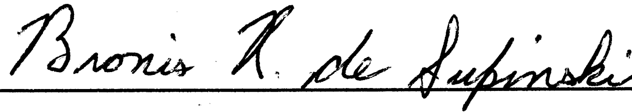
APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the

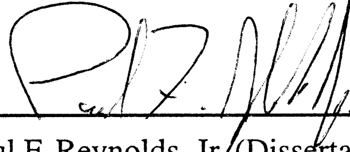
requirements for the degree of

Doctor of Philosophy (Computer Science)



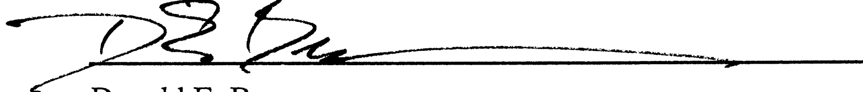
Bronis R. de Supinski

This dissertation has been read and approved by the Examining Committee:



Paul F. Reynolds, Jr. (Dissertation Advisor)

Worthy N. Martin (Committee Chairperson)



Donald E. Brown

Jack W. Davidson

Andrew S. Grimshaw

C. Craig Williams

Accepted for the School of Engineering and Applied Science:

Dean Richard W. Miksad
School of Engineering and Applied Science

May 1998

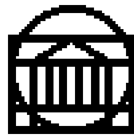
Logical Time Coherence Maintenance

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

at the



University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Bronis R. de Supinski

May 1998

© Copyright by

Bronis R. de Supinski

All Rights Reserved

May 1998

Abstract

We apply techniques based on isotach logical time to the problem of maintaining a coherent shared memory. In isotach logical time systems, processes can predict and control the logical times at which their messages are received. This control over the logical receive time of messages provides a powerful basis for implementing coherence protocols. Existing isotach-based memory coherence protocols are more concurrent than other protocols, but are limited in the topologies on which they work and the reference patterns for which they are suited. We define a new framework for isotach shared memory systems that supports protocols that work for arbitrary topologies and are suited to a wide range of reference patterns. By extending isotach protocols to a wider class of applications and networks, we contribute to the solution of the memory coherence problem.

In addition to extending isotach-based coherence protocols, we advance the theory of isotach systems. We redefine isotach systems to be consistent with potential causality, a new relation among events that captures causality in a less conservative way than Lamport's happens before relation. This redefinition expands the class of correct implementations of isotach systems. We introduce the flex algorithm, a new implementation of isotach logical time that allows different links to be assigned different logical distances. We expect that increased flexibility in assigning logical distances will improve the performance of isotach systems in cases in which links have significantly different real time latencies.

Dissertation Advisor: Paul F. Reynolds, Jr.

To My Children

Acknowledgments

My wife, Ronnie, has been an invaluable source of support and assistance. She encouraged me to return to school when she perceived that I could never feel fulfilled in the job I had. I am deeply grateful for her willingness to be the primary breadwinner for our family, while living away from her parents and her sister's family. Most of all, I thank Ronnie for our two wonderful children.

There are more reasons than I can mention that I am grateful to my advisor, Paul Reynolds. I came to the University of Virginia because he convinced the Dean's Office to provide funding for me despite my letters of recommendation arriving late. His advice, which I value greatly, has always been available when I needed it, whether on the little details about academic life or on more important matters about life in general. And, I will never forget how well he has always spoken of me, even in places I am sure he never imagined himself discussing his students.

I am also very grateful to my co-advisor, Craig Williams. She, too, has always expressed good opinions of me. I have always enjoyed her reaction to my requests for legal advice. She has always had time to discuss even my most outlandish ideas. Her ideas and assistance have been invaluable to me. Most of all, my work expands on many ideas that first appeared in her dissertation.

I thank my daughter, Michelle, for being a great source of joy throughout the entire time we have been in Charlottesville. She has been a wonderful child. The terrific times I have spent with her, whether dropping her off at daycare, playing in the park or just watching TV, have often kept me going. I also thank my son, Paul, who has never failed to

bring a smile to my face in the last year and a third. His glowing attitude and sense of humor make any problem fade into the background.

In addition to their many financial contributions, my parents have always inspired me to do my best and never to give up. Most importantly, they have always been there for me when I have needed them. In addition, I thank their spouses for their encouragement. To many members of my family, including all of my grandparents, I express sincere gratitude for instilling in me a sense of the value of intellectual pursuits. Also, throughout the dissertation process, I have thought of the times I spent as boy talking with my great-grandmother, Granny, about the many doctors and academicians in her family in Poland.

Many other students in my department have helped me work through my ideas and allowed me to bend their ears at length. In particular, I thank Adam Ferrari, Chris Milner, Anand Natrajan, Sudhir Srinivasan, and Kevin Wika for their time and patience.

I thank my committee members Professors Worthy Martin, Donald Brown, Jack Davidson and Andrew Grimshaw for their insights and suggestions about my research. Also, I thank the faculty of the Computer Science Department, particularly, John Knight, for their support. I am grateful to Ginny, Brenda, Tammy and Barbara for their invaluable assistance with the many bureaucratic aspects of graduate life.

I offer thanks to many others just for making life better. To Mark Scolforo, a great friend, for his phone calls and visits that have always lifted my spirits. To Ed Forsman and Josh Goldstein, for many great times during my years in Charlottesville. To Neil Kruger and Steve Wiley, for having been there first. To Pete Garber, for being next.

Finally, I thank those that funded my graduate education, including the University of Virginia, Professor James Ortega and NASA, and the National Science Foundation.

Table of Contents

Chapter *page*

1 Logical Time Coherence Maintenance 1
1.1. Introduction 1
1.2. The Problem 1
1.2.1. Coherence Maintenance 1
1.2.2. Isotach Logical Time Systems 2
1.3. Contributions 3
1.4. Outline of Thesis 4

2 Coherence, Time and the Anticipated Ordering of Events 8
2.1. Introduction 8
2.2. Coherence Maintenance 8
2.2.1. Defining Coherence 9
2.2.2. Coherence Objects 9
2.2.3. Shared Memory Executions 10
2.2.4. Ordering Constraints 11
2.2.4.1. Uniprocessor Ordering Constraints 12
2.2.4.2. Sequential Consistency 12
2.2.4.3. Weak Consistency Semantics 13
2.2.4.4. Isochronicity 15
2.2.5. Coherence Mechanisms 17
2.2.5.1. Copy Tracking 17
2.2.5.2. Coherence Actions and Operations 20
2.2.5.2.1. Miss Actions 20
2.2.5.2.2. Dynamic Protocols 20
2.2.5.2.3. Software-Assisted or Static Protocols 22
2.2.5.2.4. Hardware and Software DSM 23
2.2.6. Locality and Coherence 23
2.2.7. False Sharing 25
2.3. Logical Time 26
2.3.1. Causality and Logical Time 26
2.3.2. Logical Time Systems 27
2.3.3. Coherence Maintenance and Logical Time 28
2.4. Isotach Systems 29
2.4.1. Isotach Logical Time 30
2.4.2. Isotach Networks 31
2.4.3. Isotach Applications 34
2.5. Chapter Summary 35

3 Potential Causality 36
3.1. Introduction 36
3.2. System Model 36
3.3. Defining Potential Causality 38
3.4. Consistency with Potential Causality 39
3.5. Chapter Summary 41

4 Flexibility for Logical Distances 42
4.1. Introduction 42
4.2. Flex Algorithm 43
4.3. Correctness 47
4.4. Logical Time Deadlock 54
4.5. Performance Optimizations 58
4.6. Chapter Summary 60

5 Execution Time and Replication 61
5.1. Introduction 61
5.2. System Assumptions 62
5.3. Logical Execution Time 62
5.4. Scheduled Logical Time 65
5.5. Sequentially Consistent and Isochronous Isotach Systems 66
5.6. Basic Scheduling Algorithm 71
5.7. Delta Coherence Protocols 72
5.7.1. Copy Types 73
5.7.2. Static Owner Update Protocol Coherence Actions 75
5.7.3. Scheduling and Execution Displacements with Replication 77
5.7.4. Protocol Correctness 79
5.7.5. Implementing Split Operations 87
5.7.6. Protocol Design Space 89
5.7.7. Previously Identified Delta Protocols 91
5.8. Chapter Summary 92

6 Owner Update Protocol 93
6.1. Introduction 93
6.2. Overview 93
6.3. Scheduling Horizon 98
6.4. Migration Action Details 100
6.4.1. Basic Mechanics 100
6.4.2. Home Copy Algorithm 102
6.4.3. Replacement of Existing Copies 103
6.4.4. Instantiation of New Local Copies 106
6.4.5. New Owner Directory Initialization 108
6.5. Protocol Correctness 110
6.6. Split Operations and Migration 114
6.7. Releases During Migration 115

6.8. Optimizations 116
6.9. Chapter Summary 117

7 Owner Invalidation Protocol 118
7.1. Introduction 118
7.2. Protocol Overview 118
7.3. Invalidation Action Details 122
7.3.1. Destruction of Existing Copies 123
7.3.2. Instantiation of New Local Copies 126
7.3.3. New Owner Assign Directory Initialization 129
7.4. Protocol Correctness 131
7.5. Optimizations 133
7.6. Chapter Summary 134

8 Local Update Protocol 135
8.1. Introduction 135
8.2. Protocol Overview 136
8.3. Local Copy States 137
8.4. Request Coherence Actions 138
8.5. The Instantiation Action 138
8.5.1. Overview of the Instantiation Action 139
8.5.2. Details 141
8.6. Releases 145
8.7. Split Operations 147
8.8. Optimizations 148
8.9. Chapter Summary 149

9 Conclusion 150
9.1. Introduction 150
9.2. Contributions 150
9.3. Future Work 153
9.4. Concluding Remarks 156

10 References 156

Glossary 170

List of Tables

Table *page*

5.1 Owner Update Protocol Displacements and Distances 78
5.2 Example Scheduling Displacements 85
6.1 Ownership Transition Notation 102
6.2 Home Copy Algorithm 102
6.3 Local Copy Algorithm 104
6.4 Separated IR Action 107
6.5 New Owner Directory Initialization Algorithm 109
7.1 Local Copy Algorithm (Except at the New Owner Location)..................... 124
7.2 Separated IR Action 127
7.3 Initialization of New Owner Directories Algorithm.................................... 128
8.1 Local Update Protocol Displacements and Distances 138
8.2 Instantiation Action Copy Algorithms... 141

List of Figures

Figure *page*

3.1 System Model and Message Notation 37
4.1 Switch Routing Algorithm 45
4.2 Output Buffer Pair 46
4.3 SIU Send/Receive Algorithm 46
4.4 Example Isotach Network 52
4.5 Model of Network Node $\mathbf{v_i}$ 56
4.6 Model of Link Between $\mathbf{v_i}$ and $\mathbf{v_k}$ 57
4.7 Model of Example Isotach Network 58
5.1 Time Line Mapping 64
5.2 Example Physical Topology 65
5.3 Example Logical Topology 65
5.4 Example Effective Topology 65
5.5 Shared Memory Meta-Isotach Time Systems 66
5.6 Correctness Framework 67
5.7 Possible Example Execution 69
5.8 Static Owner Update Protocol Coherence Actions 75
5.9 Owner Action 76
5.10 Components of Correctness Framework 79
5.11 Copy Initialization 81
5.12 Directory Correctness 83
5.13 Possible Owner Update Execution 86
6.1 Migration Distances 94
6.2 Overlapping Copies 95
6.3 Disjoint Copies 95
6.4 Home Action 102
6.5 Disjoint Action 105
6.6 Overlapping Action 105
6.7 Directory Action 109
7.1 Exclusive Period 128
8.1 Local Update Protocol Coherence Actions 137
8.2 Instantiation Action 142
8.3 Release Action 146

List of Symbols

Symbol

$\mathbf{a} \Rightarrow \mathbf{b}$	Event **a** causes event **b**
$\mathbf{a} \rightarrow \mathbf{b}$	Event **a** happens before event **b**
$\mathbf{a} \ulcorner\!\!\rightarrow \mathbf{b}$	Event **a** potentially causes event **b**
$\mathbf{D}$	Logical diameter of an isotach network
$\mathbf{d_{a,\,b}}$	Logical distance from **a** to **b**
$\mathbf{d_m}$	Logical distance traveled by message **m**
$\mathbf{E}$	Shared memory execution
$\mathbf{e_i}$	$\mathbf{i^{th}}$ execution event
$\mathbf{E_S}$	Scheduled execution
$\mathbf{H}$	Scheduling horizon
$\mathbf{last_p}$	Last scheduled execution pulse
$\mathrm{lrd}_{\mathbf{s}}(\mathbf{q_i}, \mathbf{q_o})$	Logical routing distance across switch, **s**, of a message that arrives on port, $\mathbf{q_i}$, of **s** and **s** routes to its port, $\mathbf{q_o}$
$\max(\mathbf{d_{TO}})$	Maximum logical distance from any recipient of Transition Operation (TO) to old owner copy
min_send	Current minimum possible local send pulse
$\mathbf{N}$	Number of processes
$\mathbf{p_i}$	$\mathbf{i^{th}}$ process
$\mathbf{t_a}$	Logical time of event **a**
$\mathbf{t_e}$	Execution time of execution event **e**
$\mathbf{t_q}$	Initial token count of port **q**
$\mathbf{t_r}$	Logical receive time (of a message)
$\mathbf{t_s}$	Logical send time (of a message)
χ	Scheduling displacement (of a request)
$\chi_{\mathbf{a}_{\mathrm{MAX}}}$	Maximum of possible scheduling displacements for request **a**
$\chi_{\mathbf{I}}$	Maximum scheduling displacement of any request in isochron **I**
$\chi_{\mathbf{T}}$	Scheduling displacement of an ownership transition
$\tau_{\mathbf{max}}$	Maximum scheduled execution time of any already scheduled request
δ	Execution displacement (of a copy)
Φ	Execution distance of an execution event
τ	Scheduled execution time (of a request)
$\tau_{\mathbf{T}}$	Scheduled execution time of an ownership transition
$\#(\mathbf{p}, \mathbf{B})$	Number of occurrences of place **p** in bag **B**

Chapter 1:
Logical Time Coherence Maintenance

1.1. Introduction

Isotach logical time systems support novel, powerful techniques for maintaining shared memory coherence. They allow processes to control the logical times at which the messages they send are received and, in some cases, the logical receive times of response messages as well. Our thesis is that the theory of isotach systems, first given by Williams [Wil93], can be extended to increase the flexibility of isotach systems in ways that allow them to serve a wider range of networks and applications.

1.2. The Problem

We address problems in two separate but related areas: coherence maintenance and isotach systems. We discuss each below.

1.2.1. Coherence Maintenance

The problem we address is how to use isotach systems to maintain coherence in shared memory. Maintaining memory coherence concerns enforcement of ordering constraints on accesses to replicated shared addresses. This problem is known as the cache coherence problem in parallel computation and the distributed shared memory (DSM) problem in distributed computation. Coherence maintenance is difficult because copies of shared memory addresses are distributed but the semantics of memory accesses require that they appear to occur on a monolithic memory.

Our definition of the coherence maintenance problem differs slightly from that of other researchers since we include atomicity in the problem. Most solutions enforce sequencing constraints represented by some type of consistency semantics but leave atomicity to be enforced through other mechanisms. We extend the coherence maintenance problem to include how to enforce atomicity constraints as well as sequencing constraints.

The existing isotach-based memory coherence protocols allow greater concurrency than other coherence protocols in the following ways: they enforce atomicity constraints without requiring the use of locks; they allow multiple readers and writers to the same shared data; and they allow pipelining without sacrificing sequential consistency. In fact, they can enforce sequential consistency and still offer more concurrency than systems that enforce weaker consistency semantics with traditional technology. Simulation studies [dWR96, RWW97] have established the potential of isotach shared memory systems. These studies show they outperform traditional systems for workloads that include atomicity requirements or hot spots. However, previously existing isotach-based protocols are limited in the topologies on which they work and the reference patterns they support. We define a theory that enables us to develop protocols without those limitations.

1.2.2. Isotach Logical Time Systems

In addition to addressing the problem of how to use isotach systems to achieve memory coherence we address improving isotach systems themselves.

The existing theory of isotach systems requires that they be consistent with Lamport's *happens before* relation. We seek a less constraining requirement that would still capture important causal relations among events. A less constraining requirement is desirable because it increases the number of correct implementations.

Another way in which isotach systems are currently overly constrained is in the assignment of logical distances to links. Logical distance and logical time are related in isotach systems and logical distance is important in determining message latency. Existing algorithms for implementing isotach systems do not allow real time latency to be taken into account in assigning logical distances. This assumption reduces the applicability of isotach systems to networks with non-uniform link latencies. We seek a set of less constraining rules for assigning logical distances, and algorithms that implement these rules.

1.3. Contributions

We define an extended theory of isotach systems that expands the class of correct implementations of isotach systems, increases the applicability of isotach systems to networks with non-uniform link latencies, and creates a unifying framework for isotach shared memory systems that supports the design of several new coherence protocols.

Our contributions are as follows:

- We redefine isotach systems to be consistent with *potential causality*, a new relation among events that captures causality in a less conservative way than Lamport's *happens before* relation [Lam78]. This change expands the class of correct implementations of isotach systems. Previously, isotach systems were required to be consistent with the *happens before* relation. However, prototype isotach systems and other proposed implementations can be inconsistent with the *happens before* relation although they are causally consistent. Redefining isotach systems to be consistent with *potential causality* supports these causally consistent implementations, as well as isotach network algorithms that are difficult to implement under the old definition.

- We introduce a new isotach network algorithm that allows isotach systems to assign different logical time distances to different links. We expect that increased flexibility in assigning logical latencies will improve the performance of isotach systems in cases in which links have significantly different real time latencies.

- We redesign the framework for isotach shared memory systems to provide a unifying theory that addresses several issues that were not integrated in the

previously existing framework. By eliminating the use of a physical canonical copy, the new framework supports the design of new isotach-based coherence protocols that extend isotach-based coherence techniques to systems with arbitrary topologies, to applications with a wider range of access patterns and to a simpler class of isotach systems than that required by previous isotach-based coherence protocols. In addition, the new framework directly supports optimizations not addressed by previous research and demonstrates that each correct isotach-based protocol represents a class of correct protocols.

1.4. Outline of Thesis

The remainder of this thesis is organized as follows:

In Chapter 2, we discuss three areas of research related to this thesis. We provide extensive background in coherence maintenance and briefly discuss other logical time systems, particularly as applied to the coherence maintenance problem. Then, we present previous research in isotach systems.

In Chapter 3, we define *potential causality* and its system model. Also, we present conditions that ensure a logical time system (LTS) is consistent with *potential causality.*

In Chapter 4, we present the *flex algorithm*, the first isotach network algorithm that allows logical distances to differ from the number of switches through which messages travel without requiring each pair of network nodes to communicate directly. This flexibility allows the logical distances in an isotach network to reflect the point-to-point message latency of the underlying hardware. This ability is an important advance in isotach technology since end-to-end message latency in isotach networks is proportional to the logical distance that the message travels. We prove the flex algorithm implements an isotach LTS and show it provides great flexibility for logical distance assignments. We then present a Petri net model of the algorithm that allows us to determine if a set of logical distance

assignments will cause the algorithm to deadlock. This model indicates similar Petri net models may be a rich source of additional isotach network algorithms.

In Chapter 5, we describe our new framework for isotach shared memory systems. This framework enables the design of several protocols for non-equidistant networks. Since all messages travel the same logical distance in an equidistant network, the sender of a message knows the logical distance that any response message travels even if the destination of the response is not known. In a non-equidistant network, the sender of the original message cannot know the logical distance that the response travels if its destination is unknown. Our new framework allows the sender to anticipate the logical times of execution events despite this incomplete knowledge. After defining the framework for systems without replication, we apply it to isotach-based coherence protocols. Section 5.7 develops correctness criteria for these coherence protocols and presents the *static owner update protocol*, in which the *owner copy* is a distinguished copy that services misses and distributes updates. This protocol extends a previously defined isotach-based coherence protocol [Wil93] to non-equidistant networks.

In Chapter 6, we present the *owner update protocol*, which modifies the static owner update protocol to include our mechanism for dynamically relocating the owner copy. An ownership migration mechanism is desirable since the appropriate location may not be static and is often difficult to predict. Although Williams's equidistant protocol includes a similar migration mechanism, the problem is substantially more difficult in a non-equidistant network since a migration generally changes the logical distance from the owner copy location to any other location. Our highly concurrent migration mechanism does not suspend access to the copies and nodes can retain their copies despite the migra-

tion. Also in this chapter, we introduce the concept of the *scheduling horizon* of an isotach shared memory system and use it to allow a migration to proceed without affecting the execution of requests that a node scheduled before it received notification of the migration.

In Chapter 7, we present the *owner invalidation protocol*, the first invalidation protocol designed for isotach networks. This protocol provides a writer with exclusive access through modifications of the migration mechanism of the owner update protocol. Since it provides a writer with exclusive access, the owner invalidation protocol can exploit long write runs, a reference pattern in which a single process repeatedly accesses the coherence unit. Unlike traditional invalidation protocols, our protocol naturally adapts to reference patterns that do not exhibit long write runs and allows the initiating write to complete prior to providing the writer with exclusive access. Supporting *split operations*, a mechanism that allows isotach systems to execute structured atomic actions without using locks, in an invalidation protocol is difficult. We demonstrate that the owner invalidation protocol implements split operations correctly.

In Chapter 8, we present the *local update protocol*, the first isotach-based protocol to support dynamic replication without requiring an *extensible isotach network*. An extensible isotach network ensures that the logical send time of a response message is a known function of the logical receive time of the original message. Extensibility simplifies dynamic replication of shared data but extensible isotach networks are more complicated and may have higher message latency than isotach networks that do not support extensibility. The prototype systems being built by the Isotach Project are not extensible. Currently, these prototype systems create all copies statically during system initialization. Since the local update protocol can create and destroy copies dynamically in response to

the observed reference pattern, it should improve performance in these systems significantly. As with the existing mechanism, the protocol has the drawback that each copy must have a directory of all other copies. We show that the local update protocol is correct and does not require an extensible isotach network.

In Chapter 9, we present our conclusions and ideas for future work.

Chapter 2: Coherence, Time and the Anticipated Ordering of Events

2.1. Introduction

We present basic definitions, concepts and research related to shared memory coherence maintenance. We review concepts and previous work in coherence maintenance and then discuss causality and logical time, including other applications of these concepts to coherence maintenance. We conclude with a discussion of isotach systems, which can anticipate the logical times of causally related events.

2.2. Coherence Maintenance

Coherence maintenance extends the concept of cache coherence to more general systems. Coherence maintenance was originally explored in systems with memory physically shared by multiple processors, each associated with a local cache memory. *Distributed shared memory* (DSM) also requires coherence maintenance. DSM provides transparent shared memory in systems where physical access to each memory unit is limited to the local processing node [AbK85, Che85, LiH86]. The coherence maintenance problem also occurs in distributed object oriented systems that replicate objects to improve performance [DLA91, LeA92, TKB92]. Our work explores this problem in the context of cache coherent systems and DSM.

2.2.1. Defining Coherence

We define the coherence maintenance problem as the concurrency control problem with replication of shared data. The concurrency control problem is to ensure that every execution of a parallel program is consistent with its ordering constraints. Our definition of coherence expands other definitions to include all aspects of concurrency control.

The definition of coherence has evolved with the exploration of the ordering constraints that a parallel system can enforce. Censier and Feautrier defined a system to be cache coherent if each read returns the latest write [CeF78]. Unfortunately, the possibility of concurrent writes complicates the determination of the latest write in a multiprocessor [DSB86]. Rudolph and Segall defined a virtual serial execution that determines the latest write in bus-based architectures [RuS84]. However, this total order does not extend to general networks since it depends on the serialization provided by the memory bus.

Consistency is a later definition of coherence [Col92]. This ordering constraint requires that all processes observe writes to a given memory location in the same order, although processes need not observe writes to different locations in the same order [GLL90, Adv93, AdG96]. Consistency prohibits executions that are allowed by some ordering constraints, such as causal memory [JoA94].

2.2.2. Coherence Objects

Shared memory systems provide a global address space that is accessible by all processes. Coherence protocols service shared memory requests. Each request reads or writes a shared memory *variable*, the basic unit of all shared memory accesses. Protocols maintain state information in order to service requests correctly for each *coherence unit*,

which is a group of a variables. The *coherence granularity*, the size of a coherence unit, is a cache line in cache coherent systems and a main memory page in most DSM systems.

Memory systems place *copies* of coherence units near processes that access them in order to reduce the latency of shared memory requests. The locations of *static* copies are determined at the beginning of program execution. Systems with *dynamic* copies can create and destroy copies during program execution.

Shared memory systems can directly support objects of varied sizes or use fixed size locations, which generally allow more efficient hardware support. In our model, the system determines the fixed size of a variable. Program-level accesses of varied sizes are emulated through multiple accesses. Our results only apply to systems that support objects of varied size if the program-level entities are disjoint.

2.2.3. Shared Memory Executions

In our shared memory model, a program consists of **N** processes, $\{\mathbf{p_0}, ..., \mathbf{p_{N-1}}\}$, that are logically connected by the shared memory system. The processes issue shared memory *requests* in addition to performing computation and private memory accesses. The shared memory system must associate a value with each shared read request and store the value associated with a shared write request by the process in at least one copy.

Several significant events occur during the service of each request. The *issue* event occurs when the process provides the request to the memory system. The request is associated with the *send* and *receive* event of each message that its service requires. The *deliver* event of a read request occurs when its value is returned to the process. Exactly one *execution event* associates the value of a copy with any read request. There is at least one execution event for each write request and each stores its value into a distinct copy.

An *execution* of a shared memory program consists of every execution event for all shared memory requests and their associated values. A total order over the execution events, the *execution order*, is defined by the real times that the events occur, which we assume are distinct. *Equivalent executions* consist of the same requests associated with the same values. Formally, the executions $\mathbf{E} = \langle \mathbf{e}_0, \ldots, \mathbf{e}_\mathbf{n} \rangle$ and $\mathbf{E}' = \langle \mathbf{e}'_0, \ldots, \mathbf{e}'_\mathbf{m} \rangle$ are equivalent if $\forall \mathbf{i}, 0 \le \mathbf{i} \le \mathbf{n}, \exists \mathbf{j}, 0 \le \mathbf{j} \le \mathbf{m}$, such that $\mathbf{e}'_\mathbf{j}$ executes the same request as $\mathbf{e}_\mathbf{i}$ and associates the same value with the request as $\mathbf{e}_\mathbf{i}$ and $\forall \mathbf{j}, 0 \le \mathbf{j} \le \mathbf{m}, \exists \mathbf{i}, 0 \le \mathbf{i} \le \mathbf{n}$, such that $\mathbf{e}_\mathbf{i}$ executes the same request as $\mathbf{e}'_\mathbf{j}$ and associates the same value with the request as $\mathbf{e}'_\mathbf{j}$.

For systems without replication, we use *conflict equivalence* [Pap86]. This simpler formulation is based on *conflicting* requests. Two requests conflict if they access the same variable and at least one is a write. Accesses to distinct variables never conflict since variables are disjoint. Without replication, two executions are equivalent if they are identical other than the order of the execution of non-conflicting requests. Since there is only one copy of each variable, the same value is associated with each request.

2.2.4. Ordering Constraints

A shared memory system can enforce several types of ordering constraints. *Consistency semantics* are ordering constraints that limit the values that the system can associate with a read request [AdG96]. Other constraints enforce grouping of requests. Our shared memory systems enforce the consistency semantics of sequential consistency and guarantee the grouping constraint of isochronicity. Guaranteeing these properties, like many other ordering constraints, is non-trivial even in systems without replication.

Ordering constraints define a *correctness set*. This correctness set consists of all shared memory executions that conform to the constraint. A shared memory system

enforces an ordering constraint if every possible execution in the system is in the correctness set of the constraint. We relate constraints based on their correctness sets. *Equivalent* constraints have the same correctness sets. The correctness set of a *stronger* constraint is a proper subset of that of a *weaker* constraint. Generally, programming is simpler with a stronger constraint since fewer executions can occur for a given program.

2.2.4.1. Uniprocessor Ordering Constraints

Program order is an ordering constraint that requires memory requests appear to execute in the sequential order specified by the program. With uniprocessors that enforce program order, every execution is equivalent to one in which each instruction is issued and completed one at a time. Techniques like interlocks and scoreboarding allow instruction level parallelism in these uniprocessors. Program order, or *processor consistency*, for shared memory systems requires that an equivalent execution exists in which the requests of each process occur in the sequential order specified by its program [Goo89].

The apparent indivisibility of writes is another uniprocessor ordering constraint that is adapted for shared memory systems. *Write atomicity* requires that an equivalent execution exists in which the multiple execution events of each write request occur consecutively [Col92, AdG96]. Thus, there exists an equivalent execution without replication.

2.2.4.2. Sequential Consistency

Sequential consistency extends uniprocessor memory semantics to multiprocessors. A machine 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 each individual processor appear in this sequence in the order specified by its pro-

gram [Lam79].” This ordering constraint requires that the shared memory system enforces write atomicity and program order. Thus, for each possible execution, an equivalent execution must exist in which the requests of each process occur in the sequential order specified by its program and the execution events of each write request occur consecutively.

Many sequentially consistent multiprocessors exist. Sequentially consistent bus-based systems limit pipelining of requests to ensure program order and enforce write atomicity with the serialization provided by bus acquisitions. However, bus saturation limits these machines to about twenty processors even with low bandwidth protocols [ASH88].

Enforcing sequential consistency is more difficult in general interconnection networks. Few sequentially consistent protocols for general interconnection networks allow concurrent write copies. The protocols that do allow them generally require additional message rounds to ensure a total order of write requests [WiL92, AdG96]. As with the bus-based systems, most sequentially consistent protocols for general interconnection networks limit pipelining to ensure program order.

2.2.4.3. Weak Consistency Semantics

Sequential consistency is generally the strongest consistency semantics that systems enforce. Several researchers have proposed weaker consistency semantics. Systems use these semantics to allow pipelining of shared memory requests. Our protocols enforce sequential consistency and allow pipelining. Weak consistency can alleviate the effect of false sharing, a problem for coherence protocols that we discuss in Section 2.2.7.

The design space of consistency semantics is large. We discuss consistency semantics based on causality in Section 2.3.3, after we define causality [HuA90, AHJ91]. Some consistency semantics, such as total store order and partial store order, weaken pro-

gram order [SPA92]. Semantics that weaken write atomicity are more common. In the remainder of this section, we explore several of these semantics in more detail.

Strong and *weak ordering* are early weak consistency semantics [DSB86]. Strong ordering requires that if process **A** observes a write by process **B** then **A** cannot subsequently observe a write that **B** observed before its write. Although strong ordering was proposed as equivalent to sequential consistency, strong ordering is weaker since processes may observe concurrent writes in different orders [AdH90]. Weak ordering prohibits overlapping a synchronization request with any other shared requests of the same process, while synchronization requests must be strongly ordered.

Constraints derived from weak ordering use special synchronization primitives to simplify pipelining of shared requests. Release consistency exploits the semantic difference between lock (acquire) and unlock (release) synchronization requests [GLL90]. Under release consistency, a process can overlap lock requests with previous requests to shared variables and unlock requests with later requests to shared variables. Lazy release consistency (LRC) weakens release consistency by allowing unlock requests to overlap with previous requests to shared variables [KCZ92]. LRC delays causally subsequent lock requests by any process until the overlapped shared requests have completed. The European Declarative System previously proposed this optimization [BoI91]. Entry consistency is similar to LRC but only restricts overlapping requests to a shared variable and the lock with which it is explicitly associated [BeZ91]. Scope consistency provides a similar benefit but uses program structure to eliminate the need for explicit associations [ISL96].

Most weak consistency semantics have operational definitions. Their specifications make reasoning about the relationships of consistency semantics difficult. For several weak consistency semantics, Adve and Gharachorloo have both explored program

restrictions for which all possible executions are sequentially consistent [Adv93, Gha95]. Although this improves the situation, demonstrating that a general program conforms to the restrictions is difficult. Also, this approach does not specify the weak semantics for programs that do not conform to the restrictions.

2.2.4.4. Isochronicity

In addition to consistency semantics, grouping shared requests is an important facet of concurrency control. *Isochronicity*, an ordering constraint that most coherence protocols do not enforce, requires that all possible executions are *isochronous* [RWW97]. An *isochron* is a group of requests that are issued consecutively by a process. An execution is isochronous if an equivalent execution exists in which each isochron executes without interleaving with other requests.

Isochronicity is closely related to *atomicity* [Lom77, OwL82, Lam86]. Atomicity requires the apparent indivisible execution of *atomic actions*. Atomic actions are also groups of shared memory requests. An atomic action is often a fault tolerance unit for which a system must guarantee that either all or none of the requests execute. Isochrons assume fault freedom. A more important distinction between isochrons and atomic actions involves internal dependences. An atomic action with internal dependences, such as A = B, is a *structured atomic action*. A *flat atomic action* has no internal true (i.e. read/write) dependences. Isochrons are flat atomic actions in fault free systems. Although isochrons are not as powerful as structured atomic actions, isochrons can execute structured atomic actions when used with *split operations* [Wil93].

A split operation divides a write request into two parts. A process uses a split operation to reserve a position in the execution order for a write before it determines the asso-

ciated value. It declares its intention to write a variable with a *sched* request. An *assign* request associates a value with the sched. No value is associated with a sched. When a sched is executed on a copy, then the value of the copy is *unsubstantiated*. When the internal dependences of a structured atomic action are satisfied, the process determines the value to associate with the write. Execution of the corresponding assign *substantiates* the write with this value. Systems that use isochronous techniques for executing structured atomic actions must represent unsubstantiated values and have a method to buffer *unsubstantiated reads*, which are read requests that are executed on unsubstantiated copies. The read is associated with the value and delivered when the write is substantiated.

Most systems provide low level primitives, such as locks, that support techniques to group shared memory requests. The programmer must use these primitives correctly to enforce atomicity. Lock-based techniques use mutual exclusion to execute atomic actions. Isochronous techniques do not rely on mutual exclusion, and thus offer greater concurrency. Several researchers have proposed more efficient lock implementations, such as distributed queues [GVW89, GrT90]. Other systems incorporate fine grain locking capabilities into the coherence protocol [BiD86, RaL96]. Although these techniques can improve lock performance, they do not recover the concurrency lost to mutual exclusion. In addition, lock-based systems generally do not allow pipelining of isochrons or atomic actions, regardless of the enforced consistency semantics.

Shared Regions and the C Region Library (CRL) are DSM systems that enforce atomic access to explicitly allocated data regions [SGZ93, JKW95]. Special region operations delimit atomic accesses to regions in CRL. Explicit coherence requests are required with Shared Regions. The consistency semantics of these systems, which enforce sequen-

tial consistency across regions, are similar to entry consistency. Unlike isochronous techniques, they do not support atomic access of arbitrary combinations of variables.

To enforce atomicity, Transactional Memory (TM) and the Oklahoma Update (OU) modify any coherence protocol that uses an exclusive copy to execute a write request [HeM92, HeM93, SSH93]. In the bus-based TM implementation, a “busy” response locks any coherence unit that an unfinished atomic action reads or writes. The general interconnection network TM implementation only locks coherence units that an unfinished atomic action writes. In OU, a two phase locking strategy groups execution of the write requests of an atomic action. Ultimately, TM and OU enforce atomicity with fine-grain locks, while isochronous techniques do not use locks.

2.2.5. Coherence Mechanisms

We divide mechanisms to maintain coherence into two major categories. The first category involves methods to track the locations and states of copies. Coherence operations, which determine how a protocol distributes the values associated with write requests to the copies, form the second category. Several mechanisms exist for both functions. The appropriate choice of mechanism depends on the reference patterns of shared memory requests, the physical components of the system and the enforced ordering constraints. We present the mechanisms in this section. We discuss reference patterns in Section 2.2.6.

2.2.5.1. Copy Tracking

When a process issues a write request and non-local copies exist, it must send coherence operations to the copies. The protocol can broadcast each coherence operation or multicast it only to existing copies. The interconnection network usually determines the

choice, although other factors, such as memory overhead, influence the decision. We discuss copy tracking methods in this section and coherence operations in Section 2.2.5.2.

Snoopy coherence protocols rely on the inherent serialization of bus accesses to enforce sequential consistency [Goo83, PaP84, RuS84, KEW85, KMR88, TSS88, TCS92]. In these systems, each cache *snoops* the bus for shared memory requests. When it detects a request, the cache controller checks for a local copy of the requested variable. If a local copy exists, the cache takes whatever action the coherence protocol requires.

General interconnection networks do not provide the efficient broadcast mechanism that snoopy protocols require. Although protocols for general interconnection networks can still use broadcasts, most accurately track the locations of all copies. Censier and Feautrier proposed a bit vector directory per coherence unit for accurate copy tracking [CeF78]. Tang made a similar proposal that required more memory overhead [Tan76].

Each directory requires a bit per processing node, which is significant memory overhead in large systems. Several methods reduce directory memory requirements. One method is to use large coherence units. However, this solution can increase false sharing, as we discuss in Section 2.2.7. Reserving regions of the memory space for private data, and thus providing fewer directories, is another simple method [BMR89]. Tamir and Janakiraman propose a dynamic scheme that uses both of these solutions [TaJ92]. This scheme maintains state information for two granularities. Only if the state of the larger coherence unit is shared are directories maintained for the smaller contained units.

Most coherence units have few, if any, copies since most variables are not actively shared [WeG89]. Thus, most directory hardware is wasted with full directory methods. Several methods rely on this observation. In order to reduce the cost for hardware directory entries, these methods incur additional costs when large directories are required.

A number of schemes provide a limited number of hardware pointers. Some protocols broadcast coherence operations if there are more copies than hardware pointers [ArB84]. *Limited directory protocols* restrict the number of copies to the number of pointers [SiH91]. These protocols exchange additional messages and cache misses for directory memory space. *Software extended directories* trap to software routines that use ordinary memory to provide full bit vector directories when the number of copies is large [CKA91, WCF93, ChA94]. Thus, directory memory overhead is exchanged for longer directory accesses. At the extreme, all directory functions are performed in software [GrS95].

Several solutions to the memory overhead problem reorganize the directory hardware. The Scalable Coherent Interface and the Galactica Net use a linked list directory, wherein each copy has a pointer to the next copy [Jam90, WiL92]. This organization avoids directory overflow but increases the latency of coherence operations with the number of copies. Other alternatives use an associative memory. Organizations of this memory include a cache of directories or pointers [GWM90, LiY90]. O'Krafka and Newton use associative memory that replaces software extended directories with hardware [OKN90].

Other methods for locating copies require specific network topologies. The Kendall Square Research Allcache system, the Data Diffusion Machine and the Hector multiprocessor have hierarchical networks, such as trees of buses or rings [HHW90, FBR93, FVS95]. Each subnetwork supports efficient broadcasts. A snoopy mechanism propagates coherence operations up and down the network tree when necessary. Directory information is maintained for entire subnetworks, which significantly reduces memory requirements. Other topology-specific protocols have been designed for a grid of buses in the Wisconsin Multicube and for multistage interconnection networks with modified switches [GoW88, GhS91, YTB92, NaB93].

2.2.5.2. Coherence Actions and Operations

Coherence actions satisfy shared memory requests. If a local copy does not exist then a *miss* occurs and the protocol must locate the most recent version of the variable. Most protocols provide a local copy with a miss response. Performance can improve if a local copy is not always provided [KMR88, CoF89]. If a local copy exists, then read requests are executed on it. The protocol must send a *coherence operation* to each non-local copy to execute a write request. In this section, we discuss coherence operations and actions and the mechanisms to implement them.

2.2.5.2.1. Miss Actions

When a miss occurs, the protocol must locate the most recent version of the variable. Li and Hudak explored several solutions to this problem, which they called the *page management problem* [LiH89]. Our protocols use their fixed distributed manager solution. A hash function determines the location of a *home copy* from the address of the variable. The home copy may not always have the current version, but maintains a pointer to a location that does. Li and Hudak also explored dynamic distributed page manager algorithms. We leave for future work adaptations of these algorithms for our systems.

2.2.5.2.2. Dynamic Protocols

Hardware or *dynamic* protocols use run-time coherence operations. Coherence protocols can execute a write on every copy through *updates* [McC85, TSS88, TKB92, WiL92, DKC93, GDF93, BLV94] or use *invalidations* to provide the writer with an exclusive copy [Goo83, PaP84, KEW85, CoF93, DSR93, SBS93].

The comparative cost of using updates versus invalidations depends on the future requests to the variable, and the optimal choice can vary during the lifetime of a program. Since the future requests are unknown, the coherence problem is similar to the page replacement problem in uniprocessors. *Off-line* algorithms use the optimal choice based on knowledge of future requests. However, systems implement *on-line* algorithms, which do not use future knowledge. Competitive analysis evaluates the performance of on-line algorithms [SlT85]. Let $\mathbf{C_{opt}}$ be the optimal cost and $\mathbf{C_{on\text{-}line}}$ the cost of an on-line algorithm. The on-line algorithm is *competitive* if $\mathbf{C_{on\text{-}line}} \leq \mathbf{c} * \mathbf{C_{opt}}$ for any set of requests where $\mathbf{c}$ is a constant called the *competitive coefficient.* An on-line algorithm is *strongly competitive* if $\mathbf{c}$ is the minimum possible competitive coefficient.

Karlin, et al. identified a strongly competitive algorithm for the coherence problem in bus-based architectures with direct-mapped caches and developed protocols with low competitive coefficients for other cache structures [KMR88]. In these protocols, processes discard local copies if the cost of updates received between local requests to the coherence unit equals the cost of a miss operation. They assumed requests were sequential and used the number of bus cycles to service all requests as the cost of an algorithm.

Many researchers have proposed update protocols for general interconnection networks that discard copies if the number of updates between local requests exceeds some threshold value [WiL92, BLV94, DDS94, SSR95]. Researchers frequently call these protocols competitive although they do not provide competitive analysis. Many researchers have explored competitive algorithms for page migration and replication of read-only data [BlS89, BGW89, BFR92, ABF93, Wes94, BFR95].

Adaptive protocols identify and exploit specific reference patterns dynamically. For example, some protocols adapt to *migratory variables*, which exhibit periods during

which only one process issues requests to them [CoF93, SBS93]. These protocols provide an exclusive copy for any request only if the coherence unit exhibits this pattern.

2.2.5.2.3. Software-Assisted or Static Protocols

Software-assisted or *static* protocols ensure an exclusive copy exists when a write is issued, and thus eliminate coherence operations. Compiler inserted coherence directives invalidate other copies before a write occurs in these protocols. These protocols generally incur more misses but do not require hardware support since they do not track copies.

Simple static protocols do not replicate shared variables or invalidate all copies at major program boundaries, such as a critical section exit [OwA89]. Many techniques that invalidate copies when a request would access stale data improve this method [CKM88, SGZ93]. These methods often need special hardware to detect when a copy contains stale data, which blurs the hardware/software distinction [ChV88, PST91, MiB92, ChY96].

Static analysis and other software techniques can improve the performance of dynamic coherence protocols, further blurring the distinction. Skeppstedt and Stenstrom use compiler directives with invalidation protocols to obtain an exclusive copy on a read request that precedes a write request [SkS94]. Mounes-Toussi and Lilja select invalidations or updates for each write request based on static analysis [MoL95].

Static methods can predict the expected reference pattern of a coherence unit to improve protocol performance [VeF92, DCZ96]. Many methods rely on accurate prediction of reference patterns by the programmer to improve performance. Munin uses programmer hints, while Tempest integrates the coherence protocol into the application, which allows the programmer to tune the protocol to the reference patterns of the application [BCZ90, CBZ91, FLR94].

2.2.5.2.4. Hardware and Software DSM

An additional hardware and software distinction arises in DSM systems. Most DSM systems modify the virtual memory mechanism to implement coherence operations [LiH89, LiS89, FlP89]. *Software* DSM systems implement the coherence protocol entirely in software, either in the operating system kernel or user level routines. *Hardware* DSM systems increase performance with special purpose hardware [CDK94, KoS95, IDF96, BKP96, RPW96, ZIL96]. Several systems link hardware coherent systems with software DSM systems, which blurs this hardware/software distinction [CDK94, ENC96, YKA96].

2.2.6. Locality and Coherence

Caching techniques improve uniprocessor performance because of *locality*, the tendency of future requests to reflect previous requests. Most programs exhibit *temporal locality*, the tendency of programs to request recently requested variables again. Larger cache lines improve uniprocessor performance since most programs exhibit *spatial locality*, the tendency to request variables with addresses near recently requested variables. Creating copies dynamically in shared memory systems improves performance because of locality. In this section, we discuss the reference patterns of shared memory programs.

Assuming a fixed total amount of local memory, the number of local copies is a function of coherence granularity. Smaller coherence units allow more copies, which increases the amount of temporal locality that the system can exploit. Larger coherence units exploit spatial locality. Goodman observed that spatial locality of write accesses decreases the cost of coherence maintenance as coherence granularity increases, although reduced exploitation of temporal locality eventually outweighs this benefit [Goo83].

Shared memory reference patterns not only include the locality exhibited by the programs of individual processes, but also interactions between the requests of the processes. Requests by different processes to the same variable determine a *true sharing* reference pattern. *False sharing*, which we discuss in Section 2.2.7, occurs when two processes access the same coherence unit although they request distinct variables.

Agarwal and Gupta proposed "*processor locality* - the tendency of a processor to access a block repeatedly before an access from another processor" as a general characterization of sharing [AgG88]. *Write run lengths*, a measure of processor locality, indicate whether an invalidation or an update protocol would provide better performance [EgK88]. The length of a write run is the number of consecutive write requests to a coherence unit by one process before any read or write request by another process. Long write runs favor invalidation protocols, while short write runs favor update protocols. Short write runs often correspond to *ping-ponging* under an invalidation protocol. In this situation, two processes alternate write requests, repeatedly invalidating the other copy. Most reference studies provide little evidence of long write runs, with average lengths generally under 2 on small systems [EgK88, FuP93].

Other proposed types of locality suit particular systems. *Cluster locality* characterizes the pattern where a proper subset of processes actively share a variable, as seems likely for a number of regular parallel algorithms [PiB92]. *Multigrain locality* extends cluster locality to indicate benefits of using multiple coherence granularities [YKA96].

Other researchers identify specific sharing patterns that suit certain coherence protocols [WeG89, BCZ90a]. We discussed migratory variables in Section 2.2.5.2.2. *Synchronization variables* are locks implemented in shared memory, which are usually replaced with special synchronization primitives. Update protocols suit the pattern of

mostly read variables for which several processes read each write. Producer/consumer variables and variables that are frequently read and written by multiple processes also suit variations of update protocols. The Shrimp project classifies variables and coherence units by the number of producers and consumers [ISL96]. Coherence unit sharing patterns, which include false sharing, determine system performance.

2.2.7. False Sharing

False sharing, which is not easily detected, reduces the performance of shared memory systems. The systems incur the cost of coherence maintenance although the ordering constraints do not require it since processes access distinct variables. Measuring and reducing false sharing are important areas of research for shared memory systems.

False sharing metrics allow the evaluation of methods to reduce false sharing. Measuring false sharing or its cost is difficult because spatial locality can reduce the number of coherence operations required as the coherence granularity increases. Bolosky and Scott conclude that separating the effects of false sharing from other performance effects related to coherence granularity may be impossible [BoS93]. Most researchers use metrics that categorize the causes of misses, but these metrics only apply to invalidation protocols [EgJ91, DSR93, TLH94, JeE95]. Others have proposed more general metrics. One metric compares the number of processes that access a variable to the number of processes that access the coherence unit that contains the variable [KLE93]. Hyde and Fleisch identify sharing patterns in reference strings with regular expressions [HyF96]. They measure unnecessary coherence operations based on several false sharing patterns.

Several methods reduce false sharing. Small coherence units can substantially reduce false sharing in DSM systems [BoS92, SFL94]. The programmer or compiler can

allocate distinct program objects to different coherence units to prevent false sharing [BFS89, TLH94, JeE95]. This approach suffers from internal fragmentation and does not reclaim any temporal locality lost due to the larger coherence units. Another compiler-based approach to reduce false sharing restructures data to group variables accessed as a unit (essentially isochronously) into the same coherence units [BFS89, EgJ91, JeE95]. Adjustable coherence unit sizes, chosen dynamically or at compile time, also reduce false sharing and exploit spatial locality [DuL92, DSR93, SGT96].

Lazy release consistent protocols can delay and combine invalidations, which can reduce false sharing [KCZ92, DSR93, IDF96, ZIL96]. Delayed invalidations can eliminate misses to other variables in the coherence unit. Combined invalidations reduce network congestion, thus decreasing latency. Most release consistent update protocols combine updates, which can reduce the cost of true and false sharing [GDF93, BLV94].

2.3. Logical Time

A *logical time system* (LTS) is a method for numbering the events of a system based on causality [Lam78]. We use isotach LTS's to maintain coherence. In this section, we discuss causality. We then present scalar and vector clocks, two common LTS's. We conclude with applications of logical time to coherence maintenance.

2.3.1. Causality and Logical Time

Logical time systems attempt to capture *causality*. Let **a** and **b** be two events of a system. If **a** determines or influences the outcome or occurrence of **b**, then **a** *causes* **b** or **a** ⇒ **b**. For example, if **a** and **b** are respectively the send and receive events of a message,

then $\mathbf{a} \Rightarrow \mathbf{b}$ since a message must be sent in order to be received. Causality, which is an irreflexive partial order over the events of a system, depends on semantic information regarding the events.

The *happens before* relation formalizes the concept of time in distributed systems [Lam78]. The transitive closure of two rules determines if **a** *happens before* **b** or $\mathbf{a} \rightarrow \mathbf{b}$:

HB1: if **a** and **b** are events in the same process and **a** occurs before **b**, then $\mathbf{a} \rightarrow \mathbf{b}$;
HB2: if **a** and **b** are respectively the send and receive events of a message, then $\mathbf{a} \rightarrow \mathbf{b}$.

If neither $\mathbf{a} \rightarrow \mathbf{b}$ nor $\mathbf{b} \rightarrow \mathbf{a}$ then **a** and **b** are *concurrent*. A logical time system is *consistent* with the *happens before* relation if $\mathbf{a} \rightarrow \mathbf{b}$ implies $\mathbf{t_a} < \mathbf{t_b}$, where $\mathbf{t_a}$ and $\mathbf{t_b}$ are the logical times that it assigns to events **a** and **b**. A *strongly consistent* logical time system ensures that $\mathbf{a} \rightarrow \mathbf{b} \Leftrightarrow \mathbf{t_a} < \mathbf{t_b}$. Most logical time systems are consistent or strongly consistent with the *happens before* relation.

The *happens before* relation captures potential causality since $\mathbf{a} \Rightarrow \mathbf{b}$ implies $\mathbf{a} \rightarrow \mathbf{b}$, but $\mathbf{a} \rightarrow \mathbf{b}$ does not imply $\mathbf{a} \Rightarrow \mathbf{b}$. Consider an execution of the statement A = B + C. Let $\mathbf{issue_B}$ and $\mathbf{issue_C}$ be the issue events of the read requests to B and C, respectively. One of these events must occur before the other. Without loss of generality, let $\mathbf{issue_B}$ occur first. Then $\mathbf{issue_B} \rightarrow \mathbf{issue_C}$ by **HB1**. However, $\mathbf{issue_B} \Rightarrow \mathbf{issue_C}$ is not true since the opposite order does not change the outcome or occurrence of either event.

2.3.2. Logical Time Systems

Researchers have proposed several logical time systems. A process represents its logical time with a single scalar clock in Lamport's original logical time system [Lam78]. The clock is incremented for each event of the process. When a process sends a message, it includes its current logical time. When a process receives a message, it ensures that its

logical time is greater than the send time of the message. This LTS is consistent with the *happens before* relation, although it is not strongly consistent.

LTS's that are strongly consistent with the *happens before* relation incur significant overhead. Fidge, Mattern and Schmuck independently proposed logical time systems that use vector clocks [Sch88, Mat89, Fid91]. For each local event, process **i** increments the $\mathbf{i}^{th}$ component of its vector clock. Each message again includes the local logical time of its send event. The logical time of a process that receives a message then becomes the component-wise maximum of its logical clock and the send time of the message. Charron-Bost demonstrated that **N**, the number of processes in the system, is the minimum length vector for a strongly consistent LTS [Cha91]. This result demonstrates that a strongly consistent LTS has overhead of O(**N**) per logical clock or message. Several approaches reduce this overhead, but either strong consistency is lost or memory overhead remains high [SiK92, JaJ94, RaS96].

2.3.3. Coherence Maintenance and Logical Time

Consistency semantics that are based on causality include causal memory and extended causal memory. *Causal memory* allows any execution that is consistent with a causal relationship defined by program order and write/read dependences [ANK94]. Causal memory is weaker than sequential consistency since processes may observe concurrent writes in different orders. Researchers have investigated program restrictions that ensure executions with causal memory are sequentially consistent [ANK94, RaS95]. *Extended causal memory* weakens causal memory. It allows executions that are consistent with a causal relationship that reflects synchronization requests [JoA94]. Hybrid consistency is similar to extended causal memory [AtF92].

Many coherence protocols use logical time to enforce weak consistency semantics. Most implementations of causality-based consistency semantics are based on vector logical time. Some causal memory implementations require *full replication*, which locates a static copy at each node [ANK94, Fri95]. Using a vector timestamp per copy allows dynamic replication [AHJ91]. Extended causal memory implementations that allow dynamic replication have similar overhead [JoA94]. Lazy release consistency (LRC) uses a causal relationship to delay coherence operations [KCZ92]. LRC protocols also associate a vector timestamp with each copy. These timestamps detect stale copies based on lock acquisition timestamps [ACD96, IDF96, ZIL96].

A sequentially consistent protocol can combine vector clocks with a central shared memory process to establish a total order for write operations [MRS93]. Many sequentially consistent coherence protocols use atomic broadcasts to distribute writes, which guarantees that processes receive messages in the same total order [ABM93, AtW94, Fri95]. Many systems implement atomic broadcast, which the protocols use to enforce write atomicity, with logical time. Atomic broadcast systems often focus on fault tolerance. We do not discuss these systems further.

2.4. Isotach Systems

Our coherence protocols rely on the properties of an isotach LTS. The isotach invariant distinguishes isotach LTS's from other LTS's. This invariant allows the logical times of causally related events to be anticipated despite stochastic real time message delays.

2.4.1. Isotach Logical Time

We present isotach LTS's in this section. Previously, consistency with the *happens before* relation has been required for isotach LTS's. We define *potential causality* in Chapter 3. This new relation captures causality more accurately than the *happens before* relation for systems that use an intermediate process to send and receive messages. We require consistency with *potential causality* for isotach LTS's, which always use such an intermediate process.

Isotach logical times are lexicographically ordered n-tuples, of which the most significant component is always the *pulse* component. We use two representations in this thesis, although others are possible. One representation uses a three-tuple, (**pulse**, **pid-rank**, **issue-rank**) for logical times. The **pid-rank** component concatenates the node id with the local process id to form a unique system-wide process id. The final component is a count of the messages issued by the process. Shared memory systems can also use a four-tuple representation, (**pulse**, **var-name**, **pid-rank**, **issue-rank**), where **var-name** is the shared memory location accessed by the message. Throughout this thesis, we assume a three-tuple representation unless otherwise noted.

Isotach LTS's are characterized by the *isotach invariant*. This invariant requires that if the send event of a message has pulse component **i**, then the receive event of the message must have pulse component $\mathbf{i} + \mathbf{d_m}$, where $\mathbf{d_m}$ is the *logical distance* that the message travels. All other components of the receive time must equal those of the send time. In other words, the message travels at unit speed. Thus, given the logical send time of a message, a process can anticipate the logical receive time. We use the shorthand $\mathbf{t} + \mathbf{c}$ to indicate adding the constant **c** to the pulse component of logical time **t**. Thus, the iso-

tach invariant requires that $\mathbf{t_r} = \mathbf{t_s} + \mathbf{d_m}$, where $\mathbf{t_s}$ and $\mathbf{t_r}$ are, respectively, the logical send and receive times of the message.

Isotach geometry can be very different from planar geometry. Let $\mathbf{d_{a,\,b}}$ be the logical distance from **a** to **b**, where **a** and **b** are nodes of an isotach network. Isotach network algorithms exist in which $\mathbf{d_{a,\,b}}$ may not equal $\mathbf{d_{b,\,a}}$. Thus, the order of the subscripts conforms to the direction in which the logical distance is measured throughout this thesis.

Extensible isotach networks allow the anticipation of the logical times of the events of *response messages*. The execution of another message can generate a response message. If $\mathbf{t_{s'}}$ is the logical send time of a response in an extensible network, then $\mathbf{t_{s'}} = \mathbf{t_r} + \mathbf{c}$, where $\mathbf{t_r}$ is the logical receive time of the original message. A process can use the isotach invariant and the logical send time of the original message to control the logical times of response message events. The logical delay, **c**, of an *immediate response* is zero. *Delayed responses*, in which $\mathbf{c} > 0$, can reduce the cost of providing extensibility. Consistency with *potential causality* prohibits $\mathbf{c} < 0$.

2.4.2. Isotach Networks

We discuss *isotach networks*, which realize isotach LTS's, in this section. Isotach network algorithms exist that accommodate arbitrary network topologies. Many of these algorithms require only local information and avoid any expensive global agreement methods. In this section, we provide several definitions that we use throughout this thesis. First, we define the physical components of an isotach network. Then, we discuss several important concepts for the implementation of isotach logical time. We conclude with the levels of logical time message service available in isotach networks.

Physically, a shared memory system is a collection of network elements linked by some interconnection network (IN), including a bus. The IN allows messages to be sent between network elements. Example IN's are a bus, a crossbar network, a multistage interconnection network (MIN), and an arbitrarily connected switch-based network. In this thesis, we assume that the IN is switch-based. Formally, the *physical topology* of a shared memory system is a connected graph, (**V**, **E**), where **V** is the set of network elements and switches and **E** is the set of message links.

Each network element is a *processing element* (PE) or a *memory module* (MM). An MM does not issue shared memory requests. Each process is located at a PE. A *combined* PE includes local memory that acts as an MM. The home copy of every variable is located at either an MM or a combined PE. The intermediate process that sends and receives all messages of a network element is located at its *switch interface unit* (SIU), an intermediate entity that manages logical time for the element. Processes use properties that the system guarantees, such as ordering constraints, instead of actively using logical time. In isotach shared memory systems, each process issues its requests to the local SIU, which is the *issuing SIU* of the request. The issuing SIU enforces ordering constraints by scheduling the logical times of events of the request.

The *routing path* of a message is the set of network nodes (elements or switches) through which it passes. A *fixed routing path* is known to the sender at the time that the message is sent. A *static fixed routing path* requires that every message between a given sender/receiver pair has the same routing path. A *dynamic fixed routing path* is chosen at the time the message is sent. The cost of using dynamic fixed routing paths in isotach networks depends, in part, on the properties that the system provides. A *dynamic routing path* is determined as the message travels through the network. Most applications of isotach

logical time systems require that the sender know the logical distance that the message will travel. This requirement complicates the use of dynamic routing paths in isotach networks. We assume static fixed routing paths.

Logical distance is a central concept of isotach networks, which ensure that each routing path has a constant logical distance. The *routing distance* of a message is the number of intermediate nodes on its routing path. Unless otherwise stated, we assume that the logical distance of a message is its routing distance. We assume all *virtual messages*, for which the sending node is the destination node, such as a message between collocated processes, travel zero logical distance. The maximum logical distance in an isotach network is its *logical diameter*, $\mathbf{D}$.

We define various *levels of message service* in isotach networks. Each level provides a guarantee for the logical receive time of a message based on its logical send time. All isotach networks must provide the strictest level of service, which requires conformance to the isotach invariant. For a message that travels logical distance $\mathbf{d_m}$ with logical send and receive times $\mathbf{t_s}$ and $\mathbf{t_r}$, respectively, we have identified several useful levels of service in the following hierarchy:

(0) No guarantee: any logical receive time is allowed

(1) Bounded: $\mathbf{t_s} \le \mathbf{t_r} \le (\mathbf{t_s} + \mathbf{d_m})$

(2) Constrained: for sender chosen logical time $\mathbf{t_l}$ such that $\mathbf{t_s} \le \mathbf{t_l} < (\mathbf{t_s} + \mathbf{d_m})$,
$\mathbf{t_l} \le \mathbf{t_r} \le (\mathbf{t_s} + \mathbf{d_m})$

(3) Standard: $\mathbf{t_r} = (\mathbf{t_s} + \mathbf{d_m})$

The standard level of service enforces the isotach invariant. Any message that uses this level of service satisfies the restrictions of the other service levels. However, performance for many applications can improve if an isotach network exploits the less strict requirements of the other levels.

2.4.3. Isotach Applications

In this section, we review applications of isotach logical time. We begin with applications that were proposed prior to the formal definition of isotach logical time. We then describe other applications of isotach logical time besides coherence maintenance. We conclude with a brief introduction of previous isotach-based coherence protocols.

Some applications of isotach logical time precede the theory of isotach logical time. In the Fluent machine, a concurrent read, concurrent write PRAM emulation uses an algorithm that implements four-tuple isotach logical time [Ran87, RBJ88, Ran89]. Awerbuch's synchronizer algorithms allow an asynchronous system to execute SIMD graph algorithms [Awe85]. These algorithms all essentially implement a single component isotach logical time system. A single component isotach logical time system also supports efficient barrier implementations [BGS89].

Significant applications of isotach logical time in message passing systems include causal message delivery and determining consistent cuts. If $\mathbf{s_1} \rightarrow \mathbf{s_2}$, where $\mathbf{s_1}$ and $\mathbf{s_2}$ are the send events of two messages that process **i** receives, then *causal message delivery* requires that $\mathbf{r_1} \rightarrow \mathbf{r_2}$, where $\mathbf{r_1}$ and $\mathbf{r_2}$ are the corresponding receive events. Causal message delivery is a consequence of the isotach invariant in logical topologies that maintain the triangle inequality [Wil93]. A *consistent cut*, **C**, is a subset of the events of a distributed system such that $\forall\ \mathbf{b} \in \mathbf{C}$, if $\mathbf{a} \rightarrow \mathbf{b}$ then $\mathbf{a} \in \mathbf{C}$ [Mat93]. Consistent cuts simplify checkpointing and the detection of properties such as deadlock or termination. Each pulse of isotach logical time represents a consistent cut [Wil97].

There are several applications of isotach logical time in shared memory systems besides coherence maintenance. Williams and Reynolds presented isotach networks that

efficiently combine isochrons [WiR95]. An asynchronous production systems algorithm that uses an isotach logical time system supports multiple concurrent rule firings. This algorithm can exploit most of the available concurrency in the rule sets [Sri96].

The *delta coherence protocols* are the isotach-based family of coherence protocols. Williams developed two delta update coherence protocols for equidistant networks [Wil93]. In addition to developing several new members of this protocol family, we combine her protocols and extend them to non-equidistant networks.

2.5. Chapter Summary

This thesis explores coherence maintenance based on logical time. We define coherence maintenance as the concurrency control problem in systems that allow replication. We reviewed previous research in coherence maintenance, including other logical time approaches. We discussed causality, the *happens before* relation and logical time systems. We concluded with isotach logical time systems and their properties.

Chapter 3:
Potential Causality

3.1. Introduction

In this chapter, we define *potential causality*, a new relation over the events of any distributed system in which intermediate processes send and receive all messages. As with the *happens before* relation, *potential causality* is a partial order that includes (**a**, **b**) if **a** causes **b** (**a** $\Rightarrow$ **b**). *Potential causality* more accurately captures causality than the *happens before* relation (**a** $\rightarrow$ **b**) since both cover causality and if **a** *potentially causes* **b** (**a** $\Rsh$ **b**) then **a** $\rightarrow$ **b**, while the converse need not hold. Thus, consistency with *potential causality* allows greater flexibility in assigning logical times since it requires a LTS to enforce no more non-causal relationships than consistency with the *happens before* relation.

Previously, isotach logical time systems were required to be consistent with the *happens before* relation by definition. Now, they must be consistent with *potential causality.* The greater flexibility in assigning timestamps allowed by *potential causality* supports causally consistent implementations that need not be consistent with the *happens before* relation, including current prototype systems [Reg97, WiR97]. In Chapter 4, we present a new isotach network algorithm that uses the flexibility provided by *potential causality* to allow greater flexibility for logical distances than previous algorithms.

3.2. System Model

We assume an intermediate *messaging process* sends and receives all messages for each of one or more collocated user processes. Each user process communicates with all

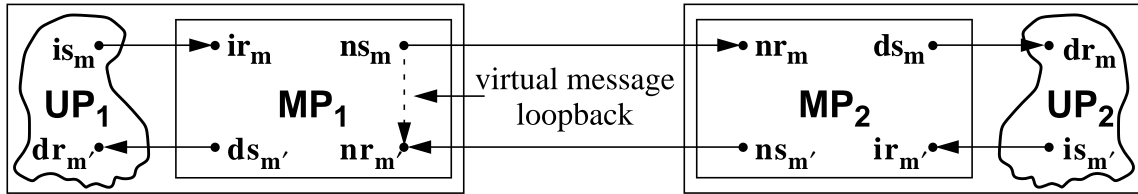


Figure 3.1: System Model and Message Notation

other user processes, including collocated ones, through exactly one messaging process. Many systems, such as ISIS, conform to this model since they use messaging processes [BSS91]. The SIU serves this role in isotach systems.

Figure 3.1 shows the events for messages $\mathbf{m}$ and $\mathbf{m'}$. User process $\mathbf{UP_1}$ ($\mathbf{UP_2}$) sends $\mathbf{m}$ ($\mathbf{m'}$) to user process $\mathbf{UP_2}$ ($\mathbf{UP_1}$) through their associated messaging processes, $\mathbf{MP_1}$ ($\mathbf{MP_2}$) and $\mathbf{MP_2}$ ($\mathbf{MP_1}$). The send event of $\mathbf{m}$ in $\mathbf{UP_1}$ is its *issuing send event*, $\mathbf{is_m}$, while the *corresponding* receive event in $\mathbf{MP_1}$ is its *issuing receive event*, $\mathbf{ir_m}$. The send event of $\mathbf{m}$ in $\mathbf{MP_1}$ is its *network send event*, $\mathbf{ns_m}$, while the *corresponding* receive event in $\mathbf{MP_2}$ is its *network receive event*, $\mathbf{nr_m}$. The send event of $\mathbf{m}$ in $\mathbf{MP_2}$ is its *delivering send event*, $\mathbf{ds_m}$, while the *corresponding* receive event in $\mathbf{UP_2}$ is its *delivering receive event*, $\mathbf{dr_m}$. (Message $\mathbf{m'}$ has the same events as $\mathbf{m}$, but in the reverse direction.) Thus, a message $\mathbf{m}$ has three pairs of *corresponding interprocess events*: ($\mathbf{is_m}$, $\mathbf{ir_m}$), ($\mathbf{ns_m}$, $\mathbf{nr_m}$) and ($\mathbf{ds_m}$, $\mathbf{dr_m}$); and it has two pairs of *corresponding messaging process events* (events internal to a single messaging process): ($\mathbf{ir_m}$, $\mathbf{ns_m}$) and ($\mathbf{nr_m}$, $\mathbf{ds_m}$). Our model assumes exactly one corresponding event for each message event. We could extend our model to allow multiple corresponding events. We do not pursue that extension in this thesis.

We assume a local messaging process sends a virtual message to itself for any message between collocated user processes. The key difference between virtual messages and other messages, such as $\mathbf{m}$, is that virtual messages loop back through the associated messaging process, as indicated by the dashed line in Figure 3.1.

3.3. Defining Potential Causality

The *happens before* relation relates all events that occur in the same process since Lamport used no knowledge about their causal relations. For *potential causality*, we use knowledge of the causal relations of events within the same messaging process. The transitive closure of the following three rules determines if **a** potentially causes **b** (**a** ↱ **b**):

PC1: if **a** and **b** are events in the same user process and **a** occurs before **b**, then **a** ↱ **b**;
PC2: if **a** and **b** are corresponding interprocess send and receive events, then **a** ↱ **b**;
PC3: if **a** and **b** are corresponding messaging process receive and send events, then **a** ↱ **b**;

Since we assume no knowledge about the causal relations of events within a user process, if **a** occurs before **b** in the same user process, then **a** could cause **b**. We capture this potential causal relationship in **PC1**. We know corresponding events are causally related. We capture the causal relation of corresponding interprocess events in **PC2** and of corresponding messaging process events in **PC3**. Since all interprocess communication is by messages, a sequence of causally related messages, $<\mathbf{m_0},\ldots,\mathbf{m_n}>$, must link any causally related events, **a** and **b**, that do not occur within the same user process. Since the messages are causally related, $\mathbf{dr}_{\mathbf{m_i}}$ must occur before $\mathbf{is}_{\mathbf{m_{i+1}}}$ in the same user process and, thus, taking the transitive closure of our three rules ensures **a** ↱ **b** if $\mathbf{a} \Rightarrow \mathbf{b}$. Events **a** and **b** are *concurrent* if neither **a** ↱ **b** nor **b** ↱ **a**.

Potential causality refines the *happens before* relation, i.e. **a** ↱ **b** implies $\mathbf{a} \rightarrow \mathbf{b}$, but $\mathbf{a} \rightarrow \mathbf{b}$ does not necessarily imply **a** ↱ **b**. Since Lamport did not distinguish messaging processes from other processes, $\mathbf{a} \rightarrow \mathbf{b}$ if **a** occurs before **b** within the same messaging process. For any corresponding messaging process events, the receive event must occur before the send event. Thus, for any events **a** and **b**, **a** ↱ **b** implies $\mathbf{a} \rightarrow \mathbf{b}$. We now give an example in which $\mathbf{a} \rightarrow \mathbf{b}$ but **a** does not *potentially cause* **b**. Let **b** be $\mathbf{ns_m}$, the network

send event of the message $\mathbf{m}$. Let $\mathbf{a}$ be an event of the same messaging process as $\mathbf{b}$ such that $\mathbf{a}$ occurs before $\mathbf{b}$ (thus, $\mathbf{a} \rightarrow \mathbf{b}$), but after $\mathbf{ir_m}$, the corresponding messaging process event of $\mathbf{b}$. Since $\mathbf{a}$ occurs after $\mathbf{ir_m}$, $\mathbf{a}$ cannot potentially cause $\mathbf{ir_m}$. Further, any event that potentially causes $\mathbf{b}$ must potentially cause $\mathbf{ir_m}$ since $\mathbf{ir_m}$ is the only event that potentially causes $\mathbf{b}$ without using any transitive applications of our three rules. Thus, $\mathbf{a}$ does not potentially cause $\mathbf{b}$, although $\mathbf{a} \rightarrow \mathbf{b}$.

3.4. Consistency with Potential Causality

A logical time system is *consistent* with *potential causality* if $\mathbf{a} \Rsh \mathbf{b}$ implies $\mathbf{t_a} \leq \mathbf{t_b}$, where $\mathbf{t_a}$ and $\mathbf{t_b}$ are the logical times that it assigns to events $\mathbf{a}$ and $\mathbf{b}$. We allow equality in order to accommodate immediate responses and virtual messages in isotach systems. Allowing equality also accommodates isotach network algorithms that support a logical distance of zero between two distinct nodes, such as the flex algorithm that we present in Chapter 4. The possibility of a chain of equality allows an LTS that is consistent with *potential causality* but is subject to a form of deadlock. Later, we will show an effective procedure for detecting isotach systems that have this problem. If a logical time system is consistent with *potential causality*, then it is consistent with causality since $\mathbf{a} \Rightarrow \mathbf{b}$ implies $\mathbf{a} \Rsh \mathbf{b}$ and therefore $\mathbf{t_a} \leq \mathbf{t_b}$ if $\mathbf{a} \Rightarrow \mathbf{b}$.

A *network logical time system* (Net LTS) is an LTS that only numbers network events. Thus, a Net LTS numbers the network events, $\mathbf{ns_m}$ and $\mathbf{nr_m}$, of the message $\mathbf{m}$ but not its other events: $\mathbf{is_m}$, $\mathbf{ir_m}$, $\mathbf{ds_m}$, and $\mathbf{dr_m}$. We use isotach Net LTS's in this thesis.

We will show that any Net LTS is consistent with *potential causality* if it assigns timestamps that conform to these conditions:

C1: For any message $\mathbf{m}$, $\mathbf{t}_{\mathbf{ns_m}} \le \mathbf{t}_{\mathbf{nr_m}}$.
C2: If $\mathbf{dr_m}$ and $\mathbf{is_{m'}}$ are events of the same user process such that $\mathbf{ds_m}$ occurs in the associated messaging process before $\mathbf{ir_{m'}}$, then $\mathbf{t}_{\mathbf{nr_m}} \le \mathbf{t}_{\mathbf{ns_{m'}}}$.

Any logical time system that is consistent with *potential causality* must conform to **C1** since $\mathbf{ns_m} \rightarrow \mathbf{nr_m}$ by **PC2**. **C2** ensures the Net LTS assigns logical times that are consistent with **PC1** and **PC3**. If the messaging process could directly observe the order of events in the associated user process, we could substitute the following rule for **C2**: if $\mathbf{dr_m}$ and $\mathbf{is_{m'}}$ are events of the same user process such that $\mathbf{dr_m}$ occurs before $\mathbf{is_{m'}}$, then $\mathbf{t}_{\mathbf{nr_m}} \le \mathbf{t}_{\mathbf{ns_{m'}}}$. Since we assume that the messaging process cannot observe events in the user process, we must use a conservative rule. **C2** is conservative since it requires $\mathbf{t}_{\mathbf{nr_m}} \le \mathbf{t}_{\mathbf{ns_{m'}}}$ even when $\mathbf{is_{m'}}$ occurs before $\mathbf{dr_m}$ in the user process and, thus, $\mathbf{nr_m}$ does not potentially cause $\mathbf{ns_{m'}}$. We show that **C1** and **C2** are sufficient to ensure consistency with *potential causality*:

Theorem 3.1: A Net LTS is consistent with *potential causality* if the logical times that it assigns conform to **C1** and **C2**.

Proof: Let $\mathbf{a} \rightarrow \mathbf{b}$ for two network events $\mathbf{a}$ and $\mathbf{b}$. Since $\mathbf{a} \rightarrow \mathbf{b}$, there exists a sequence of messages $\langle \mathbf{m_0}, \ldots, \mathbf{m_n} \rangle$ such that $\mathbf{a}$ is either $\mathbf{ns}_{\mathbf{m}_0}$ or $\mathbf{nr}_{\mathbf{m}_0}$, $\mathbf{b}$ is either $\mathbf{ns}_{\mathbf{m}_n}$ or $\mathbf{nr}_{\mathbf{m}_n}$ and $\mathbf{is}_{\mathbf{m}_{i+1}}$ occurs after $\mathbf{dr}_{\mathbf{m}_i}$ in the same user process for each $\mathbf{i}$. Since $\mathbf{is}_{\mathbf{m}_{i+1}}$ occurs after $\mathbf{dr}_{\mathbf{m}_i}$, $\mathbf{ir}_{\mathbf{m}_{i+1}}$ occurs after $\mathbf{ds}_{\mathbf{m}_i}$ in the associated messaging process and, thus, $\mathbf{t}_{\mathbf{nr}_{\mathbf{m}_i}} \le \mathbf{t}_{\mathbf{ns}_{\mathbf{m}_{i+1}}}$ by **C2** for each $\mathbf{i}$. Since $\mathbf{t}_{\mathbf{ns}_{\mathbf{m}_i}} \le \mathbf{t}_{\mathbf{nr}_{\mathbf{m}_i}}$ by **C1**, $\mathbf{t}_{\mathbf{ns}_{\mathbf{m}_0}} \le \mathbf{t}_{\mathbf{nr}_{\mathbf{m}_0}} \le \mathbf{t}_{\mathbf{ns}_{\mathbf{m}_n}} \le \mathbf{t}_{\mathbf{nr}_{\mathbf{m}_n}}$. Therefore, $\mathbf{t_a} \le \mathbf{t_b}$ if $\mathbf{a} \rightarrow \mathbf{b}$ and the Net LTS is consistent with *potential causality*. **QED**

Often, we can show the following condition for a Net LTS more easily than **C2**:

C2′: If $\mathbf{nr_m}$ occurs before $\mathbf{ns_{m'}}$ in the same messaging process, then $\mathbf{t}_{\mathbf{nr_m}} \le \mathbf{t}_{\mathbf{ns_{m'}}}$.

Since **C2′** is more conservative, we can use **C2′** in place of **C2**:

Corollary 3.1: A Net LTS is consistent with *potential causality* if the logical times that it assigns conform to **C1** and **C2′**.

Proof: If $\mathbf{ds_m}$ occurs before $\mathbf{ir_{m'}}$ in the same messaging process, then $\mathbf{nr_m}$ must occur before $\mathbf{ns_{m'}}$ since $\mathbf{nr_m}$ must occur before $\mathbf{ds_m}$ and $\mathbf{ns_{m'}}$ must occur after $\mathbf{ir_{m'}}$. Thus, a Net LTS conforms to **C2** if it con-

> forms to **C2′**. Therefore, any Net LTS that assigns logical times that conform to **C1** and **C2′** is consistent with *potential causality* by Theorem 3.1. **QED**

A messaging process can conform to **C2** or **C2′** without coordination with other messaging processes since **C2** and **C2′** are local conditions. We demonstrate in Chapter 4 that both Theorem 3.1 and Corollary 3.1 allow greater flexibility in assigning logical times than consistency with the *happens before* relation allows.

3.5. Chapter Summary

We defined *potential causality*, a new relation over the events of any distributed system that uses messaging processes, and presented conditions that ensure a logical time system is consistent with this relation. Isotach logical time systems must be consistent with *potential causality*, as well as enforce the isotach invariant. *Potential causality* allows greater flexibility for assigning logical times than the *happens before* relation since concurrent events can occur in the same messaging process. As we demonstrate in the next chapter, prototype isotach systems and our flex isotach network algorithm require this flexibility since the *happens before* relation is too strict for them.

Chapter 4:
Flexibility for Logical Distances

4.1. Introduction

This chapter presents our *flex algorithm*, the first isotach network algorithm that can assign different logical time latencies to different links. This new algorithm generalizes the isonet network algorithm [RWW97]. The isonet algorithm assumed that the logical distance of each routing path must equal the number of switches on the routing path, i.e. each switch added a cost of one unit of logical distance. The flex algorithm allows each switch to add a cost of any non-negative integer.

We expect that the flexibility in assigning logical latencies that the flex algorithm provides will improve the performance of isotach systems in which the real time latencies of links vary significantly. In an isotach network, the logical time latency of a message that uses the standard level of service equals the logical distance that the message travels. Assuming that there is little variation in the real time latency of logical pulses, the real time latency of the message is proportional to its logical distance. Thus, we expect the best performance from isotach networks in which logical distances reflect the real time latency of the underlying hardware. We leave confirmation of the hypothesis that the flex algorithm will improve performance for future work.

In this chapter, we show that the flex algorithm correctly implements an isotach Net LTS. Also, we present a Petri net model of the algorithm that allows us to determine easily if the pulse component of logical time will ever stop for a given instance of the algorithm. Finally, we show that Awerbuch's β-synchronizer [Awe85] is an instance of

the flex algorithm and that *potential causality* allows us to modify our basic implementation of the flex algorithm so that no message blocking occurs in the switches due to the requirements of isotach logical time.

4.2. Flex Algorithm

We present our flex algorithm for a switch-based network in which the physical topology is an undirected connected graph, (**V**, **E**), where **V** is the set of network nodes and **E** is a set of bidirectional FIFO message links. We assume each SIU connects its associated network element to exactly one switch.

Each network node has one port for each of its links, with an input buffer and an output buffer associated with each port. A switch *routes* a message when it moves the message from one of its inputs to one of its outputs. An SIU *sends* a message when it places the message in its output and *receives* a message when it removes the message from its input. Every switch maintains a *logical clock* for each of its ports. Each logical clock is a counter that tracks the number of *tokens* that the switch has routed to the associated output and gives the pulse component of logical time of the port. Each token is a control message that indicates when one pulse of logical time ends and the next begins. The network nodes exchange tokens to keep their logical clocks loosely synchronized.

Every SIU maintains two logical clocks for its port. The *logical send clock* of an SIU tracks the number of tokens it has sent on its output, while its *logical receive clock* tracks the number of tokens it has received on its input. Whenever an SIU sends or receives a message, its logical receive clock equals its logical send clock less any initial tokens that the SIU places into its output.

Each switch emits messages from each output in increasing logical time order and each SIU sends and receives all messages in increasing logical time order. A switch routes a message in pulse **i** if the logical clock associated with the output into which it places the message is **i** when it routes the message. An SIU sends (receives) a message in pulse **i** if its logical send (receive) clock is **i** when it sends (receives) the message. Every event of a message has the same *tag*, the minor components of isotach logical time. The definition of the logical time representation determines the tag for original messages, while the tag of a response equals the tag of the original message in extensible networks. By definition, the tag of a token is greater than that of any non-token message.

Our flex algorithm assigns a type to each port, which is either *blue* or *green*. A port's type applies to both its input and its output buffers. The endpoints of a link are not necessarily of the same type. Thus, a link can connect two green ports or two blue ports or it can connect a blue port to a green port. The algorithm is systolic or pulsing since a node waits to receive a token on each blue port, then "pumps" the tokens on its green ports. When the tokens return on the green ports, the node then pumps the tokens on its blue ports. Thus, the algorithm works in phases.

Figure 4.1 gives pseudocode for the switch algorithm. A switch places $\mathbf{t_q} \geq 0$ in the output of each of its ports, **q**. The *initial token count*, $\mathbf{t_q}$, can vary with each port and does not depend on its type.

After placing any initial tokens, the switch alternates between its *blue phase*, when it routes messages that arrive on its blue ports, and its *green phase*, when it routes messages that arrive on green ports. The port type from which the switch is routing messages is the *phase type* of the phase. The switch routes messages one at a time. It compares the tags of all messages at the heads of input ports of the current phase type and routes the

For each port, **q**
    Place $\mathbf{t_q}$ tokens in the port's output;
    $\mathbf{clock_q} = \mathbf{t_q}$;
Repeat forever
    Blue Phase: Route messages up to next token on all blue inputs in tag order;
    Remove token from each blue input, if any;
    Place a token in each green output, if any;
    Increment clock of each green port, if any;
    Green Phase: Route messages up to next token on all green inputs in tag order;
    Remove token from each green input, if any;
    Place a token in each blue output, if any;
    Increment clock of each blue port, if any;

Figure 4.1: Switch Routing Algorithm

message with the lowest tag. The switch waits if any input of the current phase type is empty. Thus, each switch routes messages arriving on the same input type in tag order. When each input of the phase type has a token at its head, the switch removes the tokens and places tokens in the outputs of the next phase type and ends the phase. A switch with only one type of port only executes the steps of each phase performed on that port type.

Switches use a pair of output buffers per port to ensure messages are emitted in logical time order. A single output buffer per port does not suffice since messages can be routed to a port out of tag order. Although messages arriving on the same input type are routed in tag order, messages arriving on different input types might be routed out of order. Messages that arrive on the same type as the output port are routed before messages routed in the same pulse that arrive on the opposite type, regardless of their tags.

Figure 4.2 shows an output buffer pair. To route a message, a switch enqueues it in the input to the splitter of the appropriate output buffer. The splitter enqueues each message routed to the port in a from-green queue or a from-blue queue depending on the input type on which the message arrives at the switch. However, the splitter enqueues a token in both intermediate queues for each token input to it. The merger emits the message with the

```
receive clock = 0;
Place t0 tokens in output; /*  Each SIU has one port, port 0 * /
send clock = t0;
If (port type is green) {
    Repeat forever {
        Blue Phase:     Place a token in output;
                        Increment send clock;
        Green Phase:    Send and receive messages in tag order;
                        Remove token from input;
                        Increment receive clock; } }
Else { /*  Port type is blue * /
    Repeat forever {
        Blue Phase:     Send and receive messages in tag order;
                        Remove token from input;
                        Increment receive clock; }
        Green Phase:    Place a token in output;
                        Increment send clock; } }
```

Figure 4.3: SIU Send/Receive Algorithm

lowest available tag, as well as recombining tokens for emission. Since each phase routes messages in tag order, our output buffer pair emits messages in logical time order.

Our switch algorithm is impractical since it routes messages one at a time and requires the special output buffers. Thus, we increase the latency through the switch significantly. This switch algorithm simplifies our analysis of the flex algorithm. We discuss more practical implementations of the flex algorithm in Section 4.5.

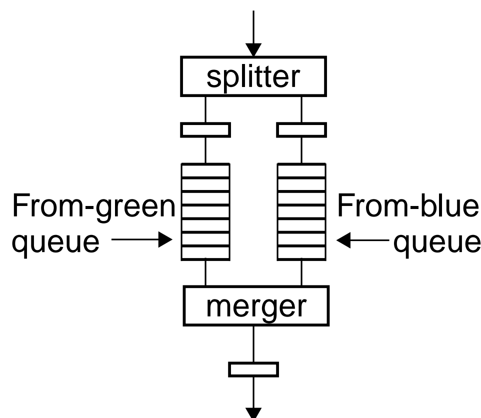


Figure 4.2: Output Buffer Pair

We give pseudocode for the SIU algorithm in Figure 4.3, which is essentially the switch algorithm when all ports are the same type. The only other differences arise from the maintenance of the **receive clock** in addition to the **send clock** of the switch algorithm. An SIU begins by placing any initial tokens in its output. The initial token count can vary with each SIU. Similar to a switch with all green ports, an SIU with a green port places an

additional token into its output before it sends or receives any messages. The SIU then sends and receives messages in tag order. The SIU increments its **receive clock** when it removes a token from its input and its **send clock** when it places a token in its output. The difference between the two clocks (i.e. **send clock - receive clock**) is the same for any send or receive event at the SIU since the SIU does not send or receive any messages between removing a token from its input and placing one in its output.

An SIU may send no messages or several messages in a pulse. We assume an SIU will eventually move any token at the head of its input to its output. The SIU sends two message streams that it merges into tag order: messages originated by an associated user process and response messages. We assume the SIU interleaves send and receive events so they occur in tag order. The network is extensible if the SIU blocks after it delivers a message while the message executes and then immediately places any response in its output. The response is delayed by the constant difference between its send and receive clocks.

4.3. Correctness

We demonstrate that our flex algorithm implements an isotach Net LTS. First, we define the logical distances that apply to the algorithm. Then, we demonstrate that it maintains the isotach invariant for all messages and is consistent with *potential causality*. We conclude this section with a discussion of how the flex algorithm generalizes both Awerbuch's β-synchronizer and the isonet algorithm [Awe85, RWW97].

We refine the concept of logical distance. In previous isotach network algorithms, logical distances equal the number of intermediate nodes on a message's routing path. However, our applications of isotach logical time only require that each routing path has a

known and fixed logical distance. In the flex algorithm, the logical distance of a routing path depends on the initial token counts of its switch outputs and the relationship of its input and output types at each switch. Each routing path has a known and fixed logical distance since we assume all port type assignments and initial token counts are disseminated during system initialization.

The logical distance that a message travels is independent of the initial token count and port type of its sending SIU in the flex algorithm. If the send pulse of a message is $\mathbf{i}$, then the sending SIU has placed exactly $\mathbf{i}$ tokens in its output before it places the message in its output. These tokens, and no more, always arrive before the message at the first switch of its routing path. These tokens include the initial tokens that the SIU places in its output. Therefore, the send pulse of the message, not its logical distance, is determined by the initial tokens of its sending SIU. The port type of the SIU affects the send pulse similarly since the only difference between the two types for an SIU is the token generated during the first blue phase of an SIU with a green port.

The logical distance that a message travels is also independent of the initial token count and port type of its receiving SIU. If the receive pulse of a message is $\mathbf{i}$, then the receiving SIU has removed exactly $\mathbf{i}$ tokens from its input before it removes the message from its input. The initial token count and port type of an SIU do not contribute to the number of tokens that it has removed from its input.

We use the *logical routing distance* across a switch to define logical distances in the flex algorithm. The logical routing distance across a switch, $\mathbf{s}$, of any message, $\mathbf{m}$, that arrives on port, $\mathbf{q_i}$, of $\mathbf{s}$ and $\mathbf{s}$ routes to its port, $\mathbf{q_o}$, is:

$$\mathrm{lrd}_{\mathbf{s}}(\mathbf{q_i}, \mathbf{q_o}) = \mathbf{t}_{\mathbf{q_o}} + \begin{cases} 1 & \text{if } \mathbf{q_i} \text{ and } \mathbf{q_o} \text{ are both green ports} \\ 0 & \text{otherwise} \end{cases}$$

where $\mathbf{t}_{q_o}$ is the initial token count of $\mathbf{q_o}$. Let the switches, <$\mathbf{s_0}$,…,$\mathbf{s_n}$>, be the intermediate nodes of the routing path from element **a** to element **b** of a message, **m**. The logical distance of the routing path is $\sum_{\mathbf{j}=0}^{\mathbf{n}} \mathrm{lrd}_{\mathbf{s_j}}(\mathbf{q_i}, \mathbf{q_o})$ where **m** arrives on port $\mathbf{q_i}$ of $\mathbf{s_j}$ and $\mathbf{s_j}$ routes **m** to its port $\mathbf{q_o}$. The logical distance of a routing path is essentially the sum of the initial token counts on the routing path. However, the initial token count must include the token placed in a green output during the first blue phase of any switch for which both its input and output ports on the routing path are green ports. Now, we prove logical time increases by the logical routing distance across a switch when the switch routes a message:

Lemma 4.1: If message **m** is sent to switch **s** by an adjacent SIU in pulse **i** or routed to **s** by an adjacent switch in pulse **i**, then **s** routes **m** to port $\mathbf{q_o}$ in pulse $\mathbf{i} + \mathrm{lrd}_{\mathbf{s}}(\mathbf{q_i}, \mathbf{q_o})$, where **m** arrives at **s** on port $\mathbf{q_i}$.

Proof: Since **m** is sent (routed) in pulse **i**, the adjacent SIU (switch) has sent (routed) exactly **i** tokens to **s** when it sends (routes) **m**.

We begin with the case where $\mathbf{q_i}$ is a blue port. When **s** routes **m**, it has removed exactly **i** tokens from every blue port.

Therefore, **s** has placed exactly **i** tokens after the initial tokens in each green port when it routes **m**. Thus, if $\mathbf{q_o}$ is a green port, **s** has incremented $\mathbf{clock}_{q_o}$ exactly **i** times after setting its initial value and **s** routes **m** to $\mathbf{q_o}$ in pulse $\mathbf{i} + \mathbf{t}_{q_o} = \mathbf{i} + \mathrm{lrd}_{\mathbf{s}}(\mathbf{q_i}, \mathbf{q_o})$.

Since **m** arrives at **s** on a blue port, **s** must route **m** in the $(\mathbf{i} + 1)^{st}$ iteration of its blue phase. Thus, **s** has completed exactly **i** iterations of its green phase and has placed exactly **i** tokens after the initial tokens in each blue port when it routes **m**. Thus, if $\mathbf{q_o}$ is a blue port, **s** has incremented $\mathbf{clock}_{q_o}$ exactly **i** times after setting its initial value and **s** routes **m** to $\mathbf{q_o}$ in pulse $\mathbf{i} + \mathbf{t}_{q_o} = \mathbf{i} + \mathrm{lrd}_{\mathbf{s}}(\mathbf{q_i}, \mathbf{q_o})$. Since $\mathbf{q_o}$ must be either a blue port or a green port, **s** routes **m** in pulse $\mathbf{i} + \mathrm{lrd}_{\mathbf{s}}(\mathbf{q_i}, \mathbf{q_o})$ if $\mathbf{q_i}$ is a blue port.

We now consider the case where $\mathbf{q_i}$ is a green port. When **s** routes **m**, it has removed exactly **i** tokens from every green port.

Therefore, **s** has placed exactly **i** tokens after the initial tokens in each blue port when it routes **m**. Thus, if $\mathbf{q_o}$ is a blue port, **s** has incremented $\mathbf{clock}_{q_o}$ exactly **i** times after setting its initial value and **s** routes **m** to $\mathbf{q_o}$ in pulse $\mathbf{i} + \mathbf{t}_{q_o} = \mathbf{i} + \mathrm{lrd}_{\mathbf{s}}(\mathbf{q_i}, \mathbf{q_o})$.

Since **m** arrives at **s** on a green port, **s** must route **m** in the (**i** + 1)st iteration of its green phase. Thus, **s** has completed exactly **i** + 1 iterations of its blue phase and has placed exactly **i** + 1 tokens after the initial tokens in each green port when it routes **m**. Thus, if $\mathbf{q_o}$ is a green port, **s** has incremented $\mathbf{clock}_{\mathbf{q_o}}$ exactly **i** + 1 times after setting its initial value and **s** routes **m** to $\mathbf{q_o}$ in pulse $\mathbf{i} + \mathbf{t}_{\mathbf{q_o}} + 1 = \mathbf{i} + \mathrm{lrd}_{\mathbf{s}}(\mathbf{q_i}, \mathbf{q_o})$. Since $\mathbf{q_o}$ must be either a blue port or a green port, **s** routes **m** in pulse $\mathbf{i} + \mathrm{lrd}_{\mathbf{s}}(\mathbf{q_i}, \mathbf{q_o})$ if $\mathbf{q_i}$ is a green port.

Thus, **s** routes **m** in pulse $\mathbf{i} + \mathrm{lrd}_{\mathbf{s}}(\mathbf{q_i}, \mathbf{q_o})$ since $\mathbf{q_i}$ must be either a blue port or a green port. **QED**

We add up the logical time that a message takes to cross any individual link to prove that the flex algorithm maintains the isotach invariant for all messages:

Lemma 4.2: The flex algorithm maintains the isotach invariant.

Proof: A switch, **s**, routes a message that arrives on its port $\mathbf{q_i}$ to its port $\mathbf{q_o}$ exactly $\mathrm{lrd}_{\mathbf{s}}(\mathbf{q_i}, \mathbf{q_o})$ pulses after the previous switch (SIU) routed (sent) the message by Lemma 4.1. The receiving SIU removes exactly **i** tokens from its output before receiving a message if the adjacent switch routes the message to it in pulse **i**. Thus, it receives the message in the same pulse that the preceding switch routes it. Therefore, the difference between the send and receive pulse of any message is the sum of the logical routing distances on its routing path, which is the logical distance that it travels. Thus, the flex algorithm maintains the isotach invariant. **QED**

Now, we show that the flex algorithm satisfies the other requirement of an isotach logical time system, consistency with *potential causality.*

Lemma 4.3: The flex algorithm is consistent with *potential causality.*

Proof: If **a** and **b** are the send and receive events, respectively, of a message, **m**, then $\mathbf{t_a} \le \mathbf{t_b} = \mathbf{t_a} + \mathbf{d_m}$ by Lemma 4.2 since $\mathbf{t_q} \ge 0$ for every port **q** and, thus, all logical distances are non-negative. Thus, the flex algorithm conforms to the condition **C1** of Section 3.4.

If **a** is a receive event and **b** is a send event that occurs at the same SIU after **a**, then $\mathbf{t_a} \le \mathbf{t_b}$ since each SIU interleaves send and receive events so as to handle all messages in tag order and its send clock is never less than its receive clock when it sends or receives a message. Thus, the flex algorithm conforms to the condition **C2′** and is consistent with *potential causality* by Corollary 3.1. **QED**

Lemmas 4.2 and 4.3 directly imply that the flex algorithm is correct.

Theorem 4.1: The flex algorithm implements an isotach Net LTS.

Proof: The flex algorithm maintains the isotach invariant for all messages by Lemma 4.2 and is consistent with *potential causality* by Lemma 4.3. Thus, it implements isotach logical time. **QED**

The **receive clock** of any SIU lags behind its **send clock** by $\mathbf{t_0}$ if its port is blue and $\mathbf{t_0}$ + 1 if its port is green. A send event, **s**, can occur before a receive event, **r**, at an SIU with a green port or $\mathbf{t_0} > 0$ such that $\mathbf{t_s} > \mathbf{t_r}$. Thus, the flex algorithm need not be consistent with the *happens before* relation. However, if **s** occurs before **r** and yet $\mathbf{t_s} > \mathbf{t_r}$, **s** cannot cause **r** since the algorithm is consistent with *potential causality* by Lemma 4.3. *Potential causality* accommodates the separate logical send and receive clocks because it models causality more accurately.

The flex algorithm generalizes the isonet algorithm. In the isonet algorithm, SIU's do not emit any initial tokens, each switch emits exactly one initial token on each link and switches consume tokens in a single step. Many port assignments and initial token counts result in identical behavior under the flex algorithm. For example, if every port is a blue port and $\mathbf{t_q} = 1$ for every switch port **q** and $\mathbf{t_q} = 0$ for every SIU port **q**, then the switch and SIU algorithms of our flex algorithm reduce to those of the isonet algorithm.

Logical distances in the flex algorithm are significantly more flexible than in the isonet algorithm. The flex algorithm supports logical distances greater than the routing distance through initial token counts greater than one. If the input or output of a switch on a routing path is a blue port and the initial token count of the output is zero, then the logical routing distance across the switch for that routing path is zero. Thus, the flex algorithm supports logical distances that are less than the routing distance.

We define an *extended isonet algorithm* that allows flexibility for logical distances without requiring two port types. The extended isonet algorithm is simpler than the flex algorithm, but less flexible. For brevity, we present the extended algorithm as an instance of the flex algorithm. As with the original isonet algorithm, every port is a blue port. However, we allow the initial token counts to vary so that $\mathbf{t_q} \geq 0$ for every port **q**.

The flex algorithm provides greater flexibility for logical distances than the extended isonet algorithm if we consider the issue of *logical time deadlock*, which describes the condition in which the pulse component of logical time never again increases. This condition leads to messages not being received since their logical receive times never arrive. Logical time deadlock occurs under the extended isonet algorithm if any link between two switches has zero cost. Under the extended isonet algorithm, a link between two switches has zero cost only if its associated ports have initial token counts of zero. Recall that a switch must route a token from a blue port before it routes any tokens to the port. Therefore, logical time deadlock occurs if both endpoints of any link are blue and its associated ports have initial token counts of zero. Thus, the extended isonet algorithm cannot allow a zero cost link between two switches. We will show that the multiple port types allow the flex algorithm to support a zero cost link between two switches.

Example: We discuss the use of both the flex and the extended isonet algorithm with the physical topology in Figure 4.4 to demonstrate the additional power of the flex algorithm. In this topology, **A** and **B** are network elements, while $\mathbf{S_0}$ and $\mathbf{S_1}$ are switches.

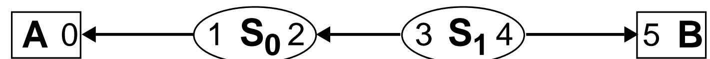


Figure 4.4: Example Isotach Network

In our flex algorithm example, the initial token count is zero for every port, while an arrow in Figure 4.4 indicates a green port. Thus, ports 0, 2 and 5 are green ports and

ports 1, 3 and 4 are blue ports. Any logical routing distance across either switch is zero since all initial token counts are zero and no switch has two green ports. Since all logical routing distances are zero, $\mathbf{d_{A,\,B}} = \mathbf{d_{B,\,A}} = 0$.

We can use the extended isonet algorithm with the example isotach network. In this case, all ports are blue. However, the extended isonet algorithm does not support $\mathbf{d_{A,\,B}} = \mathbf{d_{B,\,A}} = 0$ since that would require the link between $\mathbf{S_0}$ and $\mathbf{S_1}$ to have zero cost, which would imply that logical time deadlock must occur.

Logical time deadlock does not occur in this network under the flex algorithm. Since they have green ports, both **A** and **B** place a token in their output during their first blue phase before they receive any tokens. The token from **A** allows $\mathbf{S_0}$ to complete its first blue phase and place a token in the output of port 2. Thus, $\mathbf{S_1}$ receives a token on both its inputs and completes its first blue phase. It then immediately completes its first green phase and returns tokens to **B** and $\mathbf{S_0}$. $\mathbf{S_0}$ then completes its first green phase and returns a token to **A**, returning the network to its initial state. Thus, tokens continually pulse into $\mathbf{S_1}$ and back out. We formalize this discussion in Section 4.4, where we present a necessary and sufficient condition for logical time deadlock under the flex algorithm.

The flex algorithm can support $\mathbf{d_{A,\,B}} = \mathbf{d_{B,\,A}} = 0$ in the preceding example because we require consistency with *potential causality* rather than with the *happens before* relation. As previously discussed, if we require consistency with the *happens before* relation, then we must use a single clock at each SIU. Since each SIU has a green port, its receive clock lags behind its send clock by one pulse. This difference prevents logical time deadlock. If we use a single clock at each SIU, this difference must be reflected in the logical distances, which would then be one.

Now, we discuss the relationship of the flex algorithm and Awerbuch's network synchronizers [Awe85]. Previously, it has been observed that Awerbuch's α-synchronizer is equivalent to the isonet algorithm [RWW97]. Since the isonet algorithm is an instance of the flex algorithm so is the α-synchronizer. An adaptation of Awerbuch's β-synchronizer to implement n-tuple isotach logical time is an instance of the flex algorithm. The β-synchronizer sends tokens up a spanning tree of the network graph. When the root receives the token, it sends the token back down the tree. Now, we give port assignments that reduce the flex algorithm to a β-synchronizer when all initial token counts are zero. If a port **q** connects a network node to a child in the spanning tree of the network, then **q** is a blue port, while if **q** connects the node to its father, then **q** is a green port. The relationship of the flex algorithm to Awerbuch's γ-synchronizer is more complex.

4.4. Logical Time Deadlock

We present a Petri net model of the flex algorithm. We show that logical time deadlock will occur under a given instance of the flex algorithm if, and only if, the model for the instance is not live. Also, we show our model belongs to a class of Petri nets for which liveness is easily determined.

First, we briefly present Petri net models, as described by Peterson [Pet81]. A *Petri net structure*, **C**, is a four-tuple (**P**, **T**, **I**, **O**), where **P** is a finite set of places, **T** is a finite set of transitions, **I** is an input function and **O** is an output function. Both **I** and **O** map transitions to bags of places. Recall that a bag is a collection of objects in which each object can occur multiple times. Throughout this chapter, #(**p**, **B**) is function that returns the number of occurrences of place **p** in bag **B**. A Petri net is a *marked graph* if each place is an input of exactly one transition and an output of exactly one transition, i.e. $\forall$ **p** $\in$ **P**,

$\exists\ \mathbf{t_i}, \mathbf{t_o} \in \mathbf{T}$ such that $\#(\mathbf{p}, \mathbf{I}(\mathbf{t_i})) = \#(\mathbf{p}, \mathbf{O}(\mathbf{t_o})) = 1$, $\#(\mathbf{p}, \mathbf{I}(\mathbf{t})) = 0$ if $\mathbf{t} \neq \mathbf{t_i}$ and $\#(\mathbf{p}, \mathbf{O}(\mathbf{t})) = 0$ if $\mathbf{t} \neq \mathbf{t_o}$. Many analysis questions are easily answered for marked graphs.

A *marking*, μ, of a Petri net structure is a function that assigns tokens to the places and, thus, maps **P** to the non-negative integers. Initially, we differentiate Petri net tokens from logical time tokens. Marking μ enables transition **t** if each place, **p**, has at least as many tokens as there are inputs from **p** to **t**, i.e. $\mu(\mathbf{p}) \geq \#(\mathbf{p}, \mathbf{I}(\mathbf{t}))$. A transition can fire if it is enabled. Firing transition **t** in marking μ results in a new marking, μ', in which a token is consumed from each input of the transition and a token is created in each of its outputs, thus $\forall\ \mathbf{p} \in \mathbf{P}$, $\mu'(\mathbf{p}) = \mu(\mathbf{p}) - \#(\mathbf{p},\mathbf{I}(\mathbf{t})) + \#(\mathbf{p},\mathbf{O}(\mathbf{t}))$.

Marking μ' is reachable from marking μ of Petri net **C** if there exists a series of enabled transition firings in **C** starting from μ that results in μ'. The reachability set, $\mathbf{R}(\mathbf{C}, \mu)$ of Petri net **C** with marking μ is the set of all reachable markings from μ in **C**. Transition **t** of **C** is live in μ if $\forall\ \mu' \in \mathbf{R}(\mathbf{C}, \mu)$, there exists a series of enabled transition firings that enable **t**. Petri net **C** with marking μ is live if every transition is live in μ. A marked graph is live if, and only if, every directed cycle has at least one token on it.

Now, we present our Petri net model of the flex algorithm. Determining a necessary and sufficient condition for logical time deadlock is the primary goal of this model. Recall that logical time deadlock occurs if the pulse component of logical time never again increases. Under the flex algorithm, logical time deadlock occurs if any logical clock is never again incremented since the stoppage of one logical clock will eventually stop all other logical clocks. Since the incrementing of any logical clock occurs when a token is placed in its associated output, all token movement stops if logical time deadlock occurs. Therefore, we only model logical time token movement and the Petri net tokens in our model correspond directly to the logical time tokens.

We use the model that Figure 4.5 shows for every network node since we are only concerned with token movement. As we indicated in Section 4.2, the differences between the switch and SIU algorithms do not affect token movement. Thus, this model applies to both the switch and SIU algorithms.

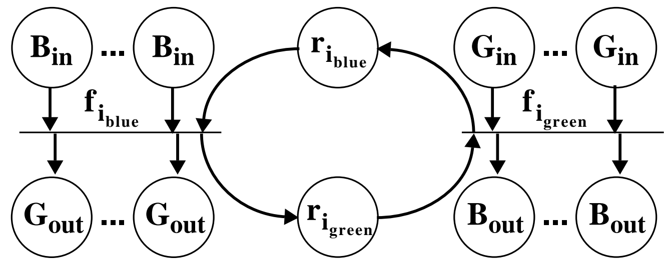


Figure 4.5: Model of Network Node $\mathbf{v_i}$

For each network node, $\mathbf{v_i}$, our model has two internal transitions, $\mathbf{f_{i_{blue}}}$ and $\mathbf{f_{i_{green}}}$. We use $\mathbf{f_{i_{blue}}}$ to model the blue phase, when the algorithm removes tokens from the inputs of blue ports and places tokens into the outputs of green ports. Similarly, $\mathbf{f_{i_{green}}}$ models the green phase. Thus, we create two places, $\mathbf{p_{i,\, q_{in}}}$ and $\mathbf{p_{i,\, q_{out}}}$, for each port, $\mathbf{q}$, of $\mathbf{v_i}$. If $\mathbf{q}$ is a blue port, then $\mathbf{p_{i,\, q_{in}}}$ is an input of $\mathbf{f_{i_{blue}}}$ and $\mathbf{p_{i,\, q_{out}}}$ is an output of $\mathbf{f_{i_{green}}}$,while if $\mathbf{q}$ is a green port, then $\mathbf{p_{i,\, q_{in}}}$ is an input of $\mathbf{f_{i_{green}}}$ and $\mathbf{p_{i,\, q_{out}}}$ is an output of $\mathbf{f_{i_{blue}}}$. Thus, these transitions model the internal logical time token movement of each phase of the algorithm. We label the places for the ports of $\mathbf{v_i}$ by their types in Figure 4.5. Thus, each $\mathbf{B_{in}}$ or $\mathbf{G_{in}}$ is a $\mathbf{p_{i,\, q_{in}}}$ and each $\mathbf{B_{out}}$ or $\mathbf{G_{out}}$ is a $\mathbf{p_{i,\, q_{out}}}$ in Figure 4.5.

Two additional places, the run places $\mathbf{r_{i_{blue}}}$ and $\mathbf{r_{i_{green}}}$, model how a node alternates its phases. Thus, $\mathbf{r_{i_{blue}}}$ is an input of $\mathbf{f_{i_{blue}}}$ and an output of $\mathbf{f_{i_{green}}}$, while $\mathbf{r_{i_{green}}}$ is an input of $\mathbf{f_{i_{green}}}$ and an output of $\mathbf{f_{i_{blue}}}$. The blue phase can proceed if a token is in $\mathbf{r_{i_{blue}}}$. The blue phase completes when $\mathbf{f_{i_{blue}}}$ fires, which removes the token from $\mathbf{r_{i_{blue}}}$ and places a token in $\mathbf{r_{i_{green}}}$. This token allows the green phase to proceed, which completes when $\mathbf{f_{i_{green}}}$ fires.

Figure 4.6 shows our model of a network link between nodes $\mathbf{v_i}$ and $\mathbf{v_k}$. We model the link with transitions $\mathbf{f_{i,\, k}}$ and $\mathbf{f_{k,\, i}}$. The only input of $\mathbf{f_{i,\, k}}$ is $\mathbf{p_{i,\, j_{out}}}$ and its only output is $\mathbf{p_{k,\, l_{in}}}$, where the link connects port $\mathbf{j}$ of $\mathbf{v_i}$ to port $\mathbf{l}$ of $\mathbf{v_k}$. Thus, $\mathbf{f_{i,\, k}}$ models the movement

of a logical time token from $\mathbf{v_i}$ to $\mathbf{v_k}$ and combines with $\mathbf{f_{k,\,i}}$ to model token movement over the link.

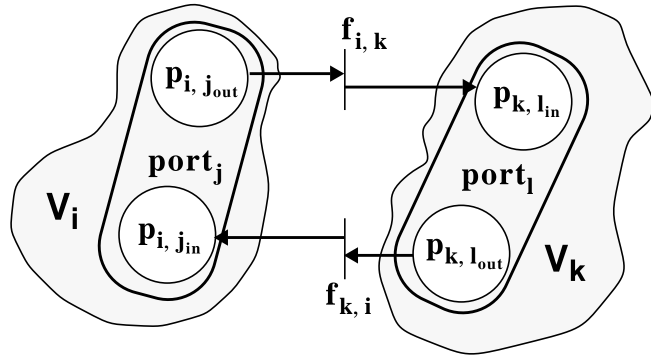


Figure 4.6: Model of Link Between $\mathbf{v_i}$ and $\mathbf{v_k}$

Now, we specify the initial marking, μ, of our model. For each port $\mathbf{q}$ of every network node $\mathbf{v_i}$, $\mu(\mathbf{p_{i,\,q_{in}}}) = 0$ and $\mu(\mathbf{p_{i,\,q_{out}}}) = \mathbf{t_q}$, the initial token count of $\mathbf{q}$. These markings capture all logical time tokens created during the initialization phase of the algorithm. We mark the run places to ensure only one phase runs at a time. Since every node begins in its blue phase, $\mu(\mathbf{r_{i_{blue}}}) = 1$ and $\mu(\mathbf{r_{i_{green}}}) = 0$. The initial token in $\mathbf{r_{i_{blue}}}$ requires a node with no blue ports to place a token in each of its green ports before it removes any tokens from its ports. Our example at the end of Section 4.3 indicates this aspect of the flex algorithm is an important element in the flexibility for logical distances that it provides.

Our Petri net model of the flex algorithm is a marked graph. Every run place is clearly the input of exactly one transition and the output of exactly one transition. For every port $\mathbf{q}$ of any node $\mathbf{v_i}$, $\mathbf{p_{i,\,q_{in}}}$ is an input of either $\mathbf{f_{i_{blue}}}$ or $\mathbf{f_{i_{green}}}$, but not both. Similarly, $\mathbf{p_{i,\,q_{out}}}$ is an output of exactly one internal transition. If $\mathbf{q}$ connects $\mathbf{v_i}$ to $\mathbf{v_k}$, then $\mathbf{p_{i,\,q_{out}}}$ is an input of $\mathbf{f_{i,\,k}}$ and $\mathbf{p_{i,\,q_{in}}}$ is an output of $\mathbf{f_{k,\,i}}$. These are the only transitions for which $\mathbf{p_{i,\,q_{in}}}$ and $\mathbf{p_{i,\,q_{out}}}$ are inputs or outputs and, thus, our model is a marked graph.

Logical time deadlock occurs if our model is not live since its transitions capture all token movement and token movement stops if logical time deadlock occurs under the flex algorithm. Since our model is a marked graph, determining if it is live only requires determining if every cycle has a token on it. System initialization can check our Petri net model to ensure that port assignments and initial token counts do not cause logical time

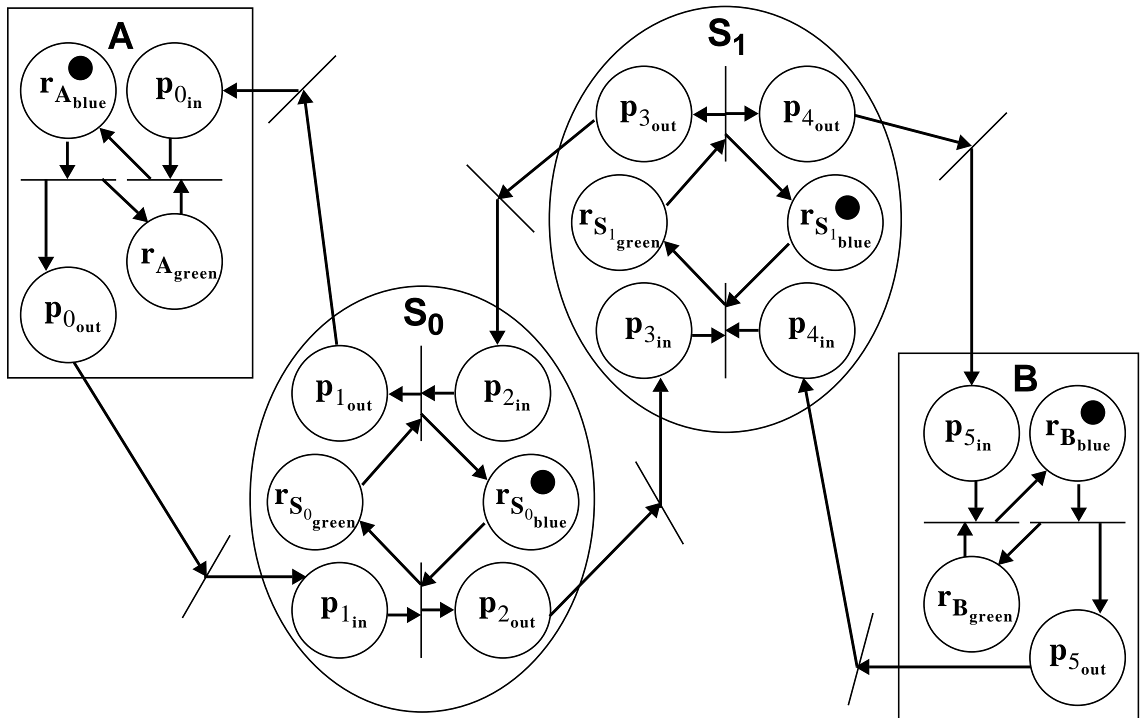


Figure 4.7: Model of Example Isotach Network

deadlock. If logical time deadlock will occur, we can use our model to revise the initial token counts so that logical time deadlock does not occur.

Example: Figure 4.7 shows our Petri net model for the example that we used at the end of Section 4.3 to demonstrate the power of the flex algorithm. Since all initial token counts are zero in this example, the only tokens in the initial marking of the model are in the blue run places. At least one of these tokens is on every directed cycle of the model and, thus, logical time deadlock will not occur, as we informally argued previously.

4.5. Performance Optimizations

Our flex algorithm ensures that every SIU receives all messages in logical time order similarly to the isonet algorithm [RWW97]. Significant message blocking occurs in the switches and the sending SIU’s in order to ensure that they route or send the message with the earliest logical route or send time. In prototype isotach systems that use the isonet

algorithm, each SIU sorts messages that arrive on its input into logical time order in order to reduce message blocking in the switches and at the sending SIU [Reg97, WiR97]. We can apply similar techniques to the flex algorithm.

The non-blocking isonet algorithm implementation assumes FIFO links between adjacent network nodes, which allows the implementation to ensure that a token never leaves a node before a message that previously arrived on the same input. The implementation removes tokens from the message stream. Tokens are sent on each output after one has been removed from each input. This removal and reinsertion of tokens allows messages to pass tokens but not vice versa. A similar flex algorithm implementation matches tokens on inputs of the same type and sends tokens on the appropriate output type.

In the non-blocking implementation, the receiving SIU must sort messages into logical time order since the network can emit messages out of order. Since the receiving SIU sorts messages into logical time order, we allow SIU's to send messages out of logical time order. The SIU sends each message as soon as it determines the receive pulse. When a token arrives at an SIU, the SIU has received all messages with receive times in the pulse that the token ends. The SIU then sorts and delivers these messages.

The non-blocking implementation uses the same logical distances as the implementation described in Section 4.2. The sending SIU appends a timestamp to the message that indicates the receive pulse, which the SIU determines from the logical time of the message and the distance to the receiving SIU. The sending SIU ensures that it sends messages on time, i.e. the send pulse of a message is at least the SIU's **send clock** value when the message is sent. Since tokens cannot pass messages, the **receive clock** of the receiving SIU is never greater than the receive pulse of the message. For any message that is not a response, we assume the SIU schedules a send pulse greater than the value of its receive

clock at the time that it schedules the message. This assumption ensures that the logical send time of any message, **m**, is greater than the logical receive time of any message that the SIU delivers before **m** is issued and, thus, the implementation conforms to the condition **C2** of Section 3.4. Since it also enforces the isotach invariant, the implementation conforms to the condition **C1** and, thus, is consistent with *potential causality.*

This implementation further demonstrates the power of *potential causality.* Two send events, or two receive events, can occur at the same SIU out of logical time order, which is not consistent with the *happens before* relation. However, no causal relation exists between such events, reflecting the increased accuracy of *potential causality.*

4.6. Chapter Summary

We presented the flex algorithm for isotach networks. The flex algorithm has significantly more flexibility for logical distances than previous algorithms. This flexibility is a significant advance for isotach technology since logical distances can reflect the raw message latency of each link. We proved that the flex algorithm implements an isotach logical time system. We presented a Petri net model of the flex algorithm that allows the inexpensive detection of logical time deadlock during system initialization. Our implementations of the flex algorithm demonstrate the power of *potential causality* over the *happens before* relation. We leave performance analysis of the flex algorithm for future work.

Chapter 5:
Execution Time and Replication

5.1. Introduction

In this chapter, we present a new framework that provides a unifying theory for isotach shared memory systems. By eliminating the use of a physical canonical copy, this framework supports the design of new delta coherence protocols that extend isotach-based coherence techniques to a wider range of networks and applications. Our framework supports optimizations not addressed by previous research and demonstrates that a correct delta coherence protocol represents a class of correct protocols.

Our framework uses concepts similar to Williams's formulation of delta coherence protocols based on effective execution times [Wil93]. Her formulation assigns logical times to execution events performed on cache copies that can be different from the logical receive times of the requests at the copies. However, the memory copy is a physical canonical copy for which execution times must equal the corresponding logical receive times. Our framework completes the separation between execution times and the logical receive times by eliminating the use of a physical canonical copy.

Our framework uses a modular design based on two *meta-isotach logical time systems*, logical time systems that are built on top of an underlying isotach Net LTS. The logical times that these systems assign are derived from the logical times that the Net LTS assigns to send and receive events between messaging processes. Our design allows isotach shared memory systems to exploit the flexibility inherent in equivalent shared memory executions without altering the requirements of an isotach logical time system.

5.2. System Assumptions

We assume isotach shared memory systems use the system model discussed in Chapter 3. Ordinary user processes issue all shared memory requests. *Memory processes* are special user processes that implement the shared memory space. Also, we assume:

1) Each SIU hosts a messaging process;
2) Shared memory requests are issued to the collocated SIU in program order;
3) The last request of each isochron is distinguished;
4) An SIU uses messages to service requests;
5) Memory processes do not perform any computation;
6) Any execution event for a shared request occurs in a memory process.

5.3. Logical Execution Time

Logical execution time, a meta-isotach logical time system, assigns an *execution time*, $\mathbf{t_e}$, to each execution event, **e**, of the system. Execution equivalence motivates our definition of logical execution time. Previous isotach shared memory systems assigned execution times within the messaging isotach logical time system [Wil93, RWW97]. Thus, those systems required execution times to be consistent with *potential causality.* Our separation of the time systems requires logical execution time only to be consistent with the causal relations captured by execution equivalence.

The execution events performed on a copy must occur in the order of their execution times. The *execution time function* of the copy, a strictly increasing function of the logical receive times of the requests at the copy, determines these execution times. Formally, if $\mathbf{e_0}$ and $\mathbf{e_1}$ are two execution events performed on a copy and $\mathbf{r_0}$ and $\mathbf{r_1}$ are their corresponding receive events, then ($\mathbf{e_0}$ occurs before $\mathbf{e_1}$) $\Leftrightarrow$ ($\mathbf{t_{e_0}} < \mathbf{t_{e_1}}$) $\Leftrightarrow$ ($\mathbf{t_{r_0}} < \mathbf{t_{r_1}}$). Although $\mathbf{r} \Rightarrow \mathbf{e}$ when **e** is an execution event and **r** is its corresponding receive event, we do not require $\mathbf{t_r} \leq \mathbf{t_e}$ since they are assigned by different logical time systems.

The execution time functions of copies of different variables or different copies of the same variable can vary, even if they are collocated. The logical receive and execution times of two execution events do not constrain the order in which the events occur if they are performed on distinct copies. Also, the relationship of their logical receive times does not constrain their execution times.

A well understood principle related to conflict equivalence justifies our execution time functions [Pap86]. Any two executions that differ only in the interleaving of execution events to different copies are equivalent. The real time order of the execution events can differ from their execution time order. However, the actual execution and the *logical execution*, the execution in which all execution events occur in their execution time order, are equivalent since execution events performed on each copy occur in the same order.

Logical execution time may not be consistent with causality since our execution time requirements do not preclude $\mathbf{t}_{\mathbf{e}_1} < \mathbf{t}_{\mathbf{e}_0}$ when $\mathbf{e_0} \Rightarrow \mathbf{e_1}$ for execution events $\mathbf{e_0}$ and $\mathbf{e_1}$ performed on different copies. However, logical execution time is useful because it is consistent with the causality between certain critical (i.e. write/read) events that occur on the same copy. Thus, logical execution time is consistent with the causal relations captured by execution equivalence. Execution equivalence only captures the causal relationship of output dependence, i.e. the causal relationship between a read, **r**, and the write, **w**, that stored the value that **r** returns. Our execution times are consistent with this relationship since the copy that executes **r** must have previously executed **w** and, thus $\mathbf{t}_{\mathbf{e}_\mathbf{w}} < \mathbf{t}_{\mathbf{e}_\mathbf{r}}$.

Our execution time requirements support many optimizations. For example, a memory process can execute read requests before previously received write requests to different variables. Previous isotach research did not address this optimization.

The *execution displacement*, δ, of a copy is the amount by which the execution time function for that copy shifts execution times relative to logical receive times. If **r** is the receive event at a copy of a request that is executed on the copy in event **e**, then the execution time function of a copy is $\mathbf{t_e} = \mathbf{t_r} + \delta$, for some integer constant δ. We leave investigating general execution time functions for future work.

The execution displacement for each copy maps the logical receive time line to an execution time line, as Figure 5.1 shows for $\delta < 0$. We stress execution times and logical receive times are distinct, although related. The logical receive time need not equal the execution time when the execution event occurs. In fact, the logical receive time cannot equal the execution time when $\delta < 0$. We allow $\delta \neq 0$, even in systems without replication, unlike previous research that assumed every execution time function was the identity function [Wil93, RWW97]. The original delta coherence protocols used a similar mechanism that shifted the effective, or apparent, execution time of execution events performed on some copies. We eliminate this replication-based distinction.

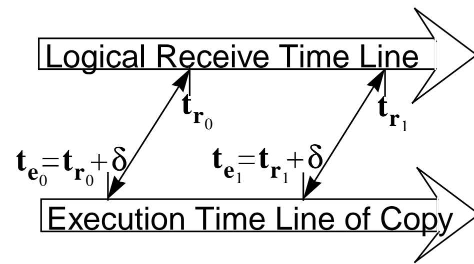


Figure 5.1: Time Line Mapping

The *execution distance*, Φ, of an execution event is $\mathbf{t_e} - \mathbf{t_s}$, where $\mathbf{t_e}$ is the execution time and $\mathbf{t_s}$ is the *initial send time* of the request, which is the logical send time of the first message used to service the request. In extensible networks, an intermediate location can forward the request to the copy. Many of our coherence protocols use *request forwarding*. Request forwarding with immediate responses results in $\Phi = \left(\sum_{\mathbf{m}} \mathbf{d_m}\right) + \delta$, the execution displacement of the copy plus the sum of the distances travelled by the messages that bring the request to the copy. With delayed responses, the execution distance includes the

sum of the logical delays. Systems without replication do not use request forwarding, so $\Phi = \mathbf{d_m} + \delta$, where $\mathbf{d_m}$ is the logical distance between the issuing SIU and the copy.

Example: Examples throughout this chapter use the physical topology depicted in Figure 5.2. Switches, shown as circles, connect two PE's, each with one process, and two MM's, each with two variables.

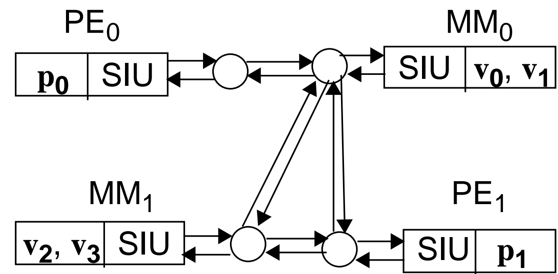


Figure 5.2: Example Physical Topology

An isotach network's *logical topology* is a weighted graph in which the edge weight between two elements is the logical distance between them. Figure 5.3 shows the logical topology for our example physical topology assuming all messages travel the shortest routing path and logical distances equal routing distances. Since MM's do not communicate in our examples, we omit that edge.

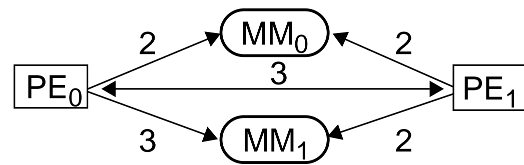


Figure 5.3: Example Logical Topology

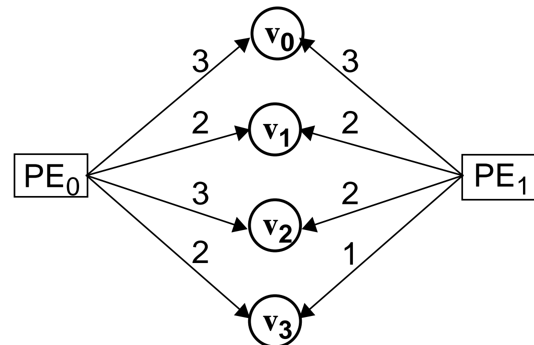


Figure 5.4: Example Effective Topology

Non-zero execution displacements effectively change the logical topology. Figure 5.4 shows the execution distances when $\delta_{\mathbf{v}_0} = 1$, $\delta_{\mathbf{v}_1} = \delta_{\mathbf{v}_2} = 0$ and $\delta_{\mathbf{v}_3} = -1$. Thus, $\mathbf{v_0}$ effectively moves away from each PE and $\mathbf{v_1}$ effectively moves closer.

5.4. Scheduled Logical Time

Scheduled logical time, our other meta-isotach logical time system, assigns a *scheduled execution time*, τ, to each request when the *scheduling decision* of the request occurs. Its scheduling decision occurs when the issuing SIU determines its initial send

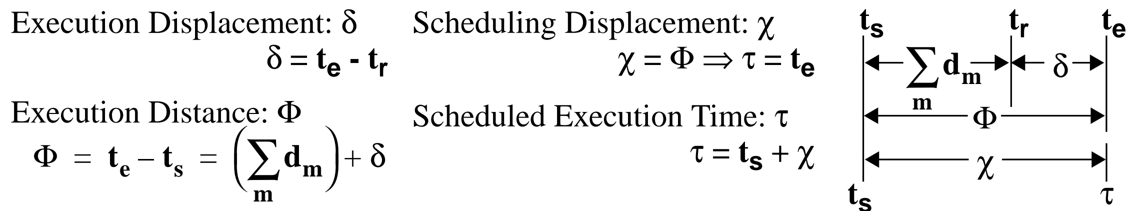


Figure 5.5: Shared Memory Meta-Isotach Time Systems

time. We ensure that the scheduling decisions of an isotach shared memory system conform to its ordering constraints. We also ensure that every execution agrees with the scheduling decisions, thus solving the concurrency control problem.

The issuing SIU selects the *scheduling displacement*, χ, of a request in order to assign its scheduled execution time. We discuss how the SIU selects χ in systems without replication in Section 5.5 and in systems with replication in Section 5.7.3. For any request, $\tau = \mathbf{t_s} + \chi$, where $\mathbf{t_s}$ is the initial send time of the request. The scheduled execution times of the requests of an isotach shared memory system define its *scheduled execution order.* The scheduled execution order defines an execution, the *scheduled execution*, which has exactly one execution event for each request. These execution events occur in the scheduled execution order. By definition, the scheduled execution associates the same value with each write request as the actual execution, while it associates with the read request, **r**, the value associated with **w**, the most closely preceding write in the scheduled execution order to the same variable, i.e. $\tau_{\mathbf{w}} < \tau_{\mathbf{r}}$ and no $\mathbf{w}'$ exists such that $\tau_{\mathbf{w}} < \tau_{\mathbf{w}'} < \tau_{\mathbf{r}}$.

Figure 5.5 summarizes the notation of our meta-isotach logical time systems. Note that the definitions of τ and $\mathbf{t_e}$ imply that $\tau = \mathbf{t_e}$ if $\chi = \Phi$.

5.5. Sequentially Consistent and Isochronous Isotach Systems

As discussed in Section 5.3, our rules for assigning execution times ensure that the actual and logical executions are equivalent. In this section, we present send order rules

Actual Execution = Logical Execution $\stackrel{?}{=}$ Scheduled Execution $\Rightarrow$ Ordering Constraints

Figure 5.6: Correctness Framework

that ensure the scheduled execution of an isotach shared memory system conforms to its ordering constraints. Given that the system enforces these send order rules, the system is correct if the scheduled and logical executions are equivalent. We conclude this section by showing how isotach shared memory systems without replication ensure this equivalence. Figure 5.6 shows our correctness framework for isotach shared memory systems.

The issuing SIU of a request determines the request's initial send time with the *scheduling algorithm* of the system. The scheduling algorithm must ensure scheduled execution times conform to the ordering constraints of the system. The scheduling algorithm controls the *scheduled execution pulse* of each request, the pulse component of its scheduled execution time, τ. A correct scheduling algorithm implements these send order rules:

IRule: Send the requests of an isochron so each has the same scheduled execution pulse.
SCRule: Send each request so its scheduled execution pulse is at least that of the request issued before it by the same process.

The issuing SIU can enforce **SCRule** since it chooses each scheduling displacement. It also can enforce **IRule** since isochrons have no internal true dependences. We now prove that **SCRule** ensures the scheduled execution is sequentially consistent and **IRule** ensures the requests of each isochron occur consecutively in the scheduled execution.

Lemma 5.1: Every scheduled execution, $\mathbf{E_S}$, is isochronous and sequentially consistent if the scheduling algorithm is correct.

Proof: By definition, the requests of any isochron, **I**, are issued by the same process and have consecutive issue ranks. Since the requests of **I** have the same scheduled execution pulse by **IRule**, their scheduled execution times form a contiguous interval of logical time. Thus, the requests of **I** are executed in $\mathbf{E_S}$ without interleaving with other requests and $\mathbf{E_S}$ is isochronous.

Let **a** and **b** be any two requests where the same process issues **a** before **b**. Issue rank is a strictly increasing function of issue order by definition and the scheduled execution pulse of **b** is at least that of **a** by **SCRule**. Thus, $\tau_{\mathbf{a}} < \tau_{\mathbf{b}}$ so **a** occurs before **b** in $\mathbf{E_S}$ and $\mathbf{E_S}$ is sequentially consistent. **QED**

An isotach system is isochronous and sequentially consistent if its scheduling algorithm is correct and it ensures each execution is equivalent to its scheduled execution. We present a correct scheduling algorithm in Section 5.6. Throughout this chapter, we develop conditions that ensure every execution is equivalent to its scheduled execution.

In an isotach system without replication, we can show the scheduled and logical executions are equivalent by showing $\mathbf{t_e}$ *always equals* τ, i.e. the execution time of any execution event equals the scheduled execution time of the request:

Lemma 5.2: Any execution, **E**, is equivalent to its scheduled execution, $\mathbf{E_S}$, in an isotach system without replication if $\mathbf{t_e}$ always equals τ.

Proof: Let **a** and **b** be any two requests to the same variable such that $\mathbf{e_a}$ occurs before $\mathbf{e_b}$ in **E**, where $\mathbf{e_a}$ and $\mathbf{e_b}$ are the respective execution events of the requests. Since the execution times of execution events performed on the same copy are consistent with the order in which the events occur, $\mathbf{t_{e_a}} < \mathbf{t_{e_b}}$. Since $\mathbf{t_e}$ always equals τ, $\tau_{\mathbf{a}} < \tau_{\mathbf{b}}$. Thus, **E** and $\mathbf{E_S}$ are conflict equivalent since $\mathbf{E_S}$ preserves the order of conflicting requests. Conflict equivalence is equivalence for our purposes. **QED**

We assume each SIU knows the logical distance, $\mathbf{d_m}$, to each copy, and its execution displacement, δ. Thus, the issuing SIU can select $\chi = \mathbf{d_m} + \delta = \Phi$, which ensures $\mathbf{t_e} = \tau$. We now show that an isotach system without replication is correct if it implements our send order rules since we assume the issuing SIU always selects $\chi = \mathbf{d_m} + \delta$:

Theorem 5.1: Any execution, **E**, is isochronous and sequentially consistent in an isotach system without replication that implements **IRule** and **SCRule**.

Proof: By Lemma 5.1, $\mathbf{E_S}$, the scheduled execution of **E**, is isochronous and sequentially consistent. Since $\chi = \mathbf{d_m} + \delta = \Phi$, $\mathbf{t_e}$ always equals τ. Thus, **E** is equivalent to $\mathbf{E_S}$ by Lemma 5.2. Therefore, **E** is also isochronous and sequentially consistent. **QED**

Subsequently, we will extend this result to systems with replication.

In systems with replication, χ depends on the state of the copy, if any, associated with the issuing SIU. Ensuring that $\chi = \Phi$ is more difficult, in part because the issuing SIU often does not know the location of some copies that must execute the request. In Section 5.7, we discuss the selection of scheduling displacements in systems with replication.

Example: We return to our example from the end of Section 5.3. In an example program that we reuse later in this chapter, $\mathbf{p_0}$ isochronously reads $\mathbf{v_0}$ and writes $\mathbf{v_1}$, and subsequently writes $\mathbf{v_2}$ while $\mathbf{p_1}$ isochronously writes $\mathbf{v_0}$ and reads $\mathbf{v_1}$. Figure 5.4 shows Φ for each process, variable pair. Since $\chi = \Phi$, $\chi_0 = 3$ and $\chi_1 = 2$ for both processes and $\chi_2 = 3$ for $\mathbf{p_0}$. **IRule** requires that each process send its request to $\mathbf{v_0}$ one pulse earlier than its request to $\mathbf{v_1}$. **SCRule** allows an initial send time for the write of $\mathbf{v_2}$ by $\mathbf{p_0}$ up to one pulse less than the initial send time of its write to $\mathbf{v_1}$.

Figure 5.7 shows schemata for one possible execution of our example. In our schemata, horizontal lines correspond to the logical times indicated. A dashed rectangle depicts the grouping of requests into an isochron. The solid rectangle indicates both the scheduled execution time, τ, of the request and the execution time, $\mathbf{t_e}$, of its execution event. The black dot shows its initial send time. The oval indicates the logical receive time of the request. The gaps shown between the logical receive times and execution times for $\mathbf{v_0}$ are due to its execution time function, which shifts execution times to one pulse greater than the corresponding receive

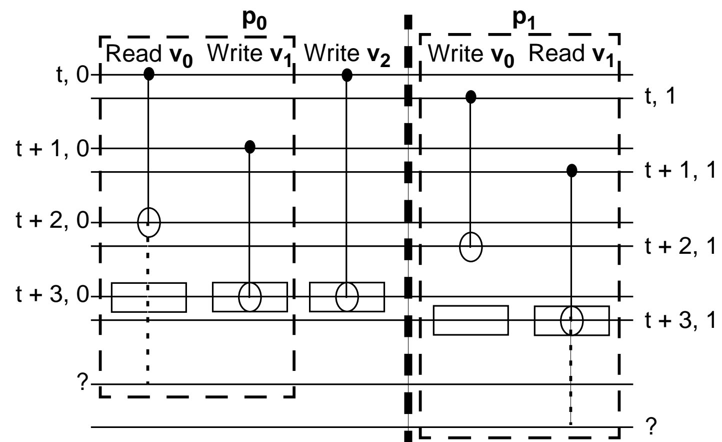


Figure 5.7: Possible Example Execution

times. Solid vertical lines are messages that use the standard level of service. The dotted vertical lines are messages that use the bounded level of service. A question mark indicates the indeterminate logical receive time, $\mathbf{t_r}$, of a bounded message. Since the logical receive time of a read response does not affect the logical execution order, it only needs to be consistent with causality, i.e. it should not appear to be received before it was sent. Sending the response at the bounded level of service ensures $\mathbf{t_s} \leq \mathbf{t_r}$.

In our example, the scheduling algorithm happens to select t + 3 for the scheduled execution pulse of all requests. Since the requests have the same scheduled execution pulse, the requests by $\mathbf{p_0}$ precede those of $\mathbf{p_1}$ in the scheduled execution due to the **pid-rank** component of the scheduled execution times. In our discussion, we assume that the execution events occur in the order of the corresponding logical receive times. Thus, the actual execution is: $\mathbf{p_0}$: read $\mathbf{v_0}$; $\mathbf{p_1}$: write $\mathbf{v_0}$; $\mathbf{p_0}$: write $\mathbf{v_1}$; $\mathbf{p_0}$: write $\mathbf{v_2}$; $\mathbf{p_1}$: read $\mathbf{v_1}$.

Our actual execution interleaves the isochrons. For example, the execution event of the write to $\mathbf{v_0}$ by $\mathbf{p_1}$ occurs before the write to $\mathbf{v_1}$ by $\mathbf{p_0}$. However, the actual execution is equivalent to the scheduled execution, in which the isochrons are not interleaved, since conflict equivalence allows us to reorder requests to different variables. Graphically, the execution events of $\mathbf{v_0}$ shift a uniform amount of logical time, preserving the order of conflicting requests. Note that our example execution pipelines the write to $\mathbf{v_2}$.

Previous results for isotach systems without replication assume that all execution displacements are zero and that each MM executes requests in their logical receive time order [Wil93, RWW97]. These assumptions prohibit the actual execution that we assume with a three-tuple isotach logical time system since $\mathbf{v_0}$ and $\mathbf{v_1}$ are located at the same MM. Our new framework for isotach shared memory systems allows this execution and more.

5.6. Basic Scheduling Algorithm

A correct scheduling algorithm ensures scheduling decisions conform to the send order rules and, thus, the scheduled execution is isochronous and sequentially consistent. In this section, we describe a completely distributed scheduling algorithm that assumes the issuing SIU buffers an isochron until it is completely issued. We can safely suspend a user process while issuing an isochron if the SIU buffers isochrons separately for each process.

For each process, **p**, that issues isochrons to the SIU, we use a variable, $\mathbf{last_p}$, that tracks the scheduled execution pulse of the last isochron scheduled for **p**. The initial value of $\mathbf{last_p}$ does not affect the correctness of the algorithm. We use two more variables to schedule a completely issued isochron, **I**: variable **min_send**, which is the current minimum possible local send pulse for messages with **pid-rank** component **p**; and variable, $\chi_{\mathbf{I}}$, which bounds the maximum scheduling displacement of any request in **I**. In some of our protocols, some requests have multiple possible scheduling displacements before the algorithm determines their scheduled execution times. In Step 1, $\chi_{\mathbf{a}_{MAX}}$ is the maximum of the possible scheduling displacements for the request **a**. The following procedure specifies our basic scheduling algorithm:

Input: All requests of isochron **I** issued by process **p**.
Step 1: Determine $\chi_{\mathbf{I}}$ equal $\max(\chi_{\mathbf{a}_{MAX}})$ over every request, **a**, of **I**.
Step 2: Set **min_send** to the current minimum possible local send pulse.
Step 3: Set $\mathbf{last_p}$ to $\max(\mathbf{min_send} + \chi_{\mathbf{I}}, \mathbf{last_p})$.
Step 4: For each request, **a**, make $\mathbf{last_p} - \chi_{\mathbf{a}}$ the pulse component of $\mathbf{t}_{\mathbf{s}_a}$, where $\chi_{\mathbf{a}}$ is the scheduling displacement of **a**.

For systems without replication, the SIU selects $\chi = \mathbf{d_m} + \delta$ in Step 1 for each scheduling displacement as discussed previously. We discuss how the SIU selects each scheduling displacement in isotach systems with replication in Section 5.7.3. The first term in the max operation specified by Step 3, $\mathbf{min_send} + \chi_{\mathbf{I}}$, ensures that the initial send

times that are required by the scheduled execution pulse that the algorithm computes for **I** are possible. The second term in the max operation specified by Step 3, $\mathbf{last_p}$, implements **SCRule** by ensuring the scheduled execution pulse is at least that of the isochron before **I** by **p**. We implement **IRule** by using $\mathbf{last_p}$ for the scheduled execution pulse for each request of the isochron when Step 4 computes their initial send times.

We now show that this basic scheduling algorithm is correct.

Lemma 5.3: The basic scheduling algorithm implements **IRule** and **SCRule**.

Proof: The algorithm sends every request, **a**, of **I** such that $\mathbf{last_p} - \chi_\mathbf{a}$ is the pulse component of its initial send time. By definition, the scheduled execution time of **a** is $\tau_\mathbf{a} = \mathbf{t_s} + \chi_\mathbf{a}$. Thus, the value of $\mathbf{last_p}$ determined in Step 3 is the scheduled execution pulse of every request of **I** and the algorithm enforces **IRule**.

Since isochrons are scheduled in the order that they are issued, $\mathbf{last_p}$ tracks the scheduled execution pulse of the isochron most recently issued by **p**. Since Step 3 sets $\mathbf{last_p}$ to max($\mathbf{last_p}$, **min_send** + $\chi_\mathbf{I}$), the scheduled execution pulse of **I** is at least that of the isochron most recently issued by **p**, which includes the request most recently issued by **p**. Since every request of **I** has the same scheduled execution pulse, the algorithm enforces **SCRule**. **QED**

All of our systems can use the basic scheduling algorithm.

5.7. Delta Coherence Protocols

We now develop our correctness framework for delta coherence protocols. Our framework extends the results established in the preceding sections to systems with replication. First, we present several basic issues and concepts for delta coherence protocols, which we illustrate with the *static owner update protocol*, a protocol that extends the static early protocol to non-equidistant networks [Wil93].

5.7.1. Copy Types

Our coherence protocols use two primary types of copies, the home and local copy types. A *local copy* is a copy located at a PE that services many of the requests that the PE issues. A local copy is a cache copy in a cache coherence protocol. We use a different term since our protocols also apply to DSM. The coherence action that the issuing SIU selects for a request depends on the type of request (i.e. read or write) and the state of the local copy. In all of our protocols, the issuing SIU uses a miss action if no local copy exists or the local copy state is invalid when a request is scheduled. Each coherence unit has exactly one *home copy*, which is the destination of the initial message of any miss action. We assume the address of the coherence unit determines the location of its home copy.

The *owner copy* is a distinguished local copy that is used to track copies in our owner protocols. The term owner does not mean the owner copy has exclusive access. The home copy tracks the owner copy location since it forwards misses to that location. The names of our protocols indicate their primary coherence operation and the copy that distributes that operation. Thus, the owner copy sends update messages in the static owner update protocol, in which the owner copy location is static. In Chapter 6, we present a protocol that allows the owner copy location to change dynamically.

Our coherence protocols can create and destroy local copies dynamically. We associate two special execution events with a local copy's creation: the *instantiation event*, which initializes the new copy; and the *supplying event*, which provides the values associated with the instantiation event. We also associate a special execution event, the *destruction event*, with a local copy's destruction. A copy's *lifetime* is the logical time period between the execution times of its instantiation and destruction events. All of our proto-

cols eventually create a local copy at any issuing SIU that uses a miss action. The new copy's instantiation event usually occurs when a response to the first miss is received.

Local copies are stored in local memory coherence units, which are cache blocks or local DSM pages. Local memory coherence units that are not allocated to local copies are invalid. In all of our protocols, a local memory coherence unit is allocated for the new local copy if one does not exist when a request is scheduled. The state of the local copy becomes *filling* when the SIU schedules the first miss. It remains filling until the copy's instantiation event occurs. Most of our protocols use miss actions if the state of the local copy is filling when the issuing SIU schedules the request. Thus, the SIU can schedule additional misses since we support pipelined requests.

The *replacement policy* selects a *victim copy* in order to provide space for the new local copy when all local memory coherence units are in use. The new copy uses the space that the victim copy had occupied. Our protocols send a *release message* to the copy that tracks copies for the coherence unit of the victim copy. Thus, the owner copy receives the release message in our owner protocols. Execution of a release message removes the sender from the directory, which saves the cost of unnecessary coherence operations. Release messages can use the bounded level of service. The destruction event of the victim copy occurs when the release message is sent.

Our protocols can use any replacement policy, although the policy cannot select a home copy, an owner copy or a *reserved* local copy. A local copy is reserved if its *reservation count* is non-zero. The SIU associated with the copy increments the reservation count whenever it schedules a request to the coherence unit. It decrements the reservation count when it delivers a message for a locally issued request to the local memory process.

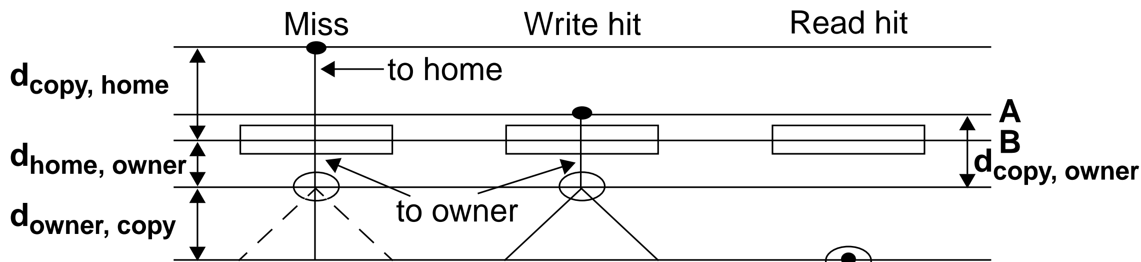


Figure 5.8: Static Owner Update Protocol Coherence Actions

5.7.2. Static Owner Update Protocol Coherence Actions

Coherence actions consist of the messages and execution events that service a request. Figure 5.8 shows schemata for the coherence actions of our static owner update protocol when $\mathbf{d_{home,\ owner}} < \mathbf{d_{copy,\ owner}}$. Other relationships between these distances only move line **A** relative to line **B**. These coherence actions use the standard level of service for all messages. The triangle of the write hit schema indicates that the owner copy sends a multicast update message. The dashed triangle of the miss schema indicates that the owner copy sends the updates if the request is a write. The bottom of each triangle indicates the logical receive time of the update at the issuing SIU. The logical receive times at other local copies depend on their distances from the owner copy. The miss actions have execution events at the owner copy and the new local copy. The write actions have execution events at every copy that is valid when it receives the update. The ovals in the schemata for these actions indicate the logical receive time at the owner copy.

The home copy, which does not execute any requests, is always invalid and its value is not maintained. Local copy states in the static owner update protocol can be: valid, filling or invalid. Hit actions are used only if the local copy state is valid. The owner copy is always valid and, thus, its SIU always uses hit actions. As discussed in Section 5.7.1, a filling local copy has not been initialized. The copy becomes valid when its instan-

tiation event occurs. An issuing SIU cannot schedule hit actions with a filling local copy, since we assume it does not know the owner copy location until its first miss returns.

The only difference between a write miss and a read miss is the sending of updates for writes. The issuing SIU sends a miss action to the home copy, which immediately forwards the request to the owner copy. If the issuing SIU is not in the directory of the owner copy, the owner copy adds its location to the directory and executes the supplying event for the location's new local copy by sending the values of the entire coherence unit to it. Read misses are always executed on the owner copy and returned to the issuing SIU.

The issuing SIU sends a write hit directly to the owner copy. As just described, the home copy forwards a write miss to the owner copy. For any write request, the owner copy sends an immediate response update message to the copies in its directory, including itself. A local copy stores the associated value when it receives the update. If the local copy no longer exists (i.e. it was released), then the update is discarded.

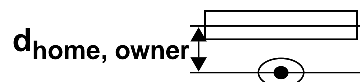


Figure 5.9: Owner Action

Figure 5.9 shows the schema of either hit action when $\mathbf{d}_{\mathbf{copy,\ owner}} = \mathbf{d}_{\mathbf{owner,\ copy}} = 0$, the distance between the owner and itself. Owner actions execute locally as soon as they are sent. Any read hit executes on the local copy as soon as it is sent. However, only the owner copy location executes a write hit on its local copy as soon as it is sent. Other locations send write hits to the owner copy location, which returns them in an update for local execution.

Each release is sent directly to the owner copy in our owner protocols. We assume that the isotach network algorithm supports the triangle inequality, which ensures that the owner does not receive a subsequent miss action before the release. The owner receives the release no later than $\mathbf{d}_{\mathbf{copy,\ owner}}$ after it is sent. The owner receives any subsequent

miss at least $\mathbf{d}_{\mathbf{copy,\ home}} + \mathbf{d}_{\mathbf{home,\ owner}}$ after the release is sent. The triangle inequality ensures that $\mathbf{d}_{\mathbf{copy,\ owner}} \leq \mathbf{d}_{\mathbf{copy,\ home}} + \mathbf{d}_{\mathbf{home,\ owner}}$, so the release is received first.

5.7.3. Scheduling and Execution Displacements with Replication

The scheduling and execution displacements of a delta coherence protocol are an extremely important aspect of its design. The protocol determines the execution displacement of each copy and associates a scheduling displacement with each coherence action. The issuing SIU selects the scheduling displacement associated with a coherence action when it uses that coherence action. Our protocols ensure that $\chi = \Phi$ for every execution event of each coherence action and, thus, $\mathbf{t}_\mathbf{e}$ always equals τ, which allows us to show that the scheduled and logical executions are equivalent, as we discussed in Section 5.5.

The issuing SIU often does not know the locations of some copies that execute the request. For example, the issuing SIU of a miss action does not know the location of the owner copy in our owner protocols. We use execution and scheduling displacements in our protocols that allow the issuing SIU to schedule the request correctly despite any incomplete knowledge about the locations of the execution events of the coherence action. In addition, we ensure the same execution distance applies to each execution event of any coherence action that has multiple execution events. Thus, our protocols can ensure that $\chi = \Phi$ for every execution event of each coherence action and, thus, $\mathbf{t}_\mathbf{e}$ always equals τ.

Table 5.1 shows the scheduling displacements and the execution distances for every execution event of each coherence action of the static owner update protocol. We derive the second and third columns of Table 5.1 directly from the descriptions of the coherence actions in Section 5.7.2. The second column indicates the copy on which the execution event is performed. The third column indicates the logical distance that mes-

Table 5.1: Owner Update Protocol Displacements and Distances

Action	Execution Event	$\sum_m d_m$	δ	Φ	χ
Miss	Owner copy	$d_{copy, home} + d_{home, owner}$	$-d_{home, owner}$	$d_{copy, home}$	$d_{copy, home}$
	Local copy (other than owner)	$d_{copy, home} + d_{home, owner} + d_{owner, loc_copy}$	$-d_{home, owner} - d_{owner, loc_copy}$	$d_{copy, home}$	$d_{copy, home}$
Write Hit	Owner copy	$d_{copy, owner}$	$-d_{home, owner}$	$d_{copy, owner} - d_{home, owner}$	$d_{copy, owner} - d_{home, owner}$
	Local copy (other than owner)	$d_{copy, owner} + d_{owner, loc_copy}$	$-d_{home, owner} - d_{owner, loc_copy}$	$d_{copy, owner} - d_{home, owner}$	$d_{copy, owner} - d_{home, owner}$
Read Hit	Issuing copy	0	$-d_{home, owner} - d_{owner, copy}$	$-d_{home, owner} - d_{owner, copy}$	$-d_{home, owner} - d_{owner, copy}$

sages travel to bring the request to the copy in column two. We emphasize that the miss and write hit actions have multiple execution events by separating the owner copy from other local copies, while the only execution event of a read hit is performed on the *issuing copy*, the local copy associated with the issuing SIU. The execution events of a read miss are the execution event of the read performed on the owner copy and, if it is the first miss, the instantiation event of the issuing copy.

For any local copy, $\delta_{\mathbf{copy}} = -\mathbf{d}_{\mathbf{home, owner}} - \mathbf{d}_{\mathbf{owner, copy}}$, which is $-\mathbf{d}_{\mathbf{home, owner}}$ for the owner. The fourth column, δ, of Table 5.1 shows the appropriate execution displacement. We add the third and fourth columns to derive the fifth column, the execution distances, $\Phi = \left(\sum_{\mathbf{m}} \mathbf{d}_{\mathbf{m}}\right) + \delta$. The sixth column shows the scheduling displacement that the issuing SIU selects when it schedules that coherence action. As we discussed, these displacements compensate for any incomplete knowledge about the locations of execution events and ensure that the same execution distance applies to each execution event of the coherence actions that have multiple execution events. Thus, $\mathbf{t_e}$ always equals τ under the static owner update protocol since $\chi = \Phi$ for every execution event, as Table 5.1 verifies.

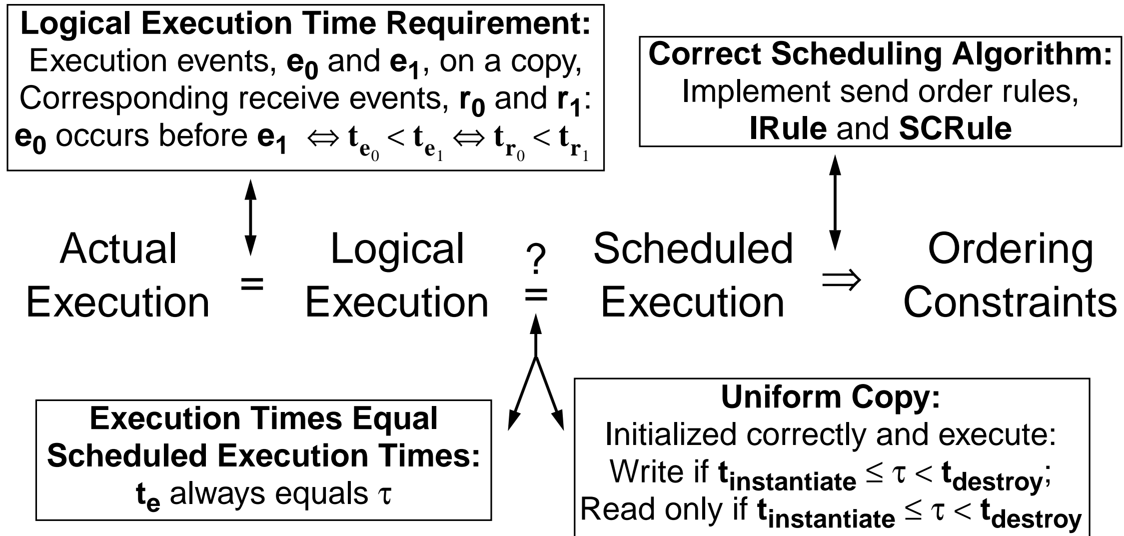


Figure 5.10: Components of Correctness Framework

5.7.4. Protocol Correctness

Figure 5.10 elaborates our correctness framework. We assign execution times that ensure the actual and logical executions are equivalent. A correct scheduling algorithm ensures an isochronous and sequentially consistent scheduled execution by Lemma 5.1. In this section, we prove that every logical execution is equivalent to its scheduled execution if $\mathbf{t_e}$ always equals τ and every copy is *uniform*.

A copy is uniform if it is initialized correctly, executes all write requests that have scheduled execution times during its lifetime and only executes read requests that have scheduled execution times during its lifetime. The scheduled execution time, τ, of a request is during the lifetime of a copy if $\mathbf{t_{instantiate}} \le \tau < \mathbf{t_{destroy}}$, where $\mathbf{t_{instantiate}}$ and $\mathbf{t_{destroy}}$ are the execution times of the instantiation and destruction events of the copy. A copy is initialized correctly if the value associated with its instantiation event is the value associated with the write request, **w**, such that $\tau_{\mathbf{w}} < \mathbf{t_{instantiate}}$ and no $\mathbf{w'}$ exists such that $\tau_{\mathbf{w}} < \tau_{\mathbf{w'}} < \mathbf{t_{instantiate}}$ for each variable of the coherence unit.

In a system with replication, ensuring $\mathbf{t_e}$ always equals τ does not ensure that the scheduled and logical executions are equivalent. Even if $\tau = \mathbf{t_e}$ for every execution event, the executions can associate different values with a read request, **r**. We now show the executions associate the same value with **r** if $\mathbf{t_e}$ always equals τ and every copy is uniform:

Lemma 5.4: Any execution, **E**, and its scheduled execution, $\mathbf{E_S}$, are equivalent if $\mathbf{t_e}$ always equals τ and all copies are uniform.

Proof: We must show **E** and $\mathbf{E_S}$ consist of the same requests and associate the same value with each request.

Every request is scheduled and every request has at least one execution event by our shared memory execution model defined in Section 2.2.3. Thus, **E** and $\mathbf{E_S}$ consist of the same requests.

$\mathbf{E_S}$ associates the same value with each write request as each execution event of the request in **E** by definition.

For any read request, **r**, $\mathbf{E_S}$ associates the value of the write request, **w**, such that $\tau_\mathbf{w} < \tau_\mathbf{r}$ and no $\mathbf{w'}$ exists such that $\tau_\mathbf{w} < \tau_\mathbf{w'} < \tau_\mathbf{r}$. Let **e** be the execution event in **E** of any read request, **r**. **E** associates the value at $\mathbf{t_e}$ of the copy on which **e** is performed. Since the copy is uniform, $\tau_\mathbf{r}$ is during the copy's lifetime. Since $\mathbf{t_e}$ always equals τ, **E** associates the value at $\tau_\mathbf{r}$ of the copy with **r**. Since $\tau_\mathbf{w} < \tau_\mathbf{r}$, either $\tau_\mathbf{w}$ is also during its lifetime or $\tau_\mathbf{w} < \mathbf{t_{instantiate}}$ and no $\mathbf{w'}$ exists such that $\tau_\mathbf{w} < \tau_\mathbf{w'} < \mathbf{t_{instantiate}}$. Since the copy is uniform, an execution event for **w** is performed on it. Since $\mathbf{t_e}$ always equals τ, write execution events on the copy occur in the scheduled execution order and, thus, its value at $\tau_\mathbf{r}$ is the value associated with **w**. Thus, **E** and $\mathbf{E_S}$ associate the same value with **r**. **QED**

Our next theorem underlies the correctness framework of our coherence protocols.

Theorem 5.2: Any execution, **E**, is isochronous and sequentially consistent if $\mathbf{t_e}$ always equals τ, all copies are uniform and the scheduling algorithm is correct.

Proof: Since the scheduling algorithm is correct, the scheduled execution, $\mathbf{E_S}$, of **E** is isochronous and sequentially consistent by Lemma 5.1. Since $\mathbf{t_e}$ always equals τ and all copies are uniform, **E** is equivalent to $\mathbf{E_S}$ by Lemma 5.4. Thus, **E** is isochronous and sequentially consistent. **QED**

We derive a simple method to prove the correctness of a delta protocol from Theorem 5.2, assuming the scheduling algorithm is correct. First we show $\chi = \Phi$ for every execution event and, thus, $\mathbf{t_e}$ always equals τ. Then we show every copy is uniform.

Figure 5.11 shows conditions we use to show that a copy is initialized correctly. The supplying event of a copy is actually a read execution event that is performed on another copy. Given that $\mathbf{t_{instantiate}} = \mathbf{t_{supply}}$, the logic of the proof of Lemma 5.4 applies to show that the value associated with that read execution event is the value associated with the correct write request if the copy on which it is performed is uniform and $\mathbf{t_e}$ always equals τ.

Associate value of write, **w**:	$\tau_{\mathbf{w}} < \mathbf{t_{instantiate}}$ and No $\mathbf{w'}$ with $\tau_{\mathbf{w}} < \tau_{\mathbf{w'}} < \mathbf{t_{instantiate}}$
Holds if:	$\mathbf{t_e}$ always equals τ, $\mathbf{t_{instantiate}} = \mathbf{t_{supply}}$ and Uniform copy supplies value

Figure 5.11: Copy Initialization

We refine our condition that a uniform copy executes exactly those writes with scheduled execution times during its lifetime. In isotach systems with replication, a write execution event is performed on a copy if, and only if, the copy receives an update for the request during its *receive lifetime*, the interval of local logical receive times that correspond to its lifetime. Thus, the receive lifetime of a copy with execution displacement $\delta_{\mathbf{copy}}$ is the period of local logical receive time between $\mathbf{t_{instantiate}} - \delta_{\mathbf{copy}}$ and $\mathbf{t_{destroy}} - \delta_{\mathbf{copy}}$. The scheduled execution time of any write request that the copy executes is during its lifetime if $\mathbf{t_e}$ always equals τ.

Every update is sent by some directory. We extend logical execution time to execution events performed on the directories in order to prove that each copy executes every write request with a scheduled execution time during its lifetime. This extension supports replication of directories. Similarly to our copies, the execution displacement, $\delta_{\mathbf{Dir}}$, of a

directory, **Dir**, determines the execution time of each execution event performed on **Dir**. For the directory of the static owner update protocol, $\delta_{\mathbf{Dir}} = -\mathbf{d}_{\mathbf{home,\ owner}}$.

Several execution events can be performed on a directory. An execution event that reads the contents of a directory determines the destinations of updates for a write request. A memory process issues an *instantiation request* (IR) in order to acquire a new local copy. The execution event, $\mathbf{add}_{\mathbf{Dir}}$, of an IR performed on a directory, **Dir**, with execution time $\mathbf{t}_{\mathbf{add}_{\mathbf{Dir}}}$ adds the location of the new copy to **Dir**. The execution event, $\mathbf{remove}_{\mathbf{Dir}}$, of a release performed on **Dir** with execution time $\mathbf{t}_{\mathbf{remove}_{\mathbf{Dir}}}$ removes the location from **Dir**. Some protocols create and destroy directories dynamically. An instantiation event initializes a directory and a destruction event destroys a directory.

A directory is *correct* if it is *complete* at all execution times during its lifetime. A directory is complete at execution time, $\mathbf{t}_{\mathbf{e}}$, if the location of any copy whose lifetime includes $\mathbf{t}_{\mathbf{e}}$ (i.e. $\mathbf{t}_{\mathbf{instantiate}_{\mathbf{copy}}} \le \mathbf{t}_{\mathbf{e}} < \mathbf{t}_{\mathbf{destroy}_{\mathbf{copy}}}$) is in the directory. A location can be in the directory although there is no copy at the location whose lifetime includes $\mathbf{t}_{\mathbf{e}}$. Thus, a directory is incorrect if, and only if, at some $\mathbf{t}_{\mathbf{e}}$ it does not include the location of some copy whose lifetime includes $\mathbf{t}_{\mathbf{e}}$. Every copy receives an update for every write with a scheduled execution time during its lifetime if all directories are correct.

Figure 5.12 shows conditions that ensure that a directory, **Dir**, is correct. **Dir** must be complete initially (i.e. at $\mathbf{t}_{\mathbf{instantiate}_{\mathbf{Dir}}}$). An $\mathbf{add}_{\mathbf{Dir}}$ must be performed on **Dir** for any copy that is created during the lifetime of **Dir** before the lifetime of the copy begins (i.e. if $\mathbf{t}_{\mathbf{instantiate}_{\mathbf{Dir}}} \le \mathbf{t}_{\mathbf{instantiate}_{\mathbf{copy}}} < \mathbf{t}_{\mathbf{destroy}_{\mathbf{Dir}}}$, then $\mathbf{t}_{\mathbf{Add}_{\mathbf{Dir}}} \le \mathbf{t}_{\mathbf{instantiate}_{\mathbf{copy}}}$). Also, the lifetime of a copy must end before a $\mathbf{remove}_{\mathbf{Dir}}$ is performed on **Dir** for it (i.e. $\forall$ $\mathbf{remove}_{\mathbf{Dir}}$ performed on **Dir**, $\mathbf{t}_{\mathbf{destroy}_{\mathbf{copy}}} \le \mathbf{t}_{\mathbf{remove}_{\mathbf{Dir}}}$). Each $\mathbf{add}_{\mathbf{Dir}}$ executes before the corresponding $\mathbf{remove}_{\mathbf{Dir}}$ since $\mathbf{t}_{\mathbf{instantiate}_{\mathbf{copy}}} < \mathbf{t}_{\mathbf{destroy}_{\mathbf{copy}}}$ by definition. However, we must ensure that

$\forall\, \mathbf{t_e}$ such that $\mathbf{t}_{\mathbf{instantiate}_{\mathbf{Dir}}} \le \mathbf{t_e} < \mathbf{t}_{\mathbf{destroy}_{\mathbf{Dir}}}$:   If $\mathbf{t}_{\mathbf{instantiate}_{\mathbf{copy}}} \le \mathbf{t_e} < \mathbf{t}_{\mathbf{destroy}_{\mathbf{copy}}}$ then **copy** $\in$ **Dir**
Holds if: Complete initially: If $\mathbf{t}_{\mathbf{instantiate}_{\mathbf{copy}}} \le \mathbf{t}_{\mathbf{instantiate}_{\mathbf{Dir}}} < \mathbf{t}_{\mathbf{destroy}_{\mathbf{copy}}}$, **copy** $\in$ **Dir** at $\mathbf{t}_{\mathbf{instantiate}_{\mathbf{Dir}}}$ If $\mathbf{t}_{\mathbf{instantiate}_{\mathbf{Dir}}} \le \mathbf{t}_{\mathbf{instantiate}_{\mathbf{copy}}} < \mathbf{t}_{\mathbf{destroy}_{\mathbf{Dir}}}$ then   execute $\mathbf{add_{Dir}}$ such that $\mathbf{t}_{\mathbf{add}_{\mathbf{Dir}}} \le \mathbf{t}_{\mathbf{instantiate}_{\mathbf{copy}}}$ $\forall$ $\mathbf{remove_{Dir}}$ performed on **Dir**, $\mathbf{t}_{\mathbf{destroy}_{\mathbf{copy}}} \le \mathbf{t}_{\mathbf{remove}_{\mathbf{Dir}}}$ If $\exists$ $\mathbf{instantiate_{next}}$ such that $\mathbf{t}_{\mathbf{destroy}_{\mathbf{copy}}} < \mathbf{t}_{\mathbf{instantiate}_{\mathbf{next}}} < \mathbf{t}_{\mathbf{destroy}_{\mathbf{Dir}}}$ and   execute $\mathbf{remove_{Dir}}$,then   execute $\mathbf{add_{Dir}}$ for **next** such that $\mathbf{t}_{\mathbf{remove}_{\mathbf{Dir}}} < \mathbf{t}_{\mathbf{add}_{\mathbf{Dir}}}$

Figure 5.12: Directory Correctness

each $\mathbf{add_{Dir}}$ executes after the $\mathbf{remove_{Dir}}$ for any preceding copy at the same location (i.e. $\mathbf{t}_{\mathbf{remove}_{\mathbf{Dir}}} < \mathbf{t}_{\mathbf{add}_{\mathbf{Dir}}}$ if $\mathbf{t}_{\mathbf{destroy}_{\mathbf{copy}}} < \mathbf{t}_{\mathbf{instantiate}_{\mathbf{next}}} < \mathbf{t}_{\mathbf{destroy}_{\mathbf{Dir}}}$).

We do not require that a $\mathbf{remove_{Dir}}$ be performed on **Dir** for every release since a complete directory can include extra locations. Similarly, if all locations are always in a directory, the directory is correct since it broadcasts its updates and, thus, sends each update to any location that requires it. However, our protocols are designed for networks in which broadcasts are expensive and, thus, our protocols benefit from the more accurate directories that our directory execution events provide.

We assume that a directory is complete initially if its lifetime begins during system initialization. In the static owner update protocol, we assume that the owner copy is the only copy whose lifetime begins during system initialization and its location is the only entry in the initial directory. We now show that throughout the lifetime of any copy in the static owner update protocol, the copy is in the directory.

Lemma 5.5: The directory of the static owner update protocol is correct.

Proof: The initial directory is complete since it includes the owner copy.

A location implicitly includes an IR in its first miss. An $\mathbf{add_{Dir}}$ adds the location to the directory when the execution event for the miss is performed on the owner copy. The instantiation event of the

new local copy is the execution event performed on the issuing copy for the miss. Table 5.1 shows that the execution times of these two execution events are equal and, thus, $\mathbf{t}_{\mathbf{add}_{\mathbf{Dir}}} = \mathbf{t}_{\mathbf{instantiate}_{\mathbf{copy}}}$.

The destruction event of any local copy occurs when the associated SIU releases it. From the execution time function of the copy, we derive that the execution time, $\mathbf{t}_{\mathbf{destroy}_{\mathbf{copy}}}$, of the destruction event is $\mathbf{t}_{\mathbf{s}_{\mathbf{rel}}} + \delta_{\mathbf{copy}} = \mathbf{t}_{\mathbf{s}_{\mathbf{rel}}} - \mathbf{d}_{\mathbf{home},\ \mathbf{owner}} - \mathbf{d}_{\mathbf{owner},\ \mathbf{copy}}$, where $\mathbf{t}_{\mathbf{s}_{\mathbf{rel}}}$ is the logical send time of the release. Since the release uses the bounded level of service, $\mathbf{t}_{\mathbf{s}_{\mathbf{rel}}} \le \mathbf{t}_{\mathbf{r}_{\mathbf{rel}}} \le \mathbf{t}_{\mathbf{s}_{\mathbf{rel}}} + \mathbf{d}_{\mathbf{copy},\ \mathbf{owner}}$, where $\mathbf{t}_{\mathbf{r}_{\mathbf{rel}}}$ is the logical receive time of the release.

The execution time, $\mathbf{t}_{\mathbf{remove}_{\mathbf{Dir}}}$, of the release on the directory is $\mathbf{t}_{\mathbf{r}_{\mathbf{rel}}} + \delta_{\mathbf{Dir}} = \mathbf{t}_{\mathbf{r}_{\mathbf{rel}}} - \mathbf{d}_{\mathbf{home},\ \mathbf{owner}}$. Since $\mathbf{t}_{\mathbf{s}_{\mathbf{rel}}} \le \mathbf{t}_{\mathbf{r}_{\mathbf{rel}}}$ and all logical distances are non-negative, $\mathbf{t}_{\mathbf{destroy}_{\mathbf{copy}}} \le \mathbf{t}_{\mathbf{remove}_{\mathbf{Dir}}}$.

Each $\mathbf{add}_{\mathbf{Dir}}$ executes after the $\mathbf{remove}_{\mathbf{Dir}}$ for any preceding copy at the same location. Let $\mathbf{t}_{\mathbf{s}_{\mathbf{IR}}}$ be the logical send time of an IR sent after a release from the same location. Since $\mathbf{t}_{\mathbf{r}_{\mathbf{rel}}} < \mathbf{t}_{\mathbf{s}_{\mathbf{IR}}} + \mathbf{d}_{\mathbf{copy},\mathbf{owner}}$, $\mathbf{t}_{\mathbf{remove}_{\mathbf{Dir}}} < \mathbf{t}_{\mathbf{s}_{\mathbf{IR}}} + \mathbf{d}_{\mathbf{copy},\mathbf{owner}} - \mathbf{d}_{\mathbf{home},\mathbf{owner}}$. The execution time of the $\mathbf{add}_{\mathbf{Dir}}$, $\mathbf{t}_{\mathbf{add}_{\mathbf{Dir}}}$, is $\mathbf{t}_{\mathbf{s}_{\mathbf{IR}}} + \mathbf{d}_{\mathbf{copy},\ \mathbf{home}}$ from Table 5.1. Since by the triangle inequality $\mathbf{d}_{\mathbf{copy},\mathbf{owner}} - \mathbf{d}_{\mathbf{home},\mathbf{owner}} \le \mathbf{d}_{\mathbf{copy},\mathbf{home}}$, $\mathbf{t}_{\mathbf{remove}_{\mathbf{Dir}}} < \mathbf{t}_{\mathbf{add}_{\mathbf{Dir}}}$. Thus, the directory of the static owner update protocol is correct since it is complete at all execution times. **QED**

We assume any copy whose lifetime begins during system initialization, such as the owner copy, has initial values that conform to the programming language semantics. We now show the static owner update protocol is correct:

Theorem 5.3: The static owner update protocol enforces isochronicity and sequential consistency if the scheduling algorithm is correct.

Proof: Since $\chi = \Phi$ for every execution event of each coherence action, $\mathbf{t}_{\mathbf{e}}$ always equals τ (see Table 5.1).

The owner copy executes every write request and, thus, any write request with a scheduled execution time during its lifetime. Since it is never destroyed, it only executes read requests with scheduled execution times during its lifetime. Since we assume that its initial values are correct, it is uniform.

We now show that **copy**, any local copy other than the owner copy, 1) is initialized correctly; 2) executes all writes with scheduled execution times during its lifetime; and 3) only executes reads with scheduled execution times during its lifetime. Thus, **copy** is uniform.

Table 5.2: Example Scheduling Displacements

Request	$d_{home,\ owner}$	$d_{owner,\ copy} = d_{copy,\ owner}$	χ
p_0: read v_0	$d_{MM_0,\ p_0} = 2$	$d_{p_1,\ p_0} = d_{p_0,\ p_1} = 3$	$-d_{home,\ owner} - d_{owner,\ copy} = -5$
p_0: write v_1	$d_{MM_0,\ p_0} = 2$	$d_{p_0,\ p_0} = 0$	$d_{copy,\ owner} - d_{home,\ owner} = -2$
p_0: write v_2	$d_{MM_1,\ p_0} = 3$	$d_{p_0,\ p_0} = 0$	$d_{copy,\ owner} - d_{home,\ owner} = -3$
p_1: write v_0	$d_{MM_0,\ p_1} = 2$	$d_{p_1,\ p_1} = 0$	$d_{copy,\ owner} - d_{home,\ owner} = -2$
p_1: read v_1	$d_{MM_0,\ p_1} = 2$	$d_{p_0,\ p_1} = d_{p_1,\ p_0} = 3$	$-d_{home,\ owner} - d_{owner,\ copy} = -5$

1) The instantiation and supplying events of **copy** are the execution events for the first miss from the location performed on the owner copy and **copy**, respectively. Thus, $\mathbf{t_{instantiate}} = \mathbf{t_{supply}}$ by Table 5.1. Thus, **copy** is initialized correctly since $\mathbf{t_e}$ always equals τ and the owner copy is uniform.

2) Since the directory is correct by Lemma 5.5 and $\mathbf{t_e}$ always equals τ, the owner copy sends an update to **copy** for each write with a scheduled execution time during the lifetime of **copy**. Since $\mathbf{t_e}$ always equals τ, the logical receive time of the update is during the receive lifetime of **copy**. Since **copy** executes any update that it receives during its receive lifetime, it executes any write with a scheduled execution time during its lifetime.

3) All read execution events performed on **copy** are for read hits. Read hits are not scheduled before its instantiation event and the reservation count ensures it is not destroyed if any already scheduled read hits have not executed. Therefore, every read executed on **copy** has a scheduled execution time during the lifetime of **copy**.

Thus, the static owner update protocol enforces isochronicity and sequential consistency by Theorem 5.2. **QED**

Example: We now use our static owner update protocol in our continuing example. Assume that $\mathbf{p_0}$ owns $\mathbf{v_1}$ and $\mathbf{v_2}$ and has a local copy of $\mathbf{v_0}$, while $\mathbf{p_1}$ owns $\mathbf{v_0}$ and has a local copy of $\mathbf{v_1}$. Assume no other local copies of $\mathbf{v_2}$ exist. In our example program, $\mathbf{p_0}$ isochronously reads $\mathbf{v_0}$ and writes $\mathbf{v_1}$, and then subsequently writes $\mathbf{v_2}$, while $\mathbf{p_1}$ isochronously writes $\mathbf{v_0}$ and reads $\mathbf{v_1}$. Table 5.2 shows the applicable scheduling displacements.

Figure 5.13 shows an example execution in which all requests happen to have the same scheduled execution pulse, **t**. The execution times of the requests by $\mathbf{p_0}$ are again earlier by their **pid-rank** component. If the real times of the execution events correspond to their logical receive times, the actual execution interleaves the processes' requests. However, it is equivalent to the scheduled execution and, thus, is isochronous and sequentially consistent. In our example execution, $\mathbf{p_0}$ reads the old value of $\mathbf{v_0}$, while $\mathbf{p_1}$ reads the new value of $\mathbf{v_1}$ and the actual execution is isochronous.

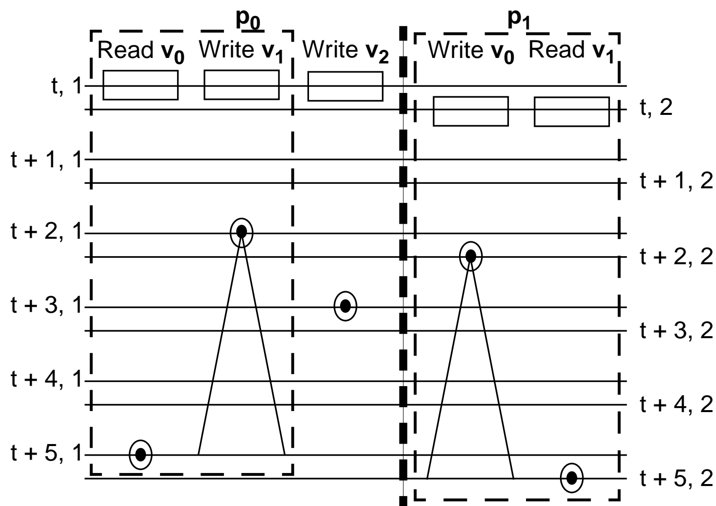


Figure 5.13: Possible Owner Update Execution

Our example demonstrates the benefit of an exclusive owner copy. Since $\mathbf{p_0}$ has the only local copy, its write to $\mathbf{v_2}$ does not require any network traffic. Note that the write to $\mathbf{v_2}$ by $\mathbf{p_0}$ is pipelined since it is sent before the read of $\mathbf{v_0}$ returns.

An advantage of the static owner update protocol is that read hits on a copy execute locally as soon as they are sent. Read requests generally account for more than 60% of shared memory requests [BaR89, WOT95]. Owner writes also execute locally as soon as they are sent and copies that are clustered around the owner derive a similar benefit for write requests. Thus, the static owner update protocol exploits cluster locality when the owner copy location is within the cluster. In Chapter 6, we present a dynamic owner update protocol that can exploit dynamically detected cluster locality.

5.7.5. Implementing Split Operations

Our coherence protocols can implement split operations [Wil93] efficiently. We require that any location that receives a sched and its corresponding assign receives the sched first. Our protocols include the issuing SIU in the destinations to which the sched is distributed. We assume the issuing SIU sends the corresponding assign after the sched returns. This assumption simplifies meeting, but does not guarantee, the requirement that a sched/assign pair arrives at each location in the correct order. We note that this assumption has little cost when used with isochronous techniques for structured atomic actions.

A *version identifier* (**vID**) associates a sched with its corresponding assign. The issuing SIU determines the **vID** when it schedules the sched. System wide process identifiers (i.e. **pid-rank**) can be **vID**'s if we limit each process to one unsubstantiated write request per variable. The issuing SIU cannot schedule a sched to the same variable until the previous sched is substantiated locally.

Our implementation of split operations uses one bit per variable in each copy to indicate whether the variable is unsubstantiated. Execution of a sched sets the associated unsubstantiated bit and stores its **vID** in the variable. Any unsubstantiated read is buffered at the copy that executes the request until it is substantiated. We use the reservation count to ensure that the lifetime of the copy does not end before the read is substantiated. We increment the reservation count for the copy if the unsubstantiated read is not a locally issued request. Note that the count already includes the read if it was locally issued.

Execution of an assign has two parts: first, it clears the unsubstantiated bit and stores the associated value in the copy if the bit is set and the **vID** of the assign matches that of the copy; second, it substantiates any buffered, unsubstantiated reads that have the

same **vID** and returns the reads to the issuing SIU's. The reservation count is decremented for each read that the assign substantiates. The assign is discarded if its **vID** does not match that of the copy or any unsubstantiated reads. No assign is improperly discarded if the protocol ensures that the copy logically receives the corresponding sched request first.

A sched is treated as a special kind of write. Thus, the issuing SIU uses either the miss or write hit action for each sched request in the static owner update protocol. The reservation count of a local copy increases when the issuing SIU schedules the sched request. The count decreases when the corresponding assign returns.

Assign requests also use write coherence actions, but with some modifications. Assign requests do not have scheduled execution times. Most assign coherence action messages do not require the standard level of service to ensure that the corresponding sched request is received first. Our protocols send each assign to any location that could have an unsubstantiated read to the corresponding **vID** in order to substantiate all reads.

In the static owner update protocol, the owner copy multicasts sched and assign updates to the locations in its directory. If a location reads the unsubstantiated **vID**, either the owner copy sent the sched update to it, or the values associated with the instantiation event of its local copy included the unsubstantiated **vID**. In either case, the location is in the owner's directory. An unsubstantiated read reserves the local copy, preventing its release. Thus, any location that reads the unsubstantiated **vID** receives the assign update.

An issuing SIU can always use the write hit action for an assign request in the static owner update protocol. Since we assume that it sends the assign after the corresponding sched returns, its local copy must be valid when it sends the assign. Thus, the owner location is known. The issuing SIU can send assigns to the owner copy at the bounded level of service. Since the issuing SIU does not send the assign before the sched

returns from the owner copy, the owner copy must receive the sched before the assign. Any technique that ensures each location receives the sched and assign updates from the owner copy in FIFO order ensures that any other local copy receives the sched first if it receives both the sched and the corresponding assign.

5.7.6. Protocol Design Space

Now, we discuss significant design choices for delta coherence protocols. We first explore options available to any coherence protocol. Our naming scheme, described in Section 5.7.1, indicates the importance of the primary type of coherence operation that the protocol uses and the copy that distributes the operations. In this thesis, we present both update and invalidation protocols and protocols with two different choices for the type of copy that distributes the operations. A distinguished local copy holds this responsibility in our owner protocols, while the protocol that we present in Chapter 8 distributes it among all local copies. Competitive protocols, limited directory protocols and dynamic page management algorithms are among the traditional options that we leave for future work.

Traditional protocols could use replicated directories. However, maintaining consistent directories is difficult without message delivery guarantees such as those provided by an isotach logical time system. The local update protocol that we present in Chapter 8 associates a copy of the directory with each local copy. Since a location that has a directory can distribute coherence operations, a location with a directory copy has special write privileges. Protocols that separate directory replication from data replication and, thus, allow a location to have those write privileges without requiring it to hold read privileges would suit some access patterns, such as that exhibited by producer/consumer variables. We leave delta protocols that separate directory and data replication for future work.

There are many design options specific to delta coherence protocols. Most of our protocols require extensibility, although our local update protocol of Chapter 8 does not. All of our protocols use the standard level of service for coherence operations, although protocols that use other levels of service for coherence operations are possible. In isotach systems, the *send discipline* of a multicast message determines the relationship of the logical send and receive times for different destinations of the multicast. In non-equidistant networks, the send discipline of coherence operations is an important design choice. The static owner update protocol uses the same logical send time for every destination of an update multicast, while our local update protocol uses the isotach invariant to ensure that the logical receive time of an update multicast is the same at each destination.

Now, we present a very basic design choice specific to delta protocols that we derive from our framework for isotach shared memory systems. A *protocol variant* adds the same constant, **C**, to every scheduling and execution displacement of the protocol. A variant does not change the original protocol in any other way. Since any execution distance, Φ, involves exactly one execution displacement and $\chi + \mathbf{C} = \Phi + \mathbf{C}$ if, and only if, $\chi = \Phi$, $\mathbf{t_e}$ always equals τ when the variant is used if and only if $\mathbf{t_e}$ always equals τ when the original protocol is used. Since the variant does not change any other aspect of the original protocol, all copies of a variant are uniform if and only if the copies of the original protocol are uniform. Thus, a variant is correct if and only if the original protocol is correct and any correct delta protocol, such as the static owner update protocol, represents a class of correct protocols. We anticipate that the complexity of using a protocol class in many isotach implementations will depend on the variant considered.

Delta coherence protocols are mutually compatible. If two protocols ensure that $\mathbf{t_e}$ always equals τ and all copies are uniform when they are used separately, then $\mathbf{t_e}$ always

equals τ and all copies are uniform when they are used concurrently for different coherence units. Thus, a correct isotach shared memory system can use different correct protocols concurrently for different coherence units.

5.7.7. Previously Identified Delta Protocols

Now, we discuss the relationship of our static owner update protocol to previously published delta coherence protocols. Williams identified the late protocol and two types of early protocols for equidistant networks [Wil93]. We have introduced new terminology that supports many new theoretical results for delta protocols. We apply this terminology to her protocols in the following discussion.

Our static owner update protocol extends her static early protocol to non-equidistant networks. In her protocol, the execution displacement of all local copies other than the owner copy was zero. This choice requires that the scheduled execution time of a miss action is the logical receive time of the response at the issuing SIU. In an equidistant network, every message travels the same distance. Thus, the issuing SIU could anticipate when the miss returns event though it does not know the location of the owner copy. Non-equidistant networks do not support this assumption, so we had to adjust the displacements of the protocol to accommodate the lack of knowledge of the issuing SIU.

Our static owner update protocol is identical to the late protocol if the home and owner copies are collocated. In this case, the home copy distributes updates. Thus, her late protocol is an instance of our static owner update protocol.

5.8. Chapter Summary

We presented a new framework that supports a unified theory for isotach shared memory systems. Williams separated the logical times of execution events that are performed on cache copies from the logical receive times at the copies. We complete the separation of execution times from the corresponding logical receive times with logical execution time. Eliminating the use of a physical canonical copy in delta coherence protocols allowed us to extend her early update protocol to non-equidistant topologies and to show that each correct delta protocol represents a class of correct protocols.

Chapter 6:
Owner Update Protocol

6.1. Introduction

In this chapter, we present an owner update protocol that can relocate the owner copy. The static owner update protocol can exploit cluster locality and producer/consumer variables when the appropriate location of the owner copy is static and known, which is unlikely in general. A migration mechanism, such a the one introduced here, allows dynamically detected access patterns to determine the owner copy location.

Our non-equidistant protocol extends the equidistant early update protocol [Wil93]. We solved several significant problems that do not arise in equidistant topologies for migration mechanisms. In Chapter 7, we present an invalidation protocol that is easily derived from our migration mechanism.

We introduce the concept of the scheduling horizon of an isotach shared memory system. This concept bounds the scheduled logical times of previously scheduled requests. We use this concept to allow the elegant execution of messages that alter local state used to schedule requests, such as the local record of the owner location.

6.2. Overview

The dynamic owner update protocol is identical to the static owner update protocol except when a migration is in progress. Migration is a special coherence action that causes

an ownership transition. This highly concurrent action does not suspend access to the coherence unit and nodes that have local copies prior to it can retain copies throughout it.

Figure 6.1 shows that two local copies are distinguished during an ownership transition. The *old owner copy* is the owner copy initially. The *new owner copy* becomes the owner copy as a result of the transition. A migration changes most scheduling and execution displacements since in general the distances that involve the owner copy change. However, the scheduling displacement of the miss action, $\mathbf{d_{copy,\ home}}$, does not change since the home copy location is static.

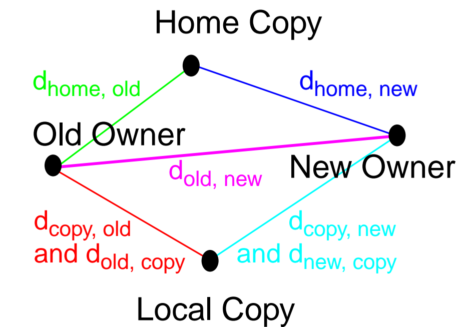


Figure 6.1: Migration Distances

Conceptually, our migration action destroys each copy of the coherence unit and creates a new one in its place. An *existing copy* is any local copy that the migration action destroys, including the old owner copy. A *replacement copy* is any local copy that replaces an existing copy, including the new owner copy.

As with any coherence action, the ownership transition has a scheduled execution time, $\tau_{\mathbf{T}}$, which is the scheduled execution time of its execution events. The lifetime of any existing copy ends at $\tau_{\mathbf{T}}$. The lifetime of any replacement copy begins at $\tau_{\mathbf{T}}$. Since the old owner copy is an existing copy and the new owner copy is a replacement copy, the owner location changes at $\tau_{\mathbf{T}}$. If the scheduled execution time of a miss or a write is before $\tau_{\mathbf{T}}$, then the old owner copy services it, while the new owner copy services it otherwise.

In addition to the three local copy states of the static protocol (valid, filling and invalid), our dynamic owner update protocol has three local copy *transition states*: migrating, disjoint, and overlapping. We describe the roles of the transition states as we describe our migration mechanism. An issuing SIU uses hit actions if its local copy is in a transition

state. The scheduling displacement that it selects for these hit actions is based on the distances applicable to the old owner copy if the scheduled execution time, τ, of the request is less than τ_T and on the distances applicable to the new owner copy otherwise. Similarly, it sends a write hit to the old owner copy if $\tau < \tau_T$ and to the new owner copy otherwise. For each local copy in a transition state, each SIU has a *transition record* that stores information it requires to schedule requests, such as τ_T and the location of the new owner copy.

Recall that in the static owner update protocol, two logical time lines are relevant to each copy: the receive and execution time lines. In a dynamic owner update protocol, a third time line is important: the replacement copy's execution time line. An existing copy's execution displacement uses distances that involve the old owner copy, while its replacement copy's execution displacement uses distances that involve the new owner copy. Thus, $\delta_{\mathbf{existing}} = -\mathbf{d}_{\mathbf{home,\ old}} - \mathbf{d}_{\mathbf{old,\ copy}}$ and $\delta_{\mathbf{replace}} = -\mathbf{d}_{\mathbf{home,\ new}} - \mathbf{d}_{\mathbf{new,\ copy}}$.

An existing copy's receive lifetime ends at $\mathbf{t}_{\mathbf{r_d}} = \tau_T - \delta_{\mathbf{existing}}$. Its replacement's receive lifetime begins at $\mathbf{t}_{\mathbf{r_i}} = \tau_T - \delta_{\mathbf{replace}}$. An existing copy and its replacement are *overlapping* if their receive lifetimes overlap, that is $\mathbf{t}_{\mathbf{r_i}} < \mathbf{t}_{\mathbf{r_d}}$, and *disjoint* otherwise. From the definitions of $\delta_{\mathbf{existing}}$ and $\delta_{\mathbf{replace}}$, we derive that the copies are overlapping if, and only if, $\mathbf{d}_{\mathbf{home,\ new}} + \mathbf{d}_{\mathbf{new,\ copy}} < \mathbf{d}_{\mathbf{home,\ old}} + \mathbf{d}_{\mathbf{old,\ copy}}$. If the distance from the home copy through the new owner copy to the node is less than through the old owner copy, the copies are overlapping. They are disjoint otherwise. Figures 6.2 and 6.3 illustrate the two cases. The hatched area of the

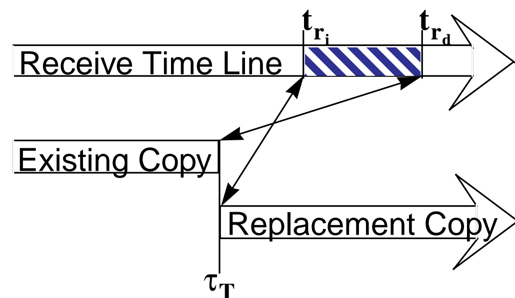


Figure 6.2: Overlapping Copies

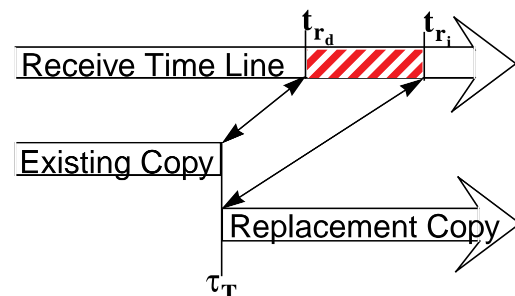


Figure 6.3: Disjoint Copies

logical receive time line in Figure 6.2 represents the interval of the local logical receive time line during which the existing and replacement copies both exist, while the hatched area in Figure 6.3 represents the interval during which neither copy exists.

Since we assume the isotach network supports the triangle inequality and the distance between a copy and itself is zero, the distance directly from the home copy to the old owner copy is never greater than the distance from the home copy to the old owner copy through the new owner copy. Thus, the old owner copy and its replacement are disjoint copies. Similarly, the new owner copy and its existing local copy are overlapping copies.

The existing copy always supplies the initial values of its replacement copy in our migration action. We use the values of the existing copy at the end of its lifetime, which ensures that $\mathbf{t_{instantiate}} = \mathbf{t_{supply}}$. Thus, the replacement copy is initialized correctly if the existing copy is uniform. When the copies are disjoint, the existing copy can easily supply the initial values since its receive lifetime ends before the receive lifetime of its replacement copy begins. When the copies are overlapping, the receive lifetime of the existing copy ends after the receive lifetime of the replacement begins. In this case, we associate values with the instantiation event of the replacement copy after the event occurs. Since the existing copy can execute a write during the overlap period, the initial values of the replacement copy are unknown when its instantiation event occurs at $\mathbf{t_{r_i}}$. Therefore, when we schedule the binding of these initial values with that event, we associate a special transition **vID**. When the receive lifetime of the existing copy ends and its final values are known at logical receive time $\mathbf{t_{r_d}}$, we assign its values to the version that the **vID** names.

We would like to use the same local memory coherence unit for each existing copy and its replacement since their lifetimes are disjoint. However, their receive lifetimes determine when we need physical storage for the copies. Since the receive lifetimes of dis-

joint copies are disjoint, we can use the same local memory coherence unit for them. However, if the copies are overlapping, we require physical storage for both copies between $\mathbf{t}_{\mathbf{r}_i}$ and $\mathbf{t}_{\mathbf{r}_d}$. Therefore, we include storage for the replacement copy in the transition record. We associate the transition **vID** with the initially unsubstantiated version of this storage.

If the copies are overlapping, we perform two actions at local logical receive time $\mathbf{t}_{\mathbf{r}_d}$. First, the existing copy provides the initial values of its replacement by assigning its final values to the version named by the transition **vID**. Second, we copy its replacement from the storage of the transition record to the local memory coherence unit used by the existing copy. Both actions are necessary in general. Updates from the new owner copy between $\mathbf{t}_{\mathbf{r}_i}$ and $\mathbf{t}_{\mathbf{r}_d}$ overwrite the initial version of the replacement copy. Thus, we must copy the storage of the transition record to the local memory coherence unit. Further, although the initial version may have been overwritten in the storage of the transition record, the assign to the initial version also substantiates any unsubstantiated reads of that version that executed during the overlap period between $\mathbf{t}_{\mathbf{r}_i}$ and $\mathbf{t}_{\mathbf{r}_d}$.

We determine the correct owner location and distances to use for a request from the relationship of its scheduled execution time, τ, to $\tau_{\mathbf{T}}$. If $\tau < \tau_{\mathbf{T}}$, the old owner location is the correct location. If $\tau > \tau_{\mathbf{T}}$, the new owner location is the correct location. Every request message includes an owner location field that is used in the overlapping case to determine on which copy to execute the request. Since the issuing SIU does not know the owner location when it schedules a miss, the home copy writes the location to which it forwards a miss in this field, while the issuing SIU writes the location that it uses to determine the scheduling displacement of a hit in this field. Lemma 6.7 shows that these rules record the correct location in this field for all requests, including hits that the issuing SIU schedules before learning about the migration.

When a node executes a request during an overlap period, it uses the owner location field to determine the copy on which to execute the request. It executes the request on the existing copy if the field has the old owner location. Otherwise, it executes the request on the replacement copy. Since the field always has the correct location, it executes the request on the copy whose lifetime includes the scheduled execution time of the request.

With disjoint copies, neither copy exists between $\mathbf{t}_{\mathbf{r}_\mathbf{d}}$ and $\mathbf{t}_{\mathbf{r}_\mathbf{i}}$. We show in Section 6.5 that the receive lifetime of any copy that executes a request includes the request's logical receive time since the request's owner location field is correct and $\mathbf{t}_\mathbf{e}$ always equals τ. Thus, no coherence actions arrive during the disjoint period.

We associate a directory copy with each owner copy. A directory at execution time $\mathbf{t}_\mathbf{e}$ is *exact* if it includes exactly the locations with a copy whose lifetime includes $\mathbf{t}_\mathbf{e}$. An exact directory is a complete directory without any spurious locations. The lifetime of the new owner directory begins at $\tau_\mathbf{T}$, when the new owner copy becomes the owner copy. Lemmas 6.4 and 6.5 show that the new owner directory is exact initially.

Our migration action ensures the correct execution of all requests that an issuing SIU schedules before it knows about the transition. The SIU must have used the old owner location and distances to schedule these requests since it did not know about the transition. We ensure that $\tau_\mathbf{T}$ is greater than the scheduled execution times of these requests, which ensures that we do not need to reschedule them.

6.3. Scheduling Horizon

We introduce the *scheduling horizon*, **H**, of an isotach shared memory system. The scheduling horizon bounds the scheduled execution time of any locally issued request rel-

ative to local logical receive times. By definition, $\mathbf{t_r} + \mathbf{H} > \tau_{\mathbf{max}}$, where $\tau_{\mathbf{max}}$ is the maximum scheduled execution time of any request already scheduled by that SIU. We can combine the scheduling horizon with the isotach invariant to bound scheduled execution times at the receiver when we send a message.

The bound provided by the scheduling horizon are useful. Remotely generated coherence actions can affect scheduling decisions by changing the state of local copies. For example, the replacement of existing local copies by our migration action affects scheduling decisions by changing the scheduling displacements. If a coherence action can affect the scheduling decision of a previously scheduled request, we must be able to reschedule requests. Since this solution is expensive, we use the scheduling horizon to prevent the problem. We delay any scheduling effects of these coherence actions by at least **H**. Since $\mathbf{t_r} + \mathbf{H} > \tau_{\mathbf{max}}$, where $\mathbf{t_r}$ is the logical receive time of the action at the node, we do not have to reschedule any requests.

We derive **H** in two parts: 1) we bound the scheduled execution time of any locally issued request relative to the minimum possible scheduled logical send time; and 2) we bound the minimum possible scheduled logical send time relative to local logical receive times. **H** is the sum of these bounds since it bounds the scheduled execution time of any locally issued request relative to local logical receive times.

The scheduling algorithm establishes the first bound. For example, in a system that uses the basic scheduling algorithm, $\mathbf{last_p} = \max(\mathbf{last_p}, \mathbf{min_send} + \chi_{\mathbf{I}})$ is the scheduled execution pulse. Since **min_send** is the pulse of the minimum possible logical send time, the scheduled execution time of any locally issued request exceeds the minimum possible scheduled logical send time by at most $\chi_{\mathbf{max}}$, the maximum scheduling displacement.

We can establish the second bound if the network is extensible. We assume an extensible network unless stated otherwise. Bounding H is more difficult, and may be impossible, in non-extensible networks. If the network supports immediate responses, **min_send** is at most one more than the current local receive pulse. Suppose a message that has a **pid-rank** component greater than the **pid-rank** of the issuing process is delivered to the issuing process immediately before it issues the last request of the isochron, so that the current local receive pulse has not changed when the isochron is scheduled. Consistency with *potential causality* requires that **min_send** is greater than the current local receive pulse. Since **min_send** is the minimum possible send pulse, **min_send** must be exactly one pulse greater than the current local receive pulse. Thus, $\mathbf{H} = \chi_{\mathbf{max}} + 1$ in a system that uses the basic scheduling algorithm and supports immediate responses.

6.4. Migration Action Details

In this section, we present details of the migration action in four parts after we discuss its basic mechanics. The simplest part changes the home copy record of the owner location. Another part replaces existing local copies with new local copies. The third part involves the instantiation of new local copies that are not replacement copies. The final part of the migration action initializes the new owner directory.

6.4.1. Basic Mechanics

We assume the new owner location is already selected in our description of the migration action. The new owner copy replaces the existing copy at that location. If there is no existing copy at the new owner location, our migration action creates one. We leave

the migration initiation policy and the selection mechanism of the new owner location for future work. Possible migration initiation policies include the owner copy observing few local requests or repeated write requests by another copy. The initiation policy might determine the new owner location. Alternatively, ownership requests could determine it.

In our initial description of the migration action, no local copy in a transition state can be a victim copy. We relax this restriction in Section 6.7. The owner copy must be valid in order to initiate a migration action. We leave for future work a protocol that allows concurrent migration actions of the same coherence unit.

The old owner changes state to migrating and multicasts an *ownership transition coherence operation* (TO) to its directory, the home copy and the new owner copy. The old owner sends the TO so that its logical receive time, $\mathbf{t}_{\mathbf{r}_{\mathbf{TO}}}$, is the same at each destination. If no local copy exists at the new owner location, then the TO to that location includes the values of the coherence unit and creates its existing copy.

We define $\tau_{\mathbf{T}}$ as $\mathbf{t}_{\mathbf{r}_{\mathbf{TO}}} + \chi_{\mathbf{T}}$, where $\chi_{\mathbf{T}}$ is the scheduling displacement of the ownership transition. The old owner sends $\chi_{\mathbf{T}} = \max(\mathbf{H}, \max(\mathbf{d}_{\mathbf{TO}}) + \mathbf{d}_{\mathbf{old,\ new}} - \mathbf{d}_{\mathbf{home,\ new}})$ as part of the TO, where $\max(\mathbf{d}_{\mathbf{TO}})$ is the maximum logical distance from any recipient of the TO to the old owner location. $\chi_{\mathbf{T}}$ is non-negative since the home copy receives the TO and $\mathbf{d}_{\mathbf{home,\ old}} + \mathbf{d}_{\mathbf{old,\ new}} \geq \mathbf{d}_{\mathbf{home,\ new}}$ by the triangle inequality. The scheduled execution time of any request scheduled prior to $\mathbf{t}_{\mathbf{r}_{\mathbf{TO}}}$ is less than $\tau_{\mathbf{T}}$ since $\chi_{\mathbf{T}} \geq \mathbf{H}$. We explain in Section 6.4.5 how using $\chi_{\mathbf{T}} \geq \max(\mathbf{d}_{\mathbf{TO}}) + \mathbf{d}_{\mathbf{old,\ new}} - \mathbf{d}_{\mathbf{home,\ new}}$, ensures the correct initialization of the new owner directory. Table 6.1 summarizes the notation for an ownership transition. We introduce the last two entries of Table 6.1 in Section 6.4.5.

Table 6.1: Ownership Transition Notation

Concept	Symbol	Definition
TO Logical Receive Time	$\mathbf{t}_{\mathbf{r}_{\mathbf{TO}}}$	Determined by locations in old owner directory
Scheduled Execution Time	$\tau_{\mathbf{T}}$	$\mathbf{t}_{\mathbf{r}_{\mathbf{TO}}} + \chi_{\mathbf{T}}$
Scheduling Displacement	$\chi_{\mathbf{T}}$	$\max(\mathbf{H}, \max(\mathbf{d}_{\mathbf{TO}}) + \mathbf{d}_{\mathbf{old,\ new}} - \mathbf{d}_{\mathbf{home,\ new}})$
Existing Copy Execution Displacement	$\delta_{\mathbf{existing}}$	$-\mathbf{d}_{\mathbf{home,\ old}} - \mathbf{d}_{\mathbf{old,\ copy}}$
Replacement Copy Execution Displacement	$\delta_{\mathbf{replace}}$	$-\mathbf{d}_{\mathbf{home,\ new}} - \mathbf{d}_{\mathbf{new,\ copy}}$
Instantiation Message Logical Receive Time	$\mathbf{t}_{\mathbf{r}_{\mathbf{i}}}$	$\tau_{\mathbf{T}} - \delta_{\mathbf{replace}}$
Destruction Message Logical Receive Time	$\mathbf{t}_{\mathbf{r}_{\mathbf{d}}}$	$\tau_{\mathbf{T}} - \delta_{\mathbf{existing}}$
Directory Message Logical Send Time	$\mathbf{t}_{\mathbf{s}_{\mathbf{DIR}}}$	$\tau_{\mathbf{T}} + \mathbf{d}_{\mathbf{home,\ new}} - \mathbf{d}_{\mathbf{old,\ new}}$
Directory Message Logical Receive Time	$\mathbf{t}_{\mathbf{r}_{\mathbf{DIR}}}$	$\tau_{\mathbf{T}} + \mathbf{d}_{\mathbf{home,\ new}}$

6.4.2. Home Copy Algorithm

The home copy algorithm ensures that the home copy forwards any miss action with a scheduled execution time less than $\tau_{\mathbf{T}}$ to the old owner and any miss action with a scheduled execution time greater than $\tau_{\mathbf{T}}$ to the new owner. The scheduled execution time of a miss cannot equal $\tau_{\mathbf{T}}$ since requests have unique tags. Table 6.2 shows the home copy algorithm of an ownership transition and Figure 6.4 shows its schema. The old owner sends the TO to the home copy. The home copy sends itself a response delayed by $\chi_{\mathbf{T}}$. When the home copy receives this response, its record of the owner location becomes the new owner location.

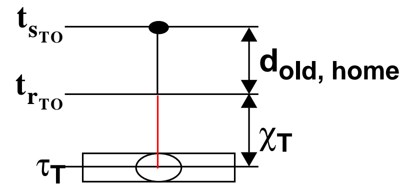


Figure 6.4: Home Action

Table 6.2: Home Copy Algorithm

Event	Logical Receive Time	Actions
Receive TO	$\mathbf{t}_{\mathbf{r}_{\mathbf{TO}}}$	Send TO response to self;
Receive TO response	$\tau_{\mathbf{T}} = \mathbf{t}_{\mathbf{r}_{\mathbf{TO}}} + \chi_{\mathbf{T}}$	Owner = new owner;

Lemma 6.1: If the scheduled execution time of a miss action is less than $\tau_{\mathbf{T}}$, then the home copy forwards it to the old owner copy. Otherwise, the home copy forwards the miss action to the new owner copy.

Proof: The scheduled execution time of a miss action is $\mathbf{t_s} + \mathbf{d_{copy,\ home}}$ since its scheduling displacement is $\mathbf{d_{copy,\ home}}$. Since the issuing SIU sends the miss action to the home copy, the home copy receives each miss at its scheduled execution time. The home copy forwards each miss action at its scheduled execution time since the forwarding message is an immediate response. Since the home copy changes its record of the owner location to the new owner location at $\tau_{\mathbf{T}} = \mathbf{t}_{\mathbf{r_{TO}}} + \chi_{\mathbf{T}}$, the lemma follows. **QED**

The owner location field of any miss has the correct location by Lemma 6.1.

6.4.3. Replacement of Existing Copies

Table 6.3 shows the local copy algorithm that replaces the existing local copies. Any location that has a valid local copy when it receives the TO, including those of the new and old owner copies, executes this algorithm. When the SIU receives the TO at $\mathbf{t}_{\mathbf{r_{TO}}}$, it allocates and initializes a transition record, sets the state of its copy to migrating and sends two delayed responses to itself, the *destruction message* and the *instantiation message*. The destruction message causes the destruction event of its existing local copy, while the instantiation message causes the instantiation event of its replacement.

We want the delays of these messages to make their execution times $\tau_{\mathbf{T}}$. Since the destruction message destroys the existing copy, $\delta_{\mathbf{existing}}$ is the execution displacement that applies to it. Thus, $\tau_{\mathbf{T}} - \delta_{\mathbf{existing}} = \mathbf{t}_{\mathbf{r_{TO}}} + \chi_{\mathbf{T}} - \delta_{\mathbf{existing}}$ should be its logical receive time, $\mathbf{t}_{\mathbf{r_d}}$, which requires a logical delay of $\chi_{\mathbf{T}} - \delta_{\mathbf{existing}}$. Similarly, we delay $\mathbf{t}_{\mathbf{r_i}}$, the logical receive time of the instantiation message, by $\chi_{\mathbf{T}} - \delta_{\mathbf{replace}}$. These delays are non-negative since $\chi_{\mathbf{T}} \geq 0$ (see Section 6.4.1), while $\delta_{\mathbf{existing}} = -\mathbf{d_{home,\ old}} - \mathbf{d_{old,\ copy}} \leq 0$ and $\delta_{\mathbf{replace}} = -\mathbf{d_{home,\ new}} - \mathbf{d_{new,\ copy}} \leq 0$. We now prove that these delays are correct.

Table 6.3: Local Copy Algorithm

Execution Event	t_r	δ	t_e	Actions
Execute TO	$\mathbf{t_{r_{TO}}}$	$\delta_{existing}$	$\mathbf{t_{r_{TO}}} + \delta_{existing}$	Send destruction message to self; Send instantiation message to self; Allocate and initialize transition record; State = migrating;

Destroy existing copy	$\mathbf{t_{r_d}}$	$\delta_{existing}$	τ_T	If (state is migrating) {
				State = disjoint; }
				Else /* state is overlapping */ {
				State = valid;
				Assign to initial version of replacement;
				Copy transition storage to main storage;
				Owner = new owner;
				Discard transition record; }

Instantiate replacement copy	$\mathbf{t_{r_i}}$	$\delta_{replace}$	τ_T	If (state is migrating) {
				State = overlapping; }
				Else /* state is disjoint */ {
				State = valid;
				Owner = new owner;
				Discard transition record; }

Execute concurrent request, **a**	$\mathbf{t_{r_a}}$	$\delta_{existing}$ or $\delta_{replace}$	τ_a	If ((state is overlapping) and
				(owner is not request owner)) {
				Execute on replacement copy; }
				Else {
				Execute on existing copy; }

Lemma 6.2: The lifetime of any existing copy ends at τ_T. The lifetime of any replacement copy begins at τ_T.

Proof: The lifetime of a copy ends at $\mathbf{t_{destroy}}$, the execution time of its destruction event. For any existing copy, $\mathbf{t_{destroy}}$ is the sum of the logical receive time of its destruction message and its execution displacement, $\mathbf{t_{r_d}} + \delta_{\mathbf{existing}} = \tau_T - \delta_{\mathbf{existing}} + \delta_{\mathbf{existing}}$. Thus, the lifetime of any existing copy ends at τ_T.

Similarly, the lifetime of a copy begins at $\mathbf{t_{instantiate}}$, the execution time of its instantiation event. For any replacement copy, $\mathbf{t_{instantiate}}$ is $\mathbf{t_{r_i}} + \delta_{\mathbf{replace}} = \tau_T - \delta_{\mathbf{replace}} + \delta_{\mathbf{replace}}$. Thus, the lifetime of any replacement copy begins at τ_T. **QED**

Figure 6.5 shows the schema for the local copy algorithm when the copies are disjoint, while Figure 6.6 shows the schema when the copies are overlapping. The only difference between these schemata is the order of the receive events of the instantiation and destruction messages. In both cases, the old owner sends the TO to the associated SIU.

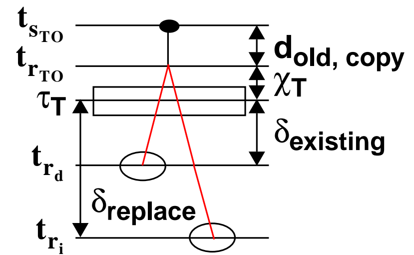


Figure 6.5: Disjoint Action

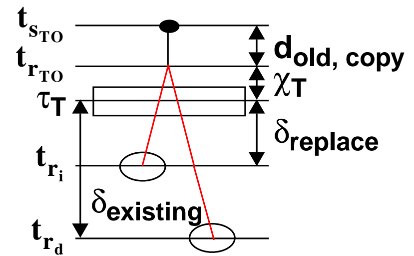


Figure 6.6: Overlapping Action

The SIU sends the virtual responses to itself, allocates and initializes a transition record, and changes the local copy state to migrating, as previously described.

The actions of the local copy algorithm for the instantiation and destruction messages depend on the order in which they are received. We encode this order in the local copy state. The state remains migrating until one of these messages is received. We change the state to disjoint when the destruction message is received first, and to overlapping when the instantiation message is received first. The local copy state does not change again until the second message is received since a local copy in a transition state cannot be a victim copy.

When the destruction message is received first, the copies are disjoint, so we change the local copy state to disjoint. When the node receives its instantiation message, we change the local copy state to valid and the local record of the owner location to the new owner location. We then discard the transition record.

When the instantiation message is received first, the copies are overlapping so we change the local copy state to overlapping. When the node receives its destruction message, we change the local copy state to valid and assign the existing copy values to the initial version of the replacement copy. These values substantiate any reads of that version and any variables of the storage of the transition record that have not been overwritten by an update from the new owner copy. We then copy this storage into the local memory coherence unit used by the existing copy, change the local record of the owner location to the new owner location and discard the transition record.

The last row of Table 6.3 shows how we determine the copy on which to execute a request that has a logical receive time during an overlap period. If the request's owner location field equals the local record of the owner location, it is executed on the existing copy. Otherwise, the request's owner location field must equal the new owner location and it is executed on the replacement copy.

6.4.4. Instantiation of New Local Copies

Instantiating a new local copy during an ownership transition is difficult because it must be replaced and the initial new owner directory must include its location. We avoid these difficulties by not instantiating any new local copies during ownership transitions. Therefore, the supplying event for a new local copy is not performed on the old owner copy after it begins sending the TO. Instead, we separate the instantiation of new local copies from the service of miss requests during an ownership transition.

After it begins sending the TO, the old owner copy continues to receive miss requests, which it services at their scheduled execution times. The old owner copy services each read miss by sending its value to the issuing SIU and each write miss by sending an update multicast to its directory. If the issuing SIU is not in its directory, the old owner copy sends a *distinguished update* to the SIU. A distinguished update decrements the reservation count for the coherence unit but is never stored in the new local copy, i.e. it satisfies the read or write request but does not supply a new copy.

The issuing SIU explicitly indicates if a miss includes an instantiation request (IR). Pipelined misses, which the SIU uses until the new local copy is instantiated, do not include an IR. Ordinarily, the owner copy executes the supplying event for the new copy when it services a miss that includes an IR. During an ownership transition, the old owner

Table 6.4: Separated IR Action

Execution Event	t_r	δ	t_e
Supplying event	$\max(\mathbf{t}_{\mathbf{r}_{\mathbf{IR}}}, \mathbf{t}_{\mathbf{r}_{\mathbf{i}}})$	$-\mathbf{d}_{\mathbf{home,\ new}}$	$\max(\mathbf{t}_{\mathbf{r}_{\mathbf{IR}}}, \mathbf{t}_{\mathbf{r}_{\mathbf{i}}}) - \mathbf{d}_{\mathbf{home,\ new}}$
Instantiation event	$\max(\mathbf{t}_{\mathbf{r}_{\mathbf{IR}}}, \mathbf{t}_{\mathbf{r}_{\mathbf{i}}}) + \mathbf{d}_{\mathbf{new,\ copy}}$	$-\mathbf{d}_{\mathbf{home,\ new}} - \mathbf{d}_{\mathbf{new,\ copy}}$	$\max(\mathbf{t}_{\mathbf{r}_{\mathbf{IR}}}, \mathbf{t}_{\mathbf{r}_{\mathbf{i}}}) - \mathbf{d}_{\mathbf{home,\ new}}$

copy separates the IR from the miss. It forwards the IR to the new owner copy at the bounded level of service. The issuing SIU location is not added to the old owner directory.

The execution time of any supplying event performed on the new owner copy must be at least $\tau_{\mathbf{T}}$, when its lifetime begins. The logical receive time that corresponds to $\tau_{\mathbf{T}}$ at the new owner copy is $\mathbf{t}_{\mathbf{r}_{\mathbf{i}}} = \tau_{\mathbf{T}} - \delta_{\mathbf{replace}} = \mathbf{t}_{\mathbf{r}_{\mathbf{TO}}} + \chi_{\mathbf{T}} + \mathbf{d}_{\mathbf{home,\ new}}$, the logical receive time of its instantiation message. The new owner copy buffers any separated IR that it receives before $\mathbf{t}_{\mathbf{r}_{\mathbf{i}}}$. When the new owner copy receives its instantiation message, it sends an instantiation multicast for these requests at the standard level of service.

The new owner copy can receive separated IR's after it receives its instantiation message. It can also receive IR's that are included with miss requests that the home copy forwards to it. For any of these IR's, the new owner copy performs an $\mathbf{add}_{\mathbf{Dir}}$ execution event on its directory and sends its values to the location at the standard level of service.

Table 6.4 shows the logical times and displacements relevant to the execution events of the separated IR action, where $\mathbf{t}_{\mathbf{r}_{\mathbf{IR}}}$ is the logical receive time of the IR at the new owner copy. We now prove that the execution times of the supplying event and instantiation event are equal when the IR is separated from the miss.

Lemma 6.3: If the separated IR is used, then $\mathbf{t}_{\mathbf{instantiate}} = \mathbf{t}_{\mathbf{supply}}$.

Proof: If $\mathbf{t}_{\mathbf{r}_{\mathbf{IR}}} < \mathbf{t}_{\mathbf{r}_{\mathbf{i}}}$, then the new owner copy buffers the IR until $\mathbf{t}_{\mathbf{r}_{\mathbf{i}}}$. The execution time of the supplying event is $\mathbf{t}_{\mathbf{r}_{\mathbf{i}}} - \mathbf{d}_{\mathbf{home,\ new}}$ since the execution displacement of the new owner copy is $-\mathbf{d}_{\mathbf{home,\ new}}$. Since the new owner copy uses the standard level of service for the instantiation multicast, the issuing SIU receives the message at

$\mathbf{t_{r_i}} + \mathbf{d_{new,\,copy}}$. Since the execution displacement of the new local copy is $-\mathbf{d_{home,\,new}} - \mathbf{d_{new,\,copy}}$, $\mathbf{t_{instantiate}} = \mathbf{t_{supply}}$.

Otherwise, the new owner copy executes the supplying event when it receives the IR. Thus, $\mathbf{t_{r_{iR}}} - \mathbf{d_{home,\,new}}$ is its execution time. Since the new owner copy uses the standard level of service to return the IR, the issuing SIU receives it at $\mathbf{t_{r_{iR}}} + \mathbf{d_{new,\,copy}}$. We again add the execution displacement of the new local copy to derive that $\mathbf{t_{instantiate}} = \mathbf{t_{supply}}$. **QED**

The separated IR can result in unreserved local copies in the filling state. The replacement policy cannot select an unreserved filling local copy as a victim copy since we cannot differentiate the response to its IR from that of a subsequently requested copy. We can relax this restriction if we associate a generation number with the new local copy. We do not detail the relaxed mechanism further.

Since the new owner location has overlapping copies, it can receive its destruction message after a separated IR or an IR included with a miss. The new owner location assigns the final values of its existing copy to the initial version of its replacement copy when it receives its destruction message. Therefore, the transition **vID** can represent the initial value of new local copies for which the supplying event is performed on the new owner copy. In order to substantiate the initial version of these copies, the new owner location sends an update multicast to its entire directory when it receives its destruction message. This update assigns the final values of its existing copy to the initial version of any new local copy that the transition **vID** names and to any reads of that version.

6.4.5. New Owner Directory Initialization

Table 6.5 shows the execution events of our algorithm that ensures the new owner directory is exact initially. Figure 6.7 shows the schema that applies to it. The old owner

Table 6.5: New Owner Directory Initialization Algorithm

Execution Event	t_r	δ	t_e	Actions
Read old owner directory	$t_{s_{DIR}}$	$\delta_{existing}$ = $-d_{home, old}$	$t_{s_{DIR}} - d_{home, old}$	Message directory = directory; Send directory message to new owner copy;
Instantiate new owner directory	$t_{r_{DIR}}$	$\delta_{replace}$ = $-d_{home, new}$	τ_T	Directory = message directory ∪ destinations of instantiation multicast;

copy sends the contents of its directory at $t_{s_{DIR}} = t_{r_{TO}} + \chi_T + d_{home, new} - d_{old, new}$ in the directory message. Although a copy cannot be destroyed after it receives the TO, the old owner copy can receive releases after it begins sending the TO. We now show that before the old owner copy sends its directory, it executes any release that is sent to it.

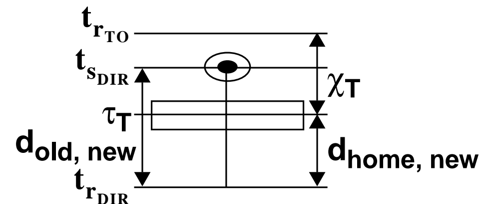


Figure 6.7: Directory Action

Lemma 6.4: Any release sent to the old owner copy is received by $t_{s_{Dir}}$.

Proof: We first show that $t_{s_{DIR}} \geq t_{r_{TO}} + d_{copy, old} \geq t_{s_{TO}}$ where **copy** is any recipient of the TO. Since $\chi_T \geq \max(d_{TO}) + d_{old,new} - d_{home,new}$ and $t_{s_{DIR}} = t_{r_{TO}} + \chi_T + d_{home, new} - d_{old, new}$ by definition, $t_{s_{DIR}} \geq t_{r_{TO}} + \max(d_{TO})$. The desired relation follows since $\max(d_{TO}) \geq d_{copy, old}$ by definition.

If the TO is not sent to the released copy's location, then a $\mathbf{remove_{Dir}}$ for the release was performed on the old owner directory before $t_{s_{TO}}$. Thus, the release was received before $t_{s_{Dir}}$.

If the TO is sent to the released copy's location, then the release is sent by $t_{r_{TO}}$ since a local copy in a transition state cannot be a victim copy. Therefore, the old owner copy receives the release by $t_{r_{TO}} + d_{copy, old}$ and, thus, $t_{s_{Dir}}$. **QED**

If the old owner directory is correct, the directory message provides exactly the locations of the replacement copies. The destinations of the instantiation multicast that the new owner copy sends are the only other locations that have a copy whose lifetime includes τ_T and, thus, must be in the new owner directory initially. The instantiation event of the new owner directory, which occurs when the directory message is executed, initial-

izes it to the message directory plus the destinations of the instantiation multicast. Every location that has a copy whose lifetime includes $\tau_{\mathbf{T}}$ is in the initial new owner directory.

Lemma 6.5: The new owner directory is complete initially if the old owner directory is correct.

Proof: Let $\mathbf{t}_{\mathbf{instantiate}_{\mathbf{Dir}}}$ be the execution time of the instantiation event of the new owner directory. We must show that any location that has a copy such that $\mathbf{t}_{\mathbf{instantiate}_{\mathbf{copy}}} \le \mathbf{t}_{\mathbf{instantiate}_{\mathbf{Dir}}} < \mathbf{t}_{\mathbf{destroy}_{\mathbf{copy}}}$ is in the new owner directory at $\mathbf{t}_{\mathbf{instantiate}_{\mathbf{Dir}}}$.

First, we show $\mathbf{t}_{\mathbf{instantiate}_{\mathbf{Dir}}}$ equals $\tau_{\mathbf{T}}$. The execution displacement of the new owner directory is $-\mathbf{d}_{\mathbf{home,\ new}}$. The new owner directory is instantiated when the directory message is received. Since $\mathbf{t}_{\mathbf{r}_{\mathbf{DIR}}} = \mathbf{t}_{\mathbf{r}_{\mathbf{TO}}} + \chi_{\mathbf{T}} + \mathbf{d}_{\mathbf{home,\ new}}$, $\mathbf{t}_{\mathbf{instantiate}_{\mathbf{Dir}}} = \mathbf{t}_{\mathbf{r}_{\mathbf{TO}}} + \chi_{\mathbf{T}} = \tau_{\mathbf{T}}$.

We now show that if the lifetime of a copy includes $\tau_{\mathbf{T}}$, then the location of the copy is in the initial new owner directory.

The lifetime of the new local copy at any destination of the instantiation multicast begins at $\tau_{\mathbf{T}}$. We construct the initial owner directory to include these locations.

Any other copy whose lifetime includes $\tau_{\mathbf{T}}$ is a replacement copy. Since $\mathbf{t}_{\mathbf{s}_{\mathbf{DIR}}} \ge \mathbf{t}_{\mathbf{s}_{\mathbf{TO}}}$ and each replacement copy location receives the TO, the directory message includes the location of any replacement copy. Thus, the new owner directory is complete initially. **QED**

Lemma 6.5 shows our new owner directory initialization algorithm is correct, while Lemma 6.4 shows that no releases are lost.

6.5. Protocol Correctness

The correctness of our dynamic owner update protocol is derived primarily from that of the static owner update protocol. Any execution without migration actions is identical to an execution of the static owner protocol. Thus, the execution is isochronous and sequentially consistent. This section shows that executions remain isochronous and sequentially consistent. in the presence of migration actions.

A *transition aware request* is any request scheduled when the local copy of its issuing SIU is in a transition state. The scheduling displacement of a correctly scheduled transition aware request is based on the old owner location if $\tau < \tau_{\mathbf{T}}$ and on the new owner location otherwise, which we now show the basic scheduling algorithm does:

Lemma 6.6: The basic scheduling algorithm schedules every transition aware request correctly.

Proof: There are two possible scheduling displacements that determine $\chi_{\mathbf{a}_{MAX}}$ in Step 1 of the algorithm if the local copy is in a transition state. For a read request, the possible scheduling displacement based on the old owner location is $-\mathbf{d}_{\mathbf{home,\,old}} - \mathbf{d}_{\mathbf{old,\,copy}}$, while it is $-\mathbf{d}_{\mathbf{home,\,new}} - \mathbf{d}_{\mathbf{new,\,copy}}$ based on the new owner location. For a write request, $\mathbf{d}_{\mathbf{copy,\,old}} - \mathbf{d}_{\mathbf{home,\,old}}$ and $\mathbf{d}_{\mathbf{copy,\,new}} - \mathbf{d}_{\mathbf{home,\,new}}$ are the possible scheduling displacements. In Step 4, we then compare $\tau_{\mathbf{T}}$ with τ, the scheduled execution time of the request that is implied by the value of $\mathbf{last_p}$ determined in Step 3. If $\tau < \tau_{\mathbf{T}}$, the algorithm bases the scheduling displacement on the old owner location and on the new owner location otherwise. Thus, it schedules every transition aware request correctly. **QED**

Many other scheduling algorithms exist that schedule transition aware requests correctly. We do not discuss the choice of scheduling algorithms further. If a request is not a transition aware request, its owner location field is correct, as we now show:

Lemma 6.7: The owner location field of every request has the correct location if every transition aware request is scheduled correctly.

Proof: Recall that the old owner location is the correct location if $\tau < \tau_{\mathbf{T}}$ where τ is the scheduled execution time of the request. Otherwise, the new owner location is the correct location.

The home copy records the location to which it forwards the request in the owner location field of any miss. Thus, the field of any miss has the correct location by Lemma 6.1.

The issuing SIU records the owner location that it uses to determine the scheduling displacement in the owner location field of any hit. We divide hit actions into three groups: 1) those scheduled before $\mathbf{t}_{\mathbf{r}_{TO}}$; 2) those scheduled after $\mathbf{t}_{\mathbf{r}_{TO}}$ but before the transition completes for the issuing SIU; 3) those scheduled after the transition completes for the issuing SIU.

1) The issuing SIU uses the old owner location to determine the scheduling displacement of any hit action scheduled before $\mathbf{t}_{\mathbf{r}_{TO}}$. Since $\chi_{\mathbf{T}} \geq \mathbf{H}$ by definition, $\tau < \tau_{\mathbf{T}}$. Thus, the owner location field of the request has the correct location.

2) Any hit action scheduled after $\mathbf{t}_{\mathbf{r}_{TO}}$ but before the transition completes for the issuing SIU is scheduled when the local copy is in a transition state and, thus, is a transition aware request. By assumption, the owner location field of the request is correct.

3) The issuing SIU uses the new owner location to determine the scheduling displacement of any hit action scheduled after the transition completes for the issuing SIU. Thus, the owner location field of the request has the correct location if $\tau > \tau_{\mathbf{T}}$.We now show that $\tau = \mathbf{t}_{\mathbf{s}} + \chi > \mathbf{t}_{\mathbf{r}_{TO}} + \chi_{\mathbf{T}} = \tau_{\mathbf{T}}$ where $\mathbf{t}_{\mathbf{s}}$ is the initial send time of the request and χ is its scheduling displacement.

The issuing SIU's local copy is either a replacement copy or a copy for which the supplying event was performed on the new owner copy. In either case, $\mathbf{t}_{\mathbf{s}} > \mathbf{t}_{\mathbf{r}_{TO}} + \chi_{\mathbf{T}} + \mathbf{d}_{\mathbf{home,\,new}} + \mathbf{d}_{\mathbf{new,\,copy}}$ since $\mathbf{t}_{\mathbf{s}}$ must be greater than the logical receive time of the message that caused the instantiation event of the copy. If the request is a read, then $\chi = \delta_{\mathbf{replace}} = -\mathbf{d}_{\mathbf{home,\,new}} - \mathbf{d}_{\mathbf{new,\,copy}}$. If it is a write, then $\chi = \mathbf{d}_{\mathbf{new,\,copy}} - \mathbf{d}_{\mathbf{home,\,new}} \geq \delta_{\mathbf{replace}}$. Thus, $\tau > \tau_{\mathbf{T}}$ and the owner location field of the request has the correct location. **QED**

We can now show that $\chi = \Phi$ for every execution event of each coherence action.

Lemma 6.8: The property "$\mathbf{t}_{\mathbf{e}}$ always equals τ" holds assuming every transition aware request is scheduled correctly.

Proof: The execution distance, Φ, of each execution event of any request is determined by the same owner location as its scheduling displacement, χ, since the request's owner location has the correct location by Lemma 6.7. Table 5.1 shows $\chi = \Phi$ and, thus, $\tau = \mathbf{t}_{\mathbf{e}}$. **QED**

We can now show that the final values of an existing copy always determine the initial values of its replacement:

Lemma 6.9: For any replacement copy, $\mathbf{t}_{\mathbf{instantiate}} = \mathbf{t}_{\mathbf{supply}}$ if every transition aware request is scheduled correctly.

Proof: We show that the destruction event of any existing copy is the supplying event of its replacement and, thus, $\mathbf{t}_{\mathbf{supply}} = \tau_{\mathbf{T}} = \mathbf{t}_{\mathbf{instantiate}}$ by Lemma 6.2. The destruction event of any existing copy occurs when the destruction message is executed.

If the copies are disjoint, the replacement copy initially uses the local memory coherence unit that the existing copy had used. Since the owner location field of any write is correct by Lemma 6.7, the scheduled execution time of any write is during the lifetime of the owner copy that distributes its updates. Since $\mathbf{t_e}$ always equals τ by Lemma 6.8, no node receives any updates during a disjoint period. Thus, the local memory coherence unit holds the final value of the existing copy at $\mathbf{t_{r_i}}$ and the existing copy's destruction event is its replacement's supplying event when the copies are disjoint.

For overlapping copies, the execution of the destruction message assigns the final value of the existing copy to the initial version of the replacement copy. Thus, the destruction event of the existing copy is also the supplying event of its replacement copy. **QED**

In order to show that every copy is uniform, we divide the scheduled execution time line of each coherence unit into epochs with a unique owner copy. By Lemma 6.2, the lifetime of each local copy belongs to exactly one of these *owner epochs*. The owner copy of the $\mathbf{i^{th}}$ owner epoch is the new owner of the $\mathbf{(i\text{-}1)^{st}}$ migration action and the old owner of the $\mathbf{i^{th}}$ migration action.

Lemma 6.10: All copies in the dynamic owner update protocol are uniform if every transition aware request is scheduled correctly.

Proof: We use induction on the number of owner epochs to show that every directory is correct and all copies are uniform.

Basis: Consider the first owner epoch. The first owner directory is correct by the same logic that showed the directory of the static protocol is correct in Lemma 5.5. The first owner distributes updates for all writes with scheduled execution times during its owner epoch by Lemma 6.7. The lifetimes of the existing copies of the first migration action end at its τ_T by Lemma 6.2. Thus, the logic of Theorem 5.3 that showed the copies of the static protocol are uniform applies and all copies of the first owner epoch are uniform.

Inductive step: Consider the $\mathbf{n^{th}}$ owner epoch. We assume that the $\mathbf{(n\text{-}1)^{st}}$ owner directory is correct and all copies of the $\mathbf{(n\text{-}1)^{st}}$ owner epoch are uniform in order to show that the $\mathbf{n^{th}}$ owner directory is correct and all copies of the $\mathbf{n^{th}}$ owner epoch are uniform.

Since the $\mathbf{(n\text{-}1)^{st}}$ owner directory is correct, the $\mathbf{n^{th}}$ owner directory is complete initially by Lemma 6.5. The remainder of the logic of Lemma 5.5 applies and, thus, the $\mathbf{n^{th}}$ owner directory is correct.

Since $\mathbf{t_e}$ always equals τ by Lemma 6.8 and the $\mathbf{n^{th}}$ owner directory is correct, any copy of the $\mathbf{n^{th}}$ owner epoch executes every write request with a scheduled execution time during its lifetime. Since the owner location field of every read request is correct by Lemma 6.7, no copy of the $\mathbf{n^{th}}$ owner epoch executes any read with a scheduled execution time outside its lifetime. Thus, any copy of the $\mathbf{n^{th}}$ owner epoch is uniform if it is initialized correctly.

Any replacement copy of the $\mathbf{n^{th}}$ owner epoch, including the $\mathbf{n^{th}}$ owner copy, is initialized by an existing copy of the $\mathbf{(n\text{-}1)^{st}}$ migration action, which is uniform by the inductive hypothesis. Since $\mathbf{t_{instantiate}} = \mathbf{t_{supply}}$ by Lemma 6.9 and $\mathbf{t_e}$ always equals τ by Lemma 6.8, the copy is initialized correctly.

Any other copy of $\mathbf{n^{th}}$ owner epoch is initialized by the $\mathbf{n^{th}}$ owner copy. Since $\mathbf{t_{instantiate}} = \mathbf{t_{supply}}$ by either Lemma 6.3 or Table 5.1 and $\mathbf{t_e}$ always equals τ by Lemma 6.8, any copy of $\mathbf{n^{th}}$ owner epoch that is not a replacement copy is initialized correctly. **QED**

We can now easily prove that the protocol is correct:

Theorem 6.1: The dynamic owner update protocol enforces isochronicity and sequential consistency if the scheduling algorithm is correct and every transition aware request is scheduled correctly.

Proof: Every copy is uniform by Lemma 6.10 and $\mathbf{t_e}$ always equals τ by Lemma 6.8 if every transition aware request is scheduled correctly. Thus, by Theorem 5.2, the protocol enforces isochronicity and sequential consistency if the scheduling algorithm is correct. **QED**

Thus, our highly concurrent migration action moves the owner copy while still maintaining isochronicity and sequential consistency.

6.6. Split Operations and Migration

The migration action has implications for our implementation of split operations. We assume that all assigns use the standard level of service. If different copies distribute a

sched and the corresponding assign, the triangle inequality ensures that the copy that distributes the assign receives the sched before the assign.

The old owner copy must receive any assign that it distributes before the destruction event of its directory occurs at $\mathbf{t}_{\mathbf{r}_d} = \mathbf{t}_{\mathbf{r}_{TO}} + \chi_T + \mathbf{d}_{\mathbf{home, old}}$. Therefore, the issuing SIU sends any assign to the new owner copy after $\mathbf{t}_{\mathbf{r}_{TO}} + \chi_T + \mathbf{d}_{\mathbf{home, old}} - \mathbf{d}_{\mathbf{copy, old}}$. Any assign that is sent earlier is sent to the new owner copy if it distributed the corresponding sched; otherwise the assign is sent to the old owner copy.

Before the new owner copy receives the directory message, it can receive assigns for which the old owner copy distributed the corresponding sched. It distributes these assigns when it receives the directory message in a single multicast.

A new local copy that a separated IR instantiates can receive a distinguished update for a local sched before it is instantiated. It can send the assign directly to the new owner if the distinguished update includes that location. Alternatively, another application of the triangle inequality shows that the assign can be sent through the home copy.

6.7. Releases During Migration

We can eliminate our restriction on the release of a local copy in a transition state. We send a release for a copy in a transition state to both the new and old owner locations. The new owner copy can receive this release before the directory message. The new owner copy collects these releases in a temporary release directory. These releases are sent before any separated IR since the local copy is in a transition state. Therefore, we subtract the release directory from the message directory and then add the destinations of the instantiation multicast to obtain the initial new owner directory.

Any release of a local copy that the new owner copy receives after the directory message executes correctly. Any such release is sent directly to the new owner copy. Correct execution of the release requires that the new owner copy receives the release before any subsequent IR from the same location. If the home copy forwards the IR directly to the new owner copy, the triangle inequality ensures that the new owner copy receives the release first, as in the static protocol. If the separated IR is used, another application of the triangle inequality shows a separated IR travels at least as far to the new owner location as an IR that the home copy forwards directly to the new owner copy and, thus, the release is received before the separated IR.

6.8. Optimizations

Other optimizations of our migration action are possible. For instance, the new owner can use the directory message as its instantiation message, since they have the same logical receive time. This change probably has little effect on performance since virtual messages do not create any network traffic and only use buffer space at the SIU.

We can reduce the network traffic of the migration action slightly. The new owner sends the assign for the transition **vID** to its entire directory. Only new local copies that the new owner instantiates can require this update. Thus, we can reduce the number of destinations for this message if the new owner tracks these copies until it sends the update.

We delay the execution time of the supplying and instantiation events of the new local copy with our separated IR. Many other options for the instantiation of new local copies during migration allow the supplying events of some or all of the new local copies to be performed on the old owner copy, eliminating the delay on their execution times. These new local copies are existing copies of the migration. The local copy algorithm for

these copies is more complex since their receive lifetimes can begin after $\mathbf{t_{r_{TO}}}$. Some options also complicate the home copy algorithm. The benefit of these options in relation to the cost of the additional complexity is uncertain.

6.9. Chapter Summary

We presented an owner update protocol that can relocate the owner copy while maintaining isochronicity and sequential consistency. Our highly concurrent migration action solves several problems that do not arise in equidistant topologies, such as changing the execution displacements of every local copy. We modify this action in the next chapter to create the first delta invalidation protocol. Finally, we introduced the concept of the scheduling horizon of an isotach shared memory system.

Chapter 7:
Owner Invalidation Protocol

7.1. Introduction

In this chapter, we present the *owner invalidation protocol*, the first invalidation protocol designed for isotach systems. As with other invalidation protocols, this protocol provides a writer with exclusive access in order to take advantage of reference streams that exhibit processor locality, but unlike other invalidation protocols, it allows writers to execute writes without obtaining ownership. This separation of the acquisition of ownership from the service of writes increases concurrency and allows the protocol to adapt to reference streams that do not exhibit processor locality. The owner invalidation protocol provides this benefit while retaining the advantages of other delta protocols.

7.2. Protocol Overview

The owner invalidation protocol is a variation of the owner update protocol. The protocols use the same coherence actions and scheduling displacements for requests and the same execution displacements for copies. Thus the values from Table 5.1 apply to this protocol as well as to the owner update protocol. Both protocols support non-owner as well as owner copies.

The principal difference between the protocols is in the ownership transition actions, and even these are similar. Recall that the migration action, the ownership transition action of the owner update protocol, destroys each copy while creating a replacement copy for each copy it destroys. In the owner invalidation protocol, the ownership transi-

tion action, called the *invalidation action*, destroys existing copies *without creating new copies* except at the new owner. Thus the new owner starts with an exclusive copy. Since the new owner copy is the only new copy created by the invalidation action, the new owner is the only location that needs a transition record.

In this respect, the invalidation protocol is simpler than the update protocol, but in others it is more complicated, notably in the need for an additional directory at the owner copy. This directory, called the *assign directory*, is required to support split operations. The principal challenge we faced in designing an invalidation protocol for isotach systems was in finding a way to invalidate copies without breaking split operations. In a system that supports split operations, a copy can have outstanding unsubstantiated reads, i.e., reads that were executed while the value of the copy was unsubstantiated. A copy with an unsubstantiated read must continue to receive assigns until the read is substantiated. Thus in an isotach system that supports split operations, copies cannot be destroyed unilaterally. The owner invalidation protocol continues to send assigns to each invalidated copy until the copy explicitly consents to its destruction by sending a release. The owner uses the assign directory to track locations that have not sent a release. It distributes assign updates to both directories, the ordinary directory, which we call the *live copy directory*, as well as the assign directory, but distributes write updates only to the live copy directory. In this chapter, we use the terms *write* and *write updates* to include scheds and sched updates.

Another difference between the protocols concerns the instantiation of new copies in response to miss requests that overlap the ownership transition. As in the update protocol, every miss request that contains an IR results in the issuing SIU receiving a copy. However, the instantiation of new non-owner copies during an ownership transition is delayed to a time later in the ownership transition by the invalidation protocol than by the

update protocol. Delaying the instantiation of new copies gives the new owner a logical time interval during which it is guaranteed to have an exclusive copy. The delay also allows a location that acquires a new copy during an invalidation action that invalidates the location's existing copy to use the same local memory coherence unit for both copies. Without the delay, the receive lifetimes of the copies potentially overlap, necessitating the use of a local memory coherence unit for each copy. We will show in Lemma 7.1 that the receive lifetimes of these copies can not overlap. With the exception of the delay in the instantiation of new non-owner copies during an invalidation action, the instantiation of new copies, both owner and non-owner, is the same in both protocols.

We leave exploration of policies for initiating invalidation actions for future work. and assume here that an invalidation action begins when the owner copy receives a write request. Unless it has an exclusive copy, the owner executes a locally issued write in the same way as any other write, with the result that it obtains a new, now exclusive, owner copy for itself. The owner does not initiate an invalidation action in response to a locally issued write if it already has an exclusive copy.

Neither protocol allows concurrent ownership transition actions. If the owner receives a write request while an invalidation is in progress, it executes the write and sends any updates required, but does not initiate an invalidation action on the writer's behalf. In traditional invalidation protocols, the execution of a write request requires exclusive access and a busy response prevents concurrent accesses to the coherence unit during an invalidation action. The separation of the acquisition of ownership from the service of write requests in our protocol makes it more concurrent than traditional invalidation protocols and allows it to adapt gracefully to accommodate multiple concurrent writers.

An invalidation action begins in the same way as a migration action: the owner multicasts a transition operation (TO) with the same logical receive time $\mathbf{t}_{\mathbf{r}_{\mathbf{TO}}}$ at all destinations. The execution time of the TO is $\mathbf{t}_{\mathbf{r}_{\mathbf{TO}}}$ plus the execution displacement of the copy on which it is executed. The execution time of the ownership transition itself is $\tau_{\mathbf{T}}$. In both the migration and invalidation actions, the lifetime of the new owner copy begins at $\tau_{\mathbf{T}}$ and the lifetime of every existing copy ends by $\tau_{\mathbf{T}}$. In both actions, $\tau_{\mathbf{T}}$ is $\mathbf{t}_{\mathbf{r}_{\mathbf{TO}}} + \chi_{\mathbf{T}}$, where $\chi_{\mathbf{T}} = \max(\mathbf{H}, \max(\mathbf{d}_{\mathbf{TO}}) + \mathbf{d}_{\mathbf{old,\,new}} - \mathbf{d}_{\mathbf{home,\,new}})$.

When a location other than the new owner location receives a TO, it invalidates its existing copy. A location with an invalid copy must use a miss action for any subsequently scheduled request. However, previously scheduled requests must execute on the existing copy. Thus, a copy must continue to execute updates until all outstanding scheduled requests on the copy have been executed. Since $\chi_{\mathbf{T}} \geq \mathbf{H}$ and all scheduled requests are executed within $\mathbf{H}$ pulses, $\tau_{\mathbf{T}}$ is an upper bound on the lifetime of existing copies. The old owner copy is an existing copy with the special responsibility of servicing misses and writes with scheduled execution times up to $\tau_{\mathbf{T}}$. Thus, its lifetime ends exactly at $\tau_{\mathbf{T}}$.

In the invalidation protocol, a copy can go through as many as three phases before it completely disappears: 1) invalidation; 2) death; and 3) release. When a copy is invalidated, no new requests can be scheduled on it; when its lifetime ends, it no longer receives write updates; and when the copy is released, it no longer receives even assign updates. A copy sends its release as soon as possible, so not every copy goes through all three phases.

The initialization of the new owner directory in the invalidation action reflects the protocol’s use of two directories. Recall that the invalidation protocol associates an assign directory and a live copy directory with the owner copy. The initialization of the new owner live copy directory is trivial — since the new owner copy is initially an exclusive

copy, the new owner location is the only location in its initial live copy directory. Initialization of the assign directory at the new owner is the same as the initialization of the new owner directory in the migration action except the directory sent by the old owner in the directory message is the union of its directories. The locations in the old owner live copy directory do not belong in the new owner live copy directory because these locations will no longer have live copies at τ_T when the new owner starts using its directories. However the locations must remain in the current owner's assign directory until they send releases.

7.3. Invalidation Action Details

If a valid owner copy receives a write request, it begins an invalidation action. It multicasts a TO with a uniform logical receive time $\mathbf{t}_{\mathbf{r}_{\mathrm{TO}}}$ to the home copy and the locations in its directories, including the new owner location. If the write request that triggers the invalidation action is not locally issued, then owner copy's state changes to invalid. Otherwise, its state changes to migrating and it uses the local copy algorithm of the migration action. As in the update protocol, an ownership transition migrates the owner copy from the old owner location to the new owner location, and, in this case, these locations happen to be the same. The owner copy also sends an update operation for the write request, as shown in write schemata of Figure 5.8. Except in equidistant networks, the update is sent separately from the TO since the TO must have a uniform logical receive time and the update a uniform logical send time.

The home copy algorithm of the invalidation action is identical to that of the migration action that we presented in Section 6.4.2. Thus, Lemma 6.1 applies and the owner location field of any miss is correct. We present the rest of the invalidation action:

1) the algorithm that destroys the existing copies; 2) the instantiation of new local copies during the invalidation; and 3) the algorithm that initializes the new owner directories.

7.3.1. Destruction of Existing Copies

The local copy algorithm of the invalidation action is significantly simpler than that of the migration action since it does not create replacement copies. This change eliminates the need for an instantiation message. Also, the destruction message is eliminated since the local copy state changes to invalid when the TO is executed.

Upon receiving a TO, a non-owner local copy sends a release when its reservation count equals zero. A reservation count of zero indicates that all previously scheduled requests have completed, including all outstanding unsubstantiated reads. The old owner copy follows the same rule except that it must not send a release before execution time $\tau_{\mathbf{T}}$ since it must service misses and write requests until that time. Thus, the old owner copy sends a release at execution time $\tau_{\mathbf{T}}$ (logical send time $\tau_{\mathbf{T}} - \delta_{\mathbf{existing}}$) if it is not reserved; otherwise it sends the release it when it later becomes unreserved. Finally, the algorithm does not release an invalidated copy if its SIU is acquiring a new local copy, i.e. if the local copy's state has changed from invalid to filling.

The lifetime of each existing copy ends as soon as possible after it is invalidated. We define $\mathbf{t}_{\mathbf{destroy}}$, the execution time of its destruction event, as $\min(\mathbf{t}_{\mathbf{s}_{\mathbf{rel}}} + \delta_{\mathbf{existing}}, \tau_{\mathbf{T}})$, where $\mathbf{t}_{\mathbf{s}_{\mathbf{rel}}}$ is the send time of the invalidation acknowledging release. The lifetime of the old owner copy, which is an invalidated copy, always ends at $\tau_{\mathbf{T}}$ since the old owner copy cannot send a release before $\tau_{\mathbf{T}} - \delta_{\mathbf{existing}}$.

As noted before, a copy can be invalid but still alive. A copy's state always changes to invalid when it executes the TO, but its lifetime does not end until execution

Table 7.1: Local Copy Algorithm (Except at the New Owner Location)

Execution Event	t_r	δ	t_e	Actions

Execute TO	$t_{r_{TO}}$	$\delta_{existing}$	$t_{r_{TO}} + \delta_{existing}$	State = invalid;
				If (not (reserved or old owner copy)) {
				Send release; }
				Else {
				Send COA to self; }

Execute COA	$t_{r_{COA}}$	$\delta_{existing}$	$\tau_T - d_{copy, old} - d_{old, copy}$	If ((state is invalid) and (not reserved) and (old owner copy)) {
				Send release to new owner; }
				If (reserved) {
				Owner = new owner; }

Execute concurrent request, **a**	t_{r_a}	$\delta_{existing}$	τ_a	If ((state is invalid) and (not reserved) and (owner is not self)) { Send release; }

time τ_T if it is still reserved. Also, a copy's lifetime can end before the copy sends a release. If the copy has executed any reads that are still unsubstantiated at τ_T, it must continue to receive assigns. In this case, the destruction event of the copy occurs before the invalidation acknowledging release is sent. Thus, its release can be sent after its receive lifetime ends.

If an issuing SIU has an invalid local copy when it sends an IR, it uses the same local memory coherence unit for the new local copy, which changes the state of the invalid copy to filling. If the invalid copy was reserved, the local copy algorithm never acknowledges the invalidation and the lifetime of the invalidated copy ends at τ_T. Just as with a reserved invalid copy, the filling copy must execute updates since previously scheduled read hits may be executed on it.

Table 7.1 shows the three parts of the local copy algorithm of the invalidation action. These parts are: 1) execution of the TO; 2) execution of a *change of address* (COA) virtual message that the algorithm uses to ensure assigns and releases are sent to the correct location; and 3) execution of concurrent requests. We now describe each part.

The algorithm always changes the copy's state to invalid when it executes the TO. Also, if the copy is neither reserved nor the old owner copy, the algorithm sends a release to the old owner copy when the TO is executed. Otherwise, a COA virtual message is sent.

The COA ensures the old owner directories are destroyed after the old owner copy receives all assigns sent to it. After an issuing SIU receives the COA, it sends any assign to the new owner copy. Before it receives the COA, it sends any assign to the new owner copy if the new owner copy distributed the corresponding sched and to the old owner copy otherwise. Since the execution time of the old owner directory's destruction event is $\tau_{\mathbf{T}}$, the old owner copy must receive all assigns by $\tau_{\mathbf{T}} - \delta_{\mathbf{existing}} = \mathbf{t}_{\mathbf{r}_{\mathbf{TO}}} + \chi_{\mathbf{T}} + \mathbf{d}_{\mathbf{home,\ old}}$. We make $\mathbf{t}_{\mathbf{r}_{\mathbf{TO}}} + \chi_{\mathbf{T}} + \mathbf{d}_{\mathbf{home,\ old}} - \mathbf{d}_{\mathbf{copy,\ old}}$ the logical receive time of the COA, $\mathbf{t}_{\mathbf{r}_{\mathbf{COA}}}$, in order to ensure the old owner copy receives any assigns in time. The COA's logical delay is $\chi_{\mathbf{T}} + \mathbf{d}_{\mathbf{home,\ old}} - \mathbf{d}_{\mathbf{copy,old}}$, which is at least $\mathbf{d}_{\mathbf{home,\ old}} + \mathbf{d}_{\mathbf{old,\ new}} - \mathbf{d}_{\mathbf{home,\ new}}$ since $\chi_{\mathbf{T}} \geq \max(\mathbf{d}_{\mathbf{TO}}) + \mathbf{d}_{\mathbf{old,\ new}} - \mathbf{d}_{\mathbf{home,\ new}}$ and $\max(\mathbf{d}_{\mathbf{TO}}) \geq \mathbf{d}_{\mathbf{copy,old}}$ by definition. Since $\mathbf{d}_{\mathbf{home,\ old}} + \mathbf{d}_{\mathbf{old,\ new}} \geq \mathbf{d}_{\mathbf{home,\ new}}$ by the triangle inequality, the delay is non-negative.

The old owner copy has completed its ownership responsibilities when the local copy algorithm executes its COA since the execution time of its COA is $\tau_{\mathbf{T}}$. Thus, the algorithm releases the old owner copy when it executes the COA if the copy is invalid and unreserved. When any location executes the COA, the algorithm changes the local record of the owner location to point to the new owner location if the local copy is reserved.

As noted, a local copy that is reserved when it receives the TO must execute requests concurrent with the transition. An invalidated copy can receive updates and previously scheduled read hits. After executing any concurrent request that it receives, the copy sends the invalidation acknowledging release if appropriate. We note that the algo-

rithm uses the local record of the owner location to ensure it does not release the old owner copy before the copy's ownership responsibilities are completed.

The invalidation acknowledging release is sent to both the old owner copy and the new owner copy if it is sent before the COA is executed. The rule is the same as the rule used in the owner update protocol for releases concurrent with a transition (see Section 6.7). This similarity in the protocols is not surprising since an invalidation acknowledging release is the same as a transitional release. After execution of the COA, the release is sent only to the new owner copy. As in the owner update protocol, the new owner copy can receive releases before its directories are instantiated. It collects these releases in a *release directory*. We discuss the execution of these releases in Section 7.3.2.

Although the new owner location has an existing copy, it does not execute the local copy algorithm shown in Table 7.1. Unlike the other existing copy locations, the existing copy at the new owner location is replaced and, thus, the algorithm of Table 7.1 does not apply. Instead, the new owner location executes the local copy algorithm of the migration action shown in Table 6.3. We do not discuss that algorithm further.

7.3.2. Instantiation of New Local Copies

The instantiation of new local copies during an invalidation is nearly identical to their instantiation during a migration. The only differences in the invalidation protocol are that the new owner location uses a variation of the separated IR during the ownership transition and it sends the instantiation multicast later. These differences allow a site to store an invalidated copy and a new local copy in the same local memory coherence unit safely. They also ensure a period of logical execution time during which the new owner copy is

Table 7.2: Separated IR Action

Execution Event	t_r	δ	t_e
Supplying event	$\mathbf{t_{r_d}}$	$-d_{home,\,new}$	$\mathbf{t_{r_d}} - d_{home,\,new}$
Instantiation event	$\mathbf{t_{r_d}} + d_{new,\,copy}$	$-d_{home,\,new} - d_{new,\,copy}$	$\mathbf{t_{r_d}} - d_{home,\,new}$

the only copy. As in the owner update protocol, the separated IR can result in an unreserved local copy in the filling state. We assume a filling local copy is never released.

In the update protocol, the new owner location sends the instantiation multicast at $\mathbf{t_{r_i}}$ when it receives its instantiation message. In the invalidation protocol, it sends the instantiation multicast at $\mathbf{t_{r_d}}$ when it receives its destruction message. Logical receive time $\mathbf{t_{r_d}}$ is always later than $\mathbf{t_{r_i}}$ at a new owner copy since a new owner copy is an overlapping copy. In addition to the separated IR's that the old owner copy forwards to it, the new owner location can receive IR's between $\mathbf{t_{r_i}}$ and $\mathbf{t_{r_d}}$ that are included with a miss that the home copy forwards to it. In this case, the new owner copy services the miss immediately, while it buffers the IR until it sends its instantiation multicast, exactly as if the IR was forwarded to it by the old owner copy. Table 7.2 shows the logical times relevant to the separated IR of the invalidation protocol. Observe that $\mathbf{t_{instantiate}} = \mathbf{t_{supply}}$ for new local copies instantiated by a separated IR.

We note that the new owner copy does not send an assign update for the transition **vID** in the owner invalidation protocol. The new owner location substantiates that **vID** when it receives its destruction message. We assume that action precedes the supplying events performed on the new owner copy for the instantiation multicast.

The new owner copy is the only copy whose lifetime includes the period of logical execution time between τ_T and $\mathbf{t_{r_d}} - \mathbf{d_{home,\,new}}$, the execution time of any supplying event of a separated IR. The old owner copy does not instantiate any copies after it begins

Table 7.3: Initialization of New Owner Directories Algorithm

Execution Event	t_r	δ	t_e	Actions
Read old owner directories	$t_{s_{DIR}}$	$\delta_{existing}$ = $-d_{home, old}$	$t_{s_{DIR}} - d_{home, old}$	Message directory = live copy directory ∪ assign directory; Send directory message to new owner copy;
Instantiate new owner directories	$t_{r_{DIR}}$	$\delta_{replace}$ = $-d_{home, new}$	τ_T	Live copy directory = self; Assign directory = message directory - release directory; Discard release directory Send assign multicast to assign directory;

sending the TO and the lifetime of every existing copy ends by τ_T. No supplying events are performed by the new owner copy before it sends the instantiation multicast. Thus, the new owner copy always has the period of exclusive access that Figure 7.1 illustrates.

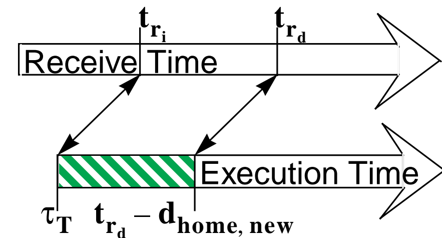


Figure 7.1: Exclusive Period

Note $\tau_T \le t_{r_d} - d_{home, new}$ since $t_{r_d} = \tau_T - \delta_{existing_{new}} = \tau_T + d_{home, old} + d_{old, new}$ and, by the triangle inequality, $d_{home, new} \le d_{home, old} + d_{old, new}$.

Delaying the instantiation multicast to t_{r_d} also ensures that an invalidated copy and a new local copy can safely use the same local memory coherence unit. Any invalidated copy's receive lifetime ends by $\tau_T - \delta_{existing_{copy}} = \tau_T + d_{home, old} + d_{old, copy}$ since its lifetime ends by τ_T. The receive lifetime of any new local copy at that location cannot begin before $\tau_T + d_{home, old} + d_{old, new} + d_{new, copy}$, when the location would receive the instantiation multicast. These receive lifetimes do not overlap, as we now prove:

Lemma 7.1: The receive lifetimes of any invalidated copy and any new local copy at the same location do not overlap.

Proof: Since an invalidated copy's lifetime ends by τ_T, its receive lifetime ends by $\tau_T + d_{home, old} + d_{old, copy}$ Since the instantiation multicast is sent at $\tau_T + d_{home, old} + d_{old, new}$, the copy's location would receive it at $\tau_T + d_{home, old} + d_{old, new} + d_{new, copy}$, which is the earliest that a new local copy's receive lifetime can begin at that

location. Since $\mathbf{d_{old,\,new}} + \mathbf{d_{new,\,copy}} \geq \mathbf{d_{old,\,copy}}$ by the triangle inequality, the lemma follows. **QED**

By Lemma 7.1, the invalidated copy and a new local copy can use the same local memory coherence unit.

7.3.3. New Owner Assign Directory Initialization

Table 7.3 shows the algorithm that initializes the new owner directories. The logical times of the events that initialize the new owner directories in the owner invalidation protocol are identical to those that initialize the new owner directory in the owner update protocol. The actions associated with the events change slightly since the new owner copy has assign and live copy directories and the existing copies are not replaced. Figure 6.7 shows the schemata that applies to the initialization of the new owner directories.

Since the new owner copy delays the instantiation of any new local copies, it is the only copy whose lifetime includes τ_T. Thus, its location is the only location in its initial live copy directory. When the new owner copy sends the instantiation multicast, the destinations are added to its live copy directory, which ensures that it is correct:

Lemma 7.2: Every live copy directory is correct.

Proof: Each live copy directory is initially complete: We assume the live copy directory of the first owner copy is initially complete since its lifetime begins during system initialization. The new owner copy is the only replacement copy. The execution time of the instantiation event of any new local copy is after τ_T. Since the new owner live copy directory includes the new owner copy, it is initially complete.

For any copy instantiated by a separated IR, an $\mathbf{add_{Dir}}$ is performed on the new owner live copy directory when the instantiation multicast is sent. By the logic of Lemma 5.5 that showed the static owner update protocol directory is correct, an $\mathbf{add_{Dir}}$ is correctly performed on the directory for any other new local copy and a $\mathbf{remove_{Dir}}$ is correctly performed on it for any release. **QED**

We use the directory message to initialize the new owner assign directory. The message directory is assigned the union of the old owner assign and live copy directories at $\mathbf{t}_{\mathbf{s}_{\mathbf{DIR}}}$. When the new owner copy receives the directory message, the initial new owner assign directory is assigned the set difference of the message directory and the new owner release directory. The new owner copy can then discard its release directory. After its assign directory is instantiated, the new owner copy sends an update for every assign it receives before $\mathbf{t}_{\mathbf{r}_{\mathbf{DIR}}}$ to the locations in its assign directory. We note that these updates can be combined into a single multicast.

The assign directory ensures all reads are eventually substantiated. As in the owner update protocol, we assume that each assign is sent at the standard level of service after the corresponding sched returns. We now prove that the owner invalidation protocol supports our implementation of split operations:

Lemma 7.3: The owner invalidation protocol eventually substantiates all reads.

Proof: All releases execute correctly. Since a filling copy cannot be released, any release that the new owner copy receives before it sends the instantiation multicast must be for an invalidated copy. These releases can only remove the location from the new owner assign directory and, thus, execute correctly. The new owner copy executes all other releases correctly by the logic of Lemma 5.5 that shows the static owner update protocol executes releases correctly.

Every location receives an assign for the version named by any **vID** on which it could perform an unsubstantiated read. Since the old owner live copy directory is correct by Lemma 7.2, it contains every location that the current ownership transition invalidates. Thus, the initial new owner assign directory includes any unreleased invalidated copies since the directory message contains the union of the old owner assign and live copy directories. Thus, every read is eventually substantiated if every assign is received after the corresponding sched.

Each assign is received after the corresponding sched. If the same owner copy distributes their updates, then any location receives the assign after the sched since the issuing SIU sends the assign after

the sched returns. If different owner copies distribute the updates, every copy receives the assign after the sched by the triangle inequality since assigns use the standard level of service. **QED**

Recall no release is sent for an invalidated copy if its SIU is acquiring a new local copy. The subsequent miss essentially moves the location from the assign directory to the live copy directory. We assume this movement is not actually performed. Instead, we assume releases execute on both directories. This assumption does not create unnecessary message traffic since assign updates are sent to locations in the live copy directory.

7.4. Protocol Correctness

The correctness of the owner invalidation protocol is derived primarily from the correctness of the owner update protocol. For example, the owner location field of every request has the correct location in the owner invalidation protocol by the logic of Lemma 6.7 since the protocols use the same coherence actions and home copy algorithm and the relevant logical times of their ownership transitions are the same.

The protocols use different methods to ensure that no write request is executed on an existing copy incorrectly. In the update protocol, the owner location field determines on which copy a concurrent write is executed. In the invalidation protocol, the new owner location still uses that mechanism, while the separated IR mechanism ensures that writes with scheduled execution times after τ_T are not executed on any invalidated copies.

Lemma 7.4: For any write request executed on an invalidated copy, $\tau < \tau_T$, where τ is the scheduled execution time of the request.

Proof: By Lemma 6.7, each owner copy distributes updates for exactly the writes and scheds with scheduled execution times during its epoch.

An invalid copy only receives updates distributed by the old owner copy, which is the owner copy of its owner epoch.

An invalid copy becomes a filling copy if a local miss is scheduled after the TO is executed. If the miss is a write with a scheduled execution time greater than $\tau_{\mathbf{T}}$, then a distinguished update for that write from the new owner copy can arrive at the invalidated copy during its receive lifetime. However, the write is not executed on the invalidated copy since distinguished updates are never executed. By Lemma 7.1, the invalidated copy's receive lifetime does not overlap with the new copy's receive lifetime and, thus, no other updates from the new owner copy arrive before the end of the invalidated copy's receive lifetime.

Thus, any write or sched executed on an invalidated copy is distributed by the old owner copy. Thus, $\tau < \tau_{\mathbf{T}}$ for any write request executed on an invalidated copy. **QED**

Lemma 7.4 implies that the old owner location is the value of the owner location field of any write executed on an invalidated copy. Further, the old owner location is the value of the owner location field of any read executed on an invalidated copy since miss actions are used after the TO is executed. Since the owner location field of every request has the correct location, any request executed at the new owner location is executed on the appropriate copy. Thus, $\mathbf{t_e}$ always equals τ since the same owner location determines the scheduling displacement of any request and the execution distance of its execution events, as in the proof of Lemma 6.8. We can now show every copy is uniform. As in Lemma 6.10, we use induction on the number of owner epochs.

Lemma 7.5: All copies in the owner invalidation protocol are uniform if every transition aware request is scheduled correctly.

Proof: Basis: The copies of the first owner epoch are uniform by the same logic that applied to the owner update protocol in Lemma 6.10.

Inductive step: We show that every copy of the $\mathbf{n^{th}}$ owner epoch is uniform if the copies of the $\mathbf{(n\text{-}1)^{st}}$ owner epoch are uniform. Each copy of the $\mathbf{n^{th}}$ owner epoch executes every write with a scheduled execution time during its lifetime since the live copy directory of the $\mathbf{n^{th}}$ owner copy is correct by Lemma 7.2.

The scheduled execution time, τ, of any read executed on a copy of the $\mathbf{n^{th}}$ owner epoch is during the copy's lifetime. If it is not exe-

cuted on an invalidated copy, then the logic used in Lemma 6.10 applies. Otherwise, $\tau_{\mathbf{T}} > \tau$ since $\chi_{\mathbf{T}} \geq \mathbf{H}$ and misses are used after the copy is invalidated. The reservation count ensures that the copy is not released before τ. Thus, τ is during the copy's lifetime and any correctly initialized copy of the $\mathbf{n^{th}}$ owner epoch is uniform.

Each copy is initialized correctly: Since $\mathbf{t_e}$ always equals τ, we need to show that the supplying event is performed on a uniform copy such that $\mathbf{t_{instantiate}} = \mathbf{t_{supply}}$ for every copy of the $\mathbf{n^{th}}$ owner epoch. The $\mathbf{n^{th}}$ owner copy is initialized correctly since $\mathbf{t_{instantiate}} = \mathbf{t_{supply}}$ by Lemma 6.9 and its existing copy is uniform by the inductive hypothesis. Any other copy of the $\mathbf{n^{th}}$ owner epoch is uniform since its supplying event is performed on the $\mathbf{n^{th}}$ owner copy such that $\mathbf{t_{instantiate}} = \mathbf{t_{supply}}$, as either Table 7.2 or Table 5.1 shows. **QED**

We can now easily prove that the owner invalidation protocol is correct:

Theorem 7.1: The owner invalidation protocol enforces isochronicity and sequential consistency if the scheduling algorithm is correct and every transition aware request is scheduled correctly.

Proof: If every transition aware request is scheduled correctly, every copy is uniform by Lemma 7.5 and $\mathbf{t_e}$ always equals τ by the logic of Lemma 6.8. Therefore, by Theorem 5.2, the owner invalidation protocol enforces isochronicity and sequential consistency if the scheduling algorithm is correct. **QED**

The owner invalidation protocol exploits processor locality since it provides exclusive access for long write runs. Unlike other invalidation protocols, the service of a request can occur concurrently with an invalidation action in our protocol.

7.5. Optimizations

We can optimize our invalidation action when the old owner copy is also the new owner copy. Since the owner location does not change, the home copy algorithm is unnecessary. Thus, the home copy does not need to receive the TO. The owner copy does not require a transition copy since its execution displacement does not change. The directory action moves the locations in the live copy directory to the assign directory at $\mathbf{t_{s_{DIR}}}$.

Also, we can use $\chi_{\mathbf{T}} = \mathbf{H}$. Recall that $\chi_{\mathbf{T}} \geq \max(\mathbf{d}_{\mathbf{TO}}) + \mathbf{d}_{\mathbf{old,\,new}} - \mathbf{d}_{\mathbf{home,\,new}}$ in order to ensure the directory message is sent after the old owner copy receives all releases sent to it. When the old owner location is the new owner location, the destination of releases does not change and, thus, no releases are lost if we use $\chi_{\mathbf{T}} = \mathbf{H}$.

7.6. Chapter Summary

We have presented the first delta invalidation protocol and proven its correctness. This protocol uses a highly concurrent invalidation action that is an adaptation of the migration action of the owner update protocol. The owner invalidation protocol supports our implementation of split operations. The protocol exploits long write runs and adapts naturally to reference patterns that do not suit invalidation protocols since it uses update messages for writes concurrent with the invalidation action.

Chapter 8:
Local Update Protocol

8.1. Introduction

We present the *local update protocol* in which each local copy is responsible for distributing updates for locally issued writes. Prototype isotach systems that use off-the-shelf components motivated the design of this protocol [Reg97, WiR97]. The initial prototype is an all software implementation of an isotach network, while special purpose hardware will improve performance in later prototypes. Support for extensibility incurs significant performance penalties in these prototypes. The local update protocol supports dynamic replication without requiring an extensible network. We expect this protocol to improve performance substantially over the static replication protocol that is currently used in these systems.

The local update protocol should perform well when used for variables that are both read and written in an interleaved manner by multiple processes [BCZ90]. Most coherence protocols perform poorly for this reference pattern since the processes read the variable frequently but are not likely to read the value of any given write. Thus, invalidations cause many misses, while many updates are unused. The local update protocol does not invalidate copies and updates for each write are distributed by the issuing SIU instead of a centralized copy. This choice for update distribution only requires one network crossing to disseminate each write and should reduce the occurrence of network hot spots. The drawback of this choice is that every copy of the protocol requires a directory.

8.2. Protocol Overview

If a network is not extensible, the logical send times of response messages cannot be controlled. Thus, in the absence of an extensible network, $\mathbf{t_e}$ cannot be ensured to equal τ for any execution event of a request caused by a response message. This restriction makes it difficult, if not impossible, for an intermediate location, such as the owner location, to distribute updates for write requests. The local update protocol can schedule write requests without relying on extensibility since the issuing SIU distributes every write.

The *instantiation action* of the local update protocol, a highly concurrent special coherence action, creates all new local copies. If an issuing SIU does not have a valid local copy or an instantiation action in progress when a request is issued, it immediately begins an instantiation action by sending an IR to the home copy. The IR is never combined with the request. The issuing SIU delays the scheduling decision for any write request until the instantiation action initializes the directory associated with the new local copy. Thus, this protocol replaces the write miss action with an instantiation action followed by a write hit action. Section 8.5 discusses the instantiation action further.

Figure 8.1 shows the other coherence actions of the local update protocol. The issuing SIU of a write hit sends an update to each copy in its directory such that the logical receive time of the update is the same at every destination, while read hits just execute locally. Since each local copy distributes updates, we must associate a directory with it. In the read miss action, the issuing SIU sends the request to the home copy, which executes the request and returns its value to the issuing SIU in a response that uses the bounded level of service. The read miss response never instantiates a new local copy.

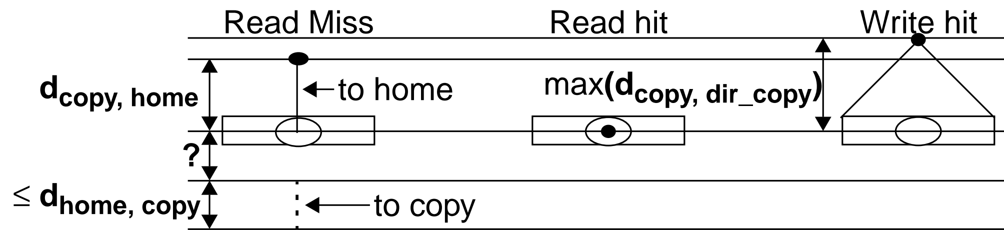


Figure 8.1: Local Update Protocol Coherence Actions

The logical receive time of each of these coherence actions is the scheduled execution time of the request. These actions do not rely on extensibility since their execution events occur after exactly one message. The response message of the read miss action occurs after its only execution event. Thus, the logical send time of its response can exceed the logical receive time of the request at the home copy by an indeterminate period of logical time, as the question mark indicates in the schema.

8.3. Local Copy States

Local copy states can be valid, write-only, filling or invalid. An issuing SIU uses read hits only if its local copy is valid. It uses write hits with a valid or write-only local copy. A new local copy is allocated in the filling state when the SIU issues an IR. When the copy's directory is instantiated (see Section 8.5), the copy's state becomes write-only. When the copy itself is instantiated, its state becomes valid. We show a copy's directory is always instantiated before the copy itself in Section 8.5.2.

The home copy must always be valid since it must execute all read misses. In the instantiation action, the home copy distributes additions to the local copy directories and, thus, it must have a directory. Since every copy in the protocol has a directory, the home copy can be viewed as a distinguished local copy.

Table 8.1: Local Update Protocol Displacements and Distances

Coherence Action	Execution Event	$\sum_m d_m$	δ	Φ	χ
Read Miss	Home copy	$d_{copy, home}$	0	$d_{copy, home}$	$d_{copy, home}$
Read Hit	Issuing copy	0	0	0	0
Write Hit	Local copy	d_{copy, loc_copy}	0	d_{copy, loc_copy}	d_{copy, loc_copy}

8.4. Request Coherence Actions

Table 8.1 shows the scheduling displacements of the local update protocol. The execution displacement of every copy, δ_{copy}, is 0. A write hit has multiple scheduling displacements. The scheduling displacement, χ_{loc_copy}, of an update sent to the local copy, **loc_copy**, is d_{copy, loc_copy}. However, each write request has a single scheduled execution time, τ. The issuing SIU ensures $\tau = t_e$ for its execution events by varying the logical send times of the updates so that the logical receive time is the same at each destination.

Lemma 8.1: The local update protocol ensures t_e always equals τ.

Proof: Table 8.1 shows that $\chi = \Phi$ and, thus, $t_e = \tau$ for the execution event of each read request. For any write request, $\tau = t_r$, where t_r is the logical receive time of every update for the request. Since $\delta_{copy} = 0$ for every copy, $t_e = t_r = \tau$ for any execution event of the request. **QED**

Since t_e always equals τ, the local update protocol is correct if all copies are uniform and the scheduling algorithm is correct by Theorem 5.2.

8.5. The Instantiation Action

Our instantiation action is highly concurrent. The issuing SIU can schedule and send read misses while it acquires the new local copy, and it can schedule and send write hits when the associated directory is instantiated, which occurs before the instantiation

action completes. The action uses several response messages, but it does not require an extensible network since its correctness does not require the anticipation of the logical receive time of any of the responses.

8.5.1. Overview of the Instantiation Action

The instantiation action must initialize the new local copy and its associated directory correctly. It must also ensure that the location of the new local copy is added to every existing directory before its lifetime begins. Since $\mathbf{t_e}$ always equals τ by Lemma 8.1, the action initializes the new local copy correctly if it performs the supplying event for the copy on a uniform copy such that $\mathbf{t_{instantiate}} = \mathbf{t_{supply}}$.

The instantiation action correctly initializes the directory of the new local copy. When the home copy receives an IR, it adds the location to its directory and sends the contents of its directory to the issuing SIU. The logical receive time of this directory message is the execution time of the new directory's instantiation event, $\mathbf{t_{instantiate_{Dir}}}$. For the new local copy (or any local copy that is created for an IR the home copy receives after it sends the directory message), the instantiation action ensures $\mathbf{t_{instantiate}} > \mathbf{t_{instantiate_{Dir}}}$, where $\mathbf{t_{instantiate}}$ is the execution time of the new local copy's instantiation event. Since a new local copy's location is added to the home copy directory before that copy's directory message is sent, the new directory includes any unreleased copy created by a previous IR and, thus, is complete initially.

We ensure the location of the new local copy is added to every directory before $\mathbf{t_{instantiate}}$. The home copy sends a message to every location in its directory when it receives the IR. When a copy receives this message, it adds the new local copy's location to its directory and sends an acknowledgment to the new local copy. The new local copy's

instantiation event occurs after it receives every acknowledgment. Thus, its location is added to every directory before its lifetime begins.

The existing copies must add the new local copy's location to their directories to guarantee that the new local copy receives every write request during its lifetime. Each copy sends its acknowledgment in such a way that $\mathbf{t}_{\mathbf{r}_{\mathrm{ACK}}}$, the logical receive time of its acknowledgment is greater than $\mathbf{max}(\tau_{\mathbf{w}})$, the maximum previously scheduled execution time of any locally issued write request to the coherence unit. An update for any subsequently scheduled, locally issued write request is sent to the new local copy.

In order to initialize the new local copy correctly, each existing copy sends an acknowledgment to the home copy as well as to the new local copy's location, such that the logical receive time of the acknowledgments is the same at both destinations. The instantiation event of the new local copy occurs when the new local copy's location receives the last acknowledgment and its supplying event occurs when the home copy receives the last acknowledgment. Since the supplying event is performed on the home copy and the logical receive time of the last acknowledgment is the same at both destinations, $\mathbf{t}_{\mathbf{instantiate}} = \mathbf{t}_{\mathbf{supply}}$ and the copy is initialized correctly if the home copy is uniform.

The home copy sends its values to the new local copy at the bounded level of service when the supplying event occurs. Since $\mathbf{t}_{\mathbf{instantiate}} = \mathbf{t}_{\mathbf{supply}}$ and the execution displacements of both copies are zero, these values must arrive after the instantiation event of the new local copy occurs. To accommodate reads occurring between $\mathbf{t}_{\mathbf{instantiate}}$ and when the values arrive, we initialize the new local copy to an unsubstantiated special transition **vID**. The values sent by the home copy are assigned to this **vID**.

We assume that the home copy is initialized correctly and its directory is complete initially since their lifetimes begin during system initialization. The home copy is always

Table 8.2: Instantiation Action Copy Algorithms

#	Copy	Logical Receive Time	Actions
1	Home Copy	$t_{r_{IR}} \leq t_{s_{IR}} + d_{issuing, home}$	Execute IR: **AcksDue**[**issuing**] = |directory|; Send AA to directory; Add issuing copy location to directory; Send directory message to issuing copy;
2	Issuing Copy	$t_{r_{DIR}} = t_{s_{AA}} + d_{home, issuing}$	Execute directory message: Directory = message directory; State = write-only; **AcksDue** = |directory| - 1;
3	Local Copy	$t_{r_{AA}} = t_{s_{AA}} + d_{home, copy}$	Execute AA: Add issuing copy location to directory; Send ACK to home and issuing copies;
4	Issuing Copy	$t_{r_{ACK}} = t_{s_{ACK, issuing}} + d_{copy, issuing}$	Execute acknowledgment: Decrement **AcksDue**; If (**AcksDue** is zero) { State = valid; }
5	Home Copy	$t_{r_{ACK}} = t_{s_{ACK, home}} + d_{copy, home}$	Execute acknowledgment: Decrement **AcksDue**[**issuing**]; If (**AcksDue**[**issuing**] is zero) { Send issuing copy transition **vID** assign; }

in every directory and, thus, receives an update for every write since it is never released and its directory initializes all other directories. Thus, it is uniform.

8.5.2. Details

We present details of the instantiation action. Table 8.2 shows its steps essentially in the order they occur. Figure 8.2 shows its schemata. In Table 8.2, the *issuing copy* is the new local copy. The actions shown in rows 2 and 3 are concurrent. Similarly, the actions shown in rows 4 and 5 are concurrent.

When a request is issued and no local copy exists or its state is invalid, the issuing SIU sends an IR message to the home copy at the bounded level of service, which we assume provides point-to-point FIFO delivery. Also, a local memory coherence unit for the new local copy is allocated if necessary. Its state is set to filling and the transition **vID** is associated with its unsubstantiated initial version.

The home copy performs several actions when it receives the IR. It sets the outstanding acknowledgment count for the issuing copy, **AcksDue**[**issuing**], to the current size of its directory. The home copy uses an array for this purpose in order to support concurrent instantiation actions from different locations to the same coherence unit. The home copy also sends an *add address* (AA) coherence operation to each location in its directory, adds the issuing copy's location to its directory and sends the directory message to the issuing SIU. The home copy receives an AA since it is always in its directory. The AA's and the directory message use the standard level of service and all have the same logical send time, $\mathbf{t}_{s_{AA}}$, which is an indeterminate amount of logical time after the logical receive time of the IR, $\mathbf{t}_{r_{IR}}$. The home copy includes its **AcksDue** array index of the issuing copy in the AA. A local copy includes this index in the acknowledgments that it sends in response to the AA as an identifier of the AA.

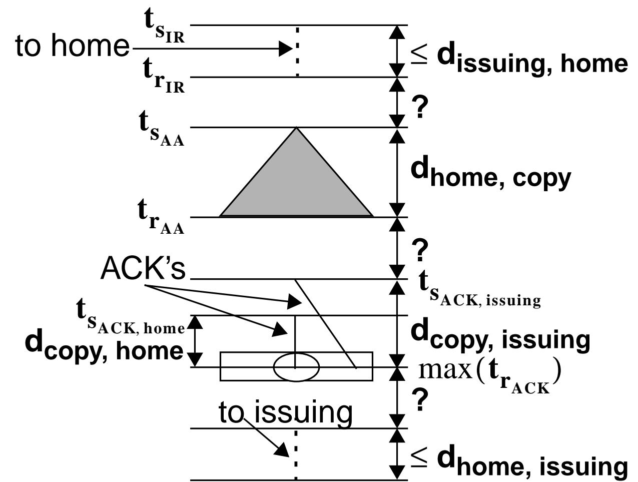


Figure 8.2: Instantiation Action

Execution of the directory message instantiates the directory for the issuing copy and changes that copy's state to write-only, which allows the issuing SIU to schedule write hits. It also sets the issuing copy's outstanding acknowledgment count, **AcksDue**, to one less than the size of the directory since the issuing copy's location is in that directory.

We can now prove that the home copy receives every write request.

Lemma 8.2: The home copy is uniform.

Proof: We assume the initial values of the home copy are correct since its lifetime begins during system initialization. Thus, we must show that it executes every write at its scheduled execution time.

> The home copy is never released and is always in its own directory. Since the directory of the home copy initializes the directory of any local copy, the home copy is always in the directory of any local copy. Thus, the home copy receives and executes an update for every write. Since it is never released, the scheduled execution time of any request is during its lifetime. **QED**

A local copy adds the issuing copy location to its directory when it receives an AA. It sends an acknowledgment to the home copy and the issuing copy with the same logical receive time, $\mathbf{t}_{\mathbf{r}_{\mathbf{ACK}}}$. Acknowledgments from different local copies can have different logical receive times, but every pair of acknowledgments from the same source has the same logical receive time. The pair of acknowledgments with the greatest logical receive time is shown in the schemata. Since the logical send time of the acknowledgment to the issuing copy, $\mathbf{t}_{\mathbf{s}_{\mathbf{ACK,\,issuing}}}$, precedes the logical send time of the acknowledgment to the home copy, $\mathbf{t}_{\mathbf{s}_{\mathbf{ACK,\,home}}}$, the schemata depicts the case where the distance from their sender is greater to the issuing copy than to the home copy. The opposite case is also possible.

Each local copy guarantees that the issuing copy receives any update it sends with a logical receive time greater than its $\mathbf{t}_{\mathbf{r}_{\mathbf{ACK}}}$. We can easily implement this restriction. If the reservation count of the sender is zero, then it has no outstanding write requests and the restriction is met automatically since the acknowledgments use the standard level of service. Otherwise, the sender can ensure the pulse component of $\mathbf{t}_{\mathbf{r}_{\mathbf{ACK}}}$ is at least the scheduled execution pulse of its most recently scheduled isochron. Thus, $\mathbf{t}_{\mathbf{r}_{\mathbf{ACK}}}$ is greater than the receive time of the update for any previously scheduled, locally issued write request to the coherence unit. In addition, the local copy must send an update for any subsequently scheduled, locally issued write request to the new local copy. If the reading of the directory to determine the update destinations and the scheduling of the requests is not

atomic, then the local copy must add the issuing copy location to the update destinations of any unscheduled write for which the directory has already been read.

Any local copy that receives an AA executes this algorithm regardless of its state. If the local copy no longer exists or is invalid, then the associated SIU has released it. If the local copy state is filling then the SIU initiated an instantiation action after releasing a previously held local copy. In these cases, the issuing copy location does not need to be added to the released copy's directory. However, the SIU must send its acknowledgments.

The home copy also sends its acknowledgments. The acknowledgment that it sends to the issuing copy provides the guarantee that the issuing copy receives its updates. The acknowledgment that it sends to itself causes the supplying event of the issuing copy when no other local copies exist.

The issuing copy decrements **AcksDue** when it executes an acknowledgment. When **AcksDue** becomes zero, the issuing copy has received an acknowledgment from every location in its initial directory and it executes its instantiation event by changing state to valid. After its state becomes valid, the issuing copy must execute any updates that it receives. We assume the network supports the triangle inequality. Therefore, the issuing copy receives the directory message before any acknowledgments. We discuss relaxing this assumption and other optimizations to the instantiation action in Section 8.8.

The home copy decrements **AcksDue**[**issuing**] when it receives an acknowledgment from any local copy for the issuing copy. If **AcksDue**[**issuing**] is zero, then the home copy sends an assign of its current values to the issuing copy for the transition **vID** since it has received the last acknowledgment. We now show that the issuing copy is uniform if releases execute correctly.

Lemma 8.3: Each local copy is uniform if releases execute correctly.

Proof: The instantiation event of any local copy other than the home copy occurs when its location receives the last acknowledgment for its AA and its supplying event occurs when the home copy receives the last acknowledgment. Since each acknowledgment pair has the same logical receive time at both destinations and all execution displacements are zero, $\mathbf{t_{instantiate}} = \mathbf{t_{supply}}$. The issuing copy is initialized correctly since the home copy is uniform by Lemma 8.2 and $\mathbf{t_e}$ always equals τ by Lemma 8.1.

Any read execution event performed on a local copy other than the home copy is for a read hit and must be scheduled after the copy's instantiation event. The reservation count ensures the read executes before the copy's destruction event. Since $\mathbf{t_e}$ always equals τ, the read's scheduled execution time is during the copy's lifetime.

We now show that the issuing copy of any IR executes any write request, **w**, with a scheduled execution time, τ, during its lifetime. Since releases execute correctly, the issuing copy receives an update for **w** if its location is correctly added to each directory.

Suppose a copy in the initial directory of the issuing copy sends the updates for **w**. Since $\mathbf{t_e}$ always equals τ and the scheduled execution time of any write is the logical receive time of its updates, τ is the logical receive time of the updates for **w**. Since τ is during the issuing copy's lifetime, τ must be greater than $\mathbf{t_{r_{ACK}}}$, the logical receive time of the acknowledgment from the copy that sends the updates. Since each copy guarantees that the issuing copy receives any updates that it sends with logical receive times greater than $\mathbf{t_{r_{ACK}}}$, the issuing copy receives an update for **w**. Thus, it executes **w**.

Suppose a copy not in the initial directory of the issuing copy sends the updates for **w**. Since this local copy is not in the initial directory of the issuing copy, the home copy must have sent a directory message to it after sending the directory message to the issuing copy. Thus, its initial directory includes the location of the issuing copy and, thus, the issuing copy executes **w**. **QED**

8.6. Releases

We now present the release action of the local update protocol. This action ensures that all releases execute correctly since every location receives instantiation and release

actions to a coherence unit in the same order. Both actions use a message to the home copy that causes the home copy to send a mutlicast to its directory. Every destination receives the actions in the same order since these messages use the standard level of service.

We assume a copy must be instantiated before it is released. Thus, a local copy in the filling or write-only state cannot be a victim copy. This assumption ensures that each acknowledgment and directory message is applied to the correct new local copy. We can relax this assumption if we associate a generation number with each local copy.

A release must inform each directory of the coherence unit about the victim copy. Figure 8.3 shows the schemata of the release action. When an SIU selects a victim copy, it sends a release initiating message to the home copy of the coherence unit at the bounded level of service. The home copy removes the location from its directory and sends a release at the standard level of service for the victim copy to every location that remains in its directory. These releases have a single logical send time and, thus, potentially different receive times in a logically non-equidistant network. The release coherence action does not require an extensible network. The home copy can send the release multicast an indeterminate amount of logical time after it receives the initiating release. A valid or write-only local copy removes the location of the victim copy from its directory when it receives the release. Invalid and filling local copies, which do not have a directory, discard any releases that they receive. We discuss optimizations of the release action in Section 8.6.

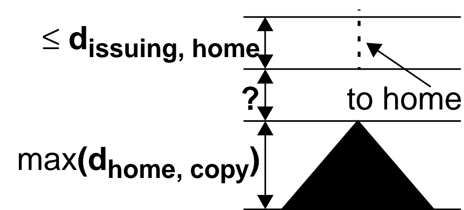


Figure 8.3: Release Action

We now show that any $\mathbf{remove_{Dir}}$ performed for a release executes such that $\mathbf{t_{destroy}} \le \mathbf{t_{remove_{Dir}}}$ and $\mathbf{t_{remove_{Dir}}} < \mathbf{t_{add_{Dir}}}$ for any $\mathbf{add_{Dir}}$ for a subsequent IR from the same SIU. Thus, the release executes correctly, as discussed in Section 5.7.4.

Lemma 8.4: All releases execute correctly.

Proof: The execution time, $\mathbf{t_{destroy}}$, of the destruction event of a copy equals the logical send time of the release initiating message that the copy-owning SIU sends to the home copy. Since that message uses the bounded level of service, $\mathbf{t_{destroy}} \le \mathbf{t_r}$, where $\mathbf{t_r}$ is its logical receive time. A $\mathbf{remove_{Dir}}$ execution event is performed on the associated directory of any write-only or valid local copy that receives the release. Since $\delta_{\mathbf{Dir}} = 0$ for every directory, $\mathbf{t_{remove_{Dir}}} = \mathbf{t_r} + \mathbf{d_{home, copy}}$ and, thus, $\mathbf{t_{destroy}} \le \mathbf{t_{remove_{Dir}}}$.

The execution time, $\mathbf{t_{add_{Dir}}}$, of any $\mathbf{add_{Dir}}$ performed for any subsequent IR issued by the associated SIU is at least $\mathbf{t_{r_{IR}}} + \mathbf{d_{home, copy}}$, where $\mathbf{t_{r_{IR}}}$ is the logical receive time at the home copy of the IR. Since the bounded level of service provides point-to-point FIFO delivery, $\mathbf{t_{r_{IR}}} > \mathbf{t_r}$ and, thus, $\mathbf{t_{remove_{Dir}}} < \mathbf{t_{add_{Dir}}}$. **QED**

Now, we show that the local update protocol is correct:

Theorem 8.1: The local update protocol enforces isochronicity and sequential consistency if the scheduling algorithm implements **IRule** and **SCRule**.

Proof: By Lemma 8.1, $\mathbf{t_e}$ always equals τ. By Lemma 8.4, releases execute correctly. Thus, every copy is uniform by Lemma 8.3 and the protocol is correct by Theorem 5.2. **QED**

8.7. Split Operations

The local update protocol is compatible with our implementation of split operations. The reservation count mechanism ensures each local copy receives an assign for any **vID** on which it performs an unsubstantiated read. Since the logical receive time of a sched update is the same for every copy and the issuing SIU sends the corresponding assign after it receives the sched update, assigns can use the bounded level of service.

The supplying event of the new local copy associates an unsubstantiated value for the transition **vID** if the home copy is unsubstantiated when the supplying event occurs. If the home copy is unsubstantiated, then it has not received the corresponding assign for a

sched request already executed on it. Since the home copy receives the corresponding assign after the last acknowledgment, the new local copy also receives the corresponding assign. However, the new local copy can receive the corresponding assign before the assign that the home copy sends for the transition **vID**. Therefore, each new local copy buffers assigns until it receives the assign for the transition **vID**. It then executes the buffered assigns after it executes the assign for the transition **vID**.

8.8. Optimizations

We discuss some possible optimizations to the instantiation and release actions of the local update protocol. A simple optimization can be used if the request that causes the instantiation action is a read. In our description of the instantiation action, the IR is sent separately from the read miss. These messages can be combined. The benefit of this choice is uncertain. Although it reduces network traffic, it delays the send time of the IR until the read request is scheduled and the single message must use the standard level of service. Thus, the execution events of the instantiation action are delayed, which can result in more read misses and a longer scheduling delay for write requests.

The AA and release multicasts that the home copy sends use the standard level of service. These messages can use the bounded level of service. The destinations still receive the actions in the same order since their messages travel the same paths and point-to-point FIFO delivery is guaranteed for the bounded level of service. We must alter the instantiation action to count acknowledgments that arrive at the new local copy before the directory message if AA messages use the bounded level of service. With this change, we do not require the network to support the triangle inequality.

We could use a different release action in which the SIU of the victim copy sends the releases directly to the locations in its directory. However, we expect our optimized release and instantiation actions to outperform this solution since our actions ensure a consistent order of their execution inexpensively by using the same message paths. If the SIU sends releases directly, we must ensure each destination executes release and instantiation actions from the same location in their issue order. Ensuring this order is difficult with the bounded level of service since the actions would use different messages paths.

We can alter the local update protocol to exploit extensibility. The home copy sends the AA and directory messages so that they all have the same logical receive time in the altered protocol. Since the issuing and home copies can use the scheduling horizon to bound the scheduled execution time of any write for which the issuing copy does not receive an update, the altered protocol can eliminate the acknowledgments.

8.9. Chapter Summary

We presented a local update protocol in which every local copy has responsibility for distributing updates for locally issued write requests. The protocol suits reference patterns with irregular and unpredictable accesses such as exhibited by frequently read and written variables. Unlike other existing delta protocols, the local update protocol does not rely on extensibility. Instead, the protocol uses a highly concurrent coherence action that is separate from the service of requests to create new local copies. We expect the protocol to improve performance in prototype isotach systems composed primarily of off-the-shelf components. Finally, we proved the correctness of the local update protocol.

Chapter 9:
Conclusion

9.1. Introduction

In this thesis, we presented an extended, more unified theory of isotach systems than that first given by Williams [Wil92]. This extended theory enlarges the class of correct implementations of isotach networks, increases the applicability of isotach systems to networks with non-uniform link latencies and creates a unifying framework for isotach shared memory systems that better supports the design of delta coherence protocols. We presented new delta coherence protocols that extend isotach-based coherence protocols to a wider range of topologies and reference patterns and to non-extensible isotach networks.

9.2. Contributions

We have advanced the theory and understanding of isotach systems as follows:

- Identified a new relation over the events of distributed systems that captures causality more faithfully than Lamport's *happens before* relation;
- Designed an isotach network algorithm that gives greater flexibility in the assignment of logical distances between communicating nodes;
- Formulated a unifying framework for isotach shared memory systems;
- Extended delta coherence protocols to non-equidistant networks;
- Extended delta coherence protocols to include protocols that target data access patterns not targeted by the original set of protocols;
- Designed a delta protocol for non-extensible isotach systems, an alternative class of isotach systems that are easier to build than standard isotach systems.

The extended theory of isotach systems presented in this thesis contributes to the solution of the coherence maintenance problem. Delta coherence protocols, the isotach-based family of coherence protocols, support the execution of structured atomic actions

without the use of locks. They allow more pipelining than traditional coherence protocols. They also allow increased concurrency for requests by different processes in several ways compared to traditional protocols. For example, they allow multiple concurrent readers and writers and can separate the service of requests from the acquisition of a new copy or exclusive access. Simulation studies show that this additional concurrency can improve performance significantly.

After we observed that proposed prototype isotach systems appeared to be consistent with causality but were not consistent with Lamport's *happens before* relation, we defined a new relation, *potential causality*, applicable to isotach systems and other systems that use a messaging process between the application process and the network. Instead of the *happens before* relation, we now require that isotach systems be consistent with this new relation. Although the relations are similar, *potential causality* allows causally consistent implementations not allowed by Lamport's relation. Thus, it increases the flexibility of isotach systems and allows more choices in implementing isotach systems.

We designed the flex algorithm, the first isotach network algorithm that allows the logical distances in an isotach network to reflect the raw message latency of the individual links. Since end-to-end message latency in isotach networks is proportional to the logical distance that the message travels, this algorithm should improve performance in networks with non-uniform link latencies. We showed that the flex algorithm correctly implements an isotach Logical time system while providing greater flexibility for logical distance assignments than a simpler generalization of the previously identified isotach network algorithm. Also, we presented a Petri net model of the algorithm that allows us to determine if a set of logical distance assignments will cause the algorithm to deadlock and indicates a potential source of additional isotach network algorithms.

We developed a new framework for isotach shared memory systems that provides a unifying theory for these systems. Our framework specifically allows optimizations not addressed by previous research. Also, it allows an issuing SIU to anticipate logical execution times although the corresponding logical receive times may not be known, thus supporting the design of delta coherence protocols for non-equidistant networks. Unlike the existing framework, ours does not use a physical canonical copy, allowing us to demonstrate that a correct delta protocol represents an infinite class of correct protocols.

Our owner update protocol extends Williams's early protocol [Wil93] to non-equidistant networks and unified it with her late protocol, the only other previously existing delta protocol. Our owner update protocol includes a highly concurrent migration mechanism that does not suspend access to the copies and allows any node that has a copy to retain a copy throughout the migration. Relocating the owner copy dynamically was a much more difficult problem than in equidistant networks since a migration generally changes the logical distance from the owner copy location to any other location.

Our owner invalidation protocol, the first delta invalidation protocol, modifies the migration mechanism of the owner update protocol to invalidate the existing copies. This protocol exploits long write runs since a node that repeatedly writes a coherence unit is guaranteed exclusive access for a period of logical execution time by our invalidation mechanism. Unlike traditional invalidation protocols, our invalidation protocol naturally adapts to reference patterns that do not exhibit long write runs and allows the initiating write to complete prior to providing the writer with exclusive access.

We can now apply isotach-based coherence techniques to a much wider range of implementations since our local update protocol supports dynamic replication without requiring an extensible isotach network. This protocol is an important addition to the fam-

ily of delta protocols since extensible isotach systems are more complicated and may have higher message latency than isotach systems that do not support extensibility. The local update protocol has several unusual features that indicate possible directions for further research, including the replication of coherence directories and the separation of the creation of copies from the service of requests.

9.3. Future Work

This thesis has revealed several promising topics for future research. This section discusses some of these topics in detail and outlines others.

An important topic for future work is the design of an isotach compiler and a body of isotach programs. Isotach shared memory systems offer consistently good performance for programs that use isochronous techniques to enforce structured atomicity requirements [dWR96]. Development of these programs will allow us to simulate our range of coherence protocols using real workloads.

Weak consistency semantics for isotach shared memory systems is another major topic for future research. We expect that isotach techniques for enforcing traditional weak consistency semantics, such as release consistency, will provide comparable performance for programs for which traditional techniques perform well. Highly concurrent isotach-based lock implementations may allow isotach techniques to improve the performance of programs that exhibit significant lock contention. Also, we anticipate that a new class of consistency semantics based on isotach as opposed to lock based coordination of accesses will emerge from a study of how to write programs that use isochronous techniques.

The non-blocking implementations of isotach network algorithms that we discussed in Chapter 4 support a rich topic of future research. Under these implementations,

an SIU can receive messages out of order. In this thesis, we assumed the SIU delivers messages in order, which requires that messages are buffered while messages with earlier logical receive times may arrive. Alternatively, *optimistic isotach systems* could deliver messages as they arrive. Correct execution would require the system to support rollback of messages if the SIU subsequently received a message with an earlier logical receive time.

Many techniques developed for optimistic parallel simulation systems, such as Time Warp [Jef85], would be applicable to optimistic isotach systems. We expect optimistic isotach systems to be more efficient than other optimistic shared memory systems. Isotach network algorithms would disseminate the equivalent of global virtual time efficiently in these systems, while the isotach invariant implies a limited logical time interval for which rollback state must be maintained.

Optimistic isotach systems would reduce the need to use weak consistency semantics. Weak consistency semantics exploit opportunities to proceed with the execution of requests that program structure ensures will not cause any violations of sequential consistency. Optimistic isotach systems would automatically exploit these opportunities, as well as other opportunities that program structure cannot reveal. Rollback in optimistic isotach systems would recover from any violations of sequential consistency. As we have mentioned, we expect the isotach invariant to limit the state space requirements, an important aspect of the cost of rollback. In addition, rollback is only required for write requests, which usually occur infrequently relative to read requests. This last fact implies lazy cancellation techniques will be very useful for optimistic isotach systems.

Performance evaluation is another major topic for future research to emerge from this thesis. A masters project currently underway is evaluating the use of the non-blocking

flex algorithm implementation. In addition, large amounts of additional evaluation of delta coherence protocols remains, particularly using real programs and machines.

We briefly outline several other topics of future research based on this thesis:

- Applications of *potential causality* in non-isotach logical time systems;
- Additional isotach network algorithms;
- Development of the theory of the geometry implied by isotach logical time;
- Design of other members of the delta protocol family, including:
  - Limited directory protocols;
  - Adaptive protocols;
  - Competitive protocols;
  - Invalidation protocols that do not require an extensible network;
  - Owner protocols that selectively replace or invalidate each existing copy during any ownership transition;
  - Protocols that separate directory replication from data replication;
  - Protocols that can negatively acknowledge an IR and, thus, not grant a copy;
  - Hybrid software/hardware protocols or compiler support for delta protocols;
- Dynamic page management techniques for isotach systems;
- Migration and invalidation policies for the owner protocols;
- Allowing concurrent ownership transitions to the same coherence unit;
- Dynamic use of split operations, i.e. supporting the ability to choose dynamically whether to execute a given assignment as a write or a sched/assign pair.

In addition, we want to explore minor changes to our protocols that we expect to improve performance under some workloads. For example, in the owner invalidation protocol, the new owner copy could delay the instantiation multicast if it has any scheduled write requests, ensuring that the requests occur while it holds exclusive access.

9.4. Concluding Remarks

We extended the theory of isotach systems in several ways that add qualitatively to known solutions of the coherence maintenance problem. This theory increases their flexibility and allows them to serve a wider range of networks and applications. Isotach systems are an exciting technology for which there is a substantial and promising body of future research.

Chapter 10:
References

[AbK85] Abramson, D.A., and J.L. Keedy, “Implementing a Large Virtual Memory in a Distributed Computing System,” *Proceedings of the 18th Annual Hawaii International Conference on System Sciences*, pp. 515-522, 1985.

[AdH90] Adve, S.V., and M.D. Hill, “Implementing Sequential Consistency in Cache-Based Systems,” *Proceedings of the 1990 International Conference on Parallel Processing*, Vol. I, pp. 47-50, 1990.

[Adv93] Adve, S.V., Designing Memory Consistency Models for Shared Memory Multiprocessors, Ph.D. Thesis, University of Wisconsin-Madison, 1993.

[AdG96] Adve, S.V., and K. Gharachorloo, “Shared Memory Consistency Models: A Tutorial,” *IEEE Computer*, Vol. 29, No. 12, pp 66 - 76, 1996.

[ABM93] Afek, Y., G. Brown and M. Merritt, “Lazy Caching,” *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 1, pp. 182-205, 1993.

[AgG88] Agarwal, A. and A. Gupta, “Memory Reference Characteristics of Multiprocessor Applications under MACH,” *Proceedings of the 1988 Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 215-225, 1988.

[ASH88] Agarwal, A., R. Simoni, J. Hennessy and M. Horowitz, “An Evaluation of Directory Schemes for Cache Coherence,” *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 280-289, 1988.

[AHJ91] Ahamad, M., P.W. Hutto and R. John, “Implementing and Programming Causal Distributed Shared Memory,” *Proceedings of the 11th International Conference on Distributed Computing Systems*, pp. 274-281, 1991.

[ABK94] Ahamad, M., G. Neiger, P. Kohli, J.E. Burns and P.W. Hutto, “Causal Memory: Definition, Implementation and Programming,” Tech. Rep. GIT-CC-93/55, Georgia Institute of Technology, College of Computing, 1994.

[ACD96] Amza, C., A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel, “Treadmarks: Shared Memory Computing on Networks of Workstations,” *IEEE Computer*, Vol. 29, No. 12, pp 18 - 28, 1996.

[ArB84] Archibald, J., and J.L. Baer, "An Economical Solution to the Cache Coherence Problem," *Proceedings of the 11th International Symposium on Computer Architecture*, pp. 355-362, 1984.

[AtF92] Attiya, H., and R. Friedman, "A Correctness Condition for High-Performance Multiprocessors," *Proceedings of the 24th ACM Symposium on the Theory of Computing*, pp. 679-690, 1992.

[AtW95] Attiya, H., and J.L. Welch, "Sequential Consistency versus Linearizability," *ACM Transactions on Computer Systems*, Vol. 12, No. 2, pp. 91-122, 1995.

[Awe85] Awerbuch, B., "Complexity of Network Synchronization," *Journal of the ACM*, Vol. 32, No. 4, pp. 804-823, 1985.

[ABF93] Awerbuch, B., Y. Bartal and A. Fiat, "Competitive Distributed File Allocation," *Proceedings of the 25th ACM Symposium on the Theory of Computing*, pp. 164-173, 1993.

[BFR92] Bartal, Y., A. Fiat and Y. Rabani, "Competitive Algorithms for Distributed Data Management (Extended Abstract)," *Proceedings of the 24th ACM Symposium on the Theory of Computing*, pp. 39-50, 1992.

[BFR95] Bartal, Y., A. Fiat and Y. Rabani, "Competitive Algorithms for Distributed Data Management," *Journal of Computer and System Sciences*, Vol. 51, pp. 341-358, 1995.

[BMR89] Baylor, S.J., K.P. McAuliffe and B.D. Rathi, "Cache Coherence Protocols for MIN-Based Multiprocessors," RC15221, IBM Research Report, 1989.

[BCZ90] Bennett, J.K., J.B. Carter and W. Zwaenepoel, "Adaptive Software Cache Management for Distributed Shared Memory Architectures," *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 125-134, 1990.

[BCZ90a] Bennett, J.K., J.B. Carter and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proceedings of the ACM Symposium on Principles and Practices of Parallel Programming*, pp. 168-176, 1990.

[BeZ91] Bershad, B.N., and M.J. Zekauskas, "Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors," Tech. Rep. CMU-CS-91-170, Carnegie Mellon University, School of Computer Science, 1991.

[BGS89] Birk, Y., P.B. Gibbons, J.L.C. Sanz and D. Soroker, "A Simple Mechanism for Efficient Barrier Synchronization in MIMD Machines," RJ7078, IBM Research Report, 1989.

[BSS91] Birman, K., A. Schiper and P. Stephenson, “Lightweight Causal and Atomic Group Multicast,” *ACM Transactions on Computer Systems*, Vol. 9, No. 3, pp. 272-314, 1991.

[BlS89] Black, D.L., and D.D. Sleator, “Competitive Algorithms for Replication and Migration Problems,” Tech. Rep. CMU-CS-89-201, Carnegie Mellon University, Department of Computer Science, 1989.

[BLV94] Bianchini, R., T.J. LeBlanc and J. Veenstra, “Eliminating Useless Messages in Write-Update Protocols on Scalable Multiprocessors,” Tech. Rep. 539, University of Rochester, Computer Science Department, 1994.

[BKP96] Bianchini, R., L.I. Kontothanassis, R. Pinto, M. De Maria, M. Abud and C.L. Amorim, “Hiding Communication Latency and Coherence Overhead in Software DSMs,” *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 198-209, 1996.

[BiD86] Bitar, P., and A.M.Despain, “Multiprocessor Cache Synchronization: Issues, Innovations, Evolution,” *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 424-433, 1986.

[BGW89] Black, D.L., A. Gupta and W.D. Weber, “Competitive Management of Distributed Shared Memory,” *Proceedings of the Spring Compcon 1989*, pp. 184-190, 1989.

[BoS92] Bolosky, W.J., and M.L. Scott, “A Trace-Based Comparison of Shared Memory Multiprocessor Architectures,” Tech. Rep. 432, University of Rochester, Computer Science Department, 1992.

[BoS93] Bolosky, W.J., and M.L. Scott, “False Sharing and its Effect on Shared Memory Performance,” *Proceedings of the 5th USENIX Symposium on Experiences with Distributed and Multiprocessor Systems,* pp. 57-72, 1993.

[BoI91] Borrmann, L., and P. Istavrinos, “Store Coherency in a Parallel Distributed-Memory Machine,” *Proceedings of the 2nd European Conference on Distributed Memory Computing*, pp. 32-41, 1991.

[CBZ91] Carter, J.B., J.K. Bennett and W. Zwaenepoel, “Implementation and Performance of Munin,” *Proceedings of the 13th ACM Symposium on Operating System Principles,* pp. 152-164, 1991.

[CeF78] Censier, L.M., and P. Feautrier, “A New Solution to Coherence Problems in Multicache Systems,” *IEEE Transactions on Computers*, Vol. 27, No. 12, pp. 1112-1118, 1978.

[CKA91] Chaiken, D., J. Kubiatowicz and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 224-234, 1991.

[ChA94] Chaiken, D., and A. Agarwal, "Software-Extended Coherent Shared Memory: Performance and Cost," *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 314-324, 1994.

[ChM79] Chandy, K.M., and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, Vol. 5, No. 5, pp. 440-452, 1979.

[Cha91] Charron-Bost, B., "Concerning the Size of Logical Clocks in Distributed Systems," *Information Processing Letters*, Vol. 39, No. 1, pp. 11-16, 1991.

[ChV88] Cheong, H., and A.V. Veidenbaum, "A Cache Coherence Scheme with Fast Selective Invalidation," *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 138-147, 1988.

[Che85] Cheriton, D.C., "Preliminary Thoughts on Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems," *Operating Systems Review*, Vol. 19, No. 4, pp. 26-33, 1985.

[ChY96] Choi, L., and P.C. Yew, "Compiler and Hardware Support for Cache Coherence in Large-Scale Multiprocessors: Design Considerations and Performance Study," *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 283-294, 1996.

[Col92] Collier, W.W., Reasoning about Parallel Architectures, Prentice Hall, 1992.

[CoF89] Cox, A.L., and R.J. Fowler, "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM," *Proceedings of the 12th ACM Symposium on Operating System Principles,* pp. 32-44, 1989.

[CoF93] Cox, A.L., and R.J. Fowler, "Adaptive Cache Coherency for Detecting Migratory Shared Data," *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 98-108, 1993.

[CDK94] Cox, A.L., S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony and W. Zwaenepoel, "Software versus Hardware Shared-Memory Implementation: A Case Study," *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 106-117, 1993.

[CKM88] Cytron, R., S. Karlovsky and K.P. McAuliffe, "Automatic Management of Programmable Caches (Extended Abstract)," *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 229-238, 1988.

[DDS94] Dahlgren, F., M. Dubois and P. Stenstrom, “Combined Performance Gains of Simple Cache Protocol Extensions,” *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 187-197, 1994.

[DLA91] Dasgupta, P., R.L. LeBlanc, Jr., M. Ahamad and U. Ramachandran, “The Clouds Distributed Operating System,” *IEEE Computer*, Vol. 24, No. 11, pp. 34-44, 1991.

[dWR96] de Supinski, B.R., C. Williams and P.F. Reynolds, Jr., “Performance Evaluation of the Late Delta Cache Coherence Protocol,” Tech. Rep. CS-96-05, University of Virginia, Department of Computer Science, 1996.

[DuL92] Dubnicki, C., and T.J. LeBlanc, “Adjustable Block Size Coherent Caches,” *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 170-180, 1992.

[DSB86] Dubois, M., C. Scheurich and F. Briggs, “Memory Access Buffering in Multiprocessors,” *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 434-442, 1986.

[DSR93] Dubois, M., J. Skeppstedt, L. Ricciulli, K. Ramamurthy and P. Stenstrom, “The Detection and Elimination of Useless Misses in Multiprocessors,” *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 88-97, 1993.

[DKC93] Dwarkadas, S., P. Keleher, A.L. Cox and W. Zwaenepoel, “Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology,” *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 144-155, 1993.

[DCZ96] Dwarkadas, S., A.L. Cox and W. Zwaenepoel, “An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System,” *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 186-197, 1996.

[EgK88] Eggers, S.J., and R.H. Katz, “A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation,” *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 373-382, 1989.

[EgJ91] Eggers, S.J., and T.E. Jeremiassen, “Eliminating False Sharing,” *Proceedings of the 1991 International Conference on Parallel Processing*, Vol. I, pp. 377-381, 1991.

[ENC96] Erlichson, A., N. Nuckolls, G. Chesson and J. Hennessy, “SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory,” *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 210-220, 1996.

[FLR94] Falsafi, B., A.R. Lebeck, S.K. Reinhardt, I. Schoinas, M.D. Hill, J.R. Larus, A. Rogers and D.A. Wood, "Application-Specific Protocols for User-Level Shared Memory," *Proceedings of Supercomputing '94*, pp. 380-389, 1994.

[FVS95] Farkas, K., Z. Vranesic and M. Stumm, "Scalable Cache Consistency for Hierarchically Structured Multiprocessors," *Journal of Supercomputing*, Vol. 8, No. 4, pp. 345-369, 1995.

[Fid91] Fidge, C., "Logical Time in Distributed Computing Systems," *IEEE Computer*, Vol. 24, No. 8, pp 28 - 33, 1991.

[FlP89] Fleisch, B.D., and G.J. Popek, "Mirage: A Coherent Distributed Shared Memory Design," *Proceedings of the 12th ACM Symposium on Operating System Principles,* pp. 211-223, 1989.

[FBR93] Frank, S., H. Burkhardt III and J. Rothnie, "The KSR1: Bridging the Gap Between Shared Memory and MPPs," *Proceedings of Compcon '93*, pp. 285-294, 1993.

[Fri95] Friedman, R., "Using Virtual Synchrony to Develop Efficient Fault Tolerant Distributed Shared Memories," Tech. Rep. 95-1506, Cornell University, Department of Computer Science, 1995.

[FuP93] Fu, J.W.C., and J.H. Patel, "Memory Reference Behavior of compiler Optimized Programs on High Speed Architectures," *Proceedings of the 1993 International Conference on Parallel Processing*, Vol. II, pp. 87-94, 1993.

[GLL90] Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 15-26, 1990.

[Gha95] Gharachorloo, K., Memory Consistency Models for Shared Memory Multiprocessors, Ph.D. Thesis, Stanford University, 1995.

[GhS91] Ghose, K., and S. Simhadri, "A Cache Coherency Mechanism with Limited Combining Capabilities for MIN-Based Multiprocessors," *Proceedings of the 1991 International Conference on Parallel Processing*, Vol. I, pp. 296-300, 1991.

[GDF93] Glasco, D.B., B.A. Delagi and M.J. Flynn, "Update-Based Cache Coherence Protocols for Scalable Shared-Memory Multiprocessors," Tech. Rep. CSL-TR-93-588, Stanford University, Computer Systems Laboratory, 1993.

[Goo83] Goodman, J.R., "Using Cache Memory to Reduce Processor-Memory Traffic," *Proceedings of the 10th International Symposium on Computer Architecture*, pp. 124-131, 1983.

[GoW88] Goodman, J.R., and P.J.Woest, "The Wisconsin Multicube: A New Large-Scale cache-Coherent Multiprocessor," *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 422-431, 1988.

[Goo89] Goodman, J.R., "Cache Consistency and Sequential Consistency," Tech. Rep. 61, SCI Committee, 1989.

[GVW89] Goodman, J.R., M.K. Vernon and P.J.Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 64-73, 1989.

[GrS95] Grahn, H., and P. Stenstrom, "Efficient Strategies for Software-Only Directory Protocols in Shared Memory Multiprocesors," *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 38-47, 1995.

[GrT90] Graunke, G., and S. Thakkar, "Synchronization Algorithms for Shared Memory Multiprocessors," *IEEE Computer*, Vol. 23, No. 6, pp 60 - 69, 1990.

[GWM90] Gupta, A., W.D. Weber and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," *Proceedings of the 1990 International Conference on Parallel Processing*, Vol. I, pp. 312-321, 1990.

[HHW90] Hagersten, E., S. Haridi and D.H.D. Warren, "The Cache Coherence Protocol of the Data Diffusion Machine," pp. 165-188, in Cache and Interconnect Architectures in Multiprocessors, M. Dubois and S. Thakkar, editors, Kluwer Academic Publishers, 1990.

[HeM92] Herlihy, M., and J.E.B. Moss, "Transactional Memory: Architectural Support of Lock-Free Data Structures," Tech. Rep. CRL 92/07, Digital Equipment Corporation, 1992.

[HeM93] Herlihy, M., and J.E.B. Moss, "Transactional Memory: Architectural Support of Lock-Free Data Structures," *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 289-300, 1993.

[HuA90] Hutto, P.W., and M. Ahamad, "Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories," *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 302-309, 1990.

[HyF96] Hyde, R.L., and B.D. Fleisch, "An Analysis of Degenerate Sharing and False Coherence," *Journal of Parallel and Distributed Computing*, Vol. 34, No. 2, pp. 183-195, 1996.

[IDF96] Iftode, L., C. Dubnicki, E.W. Felten and K. Li, "Improving Release-Consistent Shared Virtual Memory Using Automatic Update," *Proceedings of the 2nd International Symposium on High Performance Computer Architecture*, pp. 372-381, 1996.

[ISL96] Iftode, L., J.P. Singh and K. Li, "Understanding Application Performance on Shared Virtual Memory Systems," *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 122-133, 1996.

[Jam90] James, D.V., "SCI (Scalable Coherent Interface) Cache Coherence," pp. 189-208, in Cache and Interconnect Architectures in Multiprocessors, M. Dubois and S. Thakkar, editors, Kluwer Academic Publishers, 1990.

[JaJ94] Jard, C., and G.V. Jourdan, "Dependency Tracking and Filtering in Distributed Computations," Tech. Rep. 851, IRISA, Campus de Beaulieu, 1994.

[Jef85] Jefferson, D.R., "Virtual Time," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404-425, 1985.

[JeE95] Jeremiassen, T.E., and S.J. Eggers, "Reducing False Sharing on Shared Memory Multiprocessors," *Proceedings of the 5th ACM Symposium on the Principles and Practices of Parallel Programming*, pp. 179-188, 1995.

[JoA94] John, R., and M. Ahamad, "Evaluation of Causal Distributed Shared Memory for Data-Race-Free Programs," Tech. Rep. GIT-CC-94/34, Georgia Institute of Technology, College of Computing, 1994.

[JKW95] Johnson, K.L., M.F. Kaashoek and D.A. Wallach, "CRL: High Performance All-Software Distributed Shared Memory," *Proceedings of the 15th ACM Symposium on Operating System Principles,* pp. 213-228, 1995.

[KMR88] Karlin, A.R., M.S. Manasse, L. Rudolph and D.D. Sleator, "Competitive Snoopy Caching," *Algorithmica*, Vol. 3, No. 1, pp. 79-119, 1988.

[KEW85] Katz, R.H., S.J. Eggers, D.A. Wood, C.L. Perkins and R.G. Sheldon, "Implementing a Cache Consistency Protocol," *Proceedings of the 12th International Symposium on Computer Architecture*, pp. 276-283, 1985.

[KCZ92] Keleher, P., A.L. Cox and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 13-21, 1992.

[KLE93] Khera, V., R.P. LaRowe, Jr. and C.S. Ellis, "An Architecture-Independent Analysis of False Sharing," Tech. Rep. CS-1993-13, Duke University, Department of Computer Science, 1993.

[KoS95] Kontothanassis, L.I., and M.L. Scott, "Distributed Shared Memory for New Generation Networks," Tech. Rep. 578, University of Rochester, Department of Computer Science, 1995.

[Lam78] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, No. 7, pp. 558-565, 1978.

[Lam79] Lamport, L., "How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs," *IEEE Transactions on Computers*, Vol. 28, No. 9, pp. 690-691, 1979.

[Lam86] Lamport, L., "On Interprocess Communication," *Distibuted Computing*, Vol. 1, No. 2, pp. 77-101, 1986.

[LeA92] Leong, H.V., and D. Agrawal, "Type-specific Coherence Protocols for Distributed Shared Memories," *Proceedings of the 12th International Conference on Distributed Computing Systems*, pp. 434 - 441, 1992.

[LiH86] Li, K., and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *Proceedings of the 5th Annual ACM Symposium on Distributed Computing*, pp. 229-239, 1986.

[LiH89] Li, K., and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pp. 321-359, 1989.

[LiS89] Li, K., and R. Schaefer, "Shared Virtual Memory for a Hypercube Multiprocessor," *Proceedings of the International Conference on Parallel Processing*, pp. 371-378, 1989.

[LiY90] Lilja, D.J., and P.C. Yew, "A Compiler-Assisted Directory-Based Cache Coherence Scheme," Tech. Rep. CSD-990, University of Illinois, Center for Supercomputing Research and Development, 1990.

[Lom77] Lomet, D.B., "Process Structuring, Synchronization, and Recovery Using Atomic Actions," *SIGPLAN Notices*, Vol. 12, No. 3, pp. 128-137, 1977.

[Mat89] Mattern, F., "Virtual Time and Global States of Distributed Systems," pp. 215-226, in Parallel and Distributed Algorithms, M. Cosnard and P. Quinton, editors, Elsevier Science Publishers, 1989.

[Mat93] Mattern, F., "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation," *Journal of Parallel and Distributed Computing*, Vol. 18, No. 4, pp. 423-434, 1993.

[McC85] McCreight, E.M., "The Dragon Computer System, an Early Overview," pp. 83-101, in Microarchitecture of VLSI Computers, P. Antognetti, F. Anceau and J. Vuillemin, editors, Martinus Nijhoff Publishers, 1985.

[MiB92] Min, S.L., and J.L. Baer, "Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 1, pp. 25-44, 1992.

[MRS93] Mizuno, M., M. Raynal, G. Singh and M.L. Neilsen, "An Efficient Implementation of Sequentially Consistent Distributed Shared Memories," Tech. Rep. 208, IRISA, INRIA, 1993.

[MoL95] Mounes-Toussi, F., and D.J. Lilja, "The Potential of Compile-Time Analysis to Adapt the Cache Coherence Enforcement Strategy to the Data Sharing Characteristics," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 5, pp. 470-481, 1995.

[NaB93] Nanda, A.K. and Bhuyan, L.N., "Design and Analysis of Cache Coherent Multistage Interconnection Networks," *IEEE Transactions on Computers*, Vol. 42, No. 4, pp. 458-470, 1993.

[OKN90] O'Krafka, B.W., and A.R. Newton, "An Empirical Evaluation of Two Memory-Efficient Directory Methods," *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 138-147, 1990.

[OwL82] Owicki, S., and L. Lamport, "Proving Liveness Properties of Concurrent Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 455-495, 1982.

[OwA89] Owicki, S., and A. Agarwal, "Evaluating the Performance of Software Cache Coherence," *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 230-242, 1989.

[Pap86] Papadimitriou, C., Database Concurrency Control, Computer Science Press, 1986.

[PaP84] Papamarcos, M.S., and J.H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proceedings of the 11th International Symposium on Computer Architecture*, pp. 348-354, 1984.

[PST91] Peir, J.K., K. So and J.H. Tang, "Intersection Locality of Shared Data in Parallel Programs," *Proceedings of the 1991 International Conference on Parallel Processing*, Vol. I, pp. 278-286, 1991.

[Pet81] Peterson, J.L., Petri Net Theory and the Modeling of Systems, Prentice Hall, 1981.

[PiB92] Pinkston, T.M., and S.J. Baylor, "Parallel Processor Memory Reference Analysis: Examining Locality and Clustering Potential," *Proceedings of the 5th SIAM Conference on Parallel Processing for Scientific Computing*, pp. 513-518, 1992.

[RaL96] Ramachandran, U., and J. Lee, “Cache-based Synchronization in Shared memory Multiprocessors,” *Journal of Parallel and Distributed Computing*, Vol. 32, No. 1, pp. 11-27, 1996.

[Ran87] Ranade, A.G., “How to Emulate Shared Memory,” *Proceedings of the IEEE Annual Symposium on Foundations of Computer Science*, pp. 185-194, 1987.

[RBJ88] Ranade, A.G., S.N. Bhatt and S.L. Johnson, “The Fluent Abstract Machine,” Tech. Rep. 573, Yale University, Department of Computer Science, 1988.

[Ran89] Ranade, A.G., “Fluent Parallel Computation,” Tech. Rep. 663, Yale University, Department of Computer Science, 1989.

[RaS95] Raynal, M., and A. Schiper, “From Causal Consistency to Sequential Consistency in Shared Memory Systems,” Tech. Rep. 2557, IRISA, INRIA, 1995.

[RaS96] Raynal, M., and M. Singhal, “Logical Time: Capturing Causality in Distributed Systems,” *IEEE Computer*, Vol. 29, No. 2, pp 49 - 56, 1996.

[Reg97] J. Regehr, “An Isotach Implementation for Myrinet,” Tech. Rep. CS-97-12, University of Virginia, Department of Computer Science, 1997.

[RPW96] Reinhardt, S.K., R.W. Pfile and D.A. Wood, “Decoupled Hardware Support for Distributed Shared Memory,” *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 34-43, 1996.

[RWW97] Reynolds, Jr., P.F., C. Williams and R.R. Wagner, Jr., “Isotach Networks,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 4, pp. 337-348, 1997.

[RuS84] Rudolph, L., and Z. Segall, “Dynamic Decentralized Cache Schemes for MIMD Parallel Processors,” *Proceedings of the 11th International Symposium on Computer Architecture*, pp. 340-347, 1984.

[SGZ93] Sandhu, H.S., B. Gamsa and S. Zhou, “The Shared Regions Approach to Software Cache Coherence on Multiprocessors,” *Proceedings of the 4th ACM Symposium on Principles and Practices of Parallel Programming*, pp. 229-238, 1993.

[SGT96] Scales, D.J., K. Gharachorloo and C.A. Thekkath, “Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory,” *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 174-185, 1996.

[Sch88] Schmuck, F., The Use of Efficient Broadcast in Asynchronous Distributed Systems, Ph.D. Thesis, Cornell University, 1988.

[SFL94] Schoinas, I., B. Falsafi, A.R. Lebeck, S.K. Reinhardt, J.R. Larus and D.A. Wood, "Fine-grain Access Control for Distributed Shared Memory," *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 297-306, 1994.

[SSR95] Shah, G., A. Singla and U. Ramachandran, "The Quest for a Zero Overhead Shared Memory Parallel Machine," *Proceedings of the 1994 International Conference on Parallel Processing*, Vol. I, pp. 194-201, 1994.

[SiH91] Simoni, R., and M. Horowitz, "Modeling the Performance of Limited Pointer Directories for Cache Coherence," *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 309-318, 1991.

[SiK92] Singhal, M., and A. Kshemkalyani, "An Efficient Implementation of Vector Clocks," *Information Processing Letters*, Vol. 43, No. 1, pp. 47-52, 1992.

[SkS94] Skeppstedt, J., and P. Stenstrom, "Simple Compiler Alorithms to Reduce Ownership Overhead in Cahce Coherence Protocols," *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 286-296, 1994.

[SlT85] Sleator, D.D., and R.E. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Communications of the ACM*, Vol. 28, No. 2, pp. 202-208, 1985.

[SPA92] SPARC International, Inc., SPARC Architecture Manual: Version 8, Prentice Hall, 1992.

[Sri96] Srinivasa, R., Parallel Rule-Based Isotach Systems, M.S. Thesis, University of Virginia, 1996.

[SBS93] Stenstrom, P., M. Brorsson and L. Sandberg, "An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing," *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 109-118, 1993.

[SSH93] Stone, J.M., H.S. Stone, P. Heidelberger and J. Turek, "Multiple Reservations and the Oklahoma Update," *IEEE Journal of Parallel and Distributed Technology*, Vol. 1, No. 4, pp. 58-71, 1993.

[TaJ92] Tamir, Y., and G. Janakiraman, "Hierarchical Coherency Management for Shared Virtual Memory Multicomputers," *Journal of Parallel and Distributed Computing*, Vol. 15, No. 4, pp. 408-419, 1992.

[TKB92] Tanenbaum, A.S., M.F. Kaashoek and H.E. Bal, "Parallel Programming Using Shared Objects and Broadcasting," *IEEE Computer*, Vol. 25, No. 8, pp. 10-19, 1992.

[Tan76] Tang, C.K. “Cache Design in the Tightly Coupled Multiprocessor System,” *AFIPS Conference Proceedings National Computer Conference*, pp. 749-753, 1976.

[TSS88] Thacker, C.P, L.C. Stewart and E.H. Satterthwaite, Jr., “Firefly: A Multiprocessor Workstation,” *IEEE Transactions on Computers*, Vol. 37, No. 8, pp. 909-920, 1988.

[TCS92] Thacker, C.P, D.G. Conroy and L.C. Stewart, “The Alpha Demonstration Unit: A High-performance Multiprocessor for Software and Chip Developement,” *Digital Technical Journal*, Vol. 4, No. 4, pp. 51-65, 1995.

[TLH94] Torrellas, J., M.S. Lam and J.L. Hennessy, “False Sharing and Spatial Locality in Multiprocessor Caches,” *IEEE Transactions on Computers*, Vol. 43, No. 6, pp. 651-663, 1994.

[VeF92] Veenstra, J.E., and R.J. Fowler, “A Performance Evaluation of Optimal Hybrid Cache Coherency Protocols,” *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 149-160, 1992.

[WeG89] Weber, W.D., and A. Gupta, “Analysis of Cache Invalidation Patterns in Multiprocessors,” *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 243-256, 1994.

[Wes94] Westbrook, J., “Randomized Algorithms for Multiprocessor Page Migration,” SIAM *Journal on Computing*, Vol. 23, No. 5, pp. 951-965, 1994.

[Wil93] Williams, C., Concurrency Control in Asynchronous Computations, Ph.D. Thesis, University of Virginia, 1993.

[WiR95] Williams, C., and P.F. Reynolds, Jr., “Combining Atomic Actions,” *Journal of Parallel and Distributed Computing*, Vol. 24, No. 2, pp. 152-163, 1995.

[Wil97] Williams, C., Personal communication, 1997.

[WiR97] Williams, C., and P.F. Reynolds, Jr., “Isotach Prototype Design Specification,” Internal Working Document, 1997.

[WiL92] Wilson, Jr., A.W., and R.P. LaRowe, Jr., “Hiding Shared Memory Reference Latency on the Galactica Net Distributed Shared Memory Architecture,” *Journal of Parallel and Distributed Computing*, Vol. 15, No. 4, pp. 351-367, 1992.

[WOT95] Woo, S.C., M. Ohara, E. Torrie, J.P. Singh and A. Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 38-47, 1995.

[WCF93] Wood, D.A., S. Chandra, B. Falsafi, M.D. Hill, J.R. Larus, A.R. Lebeck, J.C. Lewis, S.S. Mukherjee, S. Palacharla and S.K. Reinhardt, “Mechanisms for Cooperative Shared Memory,” *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 156-167, 1993.

[YTB92] Yang, Q., G. Thangadurai and L.N. Bhuyan, “Design of an Adaptive Cache Coherence Protocols for Large Scale Multiprocessors,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 3, pp. 281-293, 1992.

[YKA96] Yeung, D., J. Kubiatowicz and A. Agarwal, “MGS: A Multigrain Shared Memory System,” *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 44-55, 1996.

[ZIL96] Zhou, Y., L. Iftode and K. Li, “Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems,” *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pp. 75-88, 1996.

Appendix: Glossary

AA: Add address operation; page 142.

Adaptive protocol: Protocol that dynamically identifies and exploits reference patterns; page 21.

Add address (AA) operation: Coherence operation that adds a location to a directory in the local update protocol; page 142.

Assign directory: Directory of invalidated copies that still need to receive assign update messages in owner invalidation protocol; page 119.

Assign request: Part of a split operation that is a request to associate a value with a write request; page 15.

Atomic action: A group of requests; page 15.

Atomicity: Ordering constraint that requires the apparent indivisible execution of atomic actions; page 15.

Blue phase: Phase type in flex algorithm; page 44.

Blue port: Port type in flex algorithm; page 44.

Causality: Concept of an event influencing or determining the outcome or occurrence of another event; page 26.

Change of address message (COA): Virtual message of local copy algorithm of invalidation action that changes local record of owner location; page 124.

Cluster locality: The tendency of a processor to access a coherence unit recently accessed by a physically proximate processor; page 24.

COA: Change of address virtual message; page 124.

Coherence action: Messages and execution events that satisfy a request; page 20.

Coherence granularity: Coherence unit size in number of variables; page 9.

Coherence maintenance problem: Concurrency control problem with replication; page 9.

Coherence operation: Message to distribute effect of a write request or alter state of copies; page 20.

Coherence unit: Coherence protocol state information unit; page 9.

Combined processing element: PE with local memory that acts as an MM; page 32.

Complete directory: Directory that at a given execution time contains any location with a copy whose lifetime includes the given execution time; page 82.

Competitive algorithm: Algorithm for which cost is less than a constant times the optimal cost for any input; page 21.

Competitive coefficient: Constant that bounds cost of a competitive algorithm for any input relative to the optimal cost; page 21.

Concurrency control problem: To ensure that every execution of a parallel program is consistent with its ordering constraints; page 9.

Concurrent events: Two events such that neither potentially causes the other; page 38.

Conflict equivalence: Agree on order of all conflicting requests; page 11.

Conflicting requests: Requests, at least one a write, that access the same variable; page 11.

Consistency: Ordering constraint in which all processes observe writes to a given memory location in the same order; page 9.

Consistency semantics: Ordering constraint that limits the values that the system can associate with a read request; page 11.

Consistency w/potential causality: **a** ↱ **b** implies $\mathbf{t_a} \le \mathbf{t_b}$; page 39.

Copy: Instance of a coherence unit; page 10.

Correct directory: Directory that is complete at every execution time during its lifetime; page 82.

Correctness set: All shared memory executions that conform to an ordering constraint; page 11.

Corresponding interprocess events: Issuing send and receive events; or network send and receive events; or delivering send and receive events; page 37.

Corresponding messaging process events: Corresponding events internal to a single messaging process: issuing receive event and network send event; or network receive event and delivering send event; page 37.

Delayed response: Response message where logical send time equals original message's logical receive time plus **c** > 0; page 31.

Deliver event: Event that returns value associated with a read request to issuing process; page 10.

Delivering receive event: Receive event in a user process; page 37.

Delivering send event: Send event by a messaging process to a user process; page 37.

Delta coherence protocol: Isotach-based coherence protocol; page 35.

Destruction event: Execution event that destroys a copy; page 73.

Destruction message: Message that causes destruction event of existing copy; page 103.

Directory: List of locations of copies of a coherence unit; page 18.

Disjoint copies: Existing copy and its replacement for which corresponding logical receive times are disjoint; page 95.

Distinguished update: Update that decrements the reservation count at issuing SIU but does not execute; used with separated IR in owner protocols; page 106.

Distributed shared memory (DSM): A mechanism that provides transparent shared memory in systems that limit physical memory access to the local node; page 8.

DSM: Distributed shared memory; page 8.

Dynamic fixed routing path: Routing path chosen at the time the message is sent; page 32.

Dynamic protocol: Protocol that uses run-time coherence operations; page 20.

Dynamic replication: Coherence mechanism that allows copy locations to change during program execution; page 10.

Dynamic routing path: Routing path chosen as message travels through network; page 32.

Equivalent constraints: Ordering constraints with the same correctness sets; page 11.

Equivalent executions: Same requests associated with the same values; page 11.

Exact directory: Complete directory at a given execution time that only contains locations that have copies whose lifetimes include the given execution time; page 98.

Execution: Every execution event of all shared memory requests and the associated values; page 11.

Execution displacement: Integer constant, δ, added to logical receive time to determine execution time; page 64.

Execution distance: Difference, Φ, between execution time of an execution event and initial send time of the request; page 64.

Execution event: Event of storing or associating request value; page 10.

Execution order: Real time (total) order of execution events; page 11.

Execution time: Logical time assigned to an execution event by the logical execution time system; page 62.

Execution time function: Function for a copy that determines execution time from logical receive time at node; page 62.

Existing copy: Copy that ownership transition destroys; page 94.

Extended isonet algorithm: Simple extension of isonet algorithm that allows logical distances to be different from routing distances; page 52.

Extensible isotach network: Isotach network in which logical send time of response message can be a known function of logical receive time of original message; pages 6, 31.

False sharing: Requests by different processors to unrelated variables in same coherence unit; page 24.

Filling state: State of a local copy that has been allocated but not instantiated; page 74.

Fixed routing path: Routing path known to sender at time message is sent; page 32.

Flat atomic action: An atomic action with no internal true dependences; page 15.

Flex algorithm: Isotach network algorithm that allows logical distances different from routing distances; pages 4, 42.

Frequently read/written variable: Reference pattern characterized by interleaved read and write requests by many processes; page 24.

Full replication: Static replication with copy at every node; page 29.

Green phase: Phase type in flex algorithm; page 44.

Green port: Port type in flex algorithm; page 44.

Happens before relation: Formalization of concept of time in distributed systems; page 27.

Hardware DSM: DSM mechanism that uses special purpose hardware; page 23.

Hardware protocol: Dynamic protocol; page 20.

Highly concurrent action: A coherence action that does not restrict the concurrent use of other coherence actions; page 93.

Home copy: Query location for miss actions; pages 20, 73.

Immediate response: Response message with logical send time equal to logical receive time of original message; page 31.

IN: Interconnection network; page 32.

Initial send time: Logical send time of first message used to service a request; page 64.

Initial token count: Number of tokens, $\mathbf{t_q} \geq 0$, placed on output of a port before any messages are routed (sent); page 44.

Instantiation action: Coherence action of the local update protocol that creates a new local copy; page 136.

Instantiation event: Execution event of that initializes a new copy; page 73.

Instantiation message: Message that causes instantiation event of replacement copy; page 103.

Instantiation request (IR): Request for a new local copy; page 82.

Invalidation: Coherence operation that destroys a copy; page 20.

Invalidation action: Ownership transition that destroys all existing copies but does not create replacements except at new owner location; page 118.

IR: Instantiation request; page 82.

Isochron: A flat atomic action in a fault free system; page 15.

Isochronicity: Ordering constraint that requires all possible executions to be isochronous; page 15.

Isochronous execution: Each isochron appears to execute indivisibly; page 15.

Isonet algorithm: Original isotach network algorithm; page 42.

Isotach invariant: Message travels one unit of logical distance per pulse of logical time, i.e. at unit speed; page 30.

Isotach logical time system: An LTS that is consistent with potential causality and enforces the isotach invariant; page 30.

Isotach network: Network that realizes an isotach LTS; page 31.

Issue event: Event that provides a request to an SIU; page 10.

Issuing copy: Local copy associated with issuing SIU; pages 77, 141.

Issuing receive event: Receive event in a messaging process that corresponds to an issuing send event; page 37.

Issuing send event: Send event by a user process to a messaging process; page 37.

Issuing SIU: SIU associated with process that issued the request; page 32.

Level of message service: Guarantee for the logical receive time of a message based on its logical send time; page 33.

Lifetime (of a copy): Period of logical execution time between the execution times of the instantiation and destruction events of the copy; page 73.

Limited directory protocol: Protocol that restricts number of copies to less than number of nodes; page 19.

Live copy directory: Directory with current copies in owner invalidation protocol; page 119.

Local copy: Any copy that is located at a PE (including the owner copy in protocols with an owner copy); page 73.

Local update protocol: Update delta coherence protocol in which issuing SIU distributes updates; pages 6, 135.

Locality: The tendency of future requests to reflect previous requests; page 23.

Logical clock: Counter for pulse component of logical time of a port; page 43.

Logical diameter: Maximum logical distance, **D**, in an isotach network; page 33.

Logical distance: Fixed logical time, $\mathbf{d_{A,B}}$, for a message to travel from node **A** to node **B** in an isotach network; page 30.

Logical execution: Execution in which execution events occur in order of their execution times; page 63.

Logical execution time: Meta-isotach logical time system of isotach shared memory systems that models execution equivalence; page 62.

Logical receive clock: Counter for pulse component of logical time of receive event at an SIU; page 43.

Logical routing distance: Logical time that a switch takes to route a message; page 48.

Logical send clock: Counter for pulse component of logical time of send event at an SIU; page 43.

Logical time deadlock: Condition in which the pulse component of logical time never again increases; page 52.

Logical time system (LTS): Causality-based method for numbering system events; page 26.

Logical topology: Fully connected weighted graph of network elements with logical distance edge weights; page 65.

LTS: Logical time system; pages 4, 26.

Marked graph: Petri net structure in which each place is an input of exactly one transition and an output of exactly one transition; page 54.

Memory process: Process that executes shared memory requests; page 62.

Messaging process: Process that sends and receives messages for a node; page 36.

Meta-isotach logical time system: Logical time system built on top of an underlying isotach Net LTS; page 61.

Migration action: Ownership transition that replaces all existing copies; page 93.

Migratory variable: Reference pattern characterized by periods during which only one process issues requests to variable; page 21.

MIN: Multistage interconnection network; page 32.

Miss action Coherence action used when no local copy exists; page 20.

Memory module: Network element that is location of home copies; page 32.

MM: Memory module; page 32.

Mostly read variable: Reference pattern characterized by read requests by many processes and few write requests by any processes; page 24.

Network element: Processing element or memory module; page 32.

Network (Net) LTS: LTS that only numbers network events; page 39.

Network receive event: Corresponding receive event for a network send event; page 37.

Network send event: Send event between messaging processes; page 37.

New owner copy: Owner copy after ownership transition; page 94.

Off-line algorithm: Algorithm that uses knowledge of the future; page 21.

Old owner copy: Owner copy before ownership transition; page 94.

On-line algorithm: Algorithm that does not use knowledge of the future; page 21.

Optimistic isotach systems: Isotach systems that deliver messages as they arrive and recover from out of logical time order delivery; page 153.

Overlapping copies: Existing copy and its replacement for which corresponding logical receive times overlap; page 95.

Owner copy: Distinguished local copy with responsibility for tracking copies and distributing coherence operations, pages 5, 73.

Owner epoch: Logical execution time period with single owner copy; page 113.

Owner invalidation protocol: Invalidation protocol derived by modifying migration action of owner update protocol; pages 6, 118.

Owner update protocol: Update delta coherence protocol that allows owner location to change dynamically; page 5.

Ownership transition: A coherence action that changes owner location; page 93.

Page management problem: To locate current version; page 20.

PE: Processing element; page 32.

Phase type: In flex algorithm, type of ports from which messages are currently being routed; page 44.

Physical topology: Connected graph, (V, E), where V is the set of network elements and switches and E is the set of message links; page 32.

Ping-ponging: Effect of alternating write requests to same coherence by two processors with an invalidation protocol; page 24.

Pipelined requests: Concurrently serviced requests by the same process; page 74.

Potential causality: Refinement of the happens before relation for systems that use messaging processes; pages 3, 38.

Processing element (PE): Network element that can serve as a process location; page 32.

Processor consistency: Program order and write atomicity simultaneously; page 12.

Processor locality: The tendency of a processor to access a block repeatedly before an access from another processor; page 24.

Producer/consumer variable: Reference pattern characterized by all write requests by one process and read requests by one or more other processes; page 24.

Program order: Ordering constraint that requires an equivalent execution exists in which the requests of each process occur in the sequential order specified by its program; page 12.

Protocol variant: Protocol that adds same constant to every execution and scheduling displacement of original protocol; page 90.

Pulse component: Major component of isotach logical time; page 30.

Receive event: Process event that occurs when it receives a message; pages 10, 43..

Receive lifetime (of a copy): Period of local logical receive time line that corresponds to lifetime of the copy; page 81.

Release directory: Directory in which the new owner stores releases until it receives the directory message in the owner invalidation protocol; page 126.

Release message: Message to remove location of victim copy from directory; page 74.

Replacement copy: Any copy that replaces an existing copy in owner protocols; page 94.

Replacement policy: Policy that determines copy to select as a victim; page 74.

Request: Shared memory access; page 10.

Request forwarding Intermediate location forwards request to copy that executes the request; page 64.

Reservation count: Number of scheduled locally issued requests to coherence unit that have not completed; page 74.

Reserved copy: Copy with non-zero reservation count; page 74.

Response message: Message that execution of another message generates; page 31.

Routing distance: Number of intermediate nodes on routing path; page 33.

Routing event: Switch moving message from input to output; page 43.

Routing path: Set of network nodes (elements or switches) through which a message passes; page 32.

Sched request: Part of a split operation that is a request to schedule the execution event of a write request; page 15.

Scheduled execution: Execution with requests in the scheduled execution order; page 66.

Scheduled execution order: Scheduled execution time (total) order of requests; page 66.

Scheduled execution pulse: Pulse component of scheduled execution time; page 67.

Scheduled execution time: Sum, τ, of initial send time of a request and its scheduling displacement; page 65.

Scheduled logical time: Meta-isotach logical time system of isotach shared memory systems that models its ordering constraints; page 65.

Scheduling algorithm: Method to select scheduled execution times and initial send times of requests; page 67.

Scheduling decision: Determination of initial send time of request; page 65.

Scheduling displacement: Offset, χ, of scheduled execution time from initial send time; selected by issuing SIU; page 66.

Scheduling horizon: Bound, **H**, of scheduled execution time of any locally issued request relative to local logical receive times; page 98.

Send discipline: Relationship of logical send and receive times for different destinations of a multicast; page 90.

Send event: Process event that occurs when it sends a message; pages 10, 43.

Separated IR: Instantiation request that the old owner copy separates from a miss request and forwards to the new owner copy in owner protocols; page 106.

Sequential consistency: Program order and write atomicity; page 12.

SIU: Switch interface unit; page 32.

Software-assisted protocol: Static protocol; page 22.

Software DSM: DSM mechanism implemented entirely in software; page 23.

Software extended directory: Software mechanism that allows number of copies to exceed number of hardware pointers; page 19.

Spatial locality: The tendency of programs to request variables whose addresses are near recently requested variables; page 23.

Split operation: Mechanism that divides a write request into a sched request and an assign request; pages 6, 15.

Standard level of service: Level of message service which maintains isotach invariant; page 33.

Static fixed routing path: Known routing path used by every message between a sender/receiver pair; page 32.

Static owner update protocol: Delta coherence protocol with fixed owner copy location; pages 5, 72.

Static protocol: Protocol that uses static methods to guarantee an exclusive copy exists whenever a write request is issued; page 22.

Static replication: Coherence mechanism that determines copy locations at start of program execution; page 10.

Strongly competitive algorithm: Algorithm with minimum possible competitive coefficient; page 21.

Structured atomic action: Atomic action with internal dependences; page 15.

Substantiate: Execute an assign request; page 15.

Supplying event: Event that supplies the values associated with an instantiation event; page 73.

Switch interface unit (SIU): Intermediate entity that manages isotach logical time for a network element; page 32.

Synchronization variable: Locks implemented in shared memory; page 24.

Tag (message tag): Minor components of logical times of all events of the message; page 44.

Temporal locality: The tendency of programs to request recently requested variables again; page 23.

TO: Transition operation; page 101.

Token (logical time token): Control message that marks the end of a pulse; page 43.

Transition aware request: Any request scheduled when the local copy of its issuing SIU is in a transition state; page 111.

Transition operation (TO): Coherence operation that announces ownership transition; page 101.

Transition record: Local record of ownership transition information; page 94.

Transition states: Local copy states used during ownership transition: migrating, disjoint and overlapping; page 94.

Transition vID: Special version identifier associated with initial value of a copy; page 96.

True sharing: Requests by different processors to same variable; page 24.

Uniform copy: A copy that is initialized correctly, executes all write requests with scheduled execution times during its lifetime and only executes read requests with scheduled execution times during its lifetime; page 79.

Unsubstantiated read: Read request associated with an unsubstantiated value; page 15.

Unsubstantiated value: The value of a write request for which the assign request has not executed; page 15.

Update: Coherence operation that executes write request on a copy; page 20.

Variable: Basic unit of all shared memory accesses; page 9.

Version identifier (vID): Tag that associates sched request and corresponding assign; page 87.

Victim copy: Copy that is destroyed to free storage for a new copy; page 74.

vID: Version identifier; page 87.

Virtual message: Message for which the sending node is the destination node; page 33.

Write atomicity: Ordering constraint that requires an equivalent execution exists in which the multiple execution events of each write request occur consecutively; page 12.

Write run length: Number of consecutive write requests to a coherence unit by one process before any read or write request by another process; a measure of processor locality; page 24.