Implementing Global Cache Coherence In $T-NG$

by

Xiaowei Shen

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of
the Requirements for the Degree of
Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May, 1995

© Massachusetts Institute of Technology 1995

Signature of Author

Department of Electrical Engineering and Computer Science

May 26, 1995

Certified by

Arvind

Professor of Electrical Engineering and Computer Science

Thesis Supervisor

Accepted by

Chairman, Frederic R. Morgenthaler

Chairman, Departmental Committee on Graduate Students
Implementing Global Cache Coherence In $\star T-NG$

by

Xiaowei Shen

Submitted to the Department of Electrical Engineering and Computer Science on May 26, 1995 in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science

Abstract

This thesis presents the design of cache coherence schemes for $\star T-NG$, a parallel system that supports both fine-grain, user-level message passing and global shared memory. The coherence schemes are implemented in software with very limited hardware assistance, allowing considerable flexibility for experimentation with various protocols.

For performance and correctness reasons, we find that buffering is preferable to retry for protocol messages that temporarily cannot be processed. The Eager Wake-up Management ensures that messages are only buffered when necessary, and then resumed at the earliest possible time. A study of typical scenarios shows that FIFO message passing usually simplifies the design of coherence schemes, and suggests that forwarding be adopted only in non-FIFO networks.

With respect to an abstract shared-memory model, we specify the operational semantics of a representative Base coherence scheme, and prove that the scheme implements sequential consistency without deadlock, livelock and starvation.

To support fine-grain synchronization, we argue that incorporating I-structures at the coherence protocol level is more efficient than implementing I-structures as a separate layer over the underlying coherence protocol. Issues surrounding the maintenance of deferred lists are examined, and different approaches are evaluated.

Thesis Supervisor: Arvind

Title: Professor of Electrical Engineering and Computer Science
Acknowledgments

First I would like to thank Prof. Arvind for his supervision. He has always been enthusiastic, encouraging and patient. Since I joined MIT in 1993, I have consistently benefited from his deep insights and constructive criticisms. He read through this thesis carefully, providing valuable comments and suggestions that made indispensable contributions to the completion of this work.

Many thanks to Prof. Jack Dennis for the helpful discussions on I-structures and other issues. Thanks also go to Prof. Guang R. Gao of the McGill University. During his visit at MIT, he listened attentively to my presentations and raised many questions which led me to improve my protocol verification technique.

Special thanks to my officemate, Boon S. Ang, whose friendly collaboration helped me start on the correct track. Many thanks to Derek Chiou, James Hoe and Andy Shaw. Derek spent a lot of time perusing the early manuscript, and provided me with much appreciated advice. James helped me to become skillful with many software tools. Andy read most of the thesis, and his feedback greatly improved the writing of this document.

I would also like to extend my gratitude to Andy Boughton, Yuli Zhou, Paul Johnson, Alex Caro, Shail Aditya, Joanna Kulik, Jan-Willem Maessen, Chris Joerg, Satyan Coorg, Yonald Chery, Richard Davis, Kyoo-Chan Cho and all the other CSG members. Their friendship makes CSG a wonderful place to work. Their talents created a challenging research environment in which I am constantly learning.

I can never forget the people who have given me precious help throughout the past years. Among them, I am truly grateful to Dr. Y. S. Cheung and Dr. Chorkin Chan of the University of Hong Kong, and Prof. Peisu Xia of the Chinese Academy of Sciences. Without their generous assistance, I would not be where I am. I also want to acknowledge Prof. Guojie Li and Prof. Yulin Feng of the Chinese Academy of Sciences.

I am always indebted to my parents for their constant love, patience, understanding and support. I owe my deepest gratitude to them.
To my parents
## Contents

1 Background .......................... 13
   1.1 The *T-AVN* Project .................. 14
   1.2 Cache Coherence Problem ............... 15
       1.2.1 Snoopy Protocols .................. 16
       1.2.2 Directory-based Protocols .......... 18
   1.3 Memory Consistency Models ............. 22
   1.4 Related Work .......................... 24
       1.4.1 The DASH System .................. 24
       1.4.2 The Alewife Machine ............... 25
   1.5 Thesis Organization ................... 26

2 Design Considerations and Implementation Issues 27
   2.1 System Abstraction .................... 27
       2.1.1 Memory Access Instructions ........ 28
       2.1.2 Cache Management Operations ....... 28
   2.2 Protocol Processor ..................... 29
       2.2.1 Directory States ................... 29
       2.2.2 Protocol Messages .................. 32
   2.3 Buffering Strategy ..................... 33
   2.4 Non-forwarding Vs. Forwarding .......... 35
       2.4.1 Non-forwarding Policy .............. 35
       2.4.2 Forwarding Policy .................. 36
       2.4.3 Aggressive Forwarding .............. 37
   2.5 Non-FIFO Order In Message Passing ....... 38
       2.5.1 Case Analysis I .................... 39
       2.5.2 Case Analysis II .................... 42
       2.5.3 Forwarding In FIFO Networks ........ 45

3 The Base Scheme: Specification and Verification 48
   3.1 State Transition Engines ............... 48
       3.1.1 Home Engine ....................... 49
       3.1.2 Site Engine ....................... 51
   3.2 Protocol Semantics Model ............... 54
       3.2.1 Uncached State .................... 55
       3.2.2 Shared State ....................... 55
3.2.3 Modified State .................................................. 56
3.2.4 Multicast-in-progress State .................................... 56
3.2.5 Forward-in-progress State ...................................... 57
3.2.6 Nak-impending State ............................................ 60

3.3 Message Processing Rules ........................................ 60
3.3.1 Home Engine Rules ............................................. 60
3.3.2 Site Engine Rules ............................................... 68

3.4 Protocol Correctness .............................................. 72
3.5 The Base+ Scheme .................................................. 73
3.5.1 Home Engine .................................................... 73
3.5.2 Site Engine ..................................................... 76

4 Incorporation Of I-Structures ....................................... 79
4.1 Introduction Of I-structures ..................................... 79
4.2 Motivation .......................................................... 81
4.3 Deferred List Management ........................................ 81
4.3.1 Maintaining Deferred Lists At Home ......................... 82
4.3.2 Maintaining Deferred Lists With Ownership ............... 83
4.3.3 Distributing Deferred Lists At Home / Modified Site .... 84
4.4 The ISC Scheme .................................................... 86
4.4.1 Protocol Messages ............................................... 86
4.4.2 Home Engine ................................................... 86
4.4.3 Site Engine ..................................................... 90

A Base Transition Rules ............................................... 97
A.1 Home Engine Rules ............................................... 97
A.2 Site Engine Rules ................................................. 102
# List of Figures

1.1 *T-N* Site Diagram ................................................. 14
1.2 Cache Coherence Problem ........................................... 15
1.3 Invalidation Vs. Update ........................................... 16
1.4 MESI State Transition Diagram ................................... 17
1.5 Full-map Directory Scheme ........................................ 19
1.6 Limited-Directory Scheme ......................................... 20
1.7 Chained-Directory Scheme ......................................... 21
1.8 Memory Access Conflict ........................................... 22

2.1 Abstract Site Diagram ............................................... 28
2.2 Home Directory Configuration ..................................... 30
2.3 Base Protocol Message Specification ............................. 33
2.4 Retrying Strategy .................................................. 34
2.5 Buffering Strategy .................................................. 35
2.6 Non-forwarding Policy ............................................. 36
2.7 Forwarding Policy .................................................. 37
2.8 Aggressive Forwarding ............................................. 38
2.9 Forwarded-Request Chain .......................................... 39
2.10 Scenario 1: Cache Line Replaced Before Invalidate-req Received .................................................. 40
2.11 Scenario 2: Load-ack Overtaken By Invalidate-req .............. 41
2.12 Scenario 3: Cache Line Replaced Before Fwd-load-req Received .................................................. 42
2.13 Scenario 4: Store-ack Overtaken By Fwd-load-req .............. 43
2.14 Ping-Pong Phenomenon ............................................. 44
2.15 Forwarding In FIFO Networks ...................................... 46

3.1 Base HEng State Diagram ........................................... 49
3.2 Base SEng State Diagram ........................................... 52
3.3 Protocol Semantics Model .......................................... 55
3.4 Base⁺ Protocol Message Specification ............................ 74
3.5 Base⁺ HEng State Diagram .......................................... 75
3.6 Base⁺ SEng State Diagram .......................................... 76

4.1 Producer and Consumer Problem ................................... 80
4.2 Multiple Slots In Cache Line ....................................... 80
4.3 Istore Performed Without Exclusive Ownership .................. 81
4.4 Maintaining Deferred Lists At Home ................................ 83
4.5 Maintaining Deferred Lists With Ownership ........................................ 84
4.6 Distributing Deferred Lists At Home / Modified Site ............................. 85
4.7 ISC Protocol Message Specification ..................................................... 87
Chapter 1

Background

Multiprocessor systems can be classified into two categories: message passing and shared memory. In message passing systems, processors access their own memories and communicate with other processors by explicitly sending and receiving messages through networks. Current message passing machines such as the SP2 system [4], however, have high overhead for user-level message passing. In shared memory systems, a global address space is provided and processors exchange information and synchronize one another by accessing shared variables. Although programming becomes much easier with a uniform address space, current shared memory machines like the DASH prototype [32] have been criticized for their hardware complexity to maintain cache coherence and lack of flexibility for various shared patterns of different applications.

Recent research experience suggests that multiprocessor systems should incorporate both fast user-level message passing and efficient shared memory assistance, without sacrificing favorable operating system functionalities. Message passing machines are starting to provide support for cache-coherent shared memory, a feature usually associated with shared memory systems. On the other hand, shared memory machines need to incorporate message passing mechanisms to handle user-level messages, a feature usually associated with message passing systems. The MIT *T-N* project[5, 6, 13], the Stanford FLASH system [26, 19, 20] and the MIT Alewife machine [1, 2, 24] are examples of such architecture convergence.
1.1 The *T-NG Project

*T-NG* is a PowerPC 620-based multiprocessor system supporting both user-level message passing and cache-coherent shared memory. Figure 1.1 shows a site configuration, where the white area represents a commercial 620-based SMP, and the grey area represents the additional hardware assistance for message passing and shared memory. Typically a site contains four network-endpoint-subsystems, each including a PowerPC 620, 4M bytes L2 cache and a network interface unit. The 620 is an aggressive 64-bit 4-way superscalar microprocessor with separate on-chip instruction and data caches, and a dedicated 128-bit L2 path to an external L2 cache [21]. A four-state MESI snoopy protocol is incorporated to maintain the cache coherence of multiple processors sitting on the same bus. The MIT prototype system will have eight sites connected by a Fat-Tree network [28].

Fine-grain user-level message passing is supported with network interface units (NIUs), which are tightly coupled to the L2 cache interfaces. The NIU buffers can be memory-mapped to an application address space, allowing user programs to send and receive messages without kernel intervention. Through NIUs the 620s may communicate di-
Cache-coherent shared memory is realized in software with very limited hardware assistance. The address capture device (ACD) allows a 620 to serve as a protocol processor to emulate a shared memory for user programs running on the other 620s within the site. The ACD has two address buffers for read and write operations respectively, and a data buffer for replied cache lines. A flush buffer is also needed, since the protocol processor must be able to issue flush operations in order to invalidate stale cache lines. The ACD snoops the L3 bus, capturing requests to global addresses and submitting the captured requests to the protocol processor. For any captured transaction, the protocol processor services the request and, if necessary, supplies the requested information to the requesting 620 through the ACD.

1.2 Cache Coherence Problem

In a shared-memory multiprocessor system, caches are often used to reduce the average memory accessing time. Many parallel programs exhibit significant spatial and temporal locality, allowing caches to satisfy almost all memory requests. The existence of multiple copies of the same memory block, however, gives rise to the cache coherence problem. Although read-only data can always be safely replicated, a store to a shared variable makes copies of the same location in other caches incoherent, that is, the cached
copies of the same variable are not identical.

An example is shown in Figure 1.2. Initially processors $A$, $B$ and $D$ all have cached location $X$ whose value is 100. Then processor $A$ writes $X$ with the value 200. The copies of $X$ in processors $B$ and $D$ become obsolete, which may result in errors if the stale values are later referenced. The primary objective of cache coherence mechanisms is to provide a coherent memory image for all processors and other devices in the system.

Basically there are two strategies to maintain cache coherence: invalidation and update. An invalidation-based protocol usually requires a processor to claim exclusive ownership of the shared location before performing a write. After the write is completed, the old copies in processors $B$ and $D$ have been invalidated, and processor $A$ becomes the only processor with copy $X$ (see Figure 1.3 a). With an update-based protocol, however, the write not only stores the new value to processor $A$’s copy, but updates the corresponding cached copies in processors $B$ and $D$ also (see Figure 1.3 b). Both invalidation and update strategies ensure that any read request will be supplied with the most up-to-date value, regardless of where the valid copy resides.

### 1.2.1 Snoopy Protocols

In a bus-based multiprocessor system, any ongoing bus transaction can be observed by all the processors with reasonable hardware support. Appropriate actions can therefore be
taken whenever an operation threatens coherence. Protocols that fall into this category are named snoopy protocols in that each cache "snoops" bus transactions to watch other processors' memory accesses. A second cache port is often provided so that snooping does not interfere with instruction execution unless state changes or data pushouts take place.

A snoopy mechanism maintains cache states, and implements coherence by coordinating activities among the caches, memory and bus interface logic. Generally speaking, when a device reads a location not in its cache, it broadcasts that read on the snoopy bus. Memory and other caches snoop the bus, comparing the referenced address with their own addresses. The device that has the most up-to-date copy will then supply the data. If the accessed location is shared by several devices, then one of them will be responsible for providing the data. Likewise, when a device broadcasts its intention to write a location which it does not own exclusively, other devices need to either invalidate or update their copies. With slightly different states and bus operations, many snoopy protocols have been proposed [18, 35, 22, 7].

As an example, Figure 1.4 presents the state transition diagram of the MESI snoopy protocol adopted in the PowerPC architecture [33]. MESI has four possible states for
each cache line: Invalid, Shared, Exclusive and Modified.

- Invalid: The addressed location is not resident in the cache.
- Shared: The addressed location is resident in the cache and coherent with memory. The location is in at least one other cache.
- Exclusive: The addressed location is only in this cache and coherent with memory.
- Modified: The addressed location is only in this cache and has been modified with respect to memory.

For a read miss, the data is supplied by memory when no modified copy exists, or by the cache that has the modified copy (in this case the data should also be written back to the memory). For a write instruction, if the cache has an Exclusive or Modified copy, the write is performed locally and the new state becomes Modified. Otherwise before the write operation is performed, all the copies of the location in the memory and other caches need to be invalidated and, if necessary, the valid data should be provided by either the memory or the cache that has the Modified copy. As for cache line replacement, a Shared or Exclusive copy can be replaced without extra action, while a Modified copy must be written back to the memory.

1.2.2 Directory-based Protocols

Directory-based protocols maintain a record for each memory block to specify the processors that currently have copies of the block. In NUMA systems where shared memories are physically distributed, directories are normally distributed also. For any global address, there is a home site where the corresponding physical memory resides. Each home site maintains the directory entries for its own shared memory. A directory entry usually records information including which processors have cached the block and which site currently holds the most up-to-date copy.

Unlike bus-based snoopy protocols, directory-based protocols do not rely upon broadcast to invalidate or update stale copies, because the locations of shared copies are known.
Coherence is maintained by sending individual point-to-point protocol messages to only the processors that have cached the accessed memory block. The elimination of broadcast overcomes the major limitation on scaling cache coherent machines to multiprocessors with a large number of processors.

The directory organization affects not only protocol complexity but also protocol performance. With appropriate hardware support, directories can be implemented using different structures [3, 10, 34, 38]. The full-map directory [29] and chained directory [37] are the most well-known organizations. Other approaches are also possible [11, 36], especially when the protocol is partially implemented in software.

**Full-map directory** In a full-map directory scheme, each directory entry keeps a complete record of which processors are sharing the block. The most common implementation has each directory entry contain one bit per processor representing if that processor has a shared copy. A dirty bit is often needed to specify if the block has been modified. When the dirty bit is set, the block is only cached by the processor which currently has write permission.

In Figure 1.5 (a), for example, the directory shows that block X is shared by processors A, B and D, and memory also has a valid copy. In Figure 1.5 (b) the copies
in processors $B$ and $D$ have been invalidated due to a write performed by processor $A$. Among the processor bits, only processor $A$’s bit is set, showing that processor $A$ currently has the modified copy, and the memory copy has become incoherent.

Since the memory size grows proportionally with the number of processors, a major scalability concern for full-map schemes is that the memory required for directory is $O(N^2)$, where $N$ is the number of processors. This overhead becomes intolerable for large systems: a 128-node multiprocessor consumes more than 50 percent memory just for the home directory, assuming the memory block size is 32 bytes.

**Limited directory** By restricting the number of processors allowed to be recorded in directories, limited-directory scheme reduces the directory size to $O(iN \log N)$, where $i$ is the number of processor pointers per entry, normally between 4 to 8. A problem for limited-directory schemes is dealing with the situation where a memory block needs to be shared by more than $i$ processors.

In Figure 1.6, for example, a problem occurs when processors $A$, $B$ and $D$ read location $X$ at the same time, since there are only two processor pointers in an entry. One solution is to intentionally invalidate a copy to make room for the new request. Another method is to use an extra bit indicating if the number of shared copies exceeds $i$. In
Figure 1.7: Chained-Directory Scheme

This case broadcast is used to invalidate or update shared copies if necessary.

**Chained directory** Chained-directory schemes keep track of shared copies by maintaining a chain of processor pointers. It has a nickname "gossip protocol" in that protocol messages are passed from one processor to another rather than being spread through multicast. The advantage is that the required directory size reduces to $O(N \log N)$, while a block is still allowed to be shared by arbitrary number of processors.

Potential cache line replacement complicates chained-directory schemes. In Figure 1.7 (a), suppose the copy of location $X$ in processor $B$ has to be evicted. To splice $B$ from the chain, the pointer in processor $A$ must be redirected to processor $D$. This can be done by either maintaining a circular chain and passing a message down from
processor \( B \), or by maintaining a doubly linked chain and sending a message up from processor \( B \) to \( A \). The “repaired” chain is displayed in Figure 1.7 (b). A simpler solution is to invalidate all the copies along the chain starting from processor \( B \). Figure 1.7 (c) shows the result of using this method.

### 1.3 Memory Consistency Models

A memory consistency model is a formal description of memory behavior in a shared-memory multiprocessor system. It specifies a “contract” between software and hardware, defining legal orders for multiple concurrent accesses to shared variables. From the point of view of implementation, the memory consistency model should avoid unnecessary restrictions to allow aggressive architecture and compiler optimizations. On the other hand, it should provide a reasonable paradigm for programmers to write and debug parallel programs.

The memory model assumed in uniprocessors is quite intuitive. A load returns the last value that was written to the given location, and a store binds the value for subsequent loads until the next store to the same location is performed. For multiprocessor systems, however, the concepts such as “last value written”, “subsequent load” and “next store” are ambiguous. The situation becomes far more confusing when optimizing techniques, for example, instruction reordering, write buffers, split-phased transactions and so on, are adopted.
The fundamental issue for any memory consistency model is to specify for each load which store operation binds the value that should be returned. The complexity of memory consistency models mainly comes from the interleaving of memory accesses. Two accesses are said to conflict if they are to the same memory location and at least one access is a store. Unless implicit data dependency or explicit barrier exists, conflicting accesses may lead to non-deterministic results.

In the example illustrated in Figure 1.8, X is a shared variable which is written by processors A and B, and read by processors C and D. Obviously the possible output for processor C is (⊥, ⊥), (⊥, 1), (⊥, 2), or (1, 2), where ⊥ representing the undefined value. And so is the possible output of processor D. An interesting question is if it is correct that processor C outputs (1, 2) while processor D outputs (2, 1)?

The simplest memory consistency model is sequential consistency [27], in which all writes to a memory location are serialized in some order and are performed in that order with respect to any processor. In modern microprocessors, multiple reads and writes are allowed to be issued in parallel and completed out of order. To avoid unnecessary penalties imposed by sequential consistency, more relaxed memory models such as location consistency [16], weak consistency [15] and release consistency [17] can be implemented.

Sequential Consistency A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. In other words, there exists a total order of the operations, compatible with program order for each processor, such that the value returned by each load is always the value last stored to the location according to the total order.

Location Consistency For each memory location, location consistency specifies a partially ordered multiset (pomset), in which an element corresponds to either a write or synchronization operation. A write operation inserts a write element in the pomset of the corresponding location, while a synchronization operation inserts a synchronization element in the pomsets of all the locations affected by the synchronization. An ordering
between two write operations is created if they are performed on the same processor, while an ordering is created between a synchronization and a write operation if the processor that performs the write operation also participates in the synchronization. For each read operation, a value set is defined as the set of legal values that can be returned.

**Weak Consistency**  Weak consistency is based on the recognition that certain shared variables behave as synchronization variables that should be treated differently from ordinary variables. Shared writable variables are classified as two non-overlapping categories, ordinary variables and synchronization variables. An access to a synchronization variable is delayed until all the ordinary accesses proceeding in program order are completed. Ordinary accesses following an access to a synchronization variable in program order are delayed until the synchronization access is completed. Ordinary accesses between two synchronization accesses, however, can be performed in any order, provided that local consistency is guaranteed.

**Release Consistency**  Release consistency improves the weak consistency model by utilizing extra information about synchronization variables. A synchronization access is labeled as an *acquire* operation if it is used to gain access to a critical section, and is labeled as a *release* operation if it is used to make a critical section available. Before an ordinary load or store access is allowed to perform with respect to any other processor, all previous acquire accesses must be performed. Before a release access is allowed to perform with respect to any other processor, all previous ordinary load and store accesses must be performed. Special accesses conform to either sequential consistency or location consistency with respect to one another.

### 1.4 Related Work

#### 1.4.1 The DASH System

The Stanford DASH system [32, 30, 31] consists of a set of processing clusters connected by a mesh interconnection network. In the prototype machine, each cluster is a Silicon Graphics Power Station 4D/340, which contains 4 MIPS R3000 processors and R3010
floating-point coprocessors running at 33 MHz. The interconnection network in DASH is a pair of wormhole meshes, which are dedicated to transferring requests and replies respectively.

The DASH memory system can be logically split to four levels. The level closest to the processor is the processor cache. A request that cannot be serviced by the processor cache is sent to the local cluster level, which consists of all the cluster's processor caches. Intra-cluster cache coherence is maintained by a bus-based snoopy protocol. If the request cannot be serviced within the cluster, it is then sent to the directory home level, which contains the directory entry and memory content of the accessed address. Usually the home cluster can satisfy the request. However, if the directory entry is in the dirty state, the request is forwarded to the remote cluster level, where eventually the request will be serviced.

In DASH's coherence protocol, the NAK reply is used to represent exceptions. For example, a request forwarded to the dirty cluster may arrive at there to find that the cluster no longer owns the data. In this case, a NAK reply is sent to the original requesting site to inform the corresponding processor that the request should be retried later as a new request. To avoid the complexity due to non-FIFO message passing, the protocol acknowledges with a NAK reply for most of the out-of-order requests. Thus it is possible that a request is kept retrying and cannot be serviced forever, although the probability this happens is very low.

1.4.2 The Alewife Machine

The MIT Alewife machine [1, 12, 25] is a set of processing nodes connected in a mesh topology. Each node has a Sparcle processor running at 33 MHz, a floating-point coprocessor, 64K bytes of direct-mapped cache, 4M bytes of globally-shared memory, a cache controller and a network router. Each node also contains 4M bytes local memory which can be used to maintain coherence directory entries. The Sparcle processor supports context switching, user-level message handling and fine-grain synchronization.

The LimitLESS scheme implements a small set of hardware pointers for each directory entry. When there are not enough pointers to record the locations of all the cached copies
of a given memory block, the memory module interrupts its local processor. The processor then emulates a full-map directory for the corresponding memory block. The coherence protocol is simple and straightforward, since a BUSY signal, similar to the NAK reply of DASH, can be used when a request cannot be serviced.

Fine-grain producer-consumer style synchronization is supported through J- and L-structures. There are three synchronization operations: locking read, non-locking read, and synchronizing write. A locking read waits until an element becomes full before emptying it and returning the value. A non-locking read also waits until the element is full, but then returns the value without emptying the element. A synchronizing write stores the corresponding value to an empty element, sets the state to full, and releases all the deferred reads.

1.5 Thesis Organization

Chapter 2 defines an abstract shared-memory model, and discusses the protocol processor design including directory organizations and message classifications. The retrying and buffering strategies are examined regarding protocol messages that temporarily cannot be serviced. Typical scenarios are analyzed in the context of non-forwarding and forwarding, with and without the FIFO order of message passing. Chapter 3 gives the operational semantics of the Base coherence scheme, and the proof that the scheme implements sequential consistency and is free from deadlock, livelock and starvation. The Base+ scheme for non-FIFO networks is also introduced. Chapter 4 presents the ISC scheme which incorporates I-structures at the cache coherence level to support fine-grain synchronization. Different approaches maintaining deferred list are studied, and the advantages and disadvantages are compared.
Chapter 2

Design Considerations and Implementation Issues

In designing coherence protocols, there are several critical issues that may greatly affect the schemes’ complexity and performance. Through analyzing several typical scenarios, this chapter examines retrying and buffering strategies, compares non-forwarding and forwarding policies, and assesses FIFO and non-FIFO orders of protocol message passing. Although the discussion is in the context of an abstract shared-memory machine, the methodology applies to all the directory-based cache coherence systems.

2.1 System Abstraction

To avoid the specific details of the PowerPC architecture, we consider an abstract shared-memory multiprocessor system consisting of a number of sites connected by network. As shown in Figure 2.1, each site contains an application processor (AP), a protocol processor (PP), a site cache, a shared memory and a network interface unit (NIU). User applications are performed on the AP, which allows concurrent split-phased memory accesses. The AP may incorporate lower-level processor caches, provided that the site cache is a superset of the processor caches\(^1\). The PP is responsible for servicing memory accesses issued from the AP and maintaining the global cache coherence.

\(^1\)In \(\star T-\mathcal{G}\), the AP is a bus-based multiprocessor subsystem, in which coherence is maintained via a hardware-supported snoopy protocol. The PP only processes the global memory accesses that cannot be serviced within the AP.
2.1.1 Memory Access Instructions

The basic memory access instructions are load and store. A load instruction reads the most up-to-date value of a given location, while a store instruction writes a specific value to a given location.

2.1.2 Cache Management Operations

A cache line may need to be flushed due to associative conflict or coherence requirement. A modified cache line may also need to provide the modified copy to other caches or to the memory. The cache management operations, delete, kill and clean, are introduced for this purpose. They can be either explicit instructions issued by the AP, or protocol commands generated by the PP. In addition, the delete and kill operations can also be caused by cache line replacement.\(^2\)

A delete operation invalidates a shared cache line. A kill operation flushes a modified cache line, changing its state to invalid. A clean operation releases the exclusive ownership of a modified cache line, updating its state to shared. For kill and clean operations, the most up-to-date data is captured by the PP, which may write the data to the memory.

\(^2\)In \(*T-NG\), without L3 cache, the clean operation can arise due to an intervention, that is, a modified processor cache supplies a shared copy to another processor cache in the same site, thereby disclaiming the exclusive ownership.
or another site.

2.2 Protocol Processor

Logically there are two components in the PP, a site engine (SEng) and a home engine (HEng). The SEng maintains the cache lines' states in a site directory (SDir) and runs a site state transition engine to process transactions regarding the site cache. The HEng maintains the memory blocks' states in a home directory (HDir) and runs a home state transition engine to process protocol messages regarding the memory blocks whose home is this site.

When the SEng receives an access instruction, it checks the SDir to decide whether the request can be satisfied within the site. If the addressed cache line is not in an appropriate state, the SEng suspends the access instruction and sends a protocol request to the corresponding home. When this message arrives at the home, the home's HEng services the protocol request based on the HDir state, and eventually sends a protocol reply to the requesting site to supply the requested data and/or exclusive ownership. In the requesting site, the suspended access instruction can therefore be resumed when the protocol reply is received.

2.2.1 Directory States

For each memory block, the HDir maintains a directory entry including three fields: \textit{state}, \textit{location}, and \textit{scratch}. Figure 2.2 shows the logical HDir configuration. A buffer heap is also needed when the buffering strategy is adopted to store messages that temporarily cannot be processed. The state field specifies the current coherence status of the block like whether the block is cached, shared or modified. The knowledge maintained in the location and scratch fields depends on the state field. The location field specifies which sites have shared copies for a shared block, and which site has the exclusive ownership for a modified block. The scratch field can be used to record additional information for transient states such as suspended requests, multicast counters and inv-pending flags.
The SDir has similar organization, except that the location field is no longer necessary\textsuperscript{3}.

States can be classified into two non-overlapping categories: stable and transient. A stable state represents normal coherence status like uncached, shared and modified. A transient state, in contrast, usually describes an unstable situation between stable states. In the Base scheme, for example, to handle a store request for a shared block, the HEng multicasts invalidation requests to the sites that currently have the shared copies. This gives rise to a transient state, H-multicast-in-progress, during the period when the multicast is in progress. The memory block will stay at this state until all the invalidations have been acknowledged.

The Base scheme has three HDir stable states, H-uncached, H-shared and H-modified, and three HDir transient states. There are three SDir stable states, C-invalid, C-shared and C-modified, and one SDir transient state\textsuperscript{4}. The number of states is an important measurement of the protocol complexity. Analogous transient states should be combined, with minor difference maintained in the flexible scratch field.

\textsuperscript{3}In \(*T-N\), without L3 cache, the SDir only maintains directory entries for cache lines with transient state, since the states of stable cache lines are kept in processor caches managed by the MESI mechanism.

\textsuperscript{4}The exclusive state of the MESI protocol is eliminated. This is because a store instruction performed on an exclusive cache line generates no bus operation that can be captured by the ACD. The protocol processor therefore cannot decide whether an exclusive cache line has been modified. This observation indicates that some currently prevalent snoopy protocols need to be revised in order to efficiently support directory-based cache-coherence systems.
Home Directory States

- **H-uncached**: The memory block is not cached by any site. The memory has the most up-to-date data.

- **H-shared\([S]\)**: The memory block is shared by the sites specified in \(S\). The memory also has the valid data.

- **H-modified\([m]\)**: The memory block is exclusively cached at site \(m\), and has been modified at that site. The memory does not have the most up-to-date data.

- **H-multicast-in-progress\([S]\)\(\langle\text{suspend}\rangle\)**: The home engine has sent invalidation requests to the sites specified in \(S\), but has not received all the acknowledgements. A suspended load request is maintained in \(\langle\text{suspend}\rangle\).

- **H-forward-in-progress\([m]\)\(\langle\text{suspend}\rangle\)**: The home engine has sent a \textit{fwd-} message to site \(m\) requesting the modified data, but has not received the reply. A suspended load or store request is maintained in \(\langle\text{suspend}\rangle\).

- **H-nak-impending\([m]\)\(\langle\text{suspend}\rangle\)**: The home engine has sent a \textit{fwd-} message to site \(m\) requesting the modified data and exclusive ownership, and then received a notification because the cache line at site \(m\) has been killed or cleaned. A suspended store request is maintained in \(\langle\text{suspend}\rangle\).

Site Directory States

- **C-invalid**: The accessed data is not resident in the cache.

- **C-shared**: The accessed data is resident in the cache, and possibly also cached in other sites. The memory has the valid data.

- **C-modified**: The accessed data is exclusively resident in this cache, and has been modified. The memory does not have the most up-to-date data.

- **C-req-in-progress\(\langle\text{instr}\rangle\)**: The site engine has sent a load or store request to the home site, but has not received the reply. The on-going memory access is recorded in \(\langle\text{instr}\rangle\).
2.2.2 Protocol Messages

Protocol messages can be classified into three categories: requests, replies and notifications. Generally speaking, a protocol request either requires the data and/or exclusive ownership, or demands that a specific coherence operation like invalidation be performed. A protocol reply is the response to a request which can be either positive or negative. A protocol notification serves as a single-trip message that requires no reply or acknowledgement. In our terminology, we use the suffix -req for requests, -ack for positive replies, -nak, -neg and -nil for negative replies, and -inf for notifications.

Whether a protocol message needs to be replied may depend on the directory state at the time the message is processed, or the behavior of some other related protocol messages. Sometimes it is not clear whether a protocol message should be classified as a request or notification, especially with non-FIFO message passing. For example, when a modified cache line is replaced, the data pushed out usually is sent to the home, which may or may not need to be acknowledged. Acknowledgements can be introduced as coordination barriers in the state transition diagram to simplify the design and verification. However, excessive use of acknowledgements will inevitably bring about serious protocol overhead.

A protocol message in the Base scheme includes four fields: command, block, site and data. The command field specifies the message type such as load-req and store-req. The block field gives the accessed block address that can be used to index HDir entries. These two fields are compulsory. Depending on the specific message type, a protocol message can be extended with optional fields: the site field records the site that issued the protocol request, and the data field carries the value when a data is transferred. Appropriate encoding techniques can be employed to compress the message size\(^5\). In addition, a transaction identifier may also need to be included to distinguish requests issued from the same AP.

The message specification is given in Figure 2.3, where req-site represents the site that issues the original access request. Notation shr-site represents a site that has a copy for a

\(^5\text{In } T-K\text{AiG, the most } n \text{ significant bits of a global address can be used to determine the address' home, where } 2^n \text{ is the number of sites. Thus home-issued protocol requests do not need the site field.}
shared block, and *mod-site* represents the site that has the exclusive copy for a modified block. A detailed explanation of the protocol messages can be found in the next chapter.

### 2.3 Buffering Strategy

The HEng in the PP may not be able to process all the received protocol messages immediately upon their arrival. In Figure 2.4, for example, suppose initially site *B* has a shared copy of the memory block. At site *A*, the AP performs a *store* instruction which causes a write cache miss, the SEng thus sends a *store-req* to the home to request the data and exclusive ownership. After receiving the *store-req*, the home’s HEng temporarily suspends this request, and then sends an *invalidation-req* to site *B* to invalidate the shared copy. Before the invalidation has been acknowledged, however, a *load-req* from site *C* arrives at the home. This message cannot be processed until the suspended request has been serviced.

A naive solution is to have the home’s HEng send a negative reply to site *C* claiming that the *load-req* at the time cannot be serviced (see Figure 2.4). The accessed address needs to be either included in the *load-nak* or preserved at site *C* until the positive reply received, so that site *C*’s SEng may send the request again later. The retrying strategy
Figure 2.4: Retrying Strategy

significantly simplifies the coherence scheme, because various transient situations do not need to be distinguished. However, it may cause livelock or starvation since a protocol message might keep retrying and cannot be serviced forever, unless appropriate preventive mechanism like time-stamp is incorporated.

The buffering strategy provides another solution where protocol messages that cannot be serviced temporarily are buffered at the home site for later processing. The home's HEng is responsible to resume the processing of buffered messages at the appropriate time. In the above example, the load-req is buffered at the home, and resumed after the suspended store-req has been serviced (see Figure 2.5).

The Eager Wake-up buffer management ensures that a protocol message is buffered only when necessary and then resumed at the earliest possible time. Furthermore, a resumed message will never be buffered again. The simple algorithm is described as follows, suggesting that each HDir entry maintain a FIFO buffer queue:

- A protocol message is buffered if and only if (a) it is a protocol request, and (b) the accessed block currently is in a transient state.

- A buffered request is resumed if and only if (a) it is the earliest arriving message among all the buffered requests with respect to the accessed block, and (b) the accessed block is in a stable state.
Figure 2.5: Buffering Strategy

Appropriate data structures like dynamic heaps can be chosen to implement the dedicated buffer queues. The maximum number of buffers required per site is \((N - 1) \cdot M\), where \(N\) is the number of sites, and \(M\) is the maximum number of concurrent global accesses allowed by an AP.

2.4 Non-forwarding Vs. Forwarding

When a protocol request arrives at the accessed block’s home, if the memory block is exclusively owned by a modified site, the home’s HEng under most situations cannot process the request without the most up-to-date data. There are two approaches to deal with this problem: non-forwarding and forwarding, both requiring a specific protocol message be sent to the site that currently has the modified data. In our terminology, we use the prefix \(fwd-\) to represent this kind of messages, regardless of whether the non-forwarding or forwarding policy is adopted.

2.4.1 Non-forwarding Policy

The non-forwarding policy ensures that a requested shared/exclusive copy can only be obtained from the accessed block’s home. If the block’s HDir state is H-modified, the
Figure 2.6: Non-forwarding Policy

home's HEng suspends the protocol request and sends an appropriate fwd- message to the modified site to request the most up-to-date data and/or exclusive ownership. The suspended request will be resumed after the protocol reply to the fwd- request has been received.

Figure 2.6 gives an example where initially site A has the exclusive copy. For the load-req, the home's HEng sends a fwd-load-req to site A to clean the modified cache line in order to obtain a shared copy (see Figure 2.6 a). For the store-req, the home's HEng sends a fwd-store-req to site A to kill the modified cache line in order to obtain the exclusive ownership (see Figure 2.6 b). Upon the receipt of the fwd- request, site A's SEng will send the most up-to-date data and/or exclusive ownership to the home.

It is possible that the modified cache line at site A has already been cleaned or killed before the fwd- request arrives, since the AP can intentionally perform a cache management operation, and the cache line can be replaced due to associative conflict. In this case, site A's SEng may need to send a negative acknowledgement to the home claiming that the fwd request cannot be serviced. This also necessitates the introduction of two HDir transient states, H-forward-in-progress and H-nak-impending, which represent the possible HDir situations between the issuance of the fwd- request and the receipt of the corresponding fwd- reply.

2.4.2 Forwarding Policy
If the memory block is in the H-modified state, the forwarding policy allows the home’s HEng to forward the load/store request to the modified site so that the request can be serviced there. Forwarding only requires 3 message passing hops (but still 4 messages) to service a protocol request. However, it may complicate coherence schemes to some extent. A more detailed discussion can be found in Section 2.5.3.

As shown in Figure 2.7, when a load/store request from site $B$ arrives at the home, the home’s HEng issues a $fwd$- message to forward the request to site $A$, where the modified data currently resides. At site $A$, if the forwarded request is a load-req, the SEng cleans the modified cache line and sends the most up-to-date data to site $B$. If the forwarded request is a store-req, site $A$’s SEng kills the modified cache line and sends the most up-to-date data and exclusive ownership to site $B$. In both situations, site $A$’s SEng needs to send an appropriate $fwd$- acknowledgement to the home to inform the home’s HEng that the forwarded request has been serviced.

### 2.4.3 Aggressive Forwarding

In the above example, the forwarded store-req can be handled more aggressively with only 3 protocol messages (see Figure 2.8). After forwarding the store-req to site $A$, the HEng maintains the HDir state as H-modified but records site $B$ in the location field as the new modified site. At site $A$, if the forwarded request is successfully serviced,
the SEng sends no acknowledgement to the home. However, if the modified cache line at site A has been cleaned or killed before the \textit{fwd-store-req} arrives, site A’s SEng may need to send a negative reply to the home claiming that the forwarded \textit{store-req} cannot be serviced. In other words, the \textit{fwd-store-req} can be either a request or notification, depending on the SDir state when site A’s SEng processes the forwarded message.

However, the adoption of aggressive forwarding introduces additional complexity to the design of coherence schemes. Figure 2.9 illustrates the phenomenon that several forwarded requests are blocked waiting for the exclusive ownership. This is very possible because when a \textit{store-req} is suspended at a site, the home’s HEng has modified the HDir’s location field as if the forwarded request has already been serviced. Later when another \textit{store-req} arrives at the home, the HEng will forward the new request to the new modified site. The forwarded store requests must be tracked somehow to make sure that if the HEng receives a kill/clean notification, all the on-going forwarded requests can still be serviced.

### 2.5 Non-FIFO Order In Message Passing

Non-FIFO message passing allows the adoption of adaptive routing that usually provides multiple paths for a given message. At each routing step, an appropriate channel can be chosen among the eligible outgoing channels, taking into account the network traffic workload and the diagnostic knowledge. The benefit becomes more attractive when cut-
through [23] and virtual channel [14] techniques are used, which allow message flits to be contiguously spread out along many channels when the head flit is blocked.

However, non-FIFO message passing may significantly complicate the design of coherence schemes, partly because certain protocol messages that would not need replies with FIFO message passing may have to be acknowledged when messages are not always received in the same order as they are issued. The situation becomes far more complicated when I- and M-structures are incorporated at the cache coherence level. This section discusses two typical cases and presents the solutions adopted in the Base$^+$ scheme.

### 2.5.1 Case Analysis I

When an *invalidate-req* arrives, the referenced cache line can be in the C-req-in-progress state with a suspended *load* instruction. With non-FIFO message passing, there are two different scenarios that may result in this situation (See Figure 2.10 and Figure 2.11). Assume initially the accessed memory block is not cached by any site.
Figure 2.10: Scenario 1: Cache Line Replaced Before Invalidate-req Received

**Scenario 1**

- Site $A$ sends a $load-req$ to the home of the accessed memory block. After receiving this request, the home’s HEng sends a $load-ack$ to site $A$ to supply the data. The $load-ack$ arrives at site $A$, and site $A$’s SEng places the data in the cache.

- At site $A$, the cache line is replaced. Since the SDir state is C-shared, the SEng does not inform the home’s HEng of this replacement.

- A $store-req$ from site $B$ arrives at the home. Since the HDir shows that currently site $A$ has a shared copy, the home’s HEng sends an $invalidate-req$ to site $A$ in order to invalidate the cache line. The $store-req$ is suspended.

- At site $A$, the AP performs a $load$ instruction before the $invalidate-req$ arrives. The SEng suspends the $load$, and sends a $load-req$ to the home.

- The $invalidate-req$ arrives at site $A$.

**Scenario 2**

- Site $A$ sends a $load-req$ to the home of the accessed memory block. After receiving this request, the home’s HEng sends a $load-ack$ to site $A$ to supply the data.

- A $store-req$ from site $B$ arrives at the home. Since the HDir shows that site $A$ has a shared copy, the home’s HEng sends an $invalidate-req$ to site $A$ to invalidate the
The invalidate-req overtakes the load-ack, and arrives at site A.

In scenario 1, the invalidate-req cannot be buffered at site A, otherwise deadlock may occur since the home’s HEng will not service any other request before the invalidation has been acknowledged. For this situation, site A’s SEng immediately sends an invalidate-ack to the home, because the cache line that needs to be invalidated has already been replaced.

In scenario 2, site A’s SEng can temporarily buffer the invalidate-req until the load-ack arrives. The shared copy carried by the load-ack must be invalidated when it is received. However, site A’s SEng cannot distinguish the two different scenarios at the time the invalidate-req arrives.

A solution is to have the home’s HEng assign a transaction identifier (tid) to all the load-acks and invalidate-reqs when they are issued. The HEng maintains a counter with an initial value 0. The tid of a load-ack is the current counter value. For an invalidate-req, the HEng increases the counter value by 1, and uses the new counter value as the message’s tid. When a site receives an invalidate-req, if the SDir state is C-req-in-progress with a suspended load, the SEng records the tid of the invalidate-req, and then immediately sends an invalidate-ack to the home to acknowledge the invalidation. Later when a load-ack arrives, the SEng compares the recorded tid with the tid of the load-ack. If the two tid’s are equal, the SEng knows that the load-ack was issued after the invalidate-req, and thus places the data in the cache with the C-shared state. If the invalidate-req’s
Figure 2.12: Scenario 3: Cache Line Replaced Before Fwd-load-req Received

tid is larger, the SEng discards the stale data immediately since the load-ack was issued prior to the invalidate-req.

A simplified approach is to have the SDir include an inv-pending flag in the scratch field. When an invalidate-req arrives at a site, if the SDir entry is in the C-req-in-progress state with a suspended load, the SEng sets the entry’s inv-pending flag. Later when a load-ack is received, the SEng checks the inv-pending flag. If the flag is set, the data is discarded regardless of whether it is stale, and the flag is cleared. This conservative strategy ensures that any data that might threaten coherence will be invalidated, although a valid cache line sometimes can be falsely invalidated.

2.5.2 Case Analysis II

Another typical case arises when a fwd- request arrives at a site where the referenced cache line is in the C-req-in-progress state with a suspended store instruction. With non-FIFO message passing, there are two different scenarios that can lead to this situation (See Figure 2.12 and Figure 2.13). Assume initially the accessed memory block is not cached by any site.

Scenario 3

6In *T-NG, the data can be used to complete the suspended load instruction to avoid retrying the load-req without violating sequential consistency.
Figure 2.13: Scenario 4: Store-ack Overtaken By Fwd-load-req

- Site A sends a *store-req* to the home of the accessed memory block. After receiving this request, the home’s HEng sends a *store-ack* to site A to supply the requested data and exclusive ownership. The *store-ack* arrives at site A, and site A’s SEng places the data in the cache with the C-modified state.

- At site A, the cache line is replaced. Since the SDir state is C-modified, the SEng sends a *kill-inf* to the home to write the most up-to-date data to the memory.

- At the home, before the *kill-inf* arrives, a *load-req* from site B is received. Since the HDir shows that site A holds the modified copy, the HEng sends a *fwd-load-req* to site A requesting the most up-to-date data.

- At site A, the AP performs a *store* instruction before the *fwd-load-req* arrives. The SEng suspends the *store*, and sends a *store-req* to the home.

- The *fwd-load-req* arrives at site A.

**Scenario 4**

- Site A sends a *store-req* to the home of the accessed memory block. After receiving this request, the home’s HEng sends a *store-ack* to site A to supply the requested data and exclusive ownership.

- A *load-req* from site B arrives at the home. Since the HDir shows that the modified data resides at site A, the HEng sends a *fwd-load-req* to site A.
- The *fwd-load-req* overtake the *store-ack*, and arrives at site *A*.

In scenario 3, the *fwd-load-req* cannot be blocked at site *A*, otherwise deadlock will happen since the home cannot service any other request before the *fwd-load-req* is acknowledged. Under this circumstance, a negative acknowledgement can be immediately sent to the home, because the modified copy has already been pushed out. In scenario 4, however, the *fwd-load-req* can be buffered at site *A* to wait for the *store-ack*, which is still on the way to site *A*. Unfortunately, the two different scenarios cannot be distinguished by site *A*'s SEng when the *fwd-load-req* arrives.

A simple solution is to use different negative acknowledgements for *fwd-* requests that cannot be serviced so as to provide more information to the home’s HEng. When a *fwd*-request cannot be processed, if the SDir clearly shows that the modified cache line has been written back to the memory, the site’s SEng sends an appropriate -nak reply to the home. If the SDir cannot decide whether the ownership has been released or has not been received yet, the SEng sends an appropriate -neg reply to the home. At the home, if a -nak acknowledgement arrives, the HEng knows that the most up-to-date data, if still not received, will definitely arrive later. For a -neg acknowledge, the home’s HEng checks if the most up-to-date data has arrived already. If not, the HEng retries the original *fwd*-request since at this time the HEng cannot tell whether the modified cache line will be
written back to the home.

Figure 2.14 presents a ping-pong phenomena that may appear when the message carrying the ownership takes a considerably longer latency. However, it should be noted that the freedom from deadlock, livelock and starvation is always guaranteed.

### 2.5.3 Forwarding In FIFO Networks

The adoption of forwarding allows a site to receive a protocol reply including the requested data/ownership from either the home or the modified site. Figure 2.15 (a) shows a scenario that an `invalidate-req` arrives at a site before the `load-ack` carrying the shared data is received, assuming site A at the beginning has the modified cache line.

- Site B sends a `load-req` to the home. After receiving this request, the home’s HEng sends a `fwd-load-req` to forward the `load-req` to site A, since the HDir shows that site A currently has the most up-to-date data. The HEng then changes the HDir state to H-forward-in-progress.

- At site A, after receiving the `fwd-load-req`, the SEng cleans the modified cache line, sends a `load-ack` to site B to provide the most up-to-date data, and sends a `fwd-load-ack` to the home to inform the home’s HEng that the forwarded request has been serviced.

- At the home, after receiving the `fwd-load-ack`, the HEng writes the most up-to-date data to the memory, and changes the HDir state to C-shared with sites A and B recorded in the location field.

- At the home, a `store-req` from site C is received. Since the HDir shows that sites A and B are sharing the memory block, the HEng sends an `invalidate-req` to both sites A and B to invalidate the shared copies.

- At site B, before the `load-ack` from site A arrives, the SEng receives the `invalidate-req` from the home.
Figure 2.15: Forwarding In FIFO Networks

Figure 2.15 (b) shows a scenario where a forwarded request arrives at a site before the store-ack transferring the exclusive ownership is received, assuming site A initially has the modified cache line.

- Site B sends a store-req to the home. After receiving this request, the home’s HEng sends a fwd-store-req to forward the store-req to site A, where the HDir shows the modified cache line resides. The HEng then changes the HDir state to H-forward-in-progress.

- At site A, after receiving the fwd-store-req, the SEng kills the modified cache line, sends a store-ack to site B to supply the most up-to-date data and exclusive ownership, and sends a fwd-store-ack to the home to acknowledge that the forwarded request has been serviced.

- At the home, after receiving the fwd-store-ack, the HEng changes the HDir state to H-modified with site B recorded in the location field.

- At the home, a load-req from site C is received. Since the HDir shows that site A holds the exclusive ownership of the accessed memory block, the HEng sends a fwd-load-req to forward the load-req to site B.
• At site $B$, before the *store-ack* from site $A$ arrives, the SEng receives the *fwd-load-req* from the home.

With FIFO message passing, the above two scenarios can still take place. This suggests that non-forwarding should be used in FIFO networks, since the adoption of the forwarding policy would introduce the similar complication discussed in Section 2.5.1 and 2.5.2. On the other hand, the forwarding policy can be used for non-FIFO networks where the additional complexity caused by forwarding is almost negligible.
Chapter 3

The Base Scheme: Specification and Verification

This chapter presents the Base scheme, a representative cache coherence protocol implementing sequential consistency, and proves its logical correctness. In the Base scheme, protocol messages that temporarily cannot be serviced are buffered for later processing. FIFO message passing is assumed and the non-forwarding policy is adopted. The Base+ scheme designed for non-FIFO networks is also introduced.

3.1 State Transition Engines

At any site, incoming protocol messages for the HEng are maintained in an *input pool* before being processed\(^1\). The HEng processes the received messages according to the state transition rules, while maintaining a buffer queue for each memory block to buffer protocol messages that cannot be processed temporarily. The Eager Wake-up management implies that the next message the HEng processes is chosen based on the block’s new HDir state and the current buffer queue status. In other words, if the block is in a stable state and the buffer queue is not empty, the HEng will begin processing the first buffered message. Otherwise the HEng accepts a protocol message from the input pool. This strategy guarantees that all buffered message will be resumed at the earliest possible time.

---

\(^1\)In \(\ast T\-N\mathcal{Q}\), the NIU’s incoming channel buffers or a dedicated memory area can serve as the input pool.
3.1.1 Home Engine

Figure 3.1 gives the HEng state transition diagram. The HEng processes a protocol message received from site $k$ as follows.

**load-req**

- If the accessed block’s HDir state is H-uncached, the HEng sends a load-ack to site $k$ to supply the requested data. The HDir state is changed to H-shared[$S$], where $S = \{k\}$.

- If the accessed block’s HDir state is H-shared[$S$], the HEng sends a load-ack to site $k$ to supply the requested data. The HDir state is changed to H-shared[$S'$], where $S' = S \cup \{k\}$.

Figure 3.1: Base HEng State Diagram
• If the accessed block’s HDir state is H-modified[\(m\)], the HEng suspends the load-req, and sends a \(\textit{fwd-load-req}\) to site \(m\) to request the most up-to-date data. The HDir state is changed to H-forward-in-progress[\(m\)]. Later when the corresponding \(\textit{fwd-load-ack}\) arrives at the home, the HEng resumes the suspended load-req, updates the memory, and changes the HDir state to H-shared[\(S\)], where \(S = \{m, k\}\).

At site \(m\), however, before the \(\textit{fwd-load-req}\) is received, the modified cache line may have already been cleaned or killed. For this case, site \(m\)’s SEng takes no action for the \(\textit{fwd}\)-request. At the home, when the corresponding \(\textit{kill-inf}\) or \(\textit{clean-inf}\) arrives, the HEng resumes the suspended load-req, updates the memory with the most up-to-date data, and changes the HDir state to H-shared[\(S\)] where \(S = \{k\}\) or \(\{k, m\}\), depending on whether the modified cache line was killed or cleaned at site \(m\).

\textbf{store-req}

• If the accessed block’s HDir state is H-uncached, the HEng sends a \(\textit{store-ack}\) to site \(k\) to supply the requested data along with the exclusive ownership. The HDir state is changed to H-modified[\(k\)].

• If the accessed block’s HDir state is H-shared[\(S\)], the HEng suspends the store-req, and sends \(\textit{invalidate-reqs}\) to all the sites specified in \(S\). The HDir state is changed to H-multicast-in-progress, while in the scratch field a counter is maintained to record the number of invalidations that are being performed. Later when all the invalidations have been acknowledged, the HEng resumes the suspended store-req, and changes the HDir state to H-modified[\(k\)].

• If the accessed block’s HDir state is H-modified[\(m\)], the HEng suspends the store-req, and sends a \(\textit{fwd-store-req}\) to site \(k\) to request the most up-to-date data and exclusive ownership. The HDir state is changed to H-forward-in-progress. The suspended message is resumed later when the home’s HEng receives the corresponding \(\textit{fwd-store-ack}\), and the HDir state is then changed to H-modified[\(k\)].
If the modified cache line at site $m$ has been cleaned or killed before the \textit{fwd-store-req} arrives, site $m$’s SEng must send a \textit{fwd-store-nak} to the home. This is because, if the current SDir state is C-shared, the SEng needs to invalidate the cache line, and inform the home of the completion of this invalidation. Another transient HDir state, H-nak-impending is introduced to handle this negative \textit{fwd-} acknowledge-ment. When the home’s HEng receives a \textit{kill-inf} or \textit{clean-inf} for the block which is in the H-forward-in-progress state, it records the most up-to-date data, and changes the HDir state to H-nak-impending. After the corresponding \textit{fwd-store-nak} arrives, the home’s HEng resumes the suspended \textit{store-req} using the recorded data, and changes the HDir state to H-modified[$k$].

**kill-inf / clean-inf**

- If the accessed block is in the H-modified state, the HEng updates the memory with the most up-to-date data. The HDir state is changed to either H-uncached or H-shared[$S$] where $S = \{m\}$, depending on whether the modified cache line was killed or cleaned.

### 3.1.2 Site Engine

Figure 3.2 gives the SEng state transition diagram. The SEng processes memory access instructions, cache management operations and protocol messages as follows.

**load**

- If the accessed cache line’s SDir state is C-shared or C-modified, the SEng supplies the data from the cache. The SDir state is not changed.

- If the accessed cache line’s SDir state is C-invalid, the SEng suspends the \textit{load} instruction, and sends a \textit{load-req} to the accessed address’ home to request the data. The SDir state is changed to C-req-in-progress. Later when the corresponding \textit{load-ack} arrives, the SEng places the data in the cache with the C-shared state, and resumes the suspended \textit{load} instruction.
store

- If the accessed cache line's SDir state is C-modified, the SEng performs the store instruction on the cache. The SDir state is not changed.

- If the accessed cache line's SDir state is C-invalid or C-shared, the SEng suspends the store instruction, and sends a store-req to the accessed address' home to request the data and exclusive ownership\(^2\). The SDir state is changed to C-req-in-progress. Later when the corresponding store-ack arrives, the SEng places the data in the cache with the C-modified state, and resumes the suspended store instruction.

delete / kill / clean

- delete: The SEng simply changes the SDir state from C-shared to C-invalid.

---

\(^2\)In \(*T-NG\), the store instruction corresponds to the PowerPC's RWITM (read-with-intent-to-modify) operation. A possible improvement is to incorporate the DClaim (data-claim) instruction to request just the ownership for a write operation performed on a shared cache line.
• kill: The SEng changes the SDir state to C-invalid, and sends a *kill-inf* to the home to write the modified data to the memory.

• clean: The SEng changes the SDir state to C-shared, and sends a *clean-inf* to the home to write the modified data to the memory.

**invalidate-req**

• If the SDir state of the referenced cache line is C-invalid or C-req-in-progress, the SEng sends an *invalidate-ack* to the home to acknowledge the invalidation. The SDir state is not changed.

• If the SDir state of the referenced cache line is C-shared, the SEng changes the SDir state to C-invalid, and sends an *invalidate-ack* to the home to notify the home’s HEng that the invalidation has been performed.

**fwd-load-req**

• If the SDir state of the referenced cache line is C-modified, the SEng changes the SDir state to C-shared, and sends a *fwd-load-ack* to the home to write the modified data to the memory.

• If the SDir state is C-invalid, C-shared or C-req-in-progress, no SEng action is taken and the SDir state remains unchanged.

**fwd-store-req**

• If the SDir state of the referenced cache line is C-modified, the SEng changes the SDir state to C-invalid, and sends a *fwd-store-ack* to the home to transfer the modified data and exclusive ownership.

• If the SDir state is C-invalid, C-shared or C-req-in-progress, the SEng sends a *fwd-store-nak* to the home. The SDir state is not changed.
3.2 Protocol Semantics Model

To specify the operational semantics of the Base scheme, we define the protocol semantics model as shown in Figure 3.3, where the grey and white areas represent home and state engines respectively. For any given block, the memory configuration can be characterized by the tuple $\Sigma$, in which directory states and values of multiple copies are accurately described.

$$\Sigma = \{Hdir, Hval, W^{(0)}, W^{(1)}, \ldots, W^{(n-1)}\}$$

where

$$W^{(i)} = \{Sdir^{(i)}, Sval^{(i)}, Snet^{(i)}, Hnet^{(i)}\} \quad (0 \leq i \leq n - 1)$$

Hdir is the home directory state, and Hval is the memory value. In the Base scheme, the home directory has three stable states and three transient states. Sdir$^{(i)}$ represents the site directory state at site $i$, while Sval$^{(i)}$ maintains the cache value. The site directory has three stable states and one transient states. Throughout the following discussion, we use lowercase Greek letters $\alpha$, $\beta$ and $\gamma$ to represent values of memory blocks, cache lines, and store accesses. Undefined value is $\perp$. $\Phi$ is the empty message pool. Moreover, the notation $\lfloor$ is introduced so that the memory configuration can be written in a shorthanded way: $\Sigma = \{Hdir, Hval, \lfloor W^{(i)}\rfloor\}$.

Message passing mechanisms can be abstracted by message pools. A message pool is a sequence of protocol messages connected by operator $\ast$. The home pool Hnet$^{(i)}$ contains protocol messages which are sent from site $i$ to the home. The site pool Snet$^{(i)}$ consists of protocol messages which are sent from the home to site $i$. The non-forwarding policy adopted in the Base scheme allows no protocol message between site engines. A protocol message is added to the appropriate pool when produced, and removed from the pool after consumed. For both home and site pools, the order of messages is the order the messages are issued and, with FIFO message passing, is also the order the messages are received.

With respect to the protocol semantics model, the Base scheme has 6 types of legal memory configurations, which are defined as follows.
3.2.1 Uncached State

\[ \Sigma = \left\{ \text{H-uncached, } \alpha, \bigcup_{i} W^{(i)} \right\} \]  \hspace{1cm} (3.1)

where

\[ W^{(i)} \in \Lambda \]

\[ \Lambda = \{ \{ \text{C-invalid, } \perp, \Phi, \Phi \}, \]
\[ \{ \text{C-req-in-progress(load), } \perp, \Phi, \text{(load-req)} \}, \]
\[ \{ \text{C-req-in-progress(store } \beta \text{), } \perp, \Phi, \text{(store-req)} \} \} \]

It should be noted that \( \Lambda \) is the set of legal sub-configurations of \( W^{(i)} \) when the home directory shows that site \( i \) has no valid copy.

3.2.2 Shared State

\[ \Sigma = \left\{ \text{H-shared[S], } \alpha, \bigcup_{i} W^{(i)} \right\} \]  \hspace{1cm} (3.2)

where

\[ W^{(i)} \in \{ \{ \text{C-req-in-progress(load), } \perp, \text{(load-ack } \alpha \text{), } \Phi \}, \]
\[ \{ \text{C-shared, } \alpha, \Phi, \Phi \}, \]
\[ \{ \text{C-invalid, } \perp, \Phi, \Phi \}, \]
\[ \{ \text{C-req-in-progress(load), } \perp, \Phi, \text{(load-req)} \} \}, \]

55
When the home directory shows that site \( i \) has a shared copy, it may have no valid data. This is possible since a shared cache line can be deleted without informing the home site.

### 3.2.3 Modified State

\[
\Sigma = \left\{ H\text{-}modified[m], \perp, \bigcup_i W^{(i)} \right\} \quad (3.3)
\]

where

\[
W^{(m)} \in \{ \{ \text{C-req-in-progress}(\text{store } \beta), \perp, \{\text{store-ack } \alpha\}, \Phi \},
\{ \text{C-modified}, \alpha, \Phi, \Phi \},
\{ \text{C-invalid}, \perp, \Phi, \{\text{kill-inf } \alpha\} \},
\{ \text{C-shared}, \alpha, \Phi, \{\text{clean-inf } \alpha\} \},
\{ \text{C-invalid}, \perp, \Phi, \{\text{clean-inf } \alpha\} \},
\{ \text{C-req-in-progress}(\text{load}), \perp, \Phi, \{\text{kill-inf } \alpha\} \ast \{\text{load-req}\} \},
\{ \text{C-req-in-progress}(\text{store } \beta), \perp, \Phi, \{\text{kill-inf } \alpha\} \ast \{\text{store-req}\} \},
\{ \text{C-req-in-progress}(\text{load}), \perp, \Phi, \{\text{clean-inf } \alpha\} \ast \{\text{load-req}\} \},
\{ \text{C-req-in-progress}(\text{store } \beta), \perp, \Phi, \{\text{clean-inf } \alpha\} \ast \{\text{store-req}\} \} \}
\]

\[
W^{(i)} \in \Lambda \quad \text{if } i \neq m
\]

While the home directory shows that site \( m \) has the exclusive ownership, the modified cache line may have already been killed or cleaned at site \( m \). In this case, \( H\text{net}^{(m)} \) contains the protocol notification which writes the most up-to-date data to the memory.

### 3.2.4 Multicast-in-progress State

\[
\Sigma = \left\{ H\text{-}multicast-in-progress[S](\text{store-req } j), \alpha, \bigcup_i W^{(i)} \right\} \quad (3.4)
\]

56
where

\[ W^{(i)} \in \{ \{C-\text{req-in-progress}(\text{load}), \bot, \langle \text{load-ack} \alpha \rangle \ast \langle \text{invalidate-req} \rangle, \Phi \}, \]
\[ \{C-\text{shared}, \alpha, \langle \text{invalidate-req} \rangle, \Phi \}, \]
\[ \{C-\text{invalid}, \bot, \langle \text{invalidate-req} \rangle, \Phi \}, \]
\[ \{C-\text{req-in-progress}(\text{load}), \bot, \langle \text{invalidate-req} \rangle, \langle \text{load-req} \rangle \}, \]
\[ \{C-\text{req-in-progress}(\text{store } \beta), \bot, \langle \text{invalidate-req} \rangle, \langle \text{store-req} \rangle \}, \]
\[ \{C-\text{invalid}, \bot, \Phi, \langle \text{invalidate-ack} \rangle \}, \]
\[ \{C-\text{req-in-progress}(\text{load}), \bot, \Phi, \langle \text{invalidate-ack} \rangle \ast \langle \text{load-req} \rangle \}, \]
\[ \{C-\text{req-in-progress}(\text{store } \beta), \bot, \Phi, \langle \text{invalidate-ack} \rangle \ast \langle \text{store-req} \rangle \}, \]
\[ \{C-\text{req-in-progress}(\text{load}), \bot, \Phi, \langle \text{load-req} \rangle \ast \langle \text{invalidate-ack} \rangle \}, \]
\[ \{C-\text{req-in-progress}(\text{store } \beta), \bot, \Phi, \langle \text{store-req} \rangle \ast \langle \text{invalidate-ack} \rangle \} \} \quad \text{if } i \in S - \{j\} \]

\[ W^{(i)} \in \Lambda \quad \text{if } i \notin S \cup \{j\} \]

\[ W^{(j)} = \{C-\text{req-in-progress}(\text{store } \beta), \bot, \Phi, \Phi \} \]

An invalidation request can always be positively acknowledged, regardless of the current site state. With FIFO message passing, if the site is in the C-req-in-progress state when an invalidation request arrives, it means that the shared copy has already been flushed.

### 3.2.5 Forward-in-progress State

When the home directory state is H-forward-in-progress, the memory configuration can be classified into two categories, depending on whether the suspended message is a load or store request.

\[ \Sigma = \left\{ \text{H-forward-in-progress}[m](\text{load-req } j), \bot, \bigcup_i W^{(i)} \right\} \quad (3.5) \]

where

\[ W^{(m)} \in \{ \{C-\text{req-in-progress}(\text{store } \beta), \bot, \langle \text{store-ack} \alpha \rangle \ast \langle \text{fwd-load-req} \rangle, \Phi \}, \]
\[ \{C-\text{modified}, \alpha, \langle \text{fwd-load-req} \rangle, \Phi \}, \]
\[ \{C-\text{invalid}, \bot, \langle \text{fwd-load-req} \rangle, \langle \text{kill-inf } \alpha \rangle \} \}, \]
\{C-shared, α, \langle fwd-load-req \rangle, \langle clean-inf α \rangle \},
\{C-invalid, \perp, \langle fwd-load-req \rangle, \langle clean-inf α \rangle \},
\{C-req-in-progress(load), \perp, \langle fwd-load-req \rangle, \langle kill-inf α \rangle * \langle load-req \rangle \},
\{C-req-in-progress(store β), \perp, \langle fwd-load-req \rangle, \langle kill-inf α \rangle * \langle store-req \rangle \},
\{C-req-in-progress(load), \perp, \langle fwd-load-req \rangle, \langle clean-inf α \rangle * \langle load-req \rangle \},
\{C-req-in-progress(store β), \perp, \langle fwd-load-req \rangle, \langle clean-inf α \rangle * \langle store-req \rangle \},
\{C-shared, α, \Phi, \langle fwd-load-ack α \rangle \},
\{C-invalid, \perp, \Phi, \langle fwd-load-ack α \rangle \},
\{C-req-in-progress(load), \perp, \Phi, \langle fwd-load-ack α \rangle * \langle load-req \rangle \},
\{C-req-in-progress(store β), \perp, \Phi, \langle fwd-load-ack α \rangle * \langle store-req \rangle \},
\{C-invalid, \perp, \Phi, \langle kill-inf α \rangle \},
\{C-shared, α, \Phi, \langle clean-inf α \rangle \},
\{C-invalid, \perp, \Phi, \langle clean-inf α \rangle \},
\{C-req-in-progress(load), \perp, \Phi, \langle kill-inf α \rangle * \langle load-req \rangle \},
\{C-req-in-progress(store β), \perp, \Phi, \langle kill-inf α \rangle * \langle store-req \rangle \},
\{C-req-in-progress(load), \perp, \Phi, \langle clean-inf α \rangle * \langle load-req \rangle \},
\{C-req-in-progress(store β), \perp, \Phi, \langle clean-inf α \rangle * \langle store-req \rangle \}

W^{(j)} = \{C-req-in-progress(load), \perp, \Phi, \Phi \}
W^{(i)} \in \Lambda \text{ if } i \neq m, j

A \textit{fwd}-message requesting the modified data can always be serviced immediately. With FIFO message passing, if the site has no exclusive ownership when a \textit{fwd-load-req} arrives, the \textit{fwd}-message can be safely ignored because the requested data has already been sent to the home.

\[ \Sigma = \left\{ \text{H-forward-in-progress}[m](store-req \_j), \perp, \bigcup_i W^{(i)} \right\} \] (3.6)

where

\[ W^{(m)} \in \{ \{C-req-in-progress(store β), \perp, \langle store-ack α \rangle * \langle fwd-store-req \rangle, \Phi \}, \{C-modified, α, \langle fwd-store-req \rangle, \Phi \} \}, \]
\{C-\text{invalid}, \bot, \langle \text{fwd-store-req} \rangle, \langle \text{kill-inf } \alpha \rangle \},
\{C-\text{shared}, \alpha, \langle \text{fwd-store-req} \rangle, \langle \text{clean-inf } \alpha \rangle \},
\{C-\text{invalid}, \bot, \langle \text{fwd-store-req} \rangle, \langle \text{clean-inf } \alpha \rangle \},
\{C-\text{req-in-progress}(\text{load}), \bot, \langle \text{fwd-store-req} \rangle, \langle \text{kill-inf } \alpha \rangle \ast \langle \text{load-req} \rangle \},
\{C-\text{req-in-progress}(\text{store } \beta), \bot, \langle \text{fwd-store-req} \rangle, \langle \text{kill-inf } \alpha \rangle \ast \langle \text{store-req} \rangle \},
\{C-\text{req-in-progress}(\text{load}), \bot, \langle \text{fwd-store-req} \rangle, \langle \text{clean-inf } \alpha \rangle \ast \langle \text{load-req} \rangle \},
\{C-\text{req-in-progress}(\text{store } \beta), \bot, \langle \text{fwd-store-req} \rangle, \langle \text{clean-inf } \alpha \rangle \ast \langle \text{store-req} \rangle \},
\{C-\text{invalid}, \bot, \Phi, \langle \text{fwd-store-ack } \alpha \rangle \},
\{C-\text{req-in-progress}(\text{load}), \bot, \Phi, \langle \text{fwd-store-ack } \alpha \rangle \ast \langle \text{load-req} \rangle \},
\{C-\text{req-in-progress}(\text{store } \beta), \bot, \Phi, \langle \text{fwd-store-ack } \alpha \rangle \ast \langle \text{store-req} \rangle \},
\{C-\text{invalid}, \bot, \Phi, \langle \text{kill-inf } \alpha \rangle \ast \langle \text{fwd-store-nak} \rangle \},
\{C-\text{invalid}, \bot, \Phi, \langle \text{clean-inf } \alpha \rangle \ast \langle \text{fwd-store-nak} \rangle \},
\{C-\text{req-in-progress}(\text{load}), \bot, \Phi, \langle \text{kill-inf } \alpha \rangle \ast \langle \text{fwd-store-nak} \rangle \ast \langle \text{load-req} \rangle \},
\{C-\text{req-in-progress}(\text{store } \beta), \bot, \Phi, \langle \text{kill-inf } \alpha \rangle \ast \langle \text{fwd-store-nak} \rangle \ast \langle \text{store-req} \rangle \},
\{C-\text{req-in-progress}(\text{load}), \bot, \Phi, \langle \text{kill-inf } \alpha \rangle \ast \langle \text{load-req} \rangle \ast \langle \text{fwd-store-nak} \rangle \},
\{C-\text{req-in-progress}(\text{store } \beta), \bot, \Phi, \langle \text{kill-inf } \alpha \rangle \ast \langle \text{store-req} \rangle \ast \langle \text{fwd-store-nak} \rangle \},
\{C-\text{req-in-progress}(\text{load}), \bot, \Phi, \langle \text{clean-inf } \alpha \rangle \ast \langle \text{fwd-store-nak} \rangle \ast \langle \text{load-req} \rangle \},
\{C-\text{req-in-progress}(\text{store } \beta), \bot, \Phi, \langle \text{clean-inf } \alpha \rangle \ast \langle \text{fwd-store-nak} \rangle \ast \langle \text{store-req} \rangle \},
\{C-\text{req-in-progress}(\text{load}), \bot, \Phi, \langle \text{clean-inf } \alpha \rangle \ast \langle \text{load-req} \rangle \ast \langle \text{fwd-store-nak} \rangle \},
\{C-\text{req-in-progress}(\text{store } \beta), \bot, \Phi, \langle \text{clean-inf } \alpha \rangle \ast \langle \text{store-req} \rangle \ast \langle \text{fwd-store-nak} \rangle \})

W^{(j)} = \{C-\text{req-in-progress}(\text{store } \gamma), \bot, \Phi, \Phi \}

W^{(i)} \in \Lambda \text{ if } i \neq m, j

A \text{fwd}-message requesting both the modified data and exclusive ownership can always be acknowledged immediately. With FIFO message passing, if the site has no exclusive ownership when a \text{fwd-store-req} arrives, a negative acknowledgement is sent to the home after the valid data, if exists, has been flushed.
3.2.6 Nak-impending State

\[ \Sigma = \left\{ \text{H-nak-impending}(m)(\text{store-req}), \alpha, \bigcup_{i} W^{(i)} \right\} \]

where

\[ W^{(m)} \in \left\{ \{\text{C-invalid}, \bot, (\text{fwd-store-req}), \Phi\}, \right. \]
\[ \{\text{C-shared}, \alpha, (\text{fwd-store-req}), \Phi\}, \]
\[ \{\text{C-req-in-progress}(\text{load}), \bot, (\text{fwd-store-req}), (\text{load-req})\}, \]
\[ \{\text{C-req-in-progress}(\text{store} \beta), \bot, (\text{fwd-store-req}), (\text{store-req})\}, \]
\[ \{\text{C-invalid}, \bot, \Phi, (\text{fwd-store-nak})\}, \]
\[ \{\text{C-req-in-progress}(\text{load}), \bot, \Phi, (\text{fwd-store-nak}) \star (\text{load-req})\}, \]
\[ \{\text{C-req-in-progress}(\text{store} \beta), \bot, \Phi, (\text{fwd-store-nak}) \star (\text{store-req})\}, \]
\[ \{\text{C-req-in-progress}(\text{load}), \bot, \Phi, (\text{load-req}) \star (\text{fwd-store-nak})\}, \]
\[ \{\text{C-req-in-progress}(\text{store} \beta), \bot, \Phi, (\text{store-req}) \star (\text{fwd-store-nak})\}\}

\[ W^{(j)} = \{\text{C-req-in-progress}(\text{store} \gamma), \bot, \Phi, \Phi\} \]

\[ W^{(i)} \in \Lambda \text{ if } i \neq m, j \]

3.3 Message Processing Rules

This section specifies the set of message processing rules to formally define the HEng and SEng operations of the Base scheme. The operation rules can be used to rewrite memory configurations while sequential consistency is always maintained. In other words, if the system is in a legal configuration, after an operation rule is performed, the system will still be in a legal configuration.

3.3.1 Home Engine Rules

When the home engine receives a protocol message, it chooses an appropriate operation rule to service the message, removing the consumed message from the corresponding Hnet, modifying Hdir and Hval, and appending generated messages at the end of appropriate Snets if necessary. However, a home engine rule cannot modify any Sdir or
Sval.

Uncached-LoadReq Rule

\[
\left\{ \text{H-uncached}, \alpha, \bigsqcup_{i} W^{(i)} \right\} \Rightarrow \left\{ \text{H-shared}[\{k\}], \alpha, \bigsqcup_{i \neq k} W^{(i)}, \hat{W}^{(k)} \right\}
\]

where

\[
W^{(k)} = \{ \text{C-req-in-progress}(\text{load}), \bot, \Phi, \langle \text{load-req} \rangle \}
\]

\[
\hat{W}^{(k)} = \{ \text{C-req-in-progress}(\text{load}), \bot, \langle \text{load-ack} \alpha \rangle, \Phi \}
\]

If the home is in the H-uncached state when a load request is received, the requested data is supplied. The home directory state is changed to H-shared.

Uncached-StoreReq Rule

\[
\left\{ \text{H-uncached}, \alpha, \bigsqcup_{i} W^{(i)} \right\} \Rightarrow \left\{ \text{H-modified}[k], \bot, \bigsqcup_{i \neq k} W^{(i)}, \hat{W}^{(k)} \right\}
\]

where

\[
W^{(k)} = \{ \text{C-req-in-progress}(\text{store} \beta), \bot, \Phi, \langle \text{store-req} \rangle \}
\]

\[
\hat{W}^{(k)} = \{ \text{C-req-in-progress}(\text{store} \beta), \bot, \langle \text{store-ack} \alpha \rangle, \Phi \}
\]

If the home is in the H-uncached state when a store request is received, the exclusive ownership is sent to the requesting site. The home directory state is changed to H-modified.

Shared-LoadReq Rule

\[
\left\{ \text{H-shared}[S], \alpha, \bigsqcup_{i} W^{(i)} \right\} \Rightarrow \left\{ \text{H-shared}[S\cup\{k\}], \alpha, \bigsqcup_{i \neq k} W^{(i)}, \hat{W}^{(k)} \right\}
\]

where

\[
W^{(k)} = \{ \text{C-req-in-progress}(\text{load}), \bot, \Phi, \langle \text{load-req} \rangle \}
\]

\[
\hat{W}^{(k)} = \{ \text{C-req-in-progress}(\text{load}), \bot, \langle \text{load-ack} \alpha \rangle, \Phi \}
\]

If the home is in the H-shared state when a load request is received, the requested data is supplied. The home directory records the site that will obtain the shared copy.
Shared-StoreReq Rule

\[
\left\{\begin{array}{l}
\text{H-shared}[S], \alpha, \bigcup_i W^{(i)}
\end{array}\right\} \Rightarrow
\left\{\begin{array}{l}
\text{H-multicast-in-progress}[S]\langle\text{store-req } k\rangle, \alpha, \bigcup_{i \notin S \cup \{k\}} W^{(i)}, \bigcup_{i \in S \cup \{k\}} \hat{W}^{(i)}
\end{array}\right\}
\] (3.11)

where

\[
W^{(k)} = \{\text{C-req-in-progress}(\text{store } \beta), \bot, \Phi, \text{store-req}\}
\]

\[
\hat{W}^{(k)} = \begin{cases}
\{\text{C-req-in-progress}(\text{store } \beta), \bot, \Phi, \Phi\} & \text{if } k \notin S \\
\{\text{C-req-in-progress}(\text{store } \beta), \bot, \langle\text{invalidate-req}, \Phi\rangle\} & \text{if } k \in S
\end{cases}
\]

\[
\hat{W}^{(i)} = \{\text{Sdir}^{(i)}, \text{Sval}^{(i)}, \text{Snet}^{(i)} \ast \langle\text{invalidate-req}, \text{Hnet}^{(i)}\rangle\} & \text{if } i \in S - \{k\}
\]

If the home is in the H-shared state when a store request is received, invalidation requests are sent to the sites that have shared copies. The home directory state becomes H-multicast-in-progress.

Modified-LoadReq Rule

\[
\left\{\begin{array}{l}
\text{H-modified}[m], \bot, \bigcup_i W^{(i)}
\end{array}\right\} \Rightarrow
\left\{\begin{array}{l}
\text{H-fwd-in-progress}[m]\langle\text{load-req } k\rangle, \bot, \bigcup_{i \neq m, k} W^{(i)}, \hat{W}^{(k)}, \hat{W}^{(m)}
\end{array}\right\}
\] (3.12)

where

\[
W^{(k)} = \{\text{C-req-in-progress}(\text{load}), \bot, \Phi, \text{load-req}\}
\]

\[
\hat{W}^{(k)} = \{\text{C-req-in-progress}(\text{load}), \bot, \Phi, \Phi\}
\]

\[
\hat{W}^{(m)} = \{\text{Sdir}^{(m)}, \text{Sval}^{(m)}, \text{Snet}^{(m)} \ast \langle\text{fwd-load-req}, \text{Hnet}^{(m)}\rangle\}
\]

If the home is in the H-modified state when a load request is received, the request is suspended and a fwd- message is sent to the modified site to request the most up-to-date data. The home directory state is changed to H-forward-in-progress.
Modified-StoreReq Rule

\[
\left\{ \text{H.modified}[m], \bot, \bigcup_i W^{(i)} \right\} \Rightarrow \\
\left\{ \text{H.fwd-in-progress}[m](\text{store-req } k), \bot, \bigcup_{i \neq m,k} W^{(i)}, \tilde{W}^{(k)}, \tilde{W}^{(m)} \right\}
\]

where

\[
W^{(k)} = \{ \text{C-req-in-progress}(\text{store } \beta), \bot, \Phi, \{\text{store-req}\} \}
\]
\[
\tilde{W}^{(k)} = \{ \text{C-req-in-progress}(\text{store } \beta), \bot, \Phi, \Phi \}
\]
\[
\tilde{W}^{(m)} = \{ \text{Sdir}^{(m)}, \text{Sval}^{(m)}, \text{Snet}^{(m)} \ast \{\text{fwd-store-req}, \text{Hnet}^{(m)}\} \}
\]

If the home is in the H-modified state when a store request is received, the request is suspended and a \textit{fwd}-message is sent to the modified site to request both the most up-to-date data and exclusive ownership. The home directory state is changed to H-forward-in-progress.

Modified-Kill Rule

\[
\left\{ \text{H.modified}[m], \bot, \bigcup_i W^{(i)} \right\} \Rightarrow \left\{ \text{H.uncached}, \alpha, \bigcup_{i \neq m} W^{(i)}, \tilde{W}^{(m)} \right\}
\]

where

\[
W^{(m)} = \{ \text{Sdir}^{(m)}, \text{Sval}^{(m)}, \text{Snet}^{(m)}, \{\text{kill-infa}\} \ast \text{Hnet}^{(m)} \}
\]
\[
\tilde{W}^{(m)} = \{ \text{Sdir}^{(m)}, \text{Sval}^{(m)}, \text{Snet}^{(m)}, \text{Hnet}^{(m)} \}
\]

If the home is in the H-modified state when a \textit{kill-inf} arrives, the most up-to-date data is written to the memory. The home directory state becomes H-uncached.

Modified-Clean Rule

\[
\left\{ \text{H.modified}[m], \bot, \bigcup_i W^{(i)} \right\} \Rightarrow \left\{ \text{H.shared}\{m\}, \alpha, \bigcup_{i \neq m} W^{(i)}, \tilde{W}^{(m)} \right\}
\]

\[\text{(3.15)}\]
where

\[
W^{(m)} = \{Sdir^{(m)}, Sval^{(m)}, Snet^{(m)}, \langle\text{clean-inf } a\rangle \ast Hnet^{(m)}_1\}
\]

\[
\hat{W}^{(m)} = \{Sdir^{(m)}, Sval^{(m)}, Snet^{(m)}, Hnet^{(m)}_1\}
\]

If the home is in the H-modified state when a \textit{clean-inf} arrives, the most up-to-date data is written to the memory. The home directory state becomes H-shared.

**Multicast-InvalidAck Rules**

\[
\begin{align*}
\left\{ \text{H-multicast-in-progress}[S](\text{store } j), \alpha, \bigsqcup_i W^{(i)} \right\} &\Rightarrow \\
\left\{ \text{H-multicast-in-progress}[S\setminus\{k\}](\text{store } j), \alpha, \bigsqcup_{i \neq k} W^{(i)} \right\} &\text{if } \{k\} \subseteq S \tag{3.16}
\end{align*}
\]

where

\[
W^{(k)} = \{Sdir^{(k)}, Sval^{(k)}, Snet^{(k)}, Hnet^{(k)}_1 \ast \langle\text{invalidate-ack}\rangle \ast Hnet^{(k)}_2\}
\]

\[
\hat{W}^{(k)} = \{Sdir^{(k)}, Sval^{(k)}, Snet^{(k)}, Hnet^{(k)}_1 \ast Hnet^{(k)}_2\}
\]

\[
\begin{align*}
\left\{ \text{H-multicast-in-progress}[S](\text{store } j), \alpha, \bigsqcup_i W^{(i)} \right\} &\Rightarrow \\
\left\{ \text{H-modified}[j], \bot, \bigsqcup_{i \neq k, j} W^{(i)} \right\} &\text{if } \{k\} = S \tag{3.17}
\end{align*}
\]

where

\[
W^{(k)} = \{Sdir^{(k)}, Sval^{(k)}, Snet^{(k)}, Hnet^{(k)}_1 \ast \langle\text{invalidate-ack}\rangle \ast Hnet^{(k)}_2\}
\]

\[
\hat{W}^{(k)} = \\
\begin{cases} 
\{Sdir^{(k)}, Sval^{(k)}, Snet^{(k)}, Hnet^{(k)}_1 \ast Hnet^{(k)}_2\} &\text{if } k \neq j \\
\{Sdir^{(k)}, Sval^{(k)}, Snet^{(k)} \ast \langle\text{store-ack } a\rangle, Hnet^{(k)}_1 \ast Hnet^{(k)}_2\} &\text{if } k = j
\end{cases}
\]

\[
\hat{W}^{(j)} = \{Sdir^{(j)}, Sval^{(j)} \ast \langle\text{store-ack } a\rangle, Hnet^{(j)}\} \text{ if } j \neq k
\]

There are two operation rules regarding the H-multicast-in-progress state when an invalidation acknowledgement is received. After all the shared copies have been invalidated, the suspended store request is serviced and the home directory state becomes H-modified.
It should be noted that in the message pool Hnet an invalidation acknowledgement can overtake other messages in order to be processed by the home engine immediately. This is critical to ensure that the scheme is free from deadlock, since the home engine would not be able to service any other protocol message before all the invalidations have been acknowledged. The load and store requests that temporarily cannot be processed need to be buffered.

**Forward-Load-Ack Rule**

\[
\left\{ \text{H-forward-in-progress}[m] \langle \text{load-req} \rangle, \bot, \bigsqcup_i W^{(i)} \right\} \Rightarrow
\left\{ \text{H-shared}[[m, j]], \alpha, \bigsqcup_{i \neq m, j} W^{(i)}, \hat{W}^{(m)}, \hat{W}^{(j)} \right\}
\]

(3.18)

where

\[
W^{(m)} = \{ \text{Sdir}^{(m)}, \text{Sval}^{(m)}, \text{Snet}^{(m)}, (fwd-load-ack \alpha) \star \text{Hnet}^{(m)} \}
\]

\[
\hat{W}^{(m)} = \{ \text{Sdir}^{(m)}, \text{Sval}^{(m)}, \text{Snet}^{(m)}, \text{Hnet}^{(m)} \}
\]

\[
\hat{W}^{(j)} = \{ \text{Sdir}^{(j)}, \text{Sval}^{(j)}, \text{Snet}^{(j)} \star \langle \text{load-ack} \alpha \rangle, \text{Hnet}^{(j)} \}
\]

**Forward-Load-Kill Rule**

\[
\left\{ \text{H-forward-in-progress}[m] \langle \text{load-req} \rangle, \bot, \bigsqcup_i W^{(i)} \right\} \Rightarrow
\left\{ \text{H-shared}[[j]], \alpha, \bigsqcup_{i \neq m, j} W^{(i)}, \hat{W}^{(m)}, \hat{W}^{(j)} \right\}
\]

(3.19)

where

\[
W^{(m)} = \{ \text{Sdir}^{(m)}, \text{Sval}^{(m)}, \text{Snet}^{(m)}, (kill-inf \alpha) \star \text{Hnet}^{(m)} \}
\]

\[
\hat{W}^{(m)} = \{ \text{Sdir}^{(m)}, \text{Sval}^{(m)}, \text{Snet}^{(m)}, \text{Hnet}^{(m)} \}
\]

\[
\hat{W}^{(j)} = \{ \text{Sdir}^{(j)}, \text{Sval}^{(j)}, \text{Snet}^{(j)} \star \langle \text{load-ack} \alpha \rangle, \text{Hnet}^{(j)} \}
\]
Forward-Load-Clean Rule

\[
\begin{align*}
\left\{ \text{H-forward-in-progress}[m](\text{load-req } j), \bot, \bigcup_i W^{(i)} \right\} & \Rightarrow \\
\left\{ \text{H-shared}[\{m, j\}], \alpha, \bigcup_{i \neq m, j} W^{(i)}, \hat{W}^{(m)}, \hat{W}^{(j)} \right\}
\end{align*}
\]

where

\[
W^{(m)} = \{Sdir^{(m)}, Sval^{(m)}, Snet^{(m)}, \langle\text{clean-inf } \alpha\rangle \ast \text{Hnet}_1^{(m)}\}
\]

\[
\hat{W}^{(m)} = \{Sdir^{(m)}, Sval^{(m)}, Snet^{(m)}, \text{Hnet}_1^{(m)}\}
\]

\[
\hat{W}^{(j)} = \{Sdir^{(j)}, Sval^{(j)}, Snet^{(j)} \ast \langle\text{load-ack } \alpha\rangle, \text{Hnet}^{(j)}\}
\]

If the home is in the H-forward-in-progress state with a suspended load request when a \text{fwd-load-ack}, \text{kill-inf} or \text{clean-inf} arrives, the most up-to-date data is written to the memory, and the suspended message is serviced. The home directory state becomes H-shared.

Forward-Store-Ack Rule

\[
\begin{align*}
\left\{ \text{H-forward-in-progress}[m](\text{store-req } j), \bot, \bigcup_i W^{(i)} \right\} & \Rightarrow \\
\left\{ \text{H-modified}[j], \bot, \bigcup_{i \neq m, j} W^{(i)}, \hat{W}^{(m)}, \hat{W}^{(j)} \right\}
\end{align*}
\]

where

\[
W^{(m)} = \{Sdir^{(m)}, Sval^{(m)}, Snet^{(m)}, \langle\text{fwd-store-ack } \alpha\rangle \ast \text{Hnet}_1^{(m)}\}
\]

\[
\hat{W}^{(m)} = \{Sdir^{(m)}, Sval^{(m)}, Snet^{(m)}, \text{Hnet}_1^{(m)}\}
\]

\[
\hat{W}^{(j)} = \{Sdir^{(j)}, Sval^{(j)}, Snet^{(j)} \ast \langle\text{store-ack } \alpha\rangle, \text{Hnet}^{(j)}\}
\]

If the home is in the H-forward-in-progress state with a suspended store request when a \text{fwd-store-ack} arrives, the suspended message is serviced. The home directory state becomes H-modified.
Forward-Store-Kill Rule

\[
\left\{ \begin{array}{c}
H\text{-forward-in-progress}[m](\text{store-req } j), \bot, \bigcup_i W^{(i)}
\end{array} \right\} \Rightarrow
\left\{ \begin{array}{c}
H\text{-nak-impending}[m](\text{store-req } j), \alpha, \bigcup_i W^{(i)}, \tilde{W}^{(m)}
\end{array} \right\} 
\]

(3.22)

where

\[
W^{(m)} = \{Sdir^{(m)}, Sval^{(m)}, Snet^{(m)}, \text{kill-inf } \alpha \ast Hnet_1^{(m)}\}
\]

\[
\tilde{W}^{(m)} = \{Sdir^{(m)}, Sval^{(m)}, Snet^{(m)}, Hnet_1^{(m)}\}
\]

Forward-Store-Clean Rule

\[
\left\{ \begin{array}{c}
H\text{-forward-in-progress}[m](\text{store-req } j), \bot, \bigcup_i W^{(i)}
\end{array} \right\} \Rightarrow
\left\{ \begin{array}{c}
H\text{-nak-impending}[m](\text{store-req } j), \alpha, \bigcup_i W^{(i)}, \tilde{W}^{(m)}
\end{array} \right\} 
\]

(3.23)

where

\[
W^{(m)} = \{Sdir^{(m)}, Sval^{(m)}, Snet^{(m)}, \text{clean-inf } \alpha \ast Hnet_1^{(m)}\}
\]

\[
\tilde{W}^{(m)} = \{Sdir^{(m)}, Sval^{(m)}, Snet^{(m)}, Hnet_1^{(m)}\}
\]

If the home is in the H-forward-in-progress state with a suspended store request when a \textit{kill-inf} or \textit{clean-inf} arrives, the home directory state is changed to H-nak-impending.

Impending-Store-Nak Rule

\[
\left\{ \begin{array}{c}
H\text{-nak-impending}[m](\text{store-req } j), \alpha, \bigcup_i W^{(i)}
\end{array} \right\} \Rightarrow
\left\{ \begin{array}{c}
H\text{-modified}[m] \bot, \bigcup_{i \neq m} W^{(i)}, \tilde{W}^{(m)}, \tilde{W}^{(j)}
\end{array} \right\} 
\]

(3.24)

where

\[
W^{(m)} = \{Sdir^{(m)}, Sval^{(m)}, Snet^{(m)}, Hnet_1^{(m)} \ast \text{fwd-store-nak} \ast Hnet_2^{(m)}\}
\]

\[
\tilde{W}^{(m)} = \{Sdir^{(m)}, Sval^{(m)}, Snet^{(m)}, Hnet_1^{(m)} \ast Hnet_2^{(m)}\}
\]

\[
\tilde{W}^{(j)} = \{Sdir^{(j)}, Sval^{(j)}, Snet^{(j)} \ast \text{store-ack } \alpha, Hnet^{(j)}\}
\]
If the home is in the H-nak-impending state when a \textit{fwd-store-nak} is received, the suspended store request is serviced. The home directory state becomes H-modified. The negative acknowledgement is allowed to overtake messages that the home engine temporarily cannot process in order to avoid deadlock.

### 3.3.2 Site Engine Rules

A site engine is responsible for processing not only memory access and cache management instructions, but also protocol messages issued from the home site. At site \( k \), when a site engine rule is applied, the site engine may modify the site directory state \( SDir^{(k)} \) and the site cache line \( Sval^{(k)} \). However, it cannot change the home directory state and the memory block value, or the site directory state and site cache line at any other site. A site engine rule can be simply expressed in the form of \( "W^{(k)} \Rightarrow \hat{W}^{(k)}" \), since other components in the memory configuration are not affected.

**Load-Issue Rule**

\[
\{\text{C-invalid, } \bot, Snet^{(k)}, Hnet^{(k)}\} \Rightarrow \\
\{\text{C-req-in-progress(load), } \bot, Snet^{(k)}, Hnet^{(k)} \ast (\text{load-req})\} \quad (3.25)
\]

A load request can be issued only when the site is in the C-invalid state. The site directory is then changed to C-req-in-progress.

**Store-Issue Rules**

\[
\{\text{C-invalid, } \bot, Snet^{(k)}, Hnet^{(k)}\} \Rightarrow \\
\{\text{C-req-in-progress(store } \beta), \bot, Snet^{(k)}, Hnet^{(k)} \ast (\text{store-req})\} \quad (3.26)
\]

\[
\{\text{C-shared, } \alpha, Snet^{(k)}, Hnet^{(k)}\} \Rightarrow \\
\{\text{C-req-in-progress(store } \beta), \bot, Snet^{(k)}, Hnet^{(k)} \ast (\text{store-req})\} \quad (3.27)
\]

A store request can be issued only when the site is in the C-invalid or C-shared state. The site directory is then changed to C-req-in-progress.
Cache-Delete Rule

\[ \{\text{C-shared}, \alpha, \text{Snet}^{(k)}, \text{Hnet}^{(k)}\} \Rightarrow \{\text{C-invalid}, \bot, \text{Snet}^{(k)}, \text{Hnet}^{(k)}\} \]  
(3.28)

When a shared cache line is deleted, the home site is not notified. The site directory state is changed to C-invalid.

Cache-Kill Rule

\[ \{\text{C-modified}, \alpha, \text{Snet}^{(k)}, \text{Hnet}^{(k)}\} \Rightarrow \{\text{C-invalid}, \bot, \text{Snet}^{(k)}, \text{Hnet}^{(k)} \ast \langle \text{kill-inf} \alpha \rangle\} \]  
(3.29)

When a modified cache line is killed, the most up-to-date data and exclusive ownership are sent back to the home. The site directory state is changed to C-invalid.

Cache-Clean Rule

\[ \{\text{C-modified}, \alpha, \text{Snet}^{(k)}, \text{Hnet}^{(k)}\} \Rightarrow \{\text{C-shared}, \alpha, \text{Snet}^{(k)}, \text{Hnet}^{(k)} \ast \langle \text{clean-inf} \alpha \rangle\} \]  
(3.30)

When a modified cache line is cleaned, the most up-to-date data is written back to the memory. The site directory state is changed to C-shared.

Load-Done Rule

\[ \{\text{C-req-in-progress(load)}, \bot, \langle \text{load-ack} \alpha \rangle \ast \text{Snet}_1^{(k)}, \text{Hnet}^{(k)}\} \Rightarrow \]

\[ \{\text{C-shared}, \alpha, \text{Snet}_1^{(k)}, \text{Hnet}^{(k)}\} \]  
(3.31)

If the site is in the C-req-in-progress state when a load-ack is received, the data is placed in Sval and the suspended load instruction is completed. The site directory state is changed to C-shared.

Store-Done rule

\[ \{\text{C-req-in-progress(store } \beta), \bot, \langle \text{store-ack} \alpha \rangle \ast \text{Snet}_1^{(k)}, \text{Hnet}^{(k)}\} \Rightarrow \]

\[ \{\text{C-modified}, \gamma, \text{Snet}_1^{(k)}, \text{Hnet}^{(k)}\} \]  
(3.32)

where \(\gamma = \text{write}(\beta, \alpha)\).
If the site is in the C-req-in-progress state when a store-ack is received, the data is placed in Sval and the suspended store instruction is performed. The site directory state is changed to C-modified.

**Invalidation Rules**

\[
\{ \text{C-shared}, \alpha, \text{invalidate-req}, \text{Hnet}^{(k)} \} \Rightarrow \\
\{ \text{C-invalid}, \bot, \Phi, \text{Hnet}^{(k)} \ast \text{invalidate-ack} \} \quad (3.33)
\]

\[
\{ \text{C-invalid}, \bot, \text{invalidate-req}, \text{Hnet}^{(k)} \} \Rightarrow \\
\{ \text{C-invalid}, \bot, \Phi, \text{Hnet}^{(k)} \ast \text{invalidate-ack} \} \quad (3.34)
\]

\[
\{ \text{C-req-in-progress(instr)}, \bot, \text{invalidate-req}, \text{Hnet}^{(k)} \} \Rightarrow \\
\{ \text{C-req-in-progress(instr)}, \bot, \Phi, \text{Hnet}^{(k)} \ast \text{invalidate-ack} \} \quad (3.35)
\]

An invalidation request is always immediately acknowledged. This is critical to avoid deadlock.

**Forward-Load Rules**

\[
\{ \text{C-modified}, \alpha, \text{fwd-load-req}, \text{Hnet}^{(k)} \} \Rightarrow \\
\{ \text{C-shared}, \alpha, \Phi, \text{Hnet}^{(k)} \ast \text{fwd-load-ack} \alpha \} \quad (3.36)
\]

\[
\{ \text{C-shared}, \alpha, \text{fwd-load-req}, \text{Hnet}^{(k)} \} \Rightarrow \\
\{ \text{C-shared}, \alpha, \Phi, \text{Hnet}^{(k)} \} \quad (3.37)
\]

\[
\{ \text{C-invalid}, \bot, \text{fwd-load-req}, \text{Hnet}^{(k)} \} \Rightarrow \\
\{ \text{C-invalid}, \bot, \Phi, \text{Hnet}^{(k)} \} \quad (3.38)
\]
When a \textit{fwd-load-req} arrives, if the site directory state is C-modified, the site engine sends the modified data to the home. However, if the site directory is C-shared, C-invalid or C-req-in-progress, the \textit{fwd-} request is ignored.

\textbf{Forward-Store Rules}

\begin{align*}
\{\text{C-modified}, \alpha, \langle \text{fwd-store-req}, \text{Hnet}^{(k)}\rangle\} & \Rightarrow \\
\{\text{C-invalid}, \perp, H, H\text{net}^{(k)} \ast \langle \text{fwd-store-ack} \alpha \rangle\}\end{align*} \quad (3.40)

\begin{align*}
\{\text{C-shared}, \alpha, \langle \text{fwd-store-req}, \text{Hnet}^{(k)}\rangle\} & \Rightarrow \\
\{\text{C-invalid}, \perp, H, H\text{net}^{(k)} \ast \langle \text{fwd-store-nak} \rangle\}\end{align*} \quad (3.41)

\begin{align*}
\{\text{C-invalid}, \perp, \langle \text{fwd-store-req}, \text{Hnet}^{(k)}\rangle\} & \Rightarrow \\
\{\text{C-invalid}, \perp, H, H\text{net}^{(k)} \ast \langle \text{fwd-store-nak} \rangle\}\end{align*} \quad (3.42)

\begin{align*}
\{\text{C-req-in-progress}(\text{instr}), \perp, \langle \text{fwd-store-req}, \text{Hnet}^{(k)}\rangle\} & \Rightarrow \\
\{\text{C-req-in-progress}(\text{instr}), \perp, H, H\text{net}^{(k)} \ast \langle \text{fwd-store-nak} \rangle\}\end{align*} \quad (3.43)

When a \textit{fwd-store-req} arrives, if the site is in the C-modified state, the site engine sends the modified data to the home. However, if the site directory state is C-shared, C-invalid or C-req-in-progress, a negative acknowledgement is sent to the home. In any case, the \textit{fwd-} request can be processed without any delay.
3.4 Protocol Correctness

The logical correctness of the Base scheme is based on the following assumptions:

- The transfer latency for any protocol message is finite. In other words, the network is reliable for message passing.
- At each site, the home engine has enough buffer space for protocol messages that cannot be processed temporarily.
- For any memory block, the initial configuration is in the form of
  \[
  \Sigma_0 = \left\{ \text{H-uncached}, \alpha, \bigcup_i W^{(i)} \right\}
  \]
  where
  \[
  W^{(i)} = \{ \text{C-invalid, } \perp, \Phi, \Phi \}.
  \]

The following claims can be easily verified, proving that the Base scheme implements sequential consistency, and is free from deadlock, livelock and starvation.

Claim 1 The memory system is in sequential consistency for all the legal configurations defined in 3.1 – 3.7.

Claim 2 If the memory system is in a legal configuration, after an operation rule defined in 3.8 – 3.43 is performed, the new memory configuration is legal.

Claim 3 At any site, when the HEng receives a protocol message, it either processes the message immediately, or buffers the message for later processing. Any protocol message, if buffered, will be resumed in finite time and will not be buffered again.

Claim 4 If the HEng begins processing a protocol message, it will complete the service. In other words, there is no circular dependencies in the Base state transition diagrams.

The last claim needs more explanation. The following properties of the Base scheme ensure that protocol messages regarding ownership transfers can always be serviced.
Protocol messages are classified into three categories: requests, replies and notifications. Only protocol requests are allowed to be buffered when they cannot be processed, while protocol replies and notifications can always be immediately serviced.

Invalidation requests are non-blocking and never buffered. At a site, when the SEng receives an invalidate-req, it immediately sends an acknowledgement to the block's home, regardless of whether the flush operation has been performed.

Forwarded requests are non-blocking and never buffered. At a site, when the SEng receives a fwd-request, it immediately sends a fwd-reply to the block's home. The reply can be either positive or negative, depending on whether the forwarded request has been satisfied.

3.5 The Base\(^+\) Scheme

This section introduces the Base\(^+\) scheme, the Base scheme version designed for non-FIFO networks. Figure 3.4 gives the protocol message specification, where the notation own-site represents the site that currently owns the accessed block, which can be either the home site for uncached/shared blocks or the modified site for modified blocks (other notations have the same meaning as in Figure 2.3). The following discussion highlights the distinction between the Base and Base\(^+\) schemes.

3.5.1 Home Engine

Figure 3.5 illustrates the HEng state transition diagram, which includes seven states: H-uncached, H-shared, H-modified, H-forward-in-progress, H-multicast-in-progress, H-nak-impending and H-inf-impending. The forwarding policy is adopted.

load-req

Suppose the HEng receives a load-req from site \(k\). If the HDir state is H-modified[\(m\)], the HEng forwards the load request to site \(m\), and changes the HDir state H-forward-in-
### Table: Base+ Protocol Message Specification

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Message Fields</th>
<th>Issuing &amp; Receiving Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>load-req</td>
<td>command block</td>
<td>req-site -&gt; home</td>
</tr>
<tr>
<td>store-req</td>
<td>command block</td>
<td>req-site -&gt; home</td>
</tr>
<tr>
<td>fwd-load-req</td>
<td>command block</td>
<td>home -&gt; mod-site</td>
</tr>
<tr>
<td>fwd-store-req</td>
<td>command block</td>
<td>home -&gt; mod-site</td>
</tr>
<tr>
<td>invalidate-req</td>
<td>command block</td>
<td>home -&gt; shr-site</td>
</tr>
<tr>
<td>load-ack</td>
<td>command block</td>
<td>own-site -&gt; req-site</td>
</tr>
<tr>
<td>store-ack</td>
<td>command block</td>
<td>own-site -&gt; req-site</td>
</tr>
<tr>
<td>fwd-load-ack</td>
<td>command block</td>
<td>mod-site -&gt; home</td>
</tr>
<tr>
<td>fwd-load-nak</td>
<td>command block</td>
<td>mod-site -&gt; home</td>
</tr>
<tr>
<td>fwd-load-neg</td>
<td>command block</td>
<td>mod-site -&gt; home</td>
</tr>
<tr>
<td>fwd-store-ack</td>
<td>command block</td>
<td>mod-site -&gt; home</td>
</tr>
<tr>
<td>fwd-store-nak</td>
<td>command block</td>
<td>mod-site -&gt; home</td>
</tr>
<tr>
<td>invalidate-ack</td>
<td>command block</td>
<td>shr-site -&gt; home</td>
</tr>
<tr>
<td>clean-inf</td>
<td>command block</td>
<td>mod-site -&gt; home</td>
</tr>
<tr>
<td>kill-inf</td>
<td>command block</td>
<td>mod-site -&gt; home</td>
</tr>
</tbody>
</table>

Figure 3.4: Base+ Protocol Message Specification

progress\[k\]. Later when the corresponding \textit{fwd-load-ack} arrives, the HEng updates the memory with the modified data, and then changes the HDir state to H-shared\[S\], where \(S = \{m, k\}\).

However, the modified cache line at site \(m\) may have been killed or cleaned before the \textit{fwd-load-req} arrives. In this case, since the message passing is non-FIFO, the negative acknowledgement regarding the \textit{fwd-load-req} may reach the home before the kill/clean notification.

If the \textit{kill-inf/clean-inf} is received by the home first, the HEng writes the most up-to-date data to the memory, and changes the HDir state to H-nak-impending\[S\], where \(S = \{k\}\) or \(\{k, m\}\), depending on whether the notification is \textit{kill-inf} or \textit{clean-inf}. Later when the corresponding \textit{fwd-load-nak} arrives, the HEng sends a \textit{load-ack} to site \(k\) to supply the data, and then changes the HDir state from H-nak-impending\[S\] to H-shared\[S\].

If the \textit{fwd-load-nak} overtakes the kill/clean notification and arrives at the home first, the HEng records the load instruction and changes the HDir state to H-inf-impending. Later when the corresponding \textit{kill-inf/clean-inf} is received, the HEng sends a \textit{load-ack} to site \(k\) to supply the data, and then changes the HDir state to H-shared\[S\], where \(S = \{k\}\).
or \( \{k, m\} \), depending on whether the protocol notification is \textit{kill-inf} or \textit{clean-inf}.

\textbf{store-req}

Suppose the HEng receives a \textit{store-req} from site \( k \). If the HDir state is H-modified\([m]\), the HEng forwards the store request to site \( m \), and changes the HDir state to H-forward-in-progress\([k]\). Later when the corresponding \textit{fwd-load-ack} arrives, the HEng updates the HDir state from H-forward-in-progress\([k]\) to H-modified\([k]\).

However, when the HDir state is H-forward-in-progress\([k]\), if a kill/clean notification arrives, the HEng records the modified data, and changes the HDir state to H-nak-impending. Later when the corresponding \textit{fwd-store-nak} or \textit{fwd-store-neg} arrives, the HEng sends the most up-to-date data and exclusive ownership to site \( k \). The HDir state
There are two other possible cases due to the non-FIFO message passing. (1) If the \textit{fwd-store-nak} overtakes the protocol notification and arrives at the home first, the HEng records the necessary knowledge about the store instruction, and changes the HDir state to H-inf-impending. Later when the \textit{kill-inf/clean-inf} is received, the HEng sends \textit{store-ack} to site $k$ to provide the data and exclusive ownership. The HDir state is changed to H-modified[$k$]. (2) If the \textit{fwd-store-neg} arrives at the home when the HDir state is H-forward-in-progress, the HEng sends the \textit{fwd-store-req} again to the site $m$, which possibly has or will have the exclusive ownership of the accessed block.

### 3.5.2 Site Engine

Figure 3.6 gives the SEng state transition diagram.
**load**

If the SDir state is C-invalid, the SEng suspends the load instruction, sends a load-req to the home to request the data, and changes the SDir state to C-req-in-progress. Later when the corresponding load-ack arrives, the SEng checks the inv-pending flag. (1) If the flag has been set, the SEng supplies the data to the suspended load instruction, and then changes the SDir state to C-invalid. Then the data is discarded. (2) If the flag is not set, the SEng caches the data, changes the SDir state to C-shared, and then resumes the suspended load instruction.

**invalidate-req**

If the SDir state is C-req-in-progress with a suspended load instruction, the SEng sends an invalidate-ack to the home, and sets the inv-pending flag. The SDir state is not changed.

**fwd-load-req**

If the SDir state is C-modified, the SEng supplies the modified data to the site from which the forwarded load request comes, and then sends a fwd-load-ack to the home to write the most up-to-date data to the memory. The SDir state is changed to C-shared.

If the SDir state is C-invalid, C-shared or C-req-in-progress with a suspended load instruction, the SEng sends a fwd-load-nak to the home. The SDir state is not changed.

If the SDir state is C-req-in-progress with a suspended store instruction, the SEng sends a fwd-load-neg to the home. The SDir state is not changed.

**fwd-store-req**

If the SDir state is C-modified, the SEng supplies the modified data and exclusive ownership to the site from which the forwarded store request comes, and then sends a fwd-store-ack to the home to acknowledge that the exclusive ownership has been transferred. The SDir state is changed to C-invalid.
If the SDir state is C-invalid, C-shared or C-req-in-progress with a suspended load instruction, the SEng sends a *fwd-store-nak* to the home. The SDir state is not changed.

If the SDir state is C-req-in-progress with a suspended store instruction, the SEng sends a *fwd-store-neg* to the home. The SDir state is not changed.
Chapter 4

Incorporation Of I-Structures

This chapter presents the ISC (I-Structure built-in Coherence) scheme, which implements I-structures at the coherence protocol level to provide efficient support for fine-grain computation and synchronization. For the performance reason, deferred lists are allowed to be maintained in a distributed fashion under restricted situations.

4.1 Introduction Of I-structures

An I-structure [8] is a data structure in which each element maintains a presence bit that has two possible states: full or empty, indicating whether a valid data has been stored in the element. There are two basic I-structure operations: Iload and Istore.

Iload An Iload operation performed on a full element returns the data of the element. An Iload performed on an empty element is blocked, and recorded in the element’s deferred list for later processing.

Istore An Istore operation performed on an empty element writes the value to the element, and modifies the element’s state to full. The blocked Iloads in the element’s deferred list, if existing, are resumed. An Istore operation performed on a full element results in an error.

Initially all the elements are in the empty state. An optional I-structure operation, reclaim, can be used to make an element reusable. A reclaim operation resets the element’s state to empty, thereby allowing the element to be stored with a new value.
Figure 4.1 shows an example in which an I-structure array serves as the rendezvous for a producer and multiple consumers. Producer $P$ is responsible for producing one item for each element, however, the order the items are produced is random. With the synchronization support of I-structures, consumers are allowed to read the array while the producer is writing the array, although when a specific element is filled cannot be predicted. Consumer $C_1$ reads an element that has been filled, and immediately obtains the value. In contrast, consumer $C_2$ reads an empty element, and therefore is deferred until the element is filled.

I-structure element size is typically a word. A cache line may, therefore, consist of multiple elements, including both I-structure and ordinary slots. Figure 4.2 gives an example where cache line $X$ is divided into 4 sub-lines: $X_0$, $X_1$, $X_2$ and $X_3$. Slot $X_0$ is an empty I-structure element which maintains two blocked $I\text{loads}$ in its deferred list. Slot $X_1$ is a full I-structure element with value 100. Slot $X_2$ is an ordinary element whose current value is 400. Slot $X_3$ is an empty I-structure element without any deferred $I\text{load}$.
4.2 Motivation

Previous implementation supporting data structures similar to I-structures treats the synchronization protocol as a separate layer over the underlying cache coherence protocol. However, the write-once behavior characteristic of I-structures allows the Istore operation to be carried out without the exclusive ownership of the accessed block. This implies that some unnecessary protocol messages can be avoided by incorporating I-structures in the coherence scheme.

In Figure 4.3, for example, suppose initially sites A and B have cached block X. Site A performs an Istore operation on slot X0. However, the shared copy of X at site B is neither invalidated nor updated. At site B, the data of slots X1 and X2 are still valid, although the data of the whole cache line is incomplete.

4.3 Deferred List Management

A critical issue in designing coherence schemes that incorporate I-structures is the maintenance of deferred lists. In the Base scheme, there are three stable HDir states: H-uncached, H-shared and H-modified, representing the coherence status that the block is not shared by any site, the block is shared by multiple sites with read-only permission, and the block is exclusively owned by a site with read-write privilege. For a memory block that is in the H-uncached or H-shared state, the deferred lists can be simply maintained at the block’s home. For a memory block with the H-modified state, however, it is not clear
whether the deferred lists should be maintained at the block’s home, or at the modified site.

4.3.1 Maintaining Deferred Lists At Home

A straightforward method is to have the deferred lists be always maintained in the accessed block’s home, regardless of the block’s HDir state. For a modified block, the memory keeps the accurate knowledge about the presence bits of all the individual I-structure slots, although the block’s ordinary slots may have been modified. With this presence bit knowledge, the home’s HEng never sends protocol request to the modified site to request the data of an empty I-structure slot.

Figure 4.4 gives an example in which block X is in the H-modified state, and X’s home is responsible for maintaining the deferred lists. At the home, if an Iload-req for slot X₁ is received, the HEng sends a protocol message to the modified site to request X’s most up-to-date data. However, if an Iload-req for slot X₀ or X₃ arrives, the home’s HEng does not send any message to the modified site, because the corresponding presence bit shows that the requested data is still not available. Instead the Iload-req is deferred at the home until an Istore operation is performed on the slot.

This approach has two serious drawbacks: (1) If an Iload operation performed in the modified site needs to be deferred, it has to be sent to the home where all the deferred lists are maintained, although the exclusive ownership is currently resident in the modified site. (2) Any Istore operation performed in the modified site requires that the modified cache line be written back to the memory, no matter if the home’s deferred list for the I-structure slot is empty. This is because the modified site’s SEng has no knowledge about the deferred lists at the home.

In Figure 4.4, for example, an Iload operation performed in the modified site on slot X₀ or X₃ needs to be sent to the home in order to be recorded in the corresponding deferred list. On the other hand, an Istore operation performed at the modified site on slot X₀ or X₃ causes the modified cache line to be written back to the home, although for slot X₃ there is no deferred Iload.

A simple improvement can be achieved if the SEng at the modified site maintains a
deferred flag (Dflag) for each I-structure slot whose presence bit is 0, recording whether the slot’s deferred list is empty. In this case, when an Istore operation is performed at the modified site, the SEng checks the corresponding Dflag. The modified cache line is written back to the home only when the slot’s Dflag has been set to 1, which indicates that deferred Iloads for the slot need to be resumed.

4.3.2 Maintaining Deferred Lists With Ownership

Maintaining the deferred lists with the accessed block’s ownership gives rise to an obvious advantage: when an Iload operation performed at the modified site needs to be deferred, the SEng just appends the Iload in the slot’s deferred list. Moreover, when an Istore takes place, the SEng knows whether the modified cache line needs to be written to the memory, since all the deferred Iloads information is available at the modified site.

Figure 4.5 shows an example, in which block X is in the H-modified state, and the modified site is in charge of maintaining X’s deferred lists. At the modified site, when an Iload operation is performed on slot X₀ or X₃, the SEng defers the Iload and keeps it in the corresponding deferred list. In this case, no protocol message is generated. Meanwhile, if an Istore operation is performed on slot X₀, the SEng resumes the deferred Iloads in the corresponding deferred list, and sends a protocol message to the home to write the modified cache line to the memory (assuming not all the resumed Iloads were issued from the modified site itself). However, if an Istore operation is performed on slot X₃, the SEng withholds the exclusive ownership, since no deferred Iload exists for the slot.
Figure 4.5: Maintaining Deferred Lists With Ownership

This approach requires that, for a memory block, the deferred lists be transferred whenever the block’s ownership is changed. In other words, if the home or any site obtains a block’s ownership, it also receives the block’s deferred lists. In Figure 4.5 for example, when a clean or kill operation is performed on the modified cache line, the modified site’s SEng must pack all the deferred lists in the protocol message that is sent to the home. If another site is later granted the ownership, the home’s HEng needs to deliver the deferred lists to that site.

An optimization that avoids moving the entire deferred lists over the network only requires the transfer of the deferred lists’ head pointers. In this case, the site that currently owns the memory block always maintains the head pointers of the corresponding deferred lists, from which all the deferred \textit{Iloads} in the network can be retrieved.

4.3.3 Distributing Deferred Lists At Home / Modified Site

The approach the ISC scheme adopts is illustrated in Figure 4.6, allowing a deferred list to be distributed at two sites under certain situations. For a memory block with the H-uncached or H-shared state, a deferred \textit{Iload} is maintained at the block’s home. For a memory block with the H-modified state, a deferred \textit{Iload} is allowed to be recorded either in the slot’s home deferred list which is maintained at the block’s home, or in the slot’s site deferred list which is maintained at the modified site, depending on the time and place the \textit{Iload} operation is performed.

For a modified memory block, the modified site’s SEng maintains a Dflag for each
individual I-structure slot whose presence bit is 0, indicating whether the slot's deferred list at the home is empty. When an *Iload* needs to be deferred, the SEng records it in the corresponding deferred list which is maintained at the modified site. In this case no protocol message is generated.

When an *Istore* is performed, the SEng resumes all the deferred *Iloads* maintained in the slot's deferred list at the site, and then checks the slot's Dflag. If the Dflag has been set to 1, the SEng changes the SDir state to C-shared, and sends the most up-to-date data, along with the remaining deferred *Iloads* that are maintained at the site, to the home. If the Dflag is not set, the SEng does not change the SDir state.

When an *Iload-req* is received at the home of the modified memory block, the HEng checks the slot's presence bit in the memory. When the presence bit is 1, the HEng sends a protocol message to the modified site to request the block's most up-to-date data. When the presence bit is 0, the HEng either records the *Iload-req* in the home's corresponding deferred list if currently the deferred list is not empty, or sends a *fwd-request* to the modified site regarding the *Iload-req* if at the time the home's deferred list of the slot is empty.

When the modified site releases the exclusive ownership, the SEng is responsible for sending all the deferred lists maintained at the modified site to the block's home. Upon receipt of the transferred deferred lists, the home's HEng merges them with the home's deferred lists. All the deferred *Iloads* maintained at the home are not allowed to be transferred to other sites.
4.4 The ISC Scheme

This section describes the ISC scheme with a focus on I-structure operations. The ISC scheme assumes non-FIFO message passing, and adopts the forwarding policy.

4.4.1 Protocol Messages

Figure 4.7 gives the ISC protocol message specification. Compared with the Base scheme, the ISC scheme has five additional message fields: `slot, cont, Dflag, Dpack` and `fwd`. The slot field specifies the accessed slot’s index in the block to which the slot belongs. The cont field can be used to carry the continuation for an Iload operation, which usually contains the address of the next instruction that the AP will execute upon the receipt of the requested data\(^1\). When a site obtains the exclusive ownership, the site’s SEng also receives the Dflag knowledge for the empty I-structure slots. When a modified site releases the exclusive ownership, the site’s SEng places the deferred lists in the Dpack field of the message that is sent to the block’s home. The fwd field is used for forwarding, representing the site that a forwarded request originally comes from.

4.4.2 Home Engine

The HEng processes a protocol message from site \(k\) as follows.

Iload-req

- If the block’s HDir state is H-uncached or H-shared[\(S\)],

1. if the accessed slot’s presence bit is 0, the HEng records the Iload-req in the slot’s home deferred list, and sends an Iload-nil to site \(k\) to inform site \(k\)’s SEng that the requested data is not available. The block’s HDir state is not changed.

\(^1\)In *T-NG*, the continuation is an instruction pointer typically occupying one or two words. For performance reason, if the continuation size is too large, only a continuation identifier needs to be transferred, while leaving other information preserved at the site where the Iload operation was performed.
<table>
<thead>
<tr>
<th>Message Type</th>
<th>Message Fields</th>
<th>Issuing &amp; Receiving Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>load-req</td>
<td>command block from</td>
<td>req-site → home</td>
</tr>
<tr>
<td>store-req</td>
<td>command block from</td>
<td>req-site → home</td>
</tr>
<tr>
<td>iload-req</td>
<td>command block from slot cont</td>
<td>req-site → home</td>
</tr>
<tr>
<td>istore-req</td>
<td>command block from slot data</td>
<td>req-site → home</td>
</tr>
<tr>
<td>reclaim-req</td>
<td>command block from slot</td>
<td>req-site → home</td>
</tr>
<tr>
<td>fwd-load-req</td>
<td>command block fwd</td>
<td>home → mod-site</td>
</tr>
<tr>
<td>fwd-store-req</td>
<td>command block fwd</td>
<td>home → mod-site</td>
</tr>
<tr>
<td>fwd-iload-req</td>
<td>command block slot cont fwd</td>
<td>home → mod-site</td>
</tr>
<tr>
<td>fwd-istore-req</td>
<td>command block slot data fwd</td>
<td>home → mod-site</td>
</tr>
<tr>
<td>fwd-reclaim-req</td>
<td>command block slot fwd</td>
<td>home → mod-site</td>
</tr>
<tr>
<td>invalidate-req</td>
<td>command block</td>
<td>home → shr-site</td>
</tr>
<tr>
<td>collect-req</td>
<td>command block slot</td>
<td>home → shr-site</td>
</tr>
<tr>
<td>ilresume-req</td>
<td>command block slot data cont</td>
<td>home → def-site</td>
</tr>
<tr>
<td>load-ack</td>
<td>command block data</td>
<td>own-site → req-site</td>
</tr>
<tr>
<td>store-ack</td>
<td>command block data Dflag</td>
<td>own-site → req-site</td>
</tr>
<tr>
<td>iload-ack</td>
<td>command block slot data cont</td>
<td>own-site → req-site</td>
</tr>
<tr>
<td>iload-nil</td>
<td>command block slot cont</td>
<td>own-site → req-site</td>
</tr>
<tr>
<td>istore-ack</td>
<td>command block slot data</td>
<td>own-site → req-site</td>
</tr>
<tr>
<td>reclaim-ack</td>
<td>command block slot</td>
<td>own-site → req-site</td>
</tr>
<tr>
<td>fwd-load-ack</td>
<td>command block slot data Dpack</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>fwd-load-nak</td>
<td>command block</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>fwd-load-neg</td>
<td>command block</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>fwd-store-ack</td>
<td>command block Dpack</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>fwd-store-nak</td>
<td>command block</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>fwd-store-neg</td>
<td>command block</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>fwd-iload-ack</td>
<td>command block slot data Dpack</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>fwd-iload-nil</td>
<td>command block slot</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>fwd-iload-neg</td>
<td>command block slot</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>fwd-istore-ack</td>
<td>command block slot data Dpack</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>fwd-istore-nak</td>
<td>command block slot</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>fwd-istore-neg</td>
<td>command block slot</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>fwd-reclaim-req</td>
<td>command block slot</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>fwd-reclaim-ack</td>
<td>command block slot</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>fwd-reclaim-nak</td>
<td>command block slot</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>invalidate-ack</td>
<td>command block</td>
<td>shr-site → home</td>
</tr>
<tr>
<td>collect-ack</td>
<td>command block slot</td>
<td>shr-site → home</td>
</tr>
<tr>
<td>ilresume-ack</td>
<td>command block slot</td>
<td>def-site → home</td>
</tr>
<tr>
<td>clean-inf</td>
<td>command block data Dpack</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>kill-inf</td>
<td>command block data Dpack</td>
<td>mod-site → home</td>
</tr>
<tr>
<td>Iupdate-inf</td>
<td>command block slot data Dpack</td>
<td>mod-site → home</td>
</tr>
</tbody>
</table>

Figure 4.7: ISC Protocol Message Specification
2. if the accessed slot’s presence bit is 1, the HEng sends an *Iload-ack* to site *k* to supply the requested data. The block’s HDir state is changed to H-shared[*S'*], where *S' = {k} or *S U{k}*, depending on the block’s current HDir state.

• If the block’s HDir state is H-modified[*m*],

1. if the slot’s home deferred list is not empty, the HEng appends the *Iload-req* in the deferred list, and sends an *Iload-nil* to site *k* to inform site *k*’s SEng that the requested data is not available. The block’s HDir state is not changed.

2. if the slot’s home deferred list is empty, the HEng sends a *fwd-Iload-req* to site *m* to forward the *Iload-req* to the remote site. The block’s HDir state is changed to H-forward-in-progress. Thereafter there are five possible cases:

(a) The forwarded request is serviced in the modified site, and the modified site’s SEng sends a *fwd-Iload-ack* to the home. After receiving this positive reply, the home’s HEng updates the memory with the most up-to-date data, and changes the block’s HDir state to H-shared[*S*], where *S = {m, k}*.

(b) The forwarded request cannot be serviced in the modified site since the accessed slot’s presence bit is 0, and the modified site’s SEng sends a *fwd-Iload-nil* to the home. After receiving this negative reply, the home’s HEng records the deferred *Iload-req* in the slot’s home deferred list, and changes the block’s HDir state to H-shared[*m*].

(c) Before the forwarded request arrives at the modified site, the modified cache line has been cleaned or killed, and the modified site’s SEng sends a *clean-inf* or *kill-inf* to the home to write the most up-to-date data to the memory.

(d) In the modified site, before the forwarded request arrives, the AP performs an *Istore* operation on a slot whose Dflag is 1, thereby causing an *Iupdate-inf* to be sent to the home to write the modified cache line to the memory.
(e) In the modified site, when the forwarded arrives, the SEng cannot deduce whether the exclusive ownership has not been received or has been released, therefore it sends a \textit{fwd-Iload-neg} to the home.

\textbf{Istore-req}

- If the block’s HDir state is H-uncached or H-shared[S], and the accessed slot’s presence bit is 0, the HEng writes the data to the slot, and sets the slot’s presence bit to 1. Meanwhile, the HEng sends an \textit{Istore-ack} to site \(k\) to acknowledge the \textit{Istore-req} and if necessary, give site \(k\) a shared copy of the block. The HEng then checks the slot’s home deferred list,

1. if the home deferred list is empty, the HEng changes the HDir state to H-shared\([S']\), where \(S' = \{k\} \) or \(S \cup \{k\}\), depending on the block’s current HDir state.

2. if the home deferred list is not empty, the HEng resumes all the deferred Iloads for the slot, and changes the block’s HDir state to H-broadcast-in-progress\([S']\), where \(S' = \{k\} \cup R\) or \(S \cup \{k\} \cup R\) (\(R\) is the set of sites that the \textit{Iresume-reqs} are sent), depending on the block’s current HDir state. Later when all the \textit{Iresume-reqs} have been acknowledged, the HEng alters the block’s HDir state to H-shared\([S']\).

- If the block’s HDir state is H-modified\([m]\), the HEng sends an \textit{fwd-Istore-req} to site \(m\) to forward the \textit{Istore-req} to the modified site. The block’s HDir state is changed to H-forward-in-progress.

\textbf{reclaim-req}

- If the block’s HDir state is H-uncached, the HEng resets the slot’s presence bit to 0, and sends a \textit{reclaim-ack} to site \(k\) to acknowledge the \textit{reclaim-req}. The block’s HDir state is not changed.
• If the block’s HDir state is H-shared[S], the HEng suspends the \textit{reclaim-req}, and sends \textit{collect-reqs} to all the sites specified in \( S \). The block’s HDir state is changed to H-multicast-in-progress[S], with a counter maintained in the scratch field to record the number of collections that are being performed.

Later when all the collections have been acknowledged, the HEng resumes the suspended reclaim-req, and sends a \textit{reclaim-ack} to site \( k \) to inform site \( k \)’s SEng that the slot has been reclaimed. The block’s HDir state is changed to H-shared[S].

• If the block’s HDir state is H-modified[m], the HEng sends a \textit{fwd-reclaim-req} to site \( m \) to forward the \textit{reclaim-req} to the modified site. The block’s HDir state is changed to H-forward-in-progress.

\textbf{Iupdate-inf}

• If the block’s HDir state is H-modified, the HEng updates the memory with the most up-to-date data, merges the deferred lists in the \textit{Iupdate-inf}’s Dpack field with the corresponding home deferred lists, and then resumes the deferred \textit{Iloads} for the slot. The block’s HDir state is changed to H-broadcast-in-progress[S], where \( S = \{m\} \cup R \) (\( R \) is the set of sites that the \textit{Iresume-reqs} are sent to).

\subsection{4.4.3 Site Engine}

The SEng processes an I-structure operation as follows.

\textbf{Iload}

• If the cache line’s SDir state is C-invalid, the SEng suspends the \textit{Iload} instruction, and sends an \textit{Iload-req} to the home to request the data. The SDir state is changed to C-req-in-progress.

• If the cache line’s SDir state is C-shared,

1. if the accessed slot’s presence bit is 0, the SEng suspends the \textit{Iload} instruction, and sends an \textit{Iload-req} to the home to request the data, The SDir state is
changed to C-req-in-progress.

2. if the accessed slots' presence bit is 1, the SEng supplies the requested data from the cache. The SDir state is not changed.

- If the cache line’s SDir state is C-modified,

1. if the accessed slot’s presence bit is 0, the SEng defers the \textit{Iload} instruction, and records it in the corresponding site deferred list. The cache line’s SDir state is not changed.

2. if the accessed slot’s presence bit is 1, the SEng supplies the requested data from the cache. The SDir state is not changed.

\textbf{Istore}

- If the cache line’s SDir state is C-invalid or C-shared, the SEng suspends the \textit{Istore} instruction, and sends an \textit{Istore-req} to the home to write the data. The SDir state is changed to C-req-in-progress.

- If the cache line’s SDir state is C-modified, and the accessed slot’s presence bit is 0, the SEng writes the data to the slot, and resets the slot’s presence bit to 1. Meanwhile, the SEng resumes all the deferred \textit{Iloads} maintained in the slot’s site deferred list. Then the SEng checks the slot’s Dflag. If the Dflag is 1, the SEng sends an \textit{Iupdate-inf} to the home to write the most up-to-date data to the memory (all the remaining site deferred lists of the cache line are packed in the message), since currently the home deferred list of the slot is not empty. The SDir state is changed to C-shared.

\textbf{reclaim}

- If the cache line’s SDir state is C-invalid or C-shared, the SEng suspends the \textit{reclaim} instruction, and sends a \textit{reclaim-req} to the home to reclaim the slot. The SDir state is changed to C-req-in-progress.
• If the cache line’s SDir state is C-modified, the SEng resets the slot’s presence bit to 0. The SDir state is not changed.
Bibliography


Appendix A

Base Transition Rules

A.1 Home Engine Rules

Home_Engine ()
{
    msg = Extract (Input_Pool);
    while (1)
    {
        status = HEng_State_Transition (msg);
        switch (status) {
            case Transient: msg = Extract (Input_Pool);
                            break;
            case Stable: if ( Buffer_Queue[msg.block] != Null )
                          msg = Extract (Buffer_Queue[msg.block]);
                        else
                          msg = Extract (Input_Pool);
                        break;
        }
    }
}

HEng_State_Transition (msg)
{ /* msg = <command, block, from, data> */
    command = msg.command; /* what does msg want to do */
    f = msg.from; /* where does msg come from */
    b = msg.block; /* global block address */
    data = msg.data; /* memory block (cache data) content */

    switch (HDir[b].state) {
      case H_UNCACHED:
        switch (command) {
          case load_req: msg1 = gen_message (load_ack, b, Memory[b]);
        }
    }
}
send_message (msg1, f);
HD[Dir[b]].state = H_SHARED[f];
return (Stable);

case store_req:
    msg1 = gen_message (store_ack, b, Memory[b]);
    send_message (msg1, f);
    HD[b].state = H_MODIFIED[f];
    return (Stable);

    case clean_inf:
        break;
    case kill_inf:
        break;
    case fwd_load_ack:
        break;
    case fwd_store_ack:
        break;
    case fwd_store_nak:
        break;
    case invalidate_ack:
        break;
}

case H_SHARED[S]:
switch ( command ) {
    case load_req:
        msg1 = gen_message (load_ack, b, Memory[b]);
        send_message (msg1, f);
        HD[b].state = H_SHARED[S+f];
        return (Stable);
    case store_req:
        c = Num_of_Sites_in_Set (S);
        for ( s in S ) {
            msg1 = gen_message (invalidate_req, b);
            send_message (msg1, s);
        }
        HD[b].state = H_multicast_in_progress[c, msg];
        return (Transient);
    case clean_inf:
        break;
    case kill_inf:
        break;
    case fwd_load_ack:
        break;
    case fwd_store_ack:
        break;
    case fwd_store_nak:
        break;
    case invalidate_ack:
        break;
}

case H_MODIFIED[m]:
switch ( command ) {
    case load_req:
        msg1 = gen_message (fwd_load_req, b, f);
        send_message (msg1, m);
        HD[b].state = H_forward_in_progress[m, msg];
        return (Transient);
    case store_req:
        msg1 = gen_message (fwd_store_req, b, f);
send_message (msgl, m);
HDir[b].state = H_forward_in_progress[m, msg];
return (Transient);

case clean_inf: Memory[b] = data;
HDir[b].state = H_SHARED[{m}];
return (Stable);
case kill_inf: Memory[b] = data;
HDir[b].state = H_UNCACHED;
return (Stable);

case fwd_load_ack: break;
case fwd_store_ack: break;
case fwd_store_nak: break;
case invalidate_ack: break;
}

case H_forward_in_progress[m, suspend]:
switch ( command ) {
    case load_req: HEng_Sleep (b, msg); return (Transient);
    case store_req: HEng_Sleep (b, msg); return (Transient);
    case clean_inf:
        Memory[b] = data;
        if ( suspend.command == load_req ) {
            msg1 = gen_message (load_ack, b, data);
            send_message (msg1, suspend.from);
            HDir[b].state = H_shared[{m,suspend.from}];
            return (Stable);
        } else {
            HDir[b].state = H_nak_impending[suspend];
            return (Transient);
        }
    case kill_inf:
        Memory[b] = data;
        if ( suspend.command == load_req ) {
            msg1 = gen_message (load_ack, b, data);
            send_message (msg1, suspend.from);
            Hdir[b].state = H_shared[{suspend}];
            return (Stable);
        } else {
            HDir[b].state = H_nak_impending[suspend];
            return (Transient);
        }
    case fwd_load_ack:
        /* suspend.command == load_req */
        Memory[b] = data;
case fwd_store_ack: /* suspend.command == store_req */
    msg1 = gen_message (store_ack, b, data);
    send_message (msg1, suspend.from);
    HDir[b].state = H_MODIFIED[suspend.from];
    return (Stable);
    break;

    case fwd_store_nak:
    break;

    case invalidate_ack:
    break;

    case H_multicast_in_progress[count, suspend]:
    switch (command) {
    case load_req: HEng_Sleep (b, msg); return (Transient);
    case store_req: HEng_Sleep (b, msg); return (Transient);

    case clean_inf:
    break;

    case kill_inf:
    break;

    case fwd_load_ack:
    break;

    case fwd_store_ack:
    break;

    case invalidate_ack: /* suspend.command == store_req */
    if (count > 1) {
        HDir[b].state = H_multicast_in_progress [count-1, suspend];
        return (Transient);
    }
    else {
        msg1 = gen_message (store_ack, b, Memory[b]);
        send_message (msg1, suspend.from);
        HDir[b].state = H_MODIFIED[suspend.from];
        return (Stable);
    }
}

case H_nak_impending[suspend]
    switch (command) {
    case load_req: HEng_Sleep (b, msg); return (Transient);
    case store_req: HEng_Sleep (b, msg); return (Transient);

    case clean_inf:
    break;

    case kill_inf:
    break;
case fwd_load_ack:    break;
case fwd_store_ack:    break;
case invalidate_ack:  break;

    case fwd_store_nak:
        /* suspend.command == store_req */
        msg1 = gen_message (store_ack, b, Memory[b]);
        send_message (msg1, suspend.from);
        HDir[b].state = H_MODIFIED[suspend.from];
        return (Stable);
    }
}
HEng_Exception_Handler ();
}
A.2 Site Engine Rules

SEng_State_Transition (msg)
{ /* msg = <command, block, data, fwd, chunk> */
  command = msg.command; /* what does msg want to do */
  b = msg.block; /* global block address */
  data = msg.data; /* cache data (memory block) content */
  fwd = msg.fwd; /* forwarding site */
  home = Home_Site (b); /* where the home is */

  switch ( SDir[b].state ){
    case C_INVALID:
      switch ( command ) {
        case LOAD: Schedule_Next_Instruction (msg, Miss);
          msgl = gen_message (load_req, b, cur_site);
          send_message (msgl, home);
          SDir[b].state = C_req_in_progress[msg];
          return (Transient);
        case STORE: Schedule_Next_Instruction (msg, Miss);
          msgl = gen_message (store_req, b, cur_site);
          send_message (msgl, home);
          SDir[b].state = C_req_in_progress[msg];
          return (Transient);
        case DELETE: break;
        case KILL: break;
        case CLEAN: break;
        case load_ack: break;
        case store_ack: break;
        case fwd_load_req: /* cache data been pushed out */
          SDir[b].state = C_INVALID;
          return (Stable);
        case fwd_store_req: /* cache data been pushed out */
          msgl = gen_message (fwd_store_nak, b);
          send_message (msgl, home);
          SDir[b].state = C_INVALID;
          return (Stable);
        case invalidate_req: /* cache data been pushed out */
          msgl = gen_message (invalidate_ack, b);
          send_message (msgl, home);
          SDir[b].state = C_INVALID;
          return (Stable);
      }
  }
}
case C_SHARED:
    switch (command) {
    case LOAD:     Schedule_NextInstruction (msg, Cache[b]);
                    SDir[b].state = C_SHARED;
                    return (Stable);
    case STORE:    Schedule_NextInstruction (msg, Miss);
                    msg1 = gen_message (store_req, cur_site, b);
                    send_message (msg1, home);
                    SDir[b].state = C_req_in_progress[msg];
                    return (Transient);
    case DELETE:   SDir[b].state = C_INVALID;
                    return (Stable);
    case KILL:     break;
    case CLEAN:    break;
    case load_ack: break;
    case store_ack: break;

    case fwd_load_req:    /* ownership been sent back home */
                        SDir[b].state = C_SHARED;
                        return (Stable);
    case fwd_store_req:   /* ownership been sent back home */
                        FLUSH (b);
                        msg1 = gen_message (fwd_store_nak, b);
                        send_message (msg1, home);
                        SDir[b].state = C_INVALID;
                        return (Stable);
    case invalidate_req:  FLUSH (b);
                        msg1 = gen_message (invalidate_ack, b);
                        send_message (msg1, home);
                        SDir[b].state = C_INVALID;
                        return (Stable);
    }

case C_MODIFIED:
    switch (command) {
    case LOAD:     Schedule_NextInstruction (msg, Cache[b]);
                    SDir[b].state = C_MODIFIED;
                    return (Stable);
    case STORE:    Write_Chunk_In_Cache (b, msg.chunk);
                    Schedule_NextInstruction (msg, Hit);
                    SDir[b].state = C_MODIFIED;
                    return (Stable);
    case DELETE:   break;
    }
case KILL:
    FLUSH (b);
    msg1 = gen_message (kill_inf, b, Cache[b]);
    send_message (msg1, home);
    SDir[b].state = C_INVALID;
    return (Stable);

case CLEAN:
    FLUSH_WITH_CLEAN (b);
    msg1 = gen_message (clean_inf, b, Cache[b]);
    send_message (msg1, home);
    SDir[b].state = C_SHARED;
    return (Stable);

break;

break;

FLUSH_WITH_CLEAN (b);
msg1 = gen_message (fwd_load_ack, b, Cache[b]);
send_message (msg1, home);
SDir[b].state = C_SHARED;
return (Stable);

FLUSH (b);
msg1 = gen_message (fwd_store_ack, b);
send_message (msg1, home);
SDir[b].state = C_INVALID;
return (Stable);

break;

break;

/* suspend.command == LOAD */
Cache[b] = data;
Schedule_Suspended_Instruction (suspend, Cache[b]);
SDir[b].state = C_SHARED;
return (Stable);

/* suspend.command == STORE */
Cache[b] = data;
Write_Chunk_In_Cache (b, suspend.chunk);
Schedule_Suspended_Instruction (suspend, Hit);
SDir[b].state = C_MODIFIED;
return (Stable);

case fwd_load_req:
    SDir[b].state = C_req_in_progress[suspend];
    return (Transient);

case fwd_store_req:
    FLUSH (b);
    msg1 = gen_message (fwd_store_nak, b);
    send_message (msg1, home);
    SDir[b].state = C_req_in_progress[suspend];
    return (Transient);

case invalidate_req:
    FLUSH (b);
    msg1 = gen_message (invalidate_ack, b);
    send_message (msg1, home);
    SDir[b].state = C_req_in_progress[suspend];
    return (Transient);

} SEng_Exception_Handler ();
}