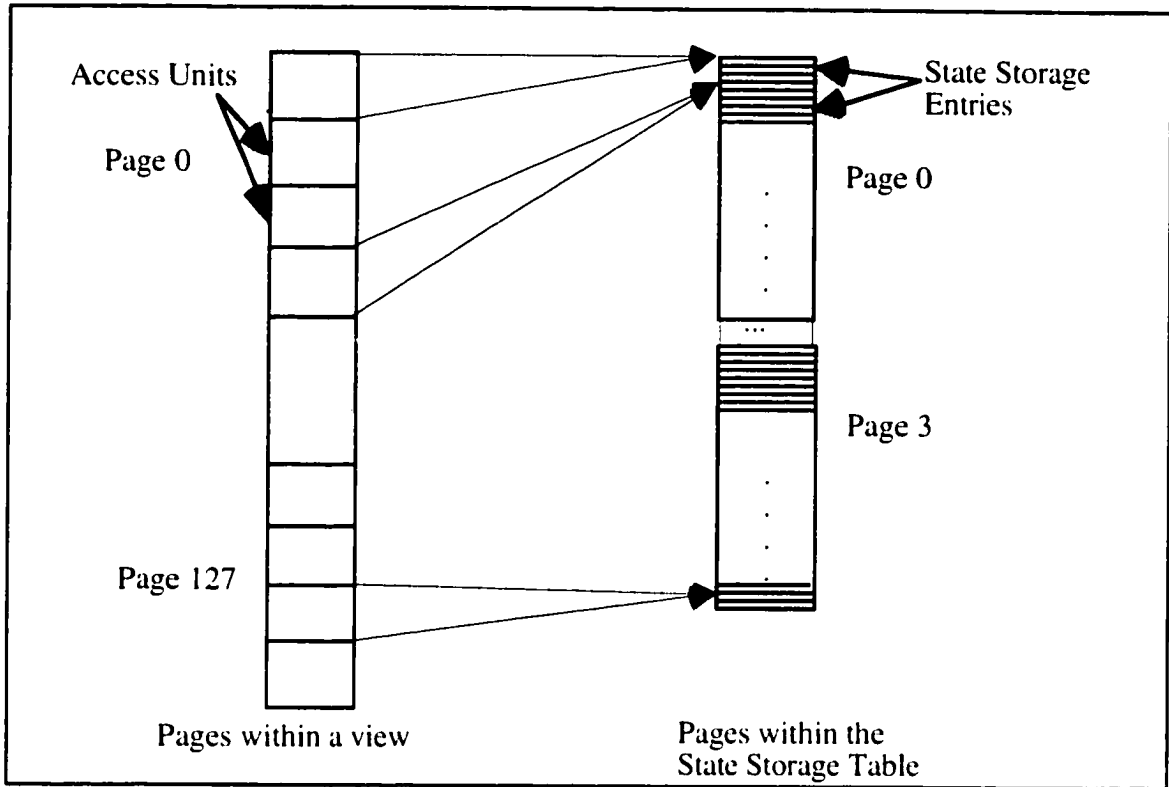
The first step (right shift by 5 bits) applied to the faulting address yields bits 00000000000000000000000011111110100. When the second step is applied (zeroing the 3 least significant bits), the VA becomes 000000000000000011111110000. The remaining offset bits are the least significant 5 bits; the lower  $\log(128)$  bits of the VFN are the 7 bits immediately left of the offset. Together they form 11111110000. This value (the leftmost bit positions in the 32-bit VA are zero-filled) is then ORed with 0010000000000000000000 to obtain 0010000000000000000011111110000.

Figure 4.4 can be used to illustrate the above calculation. The faulting address is in page 127 within the view. This can be easily verified by stripping the 10 offset bits from the faulting address given above (00000000000000001111111010000000 to 0000000000000000111111). Each page has 4 access units defined on it. Therefore 128 pages require  $(128 * 4 * \text{size of state storage entries}(8) = 4096)$  bytes or 4 pages of storage in the State storage Table. The faulting address belongs to the third access unit within page 127 of the view. Therefore the state storage entry for that access unit is in page 3 of the State Storage Table at byte offset 1008 (in decimal) from the start of the page.

*Note that there is no software handler invocation for the PLB miss handling procedure.* It is assumed that the ViewDefinition cache is large enough to contain all the views as defined on a system's virtual addresses. Also note that the TLB miss rate is affected, since access control information mappings are maintained by the TLB but that the control information (e.g. PLB and FSM entries) are not cached to the data cache, in order to avoid pollution of the data cache with control information. The number of mappings in the TLB



increases only by a small amount for this control information, since a system page can contain access control state information for a large number of virtual pages.



**Figure 4-4 The State Storage Entry Virtual Address**

## 4.5 Cache Entries

The tradeoffs among cache organizations and types, and the identification of what information is stored in each cache are presented in this section.

### 4.5.1 The Data Cache Entry

One of the objectives of the Multiview model is to provide fine grain synchronization among threads. Access control may be at a fine grain level within a task in that not all threads of one task may have identical rights to the whole address space. Subjects will have access rights to control units as defined for a view and threads must associate themselves with (bind to) the subjects to gain appropriate access rights. The architecture supporting the Multiview Memory Model provides fine grain synchronization among

subjects by tagging the data cache entries with a subjectID and by associating subject IDs with access unit state information. Copies of shared data items can appear as synonyms in the cache since each copy is accessed by a unique subject-VA address combination.

The first level of access control rights determination to an access unit is supplied by the data cache through its use of the read/write bit. For a memory access to proceed, first the subjectID (e.g. thread id) must match that found in the tag and then the memory operation type must match the status of the read/write bit in the case of a write. In the case of a read access, the R/W bit need not be checked since if a subject holds write access privileges to data, it is automatically assumed that it also has read rights. The setting of the R/W bit allows for automatic access on write hits following the first write access to cached data. If access is allowed to proceed on this level, this is the end of the access control determination procedure. However, if a *write access fault* occurs, due to the incompatibility of the memory operation with the R/W bit, or a miss occurs then access control determination is extended to another level. The protection control unit (PCU) to be discussed later must then be invoked.

#### Data Cache Entry

SubjectID	Physical tag	Data	R/W bit
-----------	--------------	------	---------

The tagging of the cache with SubjectIDs also solves the problem of homonyms, that is when two or more physical addresses map to the same virtual address. Recall that virtual addresses are reused across tasks. Hence the tags avoid having to utilize flushing of the cache on context switches to resolve homonyms. However, the support of fine grain synchronization among threads by use of a tag implies that synonyms, that is the mapping of more than one virtual addresses to a physical address are created. *In a virtually addressed cache, this situation occurs in any event.* The synonym resolution method has implications on the cache type chosen for the data cache implementation.

Two alternatives were identified for synonym resolution: one employs the reverse translation buffer (RTB) solution [Smith 1982] while the other imposes a view for access control on the cache. The former method benefits from a physically tagged cache, in that the tag can be used directly by the comparators to find all synonyms. This synonym resolution method implies that if the physical invalidation of the cache lines is to be performed in one cycle, then all lines must be identified during the time of one comparison, using the physical tag as key. This, in turn, implies either the use of a fully associative

cache or a set-associative cache where the mapping policy is such that all synonyms are mapped to the same set. More than one cache cycle will be required by our tagged data cache for synonym resolution via a view [Jutla 1995] or software. In these cases a synonym list may be maintained and a special Invalidate instruction may be issued to the data cache for each synonym.

### 4.5.2 The PLB Entry

Components of the PLB entry are given below. The Current State field maintains the present state of the access unit. The SubjectID field purpose changes according to the synchronization protocol for the view. For instance in a concurrency control view, the SubjectID field is used to store the ID of the subject which holds a write lock on the unit. For coherence control, the Subject ID field can be used to hold the identity of the owner of the cached object.

Recall that views are given unique identifiers (ViewIDs); they are unique across memory regions even if the same FSMIDs are associated with several views (see section 3.1.3). The ViewID is required for each PLB entry since a memory access unit can exist in several states if more than one protocol is defined on it.

PLB Entry

ViewID	Physical tag	Subject ID	Current State
--------	--------------	------------	---------------

### 4.5.3 The FSM Cache Entry

The Multiview memory model's support of multiple access control protocols is provided by storing protocols' state transitions in a structure such as the FSM entry illustrated below. In this implementation of the architecture for the Multiview model, the FSM entries are stored in an FSM cache. See Chapter 5 for the FSM entries for some important synchronization protocols.

FSM Entry

Memory Operation	Subject Match Bit	Current State	Proceed /Fault	New State
------------------	-------------------	---------------	----------------	-----------

The FSM controller receives the state of access to the referenced protection unit and a recorded subject ID from the PLB. The FSMID is supplied from the ViewDefinition cache (see the following section). The identifier for the currently executing subject is supplied by the CPU (executing thread). The subject match result is determined by the comparison between the SubjectID supplied by the CPU and the SubjectID retrieved from the PLB as part of the state of the protection unit. The subject match result is represented as 1 for a match, and 0 for a non-match. The Subject Match Bit content must be the same as the subject match result in the procedure for obtaining the correct FSM cache entry.

The FSM cache is accessed using an index formed from the FSMID, the current state of access retrieved from the PLB, the subject match result and the desired (read/write) access. Each cache entry defines a state transition (if any) and whether or not the desired cache access may proceed. It also specifies whether it is necessary to set the cache line state to reflect newly acquired access rights to the line. If there is a state transition, the new state (possibly with new SubjectID) must be stored in the PLB.

Thus support for multiple protocols is achieved by the loading and unloading to and from the FSM cache with the state transition tables that represent individual protocols. Essentially the protocol action is determined by a cache lookup given a particular FSMID, memory operation, subjectID Match result and current state of the access unit. A direct mapped cache is a candidate for implementation of the FSM cache since this cache may be indexed by a concatenation of FSMID, memory operation, current state and the subject match result. Consider a system which supports a mix of applications which require up to 8 different access control protocols. Assume that 2 memory operations and 16 current states are to be supported, then the index length will be 8 bits and a 256 byte direct mapped cache will suffice. In this case the mapping will be 1-1 and hence there will be no collisions unless the example maximum numbers given above increase. Of course, it is possible to use a lesser number of bits to index the cache, and use an even smaller cache size.

#### **4.5.4 ViewDefinition Cache Entry**

A specialized cache, the ViewDefinition cache is reserved for cache information pertaining to the definition of views. These entries are cached in order to effect PLB and

possibly FSM miss handling without a context switch. Entries within a ViewDefinition cache define the view imposed on a virtual memory range. A view defines the access unit sizes (through the Mask attribute) as well as an associated protocol (via the FSMID). The virtual address that signifies the start of the view is stored in the attribute VFNstart and the last address in the view is denoted by VFNend. The SID (Subject ID) entry represents the subject (task, transaction or thread) associated with the view.

A particular ViewDefinition entry is found by comparing the incoming virtual address to the VFN start and VFNend. To find the view information the virtual address must be larger than the VFNstart and smaller than VFNend.

ViewDefinition Cache Entry

ViewID	VFN start	VFN end	FSMID	Mask	Faddr	PAddr	VIDptr
--------	-----------	---------	-------	------	-------	-------	--------

VIDptr contains the next ViewID which was defined on the Virtual Memory

FAddr refers to the start of the FSM table in Virtual Memory

PAddr refers to the start of the State Storage Table (superset of the PLB entries) in memory

## 4.6 Conclusions

This chapter presents a design for the architecture which is required to support the Multiview memory model. The objectives of the model as listed in Chapter 1 are met in the following ways. Support for *variable sized protection units* is shown while still utilizing the single system page size. Locating state information on variable sized protection units is achieved through the use of a bit mask which masks out the  $\log_2(\text{size of protection unit})$  lowest bits within the address offset. A mask that represents the protection unit size is stored in the ViewDefinition entry for each view. The ViewDefinition cache maintains all management information on the views imposed on memory on a per address space basis. *Address space isolation* is achieved through the use of a view imposed on the region of memory to be protected.

The architecture provides for *fine grain synchronization* among subjects by tagging the data cache entries with a subjectID, by associating subject IDs with access unit state information and by supporting FSM definitions of protocols.

Access rights need not be checked on each and every memory access as is done in many other protection schemes [Koldinger 1992, Wilkes 1992]. The protection

architecture enables the invocation of access rights checking *only* on the *first* write access to the data item or on a data cache miss. For long write runs to the same data item this avoids unnecessary control activity. This is an advantage compared to other schemes (e.g. HP PA-RISC) which provide cache based access rights checking on each and every read/write memory access.

Greater read concurrency at the data cache level is achieved by supporting multiple concurrent read aliases. The protection architecture also includes efficient formation of physical memory addresses for backing store to the PLB cache through virtual memory.

Customizability of applications through the choice of an access control protocol per region of memory is supported by the Views (along with their associated FSMs) defined on the memory regions. Flexible architectural support for protocol information was described in the chapter. The protocols are not fixed into the hardware as in [Goodman 1987, Stenstrom 1990] but their FSM definitions may be loaded/unloaded from a specialized cache. Since hardware support is chosen for the architectural design of the Multiview memory model, pollution of the TLB and data caches is also reduced. Also software schemes will incur more kernel-user application communication or message exchange when compared to the cache supported design presented in this chapter.

# CHAPTER 5

## PROTOCOL SUPPORT

This chapter shows the model's *flexibility* in its support of different access control protocols. The protocols are defined per view. The idea is to maintain state information per access control protocol on each memory protection unit. Descriptions of a number of example protocols which may be decomposed and supported by the model are provided. First a virtual cache coherence protocol is examined in section 5.1. Next memory coherence protocols are considered in section 5.2. A further example is given in the decomposition of Mach's external pager algorithm in section 5.3. The two phase locking concurrency control protocol is yet another example. However, it is used as the example protocol for purposes of the model's evaluation within this thesis, and is fully presented in Chapter 6.

### 5.1 Virtual Cache Coherence Manager Protocol

The scheme described below is attributed to [Wheeler 1992]. It is a software implementation strategy for maintaining consistency between memory and virtually indexed caches, particularly write-back virtual caches. The scheme may be implemented on any memory whose content is cacheable. Four operations may cause the cache and memory entries to become inconsistent: CPU-read, CPU-write, DMA-read, DMA-write. CPU-read can cause a line with the same physical address to be mapped through different virtual addresses to two or more different lines in the cache. If a DMA-write occurs between loads to the cache, it is possible for at least one cache entry to be inconsistent. CPU-write can cause memory to become inconsistent with respect to the cache. DMA-read can read stale data in memory if the cached version is the up-to-date copy. DMA-write can cause cached data to become out-of-date. The control software is thus invoked on detection of any of the above operations. Virtual memory protection detects state transitions during every CPU read and write. Cache purge and cache flush operations rectify inconsistencies by eliminating copies from the cache. OS software must invoke the consistency control software before DMA operations are scheduled.

A cache line may be in any of four states: empty (E), present (P), dirty (D) or stale (S). The empty cache line does not contain data for the accessed virtual address. A present line contains the correct data. A dirty line is one where the data has been written to since it was cached. A stale cache line contains out-of-date data: the newest version of the data is either in another cache line or is in memory.

Note that the states are kept on cache pages in order to reduce the amount of stored state information. Wheeler defines a cache page as “the set of lines onto which the cache index function maps all virtual addresses within a virtual page. In the implementation evaluated by Wheeler [1992], the cache page size is the same as the physical page size. This scheme can also be used with hardware which supports multiple page sizes, with modifications to OS software whose purpose would be to ensure that a cache page is mapped only to similarly sized virtual pages.

The table below (Table 5.1) shows the state transitions upon memory operations as detailed in Wheeler [1992]. Note that a cache flush causes all flushed items to be written back to memory and invalidated in the cache; a cache purge causes the purged items to be invalidated in the cache. The purged items are not written back to memory.

The FSM definition, as would be implemented for the Multiview model for Wheeler’s scheme is as presented in Table 5.2. In the Multiview model, the object is the cache page. The requesting thread with its associated subject ID is the subject. The items in brackets (e.g. read in, flush, update) found in the New State column are external actions, mainly occurring on a fault, before the change of state for the object occurs. For example, when a CPU-read is issued, if the issuing subject does not match the current owner of the cache line and the current state of the cache page to which the line belongs to is Dirty (D), a fault occurs. The cache page must be flushed, so that memory can be updated before the requested line or lines are read into the cache. The state of the cache page is then set to Empty (E).

A fault occurs if a CPU-read is attempted on a line which is in the Stale state; software is invoked to purge the line and all other lines belonging to the cache page, then the data is read in from memory, and the state of the page and thus the line is set to Present (P). One example of where a subject is allowed to proceed immediately with its access is in the case where a subject match is made and the current state of the cache page is dirty.



**Table 5-1 State Transitions for Virtual Cache Coherence Management [Wheeler 1992]**

Operations	Target cache line		Aliases	
	Current State	New State	Current State	New State
CPU-read	E	P	E	E
	P	P	P	P
	D	D	D	(flush) E
	S	{purge) E (read in) P	S	S
CPU-write	E	D	E	E
	P	D	P	S
	D	D	D	(flush) E
	S	(purge) E (read-in and write to) D	S	S
Cache Purge	E	E	E	E
	P	E	P	P
	D	E	D	D
	S	E	S	S
Cache Flush	E	E	E	E
	P	E	P	P
	D	E	D	D
	S	E	S	S
DMA-read	E	E	E	E
	P	P	P	P
	D	(flush) E	D	(flush) E
	S	S	S	S
DMA-write	E	E	E	E
	P	S	P	S
	D	(purge) E	D	(purge) E
	S	S	S	S

**Table 5-2 Multiview FSM Definition for [Wheeler 1992] Virtual Cache Coherence**

Operations	Subject Match Bit 0 - no match 1-match	Current State	Result	New State
CPU-read	0 / 1	E	Fault	(read in) P
	0	P	Fault	(read in) P
	1	P	Proceed	P
	0	D	Fault	(flush) E (read in ) P (update SID)
	1	D	Proceed	D
	0 / 1	S	Fault	(purge) (read in) P
CPU-write	0 / 1	E	Fault	(read in) P (write) D
	0	P	Fault	S (read in) P (write) D (update SID)
	1	P	Proceed	D
	0	D	Fault	S (flush) E read in) P write D (update SID)
	1	D	Proceed	D
	0	S	Fault	(purge) E (read in) P (write) D (update SID)
	1	S	Fault	purge) E (read in) P (write) D

Cache purge and flush instructions are initiated through the software fault handlers which provide support for the Multiview memory. DMA reads and DMA writes cause forced state transitions for the cached data if the latter is in the Dirty state. A flush is initiated upon a DMA-read so that the most up-to-date copy of the data would be in memory; upon a DMA write, only a purge is needed. DMA writes will also cause cached data to change from the Present to Stale state.

Note that the objective of presenting Wheeler's scheme to enforce consistency on a

virtually addressed cache is to show how easily the Multiview model can support it. In addition, the advantages of the Multiview model can be realized in enforcing control on variable sized units (we can change Wheeler's constraint of maintaining state on a cache page), reducing the potential amount of context or thread switching and avoiding data cache pollution. Some TLB pollution is avoided in the Multiview implementation because the state information is cached in the PLB cache (i.e. hardware support for the state information is provided). With respect to the Multiview model, the TLB is affected on accessing the State Storage table and for any software fault handlers. Data cache pollution is avoided in the procedure to determine whether access can proceed or not.

## 5.2 Coherence Protocols

The memory model can be applied to state based coherence protocols by replicating the software and hardware support on each processor. All state-based coherence protocols are based on one of two mechanisms - invalidate or update. There are some extensions of the invalidate mechanism that includes update features [Rudolph 1984]. These mechanisms may be implemented in hardware, software with hardware assistance, or purely in software [Goodman 1987, Stenstrom 1990, Thacker 1987, Tomasevic 1993].

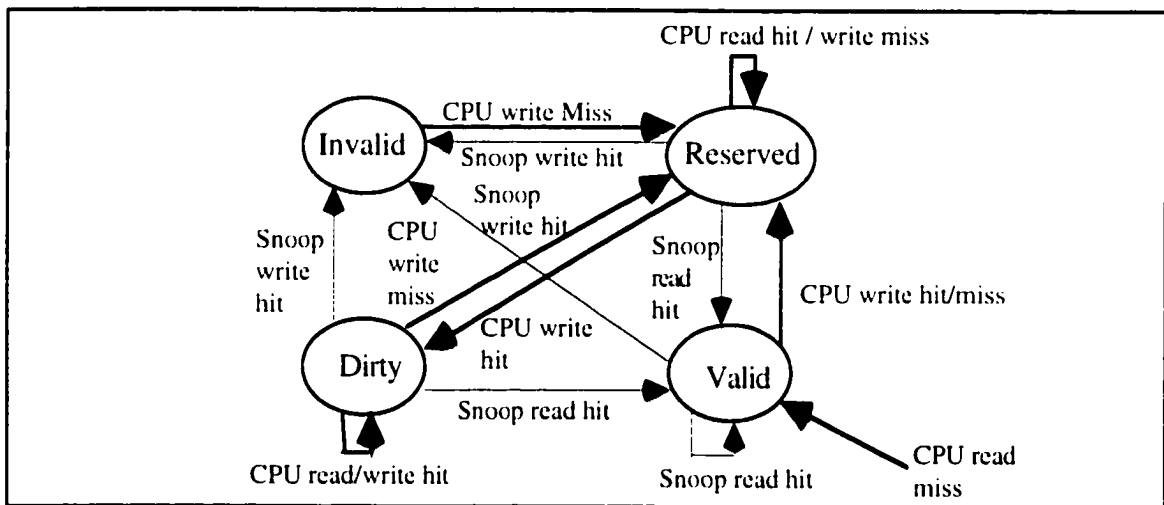
Hardware based protocols are fixed into the architectural design and all applications thus use the same protocol. It has been shown [Eggers 1988, Bennett 1990, Reinhardt 1994] that some applications' performance can improve vastly by their use of alternative cache coherence protocols. For example, Reinhardt [1994] shows how a user customized coherence protocol implemented in software outperformed an invalidation based hardware protocol. This was because an update scheme was more suited to the application under investigation. The update based protocols provide higher data availability and are good for applications that exhibit ping-pong effects within their invalidation patterns. The ping-pong effect is evidenced whenever multiple processors share a variable which is updated frequently by each, leading to heavy network traffic caused by read misses. A penalty associated with an update scheme is where processors may needlessly continue updating variables that will not be accessed locally again, but has not yet been replaced or will not be replaced for a long time due to a large cache size.

The main advantage of the FSM subsystem is that it can support both the invalidate and update-based coherence protocols within one computer system [Jutla 1994], yet it is faster

than a pure software implementation of these mechanisms. Applications are not forced to use a protocol that is not suited to their access characteristics, but can select and apply the protocol best suited to their needs.

### 5.2.1 Invalidation-based Coherence Protocol

This section describes how the Goodman [1983] write-once invalidation-based protocol could be implemented by an FSM definition. The Goodman [1983] protocol associates one of the following states with a cache line: Invalid, Valid, Reserved and Dirty. The Invalid state represents an out-of-date cache line. The Valid state signifies a coherent copy of the cache line is present, and several such copies may be present in the various memories in the system. A cache line in the Reserved state indicates that this copy of the object is the only up-to-date copy in the primary memory, and the only other consistent copy that exists is in another level of the memory hierarchy. Figure 5.1 illustrates the state diagram for the Goodman [1983] protocol.



**Figure 5-1 State Diagram for the Goodman Write-Once Protocol**

The Reserved state is used when it is known that the number of the other copies (the copy set size) of the data in the other caches within a system, is zero. It is used to enable a write operation through hardware without invoking a coherence fault, because invalidation commands need not to be initiated. Therefore, the write operation resulting in an Reserved to Dirty state transition can occur on-the-fly, without software handlers.

All other write operations on cached data that is not in the Reserved or Dirty states, generate coherence write faults. The Dirty state represents a copy of the data which is the

only up-to-date copy in the system.

The Invalid state may be entered upon a forced state transition. That is, the state transition is through software which is responsible for changing the current state of the cache line to Invalid. The Invalid state may be entered from any of the other three states.

The hardware implemented Goodman protocol additionally uses a snooping bus mechanism to maintain cache consistency, specifically by using CPU read/write requests to trigger the invalidation of cache lines. The invalidate instruction causes a forced state transition from either the Reserved, Dirty or Valid state to the Invalid state.

Table 5.3 shows the Multiview FSM definition for the Goodman Protocol. State information can be kept on a cache line or group of cache lines (access unit) basis. The Current state refers to the present state of the access unit. The New state refers to the state (defined by the FSM) corresponding to the transition, if any, caused by the memory operation on the access unit. A memory operation is either prohibited, i.e., a fault occurs, or it proceeds in which case a transition to a new state occurs, even if it is to the same state as the present one. State transitions occur not only on successful memory operations but on forced transitions, such as invalidation requests. Recall that a forced state transition is one performed in software by a fault handler. Note that forced state transitions are not explicitly shown in the table but are described in the following text. Certain state transitions depend on the result of a match between the requesting subject and the owner of the cache line. Hence the subject ID or owner information is part of the state information kept for a cache line.

A CPU read miss causes a fault which is first served and then the fault handler forces the cache line state to transition to the Valid state. The first write to data causes a write-through to main memory. Write hits, represented by matching subject IDs, cause invalidate commands to be issued to all other cached copies of the data item as well as the state transitions listed above for write hits. Read misses, as denoted by a non-match of subject IDs (0), represent the snoop read hits shown in the state diagram for the Goodman protocol.

The Goodman snooping bus mechanism does not have a hardware equivalent in the Multiview model but would be implemented through a software implemented invalidate instruction for snoop write hits. The invalidate instruction causes a forced state transition from either the Reserved, Dirty or Valid state to the Invalid state. Snoop write hits are emulated on a local write miss when the data is subsequently found cached in another processor's cache. Snoop read hits are emulated in the FSM definition by a non-match of

subject IDs.

**Table 5-3 A Multiview FSM Implementation of Goodman [1983] Invalidate Protocol**

Memory Operation	Subject ID Match 0 - no match 1 - match	Current State	Result (Proceed/ Fault)	New State
Read	0	Valid	Fault	(read-in) Valid
	1	Valid	Proceed	Valid
	0	Reserved	Fault	(read-in) Valid
	1	Reserved	Proceed	Reserved
	0	Dirty	Fault	(flush) (read-in) Valid
	1	Dirty	Proceed	Dirty
	0 / 1	Invalid	Fault	(read-in) Valid
Write	0	Valid	Fault	(read-in) Reserved
	1	Valid	Proceed	Reserved
	0	Reserved	Fault	(purge) (read-in) Reserved
	1	Reserved	Proceed	Dirty
	0	Dirty	Fault	(flush) (read-in) Reserved
	1	Dirty	Proceed	Dirty
	0 / 1	Invalid	Fault	(read-in) Reserved

The Goodman protocol is implemented in hardware with support from a bus based snooping mechanism. Hence it outperforms the Multiview implementation which relies on software fault handlers to issue invalidation instructions to other caches. However the advantage of the Multiview model is its flexibility to support a variety of protocols whereas the Goodman protocol, once implemented, is fixed and must be used by **all** applications on the system. The Multiview model allows applications to customize their consistency control requirements by choosing the most appropriate (highest performance) consistency protocol for their particular access characteristics.

### 5.2.2 Update-based Coherence Protocol

The Firefly [Thacker 1987] protocol which uses an update mechanism requires three states - Dirty, Valid-Exclusive and Shared. There are similarities in the state definitions to those of the write-invalidate mechanism. The Valid-Exclusive state is equivalent to the Reserved state above; the Shared state to the Valid state. The Invalid state is not needed since updates to copies are generated upon a subject's write operation. The Dirty state occurs only if there are no other copies to be found within the various address spaces, that is the copy set size is zero. Recall that in an update scheme writes to shared blocks are transmitted to each cache every time a write access completes. Table 5.4 shows the FSM definition for the Firefly protocol. Reads may proceed immediately once there are no pending updates to the data. Writes may also proceed immediately if the data is either in the Valid-exclusive or Dirty states since both these state signify that there is only one cached copy of the data item. However, a write to data in Shared state will generate further consistency operations to propagate the new data value to the rest of the copyset sites.

**Table 5-4 Firefly Update Protocol**

Memory Operation (hit)	Subject ID Match 0 -no match 1 - match	Current State	Result (Proceed/ Fault)	New State
Read	0	Valid-Exclusive	Fault	(read-in) S
	1	Valid-Exclusive	Proceed	Valid-Exclusive
	0	Shared	Fault	Shared
	1	Shared	Proceed	Shared
	0	Dirty	Fault	(flush) (read-in) Valid-Exclusive
	1	Dirty	Proceed	Dirty
Write	0	Valid-Exclusive	Fault	(read-in) Dirty
	1	Valid-Exclusive	Proceed	Dirty
	0	Shared	Fault	(purge copyset) (read-in) Dirty
	1	Shared	Fault	(purge copyset) Dirty
	0	Dirty	Fault	(flush) (read-in) Dirty
	1	Dirty	Proceed	Dirty

### 5.2.3 Integration of Concurrency/Coherence Protocols

Details on the viability as well as the benefits of integrating the concurrency and coherence protocols may be found in [Bellew 1990, Bodorik 1994b, Jutla 1993]. Briefly, the premise behind this integration is that a check for coherence of the data may be performed at the same time as the determination of the lockability of the data item. Generally these two tasks are handled by separate software handlers, that is a coherence manager and a transaction or lock manager respectively. Here, on-the-fly access control may be attained by the loading of the protocol transitions table into hardware and its incorporation into the protection mechanism.

The state transition table for the integration of the locking and the write invalidate views is shown in Table 5.5. There are five requisite states: Read-Only-Reserve (ROR), Read-Shared-Valid (RSV), Invalid, Write-Exclusive-Reserved (WER), and Write-Exclusive-Dirty (WED). The default state for data items is ROR; this corresponds to a combination of unlocked and reserved states.

**Table 5-5 Integration of Concurrency and Coherence Protocols**

Memory Operation	Subject ID Match	Current State	Result	New State
Read	0	ROR, RSV	Fault	(read-in) RSV
	1	ROR,RSV	Proceed	ROR,RSV
	0	WER	Fault	(purge) Invalid (read-in)
	1	WER	Proceed	WER
	0	WED	Fault	(flush) Invalid (read-in) ROR
	1	WED	Proceed	WED
Write	0	ROR	Fault	(purge) (read-in)WER
	1	ROR	Proceed	WER
	0/1	RSV	Fault	
	0	WER	Proceed	WED
	1	WED	Proceed	WED

The WER and WED states may be used to distinguish between the case of a first write by a subject and the case of subsequent writes performed by the same subject. This leads to a minimization of the work done on a write; as can be seen from Table 5.5, neither state transitions nor invalidation would be needed for a write operation issued on a data item



which was in the WED state. A restriction is imposed on the size of the protection units of views that are to be integrated: they must be of the same size.

### 5.2.4 Comments

Note that the families of invalidation and update hardware based protocols, such as the Illinois [Papamarcos 1984] and the Berkeley [Katz 1985] can be similarly decomposed into state transitions. Some protocols which require hardware support, in addition to that responsible for state transitions, such as the Dash protocol [Lenoski 1990] where the latter uses a special remote access cache to store states of pending memory requests and remote replies in order to optimize performance, will be slower in the Multiview model but are implementable.

## 5.3 Support for Mach 3.0 Pager

Table 5.6 shows the state transitions defined by the Mach 3.0 for external user-level pagers as described in [Loepere 1992], and also as would be described by a Multiview FSM implementation. The column labeled "Action" is not part of the FSM definition but has been included in the table for fuller comprehension of the Mach 3.0 external pager by the readers of this document. Note that the subject ID match column is omitted since it is not applicable to this protocol, as the subject ID for this application is equivalent to the task ID. There would always be a subject ID match as the threads of one task share the same task ID.

The r and w stand for read and write access respectively. States are denoted as a triple such as (m,r,r) and (m,r,w), and are formatted as <modified, current access, desired access>. The two values of the modified field are not shown here as this bit representation is used with the page replacement strategy, and do not affect states or transitions. It is shown here only for conformity with the presentation of the page states in [Loepere 1992]. The wait\_r and wait\_w states signify that the requested page is not yet in memory but that readers and writers are awaiting access. The wait\_r state is entered when a read access is requested for a page that is not in memory. Transition into the wait\_w state can occur in two ways: when a writer makes a request to a page that is in the wait\_r state or when a write access is requested on a page which is not in memory.

**Table 5-6 Kernel Transitions to Support the Mach 3.0 External Pager**

Memory Operation	External Operation	Current State	New State	Fault / Proceed	Action
Read		e	wait_r	Fault	obtain page from sec. memory
Write		e	wait_w	Fault	obtain page from sec. memory
Read/Write		wait_r	-	Fault	thread waits
Read		wait_r	m,n,r	Fault	kernel requests upgrade in protection
Read	Page is supplied into memory	wait_r	m,r,r	Proceed	
Read	Kernel generates a writeable page or page is supplied with write access rights	wait_r	m,w,w	Proceed	
Write	Page is supplied with read access	wait_w	m,r,w	Fault	Kernel sends message to manager requesting write access
Write	Kernel generates a writeable page or page is supplied with write access rights	wait_w	m,w,w	Proceed	
Write	Kernel supplies page with no access	wait_w	m,n,w	Fault	Kernel requests upgrade in protection
Read		wait_r	e	Fault	Memory access exception
Write		wait_w	e	Fault	Memory access exception

## 5.4 Conclusions

This chapter showed the wide applicability of the Multiview memory model in its support of a variety of important protocols. In addition, the model avoids the heavy use of software for access control purposes. This benefits in the avoidance of TLB and data cache pollution. The examples show that the Multiview model provides for flexibility - no one

protocol has to be used for all applications.

The FSM implementation for the important 2-phase locking protocol is provided in the next chapter. The evaluation of the 2-phase locking protocol FSM implementation versus a conventional software 2-phase locking implementation is presented in subsequent chapters.

# CHAPTER 6

## EVALUATION

This chapter describes the evaluation of the architectural design presented for the Multiview Memory Model in Chapter 4. The main objectives of this evaluation are the following.

- (1) To determine the delays due to the operations of the Multiview Memory model when providing access control services.
- (2) Evaluate how the sizes, and consequently the miss rates, of the TLB, PLB and L2 caches affect the overall delay in determining whether a read or write access can proceed or not.
- (3) To determine which cache component in the architectural design dominates the delays due to the PCU.

The evaluation is done for one access control protocol, that of locking, and hence for one view. Locking is commonly used by a wide range of applications to provide access control services. Specifically, the study targets the measurement of the cost of lock acquisition without conflicting access. Note that in the common case, locks do not conflict. The main performance metric is delay in machine cycles for lock acquisition on a read or write access. It would have been better to obtain execution time, but there is no actual implementation of the Multiview memory model as yet. Other metrics, used in the analysis, include number of TLB, PLB and Data cache misses/hits, and number of Page Table, State Table and data memory accesses.

The entire evaluation exercise spans this and the next chapter. Chapter 6 costs lock acquisition under the Multiview memory model and Chapter 7 costs lock acquisition implemented in the traditional manner through software. The comparison is made with a software scheme rather than a dedicated hardware locking mechanism, since software implementations are flexible and an important advantage of the Multiview model is its flexibility in its support for a wide range of protocols. Chapter 7 not only provides a *quantitative* comparison of the lock acquisition methods but also a component description of lock acquisition and a *qualitative* comparison of the methods (see Section 7.1.1).

Trace-driven simulation is used to obtain values for the performance metrics for the model since actual program traces facilitate the capture of the characteristics of an

application better than an analytical model can. Many conflicting results have been published in the literature through the use of analytical models to describe, in particular, database application behaviour [Rodriquez-Rossell 1976, Smith 1978, Kearns 1983, Effelsberg 1984]. This has mostly been because real application behaviour cannot be exactly modeled by statistical means, particularly with the numerous assumptions that are inherent to analytical modeling.

The simulation study represents one of the steps to an actual systems implementation of the Multiview memory model. Simulation can uncover or confirm expected bottlenecks and their associated costs. It provides quantitative results which can be used to justify a decision to implement or not. Simulation is particularly applicable to the architectural support selected for the Multiview model in this work since additional off-chip caches are proposed and these are expensive to implement for experimental purposes.

A database application was chosen to be traced because it is in an important class of applications which maintain shared files and thus frequently require access control services. There have been many previous attempts to support locking in a more efficient manner than through software -only means [Chang 1988, Stonebraker 1985]. The traced database application is along the guidelines of the Transaction Processing Council's TPC-C benchmark for evaluation of Database Management Systems. Descriptions of the characteristics of the traced application and the various transaction types are given in Section 6.1.2. What is important in this study is that the application is query and insertion intensive, thus generating many read and write requests for locks.

The outline of this chapter is as follows. Section 6.1 presents the simulation environment. It describes the simulation inputs; for example, the format of the memory traces, hardware parameters and cost assignments. The characteristics of the applications which were memory traced for creation of the simulator's input are presented. Section 6.2 describes the example access control protocol (locking) and its FSM description. The experiments are described in section 6.3 and the results are provided in section 6.4. Section 6.5 discusses the results produced from the simulation exercise.

## 6.1 The Simulator

For the evaluation, a simulator that traces access through a hierarchical memory system was designed. The code for the simulator is provided in Appendix A. The simulator models caches, physical/virtual memory, and thread/context switches. Statistics are kept on the activity incurred at all levels of the memory hierarchy. Statistics kept on caches include the number of hits/misses due to read access or write access, the number of misses and the entire delay due to caching activity. Statistics on memory are primarily targeted at the number of memory accesses and the number of faults to secondary storage.

Caches support a number of organizational parameters - three major ones being cache line size, the degree of cache associativity and the number of sets. Additionally a cache can be virtually or physically addressed, and virtually or physically tagged. The contents of a cache line's tag determine what is a "hit" on the cache. The tag, and the number of operations needed for the determination of a hit, differ from cache to cache. Management policies exist for caches such as line replacement policies (e.g. LRU, random), write policies (e.g. write back, write through), write miss policies (e.g. write allocate, no write allocate), and coherence policies (e.g. invalidate, update). How LRU approximation is implemented on one cache can be different from another implementation. The choice of design parameters impacts a cache's miss rate, miss penalty, hit time and hardware complexity. System designers must be allowed to configure a cache with any combination of the above parameters and policies. Therefore the simulator was designed to accept any cache configuration.

Accesses to caches are triggered on read and write references issued by an application program. Cache accesses may also occur due to explicit operations such as cache flushes and invalidations. For systems under design and development, memory traces are used to drive cache and memory accesses within simulation exercises. The Quick Profiler and Tracing Tool (QPT2) [Larus 1995] is used in this work to produce address traces.

The software for the simulator includes code which releases locks at transaction end. The costing of release operations is not included in this analysis. As stated before, the study targets lock acquisition. The interested reader is referred to Jutla [1996] for a detailed description of the design of the simulator. The source code for the simulator is provided in Appendix A.

### 6.1.1 Simulation Input Parameters for the Multiview Memory Model Architecture

This section describes the various inputs to the simulator. First the memory trace is described. Then the costs for activities in the memory hierarchy are tabulated. Arguments for the assigned costs are presented.

### 6.1.2 Memory Trace Input

An executable program is memory traced to provide input to the simulator in the form of reads and writes to virtual addresses. The format of the address trace is as follows. The R stands for a read access and the W for a write access to the 32-bit virtual address represented in hexadecimal notation.

R: efffc40

W: aaa22048

R: effdd330

The executables used to measure the locking function performance generate a repeatable number (24) of 5 types of transactions which in turn issue read and write accesses to 7 shared data tables. The read and write data accesses are skewed with respect to the targeted access units and are generated by a random number generation function. Transactions are interleaved in varying orders to provide several input files. The variation in transaction mix allows for control of the ratio of read and write accesses within an application and for different locality of access. Access units here are defined to be tuples of the relation.

Four transaction mixes (applications) are memory traced and used as input to the Multiview simulator. There are 5 different types of transactions used in the TPC-C benchmark [Transaction Processing Council 1992]. The TPC-C benchmark represents a generic wholesale supplier workload. The transactions are reproduced from [Leutenegger 1993]. They are presented to describe each transaction type. The transactions operate on seven shared data tables: Warehouse, District, Customer, Order, Stock, New-Order, and History data tables.

#### **Transaction Type 1: New Order Transaction**

1. Select (whouse-id) from Warehouse
2. Select (dist-id,whouse-id) from District
3. Update(dist-id,whouse-id) in District

4. Select(customer-id,dist-id,whouse-id) from Customer
5. Insert into Order
6. Insert into New-Order
7. For each item (10 items):
  - (a) Select(item-id) from Item
  - (b) Select(item-id,whouse-id) from Stock
  - (c) Update(item-id,whouse-id) in Stock
  - (d) Insert into Order-Line

### **Transaction Type 2: Payment Transaction**

1. Select (whouse-id) from Warehouse
2. Select (dist-id,whouse-id) from District
- 3 (a) Case 1: Select(customer-id,dist-id,whouse-id) from Customer  
 (b) Case 2: Non-unique Select(customer-name,dist-id,whouse-id) from Customer
4. Update(whouse-id) in Warehouse
5. Update(dist-id,whouse-id) in District
6. Update(customer-id, dist-id,whouse-id) in Customer
7. Insert into History

### **Transaction Type 3: Order Status Transaction**

- 1.(a) Case 1: Select (customer-id, dist-id,whouse-id) from Customer  
 Case 2:Non-Unique-Select(customer-name,dist-id,whouse-id) from Customer
2. Select(Max(order-id),customer-id) from Order
3. For each item in the order  
 Select(order-id) from Order-Line

### **Transaction Type 4: Delivery Transaction**

1. For each district within the warehouse
  - (a) Select(Min(order-id),whouse-id,dist-id) from New-Order
  - (b) Delete (oredr-id) from New-Order
  - (c) Select (order-id) from Order
  - (d) Update (order-id) Order
  - (e) For each item in the order
    - (i) Select (order-id) from Order-Line



- (ii) Update (order-id) Order-Line
- (f) Select(customer-id) from Customer
- (g) Update (customer-id) Customer

**Transaction Type 5: Stock Level Transaction (given in SQL)**

```

SELECT d_next_o_id INTO :o_id
FROM District
WHERE d_w_id = :w_id AND d_id = :d_id ;
SELECT COUNT(DISTINCT(s_i_id)) INTO :stock_count
FROM Order-Line, Stock
WHERE
    ol_w_id = : w_id AND ol_d_id = :d_id AND ol_o_id < :o_id AND
    ol_o_id >= (:o_id-20) s_w_id = :w_id AND
    s_i_id = ol_i_id AND s_quantity < :threshold;

```

The transactions in the mixes operate on data tables which are pre-initialized to the sizes as shown in Table 6.1. The tuple sizes, in bytes, are shown in Table 6.2. The Order-Line and History Tables are not pre-initialized since their entries are created as the transactions are processed.

Transaction mix 1 is created from the serial execution of 24 transactions in the following order by type. The order was randomly generated as per TPC-C specifications, and are listed by type below:

1, 2, 3, 4, 5, 1, 3, 3, 4, 4, 3, 4, 3, 3, 1, 5, 4, 1, 1, 3, 3, 3, 1, 4

Transaction mix 2 is a serial execution of 24 transactions generated in the same order as above. All database sizes are increased by a factor of 10.

Transaction mix 3 results from a change in the order of transaction execution. The database sizes are the same as for mix 2. The order in which the transactions are generated are (by type):

1, 2, 3, 4, 5, 3, 4, 4, 1, 5, 4, 2, 3, 2, 1, 1, 1, 3, 4, 1, 2, 1, 5, 3

Transaction mix 4 is generated using a size of 2000 records for the Customer database, 500 Warehouses, 15000 records in the Items database and 10000 items in the Stock database. The order of execution of the transactions is the same as transaction mix 3 except that the numbers of tuples to be inserted are changed as shown in Table 6.1

The sizes of the data tables are varied across the transaction mixes to achieve

differences in the locality of access and hence to impact the miss rates on the various caches under study. The varying data table sizes also results in a differing number of unique data accesses. The order of transaction execution is changed in Transaction mix 3 from that found Transaction mixes 1 and 2 as one means to vary the content of the various caches when one transaction finished and another started in the mixes. Other ways to vary caches' content for each transaction mixes are by specifying variations in the various cache sizes which will be discussed later.

**Table 6-1 The Number of Tuples to be Inserted and Sizes of the Data Tables used by the Transactions in Mixes 1-4.**

Transaction Number	Mix	Data Table	Size of Data Table (raw data only) (kB)	Number of Tuples to be inserted in Data Table
1		Warehouse	.64	5
		District	2.56	10
		Customer	100	20
		Item	3.2	20
		Stock	51.2	20
2		Warehouse	6.4	50
		District	25.6	100
		Customer	1000	200
		Item	32	200
		Stock	512	200
3		Warehouse	6.4	50
		District	25.6	100
		Customer	1000	200
		Item	32	200
		Stock	512	200
4		Warehouse	64	250
		District	25.6	100
		Customer	2000	800
		Item	1920	1200
		Stock	5120	900

A view, with a unique view ID, is defined on each data table in order to enforce the 2-phase locking protocol. For each view, the access unit size is defined to be the data table's tuple size. The FSM definition for the 2-phase locking protocol is associated with each view.

**Table 6-2 Tuple Sizes for Data Tables.**

Data Table Name	Tuple Size (in bytes)
Warehouse	128
District	128
Customer	1024
Item	128
Stock	512
Order	32
New_Order	8
Order-Line	64
History	64

Table 6.3 provides the number of unique lock units accessed by the respective applications. Table 6.4 shows the percentages of read and write accesses for each application.

**Table 6-3 Number of Unique Lock Units per Transaction Mix**

Transaction Mix	Number of Unique Lock Units
1	68
2	228
3	256
4	482

**Table 6-4 Percentages of Read and Write Accesses to the Data Tables for each Application**

Transaction Mix	% of Read Accesses to Data Tables	% of Write Accesses to Data Tables
1	75	25
2	77	23
3	74	26
4	76	24

It is acknowledged that the size of the data component of the data tables in this study is small. Transaction mix 4 sets the data tables sizes and, in particular, the number of tuple insertions, at the largest for the study. The number of tuple insertions were limited by available disk space on the machine on which the simulations were run. A sample size of one of the input trace files which were generated from the application working on the seven data tables is 112 Mb. The initial input file was then used as input to a utility program which detected where the shared data regions existed in the virtual address space and where the transactions began and ended. Begin-of-Transaction (BOT) and End-of-Transaction (EOT) markers were then placed at appropriate places by the utility program which generated yet another trace file while processing the first. Undoubtedly, with advances in computer technology, these sizes will appear to be puny in a short while. Nevertheless, at the time of experiments, we worked with available disk sizes.

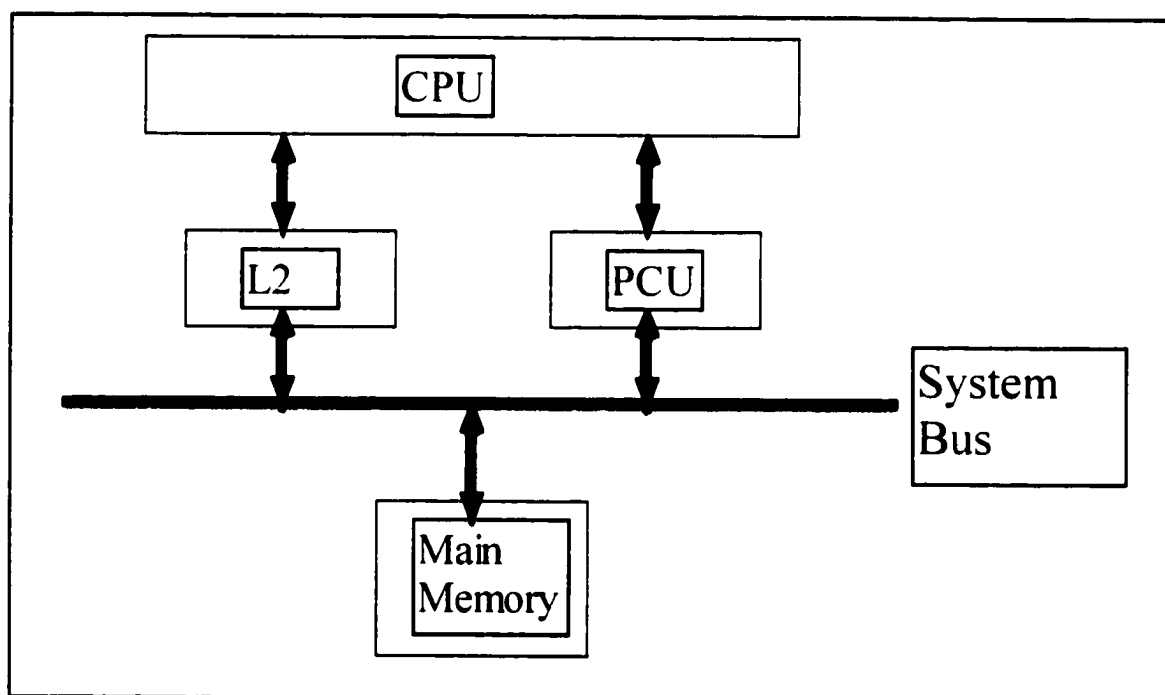
Another drawback of the simulator's input is that the tracing tool is unable to trace operating system code so it is not possible to measure the impact of context switching overhead on the TLB and data caches. Even if we had an OS tracing tool, there is no OS that currently supports the Multiview memory model. The measurements of the TLB miss rates are undoubtedly affected by the lack of OS code tracing. Because we do not have a trace of OS code on context switches, the effect of these switches was minimized by the serial execution of transactions: one transaction is run to completion before another starts. That is, the TLB miss rates are more optimistic in this study than actual values. Also the caches are not flushed on transaction completion since their entries are tagged and hence this policy has a much smaller effect at context switch time than mandatory cache flushing.

### **6.1.3 Hardware Parameters Input**

A block diagram of the simulated hardware architecture is shown in Figure 6.1. The PCU unit is situated in between the CPU and the main system bus. Dedicated data paths exist between the CPU and the L2 cache, and between the CPU and the PCU unit. Description of the PCU invocation can be reviewed in Chapter 4. Please note that when the term "data cache" is used without qualification, throughout this work, it refers to the L1 cache.

Table 6.5 summarizes the hardware cost in cycles discussed here. It is assumed that the CPU cycle time is equivalent to the data cache cycle [Przybylski 1990]. The data cache

type is virtually indexed, physically tagged. L1 and L2 caches are simulated. Read hits on the L1 cache take one CPU cycle ( $L1_R$ ). Write hits, ( $L1_W$ ), take two [Handy 1993] - one to access the tags, and one to access the R/W bit. Write-back with no fetch done on a write miss is the write policy used with the data cache. There is an automatic fetch on a read miss. On an L1 data cache read miss, there is an overlap of the memory-to-cache data transfer and the operations of the Protection Control Unit. However, memory accesses are serialized for all caches with respect to misses. A four line write buffer between the cache and the memory is assumed. A 128-byte line is assumed in the data cache. A two word (8 byte) line is sufficient for the PLB cache. It is also assumed that the memory interference from DMA devices is negligible. The loading of the L2 cache from memory consumes 50 cycles.



**Figure 6-1 Block Diagram of Hardware**

The access of the mask from the ViewDefinition cache and its transmission to the PLB takes two cycles. The masking procedure before a PLB access is assumed to take 1 cycle which is reasonable for logic that requires a parallel bank of AND gates or analogously a bank of NAND and inverter gates. It is assumed that the masking logic is laid out close to the PLB cache. Thus it takes 5 cycles for a PLB hit. The delay due to a PLB miss is

calculated by summing delays due to the masking operation, the PLB cache access time, the forming of the virtual address of the state information, the average TLB access time, the sending of the physical address to memory, memory latency and the memory-to-PLB cache transfer. It is assumed that the virtual page number component of the VA was found via a ViewDefinition cache lookup. The procedure (involving shifting, masking and ORing) required for the formation of the virtual address of the state information unit in memory is assumed to take 3 cycles. The delays due to wiring capacitance and lengths are assumed to be negligible as compared to memory access times.

The ViewDefinition and FSM caches hit ratios are assumed to be 100%. The ViewDefinition cache must be flushed and reloaded on every context switch and the number of ViewDefinition entries is expected to be small. There are 9 views as there are 9 data tables, and there is one view per table. If a system only needs to support a medium number of different protocols at any one time, say 10, then the FSM cache can be designed so that the 100% hit rate is certainly achievable. Only one protocol was modeled in this simulation study. Table 6.5 provides the hardware input costs.

**Table 6-5 Operations And Typical Delay In Cycles**

<u>Operations</u>	<u>Typical Delay (in cycles)</u>
L1 <sub>R</sub> : Data cache read hit	1
L1 <sub>W</sub> : Data cache write hit	2
L1 <sub>RM</sub> : Data cache miss: (L2 read and write hit)	13
L2 <sub>RM</sub> : Data cache miss (L2 read and write miss)	50
WCF : Write access control fault	2
PLB <sub>H</sub> : PLB hit	5
PLB <sub>M</sub> : PLB miss	40
FSM <sub>H</sub> : FSM hit	5
VT <sub>H</sub> : ViewDefinition cache hit	7
TLB <sub>H</sub> : TLB hit	1
TLB <sub>M</sub> : TLB miss	20
SSVA: State Storage VA formation	4
D <sub>PLB</sub> : Data Transfer between PLB and FSM	1
L: Accessing mask from the View Table (involves VT access and transmission)	2
D <sub>FSM</sub> : Transfer from FSM to PLB cache	2

## 6.2 The Access Control Protocol

An example access control protocol is selected for measuring the operations and overhead of the Multiview memory model when providing its synchronization services. The two phase locking protocol, the standard protocol used by database applications, is used as the access control protocol. The FSM definition of this protocol is given below.

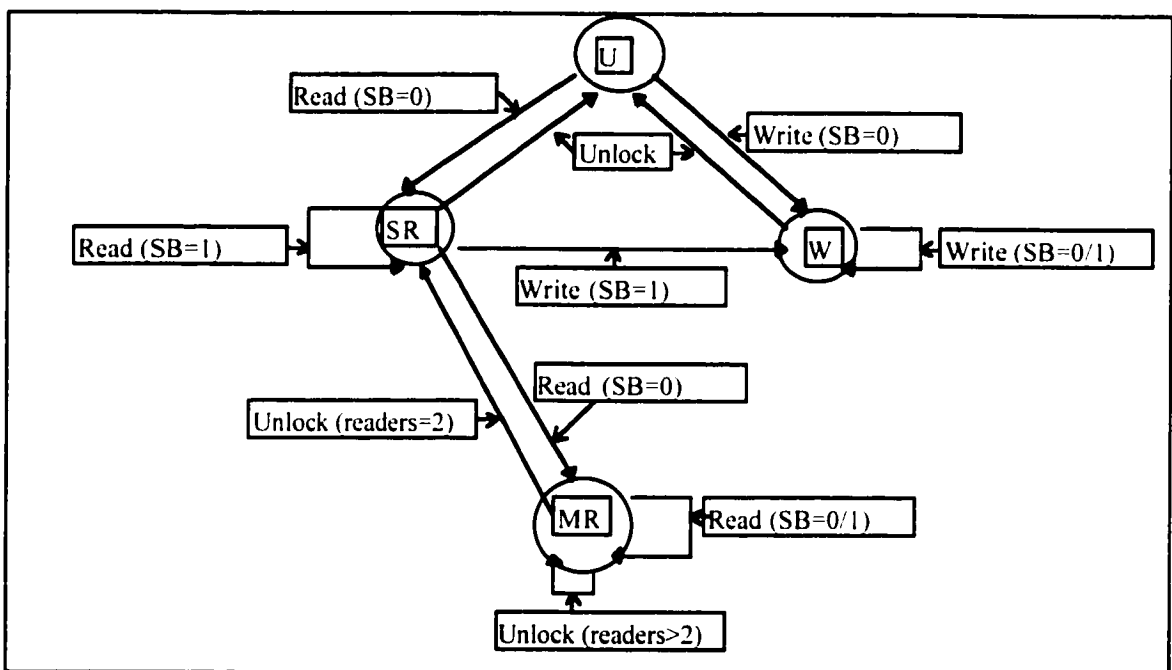
The handling of lock requests in the multiple readers-single writer locking protocol can be fully decomposed into state transitions as shown in Figure 6.2. This decomposition facilitates the determination of whether a subject can be granted a lock, or whether the subject must be suspended, **without** hashing to a Lock Table and the subsequent traversal and access of linked lists in memory. The determination of whether to grant the requested lock can be performed, in the best case, through less than five cache lookups.

Maintenance of four states - Single Reader (SR), Multiple Readers (MR), Write-locked (W) and Unlocked (U) is required. Recall that on each transaction, the subject ID of the thread issuing the read/write operation causing the transition, is recorded with the state. Also recall that some transitions depend not only on the current state of access and the operation (read/write) but also on whether the Subject ID of the thread issuing the operation matches the subject ID recorded in the state.

The SR state indicates that only one reader has a lock on the access unit. The subject ID recorded with the state represents the transaction ID holding a read lock on the access unit (tuple). The MR state represents the presence of multiple read locks on the access unit by various subjects. The W (Write-locked) state is entered when a write memory operation succeeds and all other subjects are prohibited from accessing the object. As in the single reader's case, the subject ID of the writer recorded with the state of the access unit represents the ID of the transaction holding a write lock on the access unit.

When the concurrency control view is created for a data table, the initial state in the access matrix is the Unlocked state. In the initial state, no subjects are associated with the lock units. When a read or write request is issued, the state of the lock unit transitions to SR or W, respectively, while the subject ID (transaction ID) is recorded in the state. If a read operation is issued on a unit in the SR state, the state can transition either to the MR state, or remain in the SR state, depending on whether the thread's subject ID matches that stored with the state. Complementary information, in the form of a match between the requesting subject ID and that stored in the state information for the lock unit, is needed in order to distinguish the cases SR->SR and SR->MR. The SB=0, SB=1 and SB=0/1 labels in Figure 6.2 represent the match results. SB=0 means that the requesting subject

and the subject that currently holds the lock, i.e., the subject ID recorded with the state, are not the same.  $SB=1$  represents a match, and  $SB=0/1$  means that the result of the match does not affect state transitions. The transition  $SR \rightarrow W$  represents a lock upgrade. That is, if the lock unit is in the SR state, and a write request is made by the subject which currently holds the read lock (Write ( $SB=1$ ) in Figure 6.2), the write lock will be granted. When a transaction completes, locks it holds are released which is represented by the unlock transition. For lock release when a unit is in the MR state, two possibilities exist: if there are more than two readers, transition is back to the MR state, while if there are two readers, the transition is to the SR state.



**Figure 6-2 State Transition Diagram for Locking**

Table 6.6 shows the FSM definition of the lock requests handling portion of the locking protocol. When a memory operation is issued by a subject S on the memory location contained within a concurrency control unit, the memory operation is either prohibited, i.e., a fault occurs, or access is granted. If access is granted, indicated by a "proceed" flag in the Result column of Table 6.6, transition to the state indicated in the column New State occurs. Note that the table shows only a subset of the state transitions shown in Figure 6.2. It shows only the transitions for read/write requests, the transitions determinable through table lookups only. State transitions due to unlocking are not shown



as these are achieved in software at transaction end. Recall that such software-performed transitions are, in the Multiview model, referred to as forced transitions.

The Subject Match bit is required in order to implement the protocol as mainly table lookups. The Subject Match bit caters for having the access unit in a single state when viewed by all accessing subjects, yet triggering different state transitions depending on which subject may currently hold a lock on the access unit. For example, a read operation on a unit that is in the SR state can trigger either of two state transitions (either SR->SR or SR->MR). The SR->SR state transition occurs if a subject issues a repeated read to an access unit for which it already holds a Read lock. This is represented in the Read/1/SR entry in the table. Read/1/SR stands for the Read memory operation, a Subject Match Bit of 1 and a Current State of SR. The SR->MR state transition occurs when a subject other than the one currently holding the Read lock issues a read request on the access unit.

Note that the subjects in this environment are transactions. Thus subject IDs will be transaction IDs (TIDs). The Multiview FSM definition is more storage efficient than that of IBM's 801 scheme (see section 2.2.2.2) where page table entries containing locking information were kept on a per transaction basis, thus generating more state information. That is, various transactions see the same unit in different states in the IBM 801 implementation of hardware locking because state information is stored on a per transaction basis in a special table.

Whether the subject has access rights is determined by matching the subject ID stored in the state entry for the access unit (entry in PLB table) with that of the accessing subject and by matching the issued memory operation with that in the appropriate FSM entries. Wherever there is an entry of "Proceed" in the table it means either, that locks may be granted on-the-fly, that is, without faulting to the software supported portion of the lock manager, or that access can proceed because the issuer of the memory access already owns the lock. There are no Read/1/Unlocked and Write/1/Unlocked entries in the table since an access unit in the unlocked state does not have a Subject ID associated with it.

The recording of the transaction's lockset for lock release and thus transaction commit purposes is done on each transaction's data access. The storage of the transaction's lockset is implemented in a hardware queue. On a context switch, the queue is forwarded to the view manager which records the lock acquisitions stored in the queue in its data structures.

**Table 6-6 State Transition Table for Locking**

Memory Operation	Subject Match Bit		Current State	Result (Proceed/Fault)	New State
	0 - no match	1 - match			
Read	0		Unlocked	Proceed	SR
Read	0		SR	Proceed	MR
Read	1		SR	Proceed	SR
Read	0		MR	Proceed	MR
Read	1		MR	Proceed	MR
Read	0		W	Fault	W
Read	1		W	Proceed	W
Write	0		Unlocked	Proceed	W
Write	0		SR	Fault	SR
Write	1		SR	Proceed	W
Write	0		MR	Fault	MR
Write	1		MR	Fault	MR
Write	0		W	Fault	W
Write	1		W	Proceed	W

### 6.3 The Experiments

A transaction mix is generated from a repeated selection of one of the five transaction types. The transaction mix represents a database application requiring access control services. Each transaction is unique even if the transactions are of the same type in that the access units selected for access by each transaction were randomly chosen from a skewed distribution, as specified by the TPC-C benchmark [Transaction Processing Council 1992]. Each transaction mix demonstrates differing locality of access per cache (resulting in varying miss rates), and variation in the read/write ratio.

All caches are simulated with a copy-back write policy, LRU replacement policy and write allocation. The line sizes for the PLB, TLB, ViewDefinition cache and FSM caches are set at 8 bytes. The line sizes for the L1 and L2 data caches are set at 128 bytes. The TLB and the FSM caches have capacities of 256 bytes, or 32 entries, each. The L1 data cache is set at 8Kb within all the experiments. The ViewDefinition cache has a capacity of 128 bytes, or 16 entries. Recall that the PLB, ViewDefinition cache, L2 data cache and FSM caches have cycle delays assigned to them assuming that they are situated off-chip.

The average delays, in terms of cycles, before a read or write access is allowed to proceed or a fault occurs, are measured for each transaction mix in a simulation run. This

is translated into the delay incurred by the operations of the Multiview model when providing access control services. The sizes of the L2 data cache and the PLB caches are varied to meet objective two of the evaluation, that is, to evaluate how the relative sizes, and consequently the miss rates, of the PLB and L2 caches affect the overall delay in determining whether a read or write access can proceed or not. The input of different transaction mixes to the simulated architecture results in variations in the TLB miss rate and hence shows the impact of the TLB on the architecture.

## 6.4 Multiview Lock Management Results

The results of the simulation are presented in the next two subsections. First the values for the average delay of the PCU unit in processing a read or write request is presented in subsection 6.4.1. Section 6.4.2 presents the results of the measurements of the number of page table accesses, TLB accesses, PLB accesses, state storage table accesses and L1 data cache accesses for each transaction mix.

### 6.4.1 PCU Delay Results

Table 6.7 documents the results in terms of the number of cycles expired before a read/write is allowed, i.e., before it is determined that the lock is acquired or denied. Recall that since the primary data cache(L1), as is used in experiments here, is constrained by on-chip area and in many systems is approximately 8K in size, the secondary (L2) data cache (off-chip) was varied in size to determine its effect on the lock acquisition delay. The L1, L2 and PLB caches are 2-way set associative. The L2 Miss Rate is defined as the  $(\text{Number of L2 Cache Misses} / \text{Total L2 Cache Accesses})$ . The L2 Miss Rate definition is the local miss rate on the L2 cache as opposed to the global miss rate. The global miss rate  $(\text{Number of L2 Cache Misses} / \text{Number of Data Accesses})$  on the L2 cache is much lower than its local miss rate.

The PLB Miss Rate is defined as the  $(\text{Number of PLB Cache Misses}) / (\text{Total Number of PLB Cache Accesses})$ . The Average Read Delay is obtained by the  $(\text{Total Accumulated Read Delay for a Transaction Mix}) / (\text{Total Number of Read Accesses})$ . Similarly, the Average Write Delay is obtained by the  $(\text{Total Accumulated Write Delay for a Transaction Mix}) / (\text{Total Number of Write Accesses})$ .

**Table 6-7 Delay to Obtain Read/Write Locks under Various Cache Configurations**

Transaction Mix	Number of sets (L2)	L2 Miss Rate (%)	Number of sets (PLB)	PLB Miss Rate (%)	Avg. Read Delay (cycles)	Avg. Write Delay (cycles)
1	4096	42	512	.9	49	45
	2048	42	512	.9	49	45
	1024	42	512	.9	49	45
	512	42	512	.9	49	45
	512	42	1024	.9	49	45
	512	42	256	.9	49	45
	512	42	128	1	49	45
	512	42	64	2	50	45
	512	42	32	3	50	46
	2	4096	50	512	3	63
2048		50	512	3	63	61
1024		50	512	3	63	61
512		50	512	3	63	61
512		50	1024	3	63	61
512		50	256	3	63	61
512		50	128	3	63	61
512		50	64	6	64	62
512		50	32	8	64	62
3		4096	59	512	3	79
	2048	59	512	3	79	69
	1024	59	512	3	79	69
	512	59	512	3	79	69
	512	59	1024	3	79	69
	512	59	256	3	80	69
	512	59	128	4	80	69
	512	59	64	9	80	70
	512	59	32	9	80	71
	4	4096	68	512	4	82
2048		68	512	4	82	73
1024		68	512	4	82	73
512		68	512	4	82	73
512		68	1024	4	82	73
512		68	256	5	82	73
512		68	128	6	82	73
512		68	64	9	83	75
512		68	32	13	83	77

The PLB miss rates for Transaction Mix 4 vary between 4 and 13 %. However the corresponding cycle times for read and write delays before access is allowed or denied

change by less than 2 cycles. The PLB miss rates for Transaction mix 3 varies between 3 and 9 % and the corresponding measured cycle times change by less than 1 cycle. This shows that the tested variations in the PLB miss rates have little effect on the measured cycle times for a particular transaction mix. The L2 miss rate is fairly stable within each transaction mix for variations in cache sizes from 128Kb (line size of 128 bytes \* cache associativity of 2 \* number of sets (512)) to 1Mb.

The variation in cycles measured for Transaction mixes 1 thru 4 must be explained. From above, it is not due to the PLB miss rates. If the number of cycles obtained for Transaction mix 2 with a PLB miss rate of 8 % (63 cycles) is compared to the number obtained for Transaction Mix 3 with a PLB miss rate of 3% (79 cycles), it is clear that the PLB miss rate does not affect the measured delay overhead. The L2 miss rate is a contributing factor to the increase in delay (from 49 cycles in transaction mix 1 to 83 cycles in transaction mix 4) since its miss rates change from 42 to 68%. However at this point, we cannot say the L2 miss rate is the only contributing factor to the increase in delay overhead. The values of the other metrics (e.g. number of Page Table Accesses, Number of data accesses to memory) are documented below to provide for further analysis.

In conclusion, the two main results shown in Table 6.7 are the average read and write delays incurred before the read/write request is satisfied or denied and that the variation in PLB and L2 cache sizes show little effect on the average number of cycles which expire before a read or write access is resolved within a particular transaction mix.

#### **6.4.2 Access Results for the Transaction Mixes**

A particular configuration of cache sizes is assigned as the base system configuration: 8K L1 Cache, 128K L2 cache (512 sets, 2-way associative), and 8K (1024 sets) PLB cache. These cache sizes were selected after analysis of the results shown in Table 6.7 since the PLB and L2 data cache sizes have very small effects on the overall read/write lock acquisition delays within a single transaction mix. Note that the relative values for other configurations within a particular transaction mix are similar and hence are not presented here. The number of accesses to various levels of the memory hierarchy and to the caches within the PCU unit, for the base system configuration, are recorded in Table 6.8.

**Table 6-8 Access Characteristics of Transaction Mixes**

Access Type	Mix 1	Mix 2	Mix 3	Mix 4
	Number of Accesses			
Page Table (Main Memory)	820	13304	28489	35801
State StorageTable (Memory)	702	1230	1347	2204
Data ( Main Memory)	3285	3693	3226	3378
Data Cache (L1)	251838	267810	267470	279622
PLB	10668	9664	10539	12934
TLB	252612	280982	295818	315122

Table 6.8 shows a large increase in page table accesses (equivalent to TLB misses) from Transaction mix 1 to 4. There is a difference of almost 35000 page table accesses between transaction mix 4 and transaction mix 1. The number of TLB accesses increases by 62510 accesses (315122-252612) from transaction mix 1 to transaction mix 4. The difference in PLB accesses from transaction mix 1 to transaction mix 4 is 2266, and the number of State Storage Table accesses changed by 1502 accesses. These differences are due to the larger working set for transaction mix 4 as compared to transaction mix 1. to the larger number of tuple insertions and to the larger number of collisions in the caches. Also note that on transaction commit when locks are released the PLB cache entries are updated and so are the corresponding State Storage table entries.

In a system with virtually indexed and physically tagged data caches, the TLB is accessed on each data cache access in order to retrieve the physical page number which is then compared with the tag of the data cache entry. Thus, the number of TLB accesses is equal to the number of data cache accesses. Here, however, the numbers for TLB Accesses are higher than the numbers for the L1 Data Cache accesses, as shown in Table 6.8, since PLB fault handling needs translations for virtual memory addresses.

The 4 applications exhibit high spatial locality of access as can be seen from the low miss rates on the L1 data caches (see Table 6.9). One other reason that locality of access is high is that the caches are not flushed on a context switch since their entries are tagged.

Table 6.9 shows once more that the PLB miss rate varied by approximately 3% from Transaction mix 1 to Transaction mix 4 and thus the PLB is not the dominant contributing factor to the measured overhead delay. However there is a wide variation in the TLB miss rate from Transaction Mix 1 to Transaction mix 4 and there is very little variation in the L1

miss rate. Note also that the variation in the miss rate on the L2 cache is much less significant. This is because of the high locality of access demonstrated by the transaction mixes with respect to the L1 data cache and also despite the increase of 27784 Data Cache accesses between transaction mix 1 and transaction mix 4 (shown in Table 6.8), there was only an increase of 93 accesses to memory to get data.

**Table 6-9 Miss Rates for Transaction Mixes under Base Configuration (in Percentages)**

Transaction Mix	TLB Miss Rate	L1 Miss Rate	L2 Miss Rate	PLB Miss Rate
1	.3	1.30	41.55	.91
2	4.69	1.38	49.42	2.38
3	9.58	1.21	58.56	2.43
4	11.27	1.21	68.44	3.97

Table 6.10 shows the total data accesses, the number of shared data accesses, the number of TLB accesses and the number of PLB accesses incurred by each transaction mix. The results are obtained for the base configuration of the simulation environment. The shared data accesses are the reads/writes to the data tables on which views are defined.

**Table 6-10 Data Accesses for Transaction Mixes under Base Configuration**

Transaction Mix	Total Data Accesses (Read/Write)	Shared Data Accesses (Read/Write)	TLB Accesses	PLB Accesses
1	251838	16958	252612	10668
2	267810	18684	280982	9664
3	267470	19085	295818	10539
4	279622	21647	315122	12934

The ratio of PLB accesses to shared data accesses for the transaction mixes is roughly between 52 % - 62 %, for the 4 transaction mixes . This clearly shows that access control in the Multiview model is not invoked on each and every access to the data tables. Recall that PLB access is triggered on the L1 data cache miss or on a write access control fault (where the write is to the line for the first time). However the PLB accesses shown in Table 6.10 are not only attributed to data cache and write access control faults but also to PLB line replacement and to updates to the PLB entries upon access unit state transitions.

## 6.5 Conclusions of Results

The results show that variations in the PLB size, for a given application and hardware parameters, have negligible effect on the read/write lock acquisition delays for a given transaction mix. Wide variations in the L2 data cache size also show small effects on the read/write lock acquisition delay within a transaction mix. It is most decidedly the access characteristics of the applications themselves which have the most impact on the read/write lock acquisition delay. If the application's working set is sufficiently small, resulting in a low TLB miss rate, the read/write lock acquisition delay is smaller when compared to applications with larger working sets, and thus higher TLB miss rates. This is demonstrated by comparison of Table 6.9 with Table 6.7. The largest delay is incurred for the application with the highest TLB miss rate. The TLB component dominates the costs in the memory hierarchy. This is not surprising since the TLB cache has a mere 32 entries; a 32-entry TLB is standard for TLB caches on most platforms.

The traces of the database application show that the number of read and write memory accesses to the data tables compared to data required for the application's execution is very small (see Table 6.10). This further explains why the variation in PLB cache size does not affect the lock acquisition delay appreciably. That is, the loads on the TLB and the data caches have greater impact to application performance than the delay generated from hardware support for the Multiview model. The result thus strengthens the advantage that the Multiview model holds over the conventional software schemes in that TLB and data cache pollution are avoided. It will be particularly important to applications with large working sets.

The results show that the PLB cache can be very small, say 256 bytes: the FSM size is sufficient at 256 bytes and the view definition cache's size is set at 128 bytes to provide for concurrency control views on the data tables. Less than 1Kb of cache memory is required in total for the PCU Unit to support the two phase locking protocol. Chapter 5 illustrates the FSM definitions of other access control protocols. Clearly at four bytes per FSM entry, 64 entries are more than sufficient to support all the protocols described in Chapter 5 without any misses.

Conventional lock management performance will also depend on the application's memory hierarchy costs. However the conventional lock manager will also add its memory hierarchy costs to that of the applications'. In particular, the data cache and TLB cache miss rates will be affected. Those costs are estimated in Chapter 7 and compared to the Multiview memory implementation of lock management.



# CHAPTER 7

## COMPARATIVE EVALUATION

To show how conventional approaches for access control may compare to the Multiview approach, a software scheme for a standard lock manager is implemented (see Appendix B for a program listing). The resulting program is profiled to obtain the average number of instructions for the access control protocol. The average number of calls to key procedures and instructions per procedure are obtained. The main performance metric is that of machine cycles. The instruction counts obtained for the conventional software schemes are converted to cycles for correct comparisons.

The conventional lock manager providing services to transaction mix 1 is memory traced and run through the cache simulator in order to cost the memory hierarchy and hence (conservatively) estimate the delay in cycles to grant a lock request.

The statistics for the setLock() function of the conventional lock manager is obtained using a Sun Sparc 20 platform and GNU's gcc compiler version 2.6.3 with the optimization (-O) directive set.

### 7.1 Software Implementation of a Lock Manager

The lock manager implementation described here is as outlined in [Gottemukkaka 1992, Gray 1993, Daynes 1995]. The lock information is found through named access of a fixed-size hash table of locks. A lock is implemented by a Lock Control Block (LCB) which contains information such as its name, its current mode (R or W), and links [Daynes 1995]. These are illustrated in Figure 7.1.

The LCB is at the head of two doubly linked lists of Lock Request Blocks (LRBs). One list represents granted requests (must at least contain a list of identifiers of transactions which have access to the lock unit), and the other implements the pending requests (the suspend queue where blocked transactions are held). The LRB contains the transaction identity, the requested locking modes, the access unit name and an indication as to which of the two lists it is presently on. The latter facilitates efficiency in the transaction commit or abort procedures. Each LRB is associated with exactly one transaction (requester).

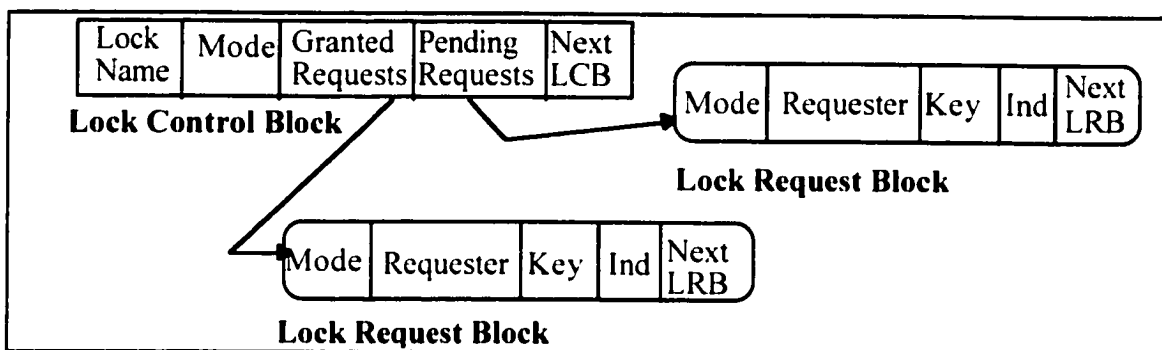


Figure 7-1 Structures of the Lock Control and Lock Request Blocks

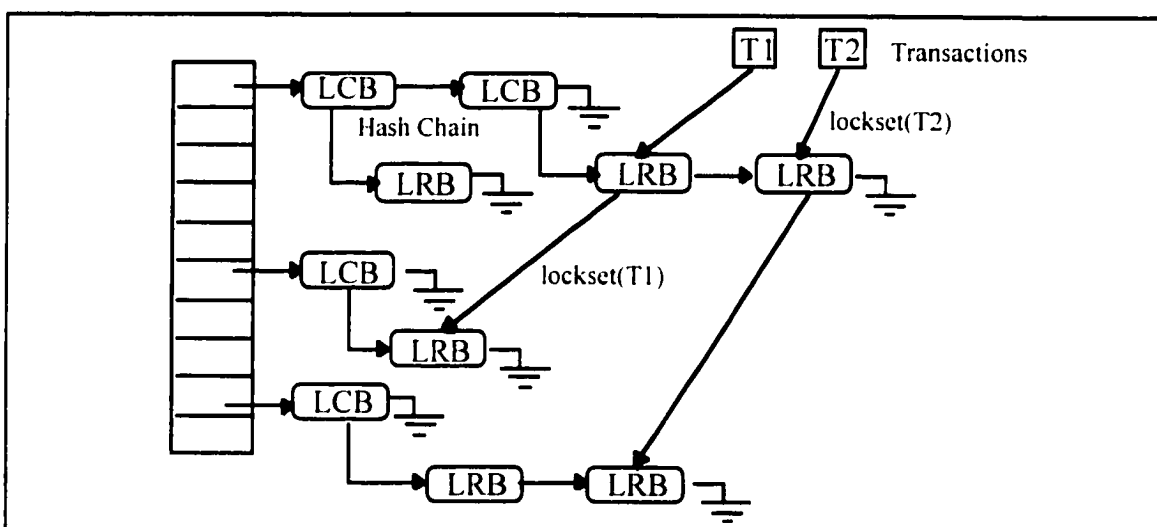


Figure 7-2 Lock Table Implementation

Obtaining a lock involves accessing the lock table (illustrated in Figure 7.2) using the resource's name as the hash key. If there is an LCB entry in the hash chain, the LRB chain for Granted Requests is scanned to determine whether the requester already has a LRB. The latter occurs for lock upgrade requests. When there is no LCB entry, a new entry is initialized and appended to the hash chain. If there is no LRB for the requester, the lock manager allocates a new LRB, chains it to the requester's lockset, and chains it to the appropriate LCB chain (granted or pending) according to whether a conflict is determined or not.

Conflict detection is performed upon a request to set a lock. First the current state of

the access unit is obtained from the LCB entry in the routine which sets the lock. Then the LRB chain of the Granted Requests is scanned (if it exists) to look for either the LRB entry of the requester or the presence of multiple readers, or the identity of the transaction holding a write lock. The latter of course is tested only if the access unit was originally locked for write. Conflicts are detected when there are multiple readers and a write lock is requested, or a single reader other than the transaction requesting the write lock is on the Granted requests list, or a write lock is held by a transaction other than the current requester.

A transaction's lock set is maintained on a linked list whose nodes point to LRB entries on both the Granted Requests and Pending queues. When a transaction commits or aborts, the lock set list is traversed to delete the corresponding entries on the Granted Requests and/or Pending lists.

### 7.1.1 Qualitative Comparison

This section presents qualitative comparisons between the conventional lock manager and the cache supported implementation of the Multiview model. First the advantages of the Multiview implementation are considered and then conflict detection is contrasted. Context switches, lightweight if the lock manager is in the same address space, heavyweight otherwise, are eliminated for setting locks on currently unlocked access units. Recall (see Section 3.3) that the state transitions from Unlocked to Single Reader or Write states (U->SR, U->W) and from SR to Write state (SR->W) and the SR to Multiple Reader (SR->MR) are all handled in the PCU unit. TLB accesses are minimized for the latter cases. Calls to expensive memory allocation routines such as malloc() are reduced in Multiview scheme since the information needed to be kept in software is minimized. For example only the IDs of multiple readers need to be kept on a list, not the ID of a writer since the latter is always stored in the PLB entries.

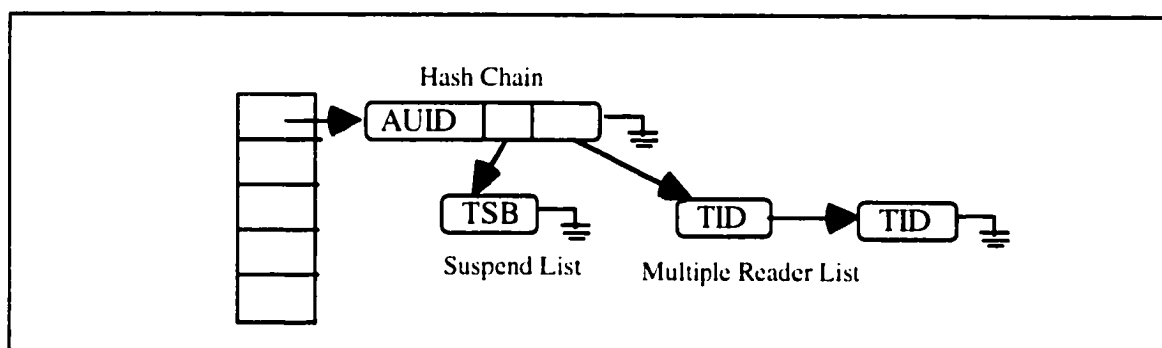
Next conflict detection is contrasted. In the conventional software implementation, the average number of memory accesses to determine a Write-Write (WW) conflict, once the hash table (Lock table) has been indexed, is the average length of the chain of LCBs plus one access to the LRB chain to obtain the ID of the transaction currently holding the lock. The average number of memory accesses needed to determine a Multiple Reader-Writer (MRW) conflict is the same as for the WW conflict; however instead of needing to compare identifiers, the variable holding the length of the LRB list is accessed. The average

number of memory accesses required to determine a Read-Write (RW) conflict is the same as for the WW conflict plus one more access to a Count variable which contains the length of the chain. A lock upgrade from R to W incurs the same average number of memory accesses as in the determination of a RW conflict.

Conflicts and RW upgrades can be ascertained immediately in the Multiview memory model once the state information for the access unit is available in PLB cache. Therefore the PLB cache hit ratio impacts the expected improvement in performance of a Multiview lock management implementation.

### 7.1.2 Software Support for Multiview Lock Management

The main support structure is a hash table of pointers (Figure 7.3) accessed by the access unit identifier (AUID). The hash chain contains nodes with the matching AUID, a pointer to a pending list and a pointer to a list of readers. This structure is only accessed after conflict is detected - not used to determine conflict - it is used to store the TIDs of multiple readers and to implement the transactions' suspend queues. It therefore facilitates the commit / abort transaction procedures. A transaction must be removed from the suspend list if it is aborted. Other transactions are taken off the suspend list and placed on a CPU ready queue when resources they are awaiting are released by a committing/aborting transaction.



**Figure 7-3 Software support for the implementation of Multiview lock management.**

## 7.2 Statistical Results for Software Scheme

Measurements of instruction counts for the software lock manager are obtained using the QPT2 tool. QPT2's output includes a Unix-style flat statistical profile for an input program (see Appendix C). For each procedure within the manager, the number of instructions, the percentage of time spent executing the procedure in ms, the number of calls to the procedure, the number of instructions per call and the routine's name are provided. See Appendix C for the profile listing of the input program. Only totals are presented here. All input programs are compiled with gcc 2.6.3 with and without the optimization flag set. Please refer to Appendix B for a program listing of the lock manager.

**Table 7-1 Statistics for setting a Lock for the Conventional Lock Manager and MultiView**

Function	Instructions (Mustang -SPARC -20 Executed (Unoptimized)	Number of Instructions (Optimized)	Multiview (cycles)
setLock()	102	41	49

The number of cycles for the setLock() function is obtained by using the following equation [Hennessey 1995]: 
$$\text{CPU clock cycles} = \sum_{i=1}^n \text{CPI}_i * \text{IC}_i$$
 where  $\text{CPI}_i$  represents the average number of clock cycles for instruction  $i$  and  $\text{IC}_i$  represents the number of times instruction  $i$  is executed in a program. If we assume that the SPARC-20's CPI is 1 and that each instruction is identical for simplification purposes, then the number of CPU clock cycles =  $1 * 41 = 41$  for the setLock() function. This estimate is low because the memory hierarchy costs are not included yet.

The memory hierarchy costs for the entire lock manager (not solely for setLock()) are obtained from the cache simulator for transaction mix 1. The following table (Table 7.2) shows a costing of the memory hierarchy. This is the minimum memory hierarchy costs since it is obtained from cold caches. That is, no other applications are competing for TLB and data cache space.

As shown in Table 7.2, there are 91037 cycles expended due to delay in the memory hierarchy (TLB, L1, L2 and Main Memory). At 699 calls (shown in Appendix C) to setLock() within Transaction mix 1, it means that on average  $91037/699 = 131$  additional cycles can be added as an estimate to the overhead to satisfy each lock request. Therefore,

roughly  $(41 + 131) = 172$  cycles will be incurred by the conventional lock manager as opposed to the 49 incurred by the Multiview memory model's implementation of lock management for transaction mix 1.

**Table 7-2 Average Cost (delay) in cycles for Transaction Mix 1 - base configuration : TLB capacity = 256 bytes, L1 DC = 8K and L2 DC = 1 Mb**

Access Type - Memory	Number * miss penalty	Total Cost in cycles
L1 Data Cache Misses	5429 * 13	70577
L2 Data Cache Misses	402 * 50	20100
TLB Misses	18 * 20	360
<b>TOTAL</b>	-	<b>91037</b>

**Table 7-3 Data and TLB Accesses for Lock Manager and Multiview for Transaction Mix 1**

Application	Total Data Accesses	TLB Accesses
Lock Manager	322978	322996
Multiview	251838	252612

From Table 7.3, the number of TLB and data cache accesses are greatly increased for the conventional lock manager, thus showing that the data cache and TLB performances decrease in a conventional scheme as opposed to the Multiview model's implementation of the same protocol.

### 7.3 Summary and Conclusions

The Multiview model provides better performance for lock acquisition at an average of 49 cycles as opposed to 172 cycles for the conventional lock manager. This is partially attributed to the reduction in kernel calls; the profile listing of the conventional lock manager (Appendix B) shows that kernel calls for memory allocation are particularly time consuming. Protocol enforcement in the particular implementation of the Multiview model found in this thesis is by simple table lookup rather than by evaluation of multiple clauses in software.

The reduction in kernel calls would be even more evident when compared to a client-server situation which uses explicit requests for access control. For example, a microkernel such as Mach 3.0 would provide an access control policy through user-level implementation, incurring two system calls and two context switches and possibly message delays to satisfy a lock request, for example.

There is less pollution of the data and TLB caches in the Multiview implementation of lock management. The reduction in TLB pollution is an important contribution considering that the TLB is becoming a major bottleneck in the high processor-speed computer systems on the market today.

# CHAPTER 8

## CONCLUSIONS

### 8.1 Contributions

This thesis presents the Multiview memory model which incorporates many desirable features for improvement of application performance. The presented work is new. Specifically, the contributions of the model are as follows:

- Variable-sized protection units are obtained through the use of view definitions without altering the underlying fixed sized paging implementation. This avoids many problems (e.g. modifying page table structures to support multiple page sizes and changes in management of physical memory) encountered by other proposals that architecturally provide variable sized pages.
- Address space isolation is provided within and across address spaces through use of the same view definition mechanism. The only other known scheme for address isolation is called sandboxing [Anderson 1993] and it has the disadvantage of mapping the shared region several times within the same address space for each application running in the address space. Sandboxing is provided by the compiler.
- Customizability of applications is achieved by enabling one or more associations with single purpose access control protocols on a region of memory.
- Access control handling *reduces* the use of software handlers in order to reduce kernel-user application communication. Hence reduced context switching costs are obtained by placing decomposable access control protocols at the hardware and OS level.
- Transparent access control which heightens programmers productivity.

The following contributions are also found in this thesis.

- The wide applicability of the model is illustrated by how several protocols can be decomposed into FSM representations. The model can be used to support a host of different protocols, where the only constraint is that the candidate protocols must be partially or fully decomposable into state transitions.



- Avoidance of affecting the miss rates of the TLB and Data Caches, by storing the decomposed access control protocols in a specialized cache, is attained. Conventional software implementations of access control schemes can cause not only context switching but also increase the TLB and Data Cache miss rates by changing the working set. Recall that the TLB miss rate is the most critical factor affecting applications' performance.
- Development of a simulator to measure memory hierarchy costs for the Multiview model. The logic for the respective cache controllers are all customized. C++ classes developed to model caches and their parameters and to track statistics for the simulation can be reused in the development of other cache architecture simulators. [Jutla 1996].
- Analysis of a multiview implementation of locking and a comparable software implementation of locking.
- Performance estimates for sizing the caches in the proposed cache architecture.

## 8.2 Future Work

Degree of conflict amongst transactions competing for access to shared data should be added as another parameter to the lock acquisition analysis done in this work, particularly since conflict determination is another area where the Multiview model locking implementation supersedes others (see Section 6.1.1). Other access control protocols should also be evaluated for performance purposes.

Traces which contain operating system code should be obtained for input applications, particularly to obtain platform dependent measurements for context switch times.

Kernel support for the Multiview memory model needs to be implemented. OS support such as modifying the Virtual Memory Mapping Manager to align views would be valuable.

Compiler aids such as indicating the beginning and end of a view at the application level should be explored. Simple "Begin Data" and "End Data" statements may suffice. It is intuitive with database applications that the smallest granularity of an access unit should be the record size, but tools for automating the choice of the appropriate access unit sizes for different regions of memory could be developed.

The application of the Multiview Memory Model in the multiprocessor environment is another area for development.

## REFERENCES

- Agarwal 1988;** Agarwal, A., Simoni, R., Hennessy, J., and Horowitz, M., "An evaluation of directory schemes for cache coherence." Proceedings of the 15th Annual International Symposium on Computer Architecture, June 1988, pp 280-289
- Anderson 1991;** Anderson T.E., Levy H.M., Bershad, B.N., Lazowska E.D., "The Interaction of Architecture and Operating System Design." Fourth International Conference on ASPLOS, 1991.
- Appel 1991;** Appel A. , Li K., "Virtual Memory Primitives for User Programs." ACM. 0-89791-380-9/91/0003-0096, 1991.
- Archibald 1985;** Archibald, J. and Baer, J.-L., "An Economic Solution to the Cache Coherence Problem." Proceedings of the 12th Annual International Symposium on Computer Architecture. June 1985, pp 355-362
- Bennett 1990;** Bennett J.K., Carter J.B., Zwaenepoel W., "Adaptive Software Cache Management for Distributed Shared Memory Architectures." 17th Int. Symp. on Computer Architecture, Rice COMP TR90-109, Dept. of Comp. Sc. Report, Rice University, Houston, Texas, 1990.
- Bershad 1995;** Bershad, B.N., Savage, S., Pardyak, P., Sirer, E.G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C., "Extensibility, Safety and Performance in the SPIN Operating System." Technical Report, The University of Washington, 1995.
- Bisiani 1991;** Bisiani, R., Ravishankar, M., Proceedings of the 17th Annual International Symposium on Computer Architecture, 1990, pp 115-124.
- Bodorik 1994a;** Bodorik P. and Jutla D., "Multi-View Access Control Memory Computer System." 22nd Annual ACM Computer Science Conference, Phoenix Arizona.

1994, pp. 241-248.

**Bodorik 1994b**; Bodorik P., Jutla D., Riordon, J.S., "Integrated On-the-fly Access Control," International Conference on Systems Integration (ICSI'94), Sao Paulo, Brazil, August 15-19, 1994.

**Bodorik 1995**; Bodorik, P., Jutla, D.N., Davis A., "A Protection Cache Architecture for the Multiview Memory Model and Its Performance". International Symposium and Workshop on Systems Engineering of Computer Based Systems, Tucson, Arizona, March 6-9, 1995. (IEEE)

**Censier 1978**; Censier, L.M. and Feautrier, P., "A new solution to coherence problems in multicache systems." IEEE Transactions on Computers C-27, 12, December 1978, pp. 1112-1118.

**Chang 1988**; Chang A. and Mergen M.F., "801 Storage: Architecture and Programming." ACM Transactions on Computer Systems, Feb. 1988, Vol. 6, No. 1, pp. 28-50.

**Cheriton 1986**; Cheriton D.R., Slavenburg, G.A., Boyle P.D., "Software -Controlled Caches in the VMP Multiprocessor." Proceedings of the 13th Annual International Symposium on Computer Architecture, 1986.

**Daynes 1995**; Daynes, L., Gruber, O., Valduriez, P., "Locking in OODBMS Client Supporting Nested Transactions", 11th International Conference on Data Engineering, March 1995, pp. 316-323.

**Dubnicki 1992**; Dubnicki C. , LeBlanc, T.J., "Adjustable Block Size Coherent Caches." Proc. of the 19th International Symposium on Computer Architecture, 1992, pp. 170-180.

- Effelsburg 1984;** Effelsburg W., Loomis, M.E.S, Logical, Internal and Physical Reference Behaviour in CODASYL Database Systems, ACM Transactions on Database Systems, Vol 9, No. 2, 1984, pp. 187-213.
- Eggers 1988;** Eggers, S., "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation.." IEEE CH2545-2/88/0000/0373.
- Engler 1995;** Engler E.R., Kaashoek, M.F., O'Toole Jr., J.W., "The Operating System Kernel as a Secure Programmable Machine." OS Review, January 1995.
- Goodman 1983;** Goodman, J.R., "Using cache memory to reduce multiprocessor - memory traffic." Proceedings of the 10th Annual International Symposium on Computer Architecture, June 1983, pp 124-131.
- Goodman 1987;** Goodman, J.R., " Coherency for Multiprocessor Virtual Address Caches," Second International Conference on Architectural Support for Programming Languages and Operating Systems, October 1987, pp. 72-81.
- Gottemukala 1992;** Gottemukala, V., Lehman T., "Locking and Latching in a Memory Resident Database System." Proc. of the Int. Conference on Very Large Data Bases, 1992, pp. 533-544.
- Gray 1993;** Gray J., Reuter, A., Transaction Processing: Concept and Techniques. Morgan Kaufmann, Palo Alto, California, 1993.
- Handy 1993;** Handy, J., The Cache Memory Book, Academic Press Inc., California, 1993.
- Heinrich 1993;** Heinrich , J., MIPS R4000 User's Manual, Prentice Hall, 1993.
- Hennessey 1990;** Hennessey J.L., Patterson D.A., Computer Architecture A Quantitative Approach, Morgan Kaufmann Publishers Inc., Palo Alto, California, 1990.

- Horowitz 1987;** Horowitz M., Chow, P., Stark, D., Simoni, R.T., Salz A., Przybylski,S., Hennessy, J., Gulak, G., Argawal, A., and Acken, J., MIPS-X: A 20 MIPS peak, 32-bit microprocessor with on-chip cache, IEEE of Solid-State Circuits. Vol 22, 1987, pp. 790-799.
- Hosking 1993;** Hosking, A.L., Moss J.E.B., "Protection Traps And Alternatives For Memory Management Of An Object-Oriented Language", ACM SIGOPS'93 0-89791-632-8/93/0012
- Howard 1988;** Howard, J.T., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan M., Sidebotham, R.N., West, M.J., "Scale and Performance in a Distributed File System," ACM Transactions on Computer Systems. Vol. 6, 1 Feb 1988, pp.51-81.
- Jutla 1996;** Jutla D. , Bodorik P., Olmstead E., "An Application FrameWork for Modelling Cache Based Virtual Memory Systems", 8th Annual International Conference on Computing and Information, ICCI'96, Waterloo, Ontario, Canada, June 1996.
- Jutla 1995;** Jutla D.N, Bodorik P., "A Fast Cache-based Method for Concurrency Control", 7th Annual International Conference on Computing and Information, ICCI'95, Trent, Ontario, Canada, July 1995.
- Jutla 1994;** Jutla, D., Bodorik, P., Riordon, J.S., "Access Control Protocols for Multi-View On-the-Fly Computer System." Canadian Conference for Electrical and Computer Engineering, Halifax, September 25-28, 1994.
- Jutla 1993;** Jutla, D., Bodorik, P., Riordon, J.S., "Integrated Concurrency-Coherence Control in Distributed Shared Memory," Fifth International Conference on Computing and Information (ICCI'93), Sudbury, Ontario, Canada, pp 251-255.

- Kagimasa 1991**; Kagimasa, T., Takahashi, K., Mori T., Yoshizumi S., "Adaptive Storage Management for Very Large Virtual/Real Storage Systems." ACM 0-89791-394-9/91/0005/0372
- Kath 1992**; The Virtual Memory Manager in Windows NT, Microsoft Developer Network Technology Group, 1992.
- Katz 1985**; Katz R., Eggers, S., Wood, D.A., Perkins G., Sheldon, R.G.. "Implementing a Cache Consistency Protocol." Proceedings of the 12th International Symposium on Computer Architecture, IEEE, New York, 1985, pp. 276-283.
- Kearns 1989**; Kearns, J.P., DeFazio S., "Diversity in Database Reference Behaviour". Performance Evaluation Review, Vol. 17, No. 1, May 1989, pp. 11-19.
- Koldinger 1991**; Koldinger, E.J., Levy, H.M., Chase, J.S., Eggers, S.J.. "The Protection Lookaside Buffer: Efficient Protection for Single Address-Space Computers." Technical Report 91-11-05. University of Washington, November 1991.
- Koldinger 1992**; Koldinger, E.J., Chase, J.S., Eggers, S.J., "Architectural Support for Single Address Space Operating Systems." ASPLOS V, October 1992.
- Kumar 1989**; Kumar A., Stonebraker M., "Performance Considerations for an Operating System Transaction Manager." IEEE Transactions on Software Engineering, Vol 15. No. 6, June 1989.
- Larus 1995**; Larus J., The QPT Tracing Tool Software Package, Computer Sciences Department, University of Wisconsin-Madison, Madison, Wisconsin, 1995.
- Lee 1980**; Lee P.A., Ghani N., Heron K., "A Recovery Cache for the PDP-11", IEEE Transactions on Computer Systems, June 1980, Vol C-29 pp. 546-549.
- Lee 1989**; Lee, R.B., "Precision Architecture." Computer, January 1989, pp 78- 91.

- Lenoski 1990;** Lenoski D., Laudon J., Stevens L., Joe T., Nakahira D., Gupta A., Hennessey J., "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor.", Proceedings of the 17th International Symposium on Computer Architecture, pps. 148-159, May 1990.
- Leutenegger 1993;** Leutenegger S.T., Dias D., "A Modeling Study of the TPC-C Benchmark", ACM SIGMOD, Washington, DC, 1993.
- Loepere 1992;** Loepere, K., "Mach 3 Kernel Principles." Revision 2. Open Software Foundation and Carnegie Mellon University, 1992
- Mirapuri 1992;** Mirapuri S., Woodacre, M., Vasseghi, N., "The MIPS R4000 Processor." IEEE Micro, April 1992.
- Moon 1987;** Moon D.A., "Symbolic Architecture." 0018-9162/87/0100-0043, IEEE 1987.
- Nelson 1988;** Nelson M.N., Welch B.B., Outerhout J.K., "Caching in the Sprite Network File System," ACM Transactions on Computer Systems, Vol. 6, 1 Feb 1988, pp.134-154
- Nutt 1992;** Nutt G.J., Centralized and Distributed Operating Systems, Prentice Hall, New Jersey, 1992.
- Papamarcos 1984;** Papamarcos M., Patel, J., " A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories." Proceedings of the 12th International Symposium on Computer Architecture, IEEE, New York, 1985, pp.276-283.
- Patterson 1996;** Hennessey J.L., Patterson D.A., Computer Architecture A Quantitative Approach, 2nd Ed., Morgan Kaufmann Publishers Inc., Palo Alto, California, 1996.

- Przybylski 1990**; Przybylski, S.A, Cache and Memory Hierarchy Design, Morgan Kaufmann Publishers, Inc., California, 1990.
- Randell 1969**; Randell B., " A Note on Storage Fragmentation and Program Segmentation." Communications of the ACM, Vol. 12, No. 7, July 1969.
- Reinhardt 1994**; Reinhardt, S.K., Larus, J.R., Wood, D.A., "Tempest and Typhoon: User-Level Shared Memory." 21st Annual Symposium on Computer Architecture, 1994. pp. 325-335.
- Rodriquez-Rosell 1976**; Rodriquez-Rosell, J. "Empirical Data Reference Behaviour in Data Base Systems, IEEE Computer, Vol. 9, No. 11, November 1976, pp. 9-13.
- Romer 1995**; Romer T.H., Olrich W.H., Karlin, A.R., Bershad, B.N., "Reducing TLB and Memory Overhead Using Online Superpage Promotion" ACM International Symposium on Computer Architecture, Italy, 1995.
- Rozier 1992**; Rozier M., Abrassimov V., Armand F., Boule I, Glen M., Guillemont, M., Herrman F., Kaiser C., Langlois, S., Leonard, P., and Neuhauser W., "Chorus Distributed Operating Systems," Computer Systems Journal, 1, 4, pp.305-370
- Rudolph 1985**; Rudolph L., Segall, Z., "Dynamic decentralized cache consistency schemes for MIMD parallel processors." In Proceedings of the 12th Annual International Symposium on Computer Architecture, June 1985, pp 340-347.
- Schimmel 1994**; Schimmel C., Unix Systems for Modern Architectures: Symmetric MultiProcessing and Caching for Modern Programmers. Addison Wesley ,1994
- Slater 1991**; Slater, M., "MIPS Previews 64-bit R4000 Architecture." Microprocessor Report 5, 2.
- Smith 1978**; Smith A.J., "Sequentiality and Prefetching in Database Systems", ACM



transactions on Database Systems, Vol. 3, No. 3, Sept. 1978, pp. 223-247.

**Smith 1982;** Smith A.J., "Cache Memories", Computing Surveys, Vol. 14, No. 3, Sept. 1982, pp. 473-530.

**Stenstrom 1990;** Stenstrom, P., "A Survey of Cache Coherence schemes for Multiprocessors," IEEE Computer, June 1990, pp. 12-24.

**Stonebraker 1984;** Stonebraker M., "Virtual Memory Transaction Management." ACM Operating Systems Review, Vol. 18, No. 2, 1984, pp. 8-16.

**Stonebraker 1985;** Stonebraker M., DuBourdieu D. and Edwards W., "Problems in Supporting DB Transactions in an Operating System Transaction Manager." ACM Operating Systems Review, Vol. 19, No. 1, 1985, pp. 6-14.

**Talluri 1992;** Talluri, M., Kong S., Hill, M.D., Patterson, D.A., "Tradeoffs in Supporting Two Page Sizes." ACM Nineteenth Annual Symposium on Computer Architecture, 1992.

**Tanenbaum 1987;** Tanenbaum A.S., Operating Systems: Design and Implementation, Prentice-Hall, Inc., New Jersey, 1987.

**Tang 1976;** Tang, C.K., "Cache Design in the Tightly Coupled Multiprocessor System." AFIPS Conference Proceedings National Computer Conference, 1976, pp. 749-753

**Thacker 1987;** Thacker C.P., Stewart, L.C., "Firefly: a Multiprocessor Workstation." Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems, October 1987, pp. 164-172.

**Tomasevic 1993;** Thomasevic, M., and Milutinovic, V., "The Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions, Los Alamitos, CA: IEEE Computer Society Press, 1993.

**Transaction Processing Council 1992;** Transaction Processing Performance

Council, "TPC Benchmark C, Standard Specification, Revision 1.0", Edited by Francois Raab, August 13, 1992.

**Wahbe 1992;** Wahbe R., "Efficient Data Checkpoints". In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 200-212, Boston, Massachusetts, Sept. 1992.

**Wahbe 1993,** Wahbe, R., Lucco S, Anderson, T.E., Graham, S., " Efficient Software-Based Fault Isolation" , 14th ACM Symposium on Operating System Principles, December 1993.

**Wheeler 1992;** Wheeler B., Bershad B., " Consistency Management for Virtually Indexed Caches", ACM SIGPLAN Notices, Vol 27, No. 9, September 1992, pp. 124-136.

**Wilkes 1992;** Wilkes J. and Sears B., "A Comparison of Protection Lookaside Buffers and the PA-RISC Protection Architecture", HP Laboratories Technical Report HPL-92-55, March 1992, Hewlett-Packard Company.

**Wulf 1995;** Wulf A., Mckee D., "Hitting the Memory Wall: Implications of the Obvious", Computer Architecture News, 1995.

# Appendix A

## Source Code for Simulator

### Header Files

#### Filename: object.h

```
class Object
{
public:
    virtual ~Object() {}
    virtual void dump( ostream &os ) = 0;

    friend ostream& operator <<(ostream &o, Object &obj);
    friend ostream& operator <<(ostream &o, Object *obj);
};
```

```
#if _DEBUG
#define DbgPrint( x ) x
#else
#define DbgPrint( x )
#endif
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

#### Filename: list.h

```
class Node : public Object
{
public:
    Object *obj;
    Node *next;
    Node *ptr;

    Node(Object *o);
    virtual ~Node() { delete obj; }
    virtual void dump ( ostream &os ) { obj->dump(os); }
};
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
class List : public Object
{
protected:
    Node *list;
    Node *tail;
```



**Filename: hash.h**

```

class HashTable : public Object
{
    List **table;
    int count;
    int max;
    int PRIME;
public:
    HashTable(int n);
    ~HashTable() {}
    List ** getTable() {return table;}
    void setTable(List **t) {table = t;}
    void setPRIME(int n) {PRIME = n;}
    void setmax(int n) {max = n;}
    void setcount(int n) {count = n;}
    int getPRIME() {return PRIME;}
    int getmax() {return max;}
    int getcount() {return count;}
    void dump(ostream &os);
    int hashaddr(int key);
    List *operator[](int);
    void add(int hash, List *l);
    List * access(unsigned int key, int &index);

};

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

**Filename: input.h**

```

class Input : public ifstream
{
    unsigned int addr;
    char at;
    char pad;

public:
    Input(char *name) : ifstream(name) {}
    ~Input();
    void readAccessType() { *this >> at; *this >> pad; }
    void readInst(){int temp; *this >> temp; *this >> pad;}
    void readaddr()      { setf(hex); operator>>(addr); }
    unsigned int getVA() {return addr;}
    int getAccessType() {return at;}

};

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

class ConfigInput : public ifstream
{
public:

```

```

    ConfigInput(char *name) : ifstream(name) {}
    ~ConfigInput();
};
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

**Filename: spec.h**

```

#define      Dread          0
#define      Dwrite        1
#define      CFlush        2
#define      NUMACCESSTYPES 4
#define      NUMMISSTYPES  4
#define      NUMPACCESSTYPES 5
#define      memsize        4096 //4096 4K pages = 16 Mb RAM
#define      numTransInPage 128 // 128 VA->PA translations to a page
#define      LINESIZE       7 // cache line size is set to 2^7 =128 bytes
#define      MAX             (memsize + 1)
#define      lgPAGESIZE     12
#define      WRITEBACK      1
#define      READ           0
#define      WRITE          1
#define      maskZeroP      (0xFFFFFFFF - memsize + 1)
#define      offsetMask     0x00000fff
#define      maskZeroUP     0x000fffff
#define      mplimit        30
#define      timeout        300
#define      AddressSpaceSize 0xffffffff

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// INPUT IN CYCLES //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// The following represent delays in cycles

```

```

#define      DFC            10000000 // number of cycles to retrieve a page from disk
#define      LRU            8 //number of cycles to find a replacement page
#define      PTE            8 //number of cycles to create a page table mapping

#define      DCRead         1 // data cache read hit
#define      DCWrite        2
#define      WControlFault  2 // write access control fault
#define      PLBhit         4
#define      TLBhit         1
#define      FSMhit         1
#define      VThit          1
#define      TLBmiss        20
#define      PLBmiss        13
#define      DCReadMiss     3
#define      DCWriteMiss    4
#define      L2Miss         6
#define      MemoryAccess   13
#define      L2access       2

```

```

#define      CacheTransfer      1      // from DC to VDU; from DC to PLB;
                                                // PLB to FSM or from VDU to PLB
#define      SynonymHandling    1
#define      CSID                1      // comparison of SubjectIDs after PLB
                                                // access and before FSM access

class CacheStats {

    int fetch[NUMACCESSTYPES]; //e.g. fetch[0] holds the number of DC read
                                // accesses
                                // fetch[1] = # of DC write accesses
                                // fetch[2] = # DC cacheline flushes
    int miss[NUMMISSTYPES]; // miss[0] = number of compulsory DC read misses
                                // miss[1] = number of compulsory DC write misses
                                // miss[2] = number of capacity DC read misses
                                // miss[3] = number of capacity DC write misses
    int L2fetch[NUMACCESSTYPES]; //e.g. fetch[0] holds the number of DC read
                                // accesses
                                // fetch[1] = # of DC write accesses
                                // fetch[2] = # DC cacheline flushes
    int L2miss[NUMMISSTYPES]; // miss[0] = number of compulsory DC read misses
                                // miss[1] = number of compulsory DC write misses
                                // miss[2] = number of capacity DC read misses
                                // miss[3] = number of capacity DC write misses
    int sfetch[NUMACCESSTYPES]; // similar to fetch but holds number of accesses
                                // to shared data.
    int smiss[NUMMISSTYPES]; // similar to miss but holds the number of
                                // misses due to shared data.

    int Pfetch[NUMACCESSTYPES]; //Pfetch[0] = # of PLB read accesses
                                // Pfetch[1] = # of PLB write accesses
                                // Pfetch[2] = # PLB cacheline flushes
                                // Pfetch[3] = # accesses due to state transitions
                                // Pfetch[4] = # accesses due to initial load

    int Pmiss[NUMMISSTYPES]; // Pmiss[0] = number of compulsory PLB read
                                // misses
                                // Pmiss[1] = number of compulsory PLB write
                                // misses
                                // Pmiss[2] = number of capacity PLB read misses
                                // Pmiss[3] = number of capacity PLB write misses
    int Tfetch[NUMACCESSTYPES]; //e.g. Tfetch[0] = # of TLB read accesses
                                // Tfetch[1] = # of TLB write accesses
                                // Tfetch[2] = # TLB cacheline flushes
    int Tmiss[NUMMISSTYPES]; // Tmiss[0] = number of compulsory TLB read
                                // misses
                                // Tmiss[1] = number of compulsory TLB write misses
                                // Tmiss[2] = number of capacity TLB read misses
                                // Tmiss[3] = number of capacity TLB write misses
    int PTable[NUMACCESSTYPES]; // PTable[0] = # of page table read accesses

```

```

// PTable[1] = # of page table write accesses
int PLBTable[NUMACCESSTYPES]; // PLBTable[0] = # of PLB table read
// accesses
// PLBTable[1] = # of PLB table write accesses due to WBs
// PLBTable[2] = # of PLB table write accesses due to
// PLB entry creation
int DTable[NUMACCESSTYPES]; // DTable[0] = # of memory data write accesses
// # of memory data read accesses = # DC cache misses
int RWDelay[NUMACCESSTYPES]; // RWDelay[0] = delay due to READ access
// RWDelay[1] = delay due to WRITE access
int FlushLine; // flushes are triggered on a write when synonyms are
// detected or when the view manager memory updates
int FlushCache; // flush is triggered when policy dictates e.g. on a context
// switch for the VDU cache.
int FlushMemWrite; // number of write backs to memory due to cache
// flushes. Differs from the flushline count in that not
// all flushes will cause a write-back e.g. if the target cache line was not dirty.
int InvalidateInTraffic;
int InvalidateOutTraffic;
int InBusTraffic; // total in values
int OutBusTraffic; // total out values
int ICount; // instruction count - can be used to state the percentage of
// data/instr
// accesses within an application.
int delay; // delay in cycles
int hWACFault; // number of write access control faults on the DC cache
int mWACFault; // number of write access control faults on the DC cache
int dcAccess; // total number of L1 data cache accesses
int L2Access; // total number of L2 data cache accesses
int tlbAccess; // total number of TLB cache accesses
int plbAccess; // total number of PLB cache accesses
int fsmAccess; // total number of FSM cache accesses
int vduAccess; // total number of VDU cache accesses
public:
CacheStats();
~CacheStats();
void incrDCAccess (int n) {dcAccess = dcAccess + n;}
void incrPLBAccess (int n) {plbAccess= plbAccess + n;}
void incrTLBAccess (int n) {tlbAccess= tlbAccess + n;}
void incrFSMAccess (int n) {fsmAccess= fsmAccess + n;}
void incrVDUAccess (int n) {vduAccess= vduAccess + n;}
void incrL2Access (int n) {L2Access = L2Access + n;}

void setDelay(int d) {delay = delay + d;}
void sethWACFault(int w) {hWACFault = hWACFault + w;}
void setmWACFault(int w) {mWACFault = mWACFault + w;}
void setICount (int i) {ICount = i;}
void setInBusTraffic (int i) {InBusTraffic = i;}
void setOutBusTraffic (int i) {OutBusTraffic = i;}
void setInvalidateOutTraffic (int i) {InvalidateOutTraffic = i;}
void setInvalidateInTraffic (int i) {InvalidateInTraffic = i;}

```



```

void setFlushMemWrite (int i) {FlushMemWrite = i;}
void setFlushCache (int i) {FlushCache = i;}
void setFlushLine (int i) {FlushLine = i;}

void setMiss(int i, int num) { miss[i] = miss[i] + num;}
void setFetch(int i, int num) { fetch[i] = fetch[i] + num;}
void setL2Miss(int i, int num) { L2miss[i] = L2miss[i] + num;}
void setL2Fetch(int i, int num) { L2fetch[i] = L2fetch[i] + num;}

void setSMiss(int i, int num) {smiss[i] = smiss[i] + num;}
void setSFetch(int i, int num) {sfetch[i] = sfetch[i] + num;}

void setPMiss(int i, int num) {Pmiss[i] = Pmiss[i] + num;}
void setPFetch(int i, int num) {Pfetch[i] = Pfetch[i] + num;}
void setTMiss(int i, int num) {Tmiss[i] = Tmiss[i] + num;}
void setTFetch(int i, int num) {Tfetch[i] = Tfetch[i] + num;}
void setPTable(int i, int num) {PTable[i] = PTable[i] + num;}
void setPLBTable(int i, int num) {PLBTable[i] = PLBTable[i] + num;}
void setDTable(int i, int num) {DTable[i] = DTable[i] + num;}

void setRWDelay(int i, int num) {RWDelay[i] = RWDelay[i] + num;}

int getDelay() { return delay;}
int getDCAccess() { return dcAccess;}
int getPLBAccess() { return plbAccess;}
int getTLBAccess() { return tlbAccess;}
int getFSMAccess() { return fsmAccess;}
int getVDUAccess() { return vduAccess;}
int getL2Access() { return L2Access;}

int gethWACFault() { return hWACFault;}
int getmWACFault() {return mWACFault;}
int getICount () { return ICount;}
int getInBusTraffic () { return InBusTraffic;}
int getOutBusTraffic () { return OutBusTraffic;}
int getInvalidateOutTraffic () { return InvalidateOutTraffic;}
int getInvalidateInTraffic () { return InvalidateInTraffic;}
int getFlushMemWrite () { return FlushMemWrite;}
int getFlushCache () { return FlushCache;}
int getFlushLine () { return FlushLine;}

int *getMiss() { return miss;}
int *getFetch() { return fetch;}
int *getL2Miss() { return L2miss;}
int *getL2Fetch() { return L2fetch;}

int *getSMiss() { return smiss;}

```



```

void setCoherencePolicy (int cpolicy) {policies.coherence = cpolicy;}

int getTagType() {return tagtype;}
int getTagWidth() {return tagwidth;}
int getBlockSize() {return subblocksize;}
int getTransferSize() {return transfersize;}
int getAssoc() {return assoc;}
int getNumISets() {return numISets;}
int getNumUorDsets() {return numUorDsets;}
int getPrefetchDisp() {return pdisplacement;}
int getFetchPolicy() {return policies.fetch;}
int getReplacePolicy() {return policies.replacement;}
int getWritePolicy() {return policies.write;}
int getAllocatePolicy() {return policies.allocate;}
int getCoherencePolicy() {return policies.coherence;}
};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
class LineControlBits : public Object
{
    int tag;           // primary tag
    int tag1;          // accommodates another tag such as VFN
    int valid;         // valid bit
    int dirty;         // write bit
    int reference;     // only used here for prefetching. Demand fetching implies
                    // that once the line is read in, it is implicitly referenced.

public:
    LineControlBits();
    ~LineControlBits();
    void dump( ostream &os );
    void setTag(int t) {tag = t;}
    void setValidBit(int v) {valid = v;}
    void setDirtyBit(int d) {dirty = d;}
    void setRefBit(int r) {reference = r;}
    void setTag1(int v) {tag1 = v;}
    int getTag() {return tag;}
    int getValidBit() {return valid;}
    int getDirtyBit() {return dirty;}
    int getRefBit() {return reference;}
    int getTag1() {return tag1;}
};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
class DataCacheLine: public Object // This class is not needed now but is
// made into a separate class in case R/W data
// is needed to be kept in the future.
{
    LineControlBits *LineBits; // reps. what is kept in the data cache
};

```



```

    int pa;
    unsigned int vaddr;
public:
    Mapping();
    ~Mapping();
    unsigned int getVA(){return va;}
    unsigned int getVAD(){return vaddr;}
    int getPA() {return pa;}

    void setVA(unsigned int n) {va = n;}
    void setPA(int n) {pa = n;}
    void setVAD(unsigned int n) {vaddr = n;}
    void dump( ostream &os );
};

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

class TLBLine
{
    int ASID;
    Mapping *mp;
    LineControlBits *LineBits;
public:
    TLBLine();
    ~TLBLine();
    void setASID(int ID){ASID = ID;}
    void setMP(Mapping *m){mp = m;}
    int getASID(){return ASID;}
    int getPA(){if (mp!=0) return mp->getPA();else return MAX;}
    unsigned int getVA(){ if (mp!=0) return mp->getVA();else return 0;}
    Mapping * getMP() {return mp;}
    LineControlBits * getLineBits() {return LineBits;}
    void setLineBits(LineControlBits *b) {LineBits = b;}
};

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

class TLB
{ // Cache contents
    Cache *cache;
    int LineAddress;
    TLBLine *Line;
    int numlines;
    int status;
public:
    TLB(int size);
    ~TLB();
    int getCacheSize();
    void setCacheSize(int SIZE);
    Cache * getCache() {return cache;}
    int getTranslation(unsigned int VA);

```





```

void MemToCache(PLBLine *line);
int findSynonyms(int PA);
List **getPLBLine() {return plbline;}
void writeCache (PLBLine *c,int newstate,int othernewstate);
void dump(ostream & os);
void replace(int index,PLBTable *pt,CacheStats *&st);
};

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

class FSMLine
{
    int FSMID;
    int accesstype;
    int CurrentState;
    int OtherCurrentState;
    int NewState;
    int OtherNewState;
    int Result;
public:
    void setFSMID(int ID) {FSMID = ID;}
    void setAccessType(int ac) {accesstype = ac;}
    void setCurrentState(int state){CurrentState = state;}
    void setOtherCurrentState(int ostate) {OtherCurrentState = ostate;}
    void setResult(int r) {Result = r;}
    void setNewState(int nstate){NewState = nstate;}
    void setOtherNewState(int onstate) {OtherNewState = onstate;}
    int getFSMID() { return FSMID;}
    int getAccessType() {return accesstype;}
    int getCurrentState() {return CurrentState;}
    int getOtherCurrentState() {return OtherCurrentState;}
    int getNewState() {return NewState;}
    int getOtherNewState() {return OtherNewState;}
    int getResult() {return Result;}
};

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

class FSM
{
    Cache *cache;
    int LineAddress;
    int numlines;
public:
    struct CacheLine
    {
        LineControlBits LineBits;
        FSMLine LineContents;
    };
    CacheLine *fsmline; // pointer to the start address of the actual cache lines
    CacheLine *cline;

```





```

    CacheLine *cline;
    int numlines;
public:
    VDU(int s);
    ~VDU();
    int getCacheSize();
    void SetCacheSize(int SIZE);
    Cache * getCache() { return cache; }
    void indexVDUCache(unsigned int v, int &mask, int &ViewID, int &notfound, int
                        &FSMID);
    int lookup(unsigned int vaddr, int &vmask, int &vstart);
    VDULine *getVDULine() { return cline->contents; }
    CacheLine *getCurrentLine() { return cline; }
    void setCurrentLine(CacheLine *c) { cline = c; }
    int calcMask(int unitsize);
};

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

template <class T>
class Stack: public Object
{
    int size;
    T *top;
    T * s;
public:
    Stack(int sz) { top = s = new T[size=sz]; }
    ~Stack() { delete[] s; }
    void push (T t) { *top++ = t; }
    T pop()      { return *--top; }
    void initPM();
    void dump( ostream &os );
    int *getTop() { return top; }
    int *getStackPointer() { return s; }
};

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

template <class T>
void
Stack<T>::dump
(
    ostream &os
)
{
    int i = 0;
    T *p = top;

    os << "Stack contents:" << endl;

    while ( p > s )
    {

```

```

        os << "\t" << "Entry " << i++ << " " << *--p << endl;
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
template <class T>
void
Stack<T>::initPM()
{
    int i;
    for (i=0; i < size; i++)
        push(i);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
template <class T> class Memory : public Object
{
    T* p;
    int sz;
public:
    Memory(int sz);
    ~Memory() { delete []p;}

    void dump( ostream &os );
    //virtual T& read(int location,int nbytes,int offset);
    virtual void write(T x,int location);
};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
template <class T>
Memory<T>::Memory
(
    int s
)
{
    if (s<=0) cerr << "memory size cannot<=0" << endl;
    sz=s;
    p=new T[s];

    for (int i = 0; i < sz; i++)
        p[i] = 0;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

template <class T>
void
Memory<T>::write(T x,int location)
{

```













**Filename: cpu.cpp**

```

#include <iostream.h>
#include <fstream.h>
#include "object.h"
#include "list.h"
#include "hash.h"
#include "input.h"
#include "trans.h"
#include "spec.h"

////////////////////////////////////
CPU::CPU
(
int dcsize,
int l2size,
int tlbsize,
int plbsize,
int fsmsize,
int vdusize,
int ptsize,
int phtablesize
)
{
    dc = new DataCache(dcsize);
    l2cache = new DataCache(l2size);
    tlb = new TLB(tlbsize);
    plb = new PLB(plbsize);
    fsm = new FSM(fsmsize);
    vdu = new VDU(vdusize);
    Stats = new CacheStats;
    PT = new PageTable(ptsiz,0);
    memPLB = new PLBTable(phtablesize);
    PMFreeList = new Stack<int>(memsize);
    TransLog = new HashTable(mplimit);
    errfile.open("error.dat", ios::beg);
    // set up cache parameters
    // dc->cache->setAssociativity(getSim()->getip()->getDCAssociativity());
    //etc.
    // below must be changed - a shortcut
    dc->getCache()->setAssociativity(2);
    l2cache->getCache()->setAssociativity(2);
    plb->getCache()->setAssociativity(2);
    fsm->getCache()->setAssociativity(1);
    vdu->getCache()->setAssociativity(1);
    tlb->getCache()->setAssociativity(1);

    dc->getCache()->setSubBlockSize(128);
    l2cache->getCache()->setSubBlockSize(128);
    plb->getCache()->setSubBlockSize(8);
    tlb->getCache()->setSubBlockSize(8);

```

```

fsm->getCache()->setSubBlockSize(8);
vdu->getCache()->setSubBlockSize(8);

dc->getCache()->setWritePolicy(0);           // copyback
l2cache->getCache()->setWritePolicy(0);     // copyback
plb->getCache()->setWritePolicy(0);
tlb->getCache()->setWritePolicy(0);
fsm->getCache()->setWritePolicy(0);
vdu->getCache()->setWritePolicy(0);

dc->getCache()->setReplacePolicy(0);        // LRU
l2cache->getCache()->setReplacePolicy(0);   // LRU
plb->getCache()->setReplacePolicy(0);
tlb->getCache()->setReplacePolicy(0);
fsm->getCache()->setReplacePolicy(0);
vdu->getCache()->setReplacePolicy(0);

dc->getCache()->setAllocatePolicy(1);       // write allocate
l2cache->getCache()->setAllocatePolicy(1); // write allocate
plb->getCache()->setAllocatePolicy(1);
tlb->getCache()->setAllocatePolicy(1);
fsm->getCache()->setAllocatePolicy(1);
vdu->getCache()->setAllocatePolicy(1);
}
/////////////////////////////////////////////////////////////////
CPU::~CPU()
{
}
/////////////////////////////////////////////////////////////////
unsigned int CPU::getDCVA(unsigned int v)
{
    unsigned int mask = 0x01ffffff;        //zeroise the upper 7 bits
    unsigned int DCVA;

    DCVA = v >> LINESIZE;
    DCVA = DCVA & mask;

    return DCVA;
}
/////////////////////////////////////////////////////////////////
unsigned int CPU::getVFN(unsigned int v)
{
    unsigned int mask = 0x000fffff;
    unsigned int VFN;

    VFN = v >> lgPAGESIZE;
    VFN = VFN & mask;

    return VFN;
}

```

```

////////////////////////////////////
// Initial assumption - no synonyms. Will flush simulator after
// each trace file is processed. A trace file may contain many
// transactions. The xns are all in the same address space.
//
// The PCU access (VDU,PLB and FSM) is triggered on a R/W miss and
// on a write access control fault
//
int CPU::accessDC(unsigned int VAddr,int accesstype, int &pa, int &mr, ofstream &lm,
                 int &vdef, int &unitID, int &maskedPA)
{
    int PFN;
    int offset, oset, omask;
    int ptag;
    int cs,ocs;
    int FSMID;
    int ns, ons;
    int viewstart, viewmask;
    PLBLine *pb;
    unsigned int dcva; // used to store the va >> linesize
    int setfull=0;
    int sfull = 0;
    int proceed = 0;
    List **c;
    Node *n;
    DataCacheLine *p, *lp; //contains pointer to cache line
    int index, dex;
    int action = 0; //action !=0 implies the process will suspend
    int dcPA;

    Stats->incrDCAccess(1);
    Stats->setFetch(accesstype,1); // increments the type of DC access (R/W)
    dcva = getDCVA(VAddr);
    index = dcva % (dc->getCache()->getNumUorDsets() - 1);
    vdef = vdu->lookup(VAddr,viewmask,viewstart);
    unitID = dcva; // for non-controlled access
    if (vdef)
    {
        Stats->incrVDUAccess(1);
        Stats->setDelay(VThit);
        Stats->setRWDelay(accesstype,VThit);
        Stats->setSFetch(accesstype,1); // adds to the number of r/w shared
        // accesses
        unitID = ((VAddr - viewstart)/(AddressSpaceSize - viewmask +
        1))*(AddressSpaceSize - viewmask + 1) + viewstart;
        Stats->setDelay(CacheTransfer); // from VDU to PLB
        Stats->setRWDelay(accesstype,CacheTransfer);
    }
    dc->indexDCCache(VAddr,p,index,setfull,SID,vdef,unitID);
    PFN = accessTLB(VAddr,accesstype); // get translation - phys.frame no.
    pa = (PFN << lgPAGESIZE) & maskZeroP;
}

```

```

offset = VAddr & offsetMask;
pa = pa | offset; // constructs full physical address
dcPA = getDCVA(pa);

if (vdef)
{
    omask = AddressSpaceSize - viewmask;
    oset = unitID & offsetMask;
    maskedPA = (pa & maskZeroP) | oset;
}

if (p!=0) // data cache set contains entries
{
    ptag = p->getLine()->getTag();
    if (ptag == dcPA) // && (p->getPDID() == getSID())
// compare tag with translation from TLB
    {
        // a hit in the data cache
        if (vdef) // view is defined on address
        {
            if (accesstype == READ)
            {
                Stats->setDelay(DCRead); // add delay of 1 cycle
                Stats->setRWDelay(accesstype,DCRead);
            }
            else
            {
                Stats->setDelay(DCWrite); // add delay of 2 cycles
                Stats->setDelay(1); // delay due to comparison of
                // the W bit
                Stats->setRWDelay(accesstype,(DCWrite + 1));
            }
        }

        if (((p->getLine()->getDirtyBit()==0) && (accesstype ==
WRITE)) || (accesstype == READ))
        {
// The check for read access is present here because access units are not presently
// aligned.. (as generated from QPT2) This means that shared and
// non-shared data can coexist in a data cache line. Thus first read to a
// shared access unit does not necessarily have to result in a miss, if access
// to non-shared data brought part of (or whole) unit in the data cache. I compensate
// here by invoking the PLB on a READ as well even on a data cache hit but do not
// increment the relevant stats for it.
// find or place PLB state info in the PLB cache
pb = accessPLB(VAddr,unitID,accesstype,cs,ocs,FSMID);
if (accesstype == WRITE)
    Stats->setHWACFault(1); // Write access
    control fault
Stats->setDelay(CSID);
Stats->setDelay(CacheTransfer); // bet. PLB and FSM
Stats->setRWDelay(accesstype,(CSID + CacheTransfer));
accessFSM(FSMID,accesstype,cs,ocs,proceed,ns,ons);

```

```

if (proceed == 1)
{
if ((cs == 0) || (cs == 3) || (cs == 1) || ((cs == 4) &&
(pb->getPDID() == getSID())))
{
    if (cs == 0)
    {
        pb->setPDID(SID); // first
        // time lock acquisition
        p->setPDID(SID);
    }
    if ((accesstype == READ) && (ns == 3))
    {
        if (pb->getPDID() != SID)
        {
            mr = 1;
        }
        else // repeated read by the
        same xn
        {
            if (cs == 1) // a single
            reader's repeated read
            {
                ns = 1;
                ons = 2;
                Stats->setPFetch(3,-1);
                Stats->setPFetch(accesstype,-1);
                Stats->incrPLBAccess(-2);
                Stats->setDelay(-(CacheTransfer));
                Stats->setRWDelay(accesstype.(CacheTransfer));
            }
        }
    }
    plb->writeCache(pb,ns,ons); // write back to PLB entry
    Stats->incrPLBAccess(1);
    Stats->setPFetch(3,1); // state transition and
Stats->setRWDelay(accesstype,CacheTransfer);
    Stats->setDelay(CacheTransfer);
    pb->getLineBits()->setDirtyBit(1);
}
    else proceed = 0; // the check for matching
subject IDs is done afterwards
}
if (proceed)
{
    if (accesstype == WRITE)
        dc->writeCache(2); // sets the dirty bit
        and ref bit
    else
        dc->writeCache(0);
}
}

```

```

        else
        {
            action = 1; // write failed
            return action;
        }
    }
    else // dirty bit is set
    {
        if ((SID != p->getPDID()) && (accesstype == WRITE))
        {
            action = 1;
            return action; // locked by another reader/writer
        }
    }

    //record shared access in input file to the conventional LM
    if (accesstype == WRITE)
        lm << "\t\t\tLocktable->setLock(0x" << hex <<
        unitID <<" ,WRITE,t[" << SID <<"]);" << endl;
    else
        lm << "\t\t\tLocktable->setLock(0x" << hex <<
        unitID <<" ,READ,t[" << SID <<"]);" << endl;
    }
    else // no view is defined on the address
    {
        if (accesstype == 1)
            dc->writeCache(2);
        else
            dc->writeCache(accesstype); // accesstype = 0
    }
}

if ((p==0) || ((p!=0) && (p->getLine()->getTag() != dcPA))) // no dc entries or
// match not found on tag comparison
// - a MISS
{
    // now obtain replacement line or add a line to a set
    Stats->setMiss(accesstype,1);
    if (accesstype == READ)
    {
        Stats->setDelay(DCReadMiss);
        Stats->setRWDelay(accesstype,DCReadMiss);
    }
    else
    {
        Stats->setDelay(DCWriteMiss);
        Stats->setRWDelay(accesstype,DCWriteMiss);
    }
}
}

```

```

p = dc->createDCLine(index,VAddr,dcPA,SID,unitID); // adds line to 1st
                                                    // position in a set
if (setfull == 1)
{
    dc->replace(index,Stats,accesstype,0); // choose which line in the
                                           // set is to be replaced
    // remember to add delay for line replacement in the dc.replace
    // routine
}
dex = dcva % (l2cache->getCache()->getNumUorDsets() - 1);
l2cache->indexDCCache(VAddr,lp,dex,sfull,SID,vdef,unitID);
Stats->incrL2Access(1);
Stats->setL2Fetch(accesstype,1);
if (lp == 0) // secondary data cache miss
{
    Stats->setL2Miss(accesstype,1);
    Stats->setDelay(MemoryAccess);
    Stats->setRWDelay(accesstype,MemoryAccess);
    if (accesstype == READ)
    {
        Stats->setDelay(L2Miss);
        Stats->setRWDelay(accesstype,L2Miss);
    }
    else
    {
        Stats->setDelay(L2Miss);
        Stats->setRWDelay(accesstype,L2Miss);
    }
    lp = l2cache->createDCLine(dex,VAddr,dcPA,SID,unitID); // adds
    line to 1st position in a set
    if (sfull == 1)
    {
        l2cache->replace(dex,Stats,accesstype,1); // choose which
        line in the set is to be replaced
    }
}
if (vdef)
{
    Stats->setSMiss(accesstype,1); // to SHARED data
    pb = accessPLB(VAddr,unitID,accesstype,cs.ocs.FSMID);
    Stats->setDelay(CSID);
    Stats->setDelay(CacheTransfer);
    Stats->setRWDelay(accesstype,(CSID + CacheTransfer));
    accessFSM(FSMID,accesstype,cs.ocs,proceed,ns.ons);
    if (proceed == 1)
    {
        if ((accesstype == READ) || ((accesstype ==
        WRITE)&&((pb->getPDID() == getSID()
        || (pb->getPDID() == 0))))

```

```

{
// for a write we test whether the cs is UNLOCKED (cs = 0)
no SID) or LOCKED (4)

    if (cs == 0) // either for single reader or first
        writer
    {
        pb->setPDID(SID);
        p->setPDID(SID);
    }

    if ((accesstype == 0) && (ns == 3))
    {
        if (pb->getPDID() != SID)
        {
            mr = 1;
        }
        else // repeated read by the same xn
        {
            if (cs == 1) // a single r
            reader's repeated read
            {
                ns = 1;
                ons = 2;
            }
        }
    }

}
if ((accesstype == READ) && (cs==4) &&
(pb->getPDID()!=SID))
    proceed = 0; // unit is locked for
write by another transaction
else
{
    plb->writeCache(pb,ns,ons); // write
    // back to PLB entry
    Stats->incrPLBAccess(1);
    Stats->setPFetch(3,1);
    Stats->setRWDelay (accesstype,
        CacheTransfer);
    Stats->setDelay(CacheTransfer);
    pb->getLineBits()->setDirtyBit(1);
}
}
else proceed = 0; // the check for matching subject IDs is
// done afterwards
}
if (proceed)
    dc->writeCache(accesstype);
else

```



```

    {
        Stats->setmWACFault(1);
        action = 1; //suspend process
        return action;
    }
    if (accesstype == 0)
        lm << "\t\t\tLocktable->setLock(0x" << hex << unitID <<
            ",READ,t[" << SID <<"]);" << endl;
    else
        lm << "\t\t\tLocktable->setLock(0x" << hex << unitID <<
            ",WRITE,t[" << SID <<"]);" << endl;
    }
    else
    {
        dc->writeCache(accesstype);
        Stats->setDelay(1);
        Stats->setRWDelay(accesstype,1);
    }
}
// access has been succesfully made at this point - run LRU and "touch" routines
c = dc->getLine();
// can test for whether a line was not created but only accessed later on
// to avoid unnecessary removes and adds
if (c[index]->getCount() > 1) // List must have more than one item
{
    n = c[index]->remove((Object *) p); // LRU on access
    c[index]->push((Object *) p, n); // removes node from position in list and
    // adds it back to the top of the list.
}
return action;
}
/////////////////////////////////////////////////////////////////
int CPU::accessTLB(unsigned int va, int accesstype)
{
    int PFN;
    int VFN;

    Stats->incrTLBAccess(1);
    Stats->setTFetch(accesstype,1); // incoming access due to either a R / W
    VFN = getVFN(va);
    PFN = tlb->getTranslation(VFN);
    if (PFN==MAX) //mapping not in TLB
    {
        Stats->setTMiss(accesstype,1);
        Stats->setDelay(TLBmiss); //delay to lookup page tables
        Stats->setRWDelay(accesstype,TLBmiss);
        PFN = PT->getTranslation(VFN); //find mapping in the PAGE
        TABLE
        Stats->setPTable(0,1); // read access to page table
        Stats->setDelay(MemoryAccess); // taken care in the TLB Miss
        if (PFN==MAX) // IF mapping is not found here then we may have
        // a "soft" fault. We need

```

```

// to determine whether the page is actually in memory. Then we will
//only have to create a mapping for it.
// ASSUMPTION:IGNORE SOFT FAULTS!
// Note: this if stmt. is not entered in this version of the simulator
// because the
// page table has been previously loaded.
{
    //fault to disk.
    // PFN=memory.replace();
    Stats->setDelay(LRU); //expired cycles due to finding a page
        // to replace
    Stats->setDelay(DFC); //cycles due to disk fault
    Stats->setDelay(PTE); //cycles due to creating page table
        entry
    Stats->setPTable(1,1); // write access to page table
    Stats->setDelay(MemoryAccess);
    PT->PTwrite(va,PFN); //place mapping in page table
}
tlb->createMapping(VFN,PFN); //place mapping in TLB.
Stats->incrTLBAccess(1);
Stats->setTFetch(3.1); // write access - tlb load
}
else // tlb hit
{
    Stats->setDelay(TLBhit);
    Stats->setRWDelay(accesstype,TLBhit);
}
return (PFN);
}
/////////////////////////////////////////////////////////////////

PLBLine * CPU::accessPLB(unsigned int VA, int uPA, int accesstype, int &cs, int &ocs,
int &FSMID)
{
    int res=0;
    int mask;
    int index;
    int notFound;
    int ViewID;
    int PDID;
    List ** p;
    PLBLine *pb = 0;
    PLBLine *pl;
    LineControlBits *c;
    int i=1;

    Stats->setPFetch(accesstype,1); // increments the # of PLB accesses of one type
    Stats->incrPLBAccess(1);
    accessVDU(VA,ViewID,FSMID,mask,notFound);
    Stats->setRWDelay(accesstype,VThit);
    if (notFound)

```

```

{
    cerr << "unable to locate view information" << endl;
    // suspend process ?
    res = 2; //error
}
else
{
    PA = PA & mask;           //use masked address to index cache
    index = uPA % (plb->getCache()->getNumUorDsets() - 1); // a number
                                                other than a power of two
    p = plb->getPLBLine();
    if ((p[index] == 0) || ((pl = (PLBLine *) p[index]->find(uPA, ViewID, i))
        == 0))
    {
        Stats->setPMiss(accesstype,1);
        Stats->setDelay(PLBmiss);
        Stats->setRWDelay(accesstype,PLBmiss);

        PDID = getSID();
        pb = memPLB->plbMissHandler (uPA,ViewID,cs,ocs,PDID .0,
            Stats ) ; // action = 0 (read)
        Stats->incrPLBAccess(1);
        Stats->setPFetch(4,1);           // load PLB cache

        pl = new PLBLine;
        pl->setPDID(pb->getPDID());
        pl->setCurrentState(pb->getCurrentState());
        pl->setOtherCurrentState(pb->getOtherCurrentState());
        pl->setViewID(pb->getViewID());
        c = new LineControlBits;
        c->setDirtyBit(pb->getLineBits()->getDirtyBit());
        c->setRefBit(pb->getLineBits()->getRefBit());
        c->setValidBit(pb->getLineBits()->getValidBit());
        c->setTag(uPA);
        pl->setLineBits(c);
        if (p[index] == 0)           // empty set - place line in plb
        {
            p[index] = new List;
            p[index]->push((Object *) pl, 0);
        }
        else
        {
            p[index]->push((Object *) pl, 0);
            if (p[index]->getCount() > plb->getCache()->getAssoc())
                plb->replace(index,memPLB,Stats);
        }
    }
}
else           // simply access
{
    Stats->setDelay(PLBhit);
}

```

```

Stats->setRWDelay(accesstype,PLBhit);

cs = pl->getCurrentState();
ocs = pl->getOtherCurrentState();

if (p[index]->getCount() > 1) // do LRU on access
{
    Node *n = p[index]->remove((Object *) pl);
    p[index]->push((Object *)pl,n);
}

}
// get PLB entry; send to FSM unit for comparison // set cs and ocs parameters etc.
}

return (pl);

}
/////////////////////////////////////////////////////////////////
void
CPU::accessVDU
(
    unsigned int va,           // Virtual address
    int &ViewID,
    int &fsmid, // FSM identifier
    int &mask, // Address Mask
    int &nfound
)
{
    Stats->incrVDUAccess(1);
    Stats->setDelay(VThit);
    vdu->indexVDUCache(va, mask, ViewID, nfound, fsmid);
    // usage of viewid is not implemented as yet since attention
    // is being restricted to the SINGLE VIEW SYSTEM
}
/////////////////////////////////////////////////////////////////
void CPU::accessFSM(int fsmid,int accesstype,int cs,int ocs,int &signal,int &ns,int
&ons)
{
    FSM::CacheLine *f;

    Stats->incrFSMAccess(1);
    fsm->indexFSM(fsmid,accesstype,cs,ocs,f);
    if (f!=0) // this is expected under our current assumption of a no-miss FSM cache
    {
        ns = f->LineContents.getNewState();
        ons = f->LineContents.getOtherNewState();
        signal = f->LineContents.getResult();
    }
    else

```

```

        {
            signal = 99;
            cout << "fault on FSM cache" << endl;
        }
    }
}
////////////////////////////////////////////////////////////////

```

**Filename: system.cpp**

```

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>

#include "object.h"
#include "list.h"
#include "hash.h"
#include "input.h"
#include "trans.h"
#include "spec.h"

int loading = 1;
#define DEBUG_INTERVAL 1024
////////////////////////////////////////////////////////////////
System::System(char *config, char *tracefile)
{
    int dcsize = 32;
    int l2size = 1024;
    int tlbsize = 32;
    int plbsize = 512;
    int fsmsize = 32;
    int vdusize = 16;
    int psize = 1023;
    int phtablesize = 1023;

    // dcsize = Config.getDCSize();
    // tlbsize = Config.getTLBSize();
    // plbsize = Config.getPLBSize();

    ip = new ConfigInput(config);
    tracefp = new Input(tracefile);
    AUtable = new HashTable(1023);
    if ( AUtable == 0)
    {
        cerr << "Unable to create lock data structure" << endl;
    }
    ReadyQueue = new List;
    MPL = new List;
    cpu = new CPU(dcsize,l2size,tlbsize,plbsize,fsmsize,vdusize,psize,phtablesize);
}

```

```

}
////////////////////////////////////
System::~~System()
{
    ltrace << "End" << endl;
    ltrace.close();
}
////////////////////////////////////
void System::dump(ostream &os)
{
    os << "Printing the CPU ReadyQueue\n" << ReadyQueue << "\n";
    os << "Printing the MPL List: \n" << MPL << "\n";
}

////////////////////////////////////
Transaction * System::TransStartup(int id)
//
// processes a BOT from the trace file. If the transaction id is new, a
// transaction is instantiated. If not, the transaction ready queue is checked
// to determine whether the transaction can proceed. If it can, the file pointer
// in the tracefile is moved to the appropriate position.
{
    Transaction *t;

    dcount = 0;
    if (MPL->getCount() < mplimit)
    {
        if ((MPL->getCount() == 0) || ((MPL->findtrans(id) == 0) &&
(ReadyQueue->getCount()==0)))
        {
            t = new Transaction(id.tracefp);
            MPL->add(t);
            xn = t;
            setSID(VA); // VA here is not a virtual address but the transaction
                        identifier
            cpu->setSID(VA);

        }
        else // transaction already started
        {
            contextSwitch();
            t = xn;
        }
    }
    else
    {
        cout << "Processes at MPL limit\n";
        t = 0;
    }
    return t;
}

```

```

}
////////////////////////////////////////////////////////////////
void System::start()
{
    long mark;
    ofstream tp;
    int i = 0;

    dcount = 0;
    ltrace.open("lmain.cpp",ios::ate); // opens an existing file and seeks to its end
    xnOrder.open("tranorder.dat",ios::ate);

    if ( tracefp->is_open() && ltrace.is_open() )
    {
        tracefp->clear();
        tracefp->seekg(0,ios::beg); // reset file ptr
        cpu->setSID(0); //will have to set SID on each context
            // switch
        setASID(0);
        xnprocessing = 0;
        tp.open("mark.dat");
        tp << "Printing Stored File Pointers" << endl;
        while ( !tracefp->eof() )
        {
            mark = tracefp->tellg(); // mark file ptr
            getNextAddress();
            if ((AT==0) || (AT == 1))
                invokeCPU();
        }
        tracefp->close();
        tp.close();
        ltrace << "\t\t} \n\t}\n } \n";
        ltrace.close();
        xnOrder.close();
        printTables();
        printResults();
    }
}
////////////////////////////////////////////////////////////////
void System::suspend(Mapping *mp)
{
    TSB *b;
    List *p, *q;
    int index;
    AUE *a;
    int mPA;

    b = saveContext(xn);
    mPA = mp->getPA();
    p = AUtable->access(mPA,index);

```

```

if ((p == 0) || ((a = (AUE *) p->findAU(mp)) == 0))
{
    p = new List;
    ATable->add(index,p);
    a = new AUE;
    q = new List;
    a->setQ(q);
    a->setReaders(0);
    a->setAUID(mp);
    p->add(a);
}
else
{
    q = a->getQ();
}
if (q == 0)
{
    q = new List;
    a->setQ(q);
}
q->add(b);
}
////////////////////////////////////
TSB * System::saveContext(Transaction * tran)
{
    TSB *t;
    int tid;
    long mark;

    t = new TSB;
    tid = tran->getID();
    mark = tracefp->tellg();
    t->setkey(tid);
    t->setTID(tran);
    t->setMark(mark);
    return t;
}
////////////////////////////////////
List * System::offSuspendQ(Transaction *tn,AUE *a)
{
    List *q;
    TSB *t;
    Node *n;

    if (a != 0)
    {
        q = a->getQ();
        if (q != 0)
        {
            if ((t = (TSB *) q->findTSB(tn)) != 0)

```



```

        {
            n = q->remove(t);
            q->del(n);
        }
    }
else
    q = 0;
return q;
}
/////////////////////////////////////////////////////////////////

void System::releaseLocks(Transaction *t, int commit)
{
    List *lset, *Q, *p;
    Node *n, *wake;
    Mapping *mp;
    AUE *au;
    int pa, index, ndex, full, uID;
    DataCacheLine *qd;
    unsigned int dcva;
    unsigned int vad;

    lset = t->getLockset();
    if (lset)
    {
        lset->reset();
        while ((n = lset->next()) != 0)
        {
            mp = (Mapping *) n->obj;
            pa = mp->getPA();
            vad = mp->getVAD();
            uID = mp->getVA();

            p = AUtable->access(uID, index);
            if (p)
            {
                au = (AUE *) p->findAU(mp);
                if (au)
                {
                    if (!commit)
                        Q = offSuspendQ(t, au);
                    else
                        Q = au->getQ();
                    if (Q)
                    {
                        wake = Q->first();
                        if (wake)
                        {
                            ReadyQueue->add(wake->obj);
                        }
                    }
                }
            }
        }
    }
}

```



```

    DbgPrint((cout << "illegal action: unit already in unlocked state: PA:
              " << pa << " VA: " << va << endl));
    p->setPDID(0);
    if (q)
        q->setPDID(0);
    break;
case 1: // Single Reader
    p->setCurrentState(0); //unlocked
    p->setOtherCurrentState(0); //unlocked
    p->setPDID(0);
    p->getLineBits()->setDirtyBit(0);

    if (q)
    {
        q->getLine()->setRefBit(0);
        q->getLine()->setDirtyBit(0);
        q->setPDID(0);
    }
    pt = cpu->getPLBTable()->access(va,vid,css,os,idp,l,cc);

    break;
case 2: // Other Single Reader - not used
    p->setCurrentState(0); //unlocked
    p->setOtherCurrentState(0); //unlocked
    p->setPDID(0);
    p->getLineBits()->setDirtyBit(0);

    if (q)
    {
        q->getLine()->setRefBit(0);
        q->getLine()->setDirtyBit(0);
        q->setPDID(0);
    }

    break;
case 3: //Multiple Reader
    lptr = AUtable->access(pa,i);
    if (lptr)
    {
        a = (AUE *) lptr->findAU(m);
        if (a->getReaders() != 0)
        {
            if ((num = a->getReaders()->getCount()) == 1)
            {
                // 2 readers

                p->setCurrentState(1); //SR
                p->setOtherCurrentState(2); //OSR
                x = a->getReaders()->first();
                p->setPDID(((Transaction *)x->obj)->
getID()); //record SID of reader that's left
            }
        }
    }

```



```

int result,i, entry=0;
Mapping *mp;
List * p, *lptr, *readq;
int state; // flag for a multiple reader
AUE *a;
Transaction *tran;
TSB *ts;
int viewdefn, uid, mPA;

result = cpu->accessDC(VA,AT,PA,state,ltrace,viewdefn,uid, mPA);
mp = new Mapping;
mp->setVA(uid);
mp->setPA(mPA); // PA is used to look up the PLB table subsequently
mp->setVAD(VA);
if (state == 1) // record reader ID when there are multiple readers
{
    tran = 0;
    lptr = ATable->access(uid,i);
    if (lptr)
    {
        a = (AUE *) lptr->findAU(mp);
        if (a)
        {
            readq = a->getReaders();
            if (readq)
                tran = (Transaction *) (readq->findtrans(xn->getID()));
            else
            {
                readq = new List;
                a->setReaders(readq);
            }
        }
        else
        {
            a = new AUE;
            readq = new List;
            a->setQ(0);
            a->setReaders(readq);
            a->setAUID(mp);
            lptr->add(a);
        }
    }
    else
    {
        lptr = new List;
        ATable->add(i, lptr);
        a = new AUE;
        readq = new List;
        a->setQ(0);
    }
}

```

```

        a->setReaders(readq);
        a->setAUID(mp);
        lptr->add(a);
    }
    if (tran == 0)
        readq->add(xn);
}

if (!result)
{
    if (xnprocessing && viewdefn)
    {
        // records all AUs touched by a transaction
        // facilitates commit procedure
        p = xn->getLockset();
        if (p)
        {
            if (p->find(uid) == 0)
                xn->add(mp); // adds mapping for AU to the transaction
                               lockset
        }
        else
            xn->add(mp);
    }
}
else // suspend xn
{
    if (xnprocessing && viewdefn)
    {
        entry = 1;
        suspend(mp);
        contextSwitch();
    }
}
if (!entry)
{
    if (dcount == timeout)
    {
        ts = saveContext(xn);
        contextSwitch(); // this is done before adding to the
        // ReadyQueue so that the transaction would not go on just to be
        // taken off immediately on context switch.
        ReadyQueue->add(ts);
    }
}
}
}

```

```

////////////////////////////////////
void System::contextSwitch()
{
    Transaction *t;
    Node *n;
    TSB *tsb;
    long spot;
    int k;
    Object *o;

    dcount = 0;

    if (ReadyQueue->getCount(>0)
    {
        n = ReadyQueue->first();
        o = n->obj;
        tsb = (TSB *) o;
        t = tsb->getTID();
        xn = t;
        DbgPrint((ltrace << "Current TID: " << xn->getID() << " dequeued
off RQ" << endl));
        spot = tsb->getmark();
        tracefp->seekg(spot,ios::beg);
        k = tsb->getKey();
        setSID(k); // trans id
        cpu->setSID(k);
        n = ReadyQueue->remove(o);
        ReadyQueue->del(n);
    }
    else //move forward in the trace file
    {
        AT = 99;
        while ((!tracefp->eof()) && (AT != 'B')) // AT = 3 ="B" = BOT
        {
            setAccessType();
            if (AT == 'I')
                tracefp->readInst();
            setCVA();
        }
        if (!tracefp->eof())
        {
            setCurrTrans(TransStartup(VA));
        }
        else
        {

            ltrace.close();
            xnOrder.close();
            ltrace << "\t\t \n\t}\n }\n";
            ltrace.close();
            printTables();
        }
    }
}

```

```

        printResults();
        exit(1); // entire file processed
    }
}

////////////////////////////////////
void System::getNextAddress()
{
    Transaction * tkill;
    Node *n;

    do
    {
        setAccessType();
        if (AT == 'I')
            tracefp->readInst();
        setCVA();
    } while ( AT != 'R' && AT != 'W' && AT != 'I' && AT != 'B' && AT != 'E'
    && AT != 'T' && AT != 'U' && !tracefp->eof() );

    if ( AT == 'R' )
    {
        AT = 0;
        dcount++; // transaction's CPU timeout count
    }

    else if ( AT == 'I' )
        AT = 2;

    else if ( AT == 'W' )
    {
        AT = 1;
        dcount++; // transaction's CPU timeout count
    }

    else if (AT == 'B')
    {
        if (!loading)
        {
            xnprocessing = 1;
            setCurrTrans(TransStartup(VA));
            AT = 3;
        }
    }

    else if (AT == 'E')
    {

```



```

if (!loading)
{
    AT = 4;
    xnprocessing = 0;
    tkill = xn;
    if (MPL->getCount() > 1)
        tkill = (Transaction *) MPL->findtrans(VA);
    ltrace << "\n\t\t\tLocktable->releaseLocks(t[" << VA << "]);\n\n";
    if (tkill)
    {
        releaseLocks(tkill, 1); //commit
        n = MPL->remove(tkill); // dequeue from active xn list
        MPL->del(n);
        delete tkill; // kill the transaction
        xn = 0;
    }
    else
        cerr << "release locks failed (Trans null)\n";
}
else if (AT == 'U')
{
    AT = 5;
}
else if (AT == 'T')
{
    AT = 6;
    cout << "Transaction type : " << VA << endl;
    xnOrder << "Transaction type : " << VA << endl;
}
else
    AT = -1;          // Error detected

}
//////////////////////////////////////////////////////////////////
void System::loadPageTable()
{
    // Note that the simulator does not include the delay for loading the
    // Page tables

    int PFN;
    unsigned int VFN;
    int mask = 0x000ffff;

    getNextAddress();

    while ( !tracefp->eof() )

```

```

    {
        VFN = (VA >> lgPAGESIZE);
        VFN = VFN & mask; // to zero fill the leftmost 12 bits
        PFN = cpu->getPageTable()->PTread(VFN);
        if (PFN==MAX) // mapping does not exist
        {
            if (cpu->getPMFreeList()->getTop()==cpu->getPMFreeList()->
                getStackPointer())
                PFN = cpu->getPageTable()->replace();
            else
                PFN = cpu->getPMFreeList()->pop();
            cpu->getPageTable()->PTwrite(VFN,PFN);
            cpu->getStats()->setPTable(1,1);// write access to page table
        }
        getNextAddress();
    }
    loading = 0;
    DbgPrint((cout << "Finished Loading Pagetable" << endl));
}

/////////////////////////////////////////////////////////////////
void System::printTables()
{
    ofstream outp;

    outp.open("output.dat");
    outp << "Printing Page Table" << endl;
    outp << cpu->getPageTable();
    outp << "Printing PLB Table" << endl;
    outp << cpu->getPLBTable();
    outp << "Printing Data Cache" << endl;
    outp << cpu->getDC();
    outp << "Printing PLB Cache" << endl;
    outp << cpu->getPLB();

    outp << "Printing Ready Queue\n" << ReadyQueue << "\n";
    outp << "Printing AU Tablet" << "\n";
    outp << AUtable << "\n";
    if (xn)
        outp << " Last xn's lockset:\n" << xn->getLockset() << "\n";

    outp.close();
}

/////////////////////////////////////////////////////////////////
void System::printResults()
{
    ofstream f;
    Cache * cp;
    CacheStats *s;
    char *cachename;

```







```

PW = (float)(W*100)/(W+R);
PR = (R*100)/(float)(W+R);
PWAC=(WAC/(float)W)*100;

```

```

ff << activity << "\t" << n << "\t\t" << R << "\t\t\t" << W << "\t\t\t" << WAC <<
"\t\t\t" << setprecision(4) << setw(5) << PR <<
"\t\t\t" << setw(5) << PW << "\t\t\t" << setw(5) << PWAC << endl;
}

```

```

void System:: getHitsStats(char * activity, int n, int mWAC, int hWAC, int *fetch, int
*miss, ofstream ff)

```

```

{
int W, R, WAC;
float PW, PR, PWAC;

W = fetch[1]-miss[1];
R = fetch [0]-miss[0];
n = W + R;
WAC = hWAC;
PW = (float)(W*100)/n;
PR = (R*100)/(float)n;
PWAC=(WAC/(float)W)*100;

ff << activity << "\t" << n << "\t\t" << R << "\t\t\t" << W << "\t\t\t" << WAC <<
"\t\t\t" << setprecision(4) <<
setw(5) << PR << "\t\t\t" << setw(5) << PW << "\t\t\t" << setw(5) << PWAC <<
endl;
}

```

### Filename: datacache.cpp

```

#include <iostream.h>
#include <fstream.h>
#include "object.h"
#include "list.h"
#include "hash.h"
#include "input.h"
#include "trans.h"

#include "spec.h"

////////////////////////////////////
DataCache::DataCache (int s)
{
if (s<=0) cerr << "Data Cache size cannot<=0" << endl;
numlines = s;

cache = new Cache;
cache->setNumUorDsets(s);
}

```

```

    line = (List **) new List *[s];

    for (int i = 0; i < s ; i++)
    {
        line[i]=0;
    }
}
////////////////////////////////////////////////////////////////

DataCache::~DataCache()
{
    delete []line;
    delete cache;
}
////////////////////////////////////////////////////////////////

void DataCache::writeCache(int status)
{
    // Assuming each line size is 128 bytes..to generalize: n bytes..this means
    // that the lower log n bits can be discarded from the address
    // stored as the tag in the cache line. ?? confirm this.
    // Assume other function previously set the "line" member in the DataCache instance
    // so that "line" points to the cache line to be written or replaced.

    switch (status)
    {
        case 0: //line fetch and read access
            cline->getLine()->setValidBit(1);
            cline->getLine()->setDirtyBit(0);
            cline->getLine()->setRefBit(1);
            break;
        case 1: //line fetch and write access
            cline->getLine()->setValidBit(1);
            cline->getLine()->setDirtyBit(1);
            cline->getLine()->setRefBit(1);
            break;
        case 2: // write access and no fetch
            cline->getLine()->setDirtyBit(1);
            cline->getLine()->setRefBit(1);
            break;
        case 3: //line fetch and no access (prefetch)
            cline->getLine()->setValidBit(1);
            cline->getLine()->setDirtyBit(0);
            cline->getLine()->setRefBit(0);
            break;
    }
}

////////////////////////////////////////////////////////////////
void
DataCache::indexDCCache(unsigned int va, // full virtual address (32 bits)

```

```

        DataCacheLine *&q, // returns a ptr to the line searched for
        int idx, // cache index
        int &full, // var indicates whether a set is full or not
        int id, // pdid
        int vflag, // indicates whether a view is degined or not
        int uid)
{
// uses the virtual address reference to find the corresponding line in the
// cache if it exists. Returns a pointer to the line in the cache

    int addr;

    if (line[idx]== 0)
        q = 0;
    else
    {
        if (vflag)
            addr = uid;
        else
            addr = getDCVA(va);
        q = (DataCacheLine *) line[idx]->search(addr,id,vflag);
        if (line[idx]->getCount() == cache->getAssoc())
            full = 1; // all lines have been created in a set
    }
    cline = q;
}
/////////////////////////////////////////////////////////////////
DataCacheLine *
DataCache::createDCLine
(
int ix, // index
unsigned int VA, // virtual address
int PA, // physical frame number
int id, // pdid that owns the rights to a line
int v // unit id - needed to differentiate units that are cached into the same line
)
{
    DataCacheLine *p;
    LineControlBits *q;
    p = new DataCacheLine;
    q = new LineControlBits;
    p->setPDID(id);
    p->setLine(q);
    p->getLine()->setTag(PA);
    p->getLine()->setValidBit(1);
    p->getLine()->setRefBit(1);
    p->getLine()->setDirtyBit(0);
    //v = getDCVA(VA);
    p->getLine()->setTag1(v);
    if (line[ix]==0) // no datacache entries in a set

```



```

        line[ix] = new List;
        line[ix]->push((Object *) p,0);           //adds entry to the top of the list so that
                                                // LRU can be maintained.

        cline = p;
        return p;
    }
    ///////////////////////////////////////////////////////////////////
void
DataCache::replace(
int index,           // identifies the list in the DC wherein an element is to be replaced
CacheStats *&s,
int ac,
int level
)
{
    List ** p;
    DataCacheLine *d;
    Node *n;

    p = getLine();
    d = (DataCacheLine *) (p[index]->last()->obj);
    if (d->getLine()->getDirtyBit() == 1)
    {
        if (level == 1)
        {
            s->setDelay(MemoryAccess); //increment stats to show a write back
            to memory
            s->setRWDelay(ac,MemoryAccess); //increment stats to show a
            // write back to memory
            s->setDTable(0,1);           // write back to memory
        }
        else
        {
            s->setDelay(L2access); // write back to secondary cache
            s->setRWDelay(ac,L2access); //increment stats to show a write back
            // to secondary cache
        }
    }
    n = p[index]->remove((Object *) d);
    p[index]->del(n);
}
    ///////////////////////////////////////////////////////////////////

void
DataCache::dump(ostream &os)
{
    int i;
    for (i=0; i<numlines; i++)
        if (line[i]!=0)

```





```

(
  Object *o
) : obj(o), next(0)
{
}

////////////////////////////////////
SuspendQueue::SuspendQueue()
{
    Q = new List;
}
////////////////////////////////////

SuspendQueue::~SuspendQueue()
{
}
////////////////////////////////////

void
List::add          // adds to the end of the list
(
  Object *o
)
{
  Node *n = new Node(o);
  if (list == 0)
  {
    list = n;
    tail = n;
  }
  else
  {
    tail->next = n;
    tail = n;
  }
  count++;
}

////////////////////////////////////

void
List::addn        // adds to the end of the list
(
  Node *n
)
{
  if (list == 0)
  {
    list = n;
  }
}

```

```

        tail = n;
    }
    else
    {
        tail->next = n;
        tail = n;
    }
    count++;
}

/////////////////////////////////////////////////////////////////
List *
List::join
(
    List *l
)
{
    if (l == 0)
    {
        return this;
    }
    else
    {
        if (list == 0)
        {
            list = l->list;
            tail = l->tail;
            count = l->count;
        }
        else
        {
            tail->next = l->list;
            tail = l->tail;
            count += l->count;
        }
    }
    return this;
}

/////////////////////////////////////////////////////////////////
Node *
List::next()
{
    Node *p;

    p = cursor;

    if (cursor != 0)
        cursor = cursor->next;

    return p;
}

```

```

}
/////////////////////////////////////////////////////////////////
Node *
List::first( void )
{
    return list;
}
/////////////////////////////////////////////////////////////////
Node *
List::last( void )
{
    return tail;
}
/////////////////////////////////////////////////////////////////
void
List::reset()
{
    cursor = list;
}

/////////////////////////////////////////////////////////////////
void
List::dump
(
    ostream &os
)
{
    Node *nptr;
    int j = 0;

    reset();
    while (nptr = next())
    {
        os << j++ << ":" << (Object *) nptr << endl;
    }
}

/////////////////////////////////////////////////////////////////
List *
addlist
(
    List *l,
    Object *o
)
{
    if (l == 0)
        l = new List;

    l->add(o);

    return l;
}

```

```

}
////////////////////////////////////
// List::push can be used for either of 2 purposes
// (1) package an object in a node and add to the start of the list
// or (2) the ptr to the node is passed as an input parameter i.e. a hanging node exists.
// The second use is part of the LRU algorithm where a list will be reordered on access.
// That is the content at the top of the list is the most recently accessed and the one at the
// bottom the Least Recently Used.
// In LRU we would want to remove a node from somewhere in the list and push it on as
// the head
// or we could have a new entry and want to add it to the top of the list.

```

```

void
List::push( Object *o, Node *n)
{
    Node *m;
    if (n==0)
    {
        m = new Node(o);
        n = m;
    }

    if ( list == n )
        cerr << "Node is already in the list" << endl;
    else
    {
        if (list == 0)
        {
            list = n;
            tail = n;
        }
        else
        {
            n->next = list;
            list = n;
        }
        count++;
    }
}

```

```

////////////////////////////////////
// Replaces the contents of the first node on a list - can be used in the LRU strategy

```

```

Object
*List::replace(Object *ob)
{
    Object *o = list->obj;

    list->obj = ob;
    return o;
}

```

```

////////////////////////////////////
void List::del(Node *n)
{
    n->obj = 0;
    delete n;
}
////////////////////////////////////
// Removes a node from anywhere in the list and leaves it hanging. Must follow
// up this function by a call to either List::del or List::push

Node * List::remove(Object *o)
{
    Node *n;
    Node *found = 0;
    Node *p;

    if (list == 0)
        cerr << "Error: Attempted deletion from an empty list" <<endl;
    else
    {
        reset();
        p = list;
        n = next();

        while (found == 0 && n!=0 )
        {
            if ( n != 0 )
            {
                if ( n->obj == o )
                    found = n;
                else
                {
                    p = n;
                    n = next();
                }
            }
        }
        if ( found != 0 )
        {
            p->next = found->next;
            if ( tail == list )
            {
                if ( list == found )
                {
                    list = list->next;
                    tail = list;
                }
            }
            else
            {

```



```

        if ( list == found )
        {
            list = list->next;
        }

        if ( tail == found )
        {
            tail = p;
        }
    }
    found->next = 0;
    count--;
}
}
return found;
}

////////////////////////////////////
Object * List::find(unsigned int VA)
{
    Node *n;
    Mapping *map;

    reset();

    n = next();
    map = (Mapping *) n->obj;
    while ((VA!=map->getVA()) && (n!=0))
    {
        n = next();
        if (n!=0)
            map = (Mapping *) n->obj;
    }
    if (n!=0) return n->obj;
    else return 0;
}

////////////////////////////////////
Object * List::findtrans(int tid)
{
    Node *n;
    Transaction *t;

    reset();

    n = next();
    t = (Transaction *) n->obj;
    while ((tid!=t->getID()) && (n!=0))
    {
        n = next();
        if (n!=0)
            t = (Transaction *) n->obj;
    }
}

```

```

    }
    if (n!=0) return n->obj;
    else return 0;
}
/////////////////////////////////////////////////////////////////
Object * List::findAU(Object *m)
{
    Node *n;
    AUE *aue;
    unsigned int va;
    int pa;

    reset();

    n = next();
    aue = (AUE *) n->obj;
    va = ((Mapping *) m)->getVA();
    pa = ((Mapping *) m)->getPA();
    while ((va != aue->getAUID()->getVA()) && (n!=0))
    {
        n = next();
        if (n!=0)
            aue = (AUE *) n->obj;
    }
    if (n!=0) return n->obj;
    else return 0;
}
/////////////////////////////////////////////////////////////////
Object * List::findTSB( Object *t)
{
    Node *n;
    TSB *b;

    reset();

    n = next();
    b = (TSB *) n->obj;
    while (((Transaction *) t != b->getTID()) && (n!=0))
    {
        n = next();
        if (n!=0)
            b = (TSB *) n->obj;
    }
    if (n!=0) return n->obj;
    else return 0;
}
/////////////////////////////////////////////////////////////////
Object * List::find(
int unitPA,           // masked physical address
int ViewID,
int &i)

```

```

{
    Node *n;
    PLBLine *entry;

    reset();

    n = next();
    entry = (PLBLine *) n->obj;
    while ((unitPA != entry->getLineBits()->getTag()) && (n!=0))
    {
        n = next();
        if (n!=0)
        {
            i++;
            entry = (PLBLine *) n->obj;
        }
    }
    if (n!=0)
    {
        if (entry->getViewID() == ViewID)
            return n->obj;
        else
            return 0;
    }
    else return 0;
}

////////////////////////////////////
// Searches a list containing instances of class DataCacheLine for a match.
// If a view is defined on the virtual address, the PDID must be checked to ascertain
// which subject owns rights to the line.

Object * List::search(int addr,          // unsigned virtual address
                     int pdid,         // PDID identifier
                     int viewflag      // 0 - no view defined 1 - view defined on addr
                     )
{
    Node *n;
    DataCacheLine *entry;

    reset();

    n = next();
    if (n !=0)
    {
        entry = (DataCacheLine *) n->obj;
        while ((addr != entry->getLine()->getTag() || (entry->getPDID() != pdid))
            && (n!=0))
        {
            n = next();
            if (n!=0)

```

```

        entry = (DataCacheLine *) n->obj;
    }
}

if (n!=0)
    return n->obj;
else return 0;

}

/////////////////////////////////////////////////////////////////
Filename: pagetable.cpp
/////////////////////////////////////////////////////////////////

#include <iostream.h>
#include <fstream.h>
#include "object.h"
#include "list.h"
#include "hash.h"
#include "input.h"
#include "trans.h"
#include "spec.h"

/////////////////////////////////////////////////////////////////
void
PageTable::dump(ostream &os)
{
    os << "Page Table ASID : " << ASID << endl;
    os << "Size          : " << size << endl;
    os << "Page table follows : " << endl;
    os << PTable;
}
/////////////////////////////////////////////////////////////////
PageTable::PageTable(int s, int VAID)
{
    PTable = new HashTable(s);
    size = s;
    ASID = VAID;
}
/////////////////////////////////////////////////////////////////
PageTable::~PageTable()
{
}
/////////////////////////////////////////////////////////////////
int PageTable::PTread(unsigned int v)
{
    List **p;
    List *lptr;
    Mapping *mp;
    int PFN;
    int index;
}

```

```

index = PTable->hashaddr(v); // calc. the hash address of virtual address v
p= PTable->getTable(); // retrieves the ptr to the hash table
lptr = p[index]; // retrieves the ptr to the list of mappings for a hash address
if (lptr!=0)
    mp = (Mapping *) lptr->find(v); // retrieves a pointer to the required
    // mapping if it exists
if ((lptr==0) || (mp==0))
    PFN = MAX; //hard page fault -- to secondary storage - create PT mapping
else
    PFN = mp->getPA();
return PFN;
}
/////////////////////////////////////////////////////////////////
int PageTable::getTranslation(unsigned int VA)
{
    int p;
    p = PTread(VA);
    // increment stats for delay due to PageTable access
    return p;
}
/////////////////////////////////////////////////////////////////
void PageTable::PTwrite(unsigned int VA, int PA)
{
    List **p;
    List *lptr;
    Mapping *mp;
    int index;

    mp = new Mapping;
    mp->setVA(VA); // sets the VA attribute of mp
    mp->setPA(PA); // sets the PA attribute of mp
    mp->setVAD(0);
    index = PTable->hashaddr(VA); // calc. the hash address of virtual address v
    p= PTable->getTable(); // retrieves the ptr to the hash table
    lptr = p[index]; // retrieves the ptr to the list of
    // mappings for a hash address

    if (lptr==0)
    {
        lptr = new List;
        PTable->add(index,lptr);
    }

    lptr->add(mp); // adds the mapping to Page Table.
}
/////////////////////////////////////////////////////////////////
Filename: plb.cpp
#include <iostream.h>

```

```

#include <fstream.h>
#include "object.h"
#include "list.h"
#include "hash.h"
#include "input.h"
#include "trans.h"

#include "spec.h"

////////////////////////////////////
PLB::PLB (int s)
{
    cache = new Cache;
    if (s<=0) cerr << "PLB size cannot<=0" << endl;
    cache->setNumUorDsets(s);
    numlines = s;
    plbline = (List **) new List *[s];

        for (int i = 0; i < s; i++)
        {
            plbline[i]=0;
        }

}
////////////////////////////////////
// plbMissHandler(...) is responsible for finding the required PLB entry in the
// PLB memory table and loading it in the PLB cache. Replacement on the PLB cache may
// be necessary.

PLBLine*
PLBTable::plbMissHandler
(
int PAu,           // Address
int ViewID,       // VIEWID
int &cs,          // Current state
int &ocs,         // Other subjects' current state
int &PDID,        // process or subject identifier
int a,
CacheStats *&st
)
{
    PLBLine *b;

    b = access(PAu, ViewID, cs, ocs,PDID, a, st); //returns output variables

    return b;
}

```

```

////////////////////////////////////
void PLB::writeCache (PLBLine *c,int newstate,int othernewstate)
{
    c->setCurrentState(newstate);
    c->setOtherCurrentState(othernewstate);
}
////////////////////////////////////

PLBTable::PLBTable(int s)
{
    plbTable = new HashTable(s);
    size = s;
}
////////////////////////////////////
void
PLB::dump(ostream &os)
{
    int i;
    for (i=0; i<numlines; i++)
        if (plbline[i]!=0)
            plbline[i]->dump(os);
}
////////////////////////////////////
void
PLBTable::dump(ostream &os)
{
    os << "PLBTable : " << endl;
    os << "\tSize : " << size << endl;
    os << "\tPLB Table follows: " << endl << plbTable;
}
////////////////////////////////////
PLBTable::~PLBTable()
{
}
////////////////////////////////////
PLBLine * PLBTable::access(
int PAunit,           // masked physical address
int ViewID,
int &cstate,
int &ocstate,
int &PDID,
int action,
CacheStats *&cst
)
{
    List **p;
    List *lptr;
    PLBLine *pline;
    int index;
    int i = 1; //not used in this function - is present for reusability of find(..)
}

```

```

index = plbTable->hashaddr(PAunit);           // calc. the hash address
p = plbTable->getTable();                     // retrieves the ptr to the hash table
lptr = p[index];                             // retrieves the ptr to the list of entries
                                              // for a hash address

if (lptr!=0)
    pline = (PLBLine *) (lptr->find(PAunit,ViewID,i)); // retrieves a pointer to
                                                         //the required entry if it exists

if ((lptr==0) || (pline==0))
{
//normally a hard page fault -- to secondary storage. I assume enough space
// available
// in memory to hold the PLB table. Therefore if an entry does not exist in it, it
// will be created with default values.

    pline = createEntry(PAunit,ViewID, PDID, 0, 0); // cs = ocs = 0 default
if (lptr==0)
{
    lptr = new List;
    plbTable->add(index,lptr);
}

lptr->add(pline);           // adds the entry to PLB Table.
cst->setPLBTable(2,1);     // incr. PLB entry creation
cst->setDelay(MemoryAccess);

}

if (action == 0) // read access
{
    cstate = pline->getCurrentState();
    ocstate = pline->getOtherCurrentState();
    PDID = pline->getPDID();
    cst->setPLBTable(0,1);
    cst->setDelay(MemoryAccess);
}
else // writeback
{
    pline->setCurrentState(cstate);
    pline->setOtherCurrentState(ocstate);
    pline->getLineBits()->setTag(PAunit);
    pline->getLineBits()->setDirtyBit(0);
    pline->getLineBits()->setRefBit(0);
    pline->getLineBits()->setValidBit(1);
    pline->setViewID(ViewID);
    pline->setPDID(PDID);
    cst->setPLBTable(1,1);
    cst->setDelay(MemoryAccess);
}

return pline;

```





```

void PLB::replace(int index, PLBTable *pt, CacheStats *&st)
// removes a line from the set. The contents of the line are written back to memory
// if the dirty bit was set.
{
    List **p;
    PLBLine *q, *r;
    int mPA, vid, s, os, pid;

    p = getPLBLine();
    q = (PLBLine *) (p[index]->last()->obj);
    if (q->getLineBits()->getDirtyBit() == 1)
    {
        //increment stats to show a write back to memory
        mPA = q->getLineBits()->getTag();
        vid = q->getViewID();
        s = q->getCurrentState();
        os = q->getOtherCurrentState();
        pid = q->getPDID();
        r = pt->access(mPA,vid,s,os,pid,1,st); //action is WB - write back
    }
    Node *n = p[index]->remove((Object*)q);
    p[index]->del(n);
}

```

```

}
/////////////////////////////////////////////////////////////////

```

**Filename: tlb.cpp**

```

#include <iostream.h>
#include <fstream.h>
#include "object.h"
#include "list.h"
#include "hash.h"
#include "input.h"
#include "trans.h"
#include "spec.h"

```

```

/////////////////////////////////////////////////////////////////

```

```

TLB::TLB (int s)
{
    cache = new Cache;
    cache->setNumUorDsets(s);
    if (s<=0) cerr << "TLB size cannot<=0" << endl;
    numlines = s;
    Line = new TLBLine[s];

    for (int i = 0; i < numlines; i++)
    {
        Line[i].setASID(0);
        Line[i].setMP(0);
        Line[i].setLineBits(0);
    }
}

```

```

}
/////////////////////////////////////////////////////////////////

TLB::~~TLB()
{
    delete []Line;
    delete cache;
}
/////////////////////////////////////////////////////////////////
// Searches the entire TLB: a sequential version of a fully associative TLB.
int TLB::getTranslation(unsigned int VA)
{
    int i = 0;
    Mapping *m;

    TLBLine *p = Line;
    if (p!=0)
        while (i < numlines)
        {
            m = p->getMP();
            if (m)
            {
                if ( VA == m->getVA())
                {
                    return m->getPA();
                }
            }
            p++;
            i++;
        }
    return MAX;
}
/////////////////////////////////////////////////////////////////
void TLB::setStatus()
{
    int j;

    j = findFree();
    if (j == numlines)
        status = 1;
    else
        status = 0;
}
/////////////////////////////////////////////////////////////////
void TLB::doLRU(int index)
{
    int i;
    TLBLine tmp;
    tmp = Line[index];
    for (i = index; i==0; i--)
        Line[i] = Line [i-1];
}

```



```

TLB::operator[](int i)
{
    return Line[i];
}

```

```

/////////////////////////////////////////////////////////////////
Filename: trans.cpp

```

```

#include <iostream.h>
#include <fstream.h>

```

```

#include "object.h"
#include "list.h"
#include "hash.h"
#include "input.h"
#include "trans.h"
#include "spec.h"

```

```

Transaction::Transaction(int ident, Input * i)
{
    fd = i;
    id = ident;
    lockset = 0;
}

```

```

/////////////////////////////////////////////////////////////////
Transaction::~Transaction()
{
}

```

```

/////////////////////////////////////////////////////////////////
TSB::TSB()
{
}

```

```

/////////////////////////////////////////////////////////////////
TSB::~TSB()
{
}

```

```

/////////////////////////////////////////////////////////////////
AUE::AUE()
{
}

```

```

/////////////////////////////////////////////////////////////////
AUE::~AUE()
{
}

```

```

/////////////////////////////////////////////////////////////////
void Transaction::add(Object * r)
{

```

```

    if (lockset == 0)
        lockset = new List;

```

```

        lockset->add(r);
    }
    ///////////////////////////////////////////////////////////////////
    void Transaction::wakeup()
    {
    }

    ///////////////////////////////////////////////////////////////////
    void Transaction::dump( ostream &os )
    {
        os << "Transaction ID: " << id << "\nTransaction Lockset : " << lockset << "\n";
    }
    ///////////////////////////////////////////////////////////////////
    void TSB::dump( ostream &os )
    {
        os << "TSB Transaction ID: " << key << "\tByte in file " << mark << "\n";
    }

    ///////////////////////////////////////////////////////////////////
    void AUE::dump( ostream &os )
    {
        os << "AU ID: " << map << "\n";
        os << "Suspend Queue:\n" << suspendQ << "\n";
        os << "Multiple Readers: \n" << mreader << "\n";
    }
    ///////////////////////////////////////////////////////////////////

```

**Filename: spec.cpp**

```

#include <iostream.h>
#include <fstream.h>
#include "object.h"
#include "list.h"
#include "hash.h"
#include "input.h"
#include "trans.h"
#include "spec.h"

    ///////////////////////////////////////////////////////////////////
    Input::~Input()
    {
    }
    ///////////////////////////////////////////////////////////////////
    ConfigInput::~ConfigInput()
    {
    }
    ///////////////////////////////////////////////////////////////////
    DataCacheLine::DataCacheLine()
    {
    }

```



```

PLBLine::PLBLine()
{

}
////////////////////////////////////////////////////////////////
PLBLine::~PLBLine()
{

}
////////////////////////////////////////////////////////////////
void
PLBLine::dump(ostream &os)
{
    os << "PLBLine Object : " << endl;
    os << "\tCount : " << Count << endl;
    os << "\tCurrent State : " << CurrentState << endl;
    os << "\tLineBits : " << LineBits << endl;
    os << "\tOther : " << OtherCurrentState << endl;
    os << "\tPDID : " << PDID << endl;
    os << "\tViewId : " << ViewID << endl;
}
////////////////////////////////////////////////////////////////
TLBLine::TLBLine()
{

}
////////////////////////////////////////////////////////////////
TLBLine::~TLBLine()
{

}
////////////////////////////////////////////////////////////////
VDULine::VDULine()
{

}
////////////////////////////////////////////////////////////////
VDULine::~VDULine()
{

}
////////////////////////////////////////////////////////////////
int VDU::calcMask(int unitsize)
{
    int mask;

    mask = AddressSpaceSize - unitsize + 1;
    return (mask);
}
////////////////////////////////////////////////////////////////

VDU::VDU(int s)
{
    int tupleSize;
    int mask;
}

```



```

cache = new Cache;
if (s<=0) cerr << "VDU size cannot<=0" << endl;
cache->setNumUorDsets(s);

numlines = s;
Lines = new CacheLine[s];

for (int i = 0; i < s; i++)
    {
        Lines[i].LineBits.setTag(0);
        Lines[i].LineBits.setValidBit(0);
        Lines[i].LineBits.setDirtyBit(0);
        Lines[i].LineBits.setRefBit(0);
        Lines[i].contents = new VDULine;
        Lines[i].contents->setViewID(999); // default
        Lines[i].contents->setFSMID(0); // default
        Lines[i].contents->setVFNStart(0xffffffff); // default
        Lines[i].contents->setVFNEnd(0xffffffff); // default
        Lines[i].contents->setMask(0xfffff000); // default 4K pages
        Lines[i].contents->setVIDPtr(0); // default

i=0:    } // multiple views defined
        // Warehouse Relation
        tupleSize = 0x00000080;
        mask = calcMask(tupleSize);
        Lines[i].LineBits.setValidBit(1);
        Lines[i].contents->setViewID(0);
        Lines[i].contents->setVFNStart(0x0001f060);
        Lines[i].contents->setVFNEnd(0x00051984);
        Lines[i].contents->setMask(mask); // 4K unit size

i=1;
// District Relation
        tupleSize = 0x00000080;
        mask = calcMask(tupleSize);

        Lines[i].LineBits.setValidBit(1);
        Lines[i].contents->setViewID(1);
        Lines[i].contents->setVFNStart(0x00052a28);
        Lines[i].contents->setVFNEnd(0x001200c4);
        Lines[i].contents->setMask(mask); // 4K unit size

i=2;
// Customer Relation
        tupleSize = 0x00000400;
        mask = calcMask(tupleSize);

        Lines[i].LineBits.setValidBit(1);
        Lines[i].contents->setViewID(2);
        Lines[i].contents->setVFNStart(0x00121148);

```

```

        Lines[i].contents->setVFNEnd(0x00602ce4);
        Lines[i].contents->setMask(mask); // 4K unit size

i=3;
// Item Relation
    tupleSize = 0x00000080;
    mask = calcMask(tupleSize);

    Lines[i].LineBits.setValidBit(1);
    Lines[i].contents->setViewID(3);
    Lines[i].contents->setVFNStart(0x006040e8);
    Lines[i].contents->setVFNEnd(0x0070514c);
    Lines[i].contents->setMask(mask); // 4K unit size

i=4;
// Stock Relation
    tupleSize = 0x00000200;
    mask = calcMask(tupleSize);

    Lines[i].LineBits.setValidBit(1);
    Lines[i].contents->setViewID(4);
    Lines[i].contents->setVFNStart(0x00960d98);
    Lines[i].contents->setVFNEnd(0x00dbceb8);
    Lines[i].contents->setMask(mask); // 4K unit size

i=5;
// Order Relation
    tupleSize = 0x00000020;
    mask = calcMask(tupleSize);

    Lines[i].LineBits.setValidBit(1);
    Lines[i].contents->setViewID(5);
    Lines[i].contents->setVFNStart(0x007061d8);
    Lines[i].contents->setVFNEnd(0x007cedd4);
    Lines[i].contents->setMask(mask); // 4K unit size

i=6;
// Neworder Relation
    tupleSize = 0x00000008;
    mask = calcMask(tupleSize);

    Lines[i].LineBits.setValidBit(1);
    Lines[i].contents->setViewID(6);
    Lines[i].contents->setVFNStart(0x007cfd8);
    Lines[i].contents->setVFNEnd(0x0095ed84);
    Lines[i].contents->setMask(mask); // 4K unit size

i=7;
// OrderLine
    tupleSize = 0x00000040;

```

```

        mask = calcMask(tupleSize);

        Lines[i].LineBits.setValidBit(1);
        Lines[i].contents->setViewID(7);
        Lines[i].contents->setVFNStart(0x0dbe098);
        Lines[i].contents->setVFNEnd(0x00e88574);
        Lines[i].contents->setMask(mask); // 4K unit size

i=8;
// History

        tupleSize = 0x00000040;
        mask = calcMask(tupleSize);

        Lines[i].LineBits.setValidBit(1);
        Lines[i].contents->setViewID(8);
        Lines[i].contents->setVFNStart(0x00e895b8);
        Lines[i].contents->setVFNEnd(0x00f53a94);
        Lines[i].contents->setMask(mask); // 4K unit size
    }
    ///////////////////////////////////////////////////////////////////
    VDU::~VDU()
    {
    ///////////////////////////////////////////////////////////////////
    int VDU::lookup(unsigned int vaddr, int &vmask,int &vstart)
    {
        VDULine *p;
        int found = 0;
        int i;

        setCurrentLine(Lines); // currentline is at the zeroth entry in the cache
        for (i=0; ((i<numlines) && (found == 0)); i++, cline++)
        {
            p = getVDULine();
            if ((vaddr >= (p->getVFNStart())) && (vaddr <= (p->getVFNEnd())))
            {
                found = 1;
                vstart = p->getVFNStart();
                vmask = p->getMask();
                break;
            }
        }
        if (found == 0)
        {
            cline = 0;
        } // note that here we assume no faults on the VDU cache.
        return found;
    }
    ///////////////////////////////////////////////////////////////////
    void VDU::indexVDUCache(unsigned int v, int &mask, int &ViewID, int &notfound, int
        &FSMID)

```

```

{
    int m,s;

    lookup(v,m,s);
    if (cline != 0)
    {
        mask = cline->contents->getMask();
        FSMID = cline->contents->getFSMID();
        ViewID = cline->contents->getViewID();
        notfound = 0;
    }
    else
        notfound = 1;
}
////////////////////////////////////////////////////////////////

```

```

CacheStats::CacheStats()
{
    int i;

    for (i=0;i<NUMACCESSTYPES;i++)
    {
        fetch[i] = 0;
        sfetch[i] = 0;
        RWDelay[i] = 0;
        L2fetch[i] = 0;
    }
    for (i=0;i<NUMMISSTYPES;i++)
    {
        miss[i] = 0;
        smiss[i] = 0;
        L2miss[i] = 0;
    }
    for (i=0;i<NUMPACCESSTYPES;i++)
        Pfetch[i] = 0;
    for (i=0;i<NUMMISSTYPES;i++)
        Pmiss[i] = 0;
    for (i=0;i<NUMACCESSTYPES;i++)
        Tfetch[i] = 0;
    for (i=0;i<NUMMISSTYPES;i++)
        Tmiss[i] = 0;
    for (i=0;i<NUMACCESSTYPES;i++)
        PTable[i] = 0;
    for (i=0;i<NUMACCESSTYPES;i++)
        PLBTable[i] = 0;
    for (i=0;i<NUMACCESSTYPES;i++)
        DTable[i] = 0;
}

```

```

FlushLine = 0;
FlushCache = 0;
FlushMemWrite = 0;
InvalidateInTraffic = 0;
InvalidateOutTraffic = 0;
InBusTraffic = 0;
OutBusTraffic = 0;
ICount = 0;
dcAccess = 0;
fsmAccess = 0;
vduAccess = 0;
plbAccess = 0;
tlbAccess = 0;
L2Access = 0;
delay = 0;
hWACFault = 0;
mWACFault = 0;

```

```

}
/////////////////////////////////////////////////////////////////
CacheStats::~CacheStats()
{
}
/////////////////////////////////////////////////////////////////
FSM::FSM (int s)
{
    cache = new Cache;
    if (s<=0) cerr << "FSM size cannot<=0" << endl;
    cache->setNumUorDsets(s);
    numlines = s;
    fsmline = new CacheLine[s];

    for (int i = 0; i < s; i++)
    {
        // the LineBits structure is not currently used in this FSM implementation
        fsmline[i].LineBits.setTag(0);
        fsmline[i].LineBits.setValidBit(0);
        fsmline[i].LineBits.setDirtyBit(0);
        fsmline[i].LineBits.setRefBit(0);
        fsmline[i].LineContents.setFSMID(0);
        fsmline[i].LineContents.setAccessType(0); //default op is a READ
    }
    // the following statements will be used in the function to load the FSM cache.
    // these are only temporarily here
    i=0;
    fsmline[i].LineContents.setCurrentState(0); // 0 - Unlocked
    fsmline[i].LineContents.setOtherCurrentState(0);
    fsmline[i].LineContents.setNewState(1); // 1 - SR
    fsmline[i].LineContents.setOtherNewState(2); // 2 - OSR
    fsmline[i].LineContents.setResult(1); // proceed

```

```

i++;
fsmline[i].LineContents.setCurrentState(1); // 1 - SR
fsmline[i].LineContents.setOtherCurrentState(2);
fsmline[i].LineContents.setNewState(3);
fsmline[i].LineContents.setOtherNewState(3); // 2 - OSR
fsmline[i].LineContents.setResult(1); // proceed
i++;
fsmline[i].LineContents.setCurrentState(2);
fsmline[i].LineContents.setOtherCurrentState(1);
fsmline[i].LineContents.setNewState(3); // 3 - MR
fsmline[i].LineContents.setOtherNewState(3);
fsmline[i].LineContents.setResult(1); // proceed
i++;
fsmline[i].LineContents.setCurrentState(3);
fsmline[i].LineContents.setOtherCurrentState(3);
fsmline[i].LineContents.setNewState(3); // 3 - MR
fsmline[i].LineContents.setOtherNewState(3);
fsmline[i].LineContents.setResult(1); // proceed
i++;
fsmline[i].LineContents.setCurrentState(4); //4 -Write
fsmline[i].LineContents.setOtherCurrentState(5); //5 - Prohibit
fsmline[i].LineContents.setNewState(4);
fsmline[i].LineContents.setOtherNewState(5);
fsmline[i].LineContents.setResult(1); // proceed
i++;
fsmline[i].LineContents.setCurrentState(5);
fsmline[i].LineContents.setOtherCurrentState(4);
fsmline[i].LineContents.setNewState(5);
fsmline[i].LineContents.setOtherNewState(4);
fsmline[i].LineContents.setResult(0); // fault
i++;
fsmline[i].LineContents.setCurrentState(5);
fsmline[i].LineContents.setOtherCurrentState(5);
fsmline[i].LineContents.setNewState(5);
fsmline[i].LineContents.setOtherNewState(5);
fsmline[i].LineContents.setResult(0); // fault

//write ops
i++;
fsmline[i].LineContents.setAccessType(1);
fsmline[i].LineContents.setCurrentState(0);
fsmline[i].LineContents.setOtherCurrentState(0);
fsmline[i].LineContents.setNewState(4);
fsmline[i].LineContents.setOtherNewState(5);
fsmline[i].LineContents.setResult(1); // proceed
i++;
fsmline[i].LineContents.setAccessType(1);
fsmline[i].LineContents.setCurrentState(1);
fsmline[i].LineContents.setOtherCurrentState(2);
fsmline[i].LineContents.setNewState(4);
fsmline[i].LineContents.setOtherNewState(5);

```

```

        fsmline[i].LineContents.setResult(1);           // proceed
        i++;
        fsmline[i].LineContents.setAccessType(1);
        fsmline[i].LineContents.setCurrentState(4);
        fsmline[i].LineContents.setOtherCurrentState(5);
        fsmline[i].LineContents.setNewState(4);
        fsmline[i].LineContents.setOtherNewState(5);
        fsmline[i].LineContents.setResult(1);         // proceed
        i++;
        fsmline[i].LineContents.setAccessType(1);
        fsmline[i].LineContents.setCurrentState(2);
        fsmline[i].LineContents.setOtherCurrentState(1);
        fsmline[i].LineContents.setResult(0);         // fault
        i++;
        fsmline[i].LineContents.setAccessType(1);
        fsmline[i].LineContents.setCurrentState(2);
        fsmline[i].LineContents.setOtherCurrentState(2);
        fsmline[i].LineContents.setResult(0);         // fault
        i++;
        fsmline[i].LineContents.setAccessType(1);
        fsmline[i].LineContents.setCurrentState(3);
        fsmline[i].LineContents.setOtherCurrentState(3);
        fsmline[i].LineContents.setResult(0);         // fault
        i++;
        fsmline[i].LineContents.setAccessType(1);
        fsmline[i].LineContents.setCurrentState(5);
        fsmline[i].LineContents.setOtherCurrentState(4);
        fsmline[i].LineContents.setResult(0);         // fault
        i++;
        fsmline[i].LineContents.setAccessType(1);
        fsmline[i].LineContents.setCurrentState(5);
        fsmline[i].LineContents.setOtherCurrentState(5);
        fsmline[i].LineContents.setResult(0);         // fault
    }
    ///////////////////////////////////////////////////////////////////

```

```

void FSM::indexFSM(int fsmid, int act, int cs, int ocs, CacheLine *&p)
{
    int condn, i;
    CacheLine *cacheptr;

    condn = 0;
    i=0;
    cacheptr = fsmline;
    while ((condn==0)&&(i<numlines))// loop through the lines of the specified FSM
    {
        condn = (fsmid == cacheptr[i].LineBits.getTag()) && (act== cacheptr[i].
LineContents.getAccessType()) && ( cs == cacheptr[i].LineContents.
getcurrentState());
        i++;
    }
}

```





# APPENDIX B

## Source Code for Conventional Lock Manager

**Filename: lock.cpp**

```
#include <iostream.h>
#include <fstream.h>
```

```
#include "object.h"
#include "list.h"
#include "hash.h"
#include "trans.h"
#include "lock.h"
```

```
void LCB::dump(ostream &os)
```

```
{
    os <<"key = " << name << " mode = " << mode <<endl;
    os <<"\nGrants list: " << endl;
    os << grants << endl;
    os <<"Pending list: " << endl;
    os << pending << endl;
}
```

```
////////////////////////////////////
```

```
void LRB::dump(ostream &os)
```

```
{
    os <<"transaction ID = " << requesterID << " lockmode = " << mode
        << " key = " << key << " listname = " << listname <<endl;
}
```

```
////////////////////////////////////
```

```
LockTable::LockTable(int s, char *name)
```

```
{
    lt = new HashTable(s);
    dbname = name;
}
```

```
////////////////////////////////////
```

```
LockTable::~LockTable()
```

```
{
}
```

```
////////////////////////////////////
```

//// Finds the LCB hash chain for the key.

```
List* LockTable::access(unsigned int key, int &index)
{
    List **p;
    List *lptr;

    index = lt->hashaddr(key); // calc. the hash address of virtual address v
    p = lt->getTable();         // retrieves the ptr to ptr to the list
    lptr = p[index];           // retrieves the ptr to the list of mappings for a hash
                                // address

    return lptr;
}
```

////////////////////////////////////  
 // Adds a Lock Request Block to either the Grants List or the Pending  
 // List. Also chains a list through the transaction's lockset.

```
void LockTable::add(List *p, LCB * c,int setpend,Transaction *tid, int lockmode)
{
    LRB * r;
    int transid, k, q;

    transid = tid->getID();
    k = c->getName();
    if (p==0)
    {
        p = new List;
        if (setpend==1)
            c->setPending(p);
        else
            c->setGrants(p);
    }
    r = new LRB; // create new LRB entry
    r->setID(transid);
    r->setMode(lockmode);
    r->setKey(k);
    if (setpend == 1)
        q=1;
    else
        q=0;
    r->setList(q);
    tid->add(r); // add to transaction's lockset
    p->add(r); // place LRB entry on either the grants or pending list
}
```

////////////////////////////////////  
 int LockTable::setLock(unsigned int key, int lmode, Transaction \*tid)
 {
 List \*lptr, \*p, \*q;
 int cs; //current state
 LCB \* lcb;

```

LRB * lrb;
int success = 0;
int i=0;
int id;

id = tid->getID();
lptr = access(key,i);
if (lptr!=0)
    lcb = (LCB *) lptr->find(key);        // retrieves a pointer to the .. if it
                                        exists
else
{
    lptr = new List;
    lt->add(i,lptr); // adds the list to the hash table
    lcb = 0;
}
if (lcb==0)
{
    lcb = new LCB;
    lcb->setName(key);
    lcb->setMode(0);
    lcb->setGrants(0);
    lcb->setPending(0);
    lptr->add(lcb); // adds the LCB to the list
}

cs = lcb->getMode();
p = lcb->getGrants();

if (cs==UNLOCK) // access unit is currently unlocked
{
    lcb->setMode(lmode);
    add(p,lcb,0,tid,lmode);
    success = 1;
}
else
{
    if (cs==READ) // access unit is currently read locked
    {
        if (lmode==WRITE) // request for a write lock
        {
            //check to see if requester already has a read lock
            // if (p!=0) //this should be always true if cs>0 , so not
            // needed
            lrb = (LRB *) p->search(id);
            if (lrb != 0) // requesting transaction has access currently
            { // here we assume write and read locks only (no intention
            // etc.)
                if (p->getCount()==1) //read locked by requester
                {
                    lcb->setMode(lmode); // upgrade to a WRITE
                }
            }
            }
        }
    }
}

```

```

                                lock
                                lrb->setMode(lmode);
                                success = 1;
                                }
                                else //read lock is held by one or more subjects
                                {
                                    q = lcb->getPending(); // place on pending queue
                                    add(q,lcb,l,tid,lmode); //place LRB entry on
                                                                pending list
                                }
                                else // conflict - read lock is held by another transaction
                                {
                                    q = lcb->getPending(); // place transaction on
                                                                pending queue
                                    add(q,lcb,l,tid,lmode); //place LRB entry on
                                                                pending list
                                }
                                }
                                if (lmode == READ) // request for a read lock
                                {
                                    add(p,lcb,0,tid,lmode);
                                    success = 1;
                                }
                                }
                                if (cs==WRITE) // if request is for a W and the unit is already locked for
                                write
                                {
                                    //check if W lock already belongs to the requesting transaction
                                    lrb = (LRB *) p->search(id);
                                    if (lrb != 0)
                                        success = 1;
                                    else // WW conflict
                                    {
                                        q = lcb->getPending(); // place transaction on pending queue
                                        add(q,lcb,l,tid,lmode); //place LRB entry on pending list
                                    }
                                }
                                }
                                return success;
                                }
                                }

////////////////////////////////////
int LockTable::releaseLocks(Transaction *tid)
{
    List *lset,*p, *lptr ;
    Node *n;
    LRB * lrb;
    LCB *lcb;
    int q, k, i;

```

```

lset = tid->getLockset();
lset->reset();
while ((n = lset->next()) != 0)
{
    lrb    =(LRB *) n->obj;
    q = lrb->getListName();
    k = lrb->getKey();
    lptr = access(k,i);
    lcb = (LCB *) lptr->find(k); // retrieves a pointer to the .. if it exists
    if (lcb == 0)
        cout << "error in release routine\n";
    if (q == 0)
        p = lcb->getGrants();
    else
        p = lcb->getPending();

    p->remove(lrb);
    lset->remove(n);
    lset->del(n);
}
return l;
}
////////////////////////////////////////////////////////////////

void
LockTable::dump(ostream &os)
{
    os << "Lock Table for file : " << dbname << "\n\n";
    os << lt;
}
////////////////////////////////////////////////////////////////

```

### Filename : init.cpp

```

#include <iostream.h>
#include <fstream.h>

```

```

#include "object.h"
#include "list.h"
#include "hash.h"
#include "trans.h"
#include "lock.h"

```

```

Transaction::Transaction(int ident)
{
    id = ident;
    lockset = 0;
}

```

```

////////////////////////////////////
Transaction::~Transaction()
{
}

```

```

////////////////////////////////////
void Transaction::add(Object * r)
{
    if (lockset == 0)
        lockset = new List;
    lockset->add(r);
}

```

```

////////////////////////////////////
Filename: serial.cpp

```

```

#include <iostream.h>
#include <fstream.h>

```

```

#include "object.h"
#include "list.h"
#include "hash.h"
#include "trans.h"
#include "lock.h"

```

```

void t1(LockTable * Locktable, Transaction *t[])
{

```

```

    Locktable->setLock(0x1f060,READ,t[1]);
    Locktable->setLock(0x1f0e0,READ,t[1]);
    Locktable->setLock(0x242f8,READ,t[1]);
    Locktable->setLock(0x243f8,READ,t[1]);
    Locktable->setLock(0x242f8,WRITE,t[1]);
    Locktable->setLock(0x243f8,WRITE,t[1]);
    Locktable->setLock(0x42d48,READ,t[1]);
    Locktable->setLock(0x44148,READ,t[1]);
    Locktable->setLock(0x41948,READ,t[1]);
    Locktable->setLock(0x3dd48,READ,t[1]);
    Locktable->setLock(0xcfb0,WRITE,t[1]);
    Locktable->setLock(0xcfb0,READ,t[1]);
    Locktable->setLock(0xcfd0,WRITE,t[1]);
    Locktable->setLock(0xd09e0,WRITE,t[1]);
    Locktable->setLock(0xe3e90,WRITE,t[1]);
    Locktable->setLock(0xe3e98,WRITE,t[1]);
    Locktable->setLock(0xe3e98,READ,t[1]);
    Locktable->setLock(0xe4698,WRITE,t[1]);
    Locktable->setLock(0xe56a0,WRITE,t[1]);
    Locktable->setLock(0xb5ed4,READ,t[1]);
    Locktable->setLock(0xb5fd4,READ,t[1]);

```

```

Locktable->setLock(0x10e220,READ,t[1]);
Locktable->setLock(0x10f420,READ,t[1]);
Locktable->setLock(0x10d020,READ,t[1]);
Locktable->setLock(0x10d220,READ,t[1]);
Locktable->setLock(0x10d020,WRITE,t[1]);
Locktable->setLock(0x10d220,WRITE,t[1]);
Locktable->setLock(0x17bba0,WRITE,t[1]);
Locktable->setLock(0x17bba0,READ,t[1]);
Locktable->setLock(0x17bca0,WRITE,t[1]);
Locktable->setLock(0x17cae0,WRITE,t[1]);
Locktable->setLock(0xb5f54,READ,t[1]);
Locktable->setLock(0x10f620,READ,t[1]);
Locktable->setLock(0x10f420,WRITE,t[1]);
Locktable->setLock(0x10f620,WRITE,t[1]);
Locktable->setLock(0x17cae0,READ,t[1]);
Locktable->setLock(0xb6054,READ,t[1]);
Locktable->setLock(0x10c020,READ,t[1]);
Locktable->setLock(0x10c020,WRITE,t[1]);
Locktable->setLock(0x110620,READ,t[1]);
Locktable->setLock(0x110620,WRITE,t[1]);
Locktable->setLock(0x110820,READ,t[1]);
Locktable->setLock(0x110820,WRITE,t[1]);

Locktable->releaseLocks(t[1]);
}

void t2(LockTable * Locktable, Transaction *t[])
{
    if (Locktable==0)
    {
        cout << "error message" << endl;
        return;
    }
    Locktable->setLock(0x1f060,READ,t[2]);

    Locktable->setLock(0x1f0e0,READ,t[2]);
    Locktable->setLock(0x242f8,READ,t[2]);
    Locktable->setLock(0x24478,READ,t[2]);
    Locktable->setLock(0x42d48,READ,t[2]);
    Locktable->setLock(0x44148,READ,t[2]);
    Locktable->setLock(0x41948,READ,t[2]);
    Locktable->setLock(0x40548,READ,t[2]);
    Locktable->setLock(0x1f060,WRITE,t[2]);
    Locktable->setLock(0x1f0e0,WRITE,t[2]);
    Locktable->setLock(0x242f8,WRITE,t[2]);
    Locktable->setLock(0x24478,WRITE,t[2]);
    Locktable->setLock(0x40548,WRITE,t[2]);
    Locktable->setLock(0x1900f0,WRITE,t[2]);
    Locktable->setLock(0x1900f0,READ,t[2]);
    Locktable->setLock(0x1901f0,WRITE,t[2]);
    Locktable->setLock(0x191030,WRITE,t[2]);

```

```

        Locktable->releaseLocks(t[2]);
    }

void t3(LockTable * Locktable, Transaction *t[])
{
    Locktable->setLock(0x42d48,READ,t[3]);
    Locktable->setLock(0x44148,READ,t[3]);
    Locktable->setLock(0x3b548,READ,t[3]);
    Locktable->setLock(0x3c548,READ,t[3]);
    Locktable->setLock(0x3a148,READ,t[3]);
    Locktable->setLock(0xcfb0,READ,t[3]);
    Locktable->setLock(0xcff0,READ,t[3]);
    Locktable->setLock(0x17bba0,READ,t[3]);
    Locktable->setLock(0x17bf20,READ,t[3]);
    Locktable->setLock(0x17bfe0,READ,t[3]);
    Locktable->setLock(0x17bee0,READ,t[3]);
    Locktable->setLock(0x17bfa0,READ,t[3]);
    Locktable->setLock(0x17c060,READ,t[3]);
    Locktable->setLock(0x17bf60,READ,t[3]);

    Locktable->releaseLocks(t[3]);
}

void t4(LockTable * Locktable, Transaction *t[])
{
    Locktable->setLock(0xe3e98,READ,t[4]);
    Locktable->setLock(0xe3e90,READ,t[4]);
    Locktable->setLock(0xe4698,READ,t[4]);
    Locktable->setLock(0xe4ea0,READ,t[4]);
    Locktable->setLock(0xe3e98,WRITE,t[4]);
    Locktable->setLock(0xe4ea0,WRITE,t[4]);
    Locktable->setLock(0xcfb0,READ,t[4]);
    Locktable->setLock(0xcff0,READ,t[4]);
    Locktable->setLock(0xcfb0,WRITE,t[4]);
    Locktable->setLock(0xcff0,WRITE,t[4]);
    Locktable->setLock(0x17bba0,READ,t[4]);
    Locktable->setLock(0x17bfa0,READ,t[4]);
    Locktable->setLock(0x17bba0,WRITE,t[4]);
    Locktable->setLock(0x17bfa0,WRITE,t[4]);
    Locktable->setLock(0x17bee0,READ,t[4]);
    Locktable->setLock(0x17bee0,WRITE,t[4]);
    Locktable->setLock(0x17bf60,READ,t[4]);
    Locktable->setLock(0x17bf60,WRITE,t[4]);
    Locktable->setLock(0x17c020,READ,t[4]);
    Locktable->setLock(0x17c020,WRITE,t[4]);
    Locktable->setLock(0x17bf20,READ,t[4]);
    Locktable->setLock(0x17bf20,WRITE,t[4]);
    Locktable->setLock(0x42d48,READ,t[4]);
}

```



```

    Locktable->setLock(0x44148,READ,t[4]);
    Locktable->setLock(0x41948,READ,t[4]);
    Locktable->setLock(0x40548,READ,t[4]);
    Locktable->setLock(0x40548,WRITE,t[4]);

    Locktable->releaseLocks(t[4]);
}

void t5(LockTable * Locktable, Transaction *t[])
{
    Locktable->setLock(0x242f8,READ,t[5]);
    Locktable->setLock(0x243f8,READ,t[5]);
    Locktable->setLock(0x17bba0,READ,t[5]);
    Locktable->setLock(0x17bee0,READ,t[5]);
    Locktable->setLock(0x10e220,READ,t[5]);
    Locktable->setLock(0x10f420,READ,t[5]);
    Locktable->setLock(0x10d020,READ,t[5]);
    Locktable->setLock(0x10d220,READ,t[5]);
    Locktable->setLock(0x10f620,READ,t[5]);

    Locktable->releaseLocks(t[5]);
}

void t6(LockTable * Locktable, Transaction *t[])
{
    Locktable->setLock(0x1f060,READ,t[6]);
    Locktable->setLock(0x1f0e0,READ,t[6]);
    Locktable->setLock(0x242f8,READ,t[6]);
    Locktable->setLock(0x24478,READ,t[6]);
    Locktable->setLock(0x242f8,WRITE,t[6]);
    Locktable->setLock(0x24478,WRITE,t[6]);
    Locktable->setLock(0x42d48,READ,t[6]);
    Locktable->setLock(0x44148,READ,t[6]);
    Locktable->setLock(0x41948,READ,t[6]);
    Locktable->setLock(0x40548,READ,t[6]);
    Locktable->setLock(0xcfbc0,READ,t[6]);
    Locktable->setLock(0xd09e0,READ,t[6]);
    Locktable->setLock(0xcfbc0,WRITE,t[6]);
    Locktable->setLock(0xcfdc0,WRITE,t[6]);
    Locktable->setLock(0xd09e0,WRITE,t[6]);
    Locktable->setLock(0xe3e98,READ,t[6]);
    Locktable->setLock(0xe3e90,READ,t[6]);
    Locktable->setLock(0xe56a0,READ,t[6]);
    Locktable->setLock(0xe3e98,WRITE,t[6]);
    Locktable->setLock(0xe46a0,WRITE,t[6]);
    Locktable->setLock(0xe56a8,WRITE,t[6]);
    Locktable->setLock(0xe3e90,WRITE,t[6]);
    Locktable->setLock(0xb5ed4,READ,t[6]);
    Locktable->setLock(0xb6054,READ,t[6]);
}

```

```

Locktable->setLock(0x10e220,READ,t[6]);
Locktable->setLock(0x10f420,READ,t[6]);
Locktable->setLock(0x110620,READ,t[6]);
Locktable->setLock(0x110820,READ,t[6]);
Locktable->setLock(0x110620,WRITE,t[6]);
Locktable->setLock(0x110820,WRITE,t[6]);
Locktable->setLock(0x17bba0,READ,t[6]);
Locktable->setLock(0x17cae0,READ,t[6]);
Locktable->setLock(0x17bba0,WRITE,t[6]);
Locktable->setLock(0x17bca0,WRITE,t[6]);
Locktable->setLock(0x17cae0,WRITE,t[6]);
Locktable->setLock(0xb5fd4,READ,t[6]);
Locktable->setLock(0x10e420,READ,t[6]);
Locktable->setLock(0x10e220,WRITE,t[6]);
Locktable->setLock(0x10e420,WRITE,t[6]);
Locktable->setLock(0xb5f54,READ,t[6]);
Locktable->setLock(0x10f620,READ,t[6]);
Locktable->setLock(0x10f420,WRITE,t[6]);
Locktable->setLock(0x10f620,WRITE,t[6]);
Locktable->setLock(0x17bce0,WRITE,t[6]);
Locktable->setLock(0x10c020,READ,t[6]);
Locktable->setLock(0x10c020,WRITE,t[6]);
Locktable->setLock(0x17bbe0,WRITE,t[6]);
Locktable->setLock(0x17bbe0,READ,t[6]);
Locktable->setLock(0x17cb20,WRITE,t[6]);
Locktable->setLock(0x17cb20,READ,t[6]);

Locktable->releaseLocks(t[6]);
}

void t7(LockTable * Locktable, Transaction *t[])
{
    Locktable->setLock(0x42d48,READ,t[7]);
    Locktable->setLock(0x44148,READ,t[7]);
    Locktable->setLock(0x3b548,READ,t[7]);
    Locktable->setLock(0x3c548,READ,t[7]);
    Locktable->setLock(0x38d48,READ,t[7]);
    Locktable->setLock(0xcfb0,READ,t[7]);
    Locktable->setLock(0xcfe0,READ,t[7]);
    Locktable->setLock(0x17bba0,READ,t[7]);
    Locktable->setLock(0x17bbe0,READ,t[7]);
    Locktable->setLock(0x17bf20,READ,t[7]);
    Locktable->setLock(0x17bf60,READ,t[7]);
    Locktable->setLock(0x17c1e0,READ,t[7]);
    Locktable->setLock(0x17c0e0,READ,t[7]);
    Locktable->setLock(0x17c220,READ,t[7]);
    Locktable->setLock(0x17c060,READ,t[7]);
    Locktable->setLock(0x17c160,READ,t[7]);
    Locktable->setLock(0x17c120,READ,t[7]);
    Locktable->setLock(0x17bfe0,READ,t[7]);
}

```

```

    Locktable->releaseLocks(t[7]);
}

void t8(LockTable * Locktable, Transaction *t[])
{
    Locktable->setLock(0x42d48,READ,t[8]);
    Locktable->setLock(0x44148,READ,t[8]);
    Locktable->setLock(0x3b548,READ,t[8]);
    Locktable->setLock(0x3c548,READ,t[8]);
    Locktable->setLock(0x38d48,READ,t[8]);
    Locktable->setLock(0xcfb0,READ,t[8]);
    Locktable->setLock(0xcfe0,READ,t[8]);
    Locktable->setLock(0x17bba0,READ,t[8]);
    Locktable->setLock(0x17bbe0,READ,t[8]);
    Locktable->setLock(0x17bee0,READ,t[8]);
    Locktable->setLock(0x17c160,READ,t[8]);
    Locktable->setLock(0x17c120,READ,t[8]);
    Locktable->setLock(0x17c020,READ,t[8]);
    Locktable->setLock(0x17c220,READ,t[8]);
    Locktable->setLock(0x17c0a0,READ,t[8]);

    Locktable->releaseLocks(t[8]);
}

void t9(LockTable * Locktable, Transaction *t[])
{
    Locktable->setLock(0xe3e98,READ,t[9]);
    Locktable->setLock(0xe3e90,READ,t[9]);
    Locktable->setLock(0xe46a0,READ,t[9]);
    Locktable->setLock(0xe4ea0,READ,t[9]);
    Locktable->setLock(0xe3e98,WRITE,t[9]);
    Locktable->setLock(0xe4ea0,WRITE,t[9]);
    Locktable->setLock(0xcfb0,READ,t[9]);
    Locktable->setLock(0xcff0,READ,t[9]);
    Locktable->setLock(0xcfb0,WRITE,t[9]);
    Locktable->setLock(0xcff0,WRITE,t[9]);
    Locktable->setLock(0x17bba0,READ,t[9]);
    Locktable->setLock(0x17bbe0,READ,t[9]);
    Locktable->setLock(0x17bfa0,READ,t[9]);
    Locktable->setLock(0x17bba0,WRITE,t[9]);
    Locktable->setLock(0x17bfa0,WRITE,t[9]);
    Locktable->setLock(0x17c120,READ,t[9]);
    Locktable->setLock(0x17c120,WRITE,t[9]);
    Locktable->setLock(0x17c0a0,READ,t[9]);
    Locktable->setLock(0x17c0a0,WRITE,t[9]);
    Locktable->setLock(0x17c020,READ,t[9]);
    Locktable->setLock(0x17c020,WRITE,t[9]);
    Locktable->setLock(0x17c060,READ,t[9]);
}

```

```

    Locktable->setLock(0x17c060,WRITE,t[9]);
    Locktable->setLock(0x42d48,READ,t[9]);
    Locktable->setLock(0x44148,READ,t[9]);
    Locktable->setLock(0x3b548,READ,t[9]);
    Locktable->setLock(0x3c548,READ,t[9]);
    Locktable->setLock(0x38d48,READ,t[9]);
    Locktable->setLock(0x38d48,WRITE,t[9]);

    Locktable->releaseLocks(t[9]);
}

void t10(LockTable * Locktable, Transaction *t[])
{
    Locktable->setLock(0xe3e98,READ,t[10]);
    Locktable->setLock(0xe3e90,READ,t[10]);
    Locktable->setLock(0xe4698,READ,t[10]);
    Locktable->setLock(0xe4ea0,READ,t[10]);
    Locktable->setLock(0xe3e98,WRITE,t[10]);
    Locktable->setLock(0xe4ea0,WRITE,t[10]);
    Locktable->setLock(0xcffc0,READ,t[10]);
    Locktable->setLock(0xcffc0,READ,t[10]);
    Locktable->setLock(0xcffc0,WRITE,t[10]);
    Locktable->setLock(0xcffc0,WRITE,t[10]);
    Locktable->setLock(0x17bba0,READ,t[10]);
    Locktable->setLock(0x17bbe0,READ,t[10]);
    Locktable->setLock(0x17bfa0,READ,t[10]);
    Locktable->setLock(0x17bba0,WRITE,t[10]);
    Locktable->setLock(0x17bfa0,WRITE,t[10]);
    Locktable->setLock(0x17bee0,READ,t[10]);
    Locktable->setLock(0x17bee0,WRITE,t[10]);
    Locktable->setLock(0x17c220,READ,t[10]);
    Locktable->setLock(0x17bbe0,WRITE,t[10]);
    Locktable->setLock(0x17c220,WRITE,t[10]);
    Locktable->setLock(0x17bfe0,READ,t[10]);
    Locktable->setLock(0x17bfe0,WRITE,t[10]);
    Locktable->setLock(0x17c0e0,READ,t[10]);
    Locktable->setLock(0x17c0e0,WRITE,t[10]);
    Locktable->setLock(0x17c0a0,READ,t[10]);
    Locktable->setLock(0x17c0a0,WRITE,t[10]);
    Locktable->setLock(0x42d48,READ,t[10]);
    Locktable->setLock(0x44148,READ,t[10]);
    Locktable->setLock(0x3b548,READ,t[10]);
    Locktable->setLock(0x3c548,READ,t[10]);
    Locktable->setLock(0x38d48,READ,t[10]);
    Locktable->setLock(0x38d48,WRITE,t[10]);

    Locktable->releaseLocks(t[10]);
}

void t11(LockTable * Locktable, Transaction *t[])

```

```

{
    Locktable->setLock(0x42d48,READ,t[11]);
    Locktable->setLock(0x44148,READ,t[11]);
    Locktable->setLock(0x41948,READ,t[11]);
    Locktable->setLock(0x3f148,READ,t[11]);
    Locktable->setLock(0xcfb0,READ,t[11]);
    Locktable->setLock(0xcfe0,READ,t[11]);
    Locktable->setLock(0x17bba0,READ,t[11]);
    Locktable->setLock(0x17bbe0,READ,t[11]);
    Locktable->setLock(0x17c0a0,READ,t[11]);
    Locktable->setLock(0x17bfa0,READ,t[11]);
    Locktable->setLock(0x17c1e0,READ,t[11]);
    Locktable->setLock(0x17c1a0,READ,t[11]);
    Locktable->setLock(0x17c060,READ,t[11]);
    Locktable->setLock(0x17bf20,READ,t[11]);
    Locktable->setLock(0x17c120,READ,t[11]);

    Locktable->releaseLocks(t[11]);
}

void t12(LockTable * Locktable, Transaction *t[])
{
    Locktable->setLock(0xe3e98,READ,t[12]);
    Locktable->setLock(0xe3e90,READ,t[12]);
    Locktable->setLock(0xe4698,READ,t[12]);
    Locktable->setLock(0xe4ea0,READ,t[12]);
    Locktable->setLock(0xe3e98,WRITE,t[12]);
    Locktable->setLock(0xe4ea0,WRITE,t[12]);
    Locktable->setLock(0xcfb0,READ,t[12]);
    Locktable->setLock(0xcfc0,READ,t[12]);
    Locktable->setLock(0xcfb0,WRITE,t[12]);
    Locktable->setLock(0xcfc0,WRITE,t[12]);
    Locktable->setLock(0x17bba0,READ,t[12]);
    Locktable->setLock(0x17bbe0,READ,t[12]);
    Locktable->setLock(0x17c1a0,READ,t[12]);
    Locktable->setLock(0x17bbe0,WRITE,t[12]);
    Locktable->setLock(0x17c1a0,WRITE,t[12]);
    Locktable->setLock(0x17bf20,READ,t[12]);
    Locktable->setLock(0x17bba0,WRITE,t[12]);
    Locktable->setLock(0x17bf20,WRITE,t[12]);
    Locktable->setLock(0x17c020,READ,t[12]);
    Locktable->setLock(0x17c020,WRITE,t[12]);
    Locktable->setLock(0x17bfe0,READ,t[12]);
    Locktable->setLock(0x17bfe0,WRITE,t[12]);
    Locktable->setLock(0x17bee0,READ,t[12]);
    Locktable->setLock(0x17bee0,WRITE,t[12]);
    Locktable->setLock(0x17bfa0,READ,t[12]);
    Locktable->setLock(0x17bfa0,WRITE,t[12]);
    Locktable->setLock(0x17c060,READ,t[12]);
}

```

```

    Locktable->setLock(0x17c060,WRITE,t[12]);
    Locktable->setLock(0x42d48,READ,t[12]);
    Locktable->setLock(0x44148,READ,t[12]);
    Locktable->setLock(0x41948,READ,t[12]);
    Locktable->setLock(0x3f148,READ,t[12]);
    Locktable->setLock(0x3f148,WRITE,t[12]);

    Locktable->releaseLocks(t[12]);
}

void t13(LockTable * Locktable, Transaction *t[])
{

    Locktable->setLock(0x42d48,READ,t[13]);
    Locktable->setLock(0x44148,READ,t[13]);
    Locktable->setLock(0x41948,READ,t[13]);
    Locktable->setLock(0xcfb0,READ,t[13]);
    Locktable->setLock(0xcfe0,READ,t[13]);
    Locktable->setLock(0x17bba0,READ,t[13]);
    Locktable->setLock(0x17bbe0,READ,t[13]);
    Locktable->setLock(0x17bee0,READ,t[13]);
    Locktable->setLock(0x17c060,READ,t[13]);
    Locktable->setLock(0x17c0e0,READ,t[13]);
    Locktable->setLock(0x17c120,READ,t[13]);
    Locktable->setLock(0x17c0a0,READ,t[13]);
    Locktable->setLock(0x17bfa0,READ,t[13]);
    Locktable->setLock(0x17c1a0,READ,t[13]);

    Locktable->releaseLocks(t[13]);
}

void t14(LockTable * Locktable, Transaction *t[])
{

    Locktable->setLock(0x42d48,READ,t[14]);
    Locktable->setLock(0x44148,READ,t[14]);
    Locktable->setLock(0x41948,READ,t[14]);
    Locktable->setLock(0x40548,READ,t[14]);
    Locktable->setLock(0xcfb0,READ,t[14]);
    Locktable->setLock(0xcfe0,READ,t[14]);
    Locktable->setLock(0x17bba0,READ,t[14]);
    Locktable->setLock(0x17bbe0,READ,t[14]);
    Locktable->setLock(0x17bee0,READ,t[14]);
    Locktable->setLock(0x17c1a0,READ,t[14]);
    Locktable->setLock(0x17c1e0,READ,t[14]);
    Locktable->setLock(0x17c0e0,READ,t[14]);
    Locktable->setLock(0x17bf20,READ,t[14]);
    Locktable->setLock(0x17bfa0,READ,t[14]);
    Locktable->setLock(0x17c060,READ,t[14]);

    Locktable->releaseLocks(t[14]);
}

```

```

}

void t15(LockTable * Locktable, Transaction *t[])
{
    Locktable->setLock(0x1f060,READ,t[15]);
    Locktable->setLock(0x1f0e0,READ,t[15]);
    Locktable->setLock(0x242f8,READ,t[15]);
    Locktable->setLock(0x243f8,READ,t[15]);
    Locktable->setLock(0x242f8,WRITE,t[15]);
    Locktable->setLock(0x243f8,WRITE,t[15]);
    Locktable->setLock(0x42d48,READ,t[15]);
    Locktable->setLock(0x44148,READ,t[15]);
    Locktable->setLock(0x41948,READ,t[15]);
    Locktable->setLock(0xcfb0,READ,t[15]);
    Locktable->setLock(0xd09e0,READ,t[15]);
    Locktable->setLock(0xcfb0,WRITE,t[15]);
    Locktable->setLock(0xcfdc0,WRITE,t[15]);
    Locktable->setLock(0xd09e0,WRITE,t[15]);
    Locktable->setLock(0xe3e98,READ,t[15]);
    Locktable->setLock(0xe3e90,READ,t[15]);
    Locktable->setLock(0xe56a8,READ,t[15]);
    Locktable->setLock(0xe3ea0,WRITE,t[15]);
    Locktable->setLock(0xe3ea0,READ,t[15]);
    Locktable->setLock(0xe46a0,WRITE,t[15]);
    Locktable->setLock(0xe56a8,WRITE,t[15]);
    Locktable->setLock(0xe3e90,WRITE,t[15]);
    Locktable->setLock(0xb5ed4,READ,t[15]);
    Locktable->setLock(0xb5fd4,READ,t[15]);
    Locktable->setLock(0x10e220,READ,t[15]);
    Locktable->setLock(0x10f420,READ,t[15]);
    Locktable->setLock(0x10c020,READ,t[15]);
    Locktable->setLock(0x10c020,WRITE,t[15]);
    Locktable->setLock(0x17bba0,READ,t[15]);
    Locktable->setLock(0x17bbe0,READ,t[15]);
    Locktable->setLock(0x17cb20,READ,t[15]);
    Locktable->setLock(0x17bbe0,WRITE,t[15]);
    Locktable->setLock(0x17bce0,WRITE,t[15]);
    Locktable->setLock(0x17cb20,WRITE,t[15]);
    Locktable->setLock(0x17bba0,WRITE,t[15]);
    Locktable->setLock(0xb6054,READ,t[15]);
    Locktable->setLock(0x10f620,READ,t[15]);
    Locktable->setLock(0x10f420,WRITE,t[15]);
    Locktable->setLock(0x10f620,WRITE,t[15]);
    Locktable->setLock(0x110620,READ,t[15]);
    Locktable->setLock(0x110820,READ,t[15]);
    Locktable->setLock(0x110620,WRITE,t[15]);
    Locktable->setLock(0x110820,WRITE,t[15]);
    Locktable->setLock(0xb5f54,READ,t[15]);
    Locktable->setLock(0xb60d4,READ,t[15]);
    Locktable->setLock(0x10d020,READ,t[15]);
}

```

```

    Locktable->setLock(0x10d220,READ,t[15]);
    Locktable->setLock(0x10d020,WRITE,t[15]);
    Locktable->setLock(0x10d220,WRITE,t[15]);
    Locktable->setLock(0x10e420,READ,t[15]);
    Locktable->setLock(0x10e220,WRITE,t[15]);
    Locktable->setLock(0x10e420,WRITE,t[15]);
    Locktable->setLock(0x17bd20,WRITE,t[15]);

    Locktable->releaseLocks(t[15]);
}

void t16(LockTable * Locktable, Transaction *t[])
{

    Locktable->setLock(0x242f8,READ,t[16]);
    Locktable->setLock(0x243f8,READ,t[16]);
    Locktable->setLock(0x17bba0,READ,t[16]);
    Locktable->setLock(0x17bbe0,READ,t[16]);
    Locktable->setLock(0x17bfa0,READ,t[16]);
    Locktable->setLock(0x10e220,READ,t[16]);
    Locktable->setLock(0x10f420,READ,t[16]);
    Locktable->setLock(0x110620,READ,t[16]);
    Locktable->setLock(0x110820,READ,t[16]);
    Locktable->setLock(0x17bf60,READ,t[16]);
    Locktable->setLock(0x10c020,READ,t[16]);
    Locktable->setLock(0x17c3e0,READ,t[16]);
    Locktable->setLock(0x10f620,READ,t[16]);

    Locktable->releaseLocks(t[16]);
}

void t17(LockTable * Locktable, Transaction *t[])
{

    Locktable->setLock(0xe3e98,READ,t[17]);
    Locktable->setLock(0xe3e90,READ,t[17]);
    Locktable->setLock(0xe3ea0,READ,t[17]);
    Locktable->setLock(0xe4698,READ,t[17]);
    Locktable->setLock(0xe4ea0,READ,t[17]);
    Locktable->setLock(0xe3e98,WRITE,t[17]);
    Locktable->setLock(0xe4ea0,WRITE,t[17]);
    Locktable->setLock(0xcffc0,READ,t[17]);
    Locktable->setLock(0xcffe0,READ,t[17]);
    Locktable->setLock(0xcffc0,WRITE,t[17]);
    Locktable->setLock(0xcffe0,WRITE,t[17]);
    Locktable->setLock(0x17bba0,READ,t[17]);
    Locktable->setLock(0x17bbe0,READ,t[17]);
    Locktable->setLock(0x17bf60,READ,t[17]);
    Locktable->setLock(0x17bba0,WRITE,t[17]);
    Locktable->setLock(0x17bf60,WRITE,t[17]);
    Locktable->setLock(0x17bf20,READ,t[17]);
}

```



```

Locktable->setLock(0x17bf20,WRITE,t[17]);
Locktable->setLock(0x17bfa0,READ,t[17]);
Locktable->setLock(0x17bfa0,WRITE,t[17]);
Locktable->setLock(0x17bee0,READ,t[17]);
Locktable->setLock(0x17bee0,WRITE,t[17]);
Locktable->setLock(0x17c3e0,READ,t[17]);
Locktable->setLock(0x17bbe0,WRITE,t[17]);
Locktable->setLock(0x17c3e0,WRITE,t[17]);
Locktable->setLock(0x17c3a0,READ,t[17]);
Locktable->setLock(0x17c3a0,WRITE,t[17]);
Locktable->setLock(0x17c360,READ,t[17]);
Locktable->setLock(0x17c360,WRITE,t[17]);
Locktable->setLock(0x42d48,READ,t[17]);
Locktable->setLock(0x44148,READ,t[17]);
Locktable->setLock(0x41948,READ,t[17]);
Locktable->setLock(0x41948,WRITE,t[17]);

Locktable->releaseLocks(t[17]);
}

void t18(LockTable * Locktable, Transaction *t[])
{

Locktable->setLock(0x1f060,READ,t[18]);
Locktable->setLock(0x1f0e0,READ,t[18]);
Locktable->setLock(0x242f8,READ,t[18]);
Locktable->setLock(0x24478,READ,t[18]);
Locktable->setLock(0x242f8,WRITE,t[18]);
Locktable->setLock(0x24478,WRITE,t[18]);
Locktable->setLock(0x42d48,READ,t[18]);
Locktable->setLock(0x44148,READ,t[18]);
Locktable->setLock(0x3b548,READ,t[18]);
Locktable->setLock(0x3c948,READ,t[18]);
Locktable->setLock(0xcfb0,READ,t[18]);
Locktable->setLock(0xd09e0,READ,t[18]);
Locktable->setLock(0xcfb0,WRITE,t[18]);
Locktable->setLock(0xcfdc0,WRITE,t[18]);
Locktable->setLock(0xd09e0,WRITE,t[18]);
Locktable->setLock(0xe3e98,READ,t[18]);
Locktable->setLock(0xe3e90,READ,t[18]);
Locktable->setLock(0xe3ea0,READ,t[18]);
Locktable->setLock(0xe56a8,READ,t[18]);
Locktable->setLock(0xe3ea0,WRITE,t[18]);
Locktable->setLock(0xe46a8,WRITE,t[18]);
Locktable->setLock(0xe56b0,WRITE,t[18]);
Locktable->setLock(0xe3e90,WRITE,t[18]);
Locktable->setLock(0xb5ed4,READ,t[18]);
Locktable->setLock(0xb5f54,READ,t[18]);
Locktable->setLock(0x10e220,READ,t[18]);
Locktable->setLock(0x10f420,READ,t[18]);
Locktable->setLock(0x110620,READ,t[18]);

```

```

Locktable->setLock(0x110820,READ,t[18]);
Locktable->setLock(0x110620,WRITE,t[18]);
Locktable->setLock(0x110820,WRITE,t[18]);
Locktable->setLock(0x17bba0,READ,t[18]);
Locktable->setLock(0x17bbe0,READ,t[18]);
Locktable->setLock(0x17cb20,READ,t[18]);
Locktable->setLock(0x17bc20,WRITE,t[18]);
Locktable->setLock(0x17bc20,READ,t[18]);
Locktable->setLock(0x17bd20,WRITE,t[18]);
Locktable->setLock(0x17cb20,WRITE,t[18]);
Locktable->setLock(0x17bba0,WRITE,t[18]);
Locktable->setLock(0xb6054,READ,t[18]);
Locktable->setLock(0x10e420,READ,t[18]);
Locktable->setLock(0x10e220,WRITE,t[18]);
Locktable->setLock(0x10e420,WRITE,t[18]);
Locktable->setLock(0x17cb60,WRITE,t[18]);
Locktable->setLock(0x10f620,READ,t[18]);
Locktable->setLock(0x10f420,WRITE,t[18]);
Locktable->setLock(0x10f620,WRITE,t[18]);
Locktable->setLock(0x17cb60,READ,t[18]);
Locktable->setLock(0x10c020,READ,t[18]);
Locktable->setLock(0x10c020,WRITE,t[18]);
Locktable->setLock(0xb5fd4,READ,t[18]);
Locktable->setLock(0x10d020,READ,t[18]);
Locktable->setLock(0x10d220,READ,t[18]);
Locktable->setLock(0x10d020,WRITE,t[18]);
Locktable->setLock(0x10d220,WRITE,t[18]);

Locktable->releaseLocks(t[18]);
}

void t19(LockTable * Locktable, Transaction *t[])
{

Locktable->setLock(0x1f060,READ,t[19]);
Locktable->setLock(0x1f0e0,READ,t[19]);
Locktable->setLock(0x242f8,READ,t[19]);
Locktable->setLock(0x24478,READ,t[19]);
Locktable->setLock(0x242f8,WRITE,t[19]);
Locktable->setLock(0x24478,WRITE,t[19]);
Locktable->setLock(0x42d48,READ,t[19]);
Locktable->setLock(0x44148,READ,t[19]);
Locktable->setLock(0x3b548,READ,t[19]);
Locktable->setLock(0x3c948,READ,t[19]);
Locktable->setLock(0xcfb0,READ,t[19]);
Locktable->setLock(0xd09e0,READ,t[19]);
Locktable->setLock(0xcfb0,WRITE,t[19]);
Locktable->setLock(0xcfdc0,WRITE,t[19]);
Locktable->setLock(0xd09e0,WRITE,t[19]);
Locktable->setLock(0xe3e98,READ,t[19]);
Locktable->setLock(0xe3e90,READ,t[19]);

```

```

Locktable->setLock(0xe3ea0,READ,t[19]);
Locktable->setLock(0xe56b0,READ,t[19]);
Locktable->setLock(0xe3ea8,WRITE,t[19]);
Locktable->setLock(0xe3ea8,READ,t[19]);
Locktable->setLock(0xe46a8,WRITE,t[19]);
Locktable->setLock(0xe56b0,WRITE,t[19]);
Locktable->setLock(0xe3e90,WRITE,t[19]);
Locktable->setLock(0xb5ed4,READ,t[19]);
Locktable->setLock(0xb60d4,READ,t[19]);
Locktable->setLock(0x10e220,READ,t[19]);
Locktable->setLock(0x10f420,READ,t[19]);
Locktable->setLock(0x110620,READ,t[19]);
Locktable->setLock(0x110820,READ,t[19]);
Locktable->setLock(0x110620,WRITE,t[19]);
Locktable->setLock(0x110820,WRITE,t[19]);
Locktable->setLock(0x17bba0,READ,t[19]);
Locktable->setLock(0x17bc20,READ,t[19]);
Locktable->setLock(0x17cb60,READ,t[19]);
Locktable->setLock(0x17bc20,WRITE,t[19]);
Locktable->setLock(0x17bd20,WRITE,t[19]);
Locktable->setLock(0x17cb60,WRITE,t[19]);
Locktable->setLock(0x17bba0,WRITE,t[19]);
Locktable->setLock(0xb5fd4,READ,t[19]);
Locktable->setLock(0x10f620,READ,t[19]);
Locktable->setLock(0x10f420,WRITE,t[19]);
Locktable->setLock(0x10f620,WRITE,t[19]);
Locktable->setLock(0xb6054,READ,t[19]);
Locktable->setLock(0x10d020,READ,t[19]);
Locktable->setLock(0x10d220,READ,t[19]);
Locktable->setLock(0x10d020,WRITE,t[19]);
Locktable->setLock(0x10d220,WRITE,t[19]);
Locktable->setLock(0xb5f54,READ,t[19]);
Locktable->setLock(0x17bd60,WRITE,t[19]);
Locktable->setLock(0x10e420,READ,t[19]);
Locktable->setLock(0x10e220,WRITE,t[19]);
Locktable->setLock(0x10e420,WRITE,t[19]);
Locktable->setLock(0x17bc60,WRITE,t[19]);
Locktable->setLock(0x17bc60,READ,t[19]);
Locktable->setLock(0x10c020,READ,t[19]);
Locktable->setLock(0x10c020,WRITE,t[19]);
Locktable->setLock(0x17cba0,WRITE,t[19]);
Locktable->setLock(0x17cba0,READ,t[19]);

Locktable->releaseLocks(t[19]);
}

void t20(LockTable * Locktable, Transaction *t[])
{

    Locktable->setLock(0x42d48,READ,t[20]);

```

```

    Locktable->setLock(0x44148,READ,t[20]);
    Locktable->setLock(0x41948,READ,t[20]);
    Locktable->setLock(0xcfb0,READ,t[20]);
    Locktable->setLock(0xcfe0,READ,t[20]);
    Locktable->setLock(0xd000,READ,t[20]);
    Locktable->setLock(0x17bba0,READ,t[20]);
    Locktable->setLock(0x17bc60,READ,t[20]);
    Locktable->setLock(0x17bc20,READ,t[20]);
    Locktable->setLock(0x17c520,READ,t[20]);
    Locktable->setLock(0x17c4a0,READ,t[20]);
    Locktable->setLock(0x17bbe0,READ,t[20]);
    Locktable->setLock(0x17c420,READ,t[20]);
    Locktable->setLock(0x17c760,READ,t[20]);
    Locktable->setLock(0x17c2e0,READ,t[20]);
    Locktable->setLock(0x17c360,READ,t[20]);
    Locktable->setLock(0x17c1e0,READ,t[20]);
    Locktable->setLock(0x17c1a0,READ,t[20]);
    Locktable->setLock(0x17bee0,READ,t[20]);
    Locktable->setLock(0x17c7a0,READ,t[20]);

    Locktable->releaseLocks(t[20]);
}

void t21(LockTable * Locktable, Transaction *t[])
{
    Locktable->setLock(0x42d48,READ,t[21]);
    Locktable->setLock(0x44148,READ,t[21]);
    Locktable->setLock(0x3b548,READ,t[21]);
    Locktable->setLock(0x3c948,READ,t[21]);
    Locktable->setLock(0xcfb0,READ,t[21]);
    Locktable->setLock(0xcfe0,READ,t[21]);
    Locktable->setLock(0xd000,READ,t[21]);
    Locktable->setLock(0x17bba0,READ,t[21]);
    Locktable->setLock(0x17bc60,READ,t[21]);
    Locktable->setLock(0x17bc20,READ,t[21]);
    Locktable->setLock(0x17bbe0,READ,t[21]);
    Locktable->setLock(0x17c3e0,READ,t[21]);
    Locktable->setLock(0x17c160,READ,t[21]);
    Locktable->setLock(0x17c1e0,READ,t[21]);
    Locktable->setLock(0x17bfe0,READ,t[21]);
    Locktable->setLock(0x17c2a0,READ,t[21]);
    Locktable->setLock(0x17c0e0,READ,t[21]);
    Locktable->setLock(0x17c060,READ,t[21]);
    Locktable->setLock(0x17c560,READ,t[21]);
    Locktable->setLock(0x17c0a0,READ,t[21]);

    Locktable->releaseLocks(t[21]);
}

void t22(LockTable * Locktable, Transaction *t[])

```

```

{
    Locktable->setLock(0x42d48,READ,t[22]);
    Locktable->setLock(0x44148,READ,t[22]);
    Locktable->setLock(0x41948,READ,t[22]);
    Locktable->setLock(0x3f148,READ,t[22]);
    Locktable->setLock(0xcfbc0,READ,t[22]);
    Locktable->setLock(0xcffe0,READ,t[22]);
    Locktable->setLock(0xd0000,READ,t[22]);
    Locktable->setLock(0x17bba0,READ,t[22]);
    Locktable->setLock(0x17bc60,READ,t[22]);
    Locktable->setLock(0x17bc20,READ,t[22]);
    Locktable->setLock(0x17bbe0,READ,t[22]);
    Locktable->setLock(0x17bee0,READ,t[22]);
    Locktable->setLock(0x17c360,READ,t[22]);
    Locktable->setLock(0x17c1e0,READ,t[22]);
    Locktable->setLock(0x17bf60,READ,t[22]);
    Locktable->setLock(0x17c520,READ,t[22]);
    Locktable->setLock(0x17c120,READ,t[22]);
    Locktable->setLock(0x17c6a0,READ,t[22]);
    Locktable->setLock(0x17c160,READ,t[22]);

    Locktable->releaseLocks(t[22]);
}

```

```
void t23(LockTable * Locktable, Transaction *t[])
```

```

{
    Locktable->setLock(0x1f060,READ,t[23]);
    Locktable->setLock(0x1f0e0,READ,t[23]);
    Locktable->setLock(0x242f8,READ,t[23]);
    Locktable->setLock(0x243f8,READ,t[23]);
    Locktable->setLock(0x242f8,WRITE,t[23]);
    Locktable->setLock(0x243f8,WRITE,t[23]);
    Locktable->setLock(0x42d48,READ,t[23]);
    Locktable->setLock(0x44148,READ,t[23]);
    Locktable->setLock(0x3b548,READ,t[23]);
    Locktable->setLock(0x3c548,READ,t[23]);
    Locktable->setLock(0x3a148,READ,t[23]);
    Locktable->setLock(0xcfbc0,READ,t[23]);
    Locktable->setLock(0xd09e0,READ,t[23]);
    Locktable->setLock(0xcfbc0,WRITE,t[23]);
    Locktable->setLock(0xcfd0,WRITE,t[23]);
    Locktable->setLock(0xd09e0,WRITE,t[23]);
    Locktable->setLock(0xe3e98,READ,t[23]);
    Locktable->setLock(0xe3e90,READ,t[23]);
    Locktable->setLock(0xe3ea8,READ,t[23]);
    Locktable->setLock(0xe56b0,READ,t[23]);
    Locktable->setLock(0xe3ea8,WRITE,t[23]);
    Locktable->setLock(0xe46b0,WRITE,t[23]);
    Locktable->setLock(0xe56b8,WRITE,t[23]);
}

```

```

Locktable->setLock(0xe3e90,WRITE,t[23]);
Locktable->setLock(0xb5ed4,READ,t[23]);
Locktable->setLock(0xb6054,READ,t[23]);
Locktable->setLock(0x10e220,READ,t[23]);
Locktable->setLock(0x10f420,READ,t[23]);
Locktable->setLock(0x10c020,READ,t[23]);
Locktable->setLock(0x10c020,WRITE,t[23]);
Locktable->setLock(0x17bba0,READ,t[23]);
Locktable->setLock(0x17bc60,READ,t[23]);
Locktable->setLock(0x17cba0,READ,t[23]);
Locktable->setLock(0x17bc60,WRITE,t[23]);
Locktable->setLock(0x17bd60,WRITE,t[23]);
Locktable->setLock(0x17cba0,WRITE,t[23]);
Locktable->setLock(0x17bba0,WRITE,t[23]);
Locktable->setLock(0xb5fd4,READ,t[23]);
Locktable->setLock(0x10f620,READ,t[23]);
Locktable->setLock(0x10f420,WRITE,t[23]);
Locktable->setLock(0x10f620,WRITE,t[23]);
Locktable->setLock(0xb5f54,READ,t[23]);
Locktable->setLock(0x110620,READ,t[23]);
Locktable->setLock(0x110820,READ,t[23]);
Locktable->setLock(0x110620,WRITE,t[23]);
Locktable->setLock(0x110820,WRITE,t[23]);
Locktable->setLock(0x10d020,READ,t[23]);
Locktable->setLock(0x10d220,READ,t[23]);
Locktable->setLock(0x10d020,WRITE,t[23]);
Locktable->setLock(0x10d220,WRITE,t[23]);
Locktable->setLock(0xb60d4,READ,t[23]);
Locktable->setLock(0x10e420,READ,t[23]);
Locktable->setLock(0x10e220,WRITE,t[23]);
Locktable->setLock(0x10e420,WRITE,t[23]);

Locktable->releaseLocks(t[23]);
}

void t24(LockTable * Locktable, Transaction *t[])
{
    Locktable->setLock(0xe3e98,READ,t[24]);
    Locktable->setLock(0xe3e90,READ,t[24]);
    Locktable->setLock(0xe3ea8,READ,t[24]);
    Locktable->setLock(0xe46b0,READ,t[24]);
    Locktable->setLock(0xe4eb0,READ,t[24]);
    Locktable->setLock(0xe3ea8,WRITE,t[24]);
    Locktable->setLock(0xe4eb0,WRITE,t[24]);
    Locktable->setLock(0xcfbc0,READ,t[24]);
    Locktable->setLock(0xd0000,READ,t[24]);
    Locktable->setLock(0xcfbc0,WRITE,t[24]);
    Locktable->setLock(0xd0000,WRITE,t[24]);
    Locktable->setLock(0x17bba0,READ,t[24]);
    Locktable->setLock(0x17bc60,READ,t[24]);
}

```

```

Locktable->setLock(0x17bc20,READ,t[24]);
Locktable->setLock(0x17c5e0,READ,t[24]);
Locktable->setLock(0x17bc20,WRITE,t[24]);
Locktable->setLock(0x17c5e0,WRITE,t[24]);
Locktable->setLock(0x17c960,READ,t[24]);
Locktable->setLock(0x17bc60,WRITE,t[24]);
Locktable->setLock(0x17c960,WRITE,t[24]);
Locktable->setLock(0x17bbe0,READ,t[24]);
Locktable->setLock(0x17c2a0,READ,t[24]);
Locktable->setLock(0x17bbe0,WRITE,t[24]);
Locktable->setLock(0x17c2a0,WRITE,t[24]);
Locktable->setLock(0x17c4e0,READ,t[24]);
Locktable->setLock(0x17c4e0,WRITE,t[24]);
Locktable->setLock(0x17c5a0,READ,t[24]);
Locktable->setLock(0x17c5a0,WRITE,t[24]);
Locktable->setLock(0x17c8e0,READ,t[24]);
Locktable->setLock(0x17c8e0,WRITE,t[24]);
Locktable->setLock(0x17c360,READ,t[24]);
Locktable->setLock(0x17c360,WRITE,t[24]);
Locktable->setLock(0x17c520,READ,t[24]);
Locktable->setLock(0x17c520,WRITE,t[24]);
Locktable->setLock(0x17c820,READ,t[24]);
Locktable->setLock(0x17c820,WRITE,t[24]);
Locktable->setLock(0x42d48,READ,t[24]);
Locktable->setLock(0x44148,READ,t[24]);
Locktable->setLock(0x3b548,READ,t[24]);
Locktable->setLock(0x3c548,READ,t[24]);
Locktable->setLock(0x3a148,READ,t[24]);
Locktable->setLock(0x3a148,WRITE,t[24]);
}

int
main(
int argc,                // Number of arguments
char *argv[ ]           // Arguments
)
{
    int i;
    LockTable *Locktable;
    ofstream lockout;
    Transaction * t[25];

    for (i = 0; i<25; i++)
        t[i] = new Transaction(i);

    if ( argc < 2 )
    {
        cerr << "Usage: " << argv[0] << " <size>" << endl;
        return 0;
    }
}

```

```

}
else
{
    cout << "Starting initialization..." << endl;
    Locktable = new LockTable(1023, argv[2]);
    if ( Locktable == 0)
    {
        cerr << "Unable to create lock ds" << endl;
        return 0;
    }
    else
    {
        lockout.open("lock.dat");

        Locktable->setLock(0x1f060,READ,t[1]);
        Locktable->setLock(0x1f0e0,READ,t[1]);
        Locktable->setLock(0x242f8,READ,t[1]);
        Locktable->setLock(0x243f8,READ,t[1]);
        Locktable->setLock(0x242f8,WRITE,t[1]);
        Locktable->setLock(0x243f8,WRITE,t[1]);
        Locktable->setLock(0x42d48,READ,t[1]);
        Locktable->setLock(0x44148,READ,t[1]);
        Locktable->setLock(0x41948,READ,t[1]);
        Locktable->setLock(0x3dd48,READ,t[1]);
        Locktable->setLock(0xcfbc0,WRITE,t[1]);
        Locktable->setLock(0xcfbc0,READ,t[1]);
        Locktable->setLock(0xcfdc0,WRITE,t[1]);
        Locktable->setLock(0xd09e0,WRITE,t[1]);
        Locktable->setLock(0xe3e90,WRITE,t[1]);
        Locktable->setLock(0xe3e98,WRITE,t[1]);
        Locktable->setLock(0xe3e98,READ,t[1]);
        Locktable->setLock(0xe4698,WRITE,t[1]);
        Locktable->setLock(0xe56a0,WRITE,t[1]);
        Locktable->setLock(0xb5ed4,READ,t[1]);
        Locktable->setLock(0xb5fd4,READ,t[1]);
        Locktable->setLock(0x10e220,READ,t[1]);
        Locktable->setLock(0x10f420,READ,t[1]);
        Locktable->setLock(0x10d020,READ,t[1]);
        Locktable->setLock(0x10d220,READ,t[1]);
        Locktable->setLock(0x10d020,WRITE,t[1]);
        Locktable->setLock(0x10d220,WRITE,t[1]);
        Locktable->setLock(0x17bba0,WRITE,t[1]);
        Locktable->setLock(0x17bba0,READ,t[1]);
        Locktable->setLock(0x17bca0,WRITE,t[1]);
        Locktable->setLock(0x17cae0,WRITE,t[1]);
        Locktable->setLock(0xb5f54,READ,t[1]);
        Locktable->setLock(0x10f620,READ,t[1]);
        Locktable->setLock(0x10f420,WRITE,t[1]);
        Locktable->setLock(0x10f620,WRITE,t[1]);
        Locktable->setLock(0x17cae0,READ,t[1]);
        Locktable->setLock(0xb6054,READ,t[1]);
    }
}

```



```

Locktable->setLock(0x10c020,READ,t[1]);
Locktable->setLock(0x10c020,WRITE,t[1]);
Locktable->setLock(0x110620,READ,t[1]);
Locktable->setLock(0x110620,WRITE,t[1]);
Locktable->setLock(0x110820,READ,t[1]);
Locktable->setLock(0x110820,WRITE,t[1]);
Locktable->releaseLocks(t[1]);

```

```

t2(Locktable,t);
t3(Locktable,t);
t4(Locktable,t);
t5(Locktable,t);
t6(Locktable,t);
t7(Locktable,t);
t8(Locktable,t);
t9(Locktable,t);
t10(Locktable,t);
t11(Locktable,t);
t12(Locktable,t);
t13(Locktable,t);
t14(Locktable,t);
t15(Locktable,t);
t16(Locktable,t);
t17(Locktable,t);
t18(Locktable,t);
t19(Locktable,t);
t20(Locktable,t);
t21(Locktable,t);
t22(Locktable,t);
t23(Locktable,t);
t24(Locktable,t);
Locktable->releaseLocks(t[24]);
lockout << "\n\n\t\t\tPrinting Released Locks" << endl;

return 30;

```

```

    }
  }
return 20;
}

```

# APPENDIX C

## Optimized Executable of the Conventional Lock Manager

QPT2: the Quick Profiler and Tracer.

Version 1.00 of May 9, 1995.

Copyright (c) 1993-1995 by James R. Larus. All Rights Reserved.

Quick profiling with estimated weights:

Flat Procedural Statistics: For counts > 1.000000

836460 Total Instructions

Dynamic Inst	% Time	Cum % Time	Calls	Inst/ Call	Routine
211516	25.3	25.3	1956	108.	_malloc
103735	12.4	37.7	2410	43.	.rem
98546	11.8	49.5	518	190.	_free
65456	7.8	57.3	518	126.	malloc.o
35921	4.3	61.6	1115	32.	_find__4Listi
33764	4.0	65.6	1468	23.	malloc.o@0x15f24
30806	3.7	69.3	24	1284.	_releaseLocks__9LockTableP11Transaction
29854	3.6	72.9	506	59.	_remove__4ListP6Object
28682	3.4	76.3	699	41.	<i>_setLock__9LockTableUiP11Transaction</i>
21842	2.6	78.9	1110	20.	_add__4ListP6Object
21635	2.6	81.5	2411	9.	_next__4List
19540	2.3	83.8	1954	10.	___builtin_new
15344	1.8	85.7	506	30.	_add__9LockTableP4ListP3LCBiP11Transactioni
13255	1.6	87.3	1205	11.	_access__9LockTableUiRi
13132	1.6	88.8	506	26.	_remove__4ListP4Node
9640	1.2	90.0	1205	8.	_hashaddr__9HashTablei
9504	1.1	91.1	491	19.	malloc.o@0x15e18
8880	1.1	92.2	1110	8.	___4NodeP6Object
8096	1.0	93.1	506	16.	___\$__4Node
7352	0.9	94.0	1838	4.	_reset__4List
7014	0.8	94.9	519	14.	_realloc@0x167ec
6794	0.8	95.7	506	13.	_add__11TransactionP6Object
6154	0.7	96.4	1	6154.	___9HashTablei
6072	0.7	97.1	506	12.	_del__4ListP4Node
5983	0.7	97.9	193	31.	_search__4Listi
4048	0.5	98.3	506	8.	___builtin_delete
2530	0.3	98.6	506	5.	___3LRB
2460	0.3	98.9	492	5.	_realloc@0x168f0
591	0.1	99.0	1	591.	_main
549	0.1	99.1	11	50.	_realloc@0x166e4
490	0.1	99.1	98	5.	___3LCB
450	0.1	99.2	90	5.	_add__9HashTableiP4List

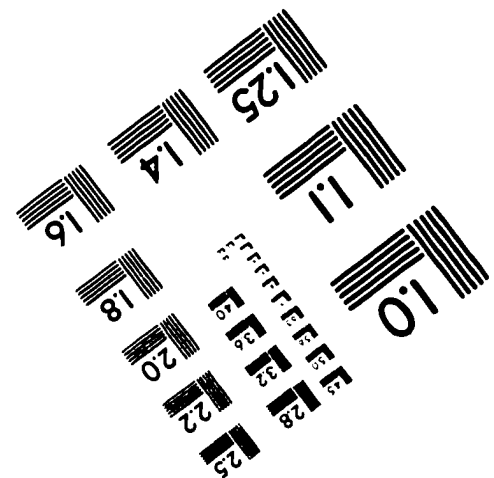
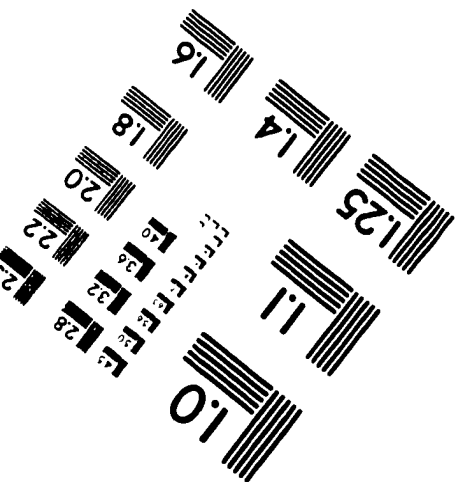
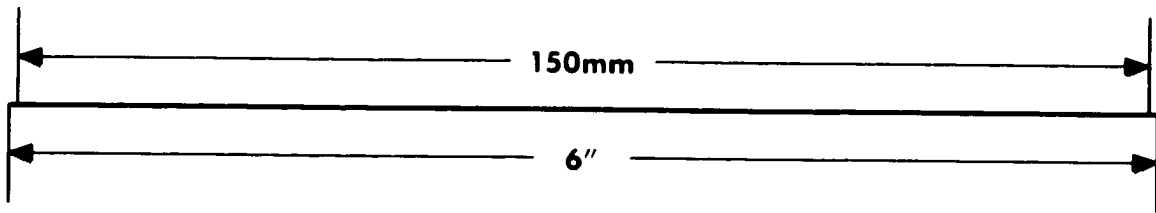
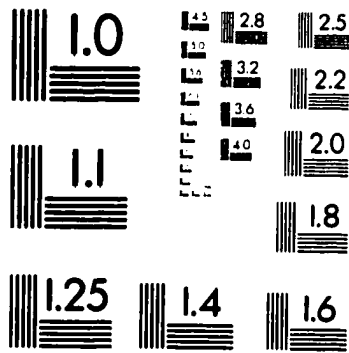
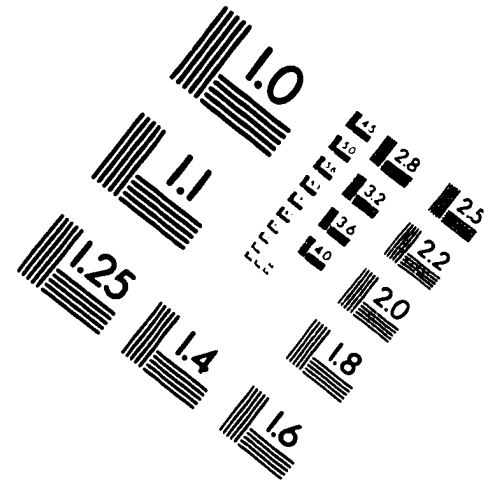
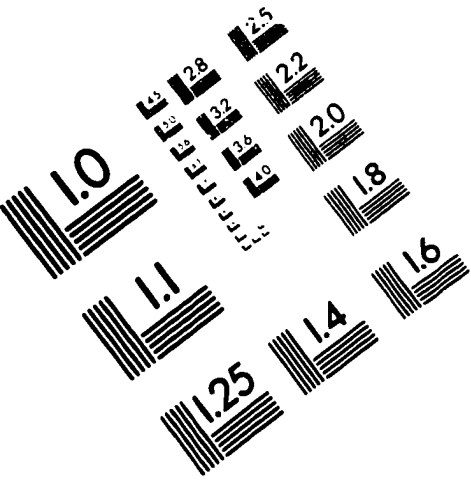
441	0.1	99.2	11	40.	.div
374	0.0	99.3	1	374.	__fwalk
339	0.0	99.3	1	339.	_t19__FP9LockTablePP11Transaction
317	0.0	99.4	1	317.	_t18__FP9LockTablePP11Transaction
312	0.0	99.4	1	312.	_t23__FP9LockTablePP11Transaction
305	0.0	99.4	1	305.	_t15__FP9LockTablePP11Transaction
289	0.0	99.5	1	289.	_t6__FP9LockTablePP11Transaction
240	0.0	99.5	1	240.	_t24__FP9LockTablePP11Transaction
198	0.0	99.5	11	18.	.umul
192	0.0	99.5	12	16.	__sbrk
190	0.0	99.6	1	190.	_t12__FP9LockTablePP11Transaction
190	0.0	99.6	1	190.	_t17__FP9LockTablePP11Transaction
185	0.0	99.6	1	185.	_t10__FP9LockTablePP11Transaction
179	0.0	99.6	2	90.	__fwrite
174	0.0	99.7	2	87.	__memchr
169	0.0	99.7	1	169.	_t9__FP9LockTablePP11Transaction
157	0.0	99.7	1	157.	_t4__FP9LockTablePP11Transaction
125	0.0	99.7	1	125.	_t20__FP9LockTablePP11Transaction
125	0.0	99.7	1	125.	_t21__FP9LockTablePP11Transaction
119	0.0	99.7	1	119.	_t22__FP9LockTablePP11Transaction
114	0.0	99.8	1	114.	_t7__FP9LockTablePP11Transaction
104	0.0	99.8	1	104.	_t2__FP9LockTablePP11Transaction
100	0.0	99.8	25	4.	__11Transactioni
96	0.0	99.8	1	96.	_t11__FP9LockTablePP11Transaction
96	0.0	99.8	1	96.	_t14__FP9LockTablePP11Transaction
96	0.0	99.8	1	96.	_t8__FP9LockTablePP11Transaction
94	0.0	99.8	3	31.	__fclose
90	0.0	99.8	1	90.	_t13__FP9LockTablePP11Transaction
90	0.0	99.8	1	90.	_t3__FP9LockTablePP11Transaction
84	0.0	99.9	1	84.	_t16__FP9LockTablePP11Transaction
75	0.0	99.9	1	75.	__strlen
71	0.0	99.9	2	36.	__memcpy
66	0.0	99.9	1	66.	__ls__7ostreamPCc
60	0.0	99.9	1	60.	__IO_file_overflow
60	0.0	99.9	1	60.	_t5__FP9LockTablePP11Transaction
56	0.0	99.9	1	56.	__IO_unbuffer_all
54	0.0	99.9	2	27.	__IO_default_setbuf
53	0.0	99.9	3	18.	__fflush
50	0.0	99.9	2	25.	__IO_do_write
50	0.0	99.9	1	50.	__IO_flush_all
43	0.0	99.9	2	22.	start
40	0.0	99.9	2	20.	.mul
40	0.0	99.9	2	20.	__IO_file_setbuf
39	0.0	99.9	3	13.	__IO_setb
32	0.0	99.9	1	32.	__xflsbuf
30	0.0	99.9	2	15.	__IO_file_sync
29	0.0	100.0	1	29.	__findbuf
23	0.0	100.0	1	23.	__do_global_ctors
23	0.0	100.0	1	23.	__wrtchk
22	0.0	100.0	11	2.	.udiv
22	0.0	100.0	1	22.	__exit

22	0.0	100.0	1	22.	__sync__8stdiobuf
21	0.0	100.0	3	7.	_close
17	0.0	100.0	1	17.	__endl__FR7ostream
16	0.0	100.0	1	16.	__do_global_dtors
16	0.0	100.0	1	16.	__xspn__8stdiobufPCci
15	0.0	100.0	1	15.	__IO_doallocbuf
15	0.0	100.0	1	15.	__on_exit
15	0.0	100.0	1	15.	__sys_write__8stdiobufPCci
13	0.0	100.0	1	13.	__flush__7ostream
13	0.0	100.0	1	13.	__isatty
13	0.0	100.0	1	13.	__overflow__8stdiobufi
12	0.0	100.0	1	12.	__9LockTableiPc
11	0.0	100.0	1	11.	__IO_sb_write__FP8_IO_FILEPCvi
11	0.0	100.0	1	11.	__IO_sb_xspn__FP8_IO_FILEPCvUI
10	0.0	100.0	1	10.	__IO_sb_overflow__FP8_IO_FILEi
10	0.0	100.0	1	10.	__main
9	0.0	100.0	1	9.	__overflow
7	0.0	100.0	1	7.	__IO_cleanup
7	0.0	100.0	1	7.	__getpagesize
7	0.0	100.0	1	7.	__ioctl
7	0.0	100.0	1	7.	__overflow__7filebufi
7	0.0	100.0	1	7.	__write
6	0.0	100.0	1	6.	__builtin_vec_new
6	0.0	100.0	1	6.	__cleanup
6	0.0	100.0	1	6.	__flush__FR7ostream
5	0.0	100.0	1	5.	__GLOBAL_\$D\$__11_ios_fields
2	0.0	100.0	1	2.	__GLOBAL_\$I\$__11_ios_fields
2	0.0	100.0	1	2.	__exit
2	0.0	100.0	1	2.	start_float

Total edge counter increments = 85813.

Block counter = 323058. (3.76)

# IMAGE EVALUATION TEST TARGET (QA-3)



**APPLIED IMAGE, Inc**  
 1653 East Main Street  
 Rochester, NY 14609 USA  
 Phone: 716/482-0300  
 Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved