NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

EUREKA: A DISTRIBUTED SHARED MEMORY SYSTEM BASED ON THE LAZY DATA MERGING CONSISTENCY MODEL

by

Joao Alberto Vianna Tavares

September 1995

Thesis Advisor: Amr Zaky

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1995	Master's Thesis

4. TITLE AND SUBTITLE
Eureka: a Distributed Shared Memory System Based on the Lazy Data Merging Consistency Model (U)

5. FUNDING NUMBERS

6. AUTHOR(S)
Tavares, Joao Alberto Vianna

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Distributed Shared Memory (DSM) provides an abstraction of shared memory on a network of workstations. Problems with existing DSM systems are lack of portability due to compiler and/or operating system modification requirements, and reduced performance due to significant synchronization and communication costs when compared to their message passing counterparts (e.g., PVM and MPI).

Our approach was to introduce a new DSM consistency model, *Lazy Data Merging (LDM)*, which extends *Data Merging (DM)*. LDM is optimized for software runtime implementations and differs from DM by *"lazily"* placing data updates across the communication network only when they are required. It is our belief that LDM can significantly reduce communication costs, particularly for applications that make extensive use of locks.

We have completed the design of *"Eureka"*, a prototype DSM system that provides a software implementation of the LDM consistency model. To ensure portability and efficiency we use only standard Unix™ system calls and a publicly available software thread package, Cthreads, from the University of Utah. Futhermore, we have implemented and tested some of Eureka's core components, specifically, the set of communication and hybrid (Invalidate/Update) coherence primitives, which are essential for follow on work in building the complete DSM system. The question of efficiency is still an open problem, because we did not compare Eureka with other DSM implementations.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Distributed shared memory, lazy data merging, survey on consistency models and network programming.	137
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

EUREKA: A DISTRIBUTED SHARED MEMORY SYSTEM BASED ON THE LAZY DATA MERGING CONSISTENCY MODEL

Joao Alberto Vianna Tavares
Lieutenant, Brazilian Navy
B.S., Brazilian Naval Academy, 1983

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1995

Author: _____
Joao Alberto Vianna Tavares

Approved by: _____
Amr Zaky, Thesis Advisor

_____
Man-Tak Shing, Second Reader

_____
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

Distributed Shared Memory (DSM) provides an abstraction of shared memory on a network of workstations. Problems with existing DSM systems are lack of portability due to compiler and/or operating system modification requirements, and reduced performance due to significant synchronization and communication costs when compared to their message passing counterparts (e.g., PVM and MPI).

Our approach was to introduce a new DSM consistency model, *Lazy Data Merging (LDM)*, which extends *Data Merging (DM)*. LDM is optimized for software runtime implementations and differs from DM by *"lazily"* placing data updates across the communication network only when they are required. It is our belief that LDM can significantly reduce communication costs, particularly for applications that make extensive use of locks.

We have completed the design of *"Eureka"*, a prototype DSM system that provides a software implementation of the LDM consistency model. To ensure portability and efficiency we use only standard Unix™ system calls and a publicly available software thread package, Cthreads, from the University of Utah. Futhermore, we have implemented and tested some of Eureka's core components, specifically, the set of communication and hybrid (Invalidate/Update) coherence primitives, which are essential for follow on work in building the complete DSM system. The question of efficiency is still an open problem, because we did not compare Eureka with other DSM implementations.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. BACKGROUND AND MOTIVATION 1

B. MESSAGE PASSING VERSUS SHARED MEMORY 2

C. PROBLEM STATEMENT 4

D. CONTRIBUTION 5

E. THESIS OVERVIEW 5

II. BACKGROUND 7

A. TAXONOMIES FOR CLASSIFYING DISTRIBUTED SHARED MEMORY SYSTEMS 7

1. DSM Implementation Level 8

a. Hardware Level DSM 9

b. Software Level DSM 9

c. Hybrid implementations 10

d. Factors that affect the DSM implementation level 10

2. DSM Protocols 11

A. ISSUES ON DISTRIBUTED SHARED MEMORY SYSTEMS 13

1. Granularity of Sharing 13

2. False Sharing 15

3. Synchronization 17

4. Heterogeneity 17

5. Coherence Protocol 18

B. DSM MEMORY CONSISTENCY MODELS 19

1. Strict Consistency Model 20

2. Sequential Consistency Model 20

3. Processor Consistency 22

4. Causal Consistency Model 23

5. Weak Consistency Model 24

6. Release Consistency Model 27

a. Eager Release Consistency 28

b. Lazy Release Consistency 29

7. Entry Consistency Model 31

III. DSM SYSTEMS OVERVIEW 35

A. HARDWARE IMPLEMENTATIONS 35

1. KSR-1 35

2. DASH 37

B. SOFTWARE IMPLEMENTATIONS 39

1. Operating System Level - Clouds 39

2. Runtime Libraries 40

a. Midway 40

b. Munin 43

c. Treadmarks 46

C. COMPILER INSERTED PRIMITIVES - ORCA 47

D. HARDWARE/SOFTWARE COMBINATION - FLASH 48

IV. LAZY DATA MERGING CONSISTENCY MODEL 51

A. THE DATA MERGING DSM PROTOCOL - AN OVERVIEW 51

B. DATA MERGING PROTOCOL: DEFINITIONS 52

1. Processing Element (PE) 52

2. Global Memory Unit (GMU) 52

C. THE DATA MERGING PROTOCOL 53

1. Processing Element 54

2. Global Memory Unit Actions 55

3. Actions Performed by the GMU in Response to the Processing Element Requests 56

a. Request Cache Block 56

b. Flush Data 56

c. Report Replacement 56

d. Bypass-Read Data Element 56

e. Bypass-Write Data Element 57

f. Lock 57

g. Unlock 57

h. Test and Set Lock 57

i. GMU Internal Actions 57

D. LDM RATIONALE 58

E. EXTENSIONS TO THE DATA MERGING PROTOCOL 60

F. THE LAZY DATA MERGING PROTOCOL 60

1. Protocol Notation 61

2. Description of LDM actions 62

a. Page Fault 62

b. Bypass-Cache Messages 63

c. Synchronization Operations 65

d. Partial Ordering Definitions 66

e. Read and Write Locks 68

f. Barrier Call 73

G. PUTTING IT ALL TOGETHER 75

1. Distributed Data Base Problem 75

a. Data Merging 76

b. Lazy Data Merging 77

2. Lazy Data Merging: Read and Write Locks 78

3. Data Merging and Lazy Data Merging Barrier Call 80

V. EUREKA: A "LAZY DATA MERGING" IMPLEMENTATION 85

A. DSM SYSTEM ORGANIZATION 85

1. Objects 85

2. Local Threads 86

B. EUREKA RUNTIME ENVIRONMENT 87

1. Eureka Execution Overview 88

2. Handlers Initialization 91

a. Communication Port and Communication Handler 91

b. Memory Handler 92
3. Eureka Shared Data and Synchronization Objects Allocation 92
a. Static Memory Allocation 93
b. Dynamic Memory Allocation 95
c. Creation of Lock Objects 95
d. Creation of Barrier Synchronization Objects 96
4. Execution Phase 97
a. Suspend Queue 97
b. Page Directory 98
c. Synchronization Directory and Page Table 98
d. Sending / Receiving Messages 99
e. Operation Codes 101
5. Data Gathering Phase 104
6. Termination Phase 104
C. CODE EXAMPLES 104
1. Creation of an UDP Port 104
2. DSM System Calls 106
3. How to Present Debug Information 108
VI. CONCLUSION 109
A. SUGGESTIONS FOR FUTURE WORK 110
LIST OF REFERENCES 111
INITIAL DISTRIBUTION LIST 115

LIST OF FIGURES

1. DSM abstraction. 3
2. Taxonomy for classifying DSM systems as proposed by Tanenbaum in [TN95]. 7
3. DSM implementation levels. 8
4. The second criterion: the DSM algorithm. 12
5. False Sharing. 16
6. An example of a sequentially consistent program. 21
7. Communications on the Sequential Consistency Model. 21
8. Example of a processor consistent program. 23
9. Causal Consistency Model. 24
10. An example of DRF0 [AH90]. 25
11. Weak Consistency valid (a) and invalid (b) sequences of events. 26
12. Release Consistency model. 28
13. Eager Release Consistency model. 29
14. LRC Invalidate protocol (a) and LRC update protocol (b). 31
15. Comparison between RC and LRC models. 32
16. The KSR-1 ALLCACHE Hierarchy. 36
17. DASH High Level Structure. 38
18. DASH Processing Node. 38
19. Distributed queue locking scheme. 42
20. Munin Runtime System [CBZ92]. 44
21. Write-Shared Protocol: twin creation when a page is accessed for writes (a) and sending out diffs (b) when a release operation occurs. 46

22. Data Merging Components. 53
23. Processing Element Finite State Machine. 54
24. Global Memory Unit Finite State Machine. 55
25. Lazy Data Merging Runtime Environment. 59
26. Notation for description of LDM. 61
27. Performing a Data Block request. 63
28. Bypass-Read Messages. 64
29. Bypass-write message. 65
30. Diffs creation process. 66
31. Vector Clock implementation. 68
32. Synchronization event: acquiring a Write-Lock. 70
33. Synchronization event: acquiring a Read-Lock. 71
34. Synchronization event: performing a lock release. 72
35. Synchronization event: barrier call. 74
36. Data distribution across the blocks. 76
37. Data Merging. 77
38. Lazy Data Merging. 79
39. Lazy Data Merging: read and write locks. 81
40. Comparison of DM and LDM during a barrier call. 82
41. Lazy Data Merging: barrier call. 83
42. Eureka Runtime Environment. 87
43. Eureka System Initialization. 89
44. Message header format. 100

45. Diff message. 101

LIST OF TABLES

1. Block Granularity on DSM Systems 14

2. Distinction Between RT and VM implementations 43

3. Page Protection Actions 92

ACKNOWLEDGEMENTS

First and foremost, I must acknowledge the unfailing and unconditional support I have received through it all from my wife, Tereza, without which this work could never have been completed. Her positive attitude and understanding while dealing with the everyday burdens of child rearing were remarkable as they were essential to my success. I am also indebted to my parents, Dr. Bernardino and Maria Izabel Tavares and to my grandmother Helena Vianna for instilling in me the value of education and the desire to pursue my goals.

I also wish to express my deepest gratitude to Professor Amr Zaky whose support and guidance have been a constant inspiration to me. A special thanks goes to Professor Mantak Shing for his insights on the reviewing of my thesis and to Doctor Ted Lewis for his support and interest.

Although not members of this institution, Doctor John Carter from the University of Utah and Doctor Alan Karp from HP Labs were most helpful and served as a source of inspiration for my research.

It would be mostly unfair if we didn't mention the staff members of this Department, their silent work was invaluable for this achievement. My special thanks to Al Wong whose help at the initial stages of this thesis were of the most importance.

I. INTRODUCTION

A. BACKGROUND AND MOTIVATION

Parallel processing is the upcoming alternative for expanding processing power. The trend towards this technology is derived from the availability of high performance communication networks and the perception that improvements in computer performance from hardware innovations are limited as we approach various physical limits.

There are two widely accepted models for parallel programming: shared memory and message passing. The shared memory model is a direct extension of the conventional uniprocessor programming model. In this model, processors interact by modifying data objects stored in the shared address space. Multiple Instruction stream Multiple Data stream (MIMD) shared-address-space computers, often referred to as multiprocessors, are examples of parallel architectures that employ such memory model. A major drawback of such architectures is that the bandwidth of the interconnection network must be substantial to ensure a scalable performance. To reduce the problem, a fast local memory (cache) is incorporated. This local memory concept can further be extended to eliminate physically shared memory entirely as in Cache Only Memory Architectures (COMA) [ML95].

According to the memory access time to global and local data, multiprocessor architectures can be classified as Uniform Memory Access (UMA), when the time taken by a processor to access any memory word in the system is identical, or Nonuniform Memory Access (NUMA), whenever the time to access a remote memory bank is longer than the time to access a local one.

Distinct from the previous programming model, the "*message passing model*" does not support the single shared memory abstraction. Instead, each processor has its own private memory which is invisible to other processors. These types of architectures can be referred as No Remote Memory Access (NORMA) architectures [TN95]. In this model, processors can communicate only through explicit "*message passing*", dispensing the need for enforcing cache coherence as is the case for shared memory systems.

The message passing paradigm is widely used on MIMD distributed memory multiprocessors. This type of architecture allows combining inexpensive, off-the-shelf processors connected through an interconnection network or through a hierarchical bus or ring architecture, resulting in a very scalable design, delivering high performance computing. The next section addresses the issues involved on both programming models.

B. MESSAGE PASSING VERSUS SHARED MEMORY

There are advantages and disadvantages to message-passing as well as to shared memory. *"Message-passing"* gives programmers and compilers explicit control over the choice of data communicated and over the time of transmission as opposed to the shared memory paradigm. With appropriate interfaces and protocols, it is relatively easy to overlap computation with communication. The explicit nature of message-passing is perceived also as its main weakness [CBZ91]; programmers and compilers need to plan and to program explicitly every communication action. Such planning is especially difficult for applications that use data dependent communication actions. When exact access patterns are not known, performance is affected by the volume of data communicated, the number of messages sent, and the amount of time that processes must wait for messages to be delivered.

In contrast, *Shared Memory* can be viewed as a simpler and more intuitive abstraction [ACDB94]. A Distributed Shared Memory (DSM) system can be defined as an abstraction of a single shared address space, implemented on a distributed memory multiprocessor. DSM allows processes to assume a globally shared virtual memory even though they execute in nodes that do not physically share memory. The DSM software provides the abstraction of a globally shared memory, in which each processor can access any data item, without the programmer having to worry about where the data is, or how to move this value to the appropriate processor. On a DSM system the programmer can concentrate on algorithmic development rather than on managing partitioned data sets and communicating values. In addition to ease of programming, DSM provides the same programming

environment as that on hardware implementations of shared-memory multiprocessors, simplifying the portability between the two environments. A common application for the DSM concept allows for seamless integration of shared memory workstations in a network environment. Figure 1 illustrates a DSM system consisting of N networked workstations, each with its own memory, connected by a network.

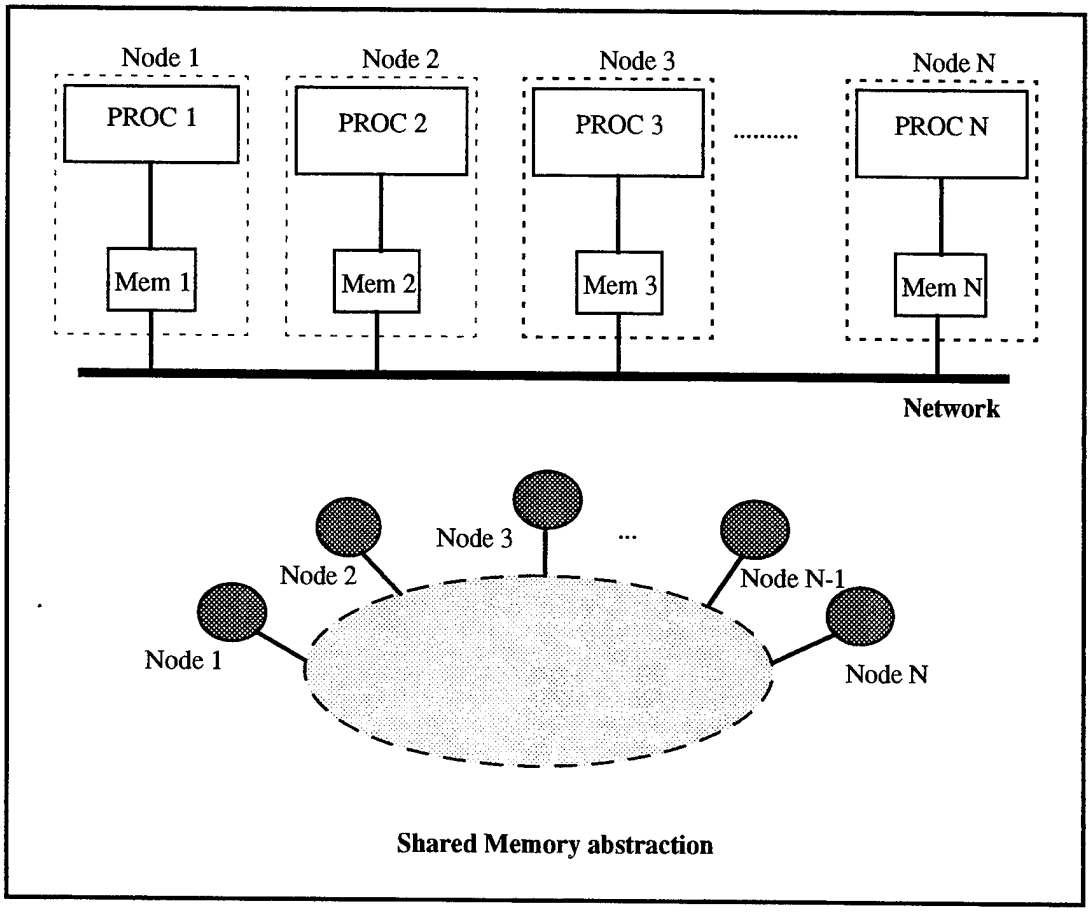


Figure 1: DSM abstraction

In summary, we can state that the advantages of the DSM model over message passing are:

- Shared memory programs are usually shorter and easier to understand than their message passing counterparts.

- Large or complex data structures may easily be communicated.
- Shared memory gives transparent process-to-process communication.
- Programming with shared memory is a well-understood problem.

C. PROBLEM STATEMENT

DSM systems combine features of shared memory and distributed memory multiprocessors. They support the relatively simple and portable programming model of shared memory on physically distributed memory hardware, which is more scalable and less expensive to build than the shared memory hardware. Although a large number of DSM systems (Munin [CBZ92], Midway [BZS93] and [ZSB94], etc.) have been proposed and implemented they are still not widely in use. Some evident reasons are lack of portability, low performance when compared with their message passing counterparts, and the need for extensive modifications of existing programs.

The challenge in building a DSM system is to achieve good performance over a wide range of parallel programs without requiring extensive program restructuring by the programmer [CBZ92]. The overhead of maintaining consistency in software and the high latency of sending messages make this rather difficult. The primary source of DSM overhead is the large amount of communication that is required to maintain consistency and the operating system cost to prepare a message associated with the network latency. The conjunction of these two factors penalizes the overall system performance.

Data Merging [KS93] provides means for implementing a DSM system with communication costs comparable to the previous proposals, but requiring only small hardware changes. The goal of this research is to extend such protocol to a portable software implementation and verify its feasibility. Since for software implementations, the communication costs are much higher than on a shared memory multiprocessor, we impose some restrictions on the Data Merging protocol as originally proposed.

D. CONTRIBUTION

The major contribution of this thesis is to extend the Data Merging as a viable consistency protocol for implementing DSM. In order to achieve performance results comparable to previous DSM implementations and to provide a easily portable solution without requiring changes to the application programs, we introduced some modifications to the original protocol as proposed by Karp and Sarkar in [KS93].

E. THESIS OVERVIEW

The remainder of this thesis is organized as follows. Chapter II gives a comprehensive overview of DSM, starting by describing a taxonomy for classification of DSM systems, followed by a broad discussion on the issues involving such systems and the existing memory consistency model proposals. In Chapter III we briefly describe representative DSM implementations of each category. In Chapter IV we analyze the Data Merging and Lazy Data Merging protocols pointing out their mean features. Chapter V outlines the design decisions and the modifications introduced for a portable software implementation and we relate the implementation details applied to the Eureka DSM system. In Chapter VI we conclude our work and provide directions for further research.

II. BACKGROUND

This chapter gives a comprehensive overview of Distributed Shared Memory (DSM) systems principles. We start by describing two taxonomies for classifying DSM systems. Further on, we enumerate the major issues that are involved in DSM system implementations and give a description of existing memory consistency models.

A. TAXONOMIES FOR CLASSIFYING DISTRIBUTED SHARED MEMORY SYSTEMS

Shared memory systems cover a broad spectrum, from systems that maintain consistency entirely in hardware, to those that do it entirely in software. To present such wide spectrum we need some sort of criteria for their classification. We introduce two taxonomies for classifying such systems. The first was informally introduced by Tanembaum [TN95]. Figure 2 describes this taxonomy as originally proposed.

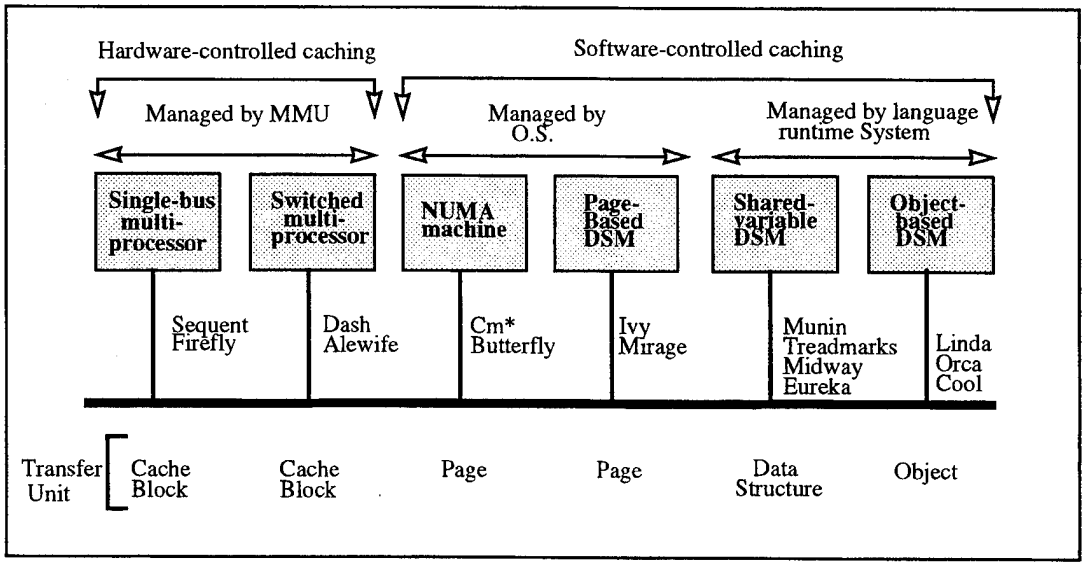


Figure 2: Taxonomy for classifying DSM systems as proposed by Tanenbaum in [TN95].

The other taxonomy was proposed by Milutinovic [ML95] and has the merit of being one of the pioneer attempts for establishing a formal criteria for classifying DSM systems. The proposed taxonomy adopts two variables for categorizing DSM systems: DSM implementation level (Hardware, Software or Hybrid) and the DSM algorithm (SRSW, MRSW, MRMW). The following paragraphs will briefly introduce this taxonomy and some of the issues involved in its application.

1. DSM Implementation Level

There are three basic types for classifying systems under this criterion: Hardware, Software and Hybrid DSM systems. Figure 3 summarizes the three implementation levels in which current DSM systems fall.

The implementation level affects both the programming model as well as the overall system performance. While hardware solutions bring transparency and small access latencies, software solutions can better exploit the application behavior and present more flexibility, especially as a source for experiments of new concepts and algorithms.

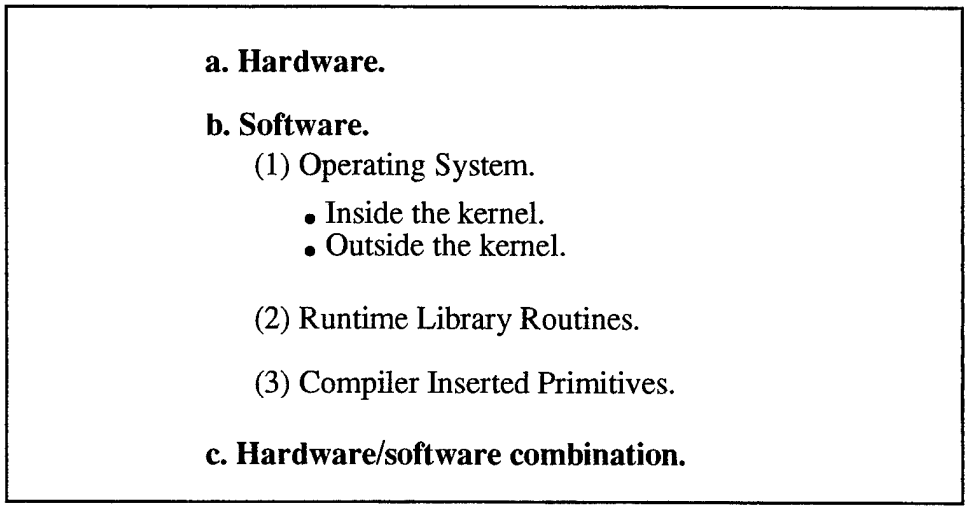


Figure 3: DSM implementation levels.[1]

1. As described in [ML95].

a. Hardware Level DSM

Most of the hardware DSM systems concentrate in one of three categories:

- *Cache Coherent Non-Uniform Memory Architecture (CC-NUMA).*
- *Cache-Only Memory Architecture (COMA).*
- *Reflective Memory System Architecture (RMS).*

On CC-NUMA architectures the shared virtual address space is statically distributed across the clusters. It is accessible by the local processors and by processors from other clusters with distinct access latencies. In general, the DSM mechanism relies on directories with organizations varying from a full-map storage (DASH) to dynamic structures (i.e., linked lists, fat trees, etc.).

COMA architectures provide the dynamic partitioning of data in the form of distributed memories organized as large second level caches. Distinct from the previous architectures, there is no physical home location for any particular data item. These architectures allow a particular data item to be simultaneously replicated on many caches. The typical architecture consists of hierarchical network topology (i.e., KSR-1).

RMS architectures adopt a hardware-implemented update mechanism. This is achieved by declaring some parts of the local memory on each cluster as shared and mapped into a common virtual address space. Coherence maintenance is enforced by full replication. This is achieved by broadcasting/multicasting every write operation to the other units. The result is a high cost for write operations. Typical examples of these architectures are Scrannet and Encore's RMS.

b. Software Level DSM

As described in **F**igure 3 software level implementations can be divided into three basic categories: compiler implementations, user-level runtime packages and operating system level implementations. The latter can further be subdivided into inside/ outside the kernel.

Operating system (inside the kernel) implementations are incorporated to the actual operating system kernel. The advantage of this approach is that the semantics of the underlying operating system can be preserved. An example of such system is Mirage [ML95].

Operating system (outside the kernel) implementations are those on which the same mechanism can be accessed by both the user and the kernel. An example of this systems can be found in Clouds.

User level runtime packages consists of libraries that are linked to the actual application programs. Examples of such systems are Munin, Treadmarks, Quarks, Midway, etc.

For *compiler implementations* the shared memory paradigm is applied at the language level. For these applications, shared data is structured into logical units of sharing and accesses to these shared elements are automatically converted into synchronization and coherence primitives.

c. Hybrid implementations

Hardware DSM implementations have the disadvantage of limited flexibility, especially for enforcing multiple coherence protocols. They also present some limitations on scalability due to the requirements for maintaining directories, for example. On the other hand, software implementations have lack on performance due to the larger granularity and the latency for exchanging messages.

Hybrid implementations try to combine both levels to resolve some of these problems. Typical examples for these architectures are the MIT Alewife in which the full-map directory is implemented part in hardware and part in software and the Stanford FLASH, which provides the means for implementing multiple coherence protocols.

d. Factors that affect the DSM implementation level

For better understanding we divide this item in two topics:

- System architectural configuration; and

• Shared data organization.

System architectural configuration affects the system performance, since it can offer or restrict a good potential for parallel processing of requests related to the DSM management. It also strongly affects the scalability. Since a system applying a DSM mechanism is usually organized as a set of clusters connected by an interconnection network, architectural parameters include:

• *Cluster configuration* (single/multiple processors, with/without, shared/ private, single/multiple level caches, etc.);

• *Interconnection network* (bus hierarchy, ring, mesh, hypercube, specific LAN, etc.). Almost all types of interconnection networks found in multiprocessors and distributed systems have also been used in DSM implementations. The software-oriented systems are, in general, built on top of Ethernet (Munin, Treadmarks, Eureka) or ATM (Midway), while topologies such as bus hierarchies (DASH, FLASH, Alewife), meshes or rings (Memnet) are typical for hardware or hybrid solutions.

Shared Data organization represents the global layout of shared address space, as well as the size and organization of data items in it, and can be distinguished as:

• *Structure of shared data* (e.g., non structured or structured into objects, languages types, etc.);

• *Granularity of coherence unit* (e.g., word, cache, block, page, complex data structure, etc.).

Hardware solutions generally deal with non-structured data objects (typically cache blocks), while many software implementations tend to use data items that represent logical entities in order to take advantage of the locality naturally expressed by the application. On the other hand, some software solutions, based on virtual memory mechanisms, organize data in larger physical blocks (pages), counting on the coarse-grain sharing.

2. DSM Protocols

This classification deals with the possible existence of multiple copies of a data item. It also considers access rights to these copies. The complexity of maintaining coherence among different copies of a data item varies strongly with the algorithm. Many policies have been proposed, the majority of them adopting "*multiple readers single writer*" (MRSW) algorithms. Figure 4 depicts this classification criteria.

a. SRSW (Single Reader/ Single Writer)

- Without Migration.
- With Migration.

b. MRSW (Multiple Readers / Single Writer).

c. MRMW (Multiple Readers / Multiple Writers).

Figure 4: The second criterion: the DSM algorithm.[2]

Basically, there are three parameters closely related to the algorithm:

- Responsibility for the DSM management (e.g., centralized, distributed/fixed, distributed/dynamic).
- Consistency Model (e.g., strict, sequential, causal, weak, release, etc.).
- Coherence policy (e.g., write-invalidate, write-update, type-specific, etc.).

The responsibility for the DSM management can be either centralized or distributed. *Centralized management* is easier to implement, but suffers from the lack of fault tolerance and can become a performance bottleneck. On the other hand, *distributed management* policy can be defined either statically (fixed) or dynamically, eliminating bottlenecks and providing scalability. In the case of a *static management* each manager is assigned a predetermined subset of the data space, which remains fixed throughout the

2. The classification of Figure 4 is equivalent to the PRAM classification as defined in [KGGK94]. The equivalences are SRSW and EREW, MRSW and CREW and MRMW and CRCW.

lifetime of an application. In contrast, the *dynamic approach*[3] management responsibility shifts from a node to another at runtime.

The *consistency model* defines and enforces acceptable ordering of accesses to shared data by different processes such that at pre-specified points of time the state of shared data (as viewed by each individual process) is "*correct*" in some process-defined sense. Also, in [AH90] a consistency model is described as a contract between the software and the hardware in which, by this contract, the software agrees to some formally specified constraints, and the hardware agrees to appear consistent to at least the software that obeys those constraints [TN95].

Stricter forms of memory consistency typically increase the memory access latency and the bandwidth requirements. More relaxed models result in better performance at the expense of a higher involvement of the programmer in synchronizing accesses to shared data [ML95].

The *memory coherence protocol* determines when and how all the existing copies of the data items existing at one site will be updated or invalidated on the other sites.

It is important now to observe the difference among memory coherence and memory consistency. *Memory coherence* examines in isolation each memory location and the sequence of operations on it, without regard to other locations. *Memory consistency* deals with writes to ***different*** locations and their ordering [TN95].

B. ISSUES ON DISTRIBUTED SHARED MEMORY SYSTEMS

1. Granularity of Sharing

The issue of granularity of sharing can be better addressed by answering the question: "How large should the shared data block be?". Determining the right granularity largely depends on the problem domain and there is no general solution. Each extreme has

3. Also known as adaptive partitioning.

its advantages. The following table gives examples of some granularity choices for existing systems.

It can be observed that hardware or hybrid DSM systems adopt a much smaller granularity. Operating system implementations are, in most of the cases, restricted to use the virtual block size (or multiples thereof) as its unit of reference, while software runtime library DSM systems have the choice of using compiler support (e.g., Midway) or to explicitly declare shared variables (e.g., Munin, Treadmarks, Quarks, etc.) to avoid the burden of being forced to explicitly use pages as the unit of coherence. In the following paragraphs we will analyze some of the issues that involve different block sizes.

Table 1: Block Granularity on DSM Systems

<table>
<tr><th>DSM System</th><th>Block Size</th><th>DSM System</th><th>Block Size</th><th>DSM System</th><th>Data Size</th></tr>
<tr><td>Ivy</td><td rowspan="4">Page size</td><td>Munin</td><td rowspan="4">Shared Data Object</td><td>DASH</td><td>16 bytes</td></tr>
<tr><td>Clouds</td><td>Midway</td><td>KSR-1</td><td>128 bytes</td></tr>
<tr><td>Mach</td><td>Tread-marks</td><td>Memnet</td><td>32 bytes</td></tr>
<tr><td>Mirage</td><td>CarlOS</td><td>FLASH</td><td>128 bytes</td></tr>
</table>

For systems which use fixed block sizes (page-based systems and hardware implementations) one would like to keep the communication cost as low as possible. There are two ways to achieve this goal: using a faster transmission medium or reducing the block size as the equation below suggests.

$$\left(latency\ per\ byte = fixed\ message\ startup\ cost + \frac{block\ size}{Transmission\ Media\ Bandwidth} \right)$$

Clearly, there are advantages and disadvantages in choosing a coarser block size for a DSM system. The biggest advantage of coarse granularity is to reduce the start-up penalty

by amortizing it on a larger number of data bytes. This property is especially important because many programs exhibit "*locality of reference*" [TN95], resulting on a implicit prefetching of data that could be accessed in the near future.

Finer granularity, on the other hand, would be preferred in programs that present a high degree of sharing. For this type of application larger block sizes will only diminish the opportunity of concurrent access to different parts of a shared data block.

In summary, larger blocks are ideal for applications which exhibit low degree of sharing and good locality of reference when compared to the computational granularity, since it minimizes the fixed cost per word transferred. Meanwhile, if the degree of sharing is relatively high when compared to the computational granularity then a smaller block size becomes more attractive. The next section addresses this issue when a high degree of sharing is present.

2. False Sharing

False sharing arises because a DSM system cannot identify updates to individual bytes when protecting regions of memory, while the memory hardware provides control only at the granularity of a data block. Therefore, false sharing occurs when two or more processes update distinct portions of the same data block.

False sharing poses a problem for systems that maintain consistency at the granularity of entire pages or entire objects: every time a thread modifies a page of a shared object, these systems must invalidate or update all copies. It is a particularly serious problem for two reasons:

- The consistency units are large, so false sharing is very common; and
- the latencies associated with detecting modifications and communicating are relatively big, resulting in unnecessary faults and messages that are particularly expensive.

There are two extremes of applications addressing false sharing.

- The problem is ignored (i.e., IVY [KL88]): the consequences are that pages will

"ping-pong" back and forward between processors as can be seen from Figure 5.

• We allow multiple writers to the same data block and rely on the programmer to ensure that no two processors are writing to the same memory location. This approach is known as Multiple Writers and is used by many existing implementations (Munin, Treadmarks, and CarlOS).

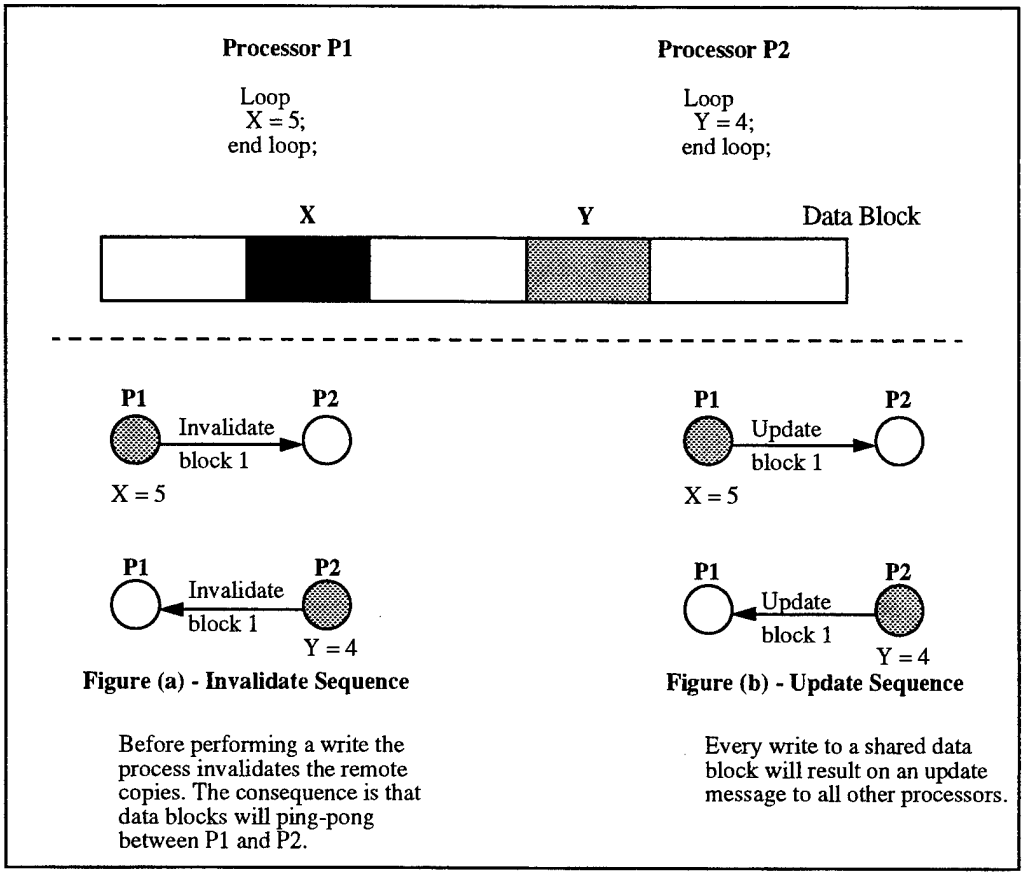


Figure 5: False Sharing.

The problem of false sharing is more extensively addressed in [BS93] in which an attempt for quantifying the problem is given. It is interesting to mention that the cost of False Sharing is small for programs that present a small degree of sharing while the cost becomes prohibitive whenever a high degree of sharing is present. At first glance the naive solution would be to reduce the coherence unit size. This would reduce or even eliminate

the problem. However, for applications in which the data is migratory by nature if this reduction is too large, exactly the opposite happens; the cost gets larger with a smaller block size due to the increase on the number of operations required [BS93].

In summary, the right size for the coherence unit to avoid false sharing without imposing an increase on the number of operations is highly dependent on the degree of sharing of the application type.

3. Synchronization

Most parallel applications running on a Shared Memory Multiprocessor rely on a set of synchronization operations to enforce mutual exclusion and avoid race conditions. For a multiprocessor environment Test-and-Set operations have a reduced cost and are widely used to implement atomic transactions. On the other hand, for a software DSM system this approach is unacceptable [MU94] due to limitations on network bandwidth.

Generally, DSM systems rely on explicit synchronization mechanisms to enforce consistency on shared data. One alternative implementation is to provide Synchronization Manager(s) which will handle the allocation/dealocation of synchronization objects and the corresponding operations (e.g., Acquire/Release of locks). This approach reduces network traffic at the expense of centralized control per synchronization object and is commonly named as "*centralized locking*".

Shared-variable systems like Munin and Midway rely on "*distributed locking*" schemes. More precisely, Munin provides a directory for synchronization variables in which each lock is mapped. On a lock request if the lock is Local (owned by the local processor) it is released to the local thread, otherwise its owner is located through consulting a Synchronization Directory.

4. Heterogeneity

This is an issue that presents no easy solution. Sharing data between two machines with different architectures, and assuming that these two machines may not even use the same representation for basic data types would seen very difficult [BL91].

Some solutions were presented for this problem. In Mermaid [ZH91], memory is shared in pages and each page can contain only one type of data. Whenever a page is moved between two architecturally different systems, a conversion routine converts the data in the page to the appropriate format. An alternative proposal is mentioned in [BL91]. It consists of organizing the shared data as variables or shared objects in the source language and relying on a DSM compiler to add conversion routines to all accesses to shared memory. An example of this approach is found in the implementation of Agora. In this system memory is structured as objects shared among heterogeneous machines.

Albeit the solution of the heterogeneity problem allows the addition of more nodes to DSM systems, it presents a drawback of requiring data conversions on every transaction among heterogeneous platforms. In general, this overhead out-performs the benefits [NL91].

5. Coherence Protocol

The choice of the coherence protocol is related to the granularity of shared data. For very fine grain data items, the cost of an update message is approximately the same as the cost of an invalidation message. Therefore, the update protocol is typical for systems with word-based coherence maintenance and invalidation is used in coarse grain systems. The efficiency of an invalidation approach is increased when the sequences of reads and writes to the same data item by various processors are not highly interleaved [ML95].

In spite of some drawbacks, update protocols are promising in one respect: the number of messages involved. It directly reflects the message passing nature of the underlying system. Updates can be thought of as sending a message containing the state that the application wishes to share among the different parts of the program. Hence, we can expect that when used carefully, the update protocol to perform as well as any message passing implementation of an application [AAL92]. The drawback of this approach is that updates may be sent to nodes that are not on demand to the updated value. To avoid this

problem and, consequently, reduce the number of update messages, two new consistency models were proposed: Lazy Release Consistency and Entry Consistency.

In contrast, the invalidate protocol involves two extra messages to achieve the same effect: the invalidate message to a node caching a given page and the get message for the same page on a subsequent access by the node.

There are also some proposals for a hybrid solution as can be seen in [DKCZ93] in which a hybrid coherence protocol is proposed for Lazy Release Consistency.

A fourth alternative is proposed for the Clouds operating system [MU94]. This approach uses direct association of locks to govern the access to shared cache blocks, allowing data associated with the lock to be sent to the requester along with the lock granting. Upon a release of a lock, the associated data is sent back (if modified) to global memory.

C. DSM MEMORY CONSISTENCY MODELS

In this section we introduce the more well known consistency models and enumerate some of their strengths and weaknesses. It is important to observe that the models are listed in increasing order of flexibility.

Before we proceed, we need to define what it means to perform a *memory request* and a *memory load.* The following formal definitions are extracted from [GLLG90]. In both definitions P_i refers to processor *i*:

Definition 1: Performing a Memory Request

A *LOAD* by P_i is considered performed with respect to P_k at a point in time when the issuing of a *STORE* to the same address by P_k cannot affect the value returned by the *LOAD*. A *STORE* by P_i is considered *performed* with respect to P_k at a point in time when an issued *LOAD* to the same address by P_k returns the value defined by this *STORE* (or a subsequent STORE to the same location). An *ACCESS (LOAD/STORE)* by P_i is performed when it is performed with respect to all processors.

Definition 2: Performing a *LOAD* Globally.

A *LOAD* is *globally performed* if it is performed and if the *STORE* that is the source of the returned value has been performed.

Definition 3: Performing a STORE Globally.

A *STORE* is *globally performed* if it is performed and if all immediate subsequent *LOADS* from the corresponding memory location return the value issued by the *STORE*.

After the three above definitions we are ready to describe some of the existing memory consistency models.

1. Strict Consistency Model

This is the most stringent consistency model. It is defined by the following condition:

"Any read to a memory location x returns the value stored by the most recent write operation to x." [TN95]

In other words, when memory is strictly consistent, all writes are instantaneously visible to all processes and an absolute global time order is maintained. If a memory location is changed, all subsequent reads from that location will see the new value, no matter how soon after the change the reads are done and no matter which processes are doing the reading and where they are located. This type of memory consistency is easily achieved on a uniprocessor system, but it is almost impossible to guarantee on a multiprocessor environment, without explicity synchronizing on all *STORE* operations.

2. Sequential Consistency Model

The sequential consistency is defined by Lamport [LAM79] as follows:

"....A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of individual processor appear in this sequence in the order specified by its program..."

The following are the sufficient conditions for providing sequential consistency:

- Before a *LOAD* is allowed to perform with respect to any other processor, all

previous *LOAD* accesses must be globally performed and all previous *STORE* accesses must be performed; and

• Before a *STORE* is allowed to perform with respect to any other processor, all previous *LOAD* accesses must be globally performed and all previous *STORE* accesses must be performed.

Figure 6 exemplifies a program that runs concurrently on two distinct processors.

Processor 1	**Processor 2**
1. x = 1;	2. y = 1;
3. if (y == 0)	4. if (x == 0)
5. kill P2;	6. kill P1;

Valid outcomes under a sequentially consistent program would be: Processor P1 or Processor P2 being killed, none of them being killed, but never both processors.

Figure 6: An example of a sequentially consistent program.

In summary, Sequential Consistency requires that the distributed memories in a DSM have the same consistency properties as a time shared uniprocessor, which requires that the global state of memory be consistent after every read or write to shared memory. This requirement imposes severe restrictions on possible performance optimizations.

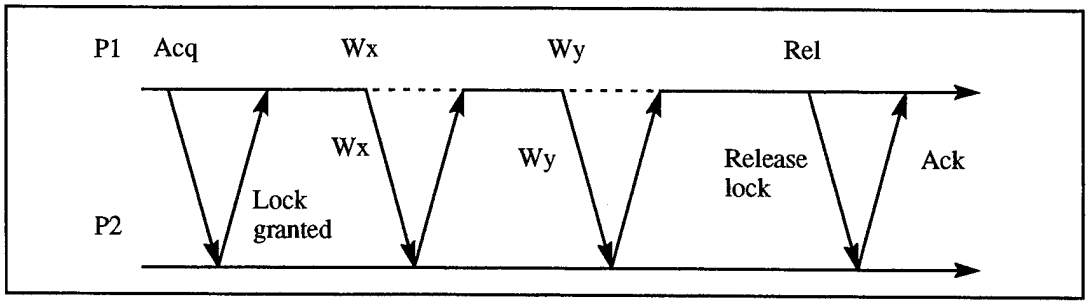


Figure 7: Communications on the Sequential Consistency Model. [4]

4. Dotted lines represent the intervals in which the processor should stall.

The reason why sequential consistency is inhenritly inefficient can be observed in Figure 7. Every write operation forces the system to *stall* until the corresponding data block is either invalidated or the updates are propagated to the other processors. More formally, every *STORE* operation stalls the processor until it is performed.

Therefore, when both processors P1 and P2, in Figure 7, have cached the same copies of the variables X and Y within a critical section, each write must be delayed until the previous write completes even within a critical section. This will, besides requiring a large number of messages, have a large delay due to the periods processor P1 must stall (represented by dotted lines on Figure 7) while communicating. The use of sequential consistency still requires synchronization when preemptive scheduling is used.

3. Processor Consistency

The concept of Processor Consistency was introduced by Goodman [GVW89]. It requires that *writes* issued from a processor may not be observed by other processors in any order other than the one in which they were issued. Specifically, this model relies on the use of explicit synchronization to guarantee strict event ordering. The following conditions are necessary for processor consistency:

- Before a *LOAD* is allowed to perform with respect to any other processor, all previous *LOAD* accesses must be performed; and
- Before a *STORE* is allowed to perform with respect to any other processor, all previous accesses (*LOADS* and *STORES*) must be performed.

The above conditions allow reads following a write to bypass the write. To avoid deadlock, the implementation should guarantee that a write that appears previously in program order will eventually perform [GLLG90]. Here we need to demonstrate the subtle difference between Processor Consistency and Sequential Consistency. We will use the example of Figure 8 to illustrate that.

According to Ahmad [AHJ90] for a sequentially consistent program outcome to be legal it must obey two constraints:

- Program order must be maintained; and

• Memory coherence must be respected.

Processor 1	Processor 2
1. x = 1;	2. y = 1;
3. if (y == 0)	4. if (x == 0)
5. kill P2;	6. kill P1;

A valid sequence for a processor consistent program would result on both processors P1 and P2 being killed. This outcome is possible since processor consistency allows reads to bypass writes. So y == 0 on P1 and x == 0 are true statements.

Figure 8: Example of a processor consistent program.

Processor Consistency, in contrast, is more relaxed since it only requires that writes issued from a processor may not be observed in any order other than that in which they were issued. As can be observed, Processor Consistency may not issue the correct result if the programmer is expecting sequential consistency, thus requiring the use of explicit synchronization by the programmer to enforce sequential consistency.

4. Causal Consistency Model

Causal Consistency can be defined as:

"An execution on causal memory is correct if the value returned by each read operation in the execution belongs to a set of correct values for that location."

More precisely, writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines [TN95]. For causally related events we mean that an event A is caused by or influenced by event B according to the definition of event ordering from Lamport [LAM79]. *Concurrent events* are those events that are not causally related. Figure 9 describes a sequence of events that are Causally consistent but violate sequential consistency. In this figure the symbol *W (x) a* represents a STORE operation where the

value "a" is stored on the variable "x". By *R (x) a* we mean that a LOAD of variable "x" has the returned the value "a".

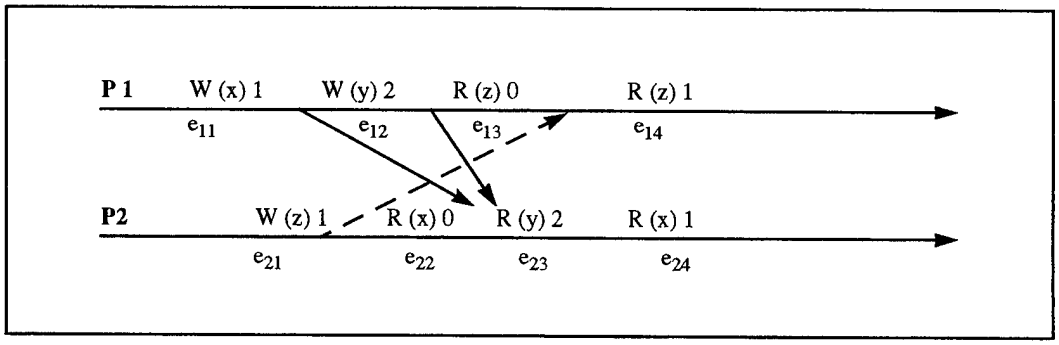


Figure 9: Causal Consistency Model.

As can be observed, the "*R (x)*" operation on processor P2 has returned the value "0" . This represents that both events "e_{11}" and "e_{22}" are concurrent, while events "e_{23}" is causally related to event "e_{12}".

Causal Consistency was implemented on the Clouds Distributed Operating System. This implementation uses a vector timestamp [LAM79] to capture the evolving causal relationships. This implementation of Causally Consistent memory will use invalidates to resolve inconsistencies and, although more relaxed than Sequential Consistency, it does not resolve the problem of false sharing.

5. Weak Consistency Model

A consistency model can be derived by relating memory requests ordering to synchronization points in the program [GLLG90]. The delays imposed by the sequential consistency model are unnecessary if appropriate synchronization mechanisms are enforced. Given that all synchronization points are identified, we need only to ensure that memory is consistent at those synchronization points. This scheme has the advantage of

permitting multiple memory accesses to be pipelined [AH90]. Sarita and Hill define Weak Ordering as follows:

"Hardware is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all software that obey the synchronization model."

The synchronization model defined in [AH90] is named Data-Race-Free-0 (DRF0) and is closely related to the happens-before relation [LAM79] which can be defined as the irreflexive transitive closure of program order and synchronization order:

$$\xrightarrow{hb} = \left(\xrightarrow{po} \cup \xrightarrow{so} \right)^{+}$$

where *po* represents program order and *so*, synchronization order, respectively.

The complete definition for the synchronization model DRF0 is given below:

A program obeys the synchronization model DRF0 if and only if: (1) all synchronization operations are recognizable by the hardware and each accesses exactly one memory location, and (2) for any execution of the idealized system all conflicting accesses are ordered by the happens-before relation corresponding to the execution. In this definition, two accesses are said to conflict if they access the same location and they are not both reads. The figure below describes an example of an execution that obeys the DRF0 model.

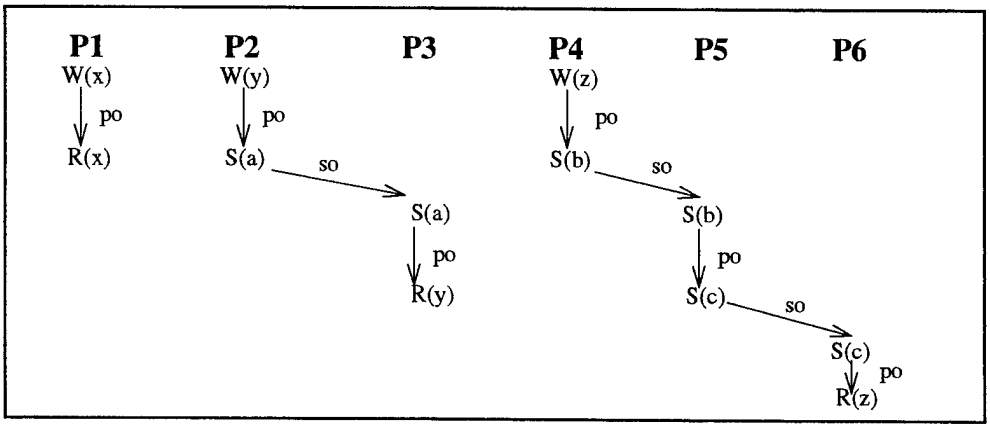


Figure 10: An example of DRF0 [AH90].

In [GLLG90] a slightly different set of conditions is listed, which for consistency reasons we will adopt for the remainder of this document:

- Before an ordinary *LOAD* or *STORE* access is allowed to perform with respect to any other processor, all previous *synchronization accesses* must be performed;
- Before a *synchronization access* is allowed to perform with respect to any other processor, all previous ordinary *LOAD* and *STORE* accesses must be performed; and
- *Synchronization accesses* are **sequentially consistent** with respect to one another.

The first and second rules assure that the instructions inside the critical section stay inside and those outside stay outside as observed by any other processor. The third rule assures that the synchronization variables can create critical sections. The example below depicts a correct and an incorrect outcome for a Weakly Ordered program.

As can be seen from Figure 11a there are no guarantees for the outcome before the synchronization point. In contrast, after the synchronization is performed the local memory should be brought up to date returning the most recently values written to it. Therefore, on Figure 11b it can be noted that the value returned by processor P2 is invalid.

In summary, Weak Consistency assures the correctness of parallel programs by placing tight restrictions on synchronizing instructions and loose restrictions on ordinary instructions.

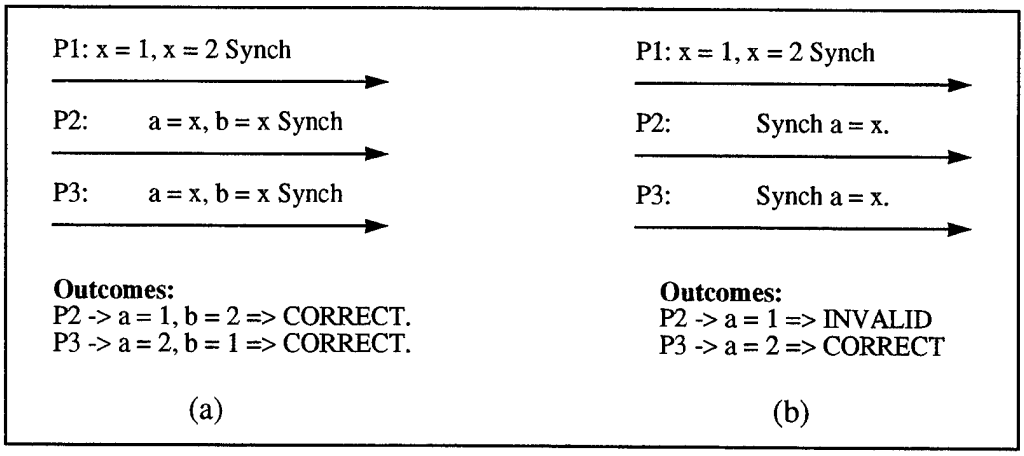


Figure 11: Weak Consistency valid (a) and invalid (b) sequences of events.

6. Release Consistency Model

Release Consistency is an extension of weak consistency that exploits the information about acquire, release, and non-synchronization accesses. To better describe the designation below we must describe the notions of *competing* and *conflicting accesses*. Two accesses by one or more processors are conflicting if they are to the same memory location and at least one of the accesses is a *STORE*. If a pair of conflicting accesses execute simultaneously, causing a race condition, then such accesses form a competing pair. If an access is involved in a *competing pair*, then the access is considered a *competing access* [GLLG90].

The following gives the conditions for ensuring release consistency:

- Before an ordinary *LOAD* and *STORE* access is allowed to perform with respect to any other processor, all previous *acquire accesses* must be performed;
- Before a *release access* is allowed to perform with respect to any other processor, all previous ordinary *LOAD* and *STORE* accesses must be performed; and
- *Special accesses* (acquire and release) are ***processor consistent*** with respect to one another.

In the above definition ordinary accesses are represented by non competing accesses and special accesses denote the competing ones.

Therefore, release consistency relaxes the constraints of sequential consistency in three ways:

- Ordinary reads and writes can be buffered and pipelined between synchronization points;
- Ordinary reads and writes following a release do not have to be delayed for the release to complete (i.e., a release only signals the state of past accesses to shared data); and
- An acquire access does not have to delay for previous ordinary reads and writes to complete. [CBZ91].

When compared with Weak Consistency we can observe that four of the ordering restrictions present in Weak Consistency are not present in Release Consistency. First is that ordinary *LOAD* and *STORE* accesses following a release access do not have to be

delayed for the release to complete. Second, an acquire synchronization access need not be delayed for previous ordinary *LOAD* and *STORE* accesses to be performed. Third, a non-synchronization special access does not wait for previous ordinary accesses and does not delay future ordinary accesses. The fourth difference lies in the ordering of special accesses. For Weak Consistency, the accesses are sequentially consistent while for Release Consistency the accesses can be Processor Consistent.

This model was developed as part of the DASH project and proved effective at hiding the effects of memory latency by pipelining invalidation messages caused by writes to shared data [CBZ91]. Figure 12 depicts the gains in the communication costs that are achievable through this approach. As can be observed, the processor needs only to stall at the time of a release operation, *STORES* are pipelined. Therefore, it introduces a large optimization when compared to a sequentially consistent program (Figure 7).

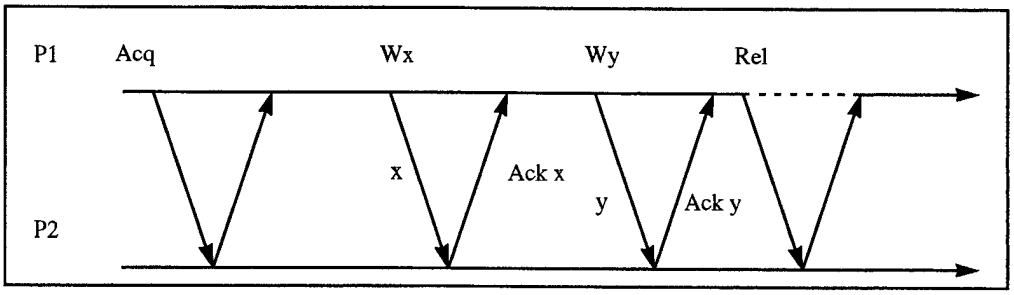


Figure 12: Release Consistency model.

Besides the generic model adopted for the DASH implementation there are two other variants for Release Consistency: Eager Release Consistency (e.g., Munin) and Lazy Release Consistency (e.g., Treadmarks).

a. Eager Release Consistency

When a thread performs a release, it stalls until all modifications to shared data have been performed (invalidated/updated). This new scheme buffers writes to shared data until the subsequent release, at which point it flushes the buffered writes. Ideally, this

strategy reduces the number of messages transmitted from one per write to one per critical section when there is a single replica of the shared data.

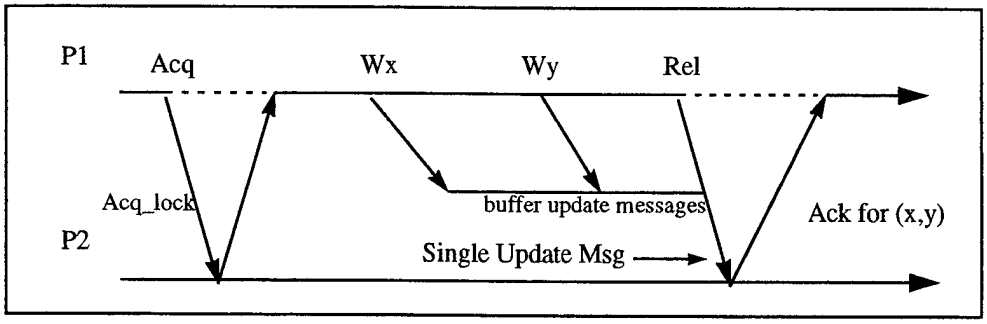


Figure 13: Eager Release Consistency model.

As we observe in Figure 13, this approach increases the latency of a release when compared to the Release Consistency Model. Nevertheless, the reduction in the number of messages may outweigh the effect of higher release latencies. Carter [CBZ92] also proposes the use of Update instead of an Invalidate-based coherence protocol since the above approach only solves the cost of writes, but has no effect on read misses. When the ratio of read/write to shared data is relatively high, the effect of read misses can be mitigated by using an update-based protocol. This approach is feasible when used in combination with the buffered approach as is the case of Munin.

b. Lazy Release Consistency

When a thread performs an acquire, all "stale" data is discarded or updated. This approach is adopted in the implementation of Treadmarks [ACDB94] and CarlOS [KF94]. Compared with Eager Release consistency it causes fewer messages to be exchanged. At the time of a lock release, Munin sends messages to *all processors* which cache data modified by the releasing processor. In contrast, in Lazy Release Consistency (LRC) messages only travel between the *last releaser and the new acquirer.*

LRC is somewhat more complicated than eager release consistency. After a release, Munin can forget about all modifications that the releasing processor made prior to the release. This is not the case for LRC, since a third processor may later acquire the lock and need to see the modifications.

More formally, in Lazy Release Consistency the propagation of modifications is further postponed until the time of the acquire. At this time, the acquiring processor determines which modifications it needs to see according to the definition of Release Consistency. To do so, LRC uses a representation of the happened-before relation introduced by Adve and Hill [AH90]. Release Consistency requires that before a processor may continue past an ***"acquire"***, all shared accesses that precede the acquire according to happens-before-1 relation must be performed at the acquiring processor. LRC guarantees that this property holds by propagating *write-notices* on the message that affects a release-acquire pair. A write-notice is an indication that a page has been modified in a particular interval, but it does not contain the actual modifications. Each new interval begins with each special access performed by the corresponding processor. Such intervals are, in turn, used inside Vector Clocks [LAM79] to enforce the *happened-before* relation among the processors.

On an *acquire*, the acquiring processor, P_i, sends its current vector timestamp to the previous releaser, P_j. Processor P_j uses this information to send to P_i the write-notices for all intervals of all processors that have performed at P_j but have not yet performed at P_i. Releases are pure local operations in LRC and no messages are exchanged.

For the case of an update coherence protocol the acquiring processor updates all pages for which it received write-notices. In contrast, for an invalidate protocol, the acquiring process invalidates all pages for which write-notices were received. Figure 14 gives an example of both update and invalidate protocols.

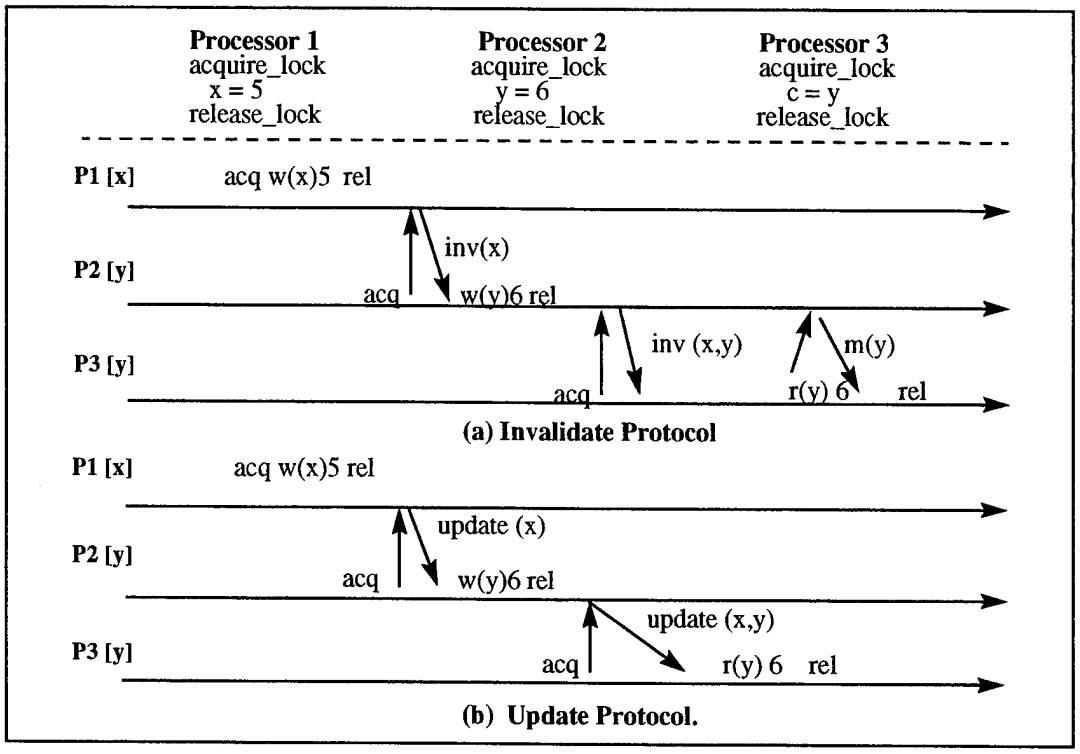


Figure 14: LRC Invalidate protocol (a) and LRC update protocol (b).

Figure 15 compares LRC with the generic version of Release Consistency. As can be observed, Lazy Release Consistency will have fewer messages, but the implementation of such mechanism will be far more complex. The number of messages will also be smaller than implementations of Eager Release Consistency (Munin), since for both invalidate and updated protocols it will be required that every process that is in the copyset receives a release message.

7. Entry Consistency Model

Entry Consistency was introduced with the Midway DSM system [BZS93]. For entry consistency, data is only consistent on an acquiring synchronization operation, and only the data known to be guarded by the acquired object is guaranteed to be consistent at

the time of the acquire. Communication between processors occurs only when a processor acquires such synchronization objects.

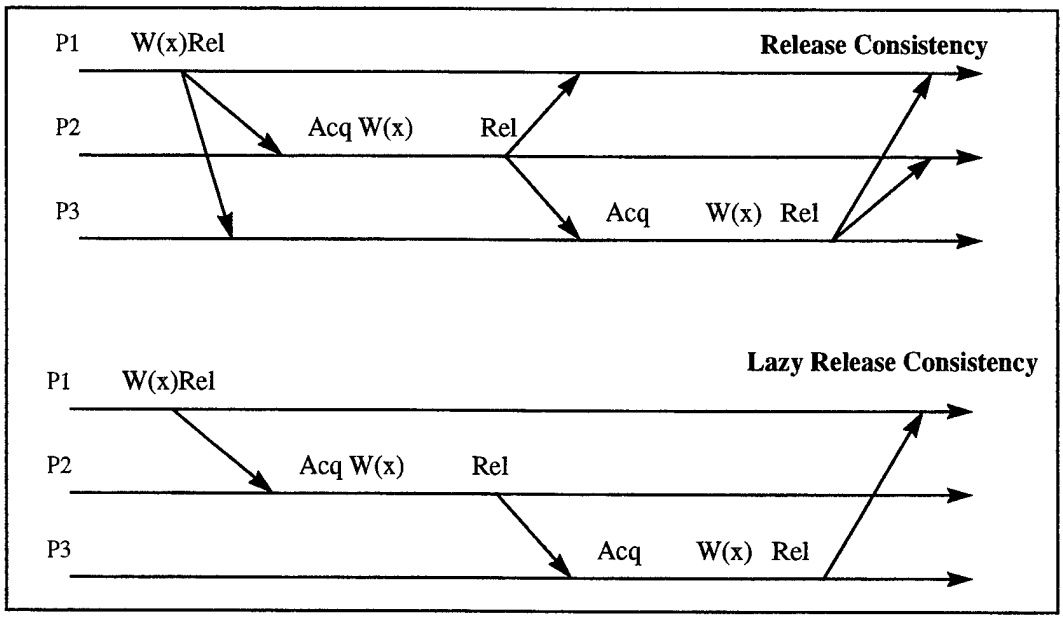


Figure 15: Comparison between RC and LRC models.

Formally, a memory exhibits entry consistency if it meets the following conditions:

- An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process;
- Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode; and
- After an exclusive mode access to a synchronization variable has been performed, any other processor's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

The first condition states that when a process does an acquire, the acquire may not complete until all the guarded shared variables have been brought up to date.

The second condition states that before updating a shared variable, a process must enter a critical region in exclusive mode to ensure that *no* other process is trying to update it at the same time.

The third condition declares that if a process wants to enter a critical region in nonexclusive mode, it must first check with the owner of the synchronization variable guarding the critical region to fetch the most recent copies of the guarded shared variables.

Although entry consistency enables the use of low overhead consistency mechanisms, writing an entry consistent program requires more work than writing one on a more stronger model. For example, every synchronization object must be identified; every use of such an object must be explicit; every shared data item must be associated with a synchronization object; and synchronization accesses should be qualified as read-only or read-write for best performance [BZS93].

In summary, entry consistency requires:

- Shared data to be accessed inside critical section;
- All shared data has to be associated with a single synchronization variable (e.g., a lock); and
- When a lock is acquired (entry to a critical section), only those variables associated with lock are made consistent.

In the next chapter we describe the main features of existing systems focusing on both hardware and software implementations.

III. DSM SYSTEMS OVERVIEW

In this chapter we review the main features of some existing DSM implementations. We divide the reviewed systems into hardware and software implementations. Our goal is to point out the strengths and weaknesses of each individual system. These features will be recalled again when our approach is described on Chapters IV and V.

A. HARDWARE IMPLEMENTATIONS

1. KSR-1

The KSR-1 implements the "*ALLCACHE*" [RO92] memory model. The "ALLCACHE" is a hardware message-based distributed virtual memory system which enforces the Sequential Consistency Memory Model. Each processor is associated with a 32 MB cache unit, all of which are tied together by a very fast slotted-ring communications mechanism, across which a single address space is defined. The KSR-1 architecture exploits locality of reference by organizing a number of "ALLCACHE" Engines in a hierarchy. At the lowest level are the ALLCACHE Group:0s (with 32 processors in each group) which consists of the ALLCACHE Engine:0s and the local caches associated with them. Therefore, an ALLCACHE Engine:0 contains the directory which maps from addresses onto the set of local caches within its group. An ALLCACHE Engine:1 includes the directory which maps from addresses into its constituent set of ALLCACHE Group:0s. The ALLCACHE Engine is constructed with a "*fat-tree*" topology so that the bandwidth increases at each higher level of the ALLCACHE Engine.

The distinctive feature in this design becomes apparent when data is required which is not located within the local memory and a request is generated: what is returned is not simply the data, but the address as well. They are returned because there are two types of addresses on the system: System Virtual Address (SVA) and Context Address (CA). The SVA is a 64-bit global system-wide reference of any given location (at byte level) in the memory system. The CA type are the addresses referenced by each individual task. CA

addresses are translated into a SVA by the processor using a translation table managed by the ALLCACHE Engine. CA memory is allocated by segments that map into SVA segments. Segments in different CA spaces may be mapped to the same segment in SVA space, thus allowing sharing between two processors. In these processors each page is a set of 128 subpages of 128 bytes each (total 16 KB). Pages (16 KB) are used as allocation unit and subpages (128 bytes) as coherence unit for the system.

Memory within the KSR-1 is formed within a scalable hierarchy based on the average latency to return an address from a given initial location. Physically, there are four levels of memory access:

- A 512 KB subcache for each processor.
- A 32 MB cache for each processor cell.
- A 996 MB remote cache, on the same local ring:0.
- A 31744 MB remote cache, on distinct rings (ring:1).

The communication rings (interconnect) are hierarchically structured, with a first-level ring (Ring:0) grouping 32 processors together, and a second-level ring (Ring:1) grouping together up to 34 first-level rings. Figure 16 describes this hierarchy.

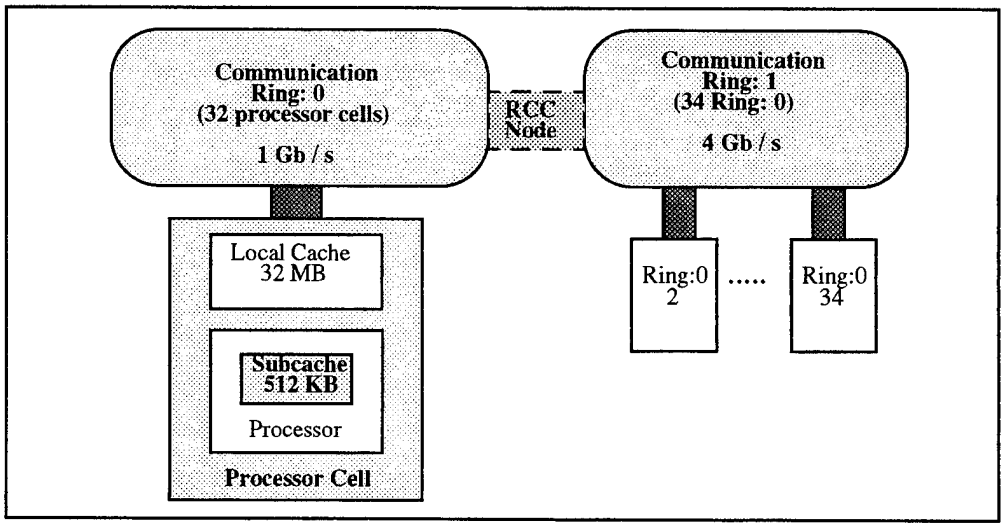


Figure 16: The KSR-1 ALLCACHE Hierarchy.

A subpage fault (128 bytes) will generate a request which is placed into an open slot on the communications ring. The slot is matched against the cached elements associated with every processor on the local ring (ring:0). If one of the processors is the owner of the data element requested it will put the data plus address on the open slot, otherwise this request is forwarded through the Ring Routing Cell (RRC) to the next hierarchical level (ring:1).

The coherence protocol enforced by this architecture is ***write-invalidate*** and a ***snoop read-broadcast*** [HN93]. Therefore, at any time there is a unique block owner on the system. Whenever a task tries to write to a location, the ownership of that subpage is transferred to that processor and an invalidate message is sent to other processors that currently cache copies of that data.

The system provides software instructions that allows for remote data store on other processors and also issues prefetch calls retrieving data before it is actually needed.

2. DASH

The DASH architecture's main feature is the introduction of a new consistency mechanism "*Release Consistency*". As for other hardware implementations in which the unit of coherence is small (16 bytes), DASH also adopts a write-invalidate coherence protocol.

The DASH system consists of a two-level, hierarchically organized structure. At the top level, the system consists of a set of processing nodes (clusters) connected through a pair of wormhole meshes (**F**igure 17) where each processing node consists of four processors linked through a bus-based connection (**F**igure 18).

Intra-cluster cache coherence is implemented using a snoopy bus-based protocol, while *inter-cluster* coherence is maintained through a distributed directory-based scheme by implementing an invalidation-based coherence scheme. It is the directory responsibility

to summarize the information for each memory line, and to specify the clusters that are currently storing it [LLJN92].

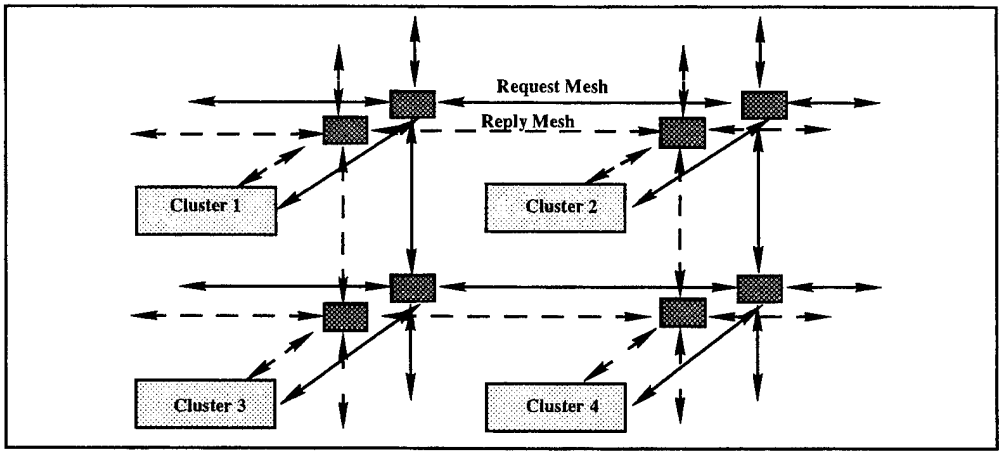


Figure 17: DASH High Level Structure.

The DSM mechanism for the DASH prototype implements a MRSW Type algorithm. Therefore, each memory location can be in one of three states:

- *Uncached:* not cached by any processing node at all;
- *Shared*: in an unmodified state in the caches of one or more nodes (Multiple Readers); or
- *Dirty*: in a modified state in the cache of some individual node (Single Writer).

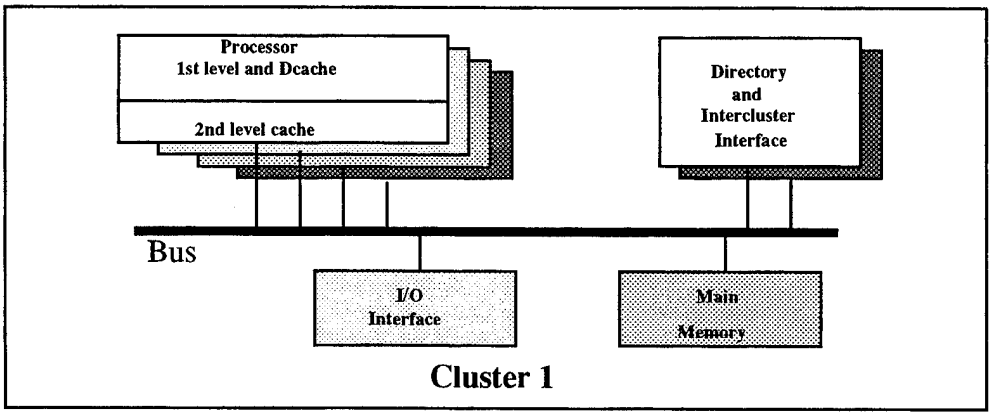


Figure 18: DASH Processing Node.

The key part of the DSM mechanism is the Distributed Directory which is implemented on all clusters. Each memory location has an assigned Home Directory. Data ownership can dynamically change whenever a processor performs a write request on the same address managed by the Home directory of another cluster. In this situation the memory location becomes dirty and the Home Directory invalidates all remote copies cached on remote clusters. Note that while data ownership can dynamically change the Home Node for any particular block remains fixed. On a memory request, the Home Directory takes one of two actions: if the memory location is dirty it forwards the request to the current owner, otherwise the requesting block is included on the copyset list and the data is forwarded. The problem with the Directory approach is its limited scalability due to the use of a *bit vector* with 1 bit for each cluster. A possible solution would be the use of a limited-pointer directory as used on the FLASH implementation [KOHH94].

Besides supporting the Release Consistency model, where writes are pipelined, DASH uses *software-controlled nonbinding prefetching* [LLJN92] to hide the network latency effect. DASH also provides efficient Fetch-and-Op primitives to reduce the synchronization overhead.

B. SOFTWARE IMPLEMENTATIONS

1. Operating System Level - *Clouds*

The Clouds operating system belongs to a class of *object-based* distributed operating systems and is built on top of a minimal kernel called *Ra*. The paradigm supported by Clouds provides an abstraction of storage called *objects* and an abstraction of execution called *threads*. All data, programs, devices, and resources are encapsulated in *objects*. Therefore, *objects* represent the *passive entities* of the system. Activity is provided by *threads*, which execute within objects.

At the conceptual level, an object is a virtual address space. In contrast to conventional operating systems, objects in Clouds are persistent and are not tied to any thread. Since it does not contain a process, it is completely passive. The contents of each

virtual address space are protected from outside accesses so that memory in an object is accessible only by the code in that object and the operating system. In summary, each object is an encapsulated address space with entry points at which threads may commence execution [DCMP91].

To allow concurrent execution of more than one computation in the same object, the system provides a set of shared memory style synchronization primitives. The unit of sharing in Clouds DSM is a segment. Associated with each segment is a node called the *owner* where the segment resides on stable storage. The DSM Server object at the owner node is responsible for maintaining the consistency of the segment.

To unify synchronization with data transfer, Clouds adopts a "*lock-based*" coherence protocol. In this protocol lock requests (both exclusive and shared) result in the page associated with the lock being sent to the requester along with the granting of the lock, if and only if the requesting mode is compatible with the current mode for the segment, otherwise the request is queued. Upon lock release, the associated page is sent back (if modified) to the server. Reads or writes to shared data without explicit locking follow single copy semantics that do not allow multiple readers or writers. For this purpose two primitives are supported: *get* and *discard*. In short, the Clouds DSM system is implemented integrated with the operating system providing a low overhead when manipulating segment misses. Also, by enforcing consistency at defined synchronization points, this DSM system enforces the Release Consistency Model. The lock-based coherence protocol was an innovation when compared with existing DSM systems.

2. Runtime Libraries

a. Midway

Midway is a Distributed Shared Memory System that supports the Entry Consistency Model. As described in Chapter II, Entry Consistency is a relaxed consistency protocol that requires the explicit association of shared data to synchronization objects. Upon a release operation the changes to the data associated with the lock are propagated to

the new acquirer by sending modifications (*diffs*) to the data object. To keep consistency, each processor stores a set of diffs to the corresponding object. To reduce communication traffic and guarantee that all changes are made visible to all processes, each process keeps a monotonically increasing counter (logical clock [LAM79]) which is incremented whenever a synchronization access is performed. At the acquire time the requester sends also its Vector Clock. The lock owner, in turn, forwards all changes that are greater than the received timestamp. It is the acquirer's duty to coalesce all changes and update its diff set. If this set's size becomes greater than the data associated with the lock, the data itself is propagated to the new lock owner.

Lock ownership is defined using a Distributed Queue Algorithm similar to the Mach's shared memory server [FBYR88]. This algorithm is based on the probable owner concept. The lock request is sent to the node that is currently the probable lock owner. If the node that has received the request does not own the lock anymore it forwards the request to the next probable owner, creating a chain of messages. This algorithm has a worst case complexity of O (n), where "*n*" is the number of processes that are accessing the same critical section. The drawback is that it has "*n*" possible points of failure per lock transaction. **F**igure 19 provides an example of this algorithm as originally proposed by Florin [FBYR88]. In **F**igure 19a processor P_1 performs a lock request. The request is sent to the "probable owner", the node designated as root. Since the lock ownership has already changed to P_2 the root node forwards the request to the next probable owner, P_2. Again the current owner has altered and the request is forwarded to P_3 which then releases the lock (**F**igure 19b). The process is repeated again when P_2 performs a lock request (**F**igure 19c).

Synchronization objects can be of two types: non-exclusive (data can only be accessed on read operations) and exclusive (data can be accessed for both read and write operations). The synchronization objects ownership is exclusive. By exclusive ownership we mean that each synchronization object, at any time, has an unique owner. Replication

of data is only allowed for data whose synchronization access is non-exclusive (read-only data).

Midway also provides two other consistency models: processor and release consistency. The idea is to allow the programmer to develop his application initially using a stronger consistency model and use Entry Consistency on further refinements based on the data access patterns.

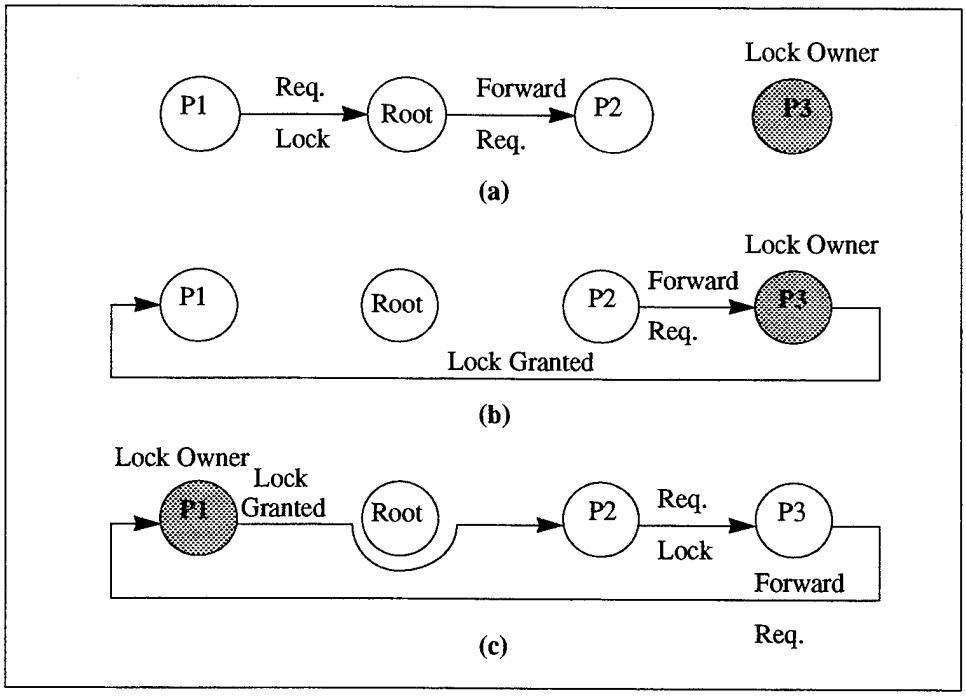


Figure 19: Distributed queue locking scheme.

There are two implementations for Midway: VM-DSM (Virtual Memory DSM) [BZS93] and RT-DSM (Runtime DSM) [ZSB94]. Although both versions adopt the Entry Consistency Model as the basic consistency mechanism, their difference lies on the embraced strategy for detecting and collecting writes to shared data in a software-based DSM. Both strategies rely on compiler assistance to insert primitives for marking shared data objects as dirty. In the VM-DSM approach the coherence is constrained to the

corresponding page size. On the other hand, on a RT-DSM the coherence unit is flexible, since the compiler inserts write-detection primitives on every store operation. It is interesting to consider the difference(s) between the two implementations. We summarize below some of the reasons for RT-DSM.

Table 2: Distinction Between RT and VM implementations

Problems with VM-DSM implementation
1. Writes have high overhead since they are detected with a page fault. This cost is amortized if there is a large number of writes per page.
2. The page size is generally too big to serve as a unit of coherence, inducing false sharing. As we saw in Chapter II, there is a limit that we can reduce the unit of coherence without inducing larger overheads due to problems of spatial locality.
Solutions on RT-DSM implementation
1. It tends to have lower average update latency because it can avoid the Operating System altogether.
2. RT-DSM directly supports variable sized objects eliminating false-sharing and the overhead necessary to accommodate it.
3. RT-DSM efficiently provides a detailed update history, which allows it to minimize the data transferred to maintain consistent memory.

In summary, for coarse-grained applications that exhibit little actual sharing, a VM-DSM is advantageous. In contrast, for a program that synchronizes frequently, the RT-DSM system may have better performance.

b. Munin

Munin ([CBZ91], [CBZ92]) is a software DSM system that implements the Eager Release Consistency Model.

The Munin runtime manipulates two major data structures as depicted in Figure 20: a delayed update queue and an object directory. The former is used to buffer the

updates until a release operation is performed. The latter maintains the state of the shared data set being used by the local user threads. On the Object Directory all shared variables on the same physical page are treated as a number of independent page-sized objects. In contrast, variables that are larger than a page are handled as a number of independent data objects.

Instead of the Centralized approach adopted on previous software implementations, Munin employs a Distributed Directory scheme similar to the one used in DASH. The scheme is enforced with the aid of two concepts: dynamic data ownership protocol (viz Midway) and by distributing the state information of write-shared data through the copyset nodes.

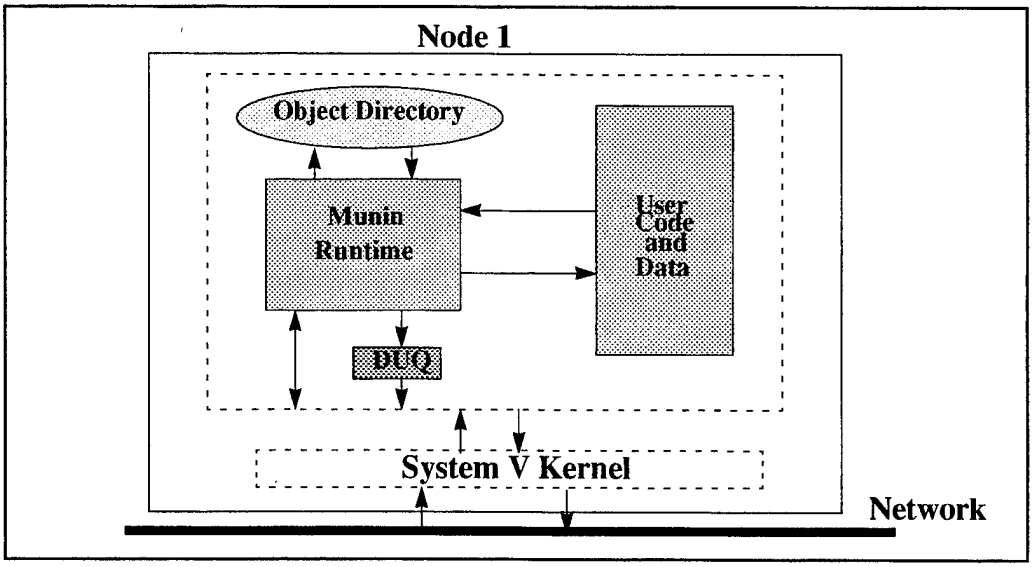


Figure 20: Munin Runtime System [CBZ92].

The major goal of this system was to reduce the amount of communication needed to support DSM. Towards this purpose, Munin introduced three innovative features when compared to previous DSM implementations:

(1) **Software Release Consistency.** Munin implements Release Consistency inspired on the DASH project [LLGN92]. The major distinction between the two approaches is that Munin buffers the updates until the release is performed, while the DASH system pipelines them. Both systems send the updates to all nodes that are known to be currently holding a copy of the page. This implementation of Release Consistency became known as Eager Release Consistency.

(2) **Multiple consistency protocols.** Based on observations on the data access patterns, five major types were distinguished: *conventional*, *read-only*, *migratory*, *write-shared* and *synchronization.*

- ***Conventional*** shared variables are replicated on demand and are kept consistent using an invalidation-based protocol, that requires an owner to be the sole owner of that copy. For ***Read-only*** shared data once initialized no further updates can occur.
- ***Migratory data*** is typically the data that is accessed within critical sections and is made consistent by sending an update message to the new owner and invalidating the local copy.
- ***Write-shared*** variables are frequently written by multiple threads concurrently. Its main advantage is to allow considerable reduction of the False Sharing effects.
- ***Synchronization Variables.*** There are three types of synchronization variables which are supported by the system: locks, barriers, and condition variables. These variables are accessed only through special synchronization primitives provided by library routines. The locking protocol is the same as the one adopted on Midway.

Modifications to the shared-variables are buffered until synchronization requires their propagation. To reduce the message size each process sends only the modifications that were applied to the page. As can be seen from **F**igure 21 each page is initially write protected. When the local thread tries to write into it, a twin copy is generated and the original page is marked as writable (**F**igure 21a). When a release operation is performed the page is compared with its twin copy and the resulting diff is send to the other copyset nodes (**F**igure 21b).

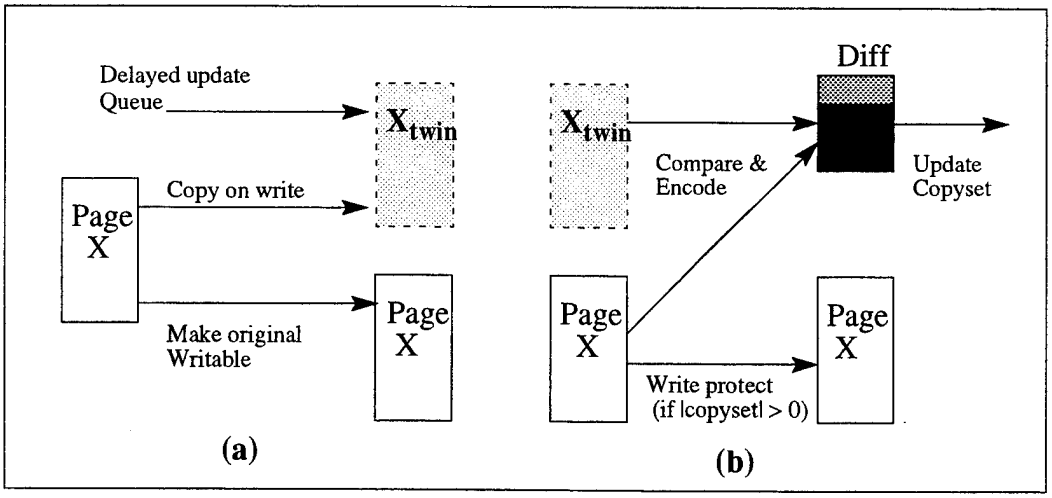


Figure 21: Write-Shared Protocol: twin creation when a page is accessed for writes (a) and sending out diffs (b) when a release operation occurs.

(3) **Update with timeout.** To avoid sending unnecessary update messages to nodes that still remain in the copyset, but are not referencing the page, Munin implements a timeout mechanism in which copies which are not accessed during the last timeout interval are discarded.

c. Treadmarks

Treadmarks [ACDB94] is a software library DSM system that implements the Lazy Release Consistency Model as a consistency mechanism. LRC, described in Chapter II, performs the updates at the time of an acquire instead of the release as is the case for the Eager model (Munin). Another difference among the two models is that the updates are forwarded only to the process that is acquiring the synchronization variable instead of sending updates to all processes that currently cache that data. This considerably reduces the number of messages and the delay on synchronization operations.

This set of optimizations do not come for free. After an update, Munin (Eager Release Consistency) can forget about all the changes the releasing processor made

prior to the release. This is not the case for Treadmarks (LRC). On a release operation the set of modifications (diffs) have to be cached as is the case of Midway [BCS91], so that a third processor is able to see all alterations that were performed to the data block. To do so Treadmark uses vector timestamps to represent the happened-before-1 partial order defined in [SH93].

When a processor executes an acquire, it sends its current vector timestamp in the acquire message. The process that has last performed a release (and is currently the lock owner) then piggybacks on its response a set of *write notices*. These write notices describe the shared data modifications that precede the acquire according to the partial order. As described on Chapter II, a write-notice is an indication that a page has been modified, but it does not contain the actual changes. The acquiring process then determines which of the incoming write notices contain vector timestamps larger than the timestamp of its copy of that page in memory. For these pages, the shared data modifications described in the write notices must be reflected in the acquirer's copy. To accomplish this Treadmark currently invalidates its copies. It is worth mentioning that in [DKCZ94] it is proposed to employ a hybrid coherence protocol in which modifications performed at the releasing node are updated while for pages for which write-notices and no updates were received will be invalidated.

C. COMPILER INSERTED PRIMITIVES - *ORCA*

Orca is an *object-based* language[1] whose sequential statements are based roughly on *Modula-2*. Orca was originally designed for the Amoeba distributed operating system and it depends on the Operating System's reliable broadcast feature to enforce consistency among objects that are replicated.

Orca provides two important features for distributed programming: *objects* and the *fork* statement. *Objects* are like Abstract Data Types in Ada83. It encapsulates internal data

1. By object-based language we mean a language with no support for inheritance and some forms of polymorphism.

structures and methods for manipulating them. Each method can be viewed as a pair of statements: guard + block. The *fork* statement is used to create new processes on a user-specified processor. Parameters, including objects, may be passed to the new process, resulting in object replication.

The provision of object replication is the main feature that distinguishes Orca from other languages like Ada83. Objects can be in two states: single copy or replicated. A method that performs on a non-replicated object is performed by simply locking/unlocking the object. For replicated objects consistency becomes a very important issue. As mentioned above, the current language implementation enforces consistency through broadcasting the objects' name, the methods, and the parameters. Each remote object then performs the operation, thus becoming consistent with the local object. An important requirement for this broadcast operation is that it must be reliable and the events should be totally-ordered.

For systems that do not enforce a reliable and totally-ordered broadcast protocol, each object will have a *primary copy* which is responsible for updating all replicated objects. The update of replicated objects is performed in two phases. The first phase will consist of the object sending a message to the primary copy, locking and updating it as before. It is the primary copy's role then to lock all remote copies. On the second phase the primary copy will update the replicated objects.

For both the primary-copy algorithm or the reliable-broadcast the final outcome is that the runtime system enforces a sequentially consistent view of the system.

D. HARDWARE/SOFTWARE COMBINATION - *FLASH*

FLASH's design exemplifies the current trend on multiprocessor systems architecture to integrate DSM and message passing. This is also valid for implementations of MIT's Alewife and *T and the Meiko CS-2 [KOHH94].

FLASH is a single-address-space machine consisting of a large number of processing nodes connected through a pair (request/reply) of wormhole meshes.

Differently from the DASH implementation, each processing node consists of an unique processor, local memory and a "*MAGIC*" chip which is responsible for integrating the memory controller, I/O controller, network interface, and a programmable protocol processor.

FLASH nodes communicate by sending intra- and inter-node commands, referred to as *messages*. The provision of a programmable protocol processor within the "*MAGIC*" chip allows the implementation of multiple protocols. These protocols are implemented on *protocols handlers* by defining the kind of messages that will be exchanged (the *message types*). Currently, the FLASH prototype enforces two types of protocols: the Cache-Coherence and Message Passing protocols.

The Cache-Coherence protocol is directory-based and is similar to the one used on the DASH implementation with minor modifications: the coherence unit was enlarged from 16 to 128 bytes and, for scalability reasons, it uses dynamic pointer allocation instead of a bitmask as was the case of the DASH prototype. Another distinction between the two machines is that invalidation acknowledgments are collected at the "Home" node for the FLASH implementation. For this protocol all messages are divided into requests (read, read-exclusive and invalidate requests) and replies (read and read-exclusive data replies and invalidation acknowledgments).

The Message Passing protocol defines a set of primitives for enforcing synchronization and block transfer. The latter set was designed to fulfill three requirements: provide user-level access to block transfer without sacrificing protection; achieve transfer bandwidth and latency comparable to a message-passing machine containing dedicated hardware support for this task; and operate in harmony with other key attributes of the machine including cache coherence, virtual memory, and multiprogramming [KOHH94].

IV. LAZY DATA MERGING CONSISTENCY MODEL

This chapter gives a comprehensive description of the Data Merging (DM) consistency protocol as proposed by Karp and Sarkar in [KS93] and the modifications introduced to it that resulted on Lazy Data Merging (LDM). Section A describes DM as originally proposed. Section B defines LDM and in Section C we introduce some examples that illustrate and compare both protocols.

A. THE DATA MERGING DSM PROTOCOL - AN OVERVIEW

In [KS93] a new DSM protocol is presented: ***Data Merging*** (DM). DM is based on the observation that any sharing that occurs between synchronization points during a parallel execution is *false*. For false sharing it is not necessary for caches to be consistent; only global memories need to be consistent. In particular, any concurrent updates to the same data block can be deterministically merged at global memory [KS93]. Therefore, DM, like other "relaxed" consistency protocols (e.g., Release Consistency, Entry Consistency, etc.) relies on explicit synchronization operations to enforce consistency. Data Merging also addresses the problem of false sharing by providing means for deterministically merging data blocks.

This protocol introduces a new feature; the ability to combine message passing and DSM. This characteristic is indicated for data sets which present poor locality of reference or to enforce sequential consistency, since it provides exclusive access to the data elements. Therefore, a remote thread will be able to explicitly perform remote *read/write* operations on individual data elements through the use of *"Bypass Cache"* messages.

The DM protocol involves two fundamental components: Global Memory Units (GMU) and Processing Elements (PE). The GMU can be better described as the "*Home*" node for a set of data blocks. The current proposal adopts a distributed/fixed manager approach for dividing the shared address space. Therefore, multiple GMUs are allowed, each one being held responsible for a part of the shared address space.

The Processing Elements represent the remote threads in which the actual computations are performed. Each PE has a "*local memory controller*" which is responsible for performing data requests/updates when necessary. In the original protocol updates are addressed to the corresponding GMU. This characteristic can also be observed in the DASH implementation.

B. DATA MERGING PROTOCOL: DEFINITIONS

Before proceeding on with this chapter, we need to define, for both the GMU and PE, each component element and its corresponding role.

1. Processing Element (PE)

PE is the processing unit that is replicated on multiple processors in a multiprocessor system and is composed of:

- **Local CPU**;

- **Local Memory** - The memory hierarchy level in the PE that interfaces with the global memory. If the PE itself has a multilevel memory hierarchy, then "local memory" refers to the lowest level (furthest from the CPU) contained within the PE; and

- **Local Memory Controller** - The control logic for issuing global memory requests from the PE.

2. Global Memory Unit (GMU)

GMU is the memory unit that is replicated to obtain a shared global memory that is addressable by all PEs.

- **Global Memory Module** - A piece of the shared global memory. Some storage in the global memory module is reserved for GMU state information; and

- **Global Memory Controller** - The control logic for handling global memory requests from PEs.

One implementation, suggested by Karp and Sarkar consists of multiple PE connected to a set of GMUs through an Interconnection Network. This organization is

depicted in Figure 22. An alternative design would allocate to each node both the processing element and the global memory unit. To provide a scalable design, such nodes should be organized using a Hierarchical Bus or Ring Network as is the case for the DASH, Alewife, and FLASH multiprocessor architectures.

C. THE DATA MERGING PROTOCOL

Similar to existing software implementations (i.e., Munin, CarlOS, and Treadmarks), this mechanism addresses the problem of false sharing by allowing multiple writers to the same data block. The distinction is that there is no required protocol-type annotation for shared variables as is the case of Munin in which data variables that allow multiple writers should be annotated as "*write-shared*".

For data that is accessed by multiple processing elements, it is assumed that a delayed memory consistency model and the synchronization mechanisms that it requires are implemented. This mechanism also provides direct accesses to the Global Memory through *"Bypass Cache"* messages.

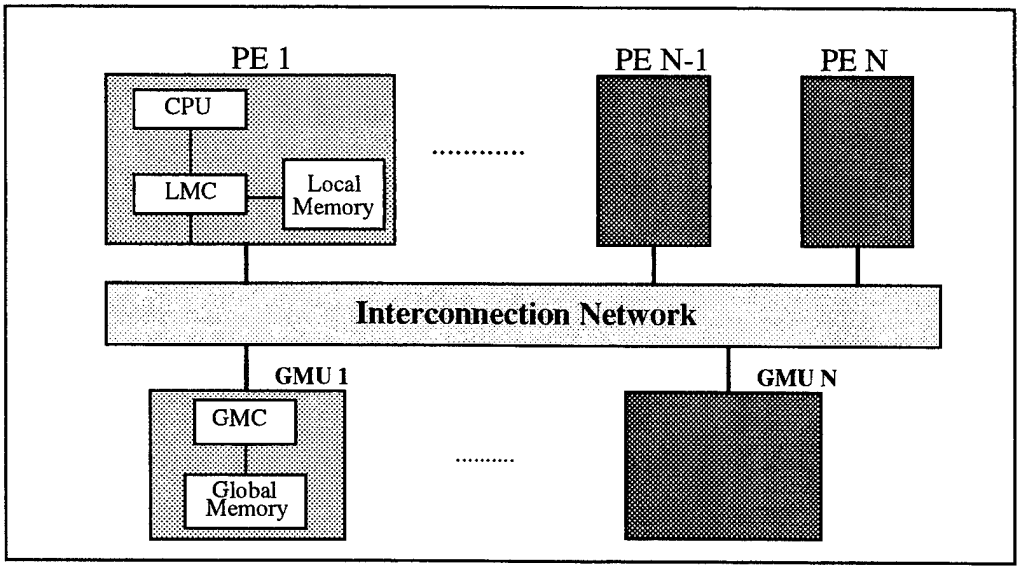


Figure 22: Data Merging Components.

The actions performed by each system component can be described as follows:

1. Processing Element

• **Request cache block:** request a copy of a data block from the GMU that owns it;

• **Flush Data:** when replacing a dirty cache block, send its contents back to the GMU that owns the original data block;

• **Report Replacement:** when replacing a clean cache block, report its replacement to the GMU that owns the original data block;

• **Bypass-read data element:** the CPU reads a data element from global memory without storing a copy in the PE's local memory;

• **Bypass-write data element:** the CPU stores a data element in global memory without storing a copy in the PE's local memory.

Bypass read and Bypass Write messages may be used to enforce sequential consistency, rather than delayed consistency, on selected accesses to global memory. They are also more efficient for reads/writes accesses that have neither temporal nor spatial locality. The state transition diagram for each block within the PE node is illustrated in Figure 23.

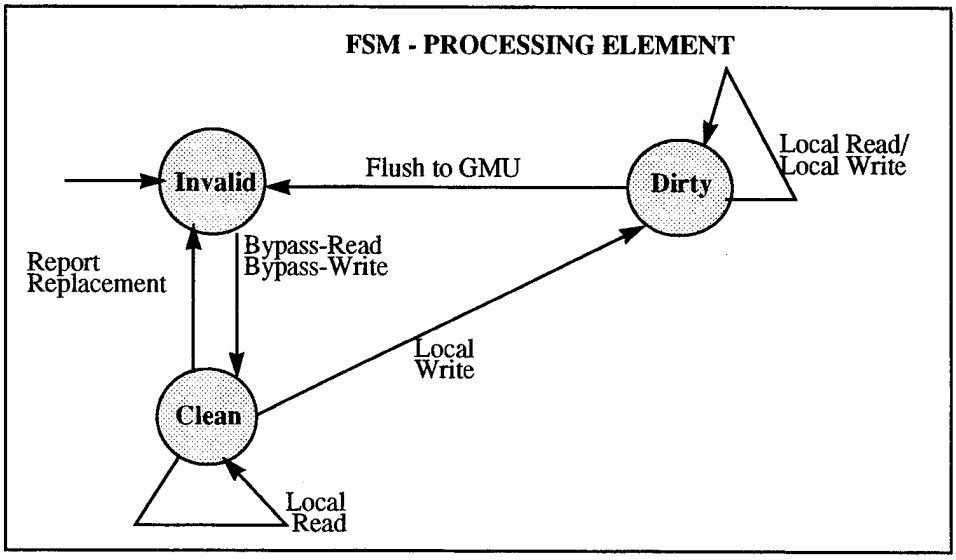


Figure 23: Processing Element Finite State Machine.

The cache block can be made invalid either by normal cache management policies or by an invalidate signal sent by the GMU. If the block is dirty it is flushed; if clean, its replacement is reported to the GM, but no data is moved.

2. Global Memory Unit Actions

The GMU keeps storage for the global data blocks and for monitoring the state of each individual block. For this purpose it has two major data structures: a dynamically-updatable "*Suspend Queue*" capable of holding up to one entry per process in the multiprocessor system and a "*bitmask*" that keeps track of the state of each individual element within a data block. Each GMU also maintain the following state variables to monitor the state of each individual block. The state transition diagram for each block within the GMU node is illustrated in Figure 24.

- **Counter (C):** Identifies the number of PEs that currently have a copy of the data block in their local memories.
- **Suspend Bit (S):** Indicates whether or not a process should be suspended when attempting to access the data block.
- **Bitmask:** The number of bits corresponding to the number of elements in the data block. Its purpose is to identify dirty elements in a data block.

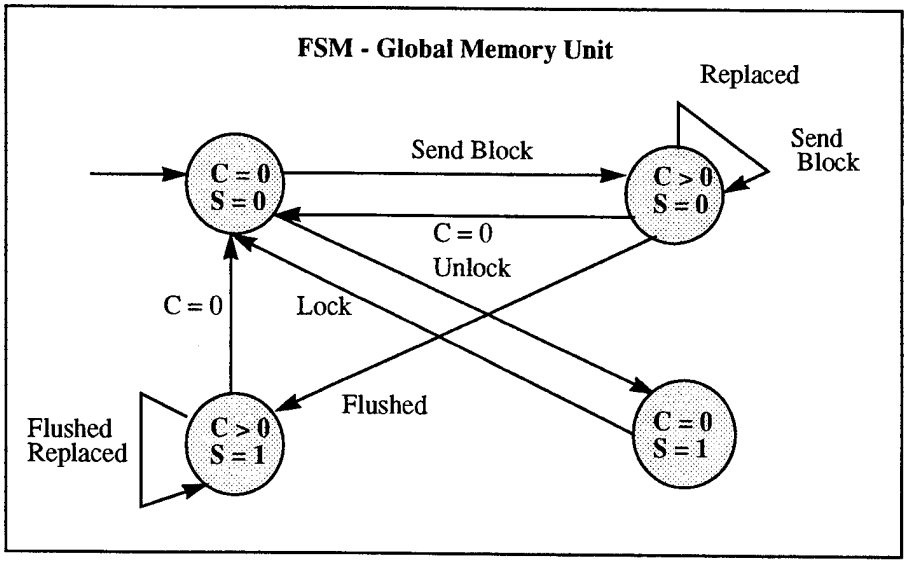


Figure 24: Global Memory Unit Finite State Machine.

3. Actions Performed by the GMU in Response to the Processing Element Requests

a. Request Cache Block

- **The suspend bit (S-bit) = 0:** The GMC increments the counter and sends a copy to the PE;
- **The suspend bit (S-bit) > 0:** The GMC inserts the data block request, requesting processor ID and current time stamp into the GMU's suspend queue.

b. Flush Data

In this case, the PE replaces a cache block in its local memory and sends the contents of the old cache block (C) to be merged with data block D in the GMU. There are two cases of interest for the state information associated with data block D:

- **Counter = 1 and S-bit = 0:** The GMC stores cache block C into data block D and resets the counter to zero.
- **Otherwise:** In this case the GMC uses the bitmask to merge selected words from cache block C into data Block D, by comparison on a word-by-word basis of the two blocks. If the words are different, set the corresponding bit of the bit mask to one and set the suspend bit to 1. If the Counter becomes zero, then perform a bitmask-reinitialize operation.

c. Report Replacement

The PE sends a notification of the replacement of a data block without actually sending any data. For this case all is needed is to:

- Decrement the Counter.
- If the Counter becomes zero, then perform a bitmask-reinitialize operation.

d. Bypass-Read Data Element

The GMU sends the data element to the requesting processor. No state information needs to be checked or modified.

e. Bypass-Write Data Element

When a PE writes a data element by bypassing local memory (cache) the GMU updates the corresponding global memory location. No state information needs to be checked or modified.

f. Lock

In this case the processing element has requested that the block be locked.

- If the S-bit is set, insert the request into the GMU's suspend queue along with the requesting process *"id"* and a special flag in place of the time stamp.
- If the suspend bit is not set, set the bit, and return its old value.

g. Unlock

Set the suspend bit to zero and perform a bitmask reinitialize operation.

h. Test and Set Lock

Set the suspend bit to one and return the previous value of the suspend bit to the requesting process.

i. GMU Internal Actions

(1) **Initialization:** For each block of global memory a *NULL* bitmask pointer, a Counter, and Suspend Bit should be initialized.

(2) **Receive a dirty page:** The counter should be decremented whenever a dirty page is received. The Bitmask pointer should be initialized with all elements set to zero. This represents the occurrence of multiple writers, but it solves the problem of false sharing.

(3) **Bitmask reinitialize:** When the Counter becomes 0, deallocate the bitmask and Suspend Bit. Check if this event affects any processor in the suspend queue.

(4) **Scan the Suspend Queue:** To avoid deadlocks the GMU controller should periodically scan the suspend queue. If any data block request is older than a threshold age, the GMU should perform a timeout procedure by broadcasting an invalidate message, forcing the cached data block to be flushed. The existing copies of the data block are then merged, becoming consistent. The requesting process is then awakened by the release of the data block.

D. LDM RATIONALE

The suggested extensions to the DM protocol led us to define a "*lazy*" version which we name "Lazy Data Merging". For this approach all data objects are considered as "write-shared". Therefore, program correctness will rely on the adequate use of synchronization variables by the application programmer.

Our approach introduces some enhancements to minimize the communication/ synchronization overhead generally encountered by software runtime libraries. The implementation details for this protocol are discussed in Chapter V.

As with DM we adopt a "distributed/fixed" policy for dividing the shared address space (static data partition as opposed to the adaptive partition scheme adopted by Munin). However, our protocol differs from DM by insisting that each node will have both PE and GMU threads sharing the same address space. Therefore, individual nodes are assigned as "home" for a specific set of data blocks.

Our goals for the LDM protocol are both the reduction of the average message size *and* the number of messages. Message size reduction is achieved by forwarding "*diffs*" at the time of a release or a flush operation. We reduce the *average* number of messages through the use of a distributed locking scheme [FB88] and the Hybrid Coherence protocol [DKCZ93]. This combination will decrease the number of messages that will be dispatched to a single node at the time of a lock release operation when compared to an invalidate protocol. Also, the main advantage of the distributed locking scheme is to reduce the contention when compared to the centralized approach.

The performance gains that can achieved through the use of "diffs" for pages that are dirty are not very well defined. The result will be highly dependent on the type of network, number of participant nodes, granularity of the data coherence unit, etc. Our proposition is to use "diffs" for all data blocks that are dirty. Our belief is that when the ratio "processing power/ network bandwidth" is high it is worthwhile in creating diffs.

The next sections will give an overview of this memory consistency protocol and in Chapter V we describe the mechanisms for implementing the modules specified on Figure 25.

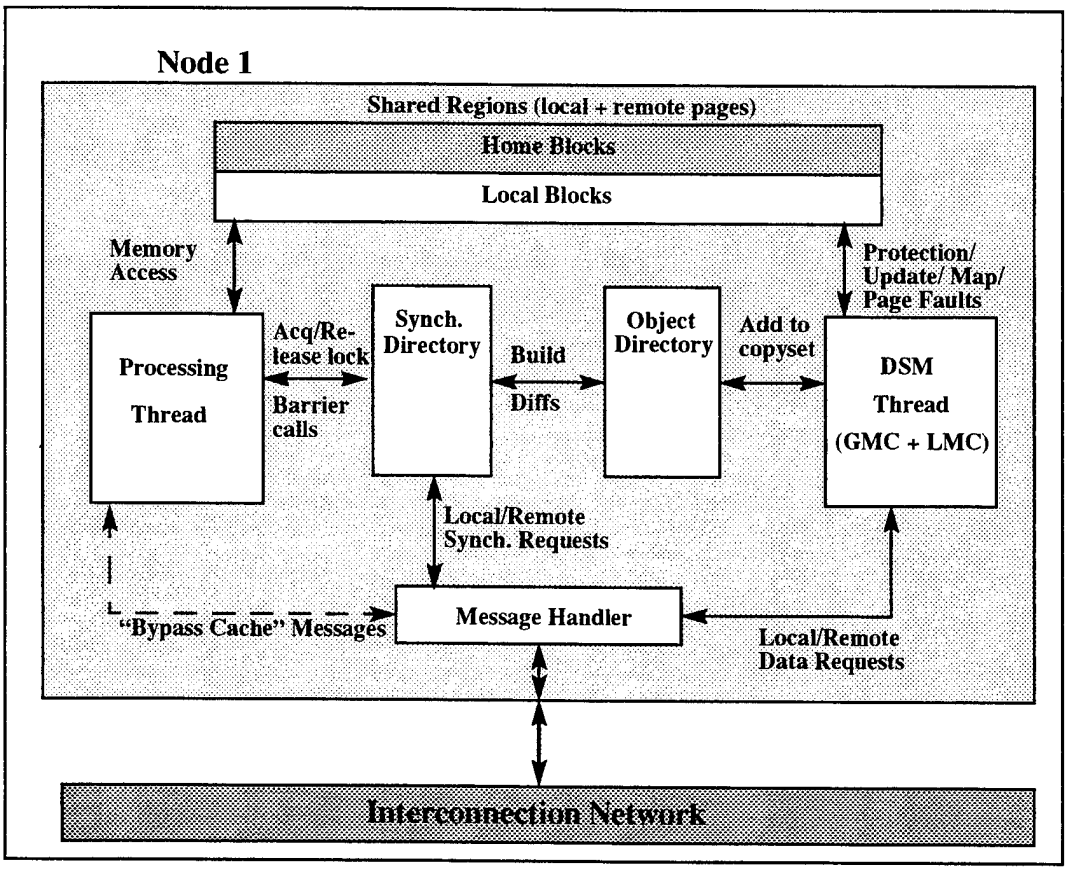


Figure 25: Lazy Data Merging Runtime Environment.

E. EXTENSIONS TO THE DATA MERGING PROTOCOL

To optimize performance for a software implementation we introduce the following extensions:

- In Lazy Data Merging the shared address space is structured as a set of shared data-objects, while in DM the address space is flat.
- In order to reduce the number of invalidate messages each Home node maintains a Directory with the nodes that currently cache the given page, instead of broadcasting invalidate messages to all PEs.
- We suggest using a distributed locking scheme [FB88] by employing a hybrid (invalidate/update) coherence protocol [DKCZ93]. The semantics for the acquirer are similar to the Lazy Release Consistency model.
- The GMU is co-located with the PE on the same node. Both threads share the same address space.
- We assume two types of locks: Read and Write. This approach is similar to the one introduced by Midway. Lock ownership will only be modified if a lock is acquired for writing.
- Updates will be encoded by using diffs to the original pages. The purpose is to reduce the network load, by reducing the size of messages. This approach imposes some overhead to compute diffs of each page and also requires extra storage for data blocks that are dirty, so that we can capture all changes introduced to the data block, but is in large compensated for relatively slow networks. The use of "diffs" is also an imposition of the protocol for allowing the correct propagation of modifications on shared data to the last acquirer.

F. THE LAZY DATA MERGING PROTOCOL

As we have already described, the shared address space will be evenly divided across the set of nodes that are part of the system. Therefore, each node will be assigned as "Home" node for certain block segments. Each "Home" node will, in turn, be responsible for performing the data merging operations whenever a block it manages is flushed from one of the remote caches.

This new approach addresses the problem of false sharing as aggressively as "Data Merging" does: we move some burden to the programmer, by requiring that all shared data blocks that are concurrently accessible by multiple nodes should be protected by the appropriate synchronization mechanism.

Besides managing "Home" data blocks, the DSM thread handles data block misses (detected by catching "*SIGSEGV*" signals) and performs the data requests to the appropriate block owner. Other roles are to create diffs for dirty pages (by the time of a lock release, a global barrier call or when an invalidate message is received), mark the page as dirty (page protection is set to *PROT_WRITE*) and update the set of write-notices.

1. Protocol Notation

Before we can describe the protocol itself we need to define the notation that is adopted, which we believe is appropriate for explaining the LDM protocol by providing means for representing the messages interchanged and its arguments and the actions undertaken at both ends (sender/receiver).

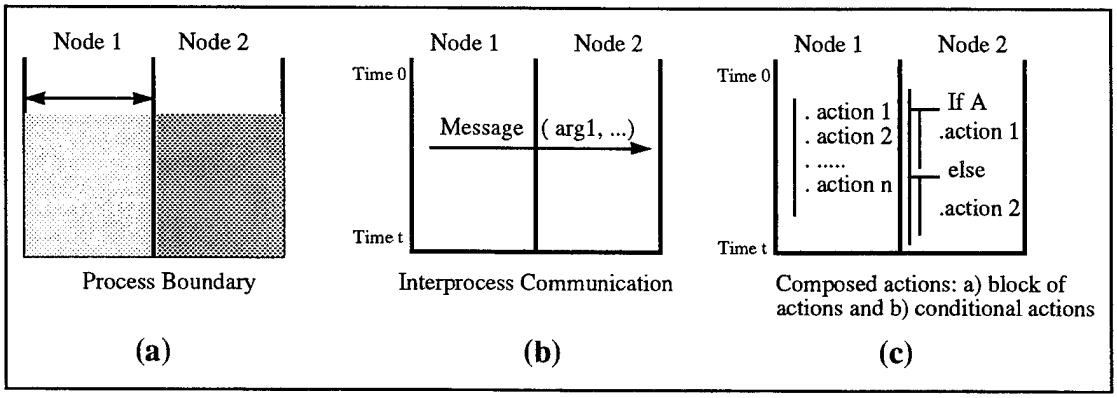


Figure 26: Notation for description of LDM.

Each process context is inserted within one frame limited by vertical lines. This representation is depicted in **F**igure 26a. There are five basic types of processes:

- *requesting node*: the node which issues either a request for a page, an acquire lock, or a barrier call;
- *home node*: the node that is responsible for the block management;
- *copyset nodes*: the group of nodes that currently cache a copy of the same page;
- *lock owner*: the current probable owner (see Chapter III for the details).

• *write-notice node:* the node that has last modified a write for that particular data block.

In general terms a message is described by an arrow that crosses the process(es) boundary (**F**igure 26b). The actions may be atomic or composed actions. Composite actions are involved by a line that extends until the end of the block (**F**igure 26c). Conditional actions are actions that depend on the internal state of the processing node (**F**igure 26c).

All of the above events are described within the domain of time, with the initial event being represented at time "*0*" and the last one at instant "*t*".

2. Description of LDM actions

This section describes the set of actions that are taken by each node in presence of the following events: page faults, bypass-cache requests, and synchronization events.

a. Page Fault

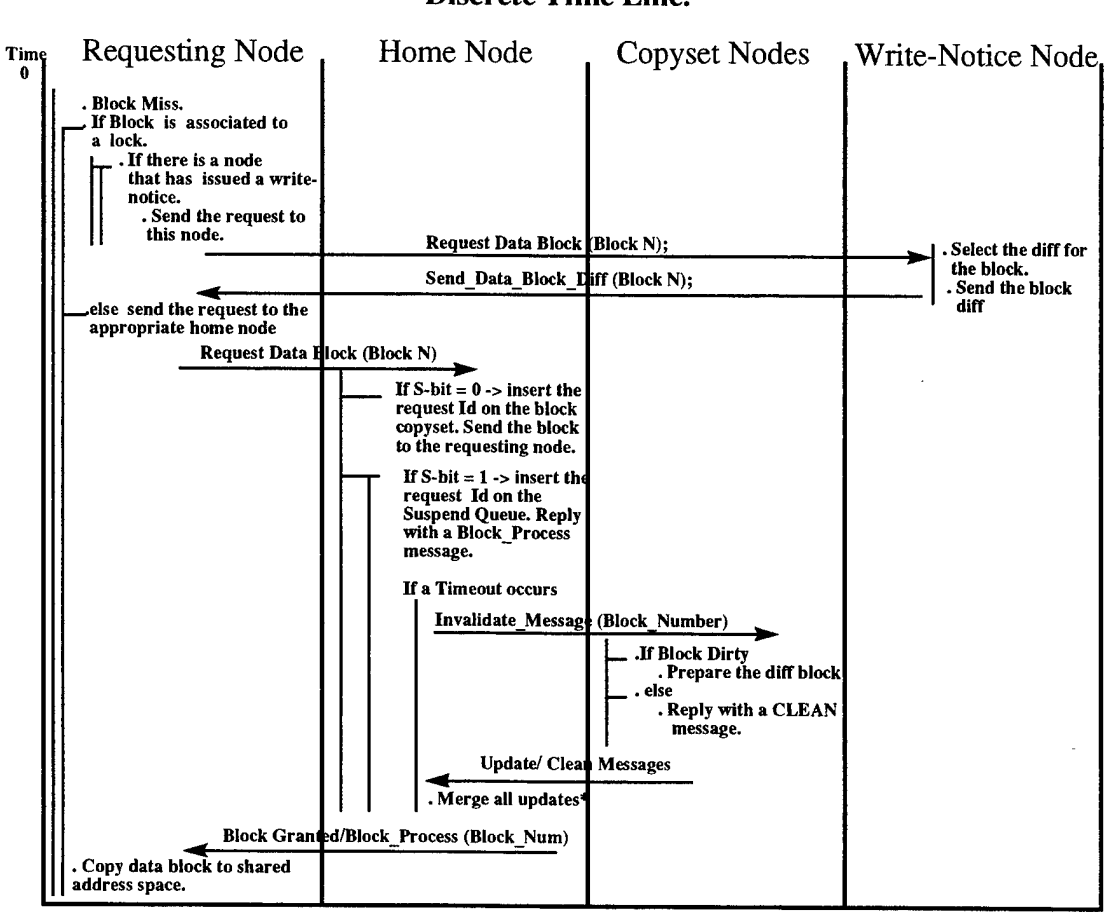
The page fault handler[1] should, in turn, convert the faulty address into the page number and "hash" it into the appropriate Home node. Our design provides means for establishing the policy for static division of data among the nodes (i.e., block partitioning, cyclic stripe partitioning, etc.). The default policy will consist of dividing the shared address space into blocks of four contiguous pages. Once the home node has been determined a request block message is issued.

Page requests are handled in the same way as defined on the Data Merging protocol. Upon receipt of a Request_Block type message, the Home node will verify if the S-bit is set. If so, the requesting node should be inserted on the Suspend Queue. This request will remain on the Suspend Queue until S-bit = 0 or a timeout occurs.

If the S-Bit is not set then the Home node includes the requester ID on the block copyset and forwards the requested block. These actions are described in **F**igure 27.

---

1. A *SIGSEGV* signal handler.



Note: If the block is missing by the first time, the request should be forwarded to the appropriate home node, before a diff request can be issued to the write-notice node. This FLAG should be reset if the data block is also associated with a barrier object. The merge of Updates will be better defined during the Barrier Call definition.

Figure 27: Performing a Data Block request.

b. Bypass-Cache Messages

(1) *Bypass Read Messages:* as mentioned before, our protocol provides means for performing remote reads. Upon receipt of such a message, the home node will request update messages from all blocks that currently cache the requested block and merge all these nodes. After this it will forward the requested block. By updating rather

than invalidating remote copies, we allow multiple processes to access the same data block for read operations. These messages should be used with data that presents poor locality of reference or to achieve a sequentially consistent program. These actions are shown in Figure 28.

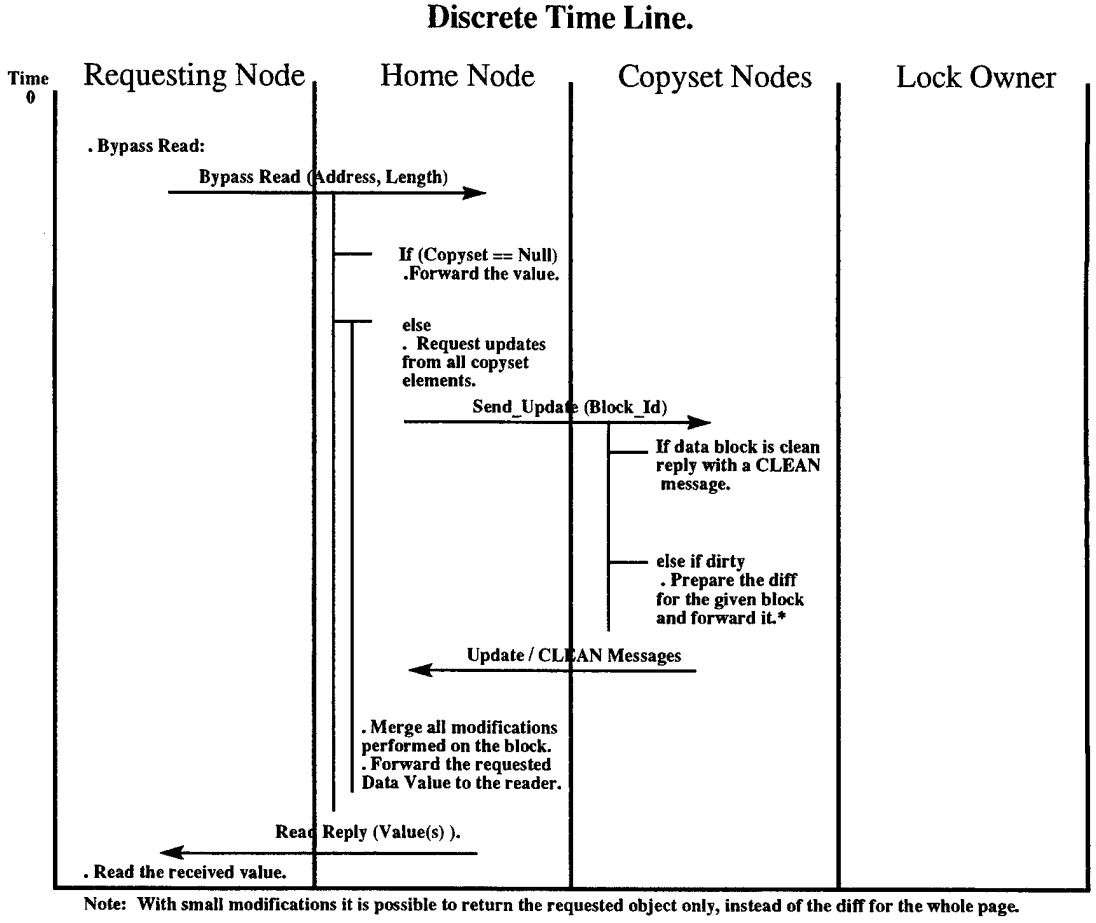
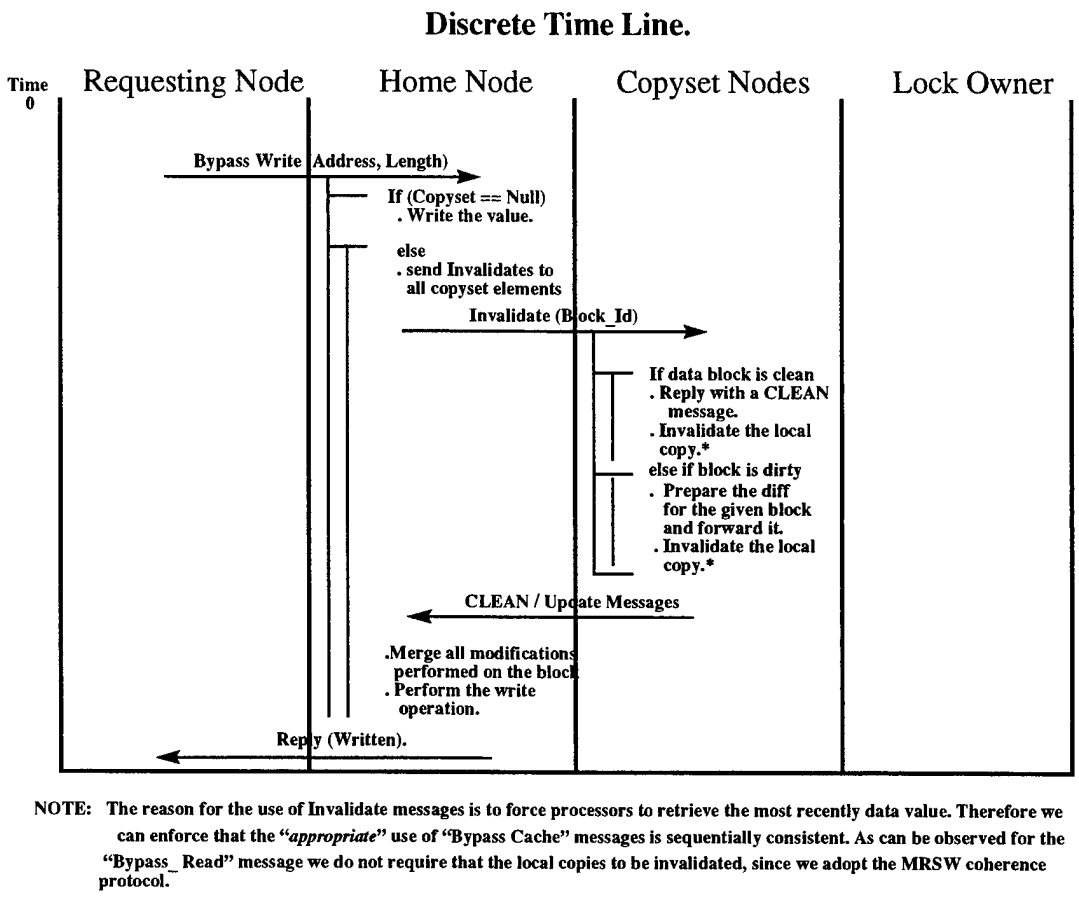


Figure 28: Bypass-Read Messages.

(2) *Bypass-Write Messages:* Once a bypass-write message is received the home node should "*invalidate*" all remote copies of the given block. Then the Home node should merge all received updates and perform the write operation. These actions are described in Figure 29.



NOTE: The reason for the use of Invalidate messages is to force processors to retrieve the most recently data value. Therefore we can enforce that the "*appropriate*" use of "Bypass Cache" messages is sequentially consistent. As can be observed for the "Bypass_Read" message we do not require that the local copies to be invalidated, since we adopt the MRSW coherence protocol.

Figure 29: Bypass-write message.

c. Synchronization Operations

We rely on synchronization operations to enforce consistency among multiple threads. Therefore, any data access that may result on data race conditions requires the use of explicit synchronization operations. For this purpose we provide two basic synchronization mechanisms: *Locks* and *Barriers*. The actions for each mechanism are described in the following figures. We use diffs to minimize the effect of network latency (by reducing the message size) and for correctness [K95]. Diffs are obtained by creating a copy of the original block (a *twin copy*) before a node tries to write into it. At the time of a

release/barrier call operation the differences between the twin copy and the original block are inserted on the diff. This action is summarized in Figure 30.

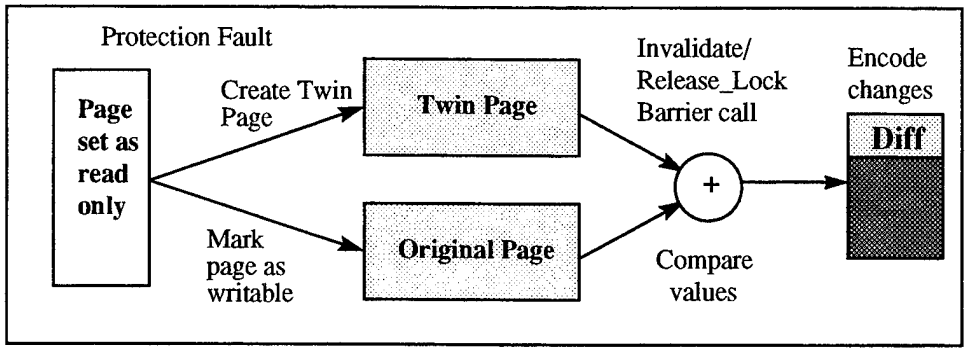


Figure 30: Diffs creation process.

d. Partial Ordering Definitions

In order to define a partial order between multiple intervals we need to enforce the requirements established in [AH90] for the relation *happens-before*. The requirements for this relation can be described as follows:

- If a_1 and a_2 are accesses on the same processor, and a_1 occurs before a_2 in program order, then a_1 *happens-before* a_2.
- If a_1 is a release on processor p_1, and a_2 is an acquire on the same memory location on processor p_2, and a_2 returns the value written by a_1, then a_1 *happens-before* a_2.
- If a_1 *happens-before* a_2 and a_2 *happens-before* a_3, then a_1 *happens-before* a_3. [KCZ92].

Each node maps to an index in the Vector Timestamp, therefore, the logical clock of $node_i$ maps into index *"i"* of the corresponding Vector Timestamp. The *happens-before* relation can be enforced through the following criteria:

- At an acquire_lock the requesting node sends the vector timestamp of the last release. The lock owner, in turn, will send all write-notices that were performed after the received Vector Timestamp or that are

concurrent to the received vector timestamp. The lock owner should also release the "diffs" of the shared objects that it might have modified.

• Once the lock is acquired the new lock owner will update the blocks for which it received the diffs and invalidate the ones for which write-notices were received, but no "diffs".

• The lock owner will update its Vector Timestamp, by incrementing its logical clock and replacing it on the received Timestamp.

The rules for updating the Vector Timestamps are as follows:

• **Rule 1:** Clock C_i is incremented between any two successive events in process P_i, such that $C_i[i] = C_i[i] + 1$;

• **Rule 2:** If event "*a*" is the sending of a message "*m*" by process P_i, then "*m*" has a Vector Timestamp $VC = C_i(a)$ (using rule 1). When process P_j receives the message it updates its Vector Clock to:

$$\forall k,\ C_j = max(C_j[k], VT[k])$$

• where $C_i(a)$ corresponds to the Vector Timestamp to any event *a* at process *i*. For our approach we consider as conspicuous events only the operations that result on the issue of "diffs" (release locks, barrier calls, and invalidates).

The example of **F**igure 31 clarifies this issue. In this example three processes, P_1, P_2, and P_3 are requesting the same write-lock. At each acquire that is granted the lock owner will update its own VC and forward the lock with the corresponding write-notices and updates that are larger than the received VC. The acquirer should, in turn, update its own logical clock.

Based on the above rules we can state that "a *happen-before-1* b" if and only if VC(a) < VC(b), otherwise they are said to be concurrent. The definitions below describe when VC(a) is less than VC (b):

• Not equal:

$$VC_a \neq VC_b \Leftrightarrow \exists i, \left(VC_a[i] \neq VC_b[i] \right)$$

• Less than or equal:

$$VC_a \leq VC_b \Leftrightarrow \forall i, \left(VC_a[i] \leq VC_b[i] \right)$$

• Less than:

$$VC_a < VC_b \Leftrightarrow \left(VC_a \leq VC_b\right) \wedge \left(VC_a \neq VC_b\right)$$

• Concurrent events:

$$VC_a \parallel VC_b \Leftrightarrow \neg\left(VC_a < VC_b\right) \wedge \neg\left(VC_b < VC_a\right)$$

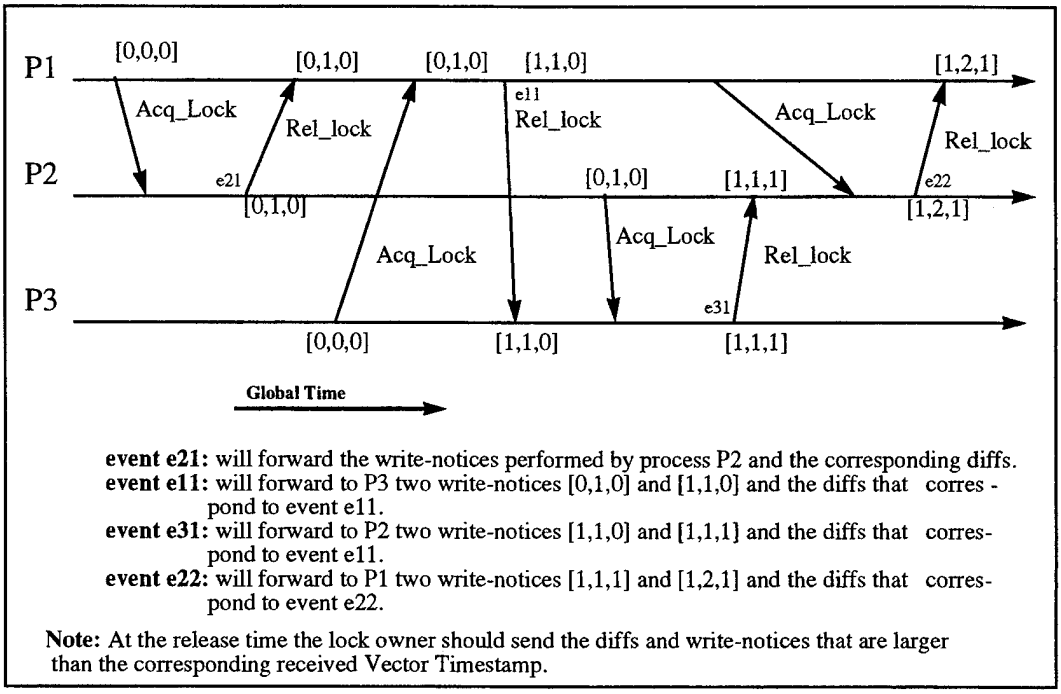


Figure 31: Vector Clock implementation.

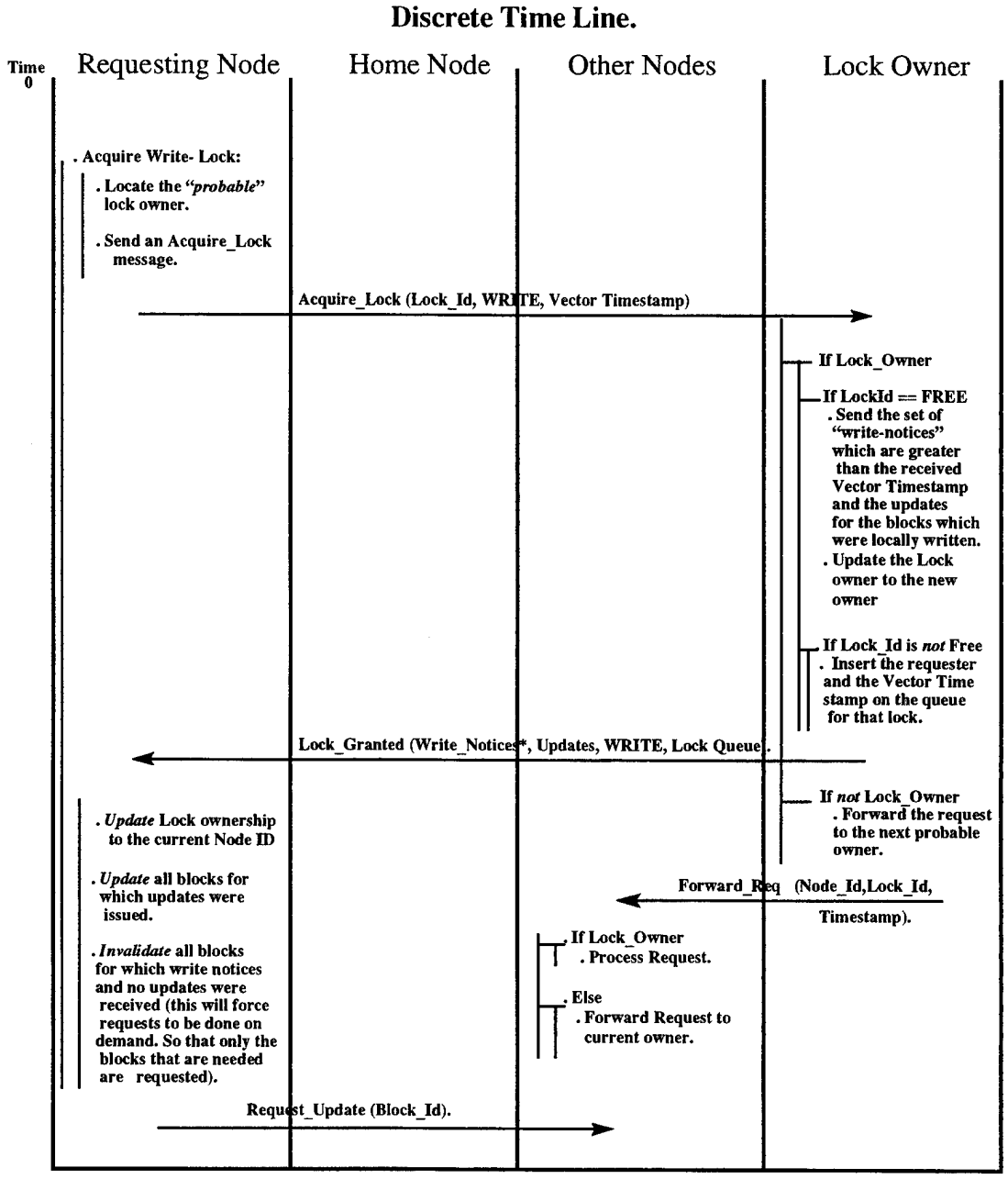
e. Read and Write Locks

As mentioned before, we introduce two types of locks: *Read* and *Write* locks. "*Lock ownership*" will only change when a lock is acquired for "*writing*". When a lock is acquired for reading, the current lock owner should introduce the requester "id" on the lock copyset and forward the write-notices and updates (**Figure** 32). The lock acquirer should, in turn, perform the updates to the local pages and invalidate all pages for which write-notices and no diffs were received (**Figure** 32). At the time of a lock release the reader

should report to the lock owner (Figure 33). The lock owner should, in turn, upon receipt of the release message remove the node from the list of readers. A lock should be considered *"WRITE-AVAILABLE"* if and only if the list of readers is *empty*. A lock should be considered *"READ-AVAILABLE"* if and only if the lock owner is not currently holding it. By adopting this policy we allow multiple readers to concurrently access a critical section, but a unique writer can be within a critical section at a time.

When a write-lock is acquired (Figure 32), the lock owner will forward to the new acquirer the set of write-notices, the diffs and the queue of processes waiting for the lock (if any). Once the lock acquired message arrives, the new owner will execute the same actions described for a read-lock. When a release on a write-lock is performed the new acquirer should check the lock queue. If there is any process waiting for the lock it should create the diffs for all pages marked as dirty and forward them to the requester. As mentioned before, if the lock is being acquired for write, the ownership is altered, otherwise the process maintains the lock ownership (read locks) (Figure 33).

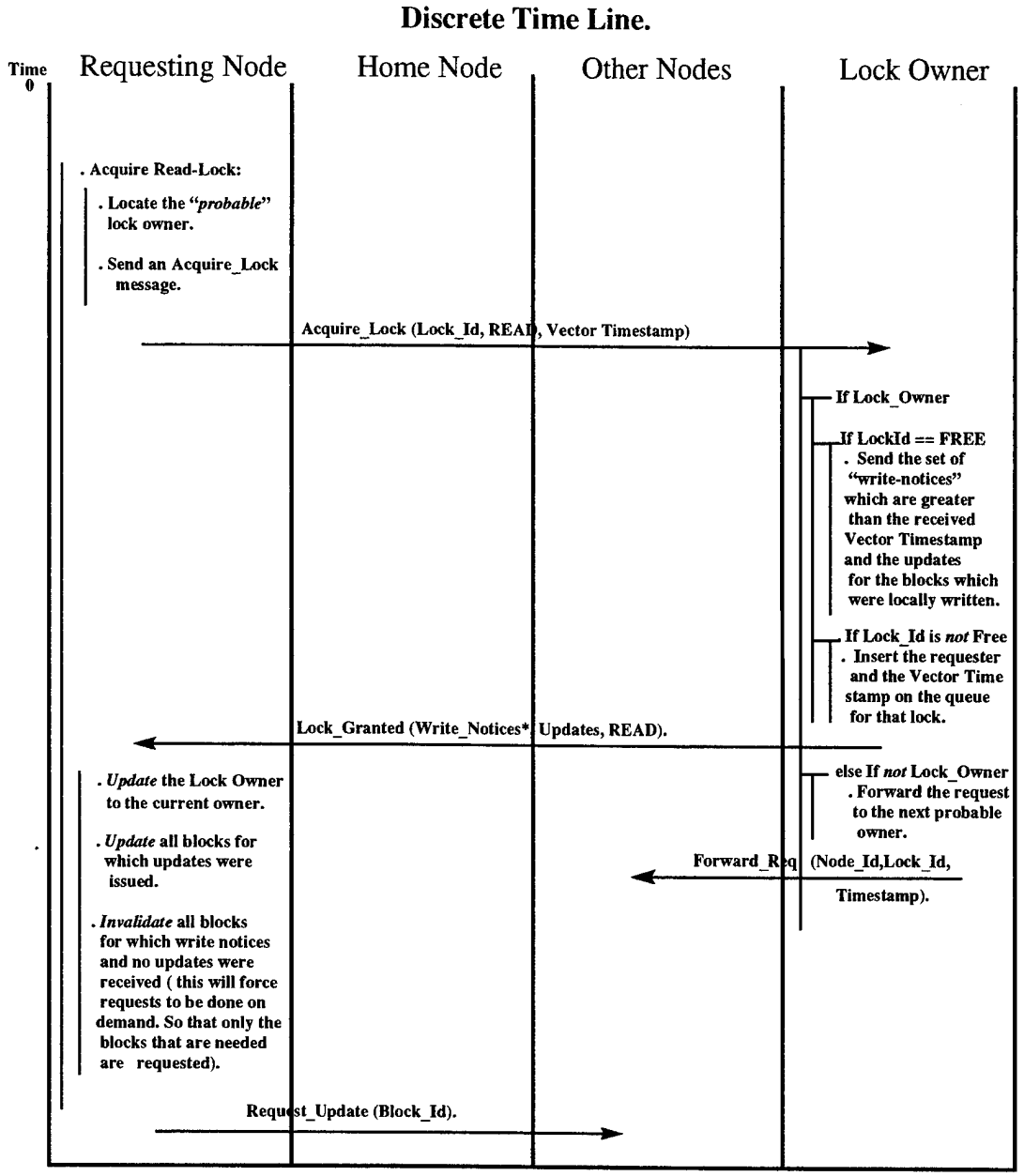
To enforce consistency, the node that is acquiring a lock must be aware of all modifications introduced on the data protected by the synchronization variable. We solve this problem by adopting the same approach as on Treadmarks [ACDB94] by forwarding the *write-notices* performed on the blocks. A write-notice is an indication that a page has been modified during a particular interval, without specifying the actual modifications. But, an acquiring node does not need to be aware of all write notices. An *acquire* operation should only receive the set of write notices that were performed by other nodes after its last *release* operation. This requirement introduces the notion of enforcing a logical time so that we can achieve a partial ordering of events. This issue is addressed through the use of vector timestamps as proposed on [KCZ92].



Note: 1- The *Write-Notices* consist of the triple (Block_Id, Timestamp, Last_Writer_Id). If two processes modify the same block their changes should be merged into a single update block and the write_notice triple should be updated to both the ID and Timestamp of the last modifier. If the size of the update block gets larger than the block itself we should replace the Update Block by the block with a special annotation.

2- For a pure software solution, invalidations can be handled by modifying the protection of the given data block. During the invalidation process the block owner should be replaced by the corresponding last writer.

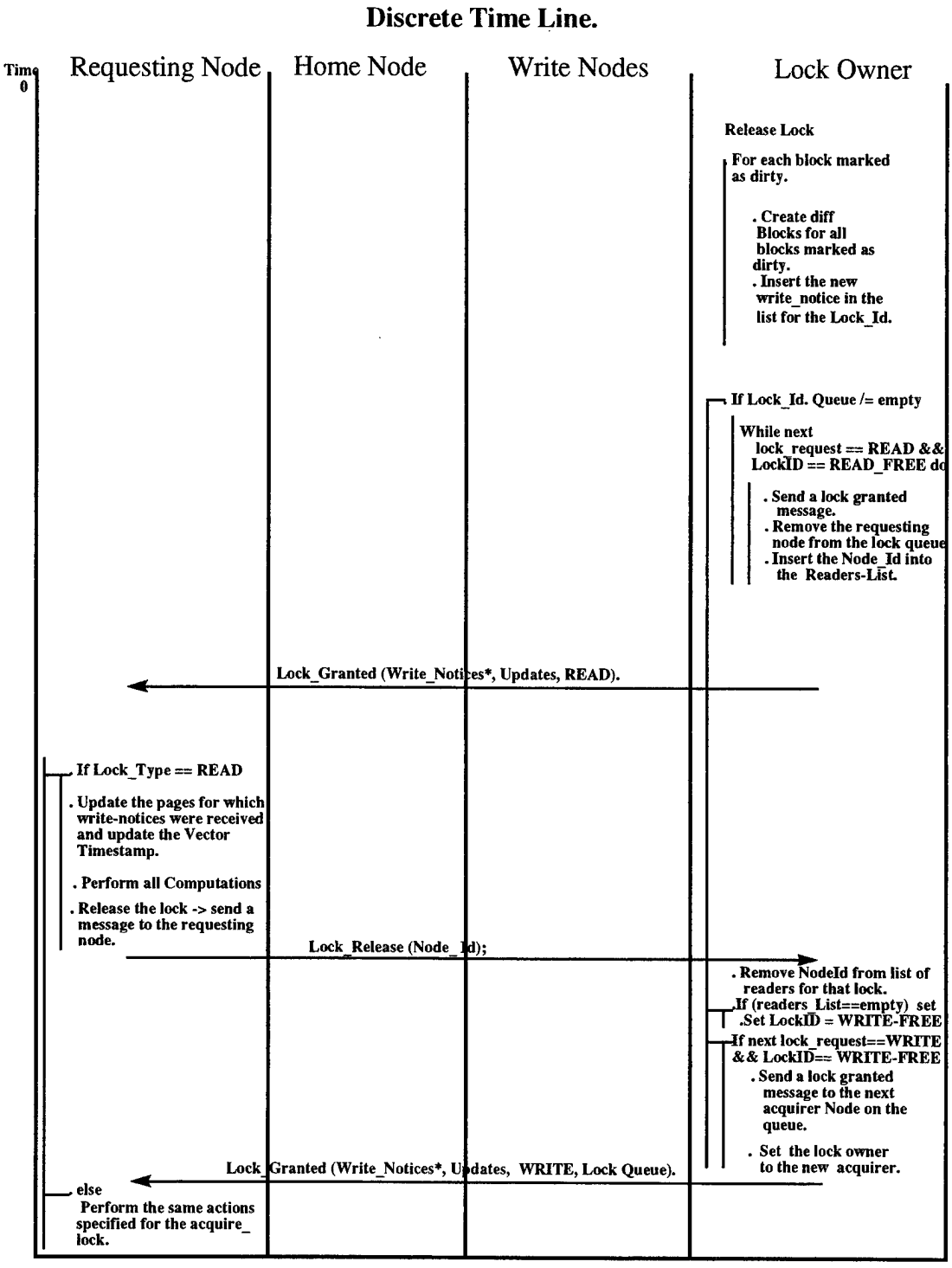
Figure 32: Synchronization event: acquiring a Write-Lock.



Note: 1- The *Write-Notices* consist of the triple (Block_Id, Timestamp, Last_Writer_Id). If two processes modify the same block their changes should be merged into a single update block and the write_notice triple should be modified to both the ID and Timestamp of the last modifier. If the size of the update block gets larger than the block itself we should replace the Update Block by the block with a special annotation.

2- For a pure software solution, invalidations can be handled by modifying the protection of the given data block. During the invalidation process the block owner should be replaced by the corresponding last writer.

Figure 33: Synchronization event: acquiring a Read-Lock.



Note: The update for the write-notice will consist of updating the Timestamp with the current one and by replacing the last writer by the current Node_Id. Therefore, the last process that writes into a data block will hold the write-notice.

Figure 34: Synchronization event: performing a lock release.

f. Barrier Call

Barrier synchronization primitives fulfill two distinct purposes: synchronization and consistency. Its use allows us to evict the dirty blocks to their corresponding *Home* nodes, forcing the global memory to enter in a consistent state. For this purpose we allow the user to specify the data set that should be flushed to its corresponding home node. If the barrier's purpose is to perform global data coherence, all dirty blocks should be flushed to their corresponding home nodes. We name this type of barrier primitive as a "*Converge Barrier*" and it should be explicitly used whenever "*global memory*" must enter in a consistent state. If the barrier is a "*local barrier*" only the data specified at its creation should be flushed.

In our protocol the barrier primitives are designed adopting a centralized approach. A barrier call should be executed in two steps. On the first step each node will send their updates to the corresponding home nodes and invalidate its local copies. Upon completion of this initial step the process should perform a call to the designated barrier manager. The barrier manager, in turn, monitors the number of processes that have executed a barrier call. When this number is equivalent to the number of registered processes the manager multicasts a "*CROSS_BARRIER*" message. Upon reception of such message each process should wake-up.

When the home node receives the updates it is possible that two modifications to the same location are received (only if the data is associated to a lock). For this purpose the relations *happens-before* should hold, otherwise, concurrent accesses are performed to the same memory location, resulting on a nondeterministic result. The rules for merging *diffs* can be summarized as follows:

- If the vector timestamp of the received diff is larger than the original one the diffs are applied to the block even if the corresponding bits on the bitmask are already set.
- If the vector timestamp of the received diff is smaller than the initial one then the received diff is discarded.
- If the vector timestamp is concurrent with the Vector Timestamps

already used, then the diff is applied to the data block if and only if no two diffs modify the same location, otherwise an error condition shall be raised. This feature will not prevent concurrent writes to the same memory location, but will detect them during runtime.

Figure 35 illustrates a barrier call execution.

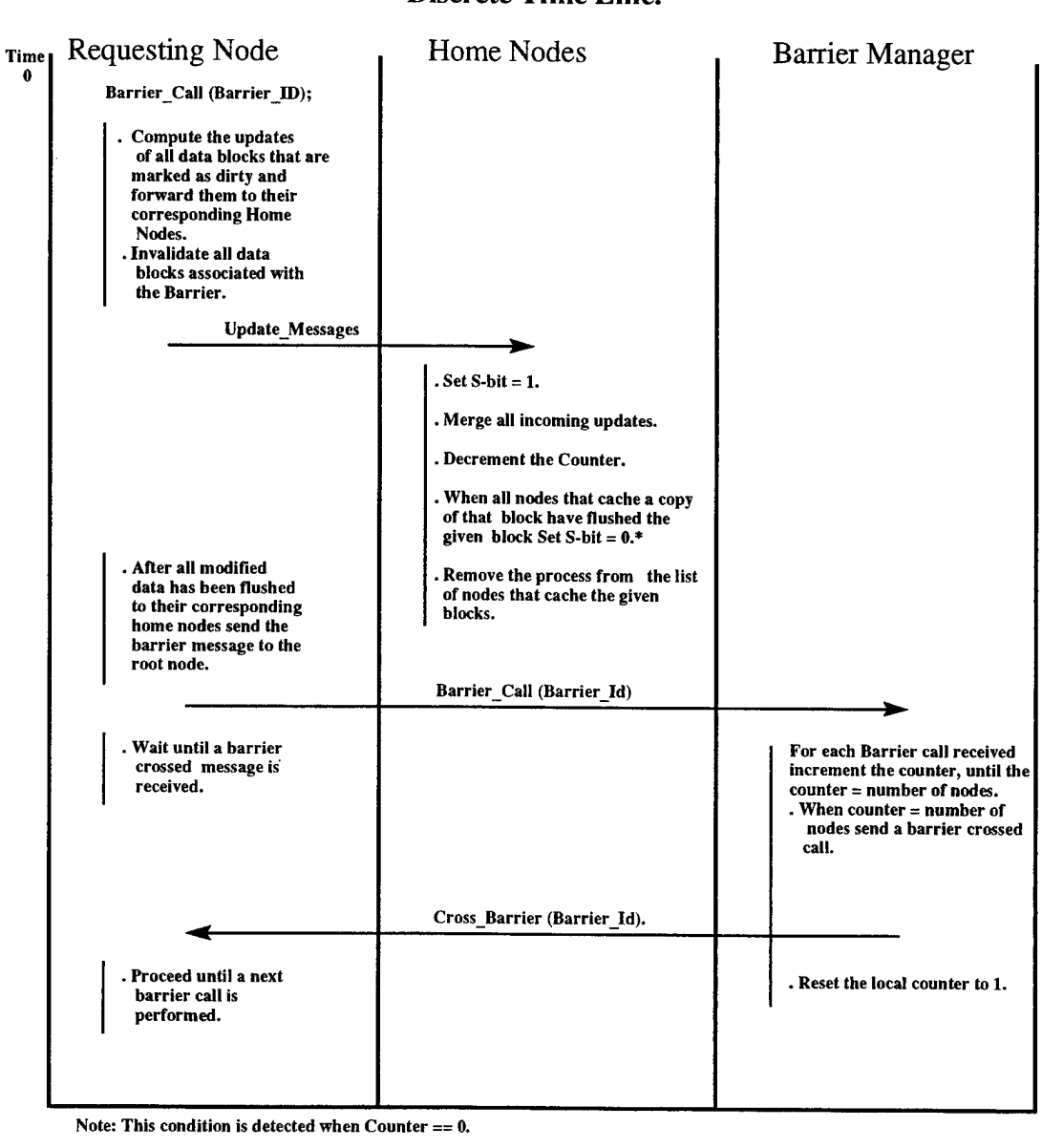


Figure 35: Synchronization event: barrier call.

G. PUTTING IT ALL TOGETHER

We use a couple of examples to clarify the differences between Data Merging and Lazy Data Merging. The first set of examples describe a more synchronization intensive problem, in which we describe the actions that are adopted by both DM and LDM protocols and provide a qualitative analysis of the communication costs involved.

For uniformity, the data distribution described in Figure 36 is adopted by all examples.

1. Distributed Data Base Problem

Assume that each node needs to lock the data base record before accessing its fields. The problem can be summarized as follows:

```
While not done loop
    acquire lock (lock_id)
    perform_modifications.
    release_lock (lock_id)
end loop.
```

For this type of problem our approach should perform better than Data Merging since each release operation will require that all data associated with the lock to be flushed to the corresponding GMUs.

In our approach the data will be released only at the time of an acquire and only to the node which is requesting the lock. The releasing node should, in turn, send all modifications (write-notices) that have happened after the vector timestamp received from the requester. As before, we also optimize by sending only updates (diffs), instead of the entire data block. The use of a hybrid coherence protocol will reduce the amount of communication since the new acquirer will be able to update all blocks for which the last releaser has introduced modifications.

To better describe the actions that are taken by both protocols under the presence of synchronization operations we use the code example below.

```
Processor 1:                    Processor 2:
  Acq_Lock (LockID)               Acq_Lock (LockID)
    A = 5;                          A = 10;
Release_Lock (LockID);          Release_Lock (LockID);
```

The next two subsections describe the interactions between the multiple DSM system components for the example described above, under both the DM and LDM protocols.

a. Data Merging

An acquire lock for the DM protocol would require a larger number of messages for performing a lock operation. Locking each individual data block requires that all remote copies should be invalidated before the lock could be granted. For the original approach an invalidate message should be multicasted to the entire data block copyset.

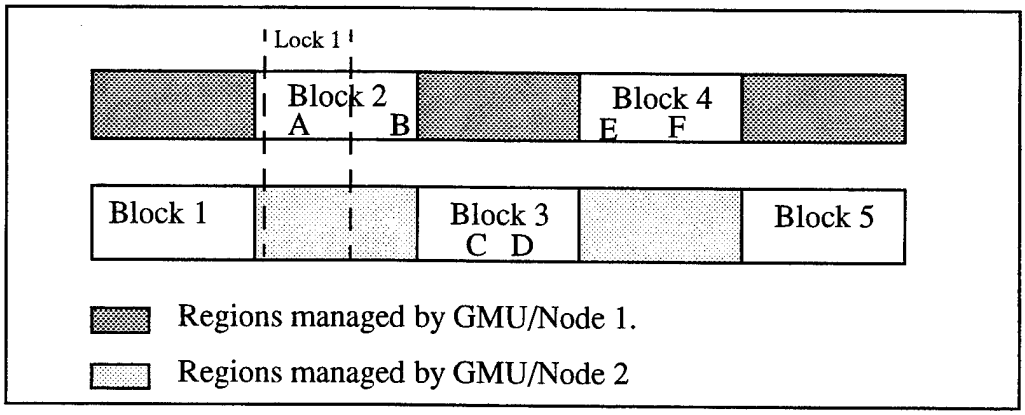


Figure 36: Data distribution across the blocks.

Each node should reply by flushing the block, if dirty, or by sending a clean message in otherwise. After the block is merged at the GMU, the GMU forwards the block to the lock requester and sets the S-bit to one to ensure that new requests are put at the Suspend Queue until the lock is released.

At the time of a lock release, the lock owner should flush the page to the corresponding GMU. The GMU should decrement the counter and reset the S-bit to zero. The changes are inserted into the corresponding data block and the bitmask is then cleared. This sequence of actions are described in Figure 37.

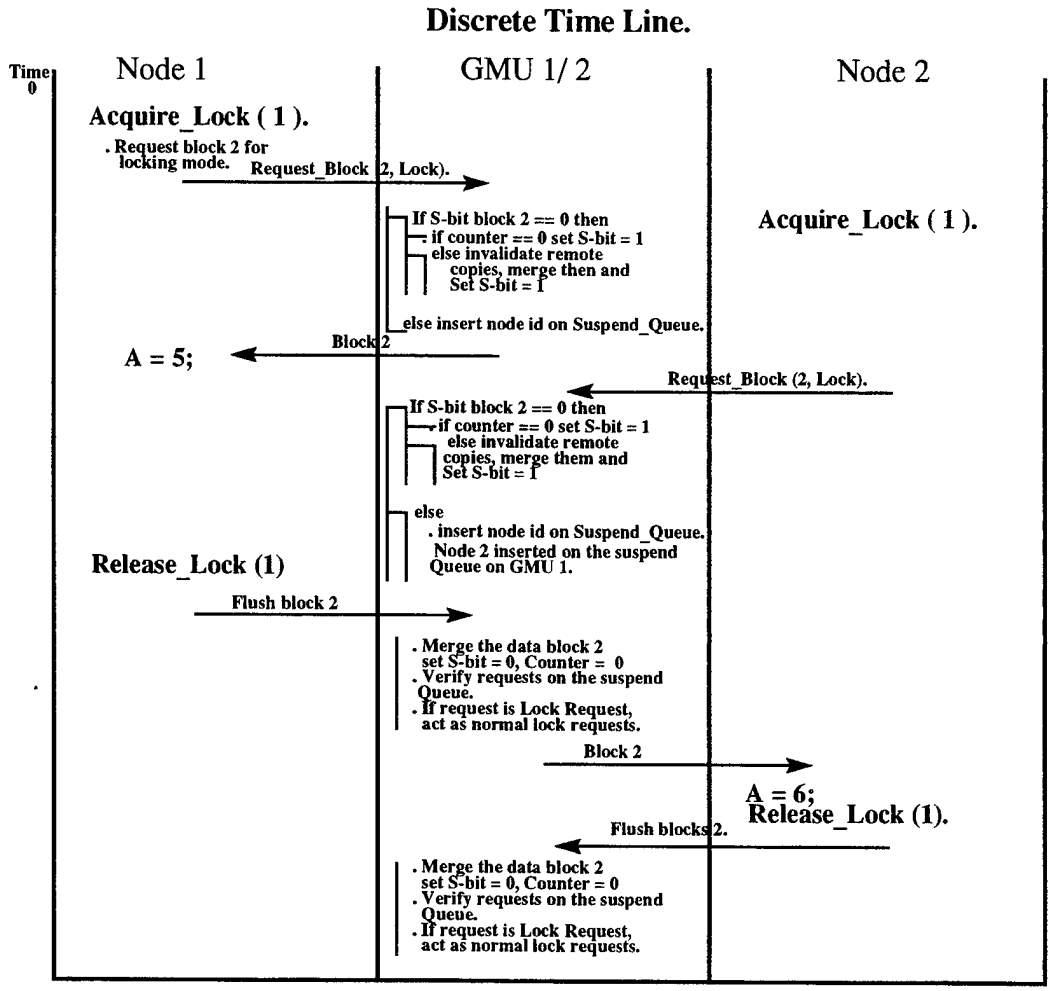


Figure 37: Data Merging.

b. Lazy Data Merging

Now we describe the actions performed by LDM for the same code example of Figure 36. In LDM each lock operation requires a smaller number of messages by

avoiding the issue of multiple invalidate messages, receiving their changes and forwarding the updated block to the acquirer. LDM is also expected to reduce the average message size by forwarding diffs to the next acquirer instead of the entire page.

A more generic example would suffice to give a quantitative view of the number of messages that each approach would require. Assuming that "n" nodes cache an arbitrary page and Process 1 performs a lock request. On the DM, the number of messages for each block that may need to be issued would be:

(n invalidate messages + n update messages + lock granted)

For "k" blocks this amount should be multiplied by k.

For the LDM approach the number of messages would be considerably reduced (for the worst case) to:

(n acquire lock messages + lock granted message)

We describe the actions that are undertaken by both lock owner and requester in Figure 38.

2. Lazy Data Merging: Read and Write Locks

One of the extensions that LDM protocol introduces is the use of read and write locks. Lock ownership will only be modified when a lock is acquired for writing. Our locking semantics allow, at any time, multiple readers, but a single writer to access the critical section they protect. The example of Figure 39 illustrates the actions that should be taken for both read and write locks for the program listed below.

```
Processor 1:
Acq_Write_Lock (1)
    A = A + 1;
Release_Lock (1);
```

```
Processor 2:
Acq_Read_Lock (1)
    D = A + 1;
Release_Lock (1);
```

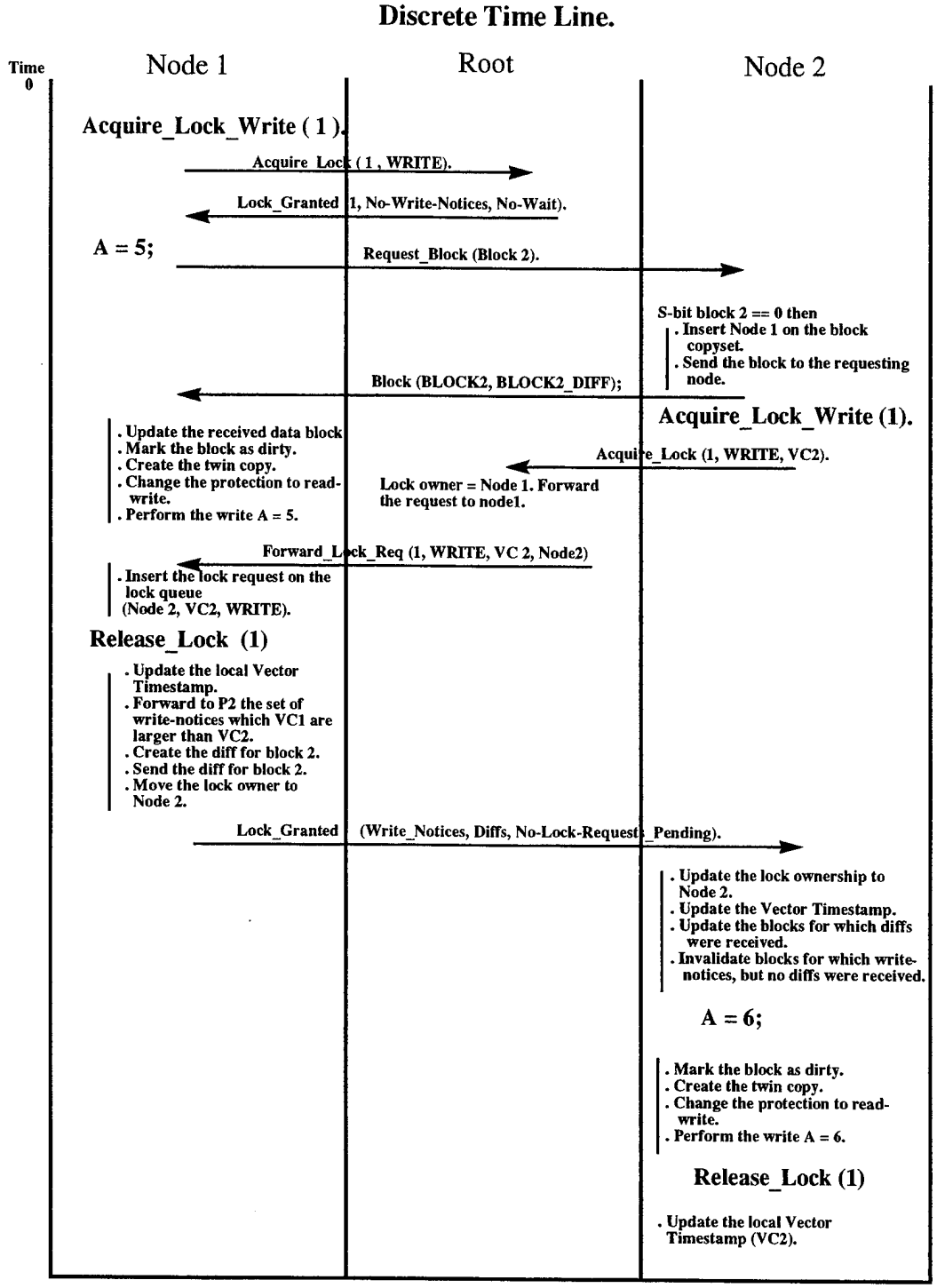


Figure 38: Lazy Data Merging.

Another point that should be made when comparing LDM and DM is the message size and the number of page requests that should be taken whenever a lock operation is performed. Assuming that the number of writes is equivalent to the number of reads to a shared page, we can assume that the average message size can be nearly 50% smaller than the message size on DM. This number can be significantly smaller depending on the ratio of reads/writes. If this ratio is relatively large (i.e., one write for the entire page) the gain becomes significant. On the other hand if the ratio is small (i.e., every word is written into) then the entire page is forwarded, resulting in the same performance of DM.

The number of page requests will also be an issue. An acquire lock operation DM invalidates all cached pages, forcing new page requests for every page even if a small portion of the page is being concurrently accessed. In contrast, LDM does not invalidate any remote pages. It is assumed that before writing into a shared value, the programmer stipulates an appropriate synchronization operation. Therefore, the number of page requests is significantly reduced. The Hybrid coherence protocol should only locally invalidate pages for which it received a write-notice but no update messages.

3. Data Merging and Lazy Data Merging Barrier Call

As mentioned before, barriers have the property of enforcing consistency of the entire global memory or for designated portions of it depending on whether the barrier is of a converge type or not. The approach adopted for barriers requires that pages associated with a barrier object should be locally invalidated and, in most cases, it should have a slightly worse performance than DM. LDM delays sending all diffs to the home nodes until the barrier call (**F**igure 40b). In contrast, DM flushes the dirty pages to the corresponding GMU whenever a page should be replaced. DM distributes the communication during the entire computation, but it requires system support for enforcing that pages are redirected to the corresponding GMU (**F**igure 40a). The figure below compares the two approaches.

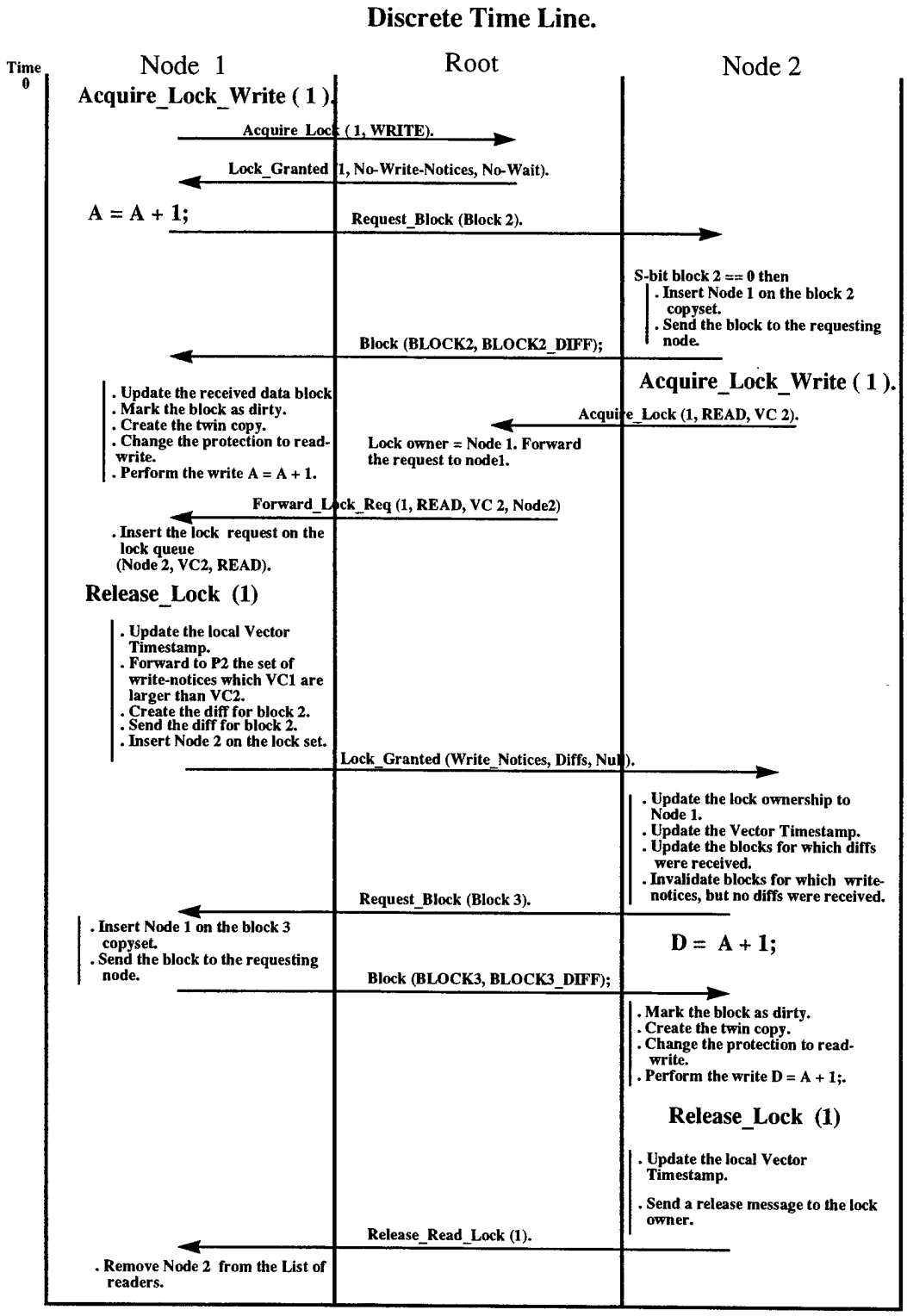


Figure 39: Lazy Data Merging: read and write locks.

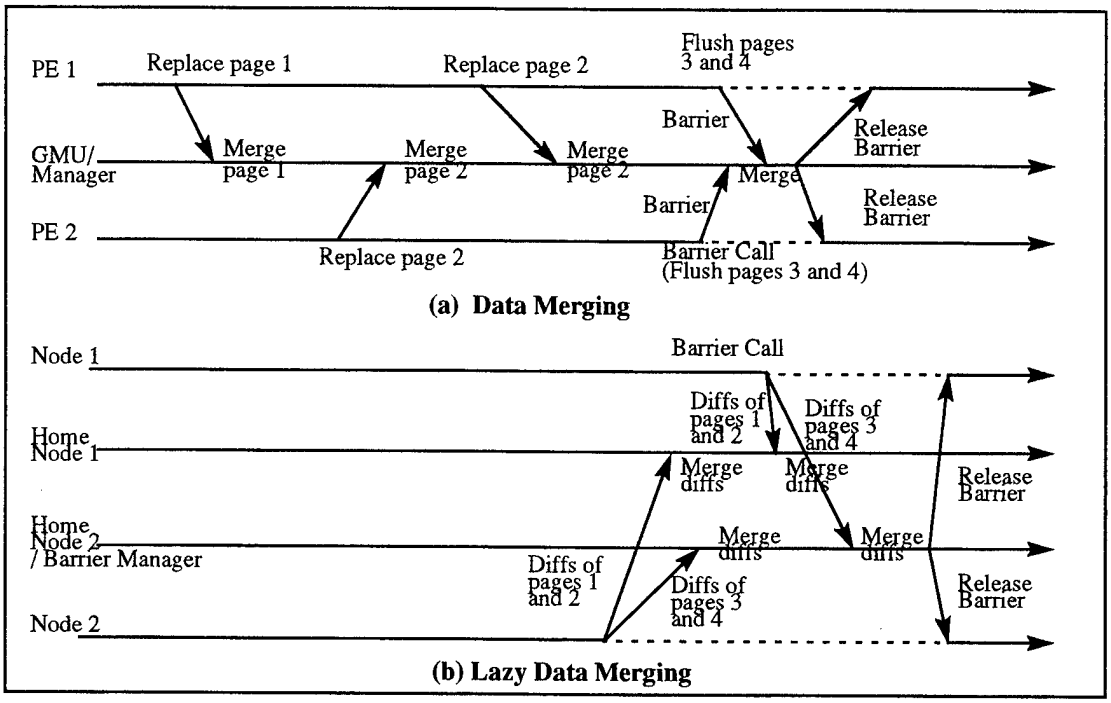


Figure 40: Comparison of DM and LDM during a barrier call.

As can be observed, DM performs slightly better than LDM, by reducing the delay of a barrier operation. To minimize this problem we use diffs instead of flushing the entire data block.

Figure 41 describes the execution of the program below. The node assigned as Root corresponds to the barrier manager and should be generally the node on which the system is being initiated.

```
Processor 1:
    C = C + 1;
    Barrier (1);

Processor 2:
    D = D + 1;
    Barrier (1);
```

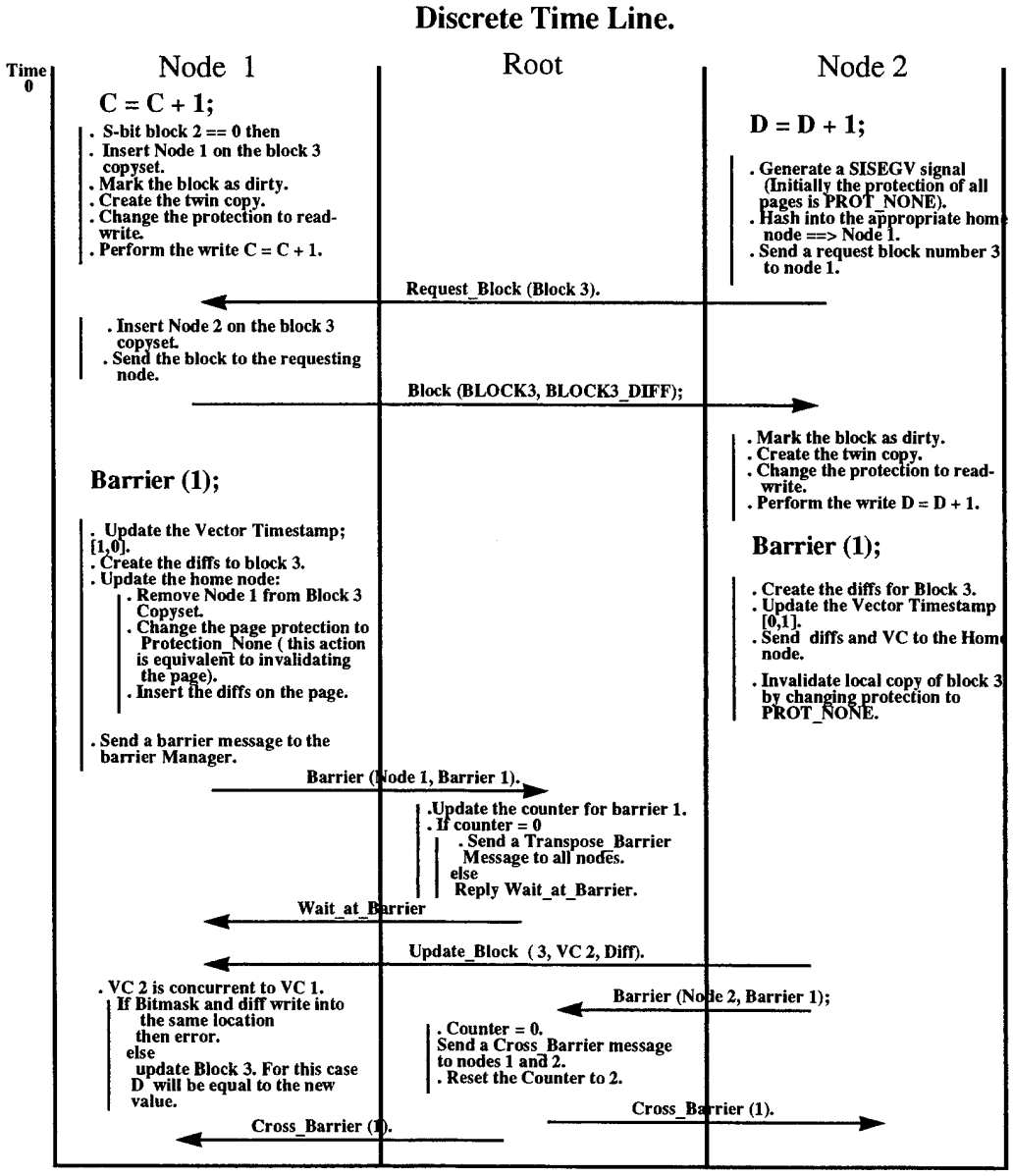


Figure 41: Lazy Data Merging: barrier call.

V. EUREKA: A "LAZY DATA MERGING" IMPLEMENTATION

In this chapter we describe the design of *"Eureka"*, a prototype DSM system that provides a software implementation of the LDM consistency model. Portability and efficiency are our major goals. For portability we use regular Unix BSD 4.3™ system calls (mmap, mprotect, etc., that are wrapped by C++ class definitions). For efficiency we should build the system on a multithreaded environment using signal handlers for detecting both page faults (detected by catching SIGSEGV signals) and received messages (detected by catching SIGIO/SIGURG signals).

In Section A we list the major system components. Section B summarizes the system runtime environment, by exemplifying the interactions that should be undertaken by our DSM system. Section C outlines some implementation details by presenting extracts from the actual system source code. By doing so, we hope to clarify the complexity that is involved in building such systems.

A. DSM SYSTEM ORGANIZATION

This section describes the organization of the *"Eureka"* DSM system. Eureka is composed of two types of entities: objects and threads. The objects are responsible for managing a specific data structure and are considered as "*reactive*" entities[1], that is: they respond to actions by updating their internal state and/or by providing replies to data requests. Objects can also act upon other objects on behalf of an initial thread request. Threads are "*active*" elements that make use of object services.

1. Objects

There are four major objects that should be active during the system lifetime:

• **Synchronization Directory:** this object is responsible for maintaining the state of each individual synchronization mechanism and for providing the interfaces that allow a computing thread to perform the necessary synchronization operations.

1. Reactive objects are objects that take actions upon the reception of an external stimulus.

• **Page Directory:** this object maintains the state of each page that is currently mapped into the local memory. This directory accumulates the roles of managing local and non-local pages. It provides interfaces for both the Synchronization Directory and to the DSM thread.

• **Suspend Queue:** this object manages the insertion and removal of pending page requests. It provides interfaces that are accessed by the timer and by the DSM threads.

• **Process Table:** this object is responsible for storing the addresses of all nodes that integrate the workstation cluster. The process table involves multiple sub-directories. The driven reason is that the system is implemented on a multithreaded environment and more than one thread may issue messages. Therefore, we need to maintain the state not only for remote processes but also for the local threads.

The following subsections will describe in more detail the interfaces that should be provided for these objects.

2. Local Threads

There should be at least three active local threads at any time:

• **Computing Thread:** this thread embodies the user application. For synchronization with the other remote computing threads we introduce synchronization primitives (locks and barriers). In practice, these synchronization operations require access to the methods provided by the "*Synchronization Directory*".

• **DSM thread:** this thread is responsible for managing local/remote block requests. It also manages the creation of "diffs" and the merging of update data that is received from other nodes. In summary, this thread processes all actions that are related to memory management.

• **Timer thread:** the unique role of this thread is to periodically scan the suspend queue. If there are no processes waiting on the suspend queue, the timer thread should yield the execution to another thread, otherwise the timer thread should scan the suspend queue and issue timeout messages which are on the queue longer than the specified threshold value if there are requests pending.

There should be two distinct signal handlers:

• ***Communication*** **handler** (SIGIO/SIGURG signals); and

• ***Memory*** **handler** (handles SIGSEGV signals).

The communication handler is responsible for handling all received messages. Based on the type of operation of the incoming message it may be delivered to one of the

local threads or forwarded to another node. The memory handler, in turn, is responsible for detecting page faults. The page faults can be caused by either page absence or by violating the page protection. In case of protection violation the memory handler actions should vary accordingly to the protocol that is currently in use. Page absences should be handled by issuing a request to the appropriate "home" node.

These threads interact through global data structures and signals (*SIGSEGV* for page faults/protection violation and *SIGIO* for messages) that are issued during program execution.

B. EUREKA RUNTIME ENVIRONMENT

This section describes the various actions that are undertaken by each one of the distinct entities of the system. Figure 42 (Figure 25 on Chapter IV, section D) provides a pictorial description of the subject.

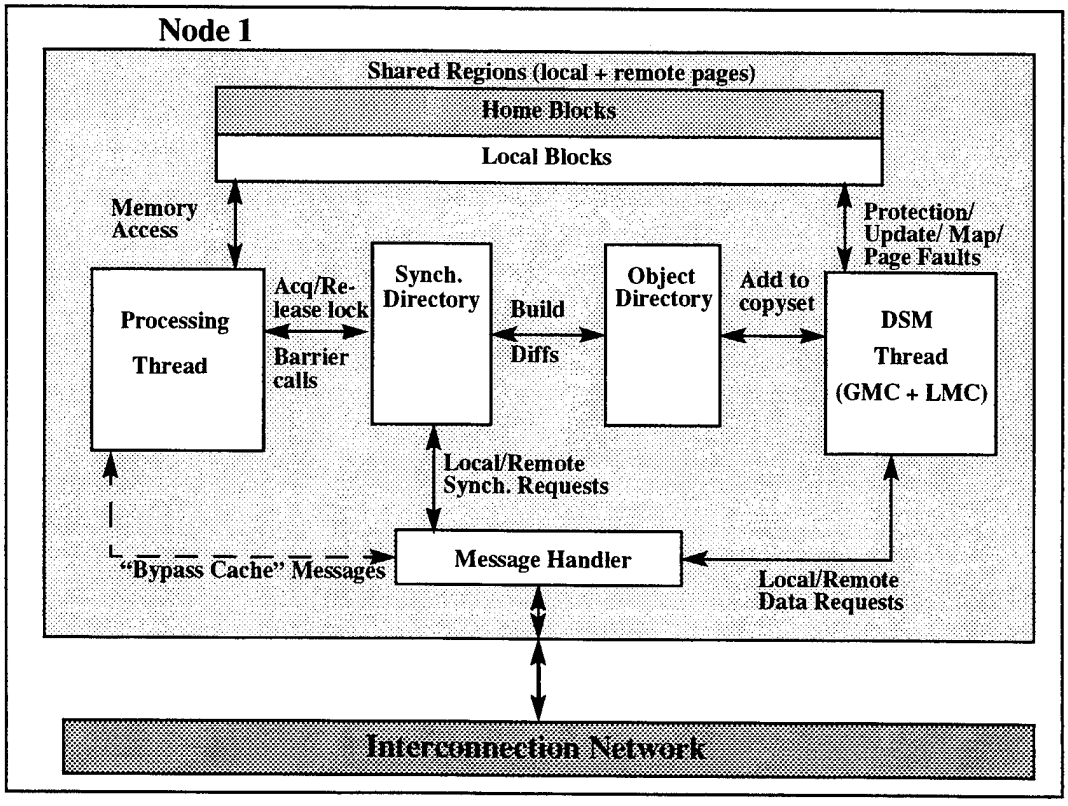


Figure 42: Eureka Runtime Environment.

The Eureka DSM system during its execution can be in one of four states:

- *System Initialization*: when the threads, global objects and statically declared data variables are initialized;
- *System Execution*: when the computing thread is forked;
- *Data Gathering*:when the results are collected;
- *System Termination*:when the global data structures are deallocated and the remote threads are terminated.

Section 1 describes the overall system activities giving an abstract view of the system behavior during the four phases. The remaining sections will complement this initial introduction with more detailed aspects of Eureka at each particular phase.

1. Eureka Execution Overview

The system session should be started by the user from one of the nodes specified at the *Erk.hosts* file. The start-up routine will consist of the call to the macro *Erk_Start (argc, argv)* from within the main routine which will, in turn, be responsible for the initialization of the DSM threads on the various remote nodes. All global variables within the system should be initialized within a function *Erk_Init ( )*. This function will be called from the *Erk_Start ( )* routine. After the initialization of the system's global variables the Master node will spawn the remaining *DSM threads* on each of the hosts defined on the *Erk.hosts* file through the use of the ***"rsh"*** system call. Figure 43 describes the system initialization.

After all DSM threads have been created, the dispatcher node should initialize the globally shared objects and synchronization variables. The main routine will then synchronize all threads through the use of a barrier call, which has no data associated with it, through the call *"Erk_BarrierWait (Num Nodes, NO_DATA)"*. This is a requirement for initiating the dispatch of the computing thread on the remote nodes. Upon crossing the barrier, the dispatcher should initiate the process of forking the computing threads, initiating the execution phase.

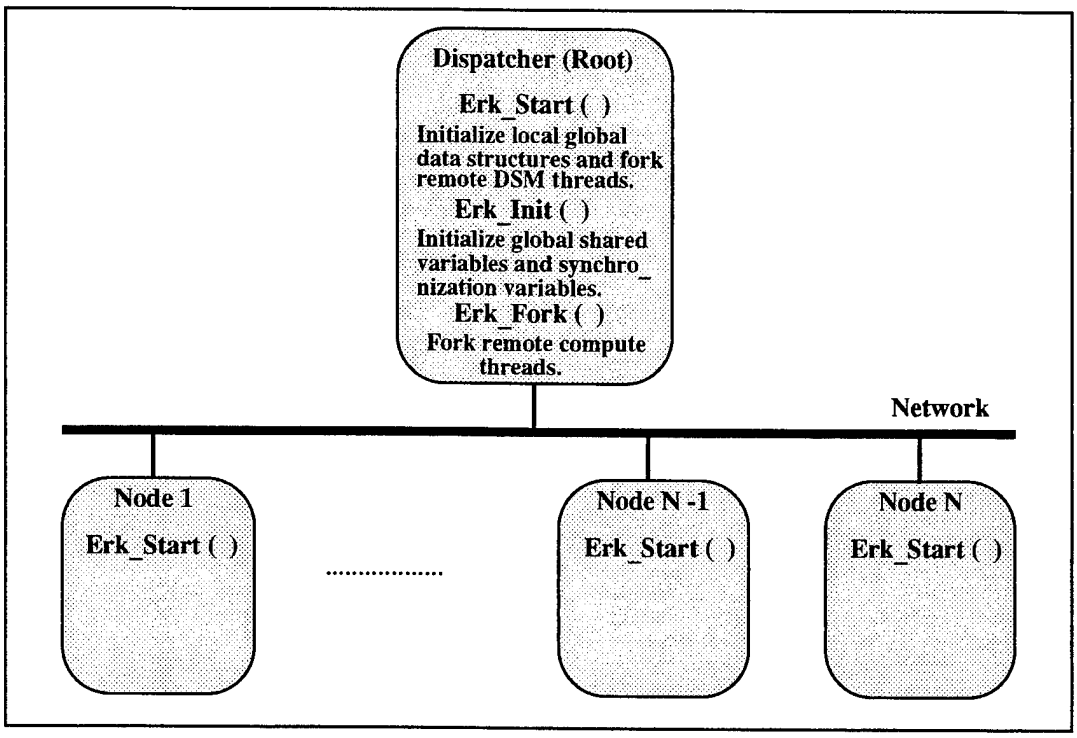


Figure 43: Eureka System Initialization.

The dynamic allocation/dealocation of shared memory as well as the management of barrier primitives are centralized in the "*dispatcher*" node. The dispatcher should also be designated as *root* for the lock variables. When all computing threads have terminated, the global memory should be brought to a consistent state. This will be achieved by a call to the barrier primitive "*Erk_Converge*". This barrier call has the dual role of acting as a control primitive, by synchronizing all processes, and of enforcing global memory consistency, by flushing all dirty blocks to their corresponding home nodes. For blocks which are clean the node should send a "*CLEAN*" message to the corresponding home node.

The next phase consists of the "Data Gathering" phase and is performed solely by the dispatcher. In it the system will present the final result through a GUI and/or by

redirecting the results to a file. The instrumentation phase can be performed concurrently with computation by performing multiple *Erk_Converge* calls. Although important for real-time applications, for which intermediate states are as important as the final one the overlap of Data Gathering tasks with computation may result on a significant loss in performance. A sample main routine should have the following format:

```
main (argc, argv)     // argv display the command line options
{
  // Initialize the Master and later on the Slaves remote threads.
  Erk_Start (argc, argv);
  // If the master initialize the Shared Objects and
  //Synchronization primitives among the Servers.
  // Initialize Shared Memory and Synchronization Primitives.
  Erk_Init ( );
  if (CHILD)
      return 1;
  if (MASTER) {
    // Fork remote computing threads on the other nodes
    Erk_Spawn_Child (Computing thread);
    Execute own local computing Thread;
    // Instrumentation phase:
    Display Results;
    // Shutdown Remote threads
    Erk_Shutdown ( );
}
```

To minimize contention we adopt a "distributed/fixed" approach for partitioning the shared data across the nodes. Therefore, each node is designated "*home*" for a set of data blocks and becomes responsible for its management. The expected performance gain by this approach was stated by Stumm in [SZ90]:

> "....One potential problem with the central server is that it may become a bottleneck, since it has to service the requests from all clients. To distribute the server load the shared data can be distributed onto several servers. In that case, clients must be able to locate the correct server for data access....."

2. Handlers Initialization

a. Communication Port and Communication Handler

The local port, remote port, and message objects are the three objects that should be instantiated as soon as the Process Table is created. Their roles are to provide the appropriate *"UDP"* communication channels that are required for the exchange of messages between the system nodes. The port object provides means to send asynchronous messages to a thread on a remote node. The message object is responsible for providing the semantics (blocking and non-blocking) for sending and receiving messages and to maintain storage for received messages. The remote port object temporarily stores data that describes the source of a received message.

Each node should have a well known port number which is determined by the local ProcessID. The ProcessID represents the order that the nodes were read from the file *"Erk.hosts"*. The Dispatcher node will be assigned a well known port number (between 1024 and 5000) and all other ports should consist of adding the ProcessID to the initial port number. By doing this we allow more than one process on a single node. If it is known that no two processes will ever be assigned to the same node, then the port number can have an arbitrary number greater than 1024.

Received messages are handled in an asynchronous way by defining the appropriate signal handler for *SIGIO* signals and modifying the local port attributes through the use of the *"fcntl"* system call with the flags *"FASYNC | FNDELAY"*. Similar to Quarks [CKK95], once a message is received it is inserted in the message list for the specific thread, in the process subdirectory. If the message is a synchronous message, the thread that is blocked should then be awakened and the corresponding actions (i.e. lock acquired, data block granted, etc.) should be performed. If it is an asynchronous message (page request, lock request, etc.), it should be handled by the "message handler" (i.e., forward of lock requests, etc.) or delivered to the DSM thread (i.e., page requests, lock requests, updates, etc.).

b. Memory Handler

The major role of the memory handler is to detect violations to specified pages access rights. The following table describes the relation between the current page protection value and the actions the memory handler will perform in the event of a "*SIGSEGV*" signal.

Table 3: Page Protection Actions

Current Protection	Actions taken by the memory handler	New Protection
PROT_NONE	Page Fault - perform a page request to the appropriate home node. Change the page protection as soon as a page is received to *PROT_READ*.	*PROT_READ*
PROT_READ	If the page is associated with a lock that was acquired for reading an error condition is reached and the program should abort. Otherwise the protection is altered to *PROT_READ* / *PROT_WRITE* and the appropriate actions should be taken to mark the page as dirty and the creation of its twin copy.	*PROT_READ* / *PROT_WRITE* or generate an error condition.

It is also possible to generate an invalid address. To deal with this particular case the memory handler should verify if the corresponding address maps into a page that has been inserted in the page table. If the page number is not valid then an error condition should be raised.

3. Eureka Shared Data and Synchronization Objects Allocation

This section describes the actions that should be undertaken by the Eureka DSM system for object allocation. This subject is divided into static/dynamic memory allocation and creation of synchronization objects.

a. Static Memory Allocation

The allocation of statically defined shared objects is performed on each individual node. This action should be carried out by the "*Erk_Init ()*" routine. The initialization of shared data is, consequently, performed on a distributed fashion. An image of the shared virtual address space (VAS) is allocated for every process, however, each node is responsible for the integrity of one portion of that space. For these portions we name the node as *"Home"* for this pages.

Static memory allocation is performed at a higher level by the call to the function *Erk_ShMalloc (Var_name, sizeof (Object_ID) * Number of objects).* The underlying system will, in turn, be responsible for allocating the appropriate data structures that should maintain the state of each individual data block.

The allocation of a shared data object involves requests to the Page Directory. The Page Directory's role is to maintain the state of each individual page mapped on the Node. Within each node, shared pages can be divided in two groups: *global pages* and *local pages*. Global pages are the shared pages for which the local node is designated as "*Home*" node and, therefore, is responsible for their management (i.e. merging updates, monitoring the number of cached copies on remote nodes, allocation/ dealocation of bitmasks, storing the list of write-notices, controlling the S-bit, etc.). In a page fault the page should be mapped to the corresponding Home node and perform the request. On the other hand, local pages are acquired remotely from their corresponding Home nodes and temporarily cached at the local process.

The actual allocation of the shared data will consist of two basic steps:

- Mapping the object into memory; and
- Allocation of the Data structures that will manage the shared blocks.

The algorithm that describes this two steps can be summarized as below.

```
Erk_ShMalloc (Any_T * ObjectID, int ObjectSize)

// Map the shared object into memory - by using mmap system call or
// shared memory allocation.
```

```
ObjectID = PageTable.Map_Object (ObjectSize).

// With the initial address and object size allocate the data structures
// that are needed for managing the pages. In accordance with the
// data partitioning policy, identify the pages that should be marked as
// "global" pages, and initialize the data structures that are required for
// its management.
PageTable.UpdatePageTable (ObjectID, ObjectSize).
```

The description of the method Map_Object is described in section C. For the method UpdatePageTable we provide the following algorithm:

```
UpdatePageTable (ObjectID, ObjectSize)
{
  // Verify the number of pages that the object requires.
  NumPages = ObjectSize / PageSize;

  // For each page verify if the page is global - the node is the "Home"
  // or if the page is local. For global pages it is necessary to allocate elements
  // for providing control of the copyset elements.
  for (I = 0 to NumPages -1) loop
    if (IsGlobal (PageID + PageSize * I) ) {
      createControl (PageID + PageSize * I, GLOBAL);
    }
    else
      createControl (PageID + PageSize * I, LOCAL);
  }
```

In reality, both types of page objects (global and local) are constructed in a similar way, but they behave differently. Global pages should allocate copyset lists and perform the coherence operations on updates that are received from the copyset elements.

The major difference between this method and the one introduced by Munin and later on by Quarks is that in these two DSM systems the allocation and initialization of the shared data is performed at a single and predefined node. After the allocation, pages are transferred to remote nodes on demand. This approach presents a relatively high startup time, which can penalize short programs.

In Eureka we propose that the UNIX™ "*fork*" semantics to be followed by performing the allocation of objects defined as shared in parallel. Therefore, when a remote DSM thread is forked the corresponding shared address space is also allocated. The

reduction on the startup time will depend of the data partitioning algorithm that is provided. In Section C we describe the implementation details for the above routines.

b. Dynamic Memory Allocation

For dynamic allocation of shared data we adopt a centralized approach. Therefore, the dispatcher node is assigned the controller for dynamic data allocation and deallocation. As in other implementations (Munin, Quarks, etc.) this problem is solved through the use of *"RPC"* (Remote Procedure Call) style calls to the dispatcher. It is required that the user provide the appropriate *"stubs"* for each method that is needed (e.g., allocation/deallocation, read and writing into the object). The operations that are performed on these shared objects (i.e., Queues, Lists, etc.), should be generally be mutually exclusive. For an appropriate result, the user might need to protect the critical section of its code with locks.

c. Creation of Lock Objects

Locks are managed in a distributed fashion, using the distributed queue algorithm suggested by Florin in [FBYR88]. The requests for lock creation and management should be performed within the *"Erk_Init"* routine. The global LockId will correspond to the actual address of the synchronization object, that should be mapped in global memory by using the *"mmap"* system call. Within the node context of a node the LockID corresponds to its index on the Synchronization Directory. The Synchronization Directory corresponds to a table in which the locks are stored. The constructor for each lock object should include the initial lock owner (the Dispatcher) and the list of pages that are associated with the lock. The algorithm for lock allocation is described below.

```
LockID SynchDirectory.CreateLock (ObjectID *ObjectInitAddr, int Offset,
                                  int Size)
{
  static int i = 0;
  // If ObjectAddr == NULL the lock is a control lock, therefore, no
  // data is associated with it. Otherwise, compute the pages that
  // need to be verified at the time of a lock release.
```

```
    If ((ObjectAddr != NULL) ∧ (Size > 0)) {
        // Based on the address build the list of pages that should be associated
        // with the lock.
        Page_List = Build_ListOfPages (ObjectInitAddr, Offset, Size);
        // Create the new lock having the Dispatcher node as root.
        Lock_List [ i ] = new Lock (Dispatcher, Page_List);
      }
    else
        Lock_List [ i ] = new Lock (Dispatcher, NULL);
    return Lock_List [ i++ ]->LockID ( );
  }
```

The Synchronization Directory is, in essence, a lock table that provides the appropriate mechanisms for allocating, deallocating, acquiring, and releasing of locks. Upon its creation it needs the information of the address from the Page Directory, so that it can request its services such as creation of diffs and update the pages' Vector-Timestamp.

d. Creation of Barrier Synchronization Objects

The creation of barrier objects should be performed in two steps:

- Associate the barrier object with the shared pages that should be updated at the time of a release.
- Register with the barrier manager (Dispatcher node).

Barrier calls can also be used solely for synchronization purposes. If that happens barriers will not update global memory.

Barrier Converge primitives will require that each node traverse its corresponding Page Table and forward the diffs for pages that are dirty to the corresponding Home node, forcing global memory to enter in a consistent state. The algorithm below describes these actions.

```
BarrierID Erk_CreateBarrier (ObjectID *ObjectInitAddr, int Offset,
                                    int Size, int NumProcesses)
{
  static int i = 0;
  // If ObjectAddr == NULL the Barrier is a control lock, therefore,
  // no data is associated with it. Otherwise, compute the pages that
  // need to be verified at the time of a barrier_wait call.
  If ((ObjectAddr != NULL) && (Size > 0)) {
      Page_List = Build_ListOfPages (ObjectAddr, Offset, Size);
      Barrier_List [ i ] = new Barrier (Dispatcher, Page_List, i);
```

```
        Barrier_List [ i ] = Page_addr (Page_Table_addr);
      }
    else
        Barrier_List [ i ] = new Barrier (Dispatcher, NULL, i);
    return Barrier_List[i++]->RegisterBarrier(ThreadID,NumProcesses);
}
```

Although barriers are managed using a centralized approach, a regular barrier call is processed in two steps. The first one consists of sending updates (diffs) to the corresponding home nodes for pages that are dirty and *CLEAN* messages for those pages which are clean. Once the first step is performed the Barrier Primitive will call the barrier manager and wait until a transpose reply is received.

4. Execution Phase

In this section we describe the desired system behavior during the Execution Phase. In Eureka data and synchronization management are closely related. Recall that for correctness we rely on the appropriate use of synchronization primitives.

After the initialization of global data structures (i.e. Process Table, Synchronization Directory, etc.) the computing thread is forked. At this point in time the system can be viewed as three threads running concurrently (timer, DSM and computing threads) and globally defined objects (Page Directory, Synchronization Directory, Process Table and Communication objects) upon which they should act. The next paragraphs describe how these threads and objects interact. We describe these interactions by listing the objects and how each thread relates to it. The algorithmic details are explained in Chapter IV (sub-subsections c, d, e, and f of section E, subsection 2).

a. Suspend Queue

Both the Timer and DSM threads will act over the Suspend Queue. The former by performing insertions and deletions of the IDs of processes which are waiting for an arbitrary data block and the latter by verifying if a timeout has occurred.

b. Page Directory

The Page Directory object receives messages from the DSM and Timer threads and eventually from the memory signal handler (SISEGV signal handler). It also processes requests from the Synchronization Directory for creation of diffs. The DSM thread will interact with the Page Directory whenever a data request/update message is received. The actions for each one of these messages are described in Chapter IV.

Whenever a page fault occurs, the memory handler should take the actions specified in Table 3. The signal handler should consult the Page Table in order to verify the protection attribute of the faulty page. Depending on the protection attribute, it may be needed to perform a page request to the home node. If the page is already in memory the actions can be reduced to modification of its protection.

The interactions between the Timer thread and the Page Directory take place whenever a timeout occurs. At this instant, the Timer should verify the copyset of processes that currently cache a copy of that page and invalidate their pages. Once all updates have been received and appropriately merged, the Counter for that page will equal 0 and the Timer thread should be able to remove the ProcessID from the Suspend Queue and forward the requested page.

c. Synchronization Directory and Page Table

The computing thread, Synchronization Directory, and Page Table should interact whenever a synchronization operation is performed. At the time of an acquire_lock operation a request should be issued to the Current Lock Owner. Upon the receipt of an "acquire" message it will be followed by the set of write-notices as well as by the updates that were issued by the last lock owner. The received write-notices should be appended/ coalesced to the existing set and the diffs should be inserted into memory. The pages for which write-notices and no updates were received should be Invalidated (by setting their protection to *"PROT_NONE"*) and they should be marked as pages that are associated with

a lock by inserting the identity (by inspecting the write-notice tuple) of the node which has last performed a write operation into the page.

At the time of a release operation the Synchronization Directory should inspect the lock queue for pending requests. If there are any, the Synchronization Directory should request that the Page Table the creation of diffs for pages marked as dirty (only the pages associated with the lock), and should also update the state of the write-notices for these pages and then forward this information with the lock granted message and the list of processes waiting for this lock.

Barrier calls should behave in a similar manner, except that all diffs and write-notices that are associated with the barrier object are forwarded to the corresponding home node and all data that is associated with the barrier should be locally invalidated.This will ensure that values that are shared among multiple nodes will be updated after the barrier have been "crossed".

d. Sending / Receiving Messages

Messages are exchanged in Eureka for several different purposes and multiple threads can issue a message. Therefore, when sending a message, it is not enough to observe the IP address. We also need to specify to which remote thread this message should be delivered. To fulfill this requirement some conventions were adopted. The first one is how to specify the Thread ID. The underlying threads package (Cthreads) provides a unique handle for each local thread. At the creation of any local thread this handle should be inserted into the Local Process Table. This will be useful for providing joining/detaches of threads. But what is really needed is a means for globally identifying each remote thread. This will be achieved by combining the Global Process ID with the thread number. The combination is performed as below:

```
ThreadID = Global Process ID << 16 | LocalThreadID.
```

Therefore, the LocalThreadID for the DSM thread running at the Dispatcher node will be the number 0, the Timer thread 1, and the computing thread 2. For the Process

1 the DSM thread will be the number 65536 and so on. For decoding the ThreadID we apply the reverse process.

```
LocalThreadID = ThreadID & 0000FFFF
```

By doing this the system can appropriately identify the source and destination threads for each message. A Eureka's typical message is composed by the fields described in Figure 44. The destination and source ThreadID are necessary to identify the threads within the context of a process. The Operation Code is used to identify the type of message that is currently being used. The Size field specifies the size of the data part. The Packet Number is necessary for messages that contain more than one datagram packet. The message header size is 20 bytes.

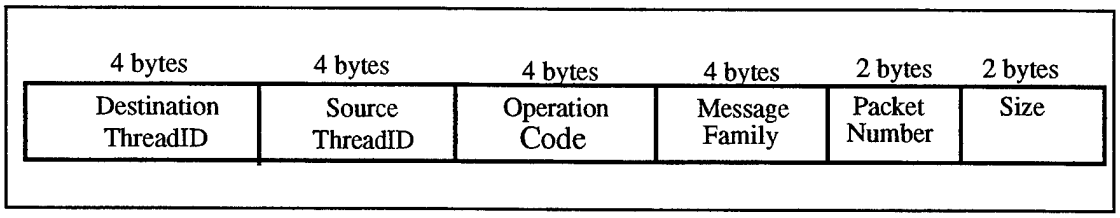


Figure 44: Message header format.

One particular situation for the data field is the case in which the Operation Code consists of an Update message. For this case the message data field may have one of two formats. The first one consists of the diff bitmask and has a size of 128 bytes (Figure 45). Each bit maps to a word within the page. A bit set to one represents the word is dirty and a "0" represents a clean word.

At runtime, whenever a message arrives at a node a *"SIGIO"* signal is raised. The signal handler should then either post the message into the message list for the appropriate local thread (DSM thread for page requests, updates, etc., compute thread for Bypass-cache and synchronization messages), forward the message to another node (i.e.,

lock requests) or even discard the message and reply with an "*INVALID*" message, if there is no such thread on the node.

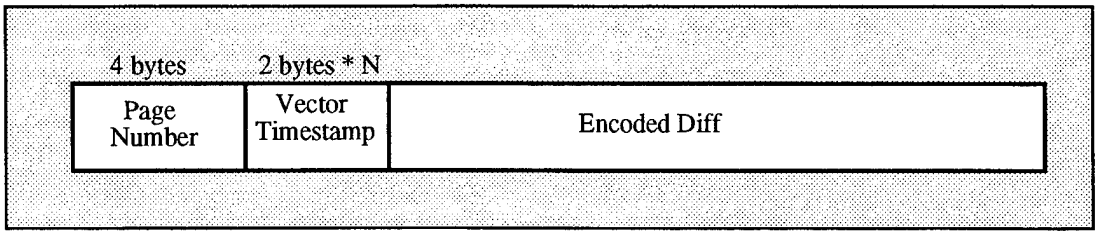


Figure 45: Diff message.

Whenever a thread sends a synchronous message or is waiting for a message to arrive (DSM thread), it should periodically peek on the message list. If the list is not empty it should handle the message, otherwise the given thread will yield the execution to another thread. The C++ code below describes this actions:

```
Receive_Message (ThreadID)
{
  while (LocalProcessTable [ ThreadID ]->MessageList.IsEmpty ( ) )
    cthread_yield ( );
  return LocalProcessTable [ ThreadID ]->MessageList.GetMessage ( );
}
```

e. Operation Codes

The operation codes are divided in four basic types:

- Memory management requests;
- RPC calls;
- Synchronization messages;
- System control messages.

These operations are summarized in the following paragraphs.

Memory Management Messages:

- *REQUEST_BLOCK*: this type of message is sent to the home node of the corresponding block. The message data field is inserted with the block

number.

• *UPDATE_DIFF*: this message will consist of an update message as the result of a page flush or process termination.

• *WAIT_FOR_BLOCK*: whenever a process poses a request and the Suspend-Bit for that page is set at the home node it should reply with a wait message such that a unit does not need to timeout and perform the request again. This action should force the requesting thread to block and wait until the reply arrives. The data field should contain the page number for which the request was performed.

• *BYPASS_READ:* this message should carry the address and size of the data to be read.

• *BYPASS_READ_REPLY*: this message carries the size and the actual data that has been read (the data is considered to be placed at a contiguous address).

• *BYPASS_WRITE*: the data field contains the initial address, the number of bytes to write and the actual data. When the operation is completed the Home node should reply with a "*DATA_WRITTEN*" message.

• *DATA_WRITTEN*: a reply from the Home node is issued when the bypass_write operation is completed (when the remote write has been performed).

• *INVALIDATE_BLOCK:* whenever the timeout value expires the Timer thread will issue a Flush_Block type message for the corresponding block. Only the *LOCAL* processes with the corresponding block dirty will reply with an Update_Block type message. The *home node* will, in turn, update the corresponding blocks, set the S-bit to one, and decrement the counter. This type of message has the same semantics of write invalidate when used in combination with the Locking messages.

• *CLEAN*: the clean message is used to identify a page that is not dirty. Its purpose is to remove the nodeID from the page copyset at the home node.

• WRITE_NOTICE: a list of write notices. They consist of the number of write-notices plus the actual list of tuples "*(BlockNumber, Vector Timestamp, Last_Writer_ID)*".

RPC calls:

As mentioned before, we use a centralized algorithm for managing dynamic memory. The dispatcher node should execute the stub function and return the reply to the requesting node.

• *RPC_CALL*: the client node should pack the stub's name and

arguments, if any, with this message.

• *RPC_REPLY*: consist of the reply form the server stub to the requested method.

Synchronization Messages:

• *ACQUIRE_LOCK:* lock request performed by the source of the message to the "probable" owner. The data field for this message will consist of the LockID, if the lock should be acquired for write or reading and the node's current Vector Timestamp.

• *LOCK_GRANTED*: when the lock request was granted.

• *LOCK_QUEUE:* the list of nodes that are waiting for the lock. (Consist of NodeID and Vector Timestamp).

• *FORWARD_LOCK_REQUEST:* this message is sent whenever the process is not the current lock owner. The lock request is forwarded to the next current owner. The arguments for this message are the data field from the original message as well as the NodeID of the requesting node.

• *WAIT_FOR_LOCK*: reply issued by the lock owner when the lock requested has been inserted on the lock queue.

• *WAIT_AT_BARRIER:* a barrier call, performed by the remote nodes. The barrier manager should in turn decrement the counter and either reply with a *TRANSPOSE_BARRIER* or *BARRIER_WAIT*, depending if the counter value equals 0 or not.

• *BARRIER_WAIT*: a reply from the barrier manager sent whenever the counter value > 0.

• *CROSS_BARRIER*: issued by the manager to all processes registered at the barrier waking then up.

• *CONDITION_WAIT*: call to a given condition variable. The callee should block until the condition becomes true.

• *WAIT_FOR_CONDITION*: reply from the condition variable manager. It means that the condition is false.

• *CONDITION_SIGNAL*: done by any thread informing that the condition is now true.

• *CONDITION_BROADCAST*: wake-up all threads that are currently waiting at the given condition signal.

System Control Messages:

• *ACK*: issued whenever a reply is needed, but no action by the receiver is needed.

- *FORK_THREAD*: fork a new remote thread at the destination node.
- *SYSTEM_SHUTDOWN*: terminate all remote operations.
- *TERMINATED*: the reply when the node is ready to terminate its threads.

5. Data Gathering Phase

This phase is initiated after a call to a barrier converge primitive by all Eureka nodes. The result of this call is that the global memory should enter in a consistent state.

Upon reception of a *"CROSS_BARRIER"* call the dispatcher node should start collecting the necessary data. This task is performed by the underlying system, becoming transparent to the user. The user is responsible for specifying the procedure that should be run at the dispatcher node after the barrier_converge call.

6. Termination Phase

The termination phase is initiated by the dispatcher after the Data_Collection phase and consists of a *"SYSTEM_SHUTDOWN"* message sent to all nodes, issued by the Dispatcher node. Upon receipt of this message, each node should graciously terminate all threads, deallocate its global objects and unmap the shared global memory. Finally, each node replies with a *TERMINATED* message. When all nodes have terminated the dispatcher will be ready to finish.

C. CODE EXAMPLES

This section describes the actual details for implementing the DSM system.

1. Creation of an UDP Port

The constructor for a local UDP Port object should be initialized with the host name and the corresponding port number.

```
UDP_Port::UDP_Port (char *hostName, unsigned short Port)
 {
   localAddr.sin_family = AF_INET;
```

```
    /*
     * Open an UDP socket.
     */
     if ( (sockfd = socket (localAddr.sin_family,
          SOCK_DGRAM, 0) ) < 0 ) {
      printf ( "Value of  sin_family = %d \n", localAddr.sin_family );
      perror ( "Cannot allocate socket") ;
      }
     /*
      * Now bind our local address so that the other processes
      * can find us.
      */
      bzero ((char *) &localAddr, sizeof (localAddr) );
      localAddr.sin_addr.s_addr = INADDR_ANY;
      localAddr.sin_port = Port;

      if ( bind (sockfd, (struct sockaddr *) &localAddr, sizeof (localAddr)) < 0) {
       printf ( "Value of sockfd = %d\n", sockfd);
       perror ( "Cannot bind socket" );
       }

#ifdef DEBUG
   printf ("Done with initialization of the  UDP socket \n.");
#endif

}
```

To receive a message the node should perform a call to the method rcvMsg. This call will be performed within the signal Handler for the SIGIO system call.

```
/*
 * Receive a message from a remote node and returns the sender info plus a
 * pointer to the buffer and the size of the message just received.
 */
 int
 UDP_Port::rcvMsg (RemotePort *From, char *msg, int maxLgth)
{
   int s = 0;
   int rcvDatagramSize = 0;

  /*
   *If From is not a NULL pointer, the  source  address  of  the
   * message  is filled in.  S   is a value-result parameter,
   * initialized to the size of the buffer associated with  From,
   * and  modified  on  return to indicate the actual size of the
   * address  stored  there.  The  length  of  the  message   is
   * returned.  If  a message is too long to fit in the supplied
   * buffer, excess bytes may be discarded depending on the  type
   * of socket the message is received from.
```

```
*/
s = sizeof (From->remoteAddr);

if ( (rcvDatagramSize = recvfrom (sockfd, msg, maxLgth, 0,
        (struct sockaddr *) &(From->remoteAddr), &s) ) < 0)
{
   perror("Error in Recvfrom");
   if (errno != EWOULDBLOCK)
    {
      perror("Error in Recvfrom");
    }
else
    rcvDatagramSize = 0;
}

return rcvDatagramSize;
}
```

Asynchronous I/O

```
/*
 * 1- Now we need to set the process ID to receive the SIGIO or SIGURG
 * signals for the socket associated with fd.This is done with the
 * command F_SETOWN. ( S_SETOWN > 0 -> process ID and
 *                S_SETOWN < 0 -> process_Group ID ).
 * 2- Later on we need to set a flag for FASYNC (Signal process group when
 * ready) and FNDELAY (Nonblocking I/O). This is done with the command F_SETFL.
 */
 if (fnctl (sockfd,F_SETOWN, getpid ( )) < 0)
   perror ("UNIX PORT F_SETOWN error");

 if (fnctl (sockfd, F_SETFL, FNDELAY | FASYNC ) < 0)
   perror ("UNIX PORT F_SETFL error");

/*
```

2. DSM System Calls

Create a single logical address space:

```
/*
   * A zero special file is a source of zeroed unnamed memory. This file is
   * of infinite length. Mapping a zero special file creates a zero-initialized
   * unnamed memory object of a length equal to the length of the mapping rounded
   * up to the nearest page size as returned by getpagesize. Multiple processes
   * can share  such a zero special file object provided a common ancestor mapped
   * the object MAP_SHARED.
   */
```

```
if ( (mapFD = open ("/dev/zero", O_RDWR, umask(0))) < 0)
  printf ("Couldn't open the desired FD for mmap\n");

else {
  if ((virtualBaseAddr= mmap (0, regionSize,
                              PROT_NONE, MAP_PRIVATE, mapFD, 0)) < 0 ) {
     perror ("Cannot Map the correct value");
     printf ("Couldn't allocate memory \n");
  }
  else {
    matrixA = (my_type *) virtualBaseAddr;
    printf (" address of matrixA = %lu \naddress of virtualAddr = %lu",
    matrixA, (long unsigned) virtualBaseAddr);
  }
```

To modify its protection:

```
void
 segv_handler (int sig, int code, struct sigcontext *context, char *addr)
 {

   int pageSize = getpagesize ( );
   int pageAlignedAddr;

   splhigh ( );   // maximize the thread priority.
   sigblock (sigsetmask (SIGSEGV));  // block SIGSEGV signals.

   printf ("\n******* PAGE FAULT on Virtual address = %lu \n\n",
   (unsigned long) addr);
   printf ("PAGE SIZE = %d \n", pageSize);

   /*
    * Page align the faulty address otherwise mprotect will refuse
    * to change protection.
    */

   if (( pageAlignedAddr = (unsigned long) addr % pageSize) != 0)
     pageAlignedAddr = (unsigned long) addr - pageAlignedAddr;
   else
     pageAlignedAddr =(unsigned long) addr;

   /*
    * Now change the protection of the the given address so that I can write
    * into it.
    */
    if ( mprotect ( (char *)pageAlignedAddr, pageSize,
                    PROT_READ | PROT_WRITE) < 0)
      printf ("Setting an invalid address \n");

   sigsetmask (0); // unblock all signals.
```

```
spllow ( );        // reduce the thread priority.
```

3. How to Present Debug Information

To be able to monitor a program running on multiple nodes we allow the display of multiple windows at the dispatcher node through the use of the rsh and xterm system calls.

```
 sprintf(forrsh, " %s%s%s%s%s%s%d%s%s%s%s%s%s%s",
"rsh ",  hostAddress , " /usr/bin/X11/xterm -display ",  masterAddress ":0.0",
" -title Pid_",  i "-",  hostAddress, geom_string, " -e ", path,
"/matmult  /users/work4/tavares/THESYS/WORK/result",
"  /users/work4/tavares/THESYS/WORK/Erk.hosts &");
#ifdef DEBUG
   printf("Rsh command: <%s>\n", forrsh);
#endif

   /* Now execute the remote command */
   system (forrsh);
```

VI. CONCLUSION

The introduction of fast networks (e.g. ATM standard) made both message passing and shared memory systems feasible alternatives for solving computationally intensive problems at a very low cost. They allow combining clusters of workstations into a single abstraction. The major advantage of shared memory systems over their message passing counterparts is that such systems relieve the programmer from the burden of worrying about data movement, which for some applications can become a very complex task.

In this thesis we have performed a comprehensive description and analysis of existing memory consistency models and DSM systems using representative examples of each category (Chapters II and III).

Based on studying of existing DSM consistency models and their implementations, we modified Data Merging to obtain a new protocol, "Lazy Data Merging", which incorporates features from both Lazy Release Consistency and Entry Consistency memory models.

The analysis of multiple DSM systems implementations were particularly important for the design of Eureka, a DSM system that implements the Lazy Data Merging consistency model. To ensure portability we use standard Unix™ system calls (i.e. mprotect, mmap, etc.). Our expectations are that as is the case of PVM and MPI, a portable implementation of a DSM system should contribute for disseminating their use among the scientific community. In Chapter V we provide the indications for this path.

Eureka is a partial implementation of the LDM consistency model. Our preliminary results corroborate the protocol correctness and possibilities to provide performance enhacements. Quarks [CKK95] is also a portable DSM system developed at the University of Utah, and currently provides an implementation of the Eager Release Consistency Model. In order to accelerate the implementation of Eureka we have reused some of Quark's basic components. Our initial performance measurements provide results that are comparable to the original system, but our expectations are that, when fully implemented,

Eureka would present superior results since it implements a more relaxed memory consistency model.

A. SUGGESTIONS FOR FUTURE WORK

As mentioned before, Eureka partially implements the LDM protocol and the conclusion of the implementation of synchronization primitives is needed. Further research is also required for instrumenting the system and collecting an appropriate set of data (i.e., number of page faults, number of diffs for each page and average number of diffs per page, number of synchronization accesses, start-up time, average size and number of messages, etc.). Based on these statistics some improvements can be achieved, since we have designed the protocol adopting a conservative approach.

Other open questions are the consideration of the price paid for implementing a portable solution when compared to system-oriented ones and the benefits of adaptive versus static data mapping policies.

LIST OF REFERENCES

[AAL92] Ananthanarayanan, R., Mustaque Ahamad and Richard J. LeBlanc. Application Specific Coherence Control for High Performance Distributed Shared Memory. In Proc. of the Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS - III), pages 109-128, March 1992.

[ACDB94] Amza, Christiana, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared Memory on Networks of Workstations. Department of Computer Science - Rice University, 1994.

[AH90] Adve, Sarita V., and Mark D. Hill. Weak Order a New Definition. In IEEE vol 8, pages 2-14, 1990.

[AHJ90] Ahmad, Mustaque, Philip W. Hutto and Ranjit John. Implementing and Programming Causal Distributed Shared Memory. GIT-CC-90/49, College of Computing, Georgia Tech, October 1990.

[BA93] Banerji, Arindam et all. High-Performance Distributed Shared Memory Substrate for Workstations Clusters. Technical Report 93-1. Department of Computer Science and Engineering University of Notre Dame.

[BR90] Bisiani, Roberto & Mosus Ravishankar. Plus: A Distributed Shared-Memory System. Computer Magazine page 115 year 1990.

[BS93] Bolosky, William J. and Michael L.Scott. False Sharing and its Effect on Shared Memory Performance. Proceedings of the Fourth Usenix Symposium on Experiences with Distributed and Multiprocessor Systems pages 57-71, September of 1993.

[BZS93] Bershad, Brian N., Mathew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In Proceedings of the 1993 IEEE CompCon Conference, pages 528-537, 1993.

[CBZ91] Carter, John B., John K. Bennett and Willy Zwaenepoel. Implementation and Performance of Munin. In Proceedings 13 ACM Symposium on Operating Systems Principles, pages 152-164, May 1991.

[CBZ92] Carter, John B., John K. Bennet and Willy Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. Computer Systems Laboratory - Rice University 1992.

[CKK95] Carter John B., Dilip Khandekar, Linus Kamb. Distributed Shared Memory: Where We Are and Where We Should Be Headed. Proceedings of Hot Topics on Operating System Principles 1995.

[CS91] Mellor-Crummey, John & Michael L. Scott. Scalable Reader-Writer Synchronization for Shared Memory Multiprocessors. Proceedings of 3rd PPoPP, 1991.

[DCMP91] Dasgupta, P., R. C. Chen, S. Memon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabeu-Auban, P. W. Hutto, M. Y. A. Khalidi, C. J. Wilkenloh. The Design and Implementation of the Clouds Distributed Operating System. Technical Report - Georgia Institute of Technology 1991.

[DF92] Delp, G. S. and Farber, D. J., Memnet -- a different approach to network', Technical Report, Department of Electrical Engineering, University of Delaware 1992.

[FBYR88] Forin, A., Joseph Barrera, Michael Young and Richard Rashid. Design, Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach. Technical Report CMU-CS88-165 August 1988 Computer Science Department Carnegie-Mellon University.

[GLLG90] Gharachorloo, Kourosh, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennesey. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In IEEE vol 8, pages 15-26, 1990.

[GVW89] Goodman,J. R., M. K. Vernon and P. J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors, Proceedings Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, April 1989 pages 64-75.

[HN93] Han, Jay. Porting FastThreads to the KSR1. Technical Report INRIA - France 1993.

[HS93] Stone, Harold S., High Performance Computer Architecture third edition, Section 6.4 page 385 - 402.

[K95] Keleher, Peter. Lazy Release Consistency for Distributed Shared Memory. Doctoral Dissertation, Rice University, Texas. January 1995.

[KFJ94] Koch Povl T., Robert J. Fowler, and Eric Jul. Message-Driven Consistency in a Software Distributed Shared Memory. In 1 Symposium on Operating Systems Design and Implementation pages 75-85, 1994.

[KGGK94] Kumar, Vipin, Ananth Grama, Anshul Gupta and George Karapys. Introduction to Parallel Computing. Design and Analysis of Algorithms. Pages 23-24, 1994.

[KL88] Li, K. IVY: A Shared Memory Virtual System for Parallel Computing. I n Proceedings of the 1988 International Conference on Parallel Processing, pages II 94-101, August 1988.

[KOHH94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum and John Hennessy. The Stanford FLASH Multiprocessor. In Proceedings of the 21st International Symposium on Computer Architecture, pages 302-313, Chicago, IL, April 1994.

[KN93] Khalid, Yousef & Michael Nelson, The Spring Virtual Memory System. Technical Report TR-93-9 Sun Microsystems Laboratories, Inc, 1993.

[KS93] Karp, Alan H., and Vivek Sarkar. Data Merging for Shared Memory Multiprocessors. Proceedings of HICSS, 1993.

[LAM79] Lamport, Leslie. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers, C-28 - pages 241-248, September 1979.

[LJ93] Lea, Rodger & Christian Jacquemot, COOL: system support for distributed object-oriented programming. Technical Report - Chorus Systemes 1993.

[LKBT92] Lelvet, Willem G., M. Frans Kaashoek, Henri Bal and Andrew S. Tannembaum. A Comparison of Two Paradigms for Distributed Shared Memory. Technical Report of Department of Mathematics and Computer Science Vrije Universiteit, The Netherlands 1992.

[LLJN92] Lenoski, Daniel, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta and John Hennessy. The DASH Prototype: Implementation and Performance. In Proc.of the 18th Annual Int'l on Computer Architecture (ISCA'92) pages 92-102. May, 1992.

[LW93] Lee, J. William, Concord: Re-Thinking the Division of Labor in a Distributed Shared Memory System. Technical Report 93-12-05 Department of Computer Science and Engineering - University of Washington 1993.

[ML95] Milutinovic, Veljko et al, A Survey of Distributed Shared Memory Systems. Proceedings of the Twenty-Eigth Annual Hawaii International Conference on System Sciences 1995 Vol I.

[MLU95] Milutinovic, Veljko et al, A Survey of Software Solutions for Maintenance of Cache Consistency in Shared Memory Multiprocessors. Proceedings of the Twenty-Eigth Annual Hawaii International Conference on System Sciences 1995 Vol I.

[MS94] Michael, Maged and Michael Scott, Scalability of Atomic Primitives on Distributed Shared Memory Multiprocessors. Technical Report - Computer Science Department University of Rochester.

[MU94] Mohindra, Ajay and Rachamadran, Umakishore, A Comparative Study of Distributed Shared Memory System Design Issues. GIT-CC-94/35.

[NK93] Nelson, Michael & Yousef Khalid, A Flexible Paging Interface, Technical Report TR-93-20 Sun Microsystems Laboratories, Inc, 1993.

[NL91] Nitzberg, Bill & Virginia Lo, Distributed Shared Memory: A Survey on Issues and Algorithms, IEEE Computer, August 1991 page 52- 60.

[RO93] Ramanathan Gowri and Joel Oren. Survey of Commercial Parallel Machines. Computer Architecture News, Vol 21, No 3 - June 1993.

[SZ90] Stumm, Michael and Songnian Zhou. Algorithms Implementing Distributed Shared Memory. In Computer IEEE, pages 54-64, May 1990.

[TN95] Tanenbaum, Andrew S., Distributed Operating Systems, Prentice Hall 1995 pages 312 and 365 - 371.

[WA95] Watson, Ian and Rawsthorne, Alasdair, Decoupled Pre-Fetching for Distributed Shared Memory. Proceedings of the Twenty-Eigth Annual Hawaii International Conference on System Sciences 1995 Vol I.

[ZB92] Zucker, Richard and Jean-Loup Baer, A Performance Study of Memory Consistency Models. Technical Report No. 92-01-02 Department of Computer Science and Engineering - University of Washington 1992.

[ZSB94] Zekauskas, Mathew J., Wayne A. Sawdon, and Bershad Brian N. Software Write Detection for a Distributed Shared Memory. In First Symposium on Operating Systems Design and Implementation OSDI pages 87-100, 1994.

INITIAL DISTRIBUTION LIST

1. Defense Techincal Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Dudley Knox Library 2
   Code 052
   Naval Postgraduate School
   Monterey, CA 93943-5101

3. Chairman, Code CS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

4. Dr Amr Zaky, Code CS/KA 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

5. Dr Mantak Shing, Code CS/SH 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

6. Dr John Carter 1
   Department of Computer Science
   3190 Merrill Engineering Bldg.
   Salt Lake City, Utah 84112.

7. Dr Alan Karp 1
   HP Labs 3U-7
   Hewlett-Packard Company
   1501 Page Mill Road
   Palo Alto, CA 94304.

8. Instituto de Pesquisas da Marinha - IPqM 5
   Rua Ipiru, n^{o} 2 , Ilha do Governador,
   Rio de Janeiro , RJ, BRAZIL
   CEP 21931 - 090.

9. LCdr Joao Alberto Vianna Tavares 2
Av. Sernambetiba 3300 Bloco VI
apto 2103 - Barra da Tijuca -
Rio de Janeiro - BRAZIL.
CEP 22630.