MESIF Cache Coherence Protocol

Andrew Willam Hay

under the supervision of

Professor James Goodman and Professor Gill Dobbie

A thesis submitted in fulfillment of the requirements
for the degree of Master of Software Engineering in Engineering,
The University of Auckland, 2012.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor James Goodman. Working our way through the protocol design, always honest in his critiques of my ideas, forcing me to be precise in my thoughts.

My other supervisor, Professor Gill Dobbie, patiently helped me with many administration issues which were mostly my fault. Her help has made this work much smoother than it would have been without her.

As always, I thank my brother with his careful reading of the drafts of this thesis. His comments have improved this thesis substantially. Final proof reading by my family was greatly appreciated.

Contents

List of Tables

List of Figures

Abstract

Cache coherence protocols help coordinate memory accesses to shared memory in multicore systems. The two traditional protocols, snoopy and directory protocols both have strengths and weaknesses. Snoopy protocols have fast two hop cache-to-cache latency but poor scalability due to reliance on bus based interconnect. Directory protocols are more scalable by utilizing an unordered interconnect but have slower three hop cache-to-catch latency. There are protocols, such as *Token Coherence* that attempt to get the best of both snoopy and directory protocols, but cannot guarantee requests can succeed without retrying or backoff to a slower, but guaranteed to succeed request.

With the advent of high speed high bandwidth Intel® QuickPath Interconnect (Intel QPI), new cache coherence protocols can be developed that can exploit the increased bandwidth available.

We present *MESIF*, a cache coherence protocol which uses the abundant bandwidth of QPI to ensure coherence using an unordered interconnect and two hop cache-to-cache latency without backoff and retry of requests. By utilizing a unordered interconnect, *MESIF* is more scalable that protocols that require a bus based interconnect. *MESIF* only requires one extra bit per cache block in local caches.

We also present a number of extensions to the *MESIF* protocol. The two main extensions are first, allowing for concurrent read requests to execute without conflicts. The second extension is to allow for optional write-updating, even though *MESIF* is a write-invalidation cache coherence protocol.

We implemented *MESIF* in the Wisconsin Multifacet GEMS SLICC language for cache coherence protocols. While we do not have presentable results, this implementation will help in future work on evaluating how the *MESIF* protocol compares to other cache coherence protocols.

Chapter 1

Introduction

As it is becoming more difficult to improve single core performance, chip makers have turned to producing increasingly multicore chips[5]. To take advantage of this trend, programmers must write parallel programs that execute on multiple cores simultaneously.

The most commonly used paradigm for multicore programs is the *shared memory* paradigm. In shared memory, all cores logically share the same memory space and can simultaneously read and write to the same memory locations.

Complicating the issue of shared memory are caches. Caches store memory locations values so that cores will have decreased latency (quicker access times) when accessing 'cached' memory values. However with caches in a multicore systems, a read may have to search through other core's caches to find the memory location value, and a write may have to change the value of the memory location in these other caches.

A memory location is said to be *coherent* if memory accesses (reads and writes) follow the semantics for reading and writing in a single core. Reads return the value of the previous write to the memory location, and that writes have a sequential order. Ensuring a memory location is coherent allows cores to safely execute read and write accesses.

Coherence for accesses to all shared memory locations can be difficult with multiple cores accessing simultaneously, memory locations cached in various caches in the system. A cache coherence protocol (or simply, protocol) is used to ensure coherence for all shared memory locations. There are many different cache coherence protocols, but they all achieve coherency by describing when cores are allow to read and/or write a memory location, and how write values are distributed to the caches in the multicore system.

This thesis presents for the first time to the public domain the *MESIF* cache coherence protocol. The *MESIF* protocol is a scalable two hop cache-to-cache latency for memory requests with no backoff and retry of requests. We show how *MESIF* achieves coherence for memory locations. We also introduce a number of extensions to the *MESIF* protocol.

Part of this work involving implementing the *MESIF* protocol[1]. Appendices A, B, and C, contain the implementation code. The current implementation is only the *MESIF* protocol as presented in Chapter 3, and does not include any of the extensions to *MESIF* described in Chapter 4.

While at the time of writing there were no presentable results, this implementation will be used in future work on evaluating how the *MESIF* protocol compares to other cache coherence protocols.

This chapter briefly describes the history behind the *MESIF* cache coherence protocol and present the contributions of this thesis.

1.1 History of the *MESIF* Protocol

The *MESIF* protocol was developed internally in Intel in 2001[6, 7]. Subsequent work resulted in a similar protocol being developed, the Intel® QuickPath Interconnect (Intel QPI) cache coherence protocol. The Intel QPI protocol was ultimately incorporated into the latest Intel products with the new point-to-point interconnect composed of Intel QPI links. However to date, there has been no publicly available document describing and analyzing the *MESIF* cache coherence protocol in detail nor the Intel QPI cache coherence protocol.

Because we do not have access to the description of the QPI protocol, we cannot use it as a comparison to our *MESIF* protocol. Instead, we use the *Token Coherence* protocol, which is similar in many ways to the *MESIF* protocol.

The *MESIF* protocol presented in this thesis is based off of a technical report[6, 7]. This technical report contains the high level ideas of *MESIF* but misses out on a large number of details required to implement the protocol. This thesis fills in the many of the missing details, demonstrating how *MESIF* achieves coherence, especially with in the complex situation of multiple conflicting memory requests. Chapter 3 describes this *MESIF* protocol, and Chapter 4 discuss our extensions to *MESIF*.

There exists another *MESIF* cache coherence protocol in the public domain[9]. This protocol is more similar to to the QPI protocol than the *MESIF* protocol presented in this thesis, and we do not discuss it further.

1.2 Contributions

This thesis makes the following contributions:

[1]The implementation is coded in the SLICC (Specification Language including Cache Coherence) language, a part of the Wisconsin Multifacet GEMS[14] project.

- **Introduces for the first time to the public domain the *MESIF* cache coherence protocol.** *MESIF* is the first broadcasting cache coherence protocol built on a point-to-point interconnect.

- **Refines the ideas of *MESIF* into a fully realizable cache coherence protocol.** The protocol is based on a technical report[6, 7] containing the main ideas of *MESIF* but without going into detail how the protocol works. The most important detail missing that is answered in this thesis is how conflicting requests are resolved. We also demonstrate how coherence is achieved.

- **Introduces four extensions to the *MESIF* protocol.** The first extension is to allow for non-conflicting simultaneous read requests as multiple read request in the *MESIF* protocol are conflicting. The second extension is an optional write-update (*MESIF* is write-invalidate) mode for write requests. The third extension reduces the number of invalidations when *MESIF* handles conflicting requests. The last extension speeds up the completion of some types of requests.

- **Implements the *MESIF* protocol in Wisconsin Multifacet GEMS[14] simulator.** We implemented the *MESIF* protocol in SLICC, a language for writing cache coherence protocols.

In the next chapter, we go into detail about our model of coherence. We introduce cache coherence protocols in general including the interconnect networks used. We discuss the two traditional cache coherence protocols, snoopy and directory protocols, as well as *Token Coherence*. These protocols, and especially the trends in interconnect, motivate a new type of protocol that utilizes a point-to-point interconnect, but guarantees 2-hop latency for cache-to-cache misses.

In Chapter 3 we describe the *MESIF* cache coherence protocol in detail, and describe how *MESIF* ensures coherence in an unordered point-to-point interconnect network with broadcasting. In Chapter 4 we introduce four extensions we developed for the *MESIF* protocol.

We conclude our thesis with Chapter 5 and discuss future work.

Chapter 2

Coherence and Cache Coherence Protocols

In this chapter we describe our model of coherence and how cache coherence protocols achieve coherence. Cache coherence protocols are used in multicore systems to help coordinate memory operations.

In the next section we describe our model of coherence. This model is inspired by the work of D. Sorin[16] and considers an ordering of memory accesses (reads and writes) to the same memory location to be a sequence of single writer followed by multiple reader periods. The invariants presented there will be used in the next chapter to demonstrate how our protocol, *MESIF*, achieves coherence.

We then discuss cache coherence protocols in general. We describe what is a shared memory multicore system (Section 2.2.1), cache states (Section 2.2.2), performance attributes of cache coherence protocols (Section 2.2.3), and the interconnection network used for inter cache communication (Section 2.2.4). The interconnection network is pivotal to the design of the *MESIF* protocol as it was the development of high speed high bandwidth Intel QPI interconnect that originally spurred the development of *MESIF*.

In Section 2.2.5. We discuss the two traditional cache coherence protocols, snoopy and directory protocols. One of *MESIF*'s design goals is to have the latency comparable to a snoopy protocol but without the scalability issues of a snoopy protocol.

We also introduce the *Token Coherence*[12] protocol in Section 2.2.6. *Token Coherence* is similar to *MESIF* but does not rely on broadcast messages to guarantee requests complete. Instead, *Token Coherence* wanted the freedom to not have to broadcast to all caches to save bandwidth; broadcasting on all requests, as *MESIF* does, can use a lot of bandwidth.

So *transient requests* in *Token Coherence* can fail to complete, requiring retry or eventual backoff to a slower but guaranteed to succeed *persistent request.* In the average case, transient requests do complete and this results in *Token Coherence*'s two hop cache-to-cache latency. Conversely, *MESIF* guarantees all requests will complete without retry or backoff.

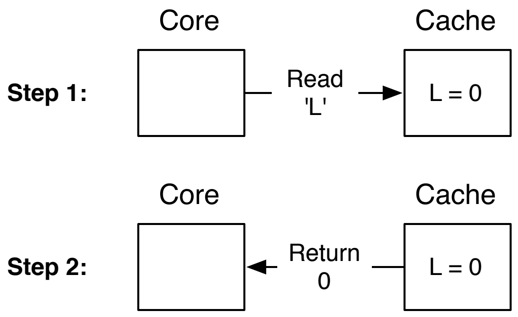
How *Token Coherence* protocol handles transient requests was used as inspiration for extending *MESIF* to handle simultaneous read requests without the requests conflicting, see Section 4.1 for details.

2.1 Coherence

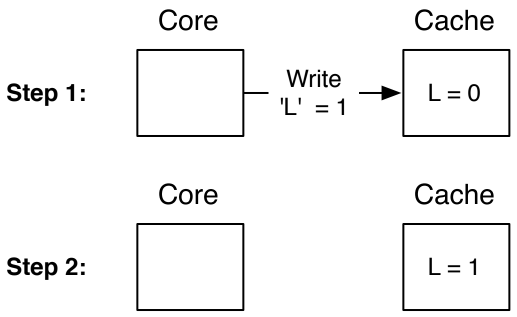
To understand coherence memory operations for memory locations, we first give an example of the problem of coherence and a simple solution. We then define our model of coherence for memory locations.

2.1.1 Coherence Example

When a single core is reading and writing a single memory location stored in the core's cache, we can see what behavior we expect out of reads and writes. Reads return the value in stored under that memory location in the cache. Figure 2.1a shows how a read by a core returns the value stored in the cache. Writes change the value stored in that memory location. Figure 2.1b shows how a write results in the value stored in the cache changing.



(a) *Core first sends read 'L' to the cache, which returns the current value of memory location 'L'.*



(b) *Core sends write 'L' writing value* 1*. Cache updates current value of memory location 'L' to be* 1

Figure 2.1: *How a core's read and write accesses work with a cache.*

The example program used in this section is that one core is writing two different values to a memory location *L*. First it writes the value 1 and then the value 0. The goal of the program is to observe and report the order of the writes.

Reads observe the value written, returning the value written by the 'last' write. For a single core system running this program, Figure 2.2 shows the timeline of the read and write requests. This core observes (with the two read requests) that the value is first 1 and then 0. Each read returns the value of the last write as we expect.

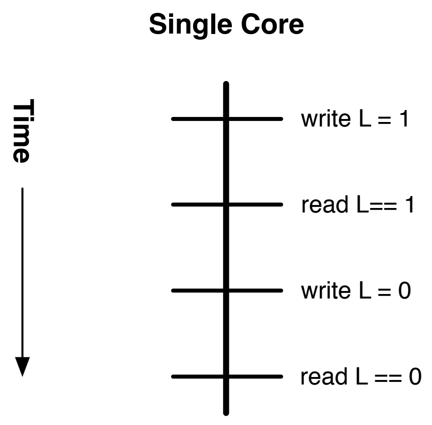


Figure 2.2: *Timeline of read and writes by the single core to memory location L. Each read returns the value written by the last write.*

Consider a three core system executing a similar program in parallel. In this multicore program, we have one of the three cores writing two values to a memory location L, and the other two cores will read L and observe and report the order of the writes. Each core has a cache which stores the value of a memory location, much like in a the single core system.

What the 'last' write is is not as simple as with a single core. A write in this multicore system must not only write the core's cache, but also the other core's caches. Figure 2.3 shows this multicore system. Each cache is joined together by an interconnection network, allowing caches to communicate (such as sending write values) with other caches.

The problem of coherence can be observed in Figure 2.4. The writing core updates its cached memory location as in a single core system. The writing core also sends a message to each of the other caches in the system, updating their cached memory location value. The reading caches observe the write value after each write message arrives.

Because of message delays in the interconnection network, the write messages to the second reading core arrive in the opposite order sent. Each core reads from the 'last' value written in its cache, but the two reading cores observe a different order for the writes: the second reading core disagrees with the writing core the order of the writes.

Accesses to shared memory location such as L to *must* agree on the order of writes, such

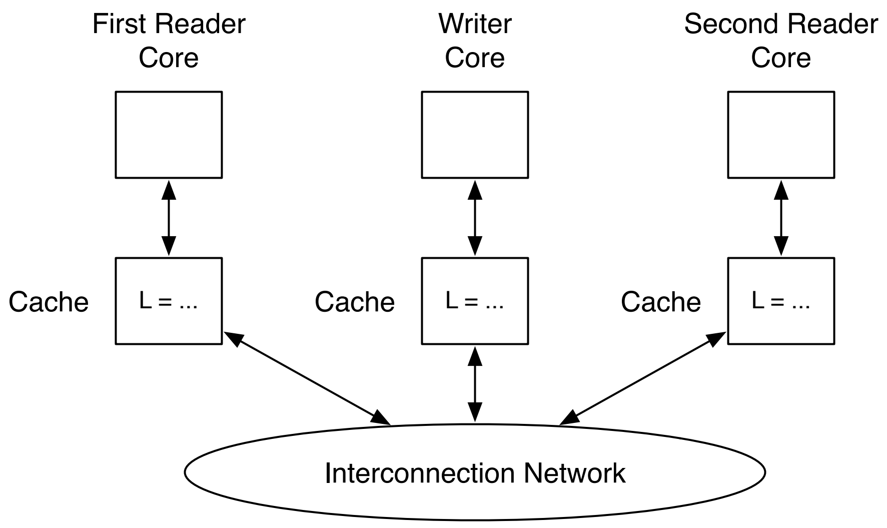


Figure 2.3: *The three core multicore system. Caches can communicate with one another via an interconnection network.*

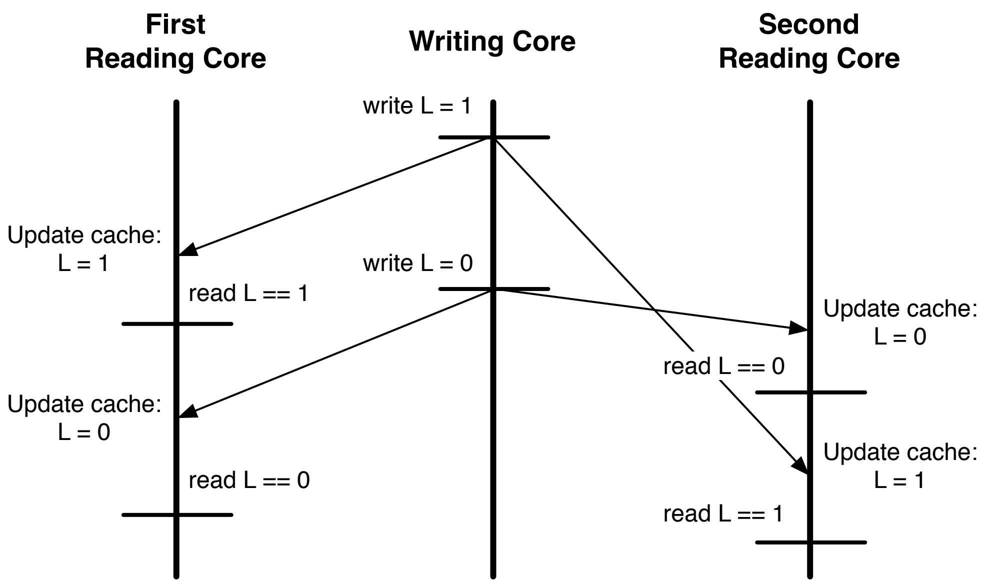


Figure 2.4: *Message delay results in the reader cores disagreeing on the order of the writes by the writing core.*

as in a single core system. The writing core sees the order of writes as first 1 then 0, so all other cores must see the write values in the same order. We call this *coherency*, or the memory accesses to a memory location are *coherent*[1].

One simple way to achieve coherency for accesses in this program is to have the writing core wait until all caches have received the first write value before sending the second write value. Figure 2.5 shows how this is achieved by each reading core sending an

[1]Reads still must read the last value written to be coherent.

acknowledgement message (ACK) to the writing core. When the writing core receives all $ACKS$ it can now send the second write value, assured that all cores will observe the writes in the same order; the accesses will be coherent.

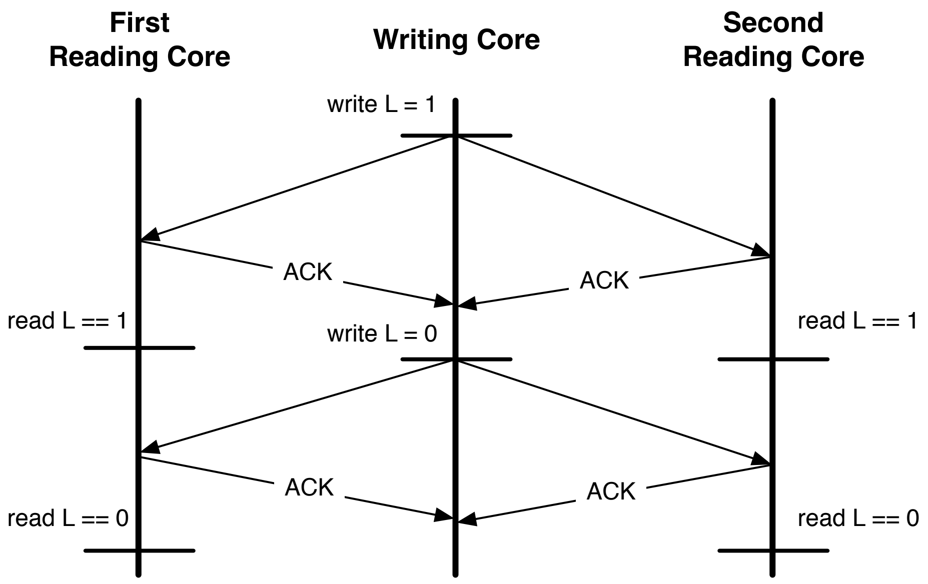


Figure 2.5: *By waiting until all caches are storing L as the write value, both reader cores agree on the order of writes by the writing core.*

In the next section we define our model of coherence based on a sequence of single writer-reader followed by (only) readers periods. When there are only readers, the value read is the last value written by the previous writer-reader. This would avoid the incoherent execution in Figure 2.4 as the first read of 0 by the second reading core would require any reads after this first read to either return 0, or a value written by a write *after* the write of 0.

Accesses may violate this, there may be a core writing at the same time as a different core reading, but as long as the outcome of such an execution of accesses is equivalent to an execution of accesses that either are all reading, or only one is writing and reading, then the accesses are coherent.

2.1.2 Coherence for Shared Memory Multicores

With multiple cores accessing shared memory locations simultaneously, the semantics for memory accesses on a single memory location applies to the accesses by all cores to a memory location, not just the accesses by a single core. Single cores in a multicore system expect that memory locations can change values even without locally writing it.

For example, if a single core continuously reads the same memory location, each read could return a different value due to writes by other cores (such as the reader cores in Figure 2.5). But if we consider all accesses to this location then the usual semantics will apply; reads will read the value of the previous write in the logical order and writes are ordered sequentially.

A single core executes memory accesses to a single memory location in-order and so the execution is correct. When we add caches and multiple cores accessing the same memory locations simultaneously, this may no longer be the case. Caches allow cores to read values written by previous writes and writes must update many caches. But not just any past value can be returned on a read as we will show.

We introduce two invariants[16] which define what is required for a parallel execution of memory accesses to be *coherent*. The invariants are:

Single-Write-Multiple-Readers: At any time, every memory location L has either one core that may write and read L (writer-reader period), or any number of cores that can only read L (readers period).

Data-Value: The value of a memory location is the previous value written by this core, if this core is writing. Or it is the last value written by the previous writing core, if there is no writer.

The first invariant is the *single-write-multiple-reader* (*SWMR*) invariant. For each memory location at any time there is either one writer (which can also read) or there are many readers (but none can write).

The semantics for memory accesses require reads return the value of the previous write to the memory location. Our second invariant is the *Data-Value* invariant, which states that the value of a read access is either the last value written by the previous writing core. Or in other words, the previous writer-reader.

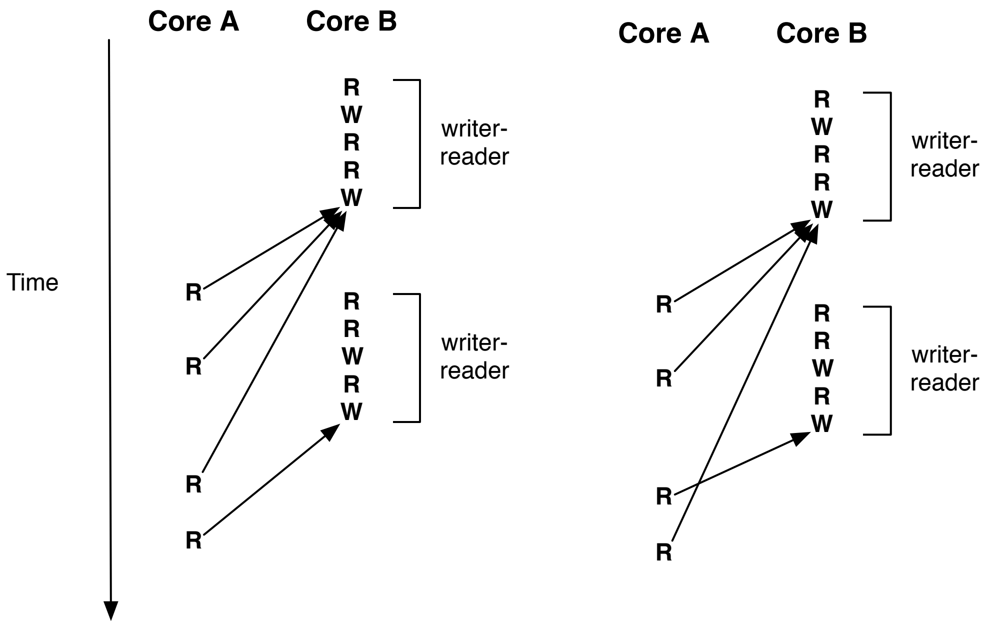
Or if the core is currently the writer-reader, and there exists a write by this core between the read access and the start of the writer-reader period, then the read returns the closest previous write value.

While cores still must execute accesses to a single memory location in-order, they do not need to return the last value written. A read can return the value written by a previous write, and not the latest write while still executing coherently.

Figure 2.6a shows two cores reading and writing the same location in time, with core B the only core writing. The reads by core A are *reordered* to *appear* to be immediately before the reader-writer period that each read is reading from.

Reordering of a read accesses means that the read accesses appear to have executed immediately after the write access that the read is reading from, even if between these two accesses is another write access.

It is not possible to read a later write value then read an earlier write value. Figure 2.6b shows how the reads by core A are reordered so that they appear before the write that the read is reading from. But the order of accesses by core A is now in a different order than the in-order (all cores execute accesses to the same memory location in-order) execution of these accesses. Figure 2.4 also gave an example where if we reorder the read accesses by the second reading core, the reads appear to execute in the opposite order than they actually executed in.



(a) *Coherent execution with the reads of core A reordered to ensure the* SWMR *and* Data-Value *invariants hold. The reads inside the writer-reader period in core B return the previous write value, which may be a write in this same writer-reader period.*

(b) *Incoherent execution as the reads of core A cannot be reordered without breaking core A's program order.*

Figure 2.6: *Coherent and incoherent executions. The order executed is shown directly and the reordering of the access into the is shown by the arrows. Thus the reads by core A are reordered to appear directly after the* writer-reader *epoch the reads are reading from.*

With caches it possible to read an old value of a memory location, as shown with the second to last read by core A in Figure 2.6a. Write accesses do not always execute atomically, such as shown in Figure 2.5 where the write takes some time update all caches values.

Memory access can be separated into when they have permission to execute (*retiring*) and when their effects are made visible to all cores (*committing*). We end this section on coherence by discussing caches and the separation of retiring and committing a memory access, and how this fits in with write-invalidation cache coherence protocols.

Coherence with Caches

By separating out retiring from committing a write access, it is possible to speed up execution of the access while keeping coherence. In *MESIF*, it is possible to retire a write access while there exists cores reading the memory location. If we had to always wait until a write committed, we would lose the opportunity to speed up the access. See Figure 3.6 for details.

Figure 2.7 shows an example of a multicore program (Figure 2.7a) and a coherent execution (Figure 2.7b). We also include the ordering of accesses by which period they belong to (Figure 2.7c) allowing us to see reordering of a read access.

Each core has a cache in which to store values of memory locations. In the next section we discuss caches in more detail, for now we assume that a read returns the value in the core's cache, and writes update the value in the core's cache immediately, and in the other core's caches eventually.

In this program, first core A and the core B execute their codeblock instructions, with the shared memory location L ensuring they do not execute the core blocks simultaneously. Initially the value of L is 0, so core A will execute its codeblock first. Core A then writes 0 to L, but this value is not propagated to core B's cache until time 5. Core B at time 4 reads the old value for L stored in its cache and at time 5 reads the new value for L.

Each memory access to location L can be ordered such that it is either in a readers period or a reader-writer period, which satisfies the *SWMR* invariant. The *Data-Value* invariant holds as the reads at time 1 and 4 read the original value, and the read at time 5 reads the value written at time 3. Finally, L is written again at time 7.

The example demonstrates that, due to caching, the reordering can change the executed order of memory accesses, unlike in a single core system (they execute accesses to the same memory location in-order). The affects of writes, such as the write by core A at time 3 in Figure 2.7b, can take some time to propagate to all cores. Core B only received the new value at time 5. Memory accesses do not always execute atomically.

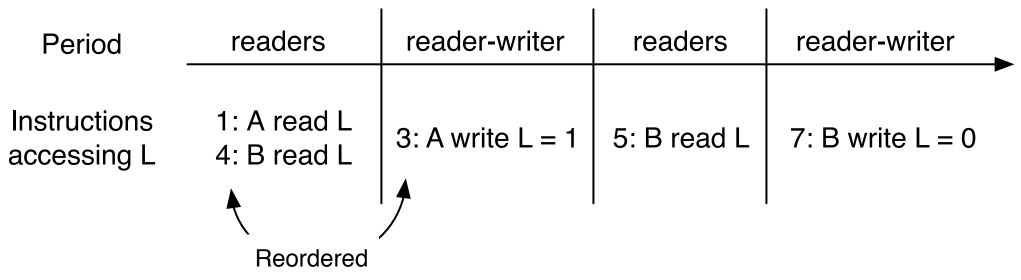
Memory accesses can be separated into when they have permission to locally execute the memory access (*retiring*) and when the effects of the memory access are made visible to all cores (*committing*). As read accesses are observations of write values, retiring and committing occur at the same time. Only a write access can separate retiring and committing.

	Core A		**Core B**
1	while(L == 1);	1	while (L == 0);
2	codeblock$_1$	2	codeblock$_2$
3	L = 0	3	L = 1

(a) *Program for each core. These instructions are executed in this sequential order.*

Time	Instruction	Value of L in *A's* cache	Value of L in *B's* cache
1	A: read L (0)	0	0
2	A: codeblock$_1$	0	0
3	A: write L = 1	1	0
4	B: read L (0)	1	0
5	B: read L (1)	1	1
6	B: codeblock$_2$	1	1
7	B: write L == 0	1	0

(b) *Order of execution of instructions and contents of caches.*



(c) *Period diagram with executed memory accesses in appropriate periods. Each instruction in Figure 2.7b shows which periods each access belongs to. Each access is denoted by the time it executed, as well as which core the access was executed in.*

Figure 2.7: *An example of a coherent execution. The read at time* 3 *is reordered behind the write at time* 2*.*

When a core retires a write access, the core may continue executing if it was delayed waiting for the memory access to complete. The core can continue executing even if the memory access has not yet committed and other cores are reading the old value for the memory location. As long as we can reorder the read accesses (that read the old value) of the other cores behind the write access, then we will have coherent execution. The write access *must wait to finish* until it has committed and all cores have seen the write's effects. We only allow for one write access to have retired but not yet committed.

Coherence in Write-Invalidate Cache Coherence Protocols

A write access can retire as long as it has permissions to do so, but must wait until the *effects* of the write access are visible to all cores, until the write access commits. These effects are that all other cores must be reading only the value written by this write access and not old values.

If a core wishes to execute another write access after the first, the core must again get permission to write (retire) and then make the effects of the write visible to all cores (commit). The effects of the write access are that no other core can be reading an old value, they must be reading the value written by this write (until another write access commits).

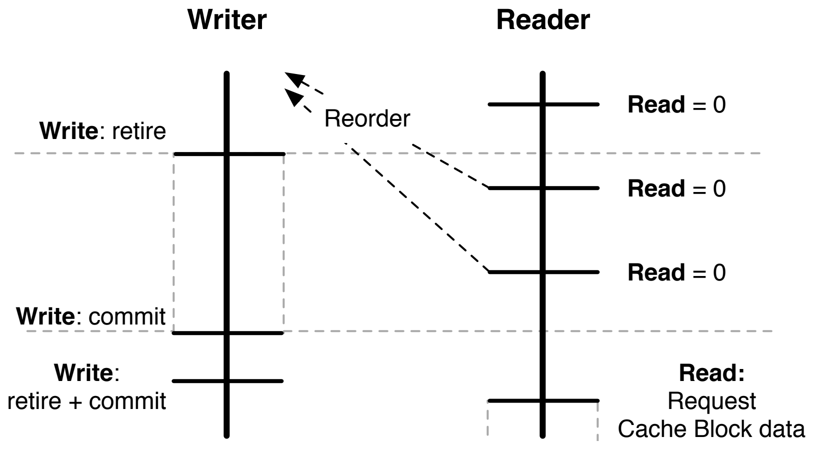
In a system with caches, the effect of the write is to either update the value of the memory location in all caches, or we can ensure no other caches can even read the memory location without requesting the memory location value again. The first option is called a *write-update* cache coherence protocol. The second is a *write-invalidate* cache coherence protocol as write requests *invalidate* copies of the memory location, preventing them from being read.

Figure 2.8 shows the difference between a write in a write-invalidate protocol (Figure 2.8a) and a write-update protocol (Figure 2.8b). The reordering of memory accesses before and after the write are shown as arrows between the read accesses in the reader cores and the write access in the writing core.

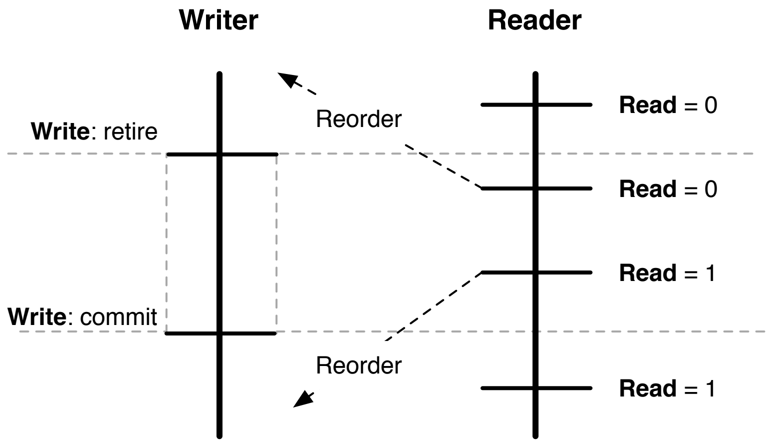
In a write-invalidate protocol, between retiring and committing there can be reads that will be reordered behind the write, but no reads reordering after the write. In write-update protocols, we can have reads both reordered before and after the write.

After a write in a write-invalidate protocol, the writing cache typically stores the information that there are no other readers out there. This writing core can then retire and commit at the same time; there are no readable copies of the memory location to update or invalidate.

In this section we have introduced the notion of coherence in multicore shared memory systems as two invariants: the *SWMR* and *Data-Value* invariants. Memory accesses can be separated into when they *retire* and can locally execute the memory access, and when the memory access *commits* making their effects visible to all cores.



(a) *Write-invalidate by commit ensure that no cores but the writing core can read the memory location. After invalidation and before any read requests from other cores, the writer can immediately retire and commit subsequent writes.*



(b) *Write-update by commit ensures that all cores are reading the new value, but individual cores may receive the new value before the write commits.*

Figure 2.8: *Cores reading and writing the same memory location. The old value is* 0 *and the value that is written is* 1.

In the next section we discuss cache coherence protocols, and how a more realistic system of cores and caches can achieve coherence. We discuss the properties of cache coherence protocols, including interconnection networks, and introduce two traditional protocols, snoopy and directory protocols, as well as the *Token Coherence* protocol.

2.2 Cache Coherence Protocols

To understand cache coherence protocols, we first must describes the components necessary for coherence in a multicore system.

2.2.1 System Components

Figure 2.9 shows the components of a multicore system relevant to this thesis. A *core* is a processing unit with a single thread of execution[2] that executes some multi-threaded program consisting of multiple sequence of instructions, each core runs one sequence of instructions.

Some of these instructions are *memory operations* (previous called memory accesses) which either read from or write to memory locations. Each core handles these memory operations using an associated finite state machine (FSM) called the *cache controller* which encodes the rules of the cache coherence protocol designed to achieve coherence.

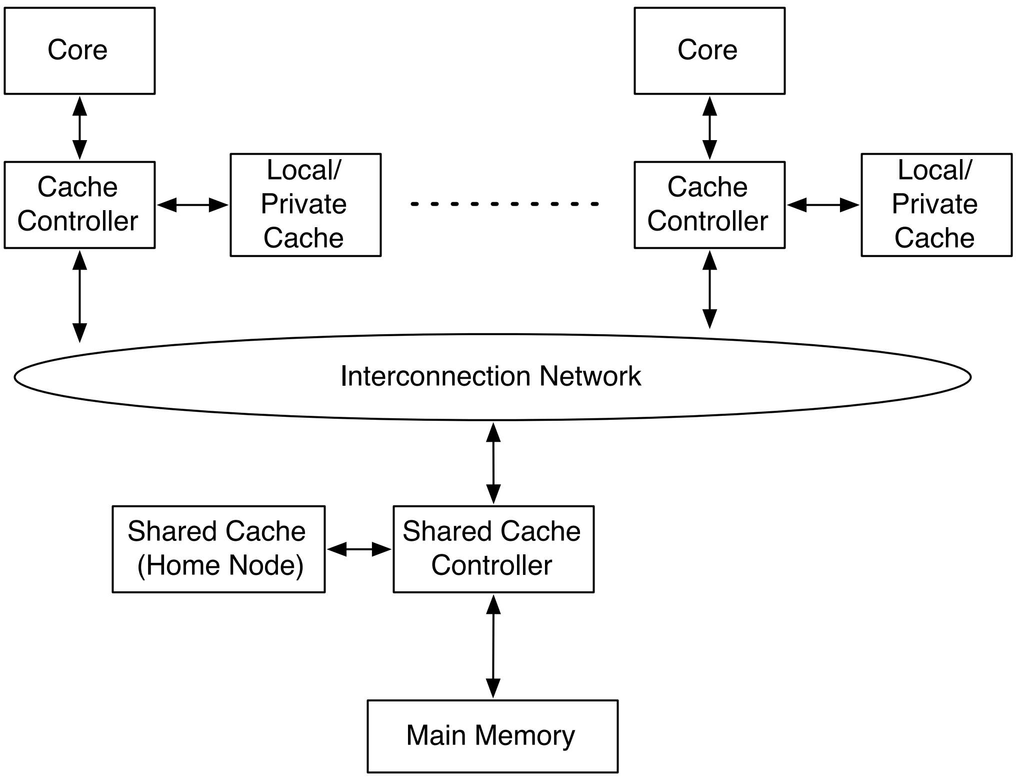


Figure 2.9: *Major system components in a multicore system.*

Each core has a local (low latency for the core to access) private (directly accessible only by the one core) cache to speed up memory operations. The cache consists of *cache blocks* which store memory location values.

A cache is typically not big enough to hold all the data a program requires, so cache blocks must be *evicted* to make room for new cache blocks if the cache becomes too full. However, systems using *write-invalidate* cache coherence protocols store the write values

[2]This is the assumption for this thesis.

in the cache, and do not immediately update memory on writes[3]. When evicting a *dirty* cache block (a cache block that has been written to) a cache initiates a *write-back* of the cache block to main memory, updating memory with the current value.

The cache controller can either service a memory operation from the local and private cache given correct permissions. If the cache does not have the right permissions to commit the request now, the cache controller must request the memory location from another cache or main memory. This produces a *memory request*. This memory request is sent on the interconnection network, which eventually results in the correct permissions and cache block data to service the memory operation.

The system as a whole consists of multiple copies of a core with its local and private cache and cache controller. All cache controllers are joined by an interconnection network and can send messages and cache block data to each other. Additionally, there usually exists a *shared cache*, which all cache controllers can access, along with an associated cache controller.

To assist in memory requests each cache block has an unique *home node* in the system. This home node typically is located at the the shared cache. The home node handles sending write data and issuing read requests to main memory[10].

In many cache coherence protocols the home node plays an important role in maintaining coherence. For example the directory in a directory cache coherence protocol, which records what cache blocks each cache is caching, resides at the home node. Memory requests in a directory protocol are first sent to the home node, which consults the directory as to which other caches this request should be sent to.

The home node, just like shared memory itself, can be physically distributed. To a core, this manifests as different latencies for requests to different cache blocks.

Memory Operations with Cache Memory

When a memory operation is initialized, the local cache controller checks if the local cache has the correct permissions to service the memory operation from the cache. These permissions are encoded in *cache states*, which help ensure the coherence invariants of *SWMR* and *Data-Value* are maintained. If the memory location has the correct permissions the memory operation hits in the cache (a *cache hit* or simply a *hit*), and can be serviced by the local cache without communicating with other caches.

Otherwise the memory operations misses in the cache (a *cache miss*) and the cache controller issues a *memory request* for the cache block, requesting the block to read

[3]The other option is *write-update*, where all copies of the cache block are updated. While *MESIF* is a write-invalidate protocol, we present additions that allow *MESIF* to act like a write-update protocol. See Section 4.2 for details.

	Local Event		**External Event**	
State	core read	core write	read request	write request
writable-readable (**WR**)	*hit*	*hit*	→ **R**	→ **N**
readable (**R**)	*hit*	issue write request → **WR**		→ **N**
neither (**N**)	issue read request → **R**	issue write request → **WR**		

Table 2.1: *Actions and state transitions the cache controller executes for a cache block in some state given and local or external read or write. Transitions are denoted by arrows. Transitions for read/write requests must wait until permissions are given. External event transitions can occur immediately. Blank cells denote no necessary actions or transitions required.*

or write. This request is sent through the interconnection network to the other cache controllers which collectively respond so that the request will eventually return the correct permissions while maintaining the coherence invariants.

These cache states can be divided into three different *states*: both writable and readable (WR), only readable (R), or neither readable nor writable (N). These states correspond to the single writer-reader period (WR), the readers period (R), and caches not the single-writer reader (N) presented in Section 2.1.2.

When we introduce cache states in the next section, we use the cache states versus the core initiated events and incoming coherence events to specify the actions the cache coherence protocol takes. Table 2.1 summarizes the state (not cache state) transitions and actions that the core's cache controller executes on either a local core read or write operation, or a external read or write request.

2.2.2 Cache States

Cache states in a multicore generally encode four pieces of information[17]:

Validity: is this cache block readable. To be valid a cache block must be the same value as the latest cache block's value.

Exclusivity: is this *valid* cache block the only valid cache block in the system. A cache block that is exclusive is guaranteed that there are no other readers in the system – the single-writer part of the $SWMR$ invariant.

Dirtiness: does the cache block value differ from the value in main memory. A dirty cache block will eventually need to update the value in main memory. This usually occurs when the *owner* of the cache block is evicted.

State	Valid	Dirty	Exclusive	Owned
M	✓		✓	✓
O	✓	✓	x	✓
E	✓	x	✓	
S	✓		x	x
I	x		x	x
F	✓	x	x	✓

Table 2.2: *MOESI cache states. A tick indicates that the cache state has that property, a cross that it does not. A blank space indicates that it may or may not have the property.*

Ownership: does this cache have responsibility over this cache block. Only one cache block can be the owner at any one time. One responsibility for the owner of a *dirty* cache block is to on eviction, update the value in memory.

These properties help the cache coherence protocol maintain the two coherence invariants. Validity, Dirtiness, and Ownership help the *Data-Value* invariant by ensuring that the last written value is transferred. Exclusivity ensures the *SWMR* invariant, as only one cache can be exclusive at any one time, multiple readers will have the cache block as valid but not exclusive.

The cache coherence protocol specifies a number of *cache states*, which have some value for each of the above. Protocols often use a subset of the Modified(M), Owned(O), Exclusive(E), Shared(S), and Invalid(I) (MOESI) cache states[17]. Another state, the Forward(F) state is similar to the S state, but is used to help readers with cache-to-cache forwarding of data[4].

Table 2.2 shows summarizes properties of each state. The M and E states are both writable and readable and if present, there can be no other valid states in other caches for the same cache block – they must be in state I which is the only state that is not valid. Readable only states are the F, O, and S states. There can be any number of S states but the F and O states are unique. There can only be one cache state in the M, E, O, or F state at any time. Finally, the I state precludes writing or reading to the cache block.

These MOESI and F states are *stable states* [16], they are the only states a cache block can be in after a memory operation has completed. In addition to these stable states, there are many *transition states* which are used when caches are transitioning between these stable states.

Appendices A, B, and C, contain the entire set of transitions for the *MESIF* cache coherence protocol. These transitions consist of an current state, a new state, the event

[4]See Section 2.2.3 why cache-to-cache forwarding is desirable.

that triggers this transition, and a set of actions that occur, similar to Table 2.1 but on a much larger scale.

2.2.3 Performance Attributes

While different cache coherence protocols use different transitions as well as different system components, how well they perform can be described by a set of three performance attributes, which helps us in comparing different protocols:

1. **Latency:** How long it takes for a memory request to complete.
2. **Scalability:** How performance changes as more cores are added.
3. **Bandwidth:** How many messages must be sent to service each memory request.

The most important attribute of a cache coherence protocol is *latency*: how fast can a memory request be serviced.

We want requests to be serviced from other caches, rather than main memory. Accessing main memory can take a long time and adversely affect latency for the request. We wish to minimize this by getting cache blocks from other caches. A cache-to-cache miss is a a request that is serviced by another cache rather than from main memory.

An important factor in this (but by no means the only one) is the number of hops, or chain of messages, on the interconnection network. Snoopy and directory cache coherence protocols, the two most common types of protocol, require different number of hops to complete a memory request with a cache-to-cache miss. Snoopy protocols requires two hops and directory protocols require three hops. If the latency for hops are the same, then directory has a disadvantage over snoopy in terms of latency.

However, we describe in Section 2.2.4 different interconnection networks have differing properties. While a snoopy protocol has a two hop cache-to-cache latency, it requires a much stricter set of guarantees for the network than a directory protocol requires. This changes not only the latency of network hops, but also the scalability and bandwidth properties of the protocol itself.

Latency of memory requests can vary with different number of cores in a system. Requests in snoopy protocols have a two hop latency. However due to the interconnect requirements, the latency increases as the number of cores in the system increases due to increasing contention on the interconnect. Directory protocols fare better, with latency usually not increasing as much as snoopy protocols when cores are added. How a protocol's latency changes as the number of cores are varied is called *scalability* of the protocol.

Considering the trend for is to become increasingly multicore[5], having poor scalability can make a cache coherence protocol protocol less future proof.

The last attribute is the *bandwidth* of the protocol, how much of the interconnect network needs to be utilized to service memory requests. Interconnection networks do not have unlimited ability to move messages and data, and the more messages it must send to service a request, the increasing chance of contention on the network – adding delays which hurt the latency of the protocol.

Liveness

Another property of cache coherence protocols is liveness, which describes the progress (or lack of) for requests in the protocol. We wish for a protocol to guarantee *forward progress* for requests, where forward progress means that given enough time, the request will complete.

There are three distinct classes of liveness problems: deadlock, livelock, and starvation. Because we are concerned about the issues in cache coherence protocols, we are interested in the liveness of memory requests.

Deadlock occurs when two or more requests stall waiting on each other. We will show in Section 3.6.2 that there is a potential for deadlock in *MESIF* with multiple conflicting requests, and how to prevent deadlock from occurring.

Livelock is similar to deadlock, except requests are continuing to change state, do actions, but never make forward progress.

Starvation is when one or more requests cannot make forward progress, while other requests can. *Token Coherence transient requests* in certain circumstances may starve, requiring a much slower but guaranteed to make forwarding progress *persistent request*. See Section 2.2.6 for details.

These three attributes, especially latency, allow system designers to choose the appropriate cache coherence protocol for the right workload. One pivotal design point is the interconnection network, which we discuss in the next section.

2.2.4 Interconnection Network

As described in Section 2.2.1, the interconnection network joins together the local caches' cache controllers along with the shared cache's cache controller. When discussing communication between caches, we refer to these as network *nodes*. Network nodes can address other network nodes, exchanging messages and data. The cache coherence protocol uses messages and data sent through the interconnection network to service memory requests while maintaining coherence.

There are a number of different types of interconnection networks each with different costs, latency and maximum bandwidth of messages, scalability, and message ordering guarantees. Some cache coherence protocols rely on an interconnection network's message ordering guarantees to achieve coherence. The three types of guarantees, most strict to least strict, are as follows:

Totally-ordered: all nodes receive all messages in the same order.

Point-to-pont: messages sent between two nodes are in delivered in order, otherwise no other guarantees.

Unordered: no guarantees on the ordering of messages sent through them. Nodes can see different orders of messages from one another. For example, a node might receive message A before message B, while a different node receives message B before message A.

In this thesis we consider two different interconnects:

Bus based interconnects: Nodes communicate over a shared medium, typically a set of wires. To communicate, a node arbitrates to get access to the bus, and broadcasts messages to all nodes.

Point-to-point interconnects: Nodes are linked together directly into some topology. Nodes can directly send messages only to nodes that they are linked together with, otherwise communication is done by sending through a series of links.

Bus based interconnects offer totally-ordered guarantees for messages, but can limit bandwidth with serialization of messages. The scalability of a protocol using a bus can be limited, as the bus becomes a bottleneck.

On the other hand, not only are point-to-point interconnects much more scalable, as they do not have the bottleneck of a bus, but they can be lower latency as well as higher bandwidth to buses[1, 11]. For example Intel *QPI*[11], a point-to-point interconnect that is the successor of Intel's Front Side Bus (FSB), is composed of individual *QPI* links. Each link has a similar bandwidth of the entire FSB, but with a lower latency; the composition of many *QPI* links results in a low latency and high bandwidth point-to-point interconnect.

In the next section we discuss traditional cache coherence protocols, snoopy and directory protocols. We discuss how these protocols dictate what interconnection network they require, and how this affects the protocol's performance. After this, we discuss

Token Coherence, a cache coherence protocol that aims to have two hop cache-to-cache latency of a snoopy protocol but with better scalability, while reducing bandwidth requirements[12].

2.2.5 Traditional Cache Coherence Protocols

While there are many different types of cache coherence protocols, the snoopy and directory protocols are the most traditional ones. One of the most difficult aspects of cache coherence protocol design is how to handle simultaneous memory requests to the same memory location. Snoopy and directory protocols use two different methods to resolve conflicting memory requests, and maintain coherence.

Snoopy protocols

Most snoopy protocols rely on the ordering guarantees of a totally-ordered interconnect (such as a bus) to maintain coherence. Memory requests are *broadcasted* to all network nodes, and nodes which have an interest in the cache block can correctly respond so that collectively, coherence is maintained.

Write requests result in all cached copies being invalidated[5] (the *writer-reader* part of the *SWMR* invariant) and data being sent to the requestor if necessary. Read requests result in a downgrade of a writer if present (*readers* part of *SWMR* invariant) and cache block data being sent to the requestor.

Conflicting memory requests are dealt with directly by the interconnect itself. The interconnect decides which conflicting request should win via an arbitration mechanism. The winning request then broadcasts to all nodes, which respond as described above. All nodes see the same order of memory requests so coherence is simple to maintain.

The advantages of a snoopy protocol are low latency two hop cache-to-cache misses, as a request requires only two network hops to service.

The disadvantage is the requirement for an ordered interconnect, such as a bus, results in poor scalability. The bus becomes a bottleneck as more and more cores request the bus for more and more memory requests. Each request must be broadcasted to all nodes, so even removing the need for a bus does not resolve all the scalability issues of a snoopy protocol.

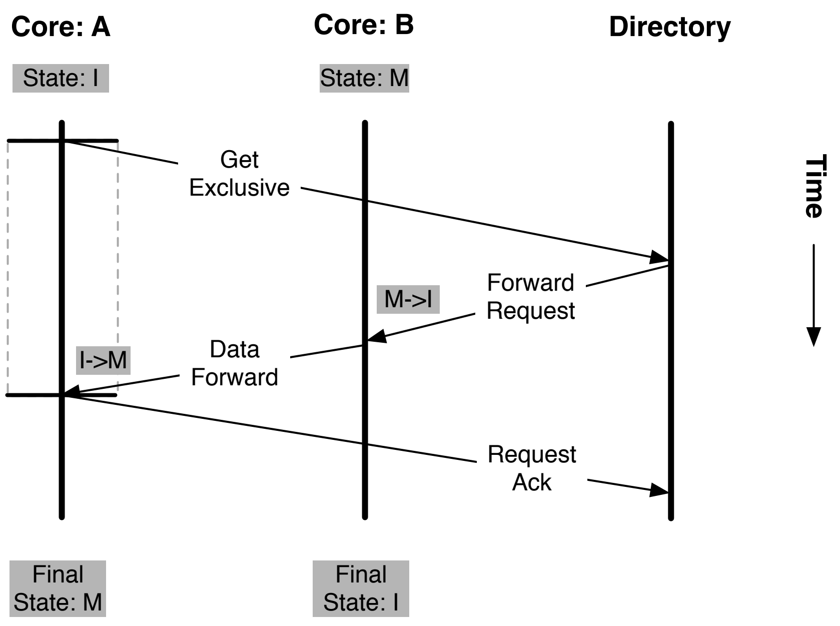


Figure 2.10: *Demonstrates the messages sent and received to service a read request in a directory protocol. Messages are represented by arrows showing the sender, receiver, and the sent and received times. The interval of time a request is active is represented by a box centered on the timeline for the processor. State transitions for the cache block are also included, as are the initial and final states.*

Directory protocols

Directory protocols rely on a *directory*, located at the home node, to store coherence information about cache blocks for the entire system[10]. The directory stores which caches currently have the node cached, and in what state. Typically, the directory is located at the home node of the cache block.

On a cache miss the core sends a memory request to the directory, which responds either with the data directly (by servicing it from the shared cache or main memory) if it is not cached in a local cache. Or the directory forwards the request to a local cache that does have the block cached, which responds with data and permissions to the memory request.

To maintain the *SWMR* invariant, the directory also forwards a write request to all caches with the cache block in the valid state. These other caches invalidate their cache block, preventing any readers in the writer-reader period, and send acknowledgement of the invalidation to the requestor. Explicit invalidation acknowledgement is not necessary in a snoopy protocol, as the total-ordering of an interconnection ensures that all nodes are invalidated at the point that the winning requestor broadcasts its memory request.

[5]Assuming the snoopy protocol is write-invalidate. If it is write-update, then broadcast of the write *updates* the value in all caches caching the written memory location.

Figure 2.10 shows which messages are sent during a write request by core A when the block is locally cached in core B (as indicated by the *State: M* directly below B).

First a message is sent to the directory (*Get Exclusive*) which forwards the request to core B. Core B downgrades from cache state **M** to **I** and forwards the data to core **A**; total number of network hops from request start to end is three. A final message from the requestor back to the directory is an acknowledgement that the request has completed, so that the directory can service another request for the memory location (which up until this point, maybe delayed by core A's request).

The directory decides the order of conflicting memory requests, unlike in snoopy protocols where the interconnect does this. On receiving more than one request, typically the first request wins and is serviced, with subsequent conflicting requests delayed until the winning request has completed and sent an acknowledgement message.

Because of the lack of broadcast, and the the ability to use higher bandwidth unordered or point-to-point interconnects, directory protocols tend to be more scalable than snoopy protocols. But with the extra indirection to the directory, memory requests take three network hops for cache-to-cache misses. This extra indirection to the directory can potentially result in latency for memory requests slower than that of a snoopy protocol, which can directly service requests from local caches with no indirection.

2.2.6 *Token Coherence*

The *Token Coherence*[12] was designed with the aim to have the best of both snoopy and directory protocols. Fast two hop cache-to-cache latency without relying on a totally-ordered interconnect. These are similar to the design goals for our *MESIF* protocol but with one exception. *Token Coherence* does not rely on broadcast messages to guarantee requests complete. Instead, *Token Coherence* wanted the freedom to not have to broadcast to all caches to save bandwidth.

To achieve these goals, *Token Coherence* specifies two parts to the protocol. A correctness substrate and a performance policy.

1. **Correctness Substrate**: Guarantees all memory requests will succeed and are coherent by using tokens. Prevents starvation via *persistent requests*, but not optimized for performance.

2. **Performance Policy**: A configurable protocol for performance. Not guaranteed to succeed but all requests are coherent. Uses various methods to guess which nodes have a cache block in what states, and selectively sends requests to only to the required nodes.

There is no need for a centralized directory[6] to keep track of which nodes have a cache block in what state.

Instead, coherence is ensured by a fixed number of *tokens* associated with each cache block. To receive permissions to write to a cache block, a node must gather all the cache block's tokens. Reading requires merely one token. The *SWMR* invariant is maintained as only one node can have all the tokens at any one time, and only by taking tokens from readers (and invalidating them: to read requires at least one token) can a node get permission to write. The *Data-Value* invariant is ensured by passing along the data with tokens as necessary.

The performance policy can guess which nodes have tokens, and send *transient requests* to only those nodes. On receiving a transient request, the node passes those tokens to the requestor: one token for a read request[7], all tokens for a write.

A transient requests is not guaranteed to succeed, as the performance policy may miss out sending to a node with tokens, or a conflicting requestor might be trying to gather tokens at the same time. Thus the performance policy can suffer from *starvation*, even if a request retries sending transient requests to all nodes, the request is not guaranteed to receive the tokens it needs.

When the performance policy fails to collect sufficient tokens for a request within a some time limit, the request times out and initiates a *persistent request*, which is guaranteed to succeed, though not necessarily quickly[8]. Without persistent requests, a transient request may continue to fail. This is the liveness issue of *starvation*, see Section 2.2.3 for details.

In the best case (and hopefully average case), requests take only two hops to complete. One performance policy is to broadcast to all nodes (**TokenB**), but can result in poor performance due to large bandwidth requirements. Other policies (such as **TokenM** use various methods such as directories or destination-set predication[13] to approach the best request latency of the broadcasting **TokenB**, but save bandwidth by sending fewer messages.

The disadvantage of this method is that the performance policy is not guaranteed to succeed, even in the case of **TokenB** which broadcasts to all requests. A persistent request requires arbitration to decide which persistent request wins over conflicting ones. This arbitration can be done using an ordered interconnect[12, 4].

[6]Directories are not necessary, but there exist performance protocols for *Token Coherence* that use directories[15].

[7]To prevent multiple nodes from responding to a read request, *Token Coherence* can use an optional *Owned* state, which only one cache may have at a time. We analyze this behavior of *Token Coherence* and apply it as an extension to the *MESIF* protocol, see Section 4.1 for details.

[8]Another option is that a request can resend the transient requests.

Additionally, tokens must be counted, and never discarded. To achieve this, valid cache states that could be silently evicted, such as the the E and S cache states, now must return tokens to the home node and/or shared cache. Even main memory may need to be modified to hold tokens that are evicted from the shared cache.

By focusing on bandwidth, *Token Coherence* loses out on an opportunity for two hop latency for cache-to-cache requests while *guaranteeing* that these requests will complete, much like as a snoopy protocol does (or *MESIF*).

In the next chapter we present the *MESIF* cache coherence protocol. This protocol is designed to guarantee two hop latency for cache-to-cache misses, like a snoopy protocol, but uses no ordering guarantees for the interconnection network. Unlike in *Token Coherence* protocol, requests in *MESIF* will complete, and there is no need to backoff and retry requests.

Chapter 3

MESIF

3.1 Introduction

The development of the *MESIF* cache coherence protocol was inspired by the development of the *QPI* interconnect[11]: a low latency, high bandwidth point-to-point interconnect.

Rather than broadcast through a totally-ordered bus such as a snoopy protocol, memory requests in *MESIF* are broadcasted to all caching nodes in an unordered interconnect (or point-to-point interconnect), allowing for a two hop cache-to-cache latency for requests. The scalability is improved compared to a snoopy protocol that relies on a less scalable bus interconnect.

The *MESIF* design goals of fast two hop cache-to-cache latency requests, coupled with improved scalability by using a more scalable unordered or point-to-point interconnect. These goals are similar to the goals of *Token Coherence*. However by relying on broadcasting, *MESIF* requests are guaranteed to complete without backoff and retry of requests. *Token Coherence* transient requests are not guaranteed to succeed and may require retry or backoff to a slower persistent request.

Coherence is achieved by splitting requests into two phases: broadcast and home phases. The broadcast phase consists of broadcasting the request to all caching nodes. A node that is caching the requested cache block in one of the four *forwarding* states (*Modified*, *Exclusive*, *Owned*, and *Forward*) forwards the cache block to the requestor, resulting in a two hop cache-to-cache latency.

When a request has the forwarding state, it has permission to *retire* the memory request but may have to wait, holding the forwarding state, until the memory request can *commit*.

On receiving all responses, the requestor enters the home phase and sends a message to the home node for that cache block, including a request for the cache block if no cache block data was forwarded. Home is responsible for retrieving the cache block from memory and forwarding it to the requestor. Conflicting requests are dealt with by a combination of the forwarding state, home node, and delaying of broadcast request messages.

We begin this chapter by describing the *MESIF* system, the cache states. Next, we discuss the *MESIF* coherence invariants, showing how these invariants ensure the coherence invariants discussed in Section 2.1.2. Finally the bulk of this chapter describes how memory requests in *MESIF* operate, including handling of simultaneous as well as conflicting requests for the same cache block.

We include a discussion about some real system issues and some common optimizations to the *MESIF* protocol.

The *MESIF* protocol presented in this chapter is based off of a technical report[6, 7]. This technical report contains the high level ideas of *MESIF* but misses out on a large number of details required to implement the protocol. This chapter fills in the many of the missing details, demonstrating how *MESIF* achieves coherence, especially with in the complex situation of multiple conflicting memory requests. In Chapter 4 we discuss our extensions to the *MESIF* protocol presented in this chapter.

3.2 *MESIF* Components

The *MESIF* cache coherence protocol is designed for a high bandwidth, low latency point-to-point interconnect such as an interconnect utilizing *QPI* links. Data structures required for *MESIF* are minimal, enough to keep track of currently active memory requests; there is no need for a directory like structure. These data structures are introduced in the following sections.

MESIF also requires 1 bit per cache block in each cache that initiates memory requests; the local caches but not the shared cache nor main memory. This bit is called the *flipping bit* and is used when we have multiple conflicting requests from the same node. See Section 3.6.3 for details.

We distinguish between two different network nodes: local nodes (or just nodes) and the home node. Local nodes consist of a core, local cache, and cache controller; there can be any number of these nodes. The home node for a cache block handles read and write requests to main memory, and acts as a arbitrator for conflicting messages. The home node does not necessarily need to be located at the shared cache, the shared cache could be in a different physical location to home, but this can be more complex. For the majority of this chapter, we assume there is no shared cache. We discuss how to add a shared cache when we discuss *MESIF* optimizations in Section 3.8.

MESIF is a write-invalidate protocol. Write values are stored in caches until eviction, in which they are written up to the next level in the memory hierarchy: local caches write back to the home node (which may contain a shared cache) which handles writing and reading to main memory.

There are five different request messages that a node (local cache) can initiate, each request is either *broadcasted* to all nodes, or sent to a single node:

Initial Requests:

1. *Get-Shared* (*GetS*): A broadcast request sent on a read miss.

2. *Get-Exclusive* (*GetX*): A broadcast request sent on a write miss. Can include an optional parameter indicating that this node has value cache block data, and only requires write permissions.

3. *Write-Back*: Sent to the home node to write back memory on local cache eviction.

Home Requests:

4. *Read*: A request sent to the home node when the request was not forwarded the cache block. May include an optional list of *conflicting nodes.*

5. *Cncl*: A request sent to the home node to cancel reading from memory as the cache block was forwarded to the requestor. May include an optional list of *conflicting nodes.*

Initial requests are the only requests not generated by the cache coherence protocol. Instead they are initiated by the cache controller. While a request has not yet completed, it is called an *active* request. Home requests are a part of an active request, and are sent only to the home node.

Responses to request messages come from two different sources: a local node or the home node. The responses sent by a local node are:

Local Node Responses:

1. *SACK*: Response from a cache that has the cache block in the *Shared* cache state. A reply to *GetS* or *GetX* message when this node has no active request for this cache block.

2. *IACK*: Response from a cache that *now* has the cache block in the *Invalid* cache state; the cache may have had the cache block as *Shared* but now downgraded to

Invalid. A reply to *GetS* or *GetX* message when this node has no active request for this cache block.

3. *Conflict*: Response from a cache that has an active request for the same cache block. Reply to *GetS* or *GetX* message. The *Conflict* reply includes the type of request this node has active: read or write request.

4. *DACK*: Response sent to the node that forwarded the cache block to the requesting node.

5. *Data[F,O,M,E]*: Response with data sent to the requesting node, along with the local cache state before and after this response. For example, *DataM* is sent if the forwarding node has the cache block in the *Modified* state *and* the forwarding node invalidates itself.

6. *Data[F,O,M,E]-XFR*: A special version of *Data[F,O,M,E]* which is transferring to the node ordered to by the home node. Does not count as a reply to *GetS* or *GetX* message, so a request that receives this message will also receive a reply from the same node.

And the responses sent by the home node are (home does not send any requests):

Home Node Responses:

1. *DataE-Home*: Response to an *Read* request. *Exclusive* data sent to the requestor. If a block is uncached[1], then this cache block will will be in the *Exclusive* state. Otherwise it is cached, and the cache block is in the *Forward* state. On a write request, the requesting cache immediately writes the block, and moves into the *Modified* state.

2. *ACK*: Acknowledges the *Cncl* request, but does not send any data as the requestor already has the cache block.

The responses send by home listed below only occur with conflicting requests (see Section 3.6 for details):

Home Node Responses With Conflicting Requests:

4. *DataE-XFR*: Response to an *Read* request. Similar to *DataE-Home*, but orders the requesting node to *transfer* (to forward) the cache block to the specified node.

5. *XFR*: Response to an *Cncl* request. Similar to *ACK*, but orders the request to transfer the cache block to the specified node.

[1]The requestor can tell if the block is uncached by other nodes by observing if it received any *SACK* responses to its broadcast. See Section 3.5 for more details.

6. *WAIT*: Response to an *Read* request. Cache block is cached, but scheduled to be forwarded to the requesting node. Requesting node must wait until this occurs, then complete its request.

7. *WAIT-XFR*: Response to an *Read* request. A combination of first *WAIT* then *XFR* responses.

8. *Conflict-Update*: Used to update a requests conflicting list. Occurs only in certain situations discussed in Section 3.6.2 and only on an *Read* request. This message is appended onto one of the other conflicting *Read* responses discussed above.

As we describe *MESIF* in detail in the chapter, we will go into more depth about each of the request and response messages.

3.2.1 *MESIF* cache states

There are five required cache states for *MESIF*, the *MESI* and the *F* states; the *MESIF* cache states. An optional state is the *O* or *Owned* state. While adding an additional state may seem like adding complexity, it is actually simpler to implement compared to just the five *MESIF* states. We use the *Owned* state in this thesis.

In addition to the four properties for cache states discussed in Section 2.2.2, *MESIF* adds a fifth property:

Forwarding: Responds to broadcast requests by *forwarding* data and permissions to the requestor.

This property states that this cache must reply to any broadcast request (*GetS* or *GetX*) by forwarding the data to the requestor. All requests landing in a forwarding state result in data forwarding. The states that have the forwarding property are the *Modified*, *Exclusive*, *Forward*, and the *Owned* state.

Table 3.1 shows the state transitions that *MESIF* undergoes. As *MESIF* is a write-invalidate cache coherence protocol, external write requests result in invalidation of all valid cache states.

There are many intermediate states that occur before entering the stable cache states as shown in the table[2].

In the next section we show how using these cache states, especially the forwarding states, can result in coherence with broadcast in a point-to-point interconnection.

[2]Appendix A lists the intermediate states for the a cache block in the local cache in *MESIF*.

	Local Event			**External Event**	
State	core read	core write	eviction	read request	write request
Modified	*hit*	*hit*	writeback → **Invalid**	forward data → **Shared**	forward data → **Invalid**
Exclusive	*hit*	*hit* → **Modified**	→ **Invalid**	forward data → **Shared**	forward data → **Invalid**
Shared	*hit*	write request → **Modified**	→ **Invalid**		→ **Invalid**
Invalid	read request → see Table 3.2	write request → **Modified**			
Forward	*hit*	write request → **Modified**	→ **Invalid**	forward data → **Shared**	forward data → **Invalid**
Owned	*hit*	write request → **Modified**	writeback → **Invalid**	forward data → **Shared**	forward data → **Invalid**

Table 3.1: *The state transitions and actions done in a certain state and a given event. Invalid state on a core read can result in four different states, see Table 3.2.*

	clean	dirty
sharers	→ **Forward**	→ **Owned**
no sharers	→ **Exclusive**	→ **Modified**

Table 3.2: *Transitions for Invalid state on a core read. A cache block can be either dirty or clean, and it can have come from home (no sharers – unless the requestor received an* SACK*) or another cache (sharers).*

3.3 *MESIF* Coherence Invariants

The basic aspects of *MESIF* are that requests *broadcast* messages to all nodes, requesting read or write permissions and cache block data. Assisting in this is a unique *forwarding* state, which is either the *Modified*, *Exclusive*, *Owned*, or *Forward* cache state. When a node with the cache block in the forwarding state receives a broadcast message for that cache block, the node *forwards* cache block data and the forwarding state to the request. This is how *MESIF* achieves two hop cache-to-cache latency.

If the forwarding state is not cached locally, we say that the forwarding state is *implicitly* in the home node/main memory; we desire that at all times, the forwarding state for a cache block exists somewhere, more on this later. A request then goes to the home node, requesting the cache block and the forwarding state.

However, as *MESIF* uses no ordering guarantees of the interconnect, messages can be delivered in any order, with arbitrary amounts of delay. Forwarding of the forwarding state and cache block on a request, implicit storing of the forwarding state in home, and an unordered interconnect together can result in a problem called *time warp*

Figure 3.1 shows an example of time warp when there are two requests active at the same time for the same cache block. A requesting node broadcasts its request to all nodes with caches. A node with the block cached in a *forwarding* state, responds to a request by forwarding the cache block data, and downgrading its own cache state to a non-forwarding state; either *Shared* on a read request or *Invalid* on a write request.

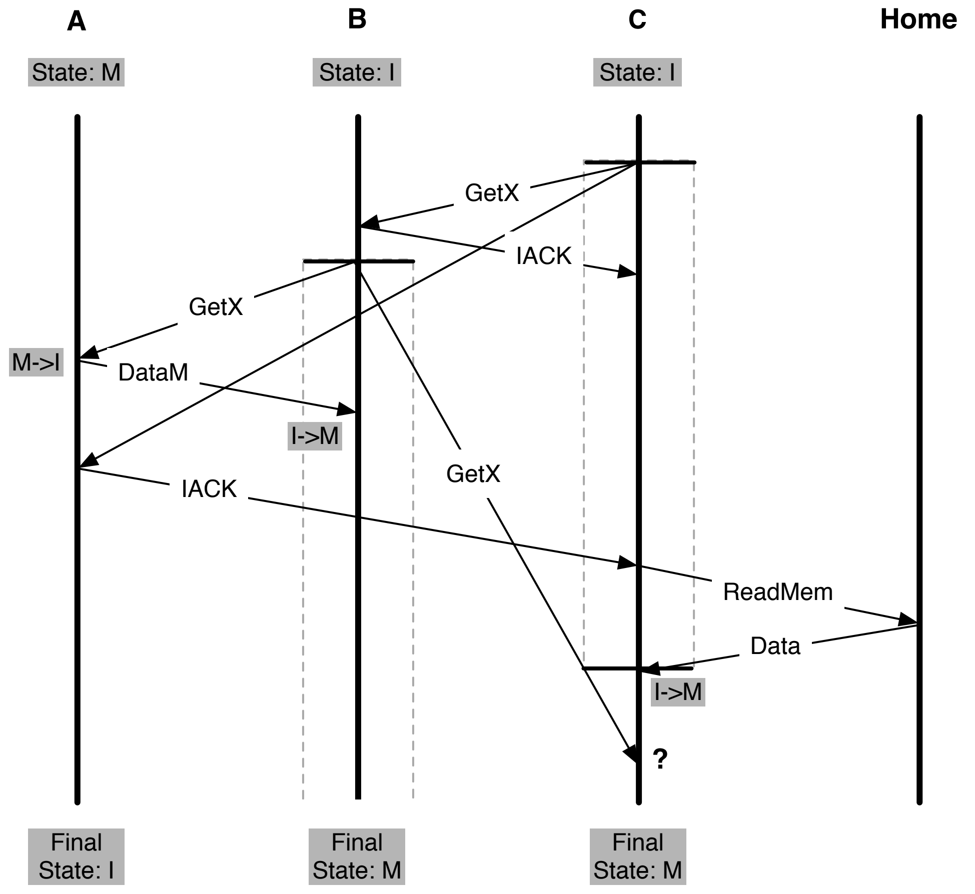


Figure 3.1: *Broadcast for two simultaneous requests results in violation of the* SWMR *invariant. We have two writers at the same time. This is the problem called* time warp.

In the example, both cores B and C are simultaneously requesting the same cache block to write. Due to the unordered nature of the interconnect, their broadcast messages miss each other: by the time time warp occurs, they both are not aware of any conflicting request. First core C sends a broadcast message to core B, which has not yet begun a request and so responds with $IACK$. Core B's requests arrives first at node A, the node with the block in the forwarding state. Core A responds by forwarding the cache block and forwarding state to B, downgrading to *Invalid*.

Next, core C's request to node A arrives, which responds with an $IACK$. On receiving this, core C believes the block is uncached in the forwarding state, and requests the cache block from the home node. Implicitly the forwarding state is in the home node/main memory only if it is not locally cached. Home responds by sending the cache block and forwarding state to node C, believing that node C discovered the forwarding state was not locally cached.

The final state of the system is that two nodes have the cache block as *Modified*, which violates *SWMR*: there is more than one writer at the same time.

To solve the issue of time warp, and achieve coherence when broadcasting with an unordered interconnect, we need to rely on requests finding the forwarding state where ever it is located: explicitly in a local cache, or implicitly at home. We need requests to only receive cache block data and read or write permissions from the forwarding state. And we need the cache block data to be the latest cache block data.

These can be represented by the four *MESIF* invariants listed below:

Forward Uniqueness: The forwarding state is unique, existing[3] in only one cache at a time.

Forward Transfer: Data and access permissions are received only from the cache with the forwarding state.

Forward Data-Value: On receiving the forwarding state, the cache block value in the requestor's cache is identical to the last cache block value in the forwarding cache, immediately before forwarding.

Forward SWMR: For every memory request there exists a time (during the request) known to the requestor, while the requestor has the cache block in the forwarding state, that the *SWMR* invariant holds.

For example in the case of time warp shown in Figure 3.1, if (somehow) **Forward Uniqueness** held, node C would never have asked for cache block data from home.

To receive new cache permissions (to upgrade the cache block's cache state), a core must initiate a request that finds the unique (**Forward Uniqueness**) forwarding state, and have the node with the forwarding state pass the latest cache block data (**Forward Data-Value**) and permissions (**Forward Transfer**) to the core. Thus, *Data-Value* invariant holds.

Once the core has received the forwarding state with the latest cache block, the request can *retire* knowing that there will eventually be a time where the request can *commit* as long as the request holds onto the forwarding state until the *SWMR* invariant for this request holds (**Forward SWMR**).

The first three *MESIF* invariants concern how the forwarding state act. The **Forward SWMR** invariant requires the cooperation of all caching nodes to achieve coherence.

[3]Home stores the forwarding state implicitly. If the forwarding state is not stored in any local caches, it is in the home node.

For example on a write request, **Forward SWMR** requires that at some point, there are no readers but the write request node (all other nodes have the cache block as *Invalid*). We will show that by broadcasting the request to all caching nodes, we can ensure **Forward SWMR** holds.

For coherence, consider how a cache block can have either a single *writer-reader* or many *readers*. These will be represented by the *MESIF* cache states, with the writer-reader either *Modified* or *Exclusive* and the readers *Owned*, *Forward*, or *Shared*.

The cache block is in either a writer-reader or readers phase. Let us show that a read or write request will, given the *MESIF* invariants, execute coherently.

If we are in writer-reader, then there is a node with the forwarding state (*Modified* of *Exclusive*) and all other caches are *Invalid*. Any request to the node in the forwarding state results in the forwarding state moving to the requestor, along with the latest cache block value, this is the *Data-Value* invariant. The *SWMR* invariant holds as **Forward SWMR** requires it to hold at some point after the request is forwarded the cache block.

On a read request, the node in the forwarding state downgrades to *Shared*, and the requesting node's cache state is either *Owned* or *Forwarded*. *SWMR* holds as we now are in a readers phase. Likewise on a write request, the node in the forwarding state downgrades to *Invalid* and the requesting node's cache state is either *Modified* or *Exclusive*. *SWMR* also holds as we are still in the writer-reader phase (but a different node is writing/reading).

Otherwise we are in a readers phase. There are two options here. Either the forwarding state is locally cached as *Forward* or *Owned* and there can be any number of nodes as *Shared*. Or the forwarding state is not locally cached, but it is implicitly cached in the home node/main memory.

In the former case, that the forwarding state is locally cached is similar to the case of a writer-reader and we will not go into detail.

However, if the forwarding state is not cached, it is implicitly cached at the home node/main memory. This is no different from the forwarding state is cached locally, except we must be careful to ensure that **Forward Uniqueness** holds. If we are not careful, then **Forward Uniqueness** may be broken as two or more requests ask home for the cache block and forwarding state.

Thus, with the *MESIF* invariants, requests can execute coherently.

For the remainder of this chapter we show how *MESIF* ensures the four *MESIF* invariants hold, and thus requests in *MESIF* are coherent. The next chapter describes what a *MESIF* request in general, the two phases it can be in, broadcast and home, and what messages a request sends and receives. We then discuss in Section 3.5 how requests in

MESIF operate in the absence of other requests while ensuring the *MESIF* invariants.

In Section 3.6.1 we talk about the basic unit of conflicting requests in *MESIF*: a pair of conflicting requests. Conflicting requests are requests that are both trying to get the forwarding state first. In that section we describe the various mechanisms used in *MESIF* to ensure the *MESIF* invariants hold.

To conclude the presentation of the basic *MESIF* cache coherence protocol by discussing multiple conflicting requests in Sections 3.6.2 and 3.6.3.

3.4 Requests in *MESIF*

The sections listed below show how *MESIF* requests are handled, and how they can be made to uphold the four *MESIF* invariants (and thus, achieves coherence). We show that *MESIF* requests always run to completion, and do not stall or fail requiring the request backoff and retry. For requests to be in conflict they must be running simultaneously. We split the discussion of *MESIF* requests into four parts based restrictions on what requests are allowed to run simultaneously:

Section 3.5: Requests without any simultaneous requests.

Section 3.6.1: Pairs of simultaneous requests.

Section 3.6.2: Simultaneous requests with up to one request per node.

Section 3.6.3: Any simultaneous requests.

We start with the most restricted and simplest version, no simultaneous requests, and work up to any possible combination of requests. Each section builds on the previous one.

This section describes in general a *MESIF* request.

3.4.1 Request Overview

All memory requests in *MESIF* have two phases: *broadcast* and *home* phases. The broadcast phase consists of broadcasting the request to all caching nodes. A node that is caching the requested cache block in a *forwarding* state *forwards* the cache block to the requestor. This results in the two hop cache-to-cache latency.

The broadcast phase has two purposes. First, it establishes the current state of the system: are there any copies of the cache block cached and are there any conflicting

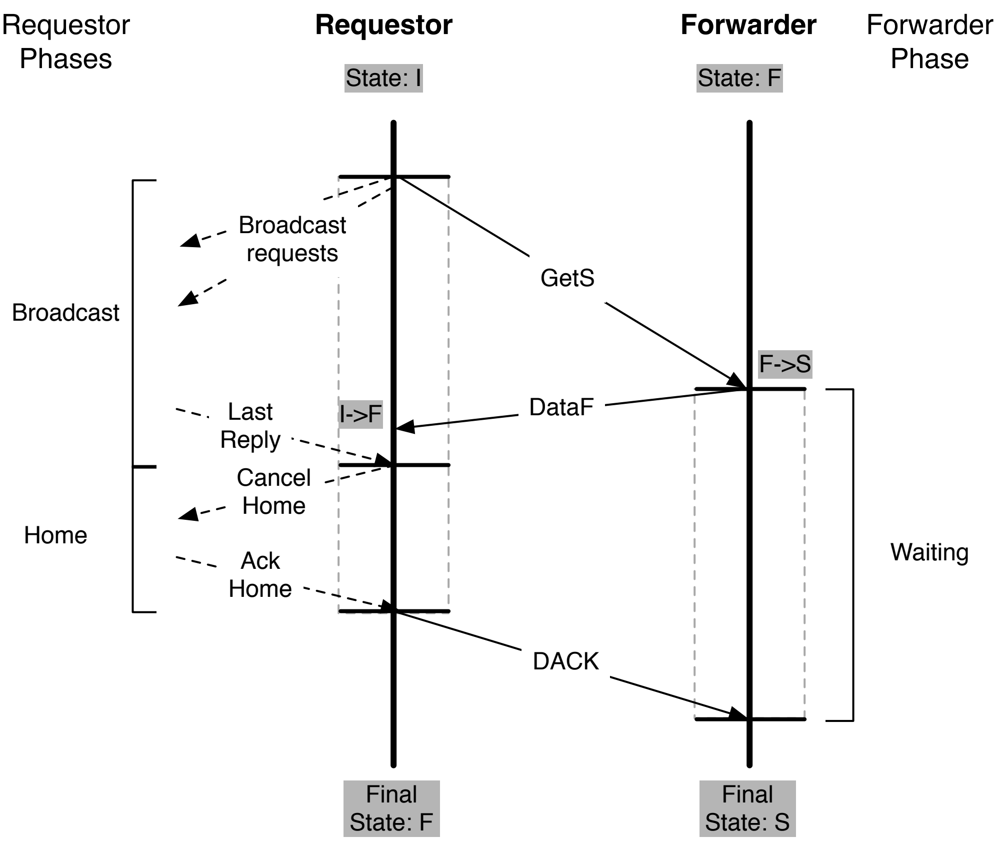


Figure 3.2: *The phases for a non-conflicting* MESIF *request with cache block forwarded to the requesting cache. Once the request is forwarded the data the request can* retire*; shown as the change in state from* Invalid *to* Forward*. The request may not necessarily* commit *at the same time as retiring. In this diagram the request commits when it has received all replies back.*

requests (see next section on conflicting requests). Second, it allows nodes to update on requests: invalidating or downgrading cache state and forwarding of data if the *forwarding* state.

On receiving a response from all nodes (forwarding of data counts as a response), the requestor enters the home phase[4]. Home phase consists of sending a message to the home node for that cache block, including a request for the cache block if no data was forwarded, and waiting for a reply. Additionally during conflicts, the home phase does not end until both home has replied *and* another node has forwarded the cache block to this request. Home is responsible for retrieving the cache block from memory and forwarding it to the requestor when the cache block is not locally cached in the forwarding state.

The forwarding node also has one phase, the *waiting* phase, which begins on the first

[4]The *MESIF* protocol requires all requests to enter the home phase, and send a *Cncl* or *Read* to home. We show in Section 4.4 that if the request is non-conflicting, the home phase is unnecessary.

request to arrive at the forwarding node. During the waiting phase, the (former) forwarding node delays handling any subsequent memory requests that arrive at the forwarding node. At the end of the requestor's memory request, the requestor sends a *DACK* message to the forwarding node, which ends the waiting phase. This phase is not important unless there are conflicting requests, and is discussed in Section 3.6. Additionally, in conflicting requests, the home node has a phase called the *conflict* phase, which lasts until the last conflicting request has sent an *Cncl* or *Read* to home.

Memory requests in *MESIF* begin with broadcasting one of the two *initial requests*, *GetS* or *GetX*, depending on whether the memory operation is a read or a write respectively. The other possible initial request, *Write-Back*, is handled similarly to how a forwarding node enters the *waiting* phase, but with the *Write-Back* node 'forwarding' the cache block to the home node.

Figure 3.2 shows the timeline of messages and phases a requestor and forwarder nodes go through when there are no conflicting requests. All requests in *MESIF* follow a similar pattern.

In the case that the cache block is not locally cached in a *forward state*, the requestor receives all replies but is not forwarded the cache block.

3.5 No Simultaneous Requests in *MESIF*

We begin describing *MESIF* requests by assuming that only one memory request can be active at any time, preventing any simultaneous requests, and thus, preventing any conflicting requests.

The case that the *MESIF* invariants hold for non-conflicting requests goes follows. Given some state of the system in which the *MESIF* invariants hold, then for all possible memory operations, the *MESIF* invariants still hold in the resulting state of the system.

Memory operations that hit in their cache (the cache block has the correct access permissions) will not affect the *MESIF* invariants (see Section 3.3), so if they held before the memory operation, they will hold afterwards.

We only need to show that for all possible *memory requests*, memory operations that do not hit in the cache and require a cache state upgrade, if the *MESIF* invariants held in the system before the request, they hold after the request. As we are assuming there are no conflicting requests, we can concentrate on individual *MESIF* memory requests.

This as well as the initial state, or base case, will be used to form an inductive proof that the *MESIF* invariants always hold and all non-conflicting memory operations are coherent.

A *MESIF* system can be characterized by the state of the cache block in main memory, and the valid stable states of the cache block in the local caches: a *stable system state*[5]. There are six possible combinations shown in Table 3.3. The transitions between the system states are shown in Figure 3.3.

	Local Nodes	
Home Node	**Forwarding**	**Shared**
Exclusive	∅	∅
Forward	∅	1 or more
Invalid	Modified	∅
Invalid	Exclusive	∅
Invalid	Forward	∅ or more
Invalid	Owned	∅ or more

Table 3.3: *The six stable system states in* MESIF*. Note that the last two states may have zero (not one) or more shared states as evictions of* Shared *cache blocks are silent.*

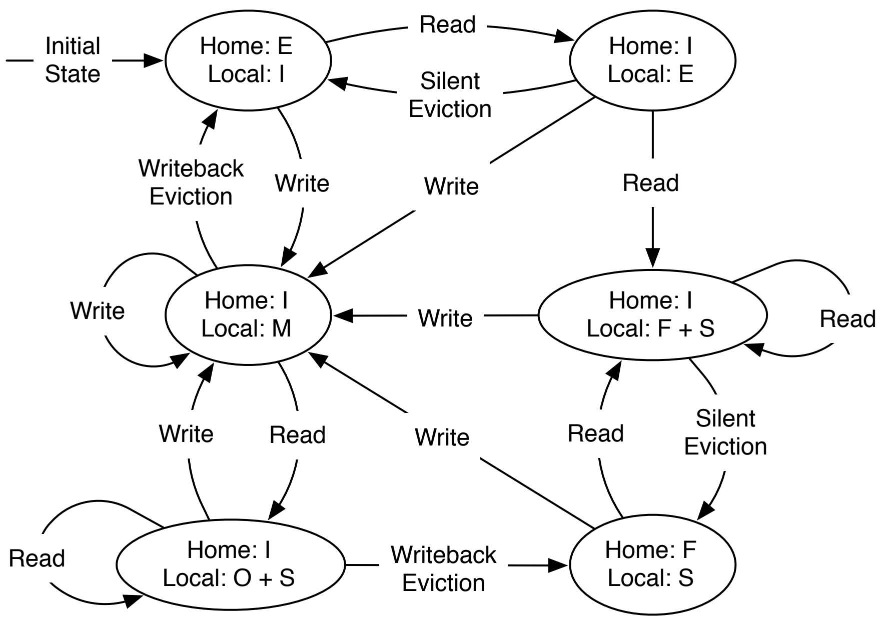


Figure 3.3: *Stable system state transitions.*

While this diagram looks complex, most of the transitions are similar to each other. There are 6 stable states, and 3 actions, giving 18 transitions, but as home node cannot evict, we only have 16 transitions. We can divide each transaction into one of three classes, based on the request type: *GetS*, *GetX*, or eviction. We then further divide each of these classes based on whether or not the *forwarding* state is locally cached or

[5]When transitioning *MESIF*, like many cache coherence protocols makes use of transition states. A cache in a transition state will have the equivalent permissions to one of the stable states, and thus, can be sorted into one of the stable system states.

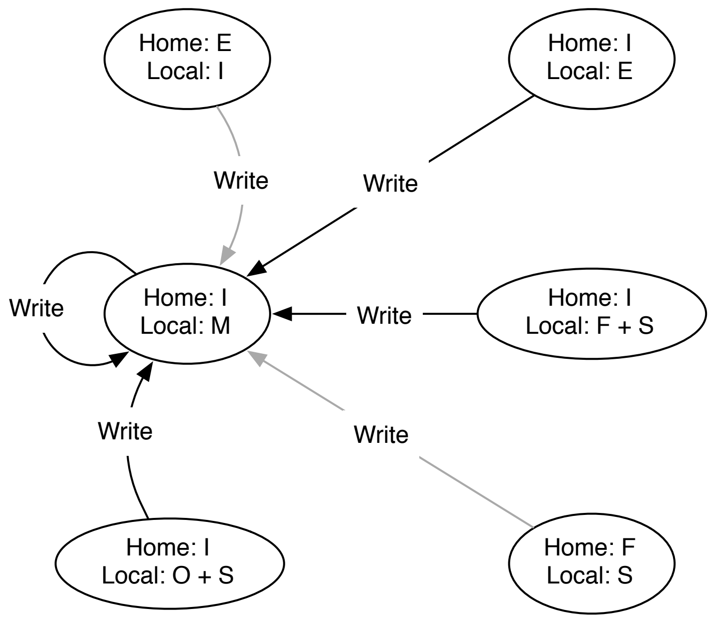


Figure 3.4: *Stable system state transitions for write requests. The gray transitions are from states with the* forwarding *state is not cached locally.*

not.

3.5.1 Write Requests

Figure 3.4 shows the six different write (*GetX*) request transitions. We must show that in all six of the system states, a *GetX* request retains the four *MESIF* invariants. As with all *MESIF* requests, the requestor broadcasts its request (*GetX*) to all caching nodes. When a write request, a node in the *Invalid* state replies with an *IACK* and does nothing else. On the other hand, a node in the *Shared* state replies with an *SACK* and downgrades to *Invalid*[6].

If the forwarding state is not cached locally (two different transitions), then when all replies have arrived, the requestor sends *Read* to home (and transitions to the home phase) as shown in Figure 3.5. The home node then retrieves the cache block from memory and forwards it to the requestor, which completes the request, transitioning to the *Modified* cache state.

The **Forward Uniqueness** invariant holds as, when forwarding the cache block, home became *Invalid* as no other memory requests can begin until this active requests completes, and when it does complete, it will be in the forwarding state (and thus, home is not in the forwarding state).

[6]The *MESIF* cache transitions are recoded in Table 3.1.

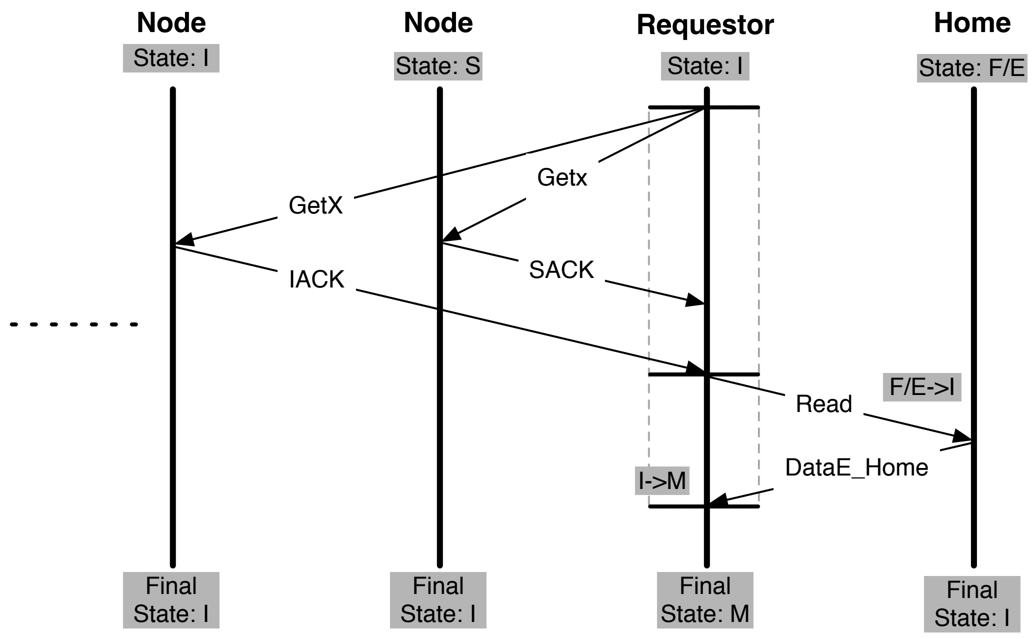


Figure 3.5: *Timeline for a* GetX *request, with the forwarding state not cached locally.*

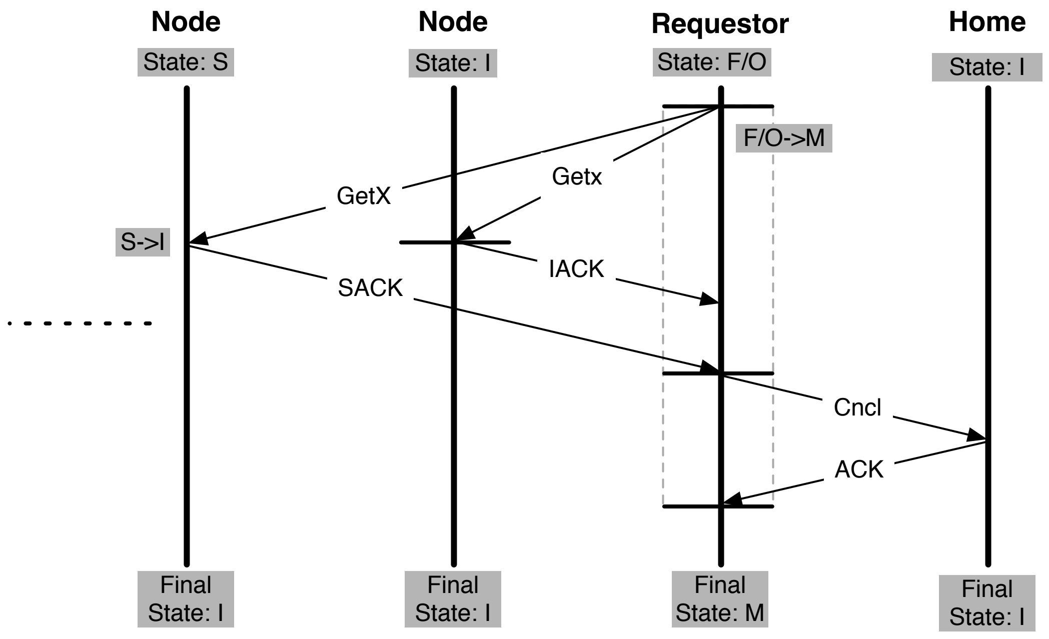
The **Forward Transfer** and **Forward Data-Value** invariant holds as when home forwarded the cache block, it forwarded both the forwarding state, and the current value for the cache block.

Finally, the **Forward SWMR** invariant holds as by the time all caching nodes had received *GetX*, they invalided (if *Shared*) or were already in the *Invalid* state. When the request receives the last *IACK* or *SACK*, then all nodes must have already received the *GetX* message, and are thus, *Invalid.* It is this point that the *SWMR* invariant is held, and it is a point known to the requestor.

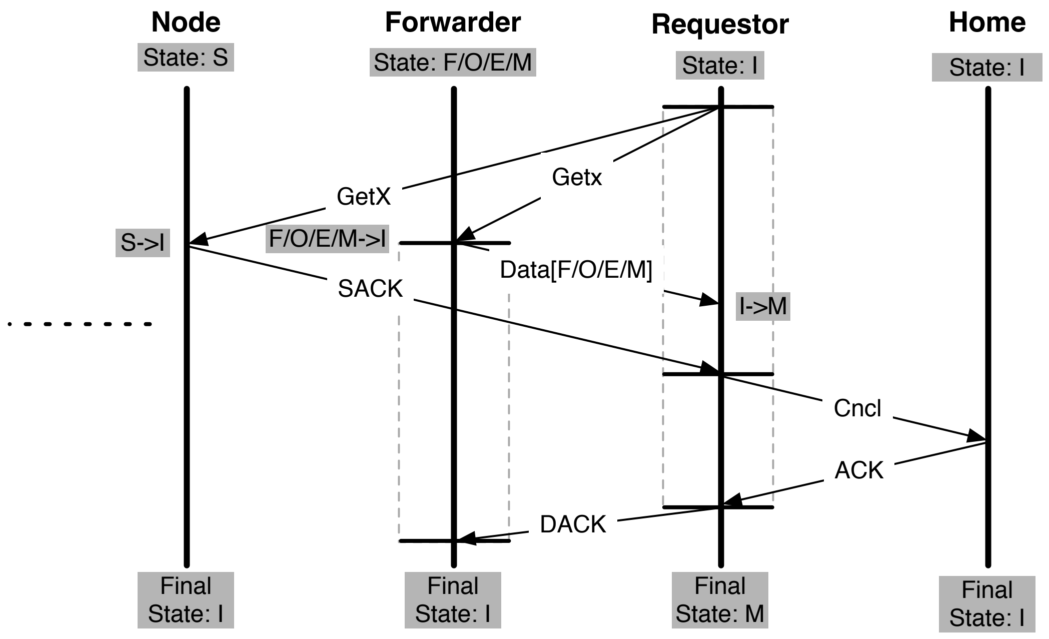
When the requestor receives the forward cache block, the *SWMR* invariant still holds (all but the requestor are *Invalid*), and the requestor has the block in the forwarding state right before the request completes. Therefore, the **Forward SWMR** invariant holds.

On the other hand, if the forwarding state is cached locally, then by the time all replies to the *GetX* message have arrived, the requestor has the cache block in the forwarding state.

Once all replies have arrived, the requestor sends *Cncl* to home, informing home that it is not necessary to read main memory. Home responds with an *ACK* message, which the requestor receives, sends *DACK* to the forwarding node, and completes its memory request.



(a) *Forwarding state at the requestor, but does not yet have permissions to write. However, the write operation may* retire.



(b) *Forwarding state in some other node.*

Figure 3.6: *Timeline for a* GetX *request, with the forwarding state cached locally.*

There are two possibilities: the requestor already had the block in the forwarding state, or another cache has it, shown in Figure 3.6a and Figure 3.6b respectively.

If the requesting cache has the cache block in a forwarding state, then it either has it in the *Modified*, *Exclusive*, *Forward*, or *Owned* states. For the *Modified* and *Exclusive* states, the requestor already has the block in a writable state, and the memory operation can immediately proceed. While in the *Forward* or *Owned* states, the requestor must initiate a write memory request: there could be nodes with the cache block as *Shared*[7].

However, even in the *Forward* and *Owned* states, the write operation can *retire.*

Otherwise, the requesting cache does not have the cache block in a forwarding state, and must broadcast a *GetX* request to have the block forwarded to it. Eventually the forwarding node forwards the cache block, and the requestor node now has the block in the forwarding state. In the previous situation, by already having the forwarding state, the requestor acts as if it was immediately forwarded the cache block.

There are two differences between these situations as shown in Figure 3.6, having the forwarding state allows the requestor to *immediately* move to the *Modified* state – it can retire the memory request as soon as it has started. The reason is the forwarding state decides the winner of a race over a cache block. As the requestor already has the forwarding state, it can decide that its own request is the winner, and will eventually invalidate all other caches.

The second difference is that by having the forwarding state, there is no forwarder, so no *waiting* phase, and no need to send a *DACK* to a node, unlike in Figure 3.6b.

The *MESIF* invariants of **Forward Uniqueness** holds in both cases as the forwarding state was never in two caches at the same time.

The **Forward Transfer** and **Forward Data-Value** are held as when the forwarding node forwarded the cache block, it forwarded both the forwarding state, and the current value for the cache block. In the case that the requestor has the forwarding state, it already has the current value stored locally.

Finally, the **Forward SWMR** held as all caching nodes were in the *Invalid* state by the time all replies arrived at the requesting node (which then sent a *Cncl* message to home). Thus, there was only a single writer-reader and *SWMR* invariant held.

This shows that any write requests as shown in Figure 3.4 preserve the *MESIF* invariants if they hold. Next we show the same for read requests.

3.5.2 Read Requests

As the forwarding states are all valid states, caches with the forwarding state already have permission to read the cache block. The only caches that initiate read requests are those in the *Invalid* cache state. The only difference between a read request and a write request is that a *GetS* message does not invalidate a cache with the cache block as *Shared.*

[7]Or all sharers might have silently evicted the block already, but the requestor cannot tell without

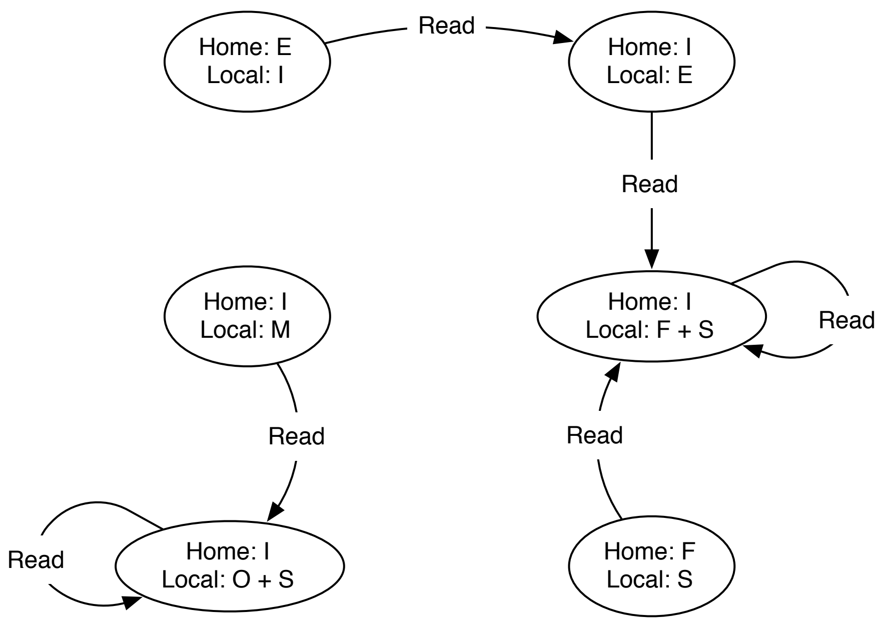


Figure 3.7: *Stable system state transitions for read requests.*

If the forwarding state is not cached locally (two different transitions), then as for write requests as shown in Figure 3.5, after all replies the requesting node sends *Read* to home. On receiving *DataE-Home* from home, the requestor transitions to either *Exclusive* if there are no sharers or *Forward* if sharers are present. The requestor already knows if there are sharers at this point, sharers reply with *SACK* rather than *IACK*, and the requestor has gotten a reply from all caches.

We can reason from how a write request upholds the *MESIF* invariants when the forwarding state is not locally cached here. The only difference is that sharers are not invalidated, but this does not affect the *SWMR* invariant as for reads, we require that there are no writers – which is true as there is no forwarding state in the system. Hence all four *MESIF* invariants hold provided they held before this read request.

On the other hand, if the forwarding state is cached locally, then (again) as for write requests shown in Figure 3.6, after all replies the requesting node will have the cache block in the forwarding state. There will be sharers here, as the *MESIF* protocol specifies that on a *GetS* request, the forwarding node downgrades to *Shared*[8]. This means that the final cache state the requesting node can be is either *Forward* if the data is not dirty (the forwarding node was *Exclusive* or *Forward*), or *Owned* if the data is dirty (the forwarding node was *Modified* of *owned*).

As before, we can reason from how a write request upholds the *MESIF* invariants when

querying all nodes, hence, it must broadcast a *GetX* request.

[8]An alternative to downgrading to *Shared* is to downgrade to *Invalidate*. See Section 3.8.2 on optimizing for the migratory sharing pattern.

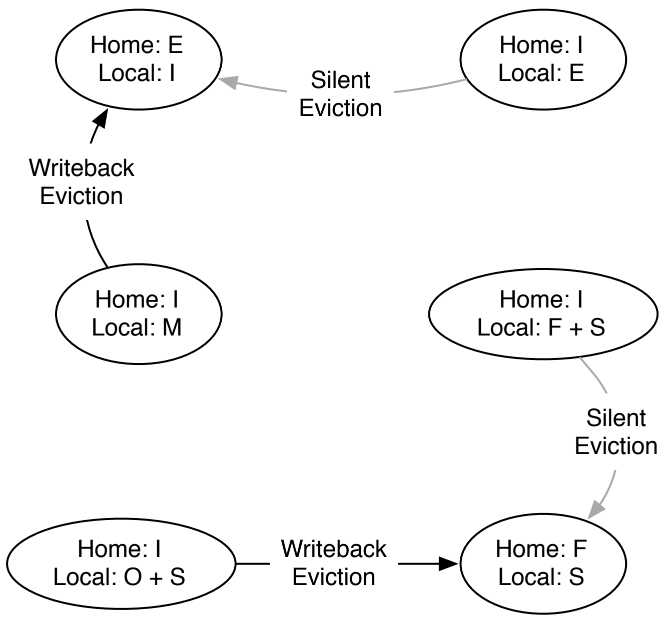


Figure 3.8: *Stable system state transitions for cache evictions.*

the forwarding state is locally cached. The *SWMR* invariant allows for multiple readers, as long as there are no writers. There are no writers as *MESIF* requires forwarding states to forward the cache block, and downgrade to *Shared* or *Invalid*. As all the cache states that can be written, are also forwarding states, the *SWMR* invariant holds, and all four *MESIF* invariants also hold provided they held before this read request.

3.5.3 Cache Block Evictions

The last of the three initial requests, *eviction* requests occur when the cache needs to make room for another cache block, and this valid cache block is to be evicted. As mentioned before, eviction of caches with the cache block as *Shared* can be ignored, downgrading a *Shared* cache block does not affect coherence in *MESIF*. It is only upgrading access permissions that can affect coherence.

Eviction of forwarding states requires the forwarding state to move to the home node (in other words, main memory, if there is no cache in the home node). This results in a forwarding of the forward state, and so we must show this holds the *MESIF* invariants.

There are two types of eviction for a forwarding state. A silent eviction occurs when the forwarding state is *clean*, the data value is the same as that in main memory. A *write back* eviction occurs when the forwarding state is *dirty*, the data value may be different from main memory.

Silent evictions do not need to send any messages, and can immediately complete. This is because the forwarding state does not need to be sent to the home node. The home node does not actually record that it has the cache block in a forwarding state – or in any state. The home node is in the forwarding state because the forwarding state does not exist in any local caches. This is the reason why *time-warp*, as shown in Figure 3.1, can occur if requests do not discover the state of the system correctly, and believe the forwarding state is uncached[9].

On the other hand the dirty cache states, *Modified* and *Owned*, must write back the data value else violate **Forward Data-Value**, that the forwarding state has the same value for the cache block as the forwarding (in this case, evicting) cache immediately before forwarding.

Figure 3.9 shows the timeline for a *write-back*. Identical to the timeline for a forwarding node, except this is 'forwarding' the cache block to home. As mentioned before, the state is not transmitted to the home node, but is a product of the system state – in this case if there are any sharers or not. The eviction completes once the home node sends an *ACK* message signaling that home has completed writing the cache block data.

While the write-back is occurring, the evicting cache delays all requests until the eviction request has completed; write-back as a *waiting* phase. As we will discuss in the next section, this prevents cases such as time-warp occurring.

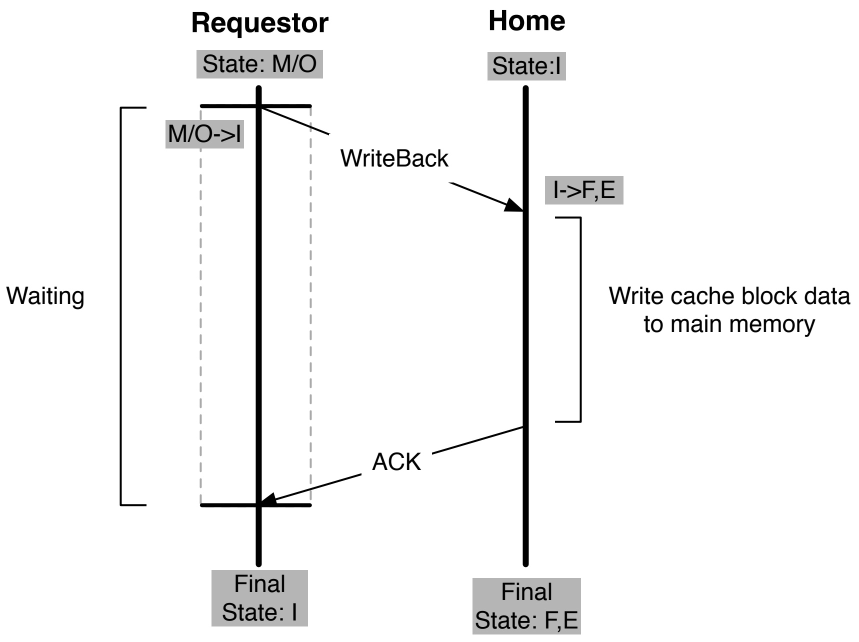


Figure 3.9: *Timeline for an* Write-Back *eviction.*

[9]We will show in Section 3.6 that this can occur, but in this case, home knows that the forwarding state is already cached, and will not send out a second forwarding state.

The **Forward Uniqueness** and **Forward Transfer** invariants hold for an eviction request as eviction downgrades the evicting cache to *Invalid*, and the forwarding transferred the forwarding state to the home node. The **Forward Data-Value** holds as at the end of the eviction, the home node has the same cache block value as the evicting cache held. Finally, **Forward SWMR** holds as eviction is not a read or write memory operation.

This section has shown how memory operations in *MESIF*, in the absence of simultaneous requests, are coherent. We showed how memory requests can be executed coherently with broadcast in an unordered interconnect (we did not need point-to-point ordering). In the following sections, we show how *MESIF* can handle conflicting requests coherently.

3.6 Simultaneous Requests in *MESIF*

The *MESIF* invariants require that for any request for cache block permission and data, the request must receive these from the node with the (unique) forwarding state. Section 3.3 introduced the four *MESIF* invariants, and an intuition that each forwarding of the cache block to a requestor becomes one link in a chain for forwarding nodes.

With no simultaneous requests, and so no conflicting requests, finding and receiving the forward state is straightforward. Broadcast of the request message finds if the forwarding state is cached or not. If it is cached, the node with the cached forwarding state forwards to the requestor. Otherwise, it is not cached and the requestor asks the home node for the cache block, receiving the block in the forwarding state. The order of which requests get the forwarding states is unambiguous: the order the requests begin in.

When there are simultaneous as well as conflicting requests, there are two problems that occur. First, what is the order that the requesting nodes should receive the forwarding state. Second, ensuring that each request *does* get the cache block forward to it precisely once.

Thus, to handle simultaneous and conflicting requests in *MESIF*, the protocol must ensure that every request is scheduled in some order so that the forwarding state hops between each request, while satisfying the four *MESIF* invariants.

As *MESIF* does not utilize any ordering guarantees of the interconnection network, messages can be delayed an arbitrary amount of time. This means that *MESIF* must handle unlikely, but physically possible scenarios. We show that any possible ordering of simultaneous, including conflicting, requests is handled by *MESIF*; see Figure 3.10b for all possible simultaneous orderings.

While two read requests are not conflicting, the reads will return the same value no matter the ordering of the two reads, *MESIF* read requests are still conflicting. This is because the *MESIF* invariants require all requests to be serviced by the node with the unique forwarding state. In the next chapter in Section 4.1, we describe a modification to *MESIF* that allows for non-conflicting, simultaneous read requests.

3.6.1 Pairs of Simultaneous Requests

The simplest case of simultaneous memory requests is a pair of simultaneous memory requests. In this section we describe all possible pairs of simultaneous memory requests and how *MESIF* handles them, including when they are conflicting and when they are just simultaneous. In later sections where more complex orderings of simultaneous requests are discussed, these scenarios can be treated as sets of simultaneous and conflicting request pairs.

A necessary condition for requests to be conflicting in *MESIF* is that they must be active at the same time; they must be *simultaneous* requests. However, not all simultaneous requests are conflicting requests. Recall that a *MESIF* read or write request[10] has two phases, the broadcast and home phase as shown in Figure 3.2.

A pair of requests are *conflicting* if at least one request's message broadcast, either *GetS* or *GetX*, arrives during the other request's broadcast phase. For a conflicting request the reply is the *Conflict* message, indicating that this request is in conflict rather than the usual *IACK* or *SACK* reply. This implies that *conflicting* requests must be *simultaneous*. But if the request message is delayed, rather than replying with a *Conflict*, there will be no conflict as we will show below.

Because memory requests, or memory requestors, are associated with network nodes, and of course a core and local cache, we can synonymously say the two nodes are in *conflict* if their active memory requests are conflicting[11].

This leads us to one of the most important invariants used to handle conflicting requests:

Pairwise Conflict-Aware : For any node A to be in conflict with a node B, A will know it is in conflict with B before it enters its home phase, and B will know it is in conflict with A before it enters its home phase. Both nodes will know which type of request (read or write) the other is undertaking.

All nodes that a request is in conflict with are recorded, and this *conflict list* is sent to the home node in the *Read* or *Cncl* message. Thus, a conflicting node will send to home

[10] An eviction request is only sent to home, which cannot initiate a read or write request so no possibility for conflicts.

[11] When we consider any orders of requests, a request might be in conflict with two requests from the same node. See Section 3.6.3 for details.

all nodes it is in conflict with, and all those nodes will do likewise as they are **Pairwise Conflict-Aware**. This will allow home to both correctly create an order of forwarding for the conflicting nodes, as well as ensuring that *all* requesting nodes will be forwarded cache block precisely once.

Thus in *MESIF*, which upholds the **Pairwise Conflict-Aware** invariant (demonstrated below, with the possible pairwise orderings), we can give the final definition of conflicting requests. A pair of requests are *conflicting* if at least one request's message broadcast arrives during the other request's broadcast phase.

Any request in *MESIF* that arrives at another request in the *home phase* is delayed until the end of the request, and hence, is not conflicting. This is then treated as two non-conflicting requests, where one request begins after the delaying request has finished.

The one counter example to this is if the request arrives at an simultaneous request that is *already* in conflict with the requestor, even if the request is in the home phase, the request responds with a *Conflict* reply rather than delaying the request message. We discuss this scenario in more detail later.

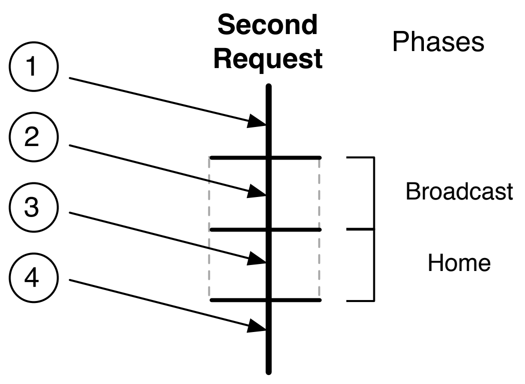
Because of the lack of ordering guarantees provided by the interconnect network in *MESIF* messages can be delayed an arbitrary amount of time. Therefore, we need to show for any possible ordering of messages between two simultaneous requests, the *MESIF* invariants hold.

Figure 3.10a shows the four places a broadcast request from one requesting node can arrive at in another node's simultaneous request, we assume that the first request begins before the second. For each of these four places, Figure 3.10b shows the possible message broadcasts for the second request, which will allow us to sort by non-conflicting and conflicting requests.

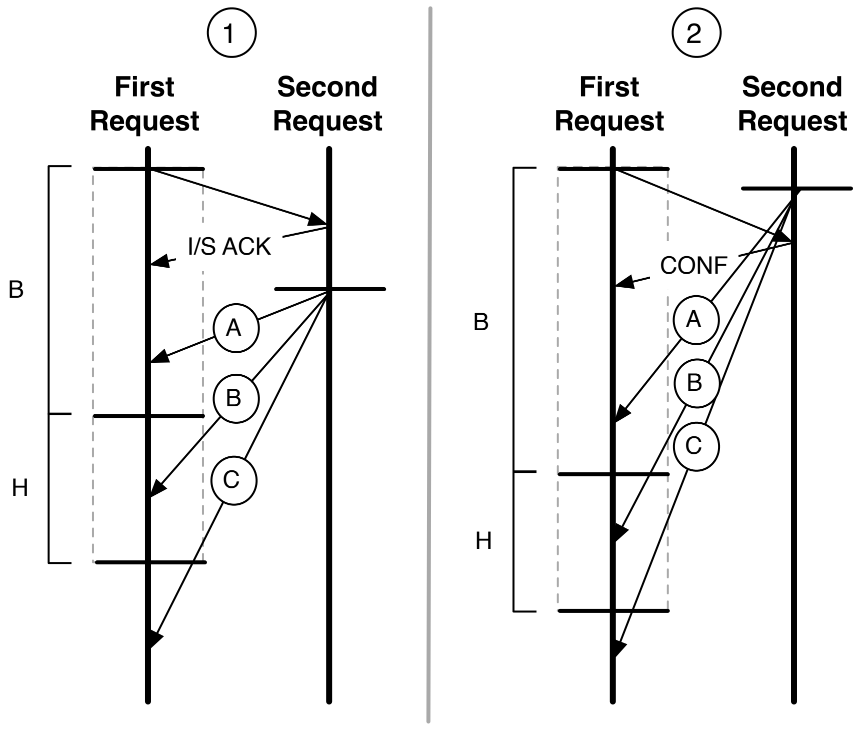
For the first ordering in Figure 3.10a, *1a* is the only conflicting ordering, *1b* and *1c* are both non-conflicting. In this first ordering, the second request node responds with either an *IACK* or *SACK*. It could instead, have the forwarding state, and thus respond by forwarding the cache block. However, the second request node would then enter the forwarding *waiting* phase, and will end only after the first request completes, preventing the second request from being simultaneous with the first.

In the second order, *2a* and *2b* are conflicting, and *2c* is not possible with pairs of simultaneous requests[12]. The reason is that the first request will send a *Read* or *Cncl* to home, specifying the second request as a conflict. Home will respond with the cache block (if *Read*) and inform the first request to forward to the second request. This first request will not complete until it receives a *DACK* from the node it forwarded to: the

[12]It is possible when we have three or more requests. This case is thus, discussed in Section 3.6.2.



(a) *The four possible locations a broadcast request message might hit in a simultaneous request.*



(b) *Potential request messages for second request base on where the first request arrived. The broadcast and home phase are abbreviated as 'B' and 'H' respectively. For space considerations,* Conflict *response has the label 'CONF' in this diagram.*

Figure 3.10: *All possible orderings of request messages between two simultaneous requests for the same cache block. Only messages relevant to demonstrating the request orderings are shown.*

second request, and because the second request will not *DACK* until it has at least, received all replies, *2c* is not possible.

In the third order, there is only one possibility, which is a mirror image of *2b*. The only difference is that in *3*, the second request starts before the first, though the 'second' request's message arrives during the 'first' requests home phase.

The fourth ordering as shown in Figure 3.10a also has only one possibility, which is the mirror image of *2c*.

We first deal with non-conflicting but simultaneous pairs of requests, orders *1b* and *1c*. Then conflicting pairs of requests, orders *1a*, *2a*, and *2b*.

Non-Conflicting Simultaneous Requests

We show that the two non-conflicting orderings, *1b* and *1c*, in Figure 3.10b are non-conflicting in that they are equivalent to two non-simultaneous requests ordered one after the other, and follow the *MESIF* invariants as shown in Section 3.5.

In the two non-conflicting orderings, the first request has already received a reply from the second request's node, before the second request starts. We need to show that the first request, the forwarding node (if it exists), and all other nodes, act identical to the case that the second request begins after the first request ends – identical to a pair non-simultaneous requests.

First request:

The first request delays any request messages when in the home phase until the end of the request. Thus for ordering *1b*, the first request node orders the second request's message after the first request has completed, and the second request is dealt with in the same way as a non-simultaneous request is: by sending an *IACK*, *SACK*, or forwarding the cache block. Usually after the first request has completed it will have the cache block in a forwarding state, and will respond to the (delayed) request from the second request by forwarding the cache block. However it is technically possible (but unlikely) for the first request's node to evict the cache block, then deal with the second request with an *IACK*.

As for ordering *1c*, in which the second request's broadcast message truly arrives after the first request has completed, then the first request's node acts exactly as if the second request started after the first request ends.

Forwarding node:

If the forwarding state is locally cached in a non-requesting node, then it also must act as if the second request starts after the first completes.

Recall from Figure 3.2, that the forwarding node enters the *waiting* phase until a *DACK* from the node forwarded to, arrives. We know that the first request has already received all replies from all nodes, including the second request's node. Also there is no possibility that the forwarding state can appear in a node *after* the first request's message has reached and been replied to. This would require the node initiate the complete the memory request without conflicting with the first request. But this is not possible as we assume no conflicting requests.

So if the forwarding state is cached locally then the first request's message must have reached it before the second request's message (by assumption, the first request has received all replies). When the second request's messages reaches the forwarding node it is delayed until the first request completes, sends a $DACK$, and the $DACK$ arrives at the forwarding node. Thus, the second request is dealt with by the forwarding node in the same way it would be if it started after the first request completes.

In the case that the forwarding state exists in the first request's node, then the second request is delayed by the first request as shown above, and is dealt with as if the second request begins after the first finishes.

Finally in the case that the forwarding state is in the second request's node, as we assumed the first request has already broadcasted to the second request's node before the second request begins, then the second request cannot begin until the first request has finished and sent a $DACK$ to the second request's node. This is why there is no simultaneous ordering in Figure 3.10b with the first request being forwarded the cache block from the second request's node.

All other nodes:

Lastly we have nodes that are not forwarding, nor have active requests: they are either in the *Invalid* or *Shared* state for this cache block. Because a request cannot add permissions to a node, only downgrade it, it does not matter if the second request's broadcast message arrives before the first's message, the $SWMR$ invariant will hold for both requests.

Thus, for the non-conflicting orderings, *1b* and *1c*, they are treated in the same way as non-simultaneous requests, and by the reasoning in Section 3.5, both requests satisfy the *MESIF* invariants.

Conflicting Simultaneous Requests

We show that the conflicting orderings, *1a*, *2a*, and *2b* in Figure 3.10b are treated similarly in *MESIF* and how this ensures the *MESIF* invariants hold.

Conflicting requests in *MESIF* are handled such that each conflicting request gets the forwarding state and passes it along. And all other nodes ensure there is a time that the request can coherently execute, where $SWMR$ holds true for the request.

Figure 3.11 shows the possible messages that can be sent for a pair of conflicting requests, absent the broadcast and reply messages. Each request is handled so that it gets the forwarding state from the previous holder of the forward state (**Forward Transfer**), which passes along the data and permissions (**Forward Data-Value**) and downgrades to a non-forwarding state (**Forward Uniqueness**).

The fourth *MESIF* invariant requires cooperation of all caching nodes. The general way we show this is for each node, that node does not violate *SWMR* for the request at some time known to the request. If all nodes do not violate, then *SWMR* holds for the request at a time known to the request (**Forward SWMR**).

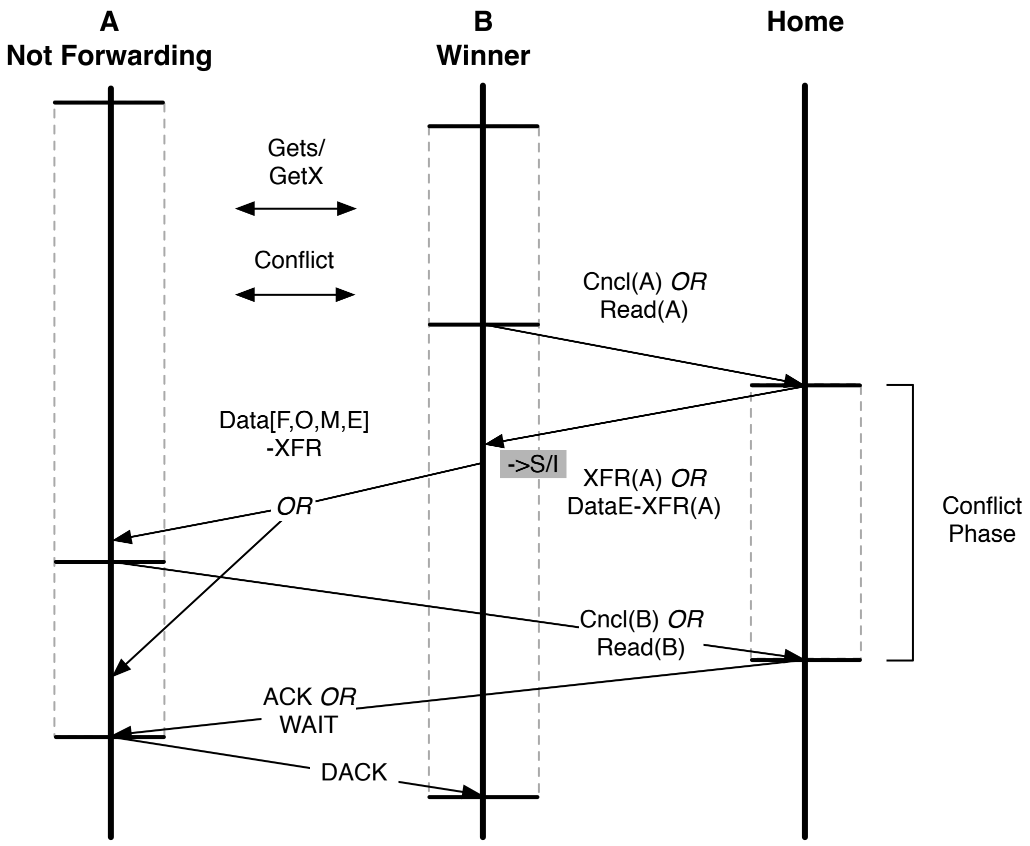


Figure 3.11: *The messages sent after the home phase, and all home messages for a pair of conflicting memory requests. For clarity, the broadcast messages and responses are not shown, all replies are received when the request sends a* Cncl *or* Read *message to home. The cache state changes are only shown for when B forwards the cache block to A. The relative times each request begins and when each request moves into the home phase, can vary. For example, the diagram shows that the home phase for node B begins before node A, but the reverse might be true – though only in the case of no forwarding state locally cached.*

We split the proof that pairs of conflicting requests follow the four *MESIF* invariants into two parts. First, we show that any pair of requests, with either request either a read or write, given **Pairwise Conflict-Aware** holds follows the timeline shown in

Figure 3.11 and thus holds the four *MESIF* invariants.

The second part of our proof joins together the potential conflicting orders for the broadcast and reply message shown in Figure 3.10b with the timeline in Figure 3.11. We show that **Pairwise Conflict-Aware** invariant holds for each order of conflicting requests.

Each of these orders (*1a*, *2a*, and *2b*), represents four different possibilities. The first and second request can each be either a read or write request. Additionally the state of the system can be any one of the stable system states, with three important categories: is the forwarding node uncached, cached at a non-requesting node, or cached at a requesting node.

The combination of these proofs shows both how *MESIF* handles conflicting pairs of requests, and that they are handled coherently.

All orderings in Figure 3.11 hold *MESIF* invariants:

One final assumption we make, and prove when we combine Figure 3.10b with Figure 3.11, is that if the forwarding state is locally cached somewhere, node B wins with its broadcast message arriving first (or wins by having the forward state originally). As shown in the previous section node A cannot be caching the forwarding state, else it could not be both simultaneous with node B and in the forwarding state waiting phase, waiting for B's $DACK$.

With this assumption and assuming the **Pairwise Conflict-Aware** invariant, node B receives all replies then sends either *Cncl* or *Read* to home; *Cncl* if it was forwarded the cache block, otherwise *Read*. Node B knows it is in conflict with A and what request A is undertaking. With the request to home, B includes a *conflict-list* to home containing one entry: node A; *Cncl*(A) and *Read*(A) in Figure 3.11.

When home receives the request from node B, it sees the non-empty conflict list and initiates a *Conflict Phase*, which only ends when it has received an *Read* or *Cncl* from all conflicting nodes. It knows when this is the case because by the **Pairwise Conflict-Aware** requires each conflicting pair of requests to put the other request in the conflict list sent to home. We expand on this when we deal with multiple conflicting nodes in the next section. Here, there is only node A is in the conflict list, so home will wait until node A sends it *Read* or *Cncl*.

Because we have assumed that node B is the winner if the forwarding state is locally cached, or is the first to send *Read* to home if not, home responds with *XFR* if B sent *Cncl* or *DataE-XFR* if B send *Get-Shared*. These messages order node B to forward the cache block when it is done with it, to node A. Additionally, *DataE-XFR* sends the cache block and forwarding state. Node B receives these the responds from home.

Let us show that the memory request from node B follows the *MESIF* invariants just before it forwards the cache block to node A. First, **Forward Uniqueness** is true as if forwarding state is locally cached, on forwarding the node would have downgraded and passed the forwarding state when the request from B arrived. Otherwise the forwarding state is uncached locally, and exists in main memory. We assumed B would be first to send *Read* to home, so up until now home has only sent data to one node. Thus **Forward Uniqueness** holds.

By this reasoning **Forward Transfer** holds, as node B got permissions and cache block data from the forwarding state. Whoever held the forwarding state, a local node or home, passed the latest value, so **Forward Data-Value** holds.

Finally **Forward SWMR** holds as $SWMR$ for B holds when node B received all replies from its broadcast message, and no other requests but B were forwarded the cache block between sending and receiving a response from home. We can use the reasoning from the previous section on non-conflicting but simultaneous requests to show this is the case.

For $SWMR$ to hold for node B, all nodes need to act correctly on the request message from node B. We showed that for nodes not in a request, including the forwarding node if it exists, they uphold (do not violate) $SWMR$ once they respond to a request message from either request. Because these nodes do not initiate a request for the cache block, they do not upgrade permissions, and so they do not violate $SWMR$ for B. Now for node A.

We know by **Pairwise Conflict-Aware** that each request knows which type of request the other is undertaking *by the time the request enters home phase.* Unfortunately, node A might not have reached the home phase by now, if so, it might not know what request node B is undertaking, and $SWMR$ for B might be violated by A[13]. If node A does know which request B is, then it downgrades as necessary and does not violate $SWMR$ for B.

There is only one way for node A to not know the request type of B when B enters the home phase. Node A must not have have received the broadcast message of B before A's request begins; the *1a* ordering in Figure 3.11, with node B the first request, and node A the second request. In this case node A already received the request message from node B, and does not violate $SWMR$ for B unless it was forwarded the cache block since. As we assumed node B won and received the forwarding state, node A could not have been forwarded the cache block, and so still does not violate $SWMR$ for B.

Therefore, all nodes are in cache states such that they do not violate the $SWMR$ invariant

[13]For example, node B might be wanting to write, but node A has the cache block as *Shared*, violating $SWMR$.

for node B by the time it receives all replies, and also when it receives cache block permissions and possibly data from home. So *SWMR* for B holds at a time known to node B, which is **Forward SWMR**.

Moving on in the timeline, after receiving a reply from home node B commits its memory operations[14] and forwards the cache block to node A as ordered by home. Node B then ensures it does not violate *SWMR* for node A by downgrading to *Shared* or *Invalid* as necessary. This forwarding is exactly the same as if node A sent a message request to B, and B responded by downgrading and forwarding, then delaying all (with one exception described below) other requests until it receives a *DACK*.

The forwarding from B to A is shown in Figure 3.11 as *Data[F,O,M,E]-XFR*. The difference in state depends on both the the type of requests made by A and B. For example if both A and B are write requests B sends *DataM-XFR*, signally that the data is dirty and that B has downgraded to *Invalid.* For A was a read request then B would send *DataO-XFR*, signally that the data is dirty and B holds it as *Shared.*

The forwarding from B can come either before or after node A receives all replies, sending to home *Cncl(B)* or *Read(B)* respectively. When home receives either of these messages, home already knows the cache block is not only locally cached, but already scheduled to be sent to node A by node B. Thus home sends *ACK* if it receives *Cncl*, or it sends *WAIT* on *Read*: wait for the cache block to be sent to you, then complete your request.

Figure 3.11 does not show the fact that the *Data[F,O,M,E]-XFR* might come after node A receives *WAIT* from home. If *WAIT* comes first the request waits until it is forwarded data before it completes the request. In either case, the last action before A completes its request is to send *DACK* to node B, freeing B to respond to any delayed requests.

Now let us show that the request at node A also follows the *MESIF* invariants. When node A receives the *Data[F,O,M,E]-XFR* from node B, we know **Forward Uniqueness** holds because node B had the forwarding state, and downgraded when it sent *Data[F,O,M,E]-XFR.* Additionally, **Forward Transfer** holds as A received data and permissions from the forwarding state; home's role is to merely say which node to transfer to, but there are no outstanding conflicting requests so home sent *ACK* or *WAIT*. When node B sent *Data[F,O,M,E]-XFR*, it sends the latest value for the cache block, so **Forward Data-Value** holds.

Finally, at the later of the times when *Data[F,O,M,E]-XFR* arrived and all replies to A's broadcast message, *SWMR* holds. For all non-requesting nodes, we know this is true from the previous section. For node B, since we assume **Pairwise Conflict-Aware**, node B knows the request type of node A, so can downgrade as necessary to not violate

[14] See Section 2.1 on committing and retiring.

SWMR for *A* when it forwards the cache block. By the same argument for node *B*, all other nodes but node *B* do not violate *SWMR* for *A*. Therefore *SWMR* holds, and it holds at a time node *A* knows, which is **Forward SWMR**.

Thus, both requests are handled so that the four *MESIF* invariants hold, and so coherence holds. Next we need to show that actual *MESIF* requests are dealt with in the timeline in Figure 3.11, upholding both **Pairwise Conflict-Aware** and that if the forward state is locally cached, then node *B* wins, otherwise node *B*'s home message arrives at home first.

Pairs of conflicting requests follow Figure 3.11 timeline:

Figure 3.12 shows the three conflicting orders of *MESIF* requests: *1a*, *2a*, and *2b*. We show that these conflict orders are handled in *MESIF* as shown in Figure 3.11 by ensuring that the **Pairwise Conflict-Aware** invariant holds, that the winning request has its request to home arrive first, and that all requests will eventually hit the home phase and so follow the timeline specified in Figure 3.11.

The proof is split into three parts. The forwarding state is uncached locally, it is cached in a non-requesting node, or it is cached at the winning node (node *B* in Figure 3.11).

For each of these orderings we need to show they uphold the **Pairwise Conflict-Aware** invariant.

If the forwarding state is uncached locally, then there is no forwarding node to delay requests. Thus there is a race for which request sends *Read* to home, with home receiving the *Read* first, to decide the winning request. In the *1a* and *2a* orderings, either the first or second request could do this. The **Pairwise Conflict-Aware** holds here because the broadcast message from the second request arrives at the broadcast phase of the first request, triggering a conflict, and a *Conflict* response, which tells the second request what type of request the first request is (in *1a*, the second request begins after the first request's broadcast message arrived at the second request's node).

For *2b*, we have an exception to the rule that all requests are delayed in the home phase. Because the first request in *2b* has already sent and received a *Conflict* response, both requests know they are in conflict and they know each other's request type: the *Conflict* message sends information. Thus, **Pairwise Conflict-Aware** holds for all orderings.

The purpose of **Pairwise Conflict-Aware** is to ensure that before *Cncl* and *Read* is sent to home, the conflict list that is sent to home is paired such that, if N_1's conflict list to home contains a node N_2, then N_2's conflict list to home contains N_1. This is immediate from **Pairwise Conflict-Aware**, and is utilized by home to ensure multiple (more than 2) conflicting requests are correctly handled by home.

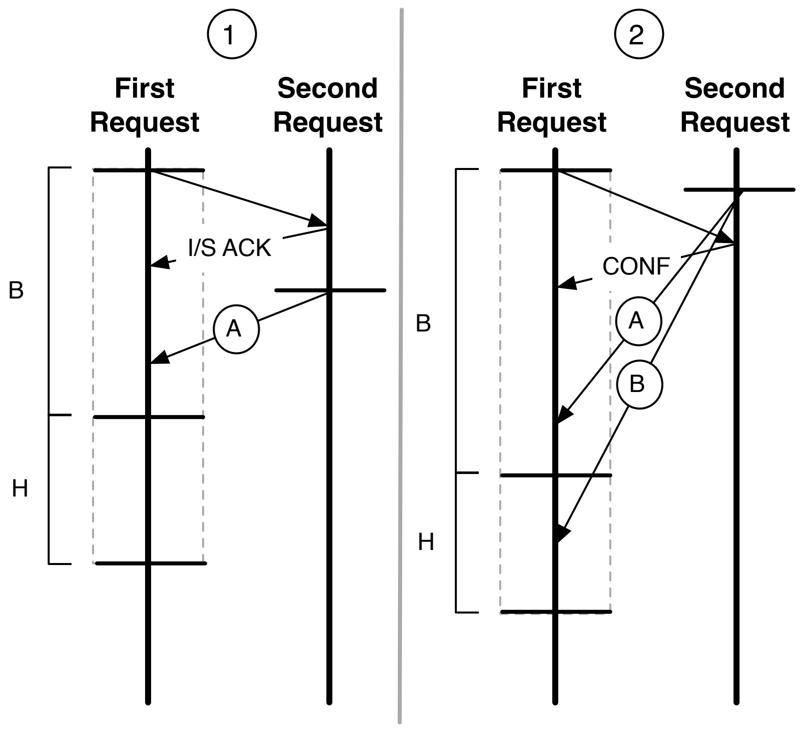


Figure 3.12: *Potential request messages for second request base on where the first request arrived, showing only the conflicting pair orders. The broadcast and home phase are abbreviated as 'B' and 'H' respectively. For space considerations,* Conflict *response has the label 'CONF' in this diagram.*

Because in *2b* both the first and second request know they are in conflict, it is safe for the first request to respond to the second request's message with *Conflict*, even in the home phase.

If the first request delayed the message, then we there would be deadlock. The first request would forward the cache block to the second request, but will not respond to the second request's delayed message until the second request sends a *DACK*. The second request could never send a *DACK* until it has all replies back from its broadcast request, which will not happen until it sends a *DACK*.

It is possible in *2b* for the second request to win when forwarding state is locally uncached. It could happen if the first request's *Read* is delayed significantly enough to allow the second requests *Read* to arrive at home first. Then home would respond to the second request by forwarding that cache block, and ordering the transfer to the first request.

No matter what ordering, **Pairwise Conflict-Aware** holds, and each request will eventually send a request to home, then follow the timeline as in Figure 3.11. The winning request's message to home by definition reaches home first.

Next, if the forwarding state is locally cached in a non-requesting node, then the winning request race is which request's broadcast message, the first or second, first arrives at the forwarding node. The forwarding node forwards the cache block to the winner and delays the other request until the forwarding node receives a *DACK* from the winning request.

So the winning request's *Cncl* to home will arrive first because of the delayed request message of the losing request in the forwarding node. Only when the winning request has received a reply from home does the winning request send a *DACK* to the forwarding node. This guarantees the winning node's *Cncl* to home arrives before the losing node's message to home.

Whether or not there is a locally cached forwarding state does not affect the broadcast messages and responses between the first and second request. Thus as with forwarding state not locally cached, **Pairwise Conflict-Aware** holds.

For the *1a* and *2a* orderings, both the first and second request can hit the forwarding node first. However, as the first request in *2b* is already in the home phase (received all replies) before the second request has received all replies, only the first request is the winner, and gets the forwarding state first.

Which ever is the losing request has its broadcast message delayed in the forwarding node. When the winning request receives a *XFR(OTHER-REQUEST)* from home, it simultaneously sends out a *DACK* to the forwarding node. When this *DACK* arrives, the forwarding node then replies to the losing request's broadcast message, allowing it to receive all replies and send either a *Cncl* (if the forward from the winning node has arrived) or *Read* (if the forward has not yet arrived) to home.

Finally the case that the forwarding node is at the winning request. If the winning request does not delay the losing request's broadcast message, then there is no forwarding node to delay the losing node from receiving all replies and immediately sending *Read* to home, and we cannot guarantee that the winning requests message to home arrives first.

We know that the winning node must have started before the request from the other node arrived, so it is not possible that in ordering *1a* for the winning node to be the second request. In all other orderings, the losing request's broadcast message arrives while the winning request is active, so the winning request can delay responding much like the forwarding node does.

The delay is until the winning request knows its *Cncl* has arrived at home. Once the winning request receives *XFR(OTHER-REQUEST)*, the winning request sends a *Conflict* response to the losing request, then forwards the data to the losing request. Again depending if the losing request gets all replies first or receives the forwarding first, it sends either *Read* or *Cncl* to home.

Because the winning request either gets a *Conflict* reply from the losing request, or get the losing request's broadcast message (which is delayed) before sending *Cncl* to home, the winning request knows about the conflict before it sends to home. The losing request's broadcast message will be responded to with *Conflict*, so the losing request knows it is in a conflict before sending *Read* or *Cncl* to home. Thus **Pairwise Conflict-Aware** holds.

Thus, for any pair of conflicting orderings as shown in Figure 3.12, **Pairwise Conflict-Aware** holds, the winning request's message to home arrives first, and all requests will eventually hit the home phase and so follow the timeline specified in Figure 3.11.

Thus any pair of simultaneous, including conflicting, requests hold the four *MESIF* invariants. In the next section when we discuss multiple conflicting requests, subsequent conflicting requests after the first all follow the same pattern as node A in Figure 3.11 except for potentially forwarding the cache block as node B does.

Conflicting Pair Example

Putting it all together, we present an example of the complete process of conflict resolution in *MESIF* occurs in the case of two conflicting requests to the same cache block. Figure 3.13 gives an example of a conflicting pair of requests, nodes B and C, with a third node, A, holding the forwarding state in the *Forward* cache state. The ordering here (see Figure 3.10b) is the *1a* ordering.

The simultaneous requests by nodes B and C are requesting the cache block to read and write respectively. Requestor B broadcast reaches A first, which forwards the cache block to B. It also reaches C before the request of C has begun, thus C replies with an $IACK$. Before B has all replies and transitions to the home phase, the request at C begins, broadcasting its *GetX* request. C's request is delayed at A when it arrives, but arrives at B during its broadcast phase resulting in the *Conflict* reply. Thus B and C are now conflicting requests.

Node B is forwarded the cache block from A. The forward in the form of $DataF$, is the last reply for B, which now enters the home phase sending *Cncl* to home along with a conflicting node list consisting of only node C.

Home receives the *Cncl* request, and responds with an *XFR* reply, informing B that it should forward the cache block to C. When node B receives the reply from home, it sends out a $DACK$ to the forwarding node A. This releases A from its *waiting* phase, allowing it to respond to the delayed message from C and downgrade to *Invalid.* Delaying prevents a time-warp situation as shown in Figure 3.1 as the $DACK$ is sent only after home has

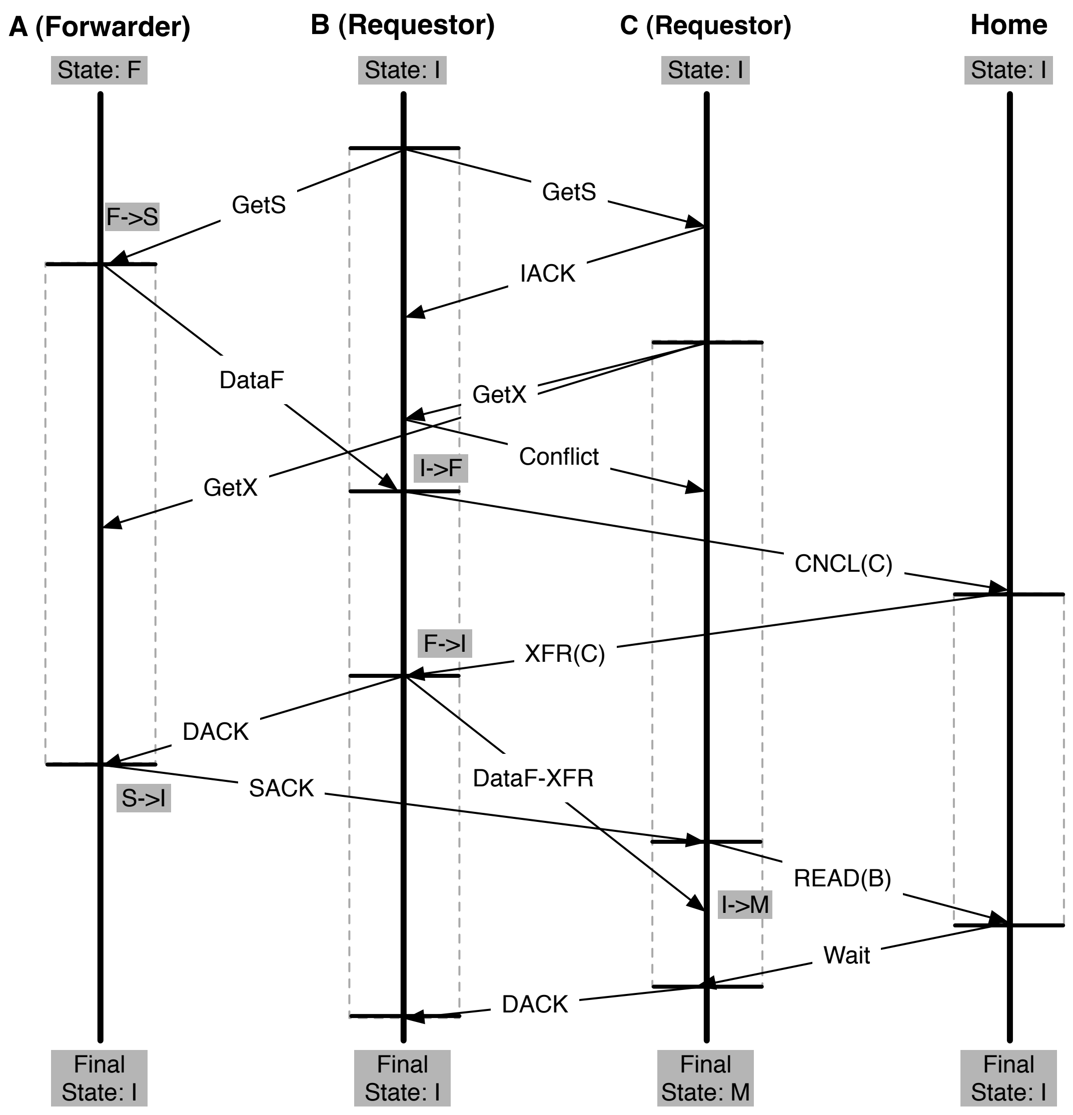


Figure 3.13: *All messages required to complete a pair of conflicting requests to a cache block.*

received the *Cncl* message from the winning node *B*.

As well as sending the *DACK* to node *A*, *B* also forwards the cache block to node *C* as ordered by home. Node *B* remembers that the conflicting request is a *GetX* request, and downgrades to *Invalid*. On node *C*'s side, it finally receives the reply from *A*, an *SACK*, which is *C*'s final reply, and it enters the home phase. Node *C* sends an *Read* request to home as it was not forwarded the cache block. The conflict list contains a single node, node *B*.

Home receives the *Read* from *C*, but knows from the previous *Cncl* that it was to receive a message from *C*. Since home knows the forwarding state is locally cached due to the *Cncl* from *B*, it sends *WAIT* to *C*, telling *C* to wait for the cache block to be forwarded, and then complete its request. Home has now received all *Read* and *Cncl* messages, and exits the conflict phase.

Before the *WAIT* arrives at *C*, the cache block is forwarded from *B*. Then, *WAIT* arrives, completing the memory request at *C*, which finishes by sending a *DACK* to the forwarding node *B*. Finally, node *B* receives the *DACK*, and completes its memory request.

3.6.2 Simultaneous with One Request Per Node

With only pairs of simultaneous nodes to deal with, the race to decide the winning request is decided by which request hits the forwarding state first, if it is locally cached, or which request hits home first. The losing request is then scheduled by home to be forwarded the cache block by the winning request.

With more than two conflicting requests, the ordering of subsequent requests after the first request is decided by home when it receives the conflicting list in *Read* or *Cncl* messages. If there are many simultaneous requests, but the winning request is in conflict with at most one other request (and that request only is in conflict with the winning request) then this is dealt with in the previous section on pairs of simultaneous requests. We deal here with conflicting orders of more than two requests.

The main goal for the home node is that it must keep track of which nodes to schedule the forwarding state to be transferred to, and which already have been scheduled and to schedule requests such that each request is forwarded exactly once. Home utilizes the **Pairwise Conflict-Aware** invariant to aid it in this task.

Each requesting node collectively ensures that it does not violate *SWMR* for all of the requests this node is in conflict with. Non-requesting nodes also don't violate *SWMR* for any request as shown already in the previous section. The forwarding state jumps from node to node in the order specified by the home node.

In this section we allow for any ordering of requests and their messages, however we restrict each node (or local cache and core) to, at most, one request until all simultaneous requests have been completed. This restriction is important as we do not use unique request/node identification. Instead, requests are associated with the node they come from, and conflicting requests are tracked at the node level.

Allowing multiple requests from the same node has some corner cases that are dealt with in Section 3.6.3. For example in the *2c* order shown in Figure 3.10b, the broadcast message from the second request might hit the first request's node as it is in the middle of a new request. Or a request might get two broadcast messages from the same node before this request enters the home phase.

We split the discussion of how *MESIF* handles multiple conflicting requests into three parts. First we show that, given an assumption about how home schedules forwarding, conflicting requests will always make forward progress; there is no deadlock. To achieve this, home may need to fix a miss match between the ordering of requests decided by home and the ordering of requests as decided by the requests themselves via non-conflicting message delay.

We then show how conflicting requests are handled so that the *MESIF* invariants hold, and that any ordering of requests holds the *MESIF* invariants. This will be based on how pairs of conflicting requests ensure the *MESIF* invariants, allowing us to use the arguments from the previous section on conflicting request pairs.

Finally we show that home can ensure the first assumption that home can schedule requests so that each request is forwarded the cache block precisely once. This will rely on the **Pairwise Conflict-Aware** invariant and that each node can only have one request. Only once all conflicting requests have completed can nodes begin another request for the cache block.

Multiple Conflicting Requests Do Not Deadlock

Here we first show that, given an assumption about how home orders requests, requests make forward progress and there is no deadlock. We prove how home ensures this assumption when we discuss how home handles *Cncl* and *Read* requests with non-empty conflict lists.

The assumption about home is:

Home assumption: Schedules requests such that each of the conflicting requests will be scheduled to receive the forwarding state from precisely one request. Requests are added to the home ordering when home receives a *Cncl* or *Read* message, and are added at the end of the ordering.

This assumption means that home will schedule requests such that the forwarding state hops from one request to another, until we reach the last request in this chain of forwarding, which keeps the forwarding state and completes.

Until this section we have assumed at most two simultaneous requests active at anytime. Because the only way a request can delay another request is by delaying the request's broadcast message, we went into detail how request delay is only temporary, that requests always eventually complete.

The reason with only pairs of conflicting requests is that the winning request cannot be delayed by the losing request, so the winning request eventually completes. This means that the winning request will eventually reply to the losing requests broadcast message, as well as unlock the (potentially) delayed broadcast message from the forwarding node (via *DACK*) if locally cached. Thus the losing request also will complete. There are no issues of deadlock with pairs of conflicting requests.

The winning request is ordered before the losing request, no matter whether the other requests are conflicting or non-conflicting. The winning request wins by being either first to the locally cached forwarding state or first to have the *Read* message received by home. Home plays no part in deciding who is the winning request. With only two requests, home only has one ordering it can chose from which is the winning request is ordered before the losing, or second, request.

When considering three or more simultaneous requests the order of requests, after the winning request, is decided by home and not the requests themselves. This ordering is represented in home by a single linked list called the *requestor-queue*. Each link in the *requestor-queue* represents the forwarding of the cache block from one request to another request. The head of the *requestor-queue* is the winning request, and the eventual tail of the *requestor-queue* is the request that does not forward at all (it receives *ACK* or *WAIT* from home). The *requestor-queue* represents a portion of the *forwarding-chain* discussed in Section 3.3.

Now we can show that conflicting requests make forward progress, that no request are deadlocked, and that requests will eventually complete.

First, we know that the head of the *requestor-queue* is the winning request, which is ordered not by home but by the race to the locally cached forwarding state, if it exists, or the first to have the *Read* message received by home. Even though there could be multiple simultaneous requests, this does prevent the winning request from receiving the cache block from the forwarding node or getting the *Read* to home first. With multiple requests, the winning request may have any number of nodes in the conflict list, and it may by delaying any number of broadcast messages from simultaneous but non-conflicting requests.

Forwarding can only occur after a request has both received the forwarding response from home, and the request has received all replies to its broadcast message. So the head of the *requestor-queue* will eventually forward the cache block, since there are no nodes that can delay the winning request. A request that forwards only completes after it receives a *DACK* from the request it forwards to, sent when that request itself forwards the cache block or simply completes.

As more and more requests sent *Read* and *Cncl* requests to home with non-empty conflict lists, the more requests are added to the tail of the *requestor-queue*.

Take any request, *R*, in the *requestor-queue* that is not the winning request we assume that the previous request has forwarded the cache block to *R*. We will use induction to show that all requests eventually forward (or the final request at the tail just completes) and thus, all requests eventually complete.

There are only two ways a request can be delayed. It can be delayed waiting for the forwarding state to be forwarded to it, but the assumption is that R has the cache block forwarded to it. The other way for a request to be delayed is by delaying the reply to one of the request's broadcast messages.

Figure 3.14 shows the situation with request R just forwarded the cache block from the previous request, R_p. If R_p is the winning request, it sends a *DACK* to the forwarding node which will complete and then respond to any delayed request messages. Or if the cache block is not locally cached in the forwarding state then there is no *DACK* to send. On the other hand if R_p is not the first request, R_p sends *DACK* to the request that forwarded the cache block to R_p. This will complete the request, allowing it to respond to any delayed request messages.

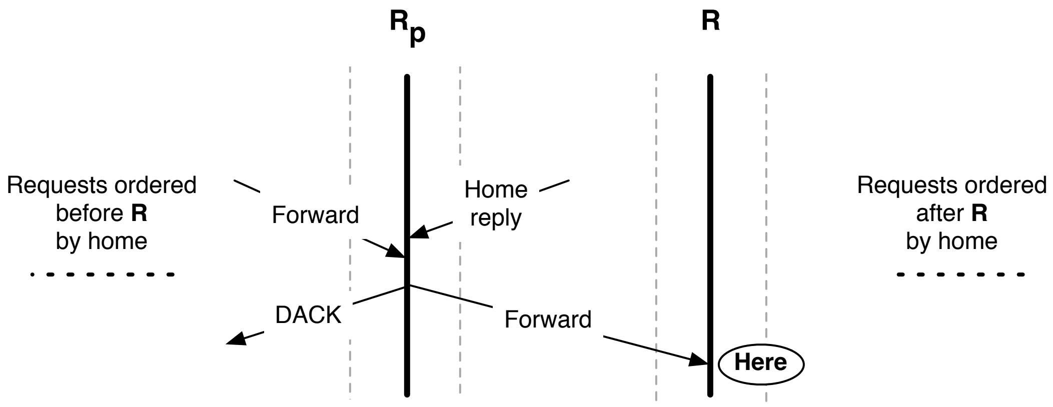

R p
R
Requests ordered before R by home
Forward
Home reply
Requests ordered after R by home
DACK
Forward
Here

Figure 3.14: *The inductive step where request R is forwarded the cache block by R_p. Must show that R also forwards the cache block on, or completes its request as there are no more requests to forward to. Shows R_p sending a* DACK *to the forwarder, but if R_p is the first request, there may be no forwarding node to* DACK.

Thus, the request before R_p if it is not the winning request will complete, else R_p is the winning request. We need to show that R will eventually either forward the cache block which sends a *DACK* to R_p completing it. Or R is the last request, the tail of *requestor-queue*, and simply completes also sending a *DACK* to R_p completing it. Then by induction all requests from the winning request to the last request complete; no requests deadlock.

Now we need to show that no request can delay the broadcast messages from R so that R will eventually receive all replies and enter the home phase. By the first assumption from home, home will respond by ordering R to forward to the next request in the *requestor-queue* or to complete as there are no more request to forward to.

Below is a list of all possible ways the request broadcast from R can be delayed. For each of these, we will show how each does not delay R indefinitely. Only requests can delay R as we have shown the winning request completes, and so the non-requesting forwarding node (if it exists) will receive a *DACK* then reply to all delayed broadcast messages.

1. Requests ordered before R.
2. Requests not currently ordered by home.
3. Requests ordered after R:
   (a) Conflicting with R.
   (b) Non-conflicting with R, with R ahead of the other request.
   (c) Non-conflicting with R, with R behind of the other request.

We already showed that all requests before R_p will eventually complete. When they do, they will respond to R if they have not done so already. Because R_p is forwarding the cache block to R in Section 3.6.1 we showed that R_p, conflicting or non-conflicting, will not delay the broadcast message of R. Thus all requests ordered before R will not delay the broadcast message of R.

If a request R_{uo} (unordered request) is not yet been ordered by home then R_{uo} is not conflicting with any request that has already sent *Read* or *Cncl* to home. If it was ordered by home, R_{uo} would have appeared in one of those requests's conflict list and have been ordered by home. The R_{uo} request can either be conflicting with R, in which case R_{uo} does not delay R, or non-conflicting with R. Because R has not yet received all replies then for R and R_{uo} to be non-conflicting, R_{uo} must have received all replies and entered the home phase, delaying (when it arrives at R_{uo}) the broadcast request of R.

As R_{uo} is non-conflicting then it must have sent a broadcast request to R and received a response before R began.However R_p has received all replies and is also non-conflicting with R_{uo}, as R_p has sent a conflict list to home, but it did not contain R_{uo}. This means R_{uo}'s broadcast request must be either still on route to R_p or delayed at R_p and therefore R_{uo} has not received all replies. This contradicts with R_p having received all replies, so R_{uo} cannot be non-conflicting with R and also have entered the home phase. R_{uo} cannot delay R.

All requests ordered after R can either be conflicting with R and so will not delay R's broadcast message. Otherwise they are non-conflicting. If they are non-conflicting, then the ordering between these requests and R are the simultaneous ordering *1b* or *1c* as shown in Figure 3.10b.

If R is the first request, then clearly it cannot be delayed by the second request; R already has received a reply from the second request's node. Now all we need to show is if R is the second request, the first request does not delay R. Then no request can delay R's broadcast message.

We discussed that other than the winning request ordering itself to be first in the ordering, the ordering of all other requests are decided by home. However, requests attempt to order themselves just like the winning request ordered itself. When we have two requests that have not yet completed and are non-conflicting but simultaneous, then the delaying request will not deal with the delayed requests broadcast message until after the delaying request is forwarded the cache block. Deadlock could occur if home instead orders the delayed request to be forwarded the cache block before the delaying request.

This ordering miss match is shown in Figure 3.15 where we have request R ordered before a request R_n that is non-conflicting with R. The figure shows R_n immediately after R but this can occur with *any* request ordered after R by home. This ordering is only possible in the *1b* simultaneous ordering shown in Figure 3.10b, with R_n delaying the request of R until it is forwarded the cache block. But R_n will never get the cache block since R must forward it on first, which cannot happen until it sends *Cncl* to home and home replies with an *XFR* telling R to forward to the next request on the *requestor-queue*. We have deadlock unless home sorts this situation out.

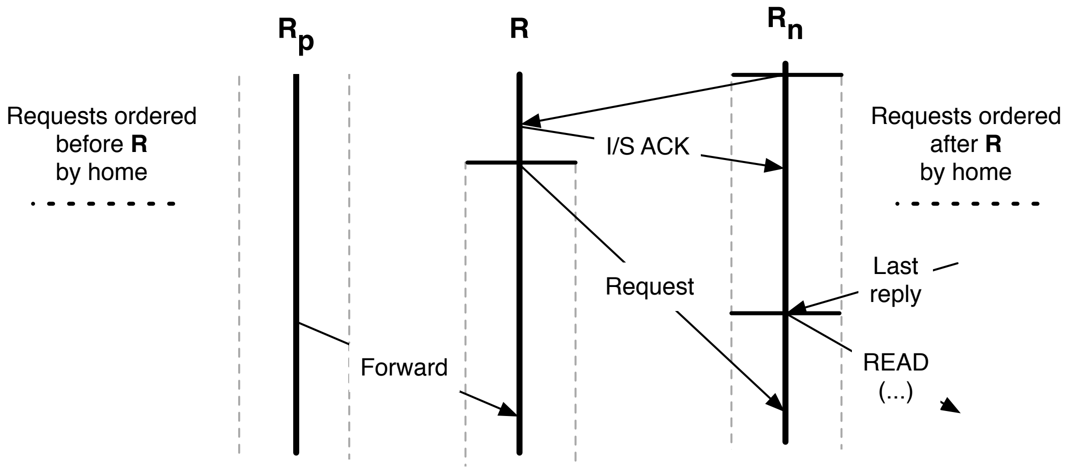

R_p
R
R_n
Requests ordered before **R** by home
Requests ordered after **R** by home
I/S ACK
Request
Last reply
READ (...)
Forward

Figure 3.15: *Miss match between the ordering of requests as decided by home, and the ordering as decided by the requests themselves. Only possible with the* 1b *ordering shown Figure 3.10b. Request R is scheduled to be forwarded the cache block before request R_n but this request is non-conflicting with R, and is delaying the broadcast message of R.*

How this can occur is discussed when we show how home orders requests and assures the home assumption. To solve this we note that if R_n is non-conflicting with R, it must be in the home phase by the time the broadcast request from R arrives. Home receives the *Read* message (it cannot be *Cncl* as R is stuck with the forwarding state) and compares the nodes in the conflict list with the nodes ordered before R_n in the *requestor-queue*.

Any request ordered before R_n that has 1) not send a message to home yet[15] and 2) does not appear in the conflict list of R_n, then R_n is in *virtual conflict* with this request. Virtual conflicts are detected by home, and elevated into real conflicts.

Home replies to the *Read* of R_n with either *WAIT-XFR* (if there are requests ordered after R_n) or *WAIT* (if there are no requests ordered after R_n[16]) and appends the *Conflict-Update* message to this reply. This update conflict includes all the requests that are in virtual conflict with R_n.

When R_n receives *Conflict-Update*, it adds those requests to its conflicting list and checks if any delayed requests are requests in virtual conflict with R_n, replying with *Conflict* if so. Because R_n is in conflict with these requests, it must not violate *SWMR* for them and so downgrades as necessary. Each of the requests in virtual conflict will not have reached the *commit* point, since R_n was delaying their request message, so **Forward SWMR** will still hold for these requests as we will show later.

The requests in virtual conflict with R_n will now receive the *Conflict* reply, and add R_n to their conflict list, and ensure they also do not violate *SWMR* for R_n.

The home assumption assumes that new requests are added at the end of the home ordering, so if R is currently ordered before R_n, it was ordered before when R_n sent *Read* to home. Thus, R will be in virtual conflict with R_n and a part of the *Conflict-Update* sent to R_n.

Figure 3.16 shows how the reply of *Conflict-Update* to the *Read* of R_n ensures that R's broadcast message will eventually be responded to by R_n with the *Conflict* message.

Thus R will not be delayed by any non-conflicting request ordered after it by home, even if the other request delays the broadcast message of R.

In Section 3.6.3 when we discuss all orderings of simultaneous requests, we rely on the **Pairwise Conflict-Aware** to ensure the home assumption: that each conflicting request is forwarded the cache block precisely once. With R_n virtual conflict, home in effect adds R to the conflict list of R_n. Now we need to be sure that R will have R_n in its conflict list.

[15]Because we only have an issue with requests delayed from receiving all replies, any request that has sent a message to home cannot be delayed by R_n and is ignored here.

[16]As we will explain later on, the last request can only receive the *WAIT* message once all other requests in the *requestor-queue* have sent home messages because the home assumption assumes new conflicting requests are added at the end of the order, if the request at the end of the order has already been told to not forward, we cannot add any more requests to the ordering.

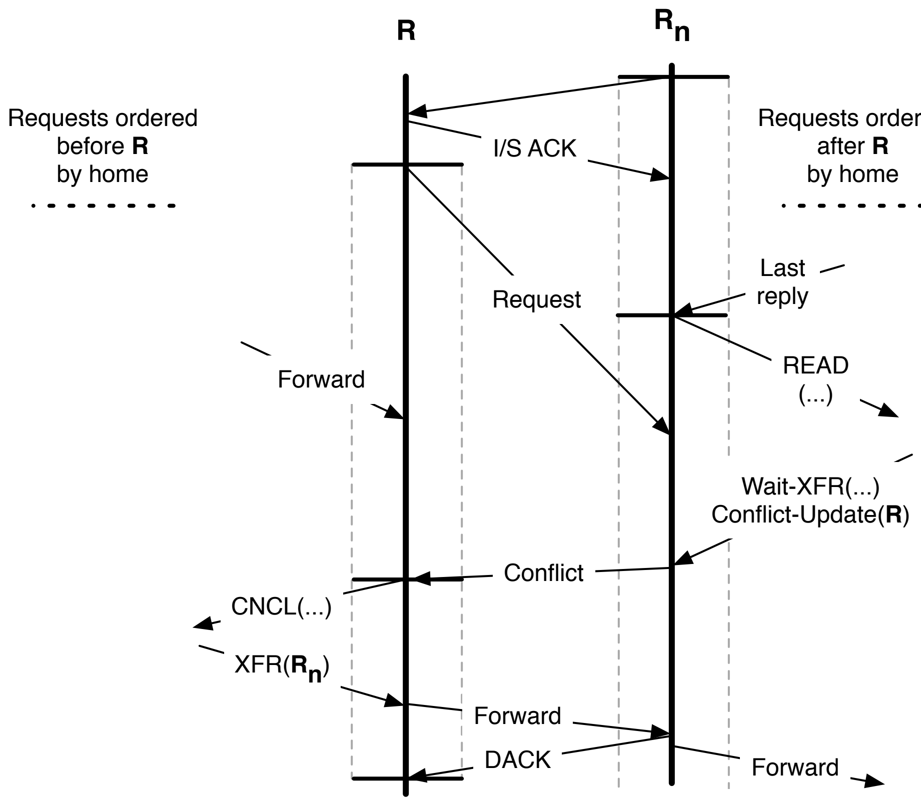


Figure 3.16: *The* Conflict-Update *message from home updates the conflict list in R_n, which frees the delayed broadcast message from R by replying with* Conflict*; the two requests are now in virtual conflict with one another. Again we show R_n immediately after R in the ordering at home, but this is not necessary. R_n can be any request after R.*

The R_n request cannot complete its transaction until it is forwarded the cache block, which will only occur after all requests ordered before it have sent a *Read* and *Cncl*. So all requests before R_n have receives all replies. For R_n to be in virtual conflict with a request *Req* ordered before it by home, it must begin before *Req* begins. This then implies that the ordering between R_n and *Req* is the *1b* ordering, and so *Req* will receive a *Conflict* message from R_n.

We have showed that no request will delay the broadcast message from R, so R will eventually get all replies and send a *Cncl* or *Read* to home. If this request is not the last request, then home will order this request to forward to the next request in the *requestor-queue*. Otherwise R is the last request and simply completes. In both cases, R sends a *DACK* to R_p, which will eventually arrive, completing R_p.

As there are only finite number of requests, eventually one request will be the last request, and so all requests (eventually) complete; no requests deadlock.

Now we need to show that these requests also hold the four *MESIF* invariants.

Multiple requests hold *MESIF* invariants

Each request will be forwarded the cache block, and then forward it on to the next (unless it is the last request). As each request that is delayed will eventually be completed (there is no deadlock), requests will eventually receive all their replies and enter home phase. Home will order the requests, and then the requests will pass the forwarding state on. Now we need to show that the *MESIF* invariants actually hold.

The first three *MESIF* invariants, **Forward Uniqueness**, **Forward Transfer**, and **Forward Data-Value**, ensure that only way to get correct cache block data as well as permissions is to be forwarded the unique forwarding state. As we demonstrated above, the forwarding state hops from request to request, each request loses the forwarding state when it forwards, and passes the current data value for the cache block. Requests only receive permissions and data in a forwarding, and the forwarding state only exists in a single node at any time. Thus the first three *MESIF* invariants hold.

The last *MESIF* invariant, **Forward SWMR**, requires cooperation between all the caching nodes. To show that **Forward SWMR** holds for each of the conflicting requests, we consider the relationship the request has with all other nodes. We consider two relationship categories for a request and a node. Either the request is not in conflict with the node's request (or it has not started a request), or this request conflicts with the request at the node. If it is not in conflict, then when the reply for the broadcast message comes back, the non-conflicting node cannot violate $SWMR$ for the request. If it is in conflict, then each request ensures they do not violate $SWMR$ for the other. We first show that the non-conflicting nodes do not violate $SWMR$ for the request.

For each request R, we need to show that the non-conflicting nodes do not violate $SWMR$ for R by the time R is forwarded the cache block or receives a reply from home, whichever is later. Non-conflicting nodes can either have simultaneous but non-conflicting requests, or all non-conflicting requests are not simultaneous with R.

If there are no requests active when R is, then by the time the reply to R's broadcast message has returned R guarantees that this node does not violate $SWMR$ for R for the duration of R's request. Only a request can upgrade cache block permissions, once the non-requesting node has responded to R's broadcast message, it cannot upgrade and violate $SWMR$ for R.

Otherwise, the node has a simultaneous but non-conflicting request with R.

Now we only need to show that all conflicting nodes do not violate $SWMR$ for R by the time R is forwarded the cache block or receives a reply from home, whichever is later. Then have shown that this is the case for all cacheing nodes, so **Forward SWMR** holds for R, and for all of the conflicting requests.

Figure 3.10b shows all possible orderings of pairs conflicting (and non-conflicting) requests. We reasoned for each conflicting ordering, other than *2c*, that the **Pairwise Conflict-Aware** invariant held, which allowed each request to not violate *SWMR* for the other request by the time each request is forwarded the cache block or receives a reply from home, whichever is later.

Likewise for virtual conflicts as we discussed above. Virtual requests only occur between a pair of non-conflicting requests, one request, R_n is in the home phase and is ordered after (by home) the other request R which has already has been forwarded the cache block. Request R eventually receives a *Conflict* respond from R_n, and R_n ensures that it does not violate *SWMR* for R. Considering request R_n is ordered after R by home, R will commit before R_n has the cache block forwarded to it. So between the time of receiving the *Conflict* reply and committing, R_n will not violate *SWNR* for R.

Adding multiple conflicting and non-conflicting request does not change this. If a pair of requests are conflicting then they must be aware of the conflict before their respective home phases, and they will know what request the other is undertaking. For all the conflicting orderings in Figure 3.12, we have the *SWMR* guarantee.

Except, we did not show it for the *2c* conflict ordering shown in Figure 3.10b as this ordering is not possible when there are only two simultaneous requests. When we have multiple conflicting requests, this ordering is now possible. Figure 3.17 shows how a *2c* ordering is possible with only three requests, and that it can be conflicting.

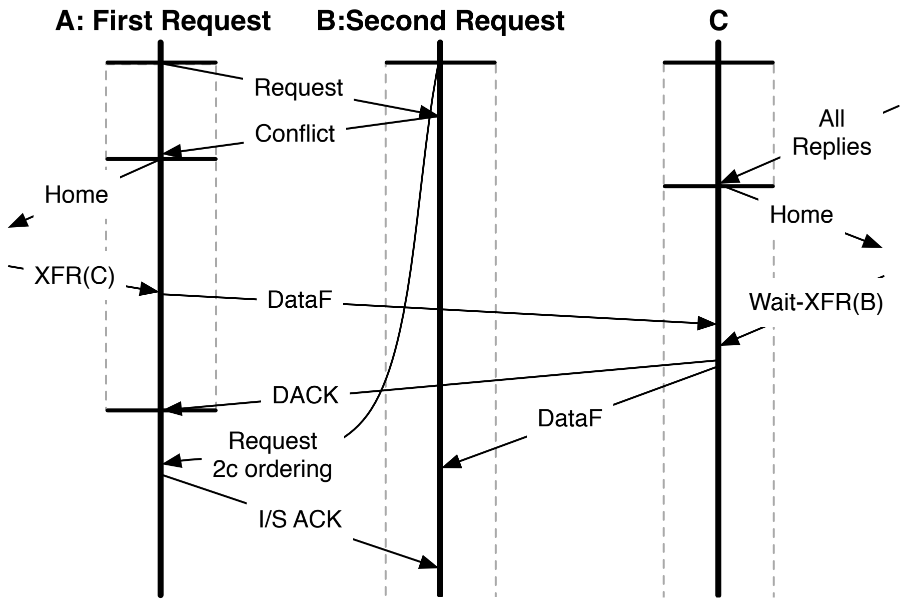


Figure 3.17: *With multiple simultaneous requests, ordering* 2c *is now possible. Will still execute concurrently. The second request cannot delay the first, as the first has already completed its request.*

For the *2c* ordering, the first request at node A knows it is in conflict with node B (as well as in conflict with node C), and what type of request the conflicting request at node B is in (it is passed with the *Conflict* reply) before entering the home phase. The second request at node B also knows it is in conflict due to receiving the message broadcast from the first request. Node B knows before it has entered home phase, since at least one request is delayed until after the first request finishes which must be after the first request's broadcast message to B has arrived. Therefore in the *2c* ordering, the **Pairwise Conflict-Aware** invariant also holds.

For the diagram we can also see that both requests ensure they do not violate the $SWMR$ invariant for the other request. When the second request receives the request from the first request, it downgrades as required, and does not get upgraded cache block permissions until after the first requests ends. Thus, the first request, by the time it receives all replies, can guarantee that node B will not violate the $SWMR$ invariant (it still needs to make sure all nodes are in a sufficient state to guarantee the $SWMR$ invariant).

Conversely, by the time the second request has the reply from the first request, then node A will not violate the $SWMR$ invariant for the second request until the second request completes – which is irrelevant, since the second request will have committed by then[17]. The reason that this lasts for the entire second request is that, we assumed that until *all* conflicting requests have committed, there can be at most one request per node. So node A will not begin another request until the second request completes.

Thus, as all conflicting orderings, all non-conflicting requests, and all non-requesting nodes do not violate $SWMR$ for a request in a multiple conflicting ordering. And that this is true at the time each request is forwarded the cache block or receives a reply from home, whichever is later, **Forward SWMR** holds for all conflicting requests.

Now we need to show how home ensures that each request will be forwarded the cache block precisely once, and this order is decided by the time the request sends *Read* or *Cncl* to home.

Multiple Conflicting Requests and Home

To reiterate the assumption we made of home in the previous section:

Home assumption: Schedules requests such that each of the conflicting requests will be scheduled to receive the forwarding state from precisely one request. Requests are added to the home ordering when home receives a *Cncl* or *Read* message, and are added at the end of the ordering.

[17]See Section 2.1 about committing.

Here we will show that home can guarantee this assumption. We also discuss a restriction on the last request in the *requestor-queue*.

Before receiving the first request with a non-empty conflict list (the winning request) home is not in the *conflict phase*. When home receives the (non-empty) conflict list from the winning request, home constructs the *requestor-queue* and enters the conflict phase.

The *requestor-queue* is a singly linked list of nodes with requests and represents the order of forwarding of the forward state by the conflicting requests. The head of the *requestor-queue* is the winning request as the winning request does not need home to schedule the forwarding of the cache block from another request. All other requests in the *requestor-queue* will be forwarded the cache block by the immediate previous request in the *requestor-queue*. When adding new requesting nodes to the *requestor-queue*, they are always placed at the end of the list as specified by the home assumption.

For all subsequent *Read* and *Cncl* messages after the winning request's message was received, beginning the home conflict phase, home has two decisions to make before it replies with one of the home response messages.

First, should home add the request to the tail of the *requestor-queue* or not. And second, for each request in the incoming conflict list, should it add that request to the tail of the *requestor-queue* or not. If the request is not yet on the list, it is added to the end of the *requestor-queue*, otherwise it is not added.

Because home does not change the order in the *requestor-queue*, once home has finished adding the requests to the tail of the list R_t, then the (now previous) tail of the list will now be ordered after at least one request. Request R_t will now need to be scheduled by home to forward the cache block.

The *Read* or *Cncl* sent by tail of the *requestor-queue* cannot be processed by home until all other requests in the *requestor-queue* have sent *Read* or *Cncl*. If we did send $WAIT$ or ACK to the tail of the *requestor-queue* before all other requests in the *requestor-queue* have sent a message to home, we would not be able to add any new conflicting requests to the *requestor-queue*; the home assumption would not hold. So home delays (by locally buffering the requests) the *Read* and *Cncl* until all other requests have sent to home.

Because the tail of *requestor-queue* is ordered after all other requests, it cannot delay any earlier requests as long as home makes sure the tail request is not in virtual conflict with any earlier request. If it is, home sends just *Conflict-Update* to the tail request, and processes the tail's *Read* and *Cncl* later.

By only adding a request to the *requestor-queue* if it is not present home will never forward more than once to a request. All we must show to demonstrate home holds the home assumption is that each conflicting request will be added to the *requestor-queue*, and so be forwarded the cache block precisely once.

To show that all conflicting requests will receive the forwarding state at least once, we show that a request that was not forwarded the cache block cannot be in conflict with any of the requests.

Let R_{uo} (unordered request) be such a request, which began after the winning request[18]. This request is not currently in the *requestor-queue*. We will show that either R_{uo} eventually is ordered by home because it becomes conflicting. Else R_{uo} is delayed until all requests in the *requestor-queue* complete, and is not conflicting.

Because of **Pairwise Conflict-Aware** (and virtual conflicts), R_{uo} is not in conflict with any request in the *requestor-queue* that has already sent *Read* or *Cncl* to home. If it was in conflict, then home would have added the request to the *requestor-queue*.

If R_{uo} is in conflict with a request in *requestor-queue* then that request's *Read* or *Cncl* message has not yet arrived at home. We showed previously that any request that is already ordered by home, in the *requestor-queue*, cannot deadlock, and so will enter home phase. Because of **Pairwise Conflict-Aware** invariant, R_{ou} will be in this conflicting requests conflict list, so home will add R_{ou} to the *requestor-queue*.

If R_{uo} is not in conflict with any request in the *requestor-queue*, R_{uo} has not yet been forwarded the cache block (since it is not in scheduled to do so as it wont be in the *requestor-queue*). R_{uo} then, has not yet completed, it is still active. For all requests in the *requestor-queue* R_{uo} is either before or after the request in one of the non-conflicting orderings.

If it is after all requests in the *requestor-queue* the R_{uo} will not receive all replies until all requests in the *requestor-queue* have completed; R_{uo} was not a conflicting request to be ordered here. If it is before any request R in the *requestor-queue* then it must have sent *Read* or *Cncl* to home. Home will then add R_{uo} to the tail of the *requestor-queue*, so R_{uo} is now ordered.

Thus in all cases, the unordered request R_{uo} eventually becomes ordered, and is scheduled to be forwarded the cache block. Or it is non-conflicting with all the other requests, and is delayed until all conflicting requests in the *requestor-queue* complete, and there is another race for the next winning request.

So each conflicting request is scheduled to receive the forwarding state never more than twice and at least once: precisely once. And requests are added to the ordering at the end. The home holds the home assumption.

Once home has added any new requesting nodes to the tail of the *requestor-queue*, home checks if it had a *Read* or *Cncl* message from the previous tail. If so, then home can

[18]If R_{uo} began before, the winning request would have been in conflict with R_{uo}.

respond to that message with an *XFR* or *WAIT-XFR*, ordering the previous tail to forward to one of the newly added nodes. At the time home received a *Read* or *Cncl* from the tail of the *requestor-queue*, home found no nodes in the conflict list that were not already in the *requestor-queue*, else those nodes would have been added to the tail of the *requestor-queue*.

The final action by home is to check for virtual conflicts as discussed previously. Requests that are ordered before the request which has just send *Read* to *Cncl* to home may be in virtual conflict.

To demonstrate the use of the home and the *requestor-queue*, we finish by giving an example of three conflicting requests. Figure 3.18 shows the *Read* and *Cncl* messages sent to home by each of the three conflicting requests, including the contents of the conflict list. Figure 3.19 shows how the *requestor-queue* changes as each message is received by home, and when home sends a reply.

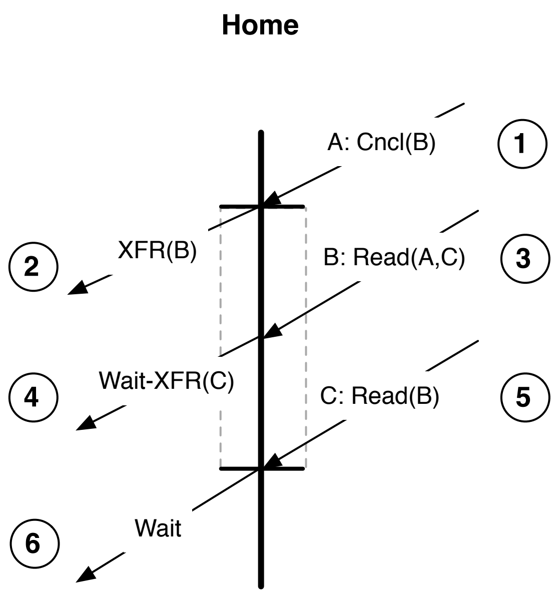


Figure 3.18: *Timeline for the home node showing the messages sent to home, including from which node and the conflict list, and the replies sent by home.*

The winning request from node A is forwarded the cache block from the node with the forwarding state (not shown). Because node A is only in conflict with node B, node A sends *Cncl(B)* to home. Home processes this message, constructing the *requestor-queue* and ordering node A to forward the cache block to node B.

Next, node B sends *Read(A,C)* to home, as it has not yet received the cache block and is in conflict with both node A and B. Home scans through the conflict list and only adds node C to the *requestor-queue* as node A is already present. Finally, home sends

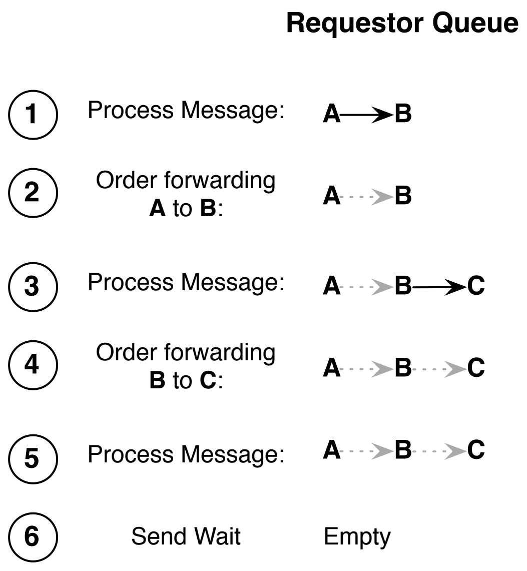


Figure 3.19: *The changes to the* requestor-queue *as it processes each message sent to home and sends replies; see Figure 3.18 for timeline of the home node. Each link in the* requestor-queue *is colored black if home has not yet scheduled the forwarding, or grey if it has been scheduled.*

WAIT-XFR(C) which orders node B to wait for to be forwarded the cache block (by some request) and then forward to node C.

Lastly, node C sends *Read(B)* to home, as it too has not yet received the cache block. There are no new nodes in the conflict list, so nothing to add to *requestor-queue*. Because node C is the tail of the *requestor-queue*, home must wait until all requests in the *requestor-queue* have sent a message to home before handling the message from node C. As this is the case, it is safe to send *WAIT* to node C, telling the node to wait to be forwarded the cache block then complete the request.

Home then exists the *conflict* phase and empties the *requestor-queue* until the next winning request arrives at home.

The role of home with simultaneous requests restricted by one request per node is straightforward. However when we allow any orderings of requests then home, and the requests themselves, must make sure that they can distinguish different requests running in the same node. Because requests and home keep track of requests in a per node basis home has a much greater role in ensuring the home assumption, that each conflicting request is forwarded the cache block precisely once.

In the next section we show how *MESIF*, and in particular home, handle any ordering of simultaneous and conflicting requests.

3.6.3 Any Simultaneous Requests

In the previous section we showed how *MESIF* ensures the four *MESIF* invariants for any ordering of simultaneous requests, provided that each node could engage in at most one request (for this cache block) at a time. To begin another request, the node had to wait until all simultaneous requests have completed before starting the new request. We treated nodes as synonyms for requests.

This is not true in general as a single request can be in conflict with more than one request *from the same node*. The conflict lists, the *requestor-queue* at home, and the forwarding responses sent by home, all assume requests and nodes are equivalent. In this section we will show how *MESIF* handles multiple conflicting requests from the same node, where requests and nodes are not equivalent.

We will show that a request can be in conflict with at most two requests from the same conflicting node. Thus, we augment the conflict-list sent to home in the *Read* or *Cncl* message to allow a request to indicate that it is in conflict with two requests from the same node. With this information, home can guarantee the home assumption stated in the previous section, reiterated below:

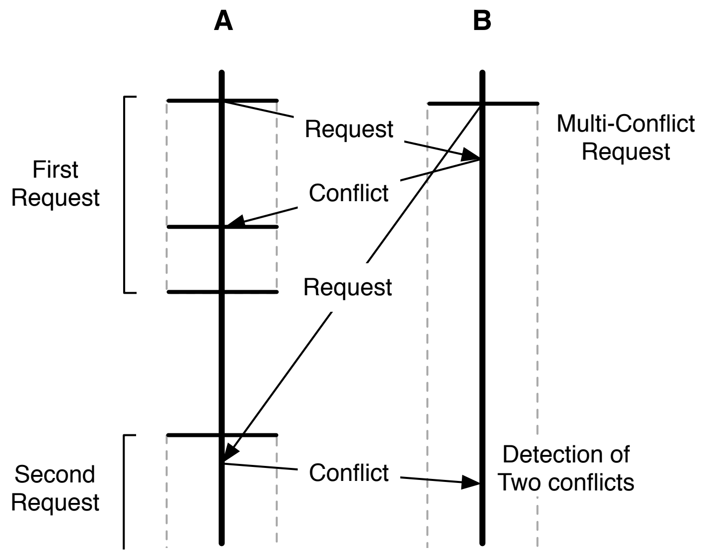
Home assumption: Schedules requests such that each of the conflicting requests will be scheduled to receive the forwarding state from precisely one request. Requests are added to the home ordering when home receives a *Cncl* or *Read* message, and are added at the end of the ordering.

We need to show that for all the orderings that a request can be in conflict with more than one request from the same node, these orderings have a maximum of two requests in simultaneous conflict. Then, we must describe how a request detects it is in conflict with two requests from the same node, as opposed to simple one request. Finally, we show how a request sends this information to home, and how home guarantees the home assumption.

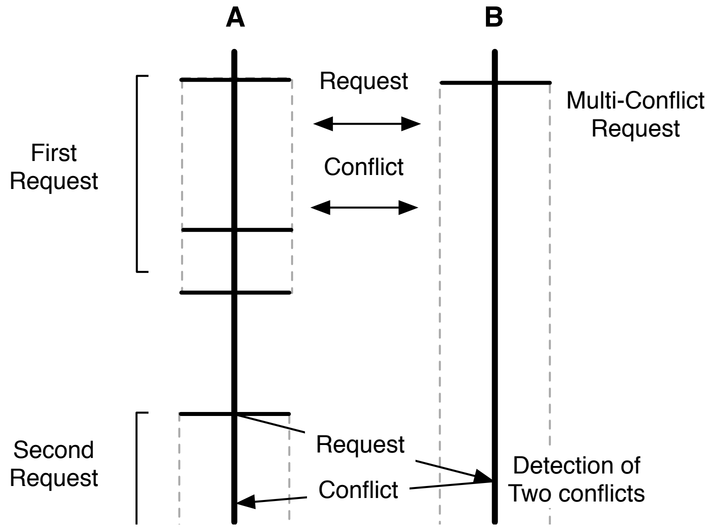
Multiple simultaneous requests require each cache block in every local cache (not the shared cache) to have a single bit that is flipped whenever a request completes. The value of this bit is then sent in both the request messages, *GetS* and *GetX*, as well as the conflict message. Without the flipping bit, it is impossible to distinguish between multiple conflicting requests from the same node, or a single request.

Maximum of Two Requests Conflicting Per Node:

There are only two different patterns of the three requests as shown in Figure 3.20. Either multi-conflict request detects the second request on its broadcast message (Figure 3.20a), or the second request is detected when the multi-conflict request receives, during its broadcast phase, the broadcast message by the second request (Figure 3.20b).



(a) *Multiple conflicts from the same node due to delayed request message arriving at the second request.*



(b) *Multiple requests due to two broadcast messages arriving at the same request. There are many different ways the first request and node B can become conflicting.*

Figure 3.20: *Shows the two different orderings of how multiple conflicting requests from the same node (node A) can be detected.*

The reason why a request can only be in conflict with a maximum of two requests from the same node has to do with how home orders requests. For there to be two requests at the same node to the same cache block, the first request must complete before the second request begins. As the first request is in conflict with the multi-conflicting request, the multi-conflicting request will be ordered in home's *requestor-queue* before the second request begins.

Thus, the second request must be ordered after the multi-conflicting requests as new requests are added onto the tail of the *requestor-queue*. Therefore before the second request gets forwarded, the multi-conflicting request must forward the cache block. By the time the second request could complete, the multi-conflicting request will have either completed or if not, it is in the home phase and will not respond with *Conflict* to the third request from the same node.

Detection of Multiple Conflicting Requests:

There are two different orderings with multiple conflicting requests from the same node shown in Figure 3.20. We need to ensure that the multi-conflicting request (request in node B in the figure) can detect that this is a multiple request situation or not before entering the home phase. If the multi-conflict request receives two *GetS* and *GetX* requests from the same node, then the multi-conflict request can detect the two requests. But we cannot rely on this.

For example, the multiple request ordering in Figure 3.20a, without changing the messages received, is indistinguishable from the *2a* and *2b* orderings in Figure 3.10b. In this case, if the multi-conflict request enters the home phase before receiving the second requests broadcast message, we would violate the **Pairwise Conflict-Aware** as the second request knows it is in conflict with the multi-conflict request, but not the other way around[19].

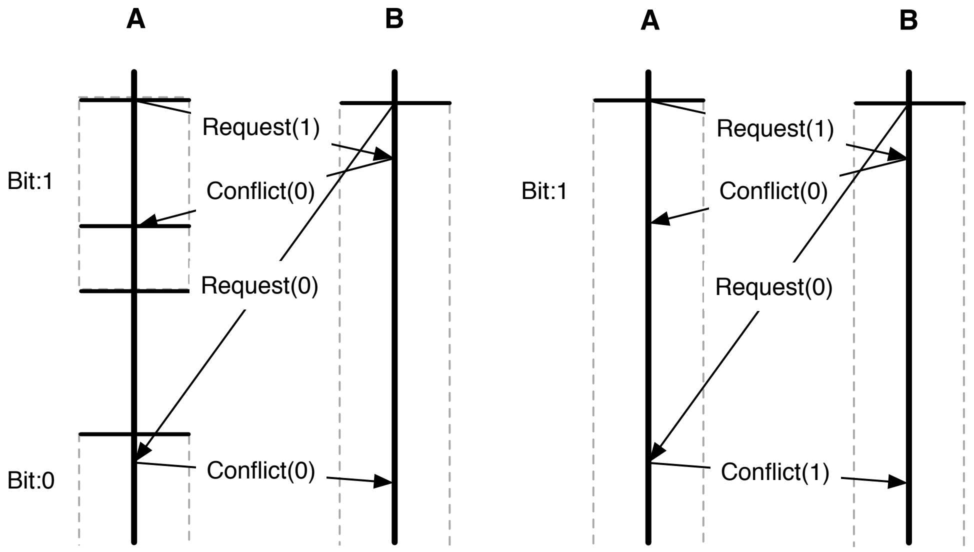
To solve this, we rely on the fact that we only have two requests from the same node and that these requests are immediately after each other. Each cache block in every local cache then has a single bit that is flipped whenever a request completes. The value of this bit is then sent in both the request messages, *GetS* and *GetX*, as well as the conflict message. Figure 3.21 shows how we can use this flipping bit to distinguish between multiple requests from the same node, and the *2a* and *2b* request orderings.

In Figure 3.21a, the multi-conflict request at node B receives first a request from node A with the bit set to 1. Then it sees a conflict response from node A with the bit now set to 0. Multiple conflicting requests from the same node are detected.

Conversely in Figure 3.21b, the single-conflict request at node B receives the same messages as before. Except the value of the bit sent in both messages are the same, 1. No multiple conflicting requests.

For the multi-conflict request to be in conflict with two requests from the same node, they

[19]This is not the same as a *virtual* conflict, where both requests do not know they are in conflict. In that case home is the one that can tell that they should be in conflict. Here, home does not have that information.



(a) *The multi-conflict request at node B sees different bits (first* 0 *then* 1*) from messages sent by node A.*

(b) *The single-conflict request at node B sees the same bits (just* 0*) from messages sent by node A.*

Figure 3.21: *How using a single bit per cache block that flips when requests complete, a request can detect multiple conflicting requests from the same node.*

must exchange messages between one another. From the two orderings in Figure 3.20, we can see that the multi-conflict request at node B detects both conflicts before the home phase, thus satisfying the **Pairwise Conflict-Aware** invariant.

To show that *MESIF* can handle in orderings or requests we now only need to show how home holds the home assumption.

Home Assumption Holds:

In the previous section where each node could execute at most one request until all requests had completed, the number of requests that are part of the same conflict phase in home are limited to the number of caching nodes in the system. This means the *requestor-queue* will never have more nodes than the number of caching nodes.

However, when we have no such limitations on requests it is possible for requests to keep on conflicting, with no limits on the number of forwards that can occur since the winning request. We must remove nodes from the *requestor-queue* when they are no longer necessary.

To show the home assumption still holds, we need to show that without using the *requestor-queue*, we only add conflicting requests once to the *requestor-queue*. As so, each conflicting request is forwarded the cache block precisely once.

Simply removing nodes from the *requestor-queue* would spell disaster, as nodes in the *requestor-queue* serve many purposes. First, we needed the nodes to ensure the forwarding order held true. We also needed the nodes for virtual conflict detection. As long as we keep a node until both 1) the node has sent a *Read* and *Cncl* to home and 2) the previous node, scheduled to forward to this node, has sent *Read* and *Cncl* to home. After both of these events, we no longer need the node for ordering reasons, as we have already scheduled the forwarding *to* the node and scheduled the forwarding *of* the node.

One final reason for the node being present in the *requestor-queue* is that we needed them to ensure we did not add a request twice.

When we start removing nodes from the *requestor-queue*, then when another request sends *Read* or *Cncl* to home with this removed node in its conflict list, if we simply add it we may end up scheduling the forwarding to the removed request more than once. This violates the home assumption that each conflicting request is forwarded the cache block precisely once. If the removed request's node is not in a request, it may find a cache block being inadvertently forwarded to it.

Home uses the **Pairwise Conflict-Aware** invariant, which states that each request knows which requests it is in conflict with by the time it reaches the home phase[20]. We have showed that this invariant holds true even in multiple conflicts with the same node.

As mentioned in the previous section, when a request (the requestors node is N) sends *Read* or *Cncl* to home, home has two decisions to make before it replies with one of the home response messages. Home must decide if it should add the N to the *requestor-queue*. And home must decide for each of the nodes in the conflict list, if those nodes should be added to the *requestor-queue*.

Deciding whether to add N is straightforward. If N is *not* in the *requestor-queue* then N has not been added for this request. Home adds N to the *requestor-queue*.

Home removes nodes from the *requestor-queue* when both the node has sent *Read* or *Cncl* to home and the previous node in the *requestor-queue* (the one scheduled to forward to this node) has also sent *Read* or *Cncl* to home. If N is in the *requestor-queue*, but N in the *requestor-queue* has been marked as 'scheduled', that N has sent *Read* or *Cncl*, then we also add N to the tail of the *requestor-queue*; the first N was added for the previous request that ran on N, not the current request.

Next, for every node N_c in the conflict list, should home add the node or not. Because home removes nodes, we cannot add N_c just because it is not in *requestor-queue*. If N_c is not in the *requestor-queue*, then either N_c was never added, or N_c has already sent *Cncl* or *Read* to home.

[20]Virtual requests are first upgraded by home to real requests by appending on the virtual conflicting requests into the conflict list before home checks the conflict list for new requests to add to the *requestor-queue*.

① A: Cncl(B)

	Conflicting		
From	A	B	C
A		✓	
B			
C			

③ B: Read(A,C)

	Conflicting		
From	A	B	C
A			
B			✓
C			

⑤ C: Read(B)

	Conflicting		
From	A	B	C
A			
B			
C			

Figure 3.22: *The progression of the* conflict-table *on the messages sent to home in Figure 3.18.*

Because of **Pairwise Conflict-Aware** invariant, when N_c sends *Cncl* or *Read* to home, N_c will have N in its conflict list. Home then records that it should expect N to have N_c on its conflict list when N sends *Read* or *Cncl* to home. Thus, home checks this record to see if N_c has marked N, and if so, we do not add N_c to the *requestor-queue*.

If there is no mark that N_c has sent to home with a conflict list including N, then we may need to add N_c. We need to be sure that no other node in conflict with N_c has added N_c. Because home records this, we just need to check that no node has marked N_c is in the conflict list of another node. If N was not in the conflict list of another node, and N has not sent *Read* or *Cncl* to home, we add N to the *requestor-queue*.

We only require an square table of length the number of caching nodes in the system to store these marks. In *MESIF* we call this the *conflict-table*.

Figure 3.22 shows the *conflict-table* during the three conflicting requests shown in Figure 3.18. While this example is small, and we could have simply looked in the *requestor-queue* to see if we should add nodes or not, we demonstrate how the *conflict-table* allows requests to see which conflicting nodes need to be added or not.

The first *Cncl* sent to home is from A, the winning request. Home adds node A as it is not in the *requestor-queue*, and also adds conflicting node B as the A entry (column) in the B row is empty. To mark that node A has sent to home with node B in its conflicting list, the *conflict-table* marks the B entry in the A row.

When node B next sends *Read* to home, it does not add its own node to the *requestor-queue*, it is already present. For node A and node C in the conflict list, home sees that the B entry in the A row is marked. This indicates that the conflicting node A has already sent a message to home; home should not add node A to the *requestor-queue*. Home wipes the mark in the B entry, A row; we have already dealt with both requests from node A and B.

On the other hand, node C can be added as the B entry in the C row is not marked (C has not yet sent to home) and there are no marks in the C entry in any row (C has not appeared in a conflict list yet). Node C is appended to the tail of the *requestor-queue*.

Finally home marks the C entry in the B row to make sure node C does not add node B again.

Lastly node C sends *Read* to home, C is already in the *requestor-queue*, so it is not added. And the only node in C's conflict list, node B, has marked that B has already sent to home. Home wipes the mark in entry C, row B, and exits the conflict phase.

The aim of removing nodes from the *requestor-queue* is because multiple requests from the same node could result in an ever expanding *requestor-queue*. We can keep track of which nodes to add using the **Pairwise Conflict-Aware** invariant, that each pair of conflicting requests both know they are in conflict with each other, to decide if a node is missing from the *requestor-queue* because it was never added or because it was added but removed.

The *conflict-table* holds this information. For each pair of conflicting nodes, the node first to home will mark the entry of the other node, and the other node will wipe the mark when it sends to home. We know a node has never been added because the node's entry is not marked in any of the *conflict-table* rows. Now to handle multiple conflicting requests from the same node.

Each entry in the conflict table now can contain two marks and each conflict list can also contain two copies of the same node. The operation of home is identical to before, except a node A can send a *Read* or *Cncl* to home twice and both times report that a node B is in conflict. Home then marks the B entry in the A row twice, one mark on the first message, and a second mark on the next. To remove these two marks requires that node B sends a conflict list with two copies of A; since the request at node B is in conflict with two requests from node A.

Likewise, when home receives a conflict list from a node, B, that contains two copies of the same node, A, home knows that node B is in conflict with two requests from the same node. Home treats each of these A nodes in the conflict list sequentially.

If node B is first to home ahead of any requests from A, home marks entry A in row B once, and then twice. If node B reached home after the first request from A sent to home but before the second request from A, then home clears entry B row A and then single marks entry A row B.

Otherwise, node B is last to home and removes the double marked entry B in row A.

In all cases, home simply treats the two copies of the same node sequentially.

In the next section we describe some issues a realistic implementation of *MESIF* would face.

3.7 Realistic System Issues

When implementing the *MESIF* protocol in a realistic system, some assumptions will not hold. We list a few of the limitations previously ignored, and discuss solutions:

1. Finite message buffer size.
2. Multiple requests flipping bit.

3.7.1 Finite Message Buffer Size

We have assumed in *MESIF* that any message required to be sent from one node to the other eventually succeeds, given some latency. However interconnects use finite sized message buffers to hold messages before sending them on the network. In *MESIF* which uses a point to point interconnect composed of individual links between two nodes, there is a potential for deadlock.

We can use virtual channels are used to prevent deadlock, messages on higher priority virtual channels are sent on the network before messages on lower priority virtual channels. The three channels are, in increasing order of priority:

1. Initial requests: *GetS*, *GetX*, and *Write-Back*.
2. Initial request responses: *IACK*, *SACK*, and *Conflict*; Requests to home: *Read* and *Cncl*.
3. *DACK*, *Data*[*F*,*O*,*M*,*E*]; Responses from home: *ACK*, *DataE-Home*, *XFR*, etc.

Messages that either terminate a request or pass the forwarding state along cannot be delayed, they are in the critical path of execution and are on the highest priority virtual channel. Next, initial requests cannot stall currently running requests, so the response messages must not be delayed by initial requests. Thus, initial requests are on the lowest priority virtual. channel.

3.7.2 Multiple Requests Flipping Bit

In Section 3.6.3 we showed that to handle any ordering of requests requires a bit per cache block in every local cache. This bit is flipped on the completion of a request to that cache block.

One way of implementing the flipping bit is to double the stable cache states for each cache block. Because *MESIF* has six stable cache states, *Modified*, *Owned*, *Exclusive*, *Shared*, *Invalid*, and *Forward*, we would now have twelve. The cache state would now require 4 bits to store, compared to 3 bits before.

3.8 Optimizations

This section introduces optimizations those ideas are part of the original *MESIF* protocol, but left out to simplify the discussion.

The different optimizations are listed below:

1. Shared cache in home node.
2. Migratory sharing data pattern: on read request, invalidate and pass line as exclusive.
3. Utilizing the *GetS* and *GetX* messages to home, to prefetch.
4. Home intelligently ordering requests when adding to the *requestor-queue* to minimize invalidations.

We will show how to modify *MESIF* to allow for each of these optimizations, and briefly discuss the benefits for each.

3.8.1 Shared Cache in Home Node

Up until this point, we have assumed that there is no shared cache in the home node. Thus, *Get-Shared* requests to home have always generated a memory request to main memory, and *Write-Back* always immediately wrote back the evicted value to main memory. In this section we show how to add a shared cache to the home node.

There are two ways of adding the shared cache to the home node, a *memory side* shared cache or a *cache side* shared cache.

It is not required to choose just one option, a system can have both a memory side shared cache as well as a cache side shared cache, depending on the cost required to have two levels of shared cache.

Memory Side Shared Cache

Conceptually the simplest cache to add to the home node is a cache on the memory side.

A memory side cache acts as a cache for main memory. Logically, a memory side shared cache adds no coherence issues[16], but simply speeds up reading and writing main memory. Requests asking for data from main memory still require 4 hops to complete. Two hops for broadcast to discover the block is not locally cached in the forwarding state. And another two hops to ask home and receive the block from home's cache.

The information that a cache block in a memory side shared cache needs to store is whether or not a cache block is dirty. There is no need to store if this cache block is the forwarding state, the forwarding state exists in main memory (and so, in the shared cache) if and only if the request sends *Read* to home, and this request is the first request (the winning request).

Not only would a shared cache speed up requests for cache block data from memory, but evictions could also be quicker. When a local cache evicts the forwarding state which is also dirty (not a silent eviction), the cache begins by sending *Write-Back* to the home node and enters the *waiting* phase as shown in Figure 3.9. With a cache in the home node, the evicted data is written to the cache itself which is faster than sending it to main memory.

We need to store if this cache block is dirty or not because a shared cache is still a cache, so it will have cache evictions. On cache block eviction, the cache block data is written to main memory. At the time of the eviction, if the forwarding state is not locally cached, then the data in the shared cache is the current value for the cache block and must be written to main memory.

On the other hand, it is possible that the 'dirty' copy of the cache block is in fact, a stale copy. For example, an eviction could have written the cache block to the shared cache, but then a subsequent writer requested the cache block, writing a new data value. This would render the write back of the evicted 'dirty' cache block pointless, as the actual value resides in a local cache. We can prevent this write back, and simply silently evict the 'dirty' but stale cache block if we recorded if the shared cache held the forwarding state, which would save memory bandwidth but it is not necessary.

Cache Side Shared Cache

The second way is to add the shared cache as a cache side shared cache. Unlike a memory side shared cache which is invisible to the other local caches, a cache side shared cache appears to the other caches as another cache, though without an associated core that is issuing memory requests.

Each *GetS* and *GetX* broadcast message then needs to send the message to the home node, and wait until it receives an *IACK*[21] or forwarding from home. If the shared cache

[21] Because the shared cache does not initiate any memory operations, forwarding will always invalidate

did forward on receiving a request for the cache block, the shared cache would need to delay other request broadcasts until it received a $DACK$.

Eviction from one of the local caches works in the same way as with a memory side shared cache.

The advantages is that memory requests are serviced with a two hop latency, compared to the 4 hop latency of the memory side cache provided the shared cache has the cache block in the forward state. The disadvantage is the complexity of having both a cache and home in the same physical location[22]. However the cache side shared cache is less complex than one of the local caches as the shared cache does not initiate memory operations, only evictions and cache block forwarding.

3.8.2 Migratory Sharing Pattern

Accesses to data objects (in this case, cache blocks) from different types of sharing patterns can be useful in predicting what future accesses nodes will initiate. A common sharing pattern is the migratory sharing pattern[18]. In migratory sharing, cache blocks only read and written by a single core at a time. This sharing pattern occurs often with shared data protected by locks[18]; the shared data can only be accesses by which core has the lock. Once the lock is released the shared data migrates to the next core with the lock.

If a cache block is in the migratory sharing pattern but a core first requests this cache block to read, when this core wishes to write to the cache block it must generate a second request for the block. It would be better to, given the block exhibits migratory sharing, give the requesting core writing permissions on the read request.

This could half the number of requests if the cache block was both migratory, and always first read. How much benefit optimizing for migratory sharing depends on how often migratory sharing occurs and how well the heuristics used in predicting migratory sharing work. Miss-predicting a cache block as migratory when it is not results in unnecessary invalidation.

There are a number of heuristics to predict if a cache block is migratory. If a core has the cache block in a writable state and writes to the cache block, then the core responds to a read request by invalidation (downgrading to *Invalid*) and forwarding the cache block as writable[18, 12]. In the case that this cache block is not migratory and many cores wish to read it, subsequent read requests after the first result in the usual forwarding and downgrade to *Shared*.

the cache block. Cache blocks are never in the *Shared* state in the shared cache.

[22]In our *MESIF* implementation, we chose the shared cache to be located cache side.

Implementing migratory sharing in *MESIF* is straightforward. In Section 3.2.1 we showed if a cache has the cache block in the *Modified* or *Exclusive* cache state, a read request results in downgrading to *Shared* while forwarding to the requestor.

Instead for migratory cache blocks, the cache with the cache block as *Modified* would forward *DataM* to the read request instead of *DataF*. At the requestors side, if it receives *DataM*, the requestor knows that the forwarding node was *Modified* but is now *Invalid*. This means it is safe to upgrade to *Modified* rather than *Owned*.

3.8.3 Prefetching Data

When a cache block is not locally cached in the forwarding state, a memory request for that block sends *Read* to home, which returns the data in either a *DataE-XFR* or *DataE-Home* depending on whether or not there are conflicting requests respectively.

This *Read* to home may or may not take a long time, depending on the structure of the interconnect. If the links between the local caches and home are the same as the links between home and main memory, then we have at least a two hop delay between home receiving the *Read* and home receiving the cache block data. This means that a request for an cache block that is not locally cached in the forwarding state could take six hops. Two hops for the message broadcast, a hop for the *Read*, two hops for reading main memory, and a hop to return the data. Figure 3.23 shows this case.

Instead, an optimization is to prefetch the data before receiving the *Read* message. We can use the broadcast message as the hint to start prefetching the cache block. This *GetX* or *GetS* sent to home as the hint does not require a response by home; unless home contains a cache side shared cache (see Section 3.8.1 for details).

By utilizing the hint to prefetch, the read to main memory is overlapped with the broadcast responses and the *Read* or *Cncl* sent to home. By the time the request to home arrives, the cache block has arrived and can be immediately used in a *DataE-XFR* or *DataE-Home* response. Figure 3.24 shows this case.

Regardless of how long the read to main memory takes, prefetching using hints sent to home saves two hops. The disadvantage of prefetching are unnecessary bandwidth use if the forward state is locally cached.

3.8.4 Intelligent Ordering at Home

In Section 3.6.2 and Section 3.6.3 we discussed how home ensures that conflicting requests are forwarded the cache block precisely once and that requests are added at the

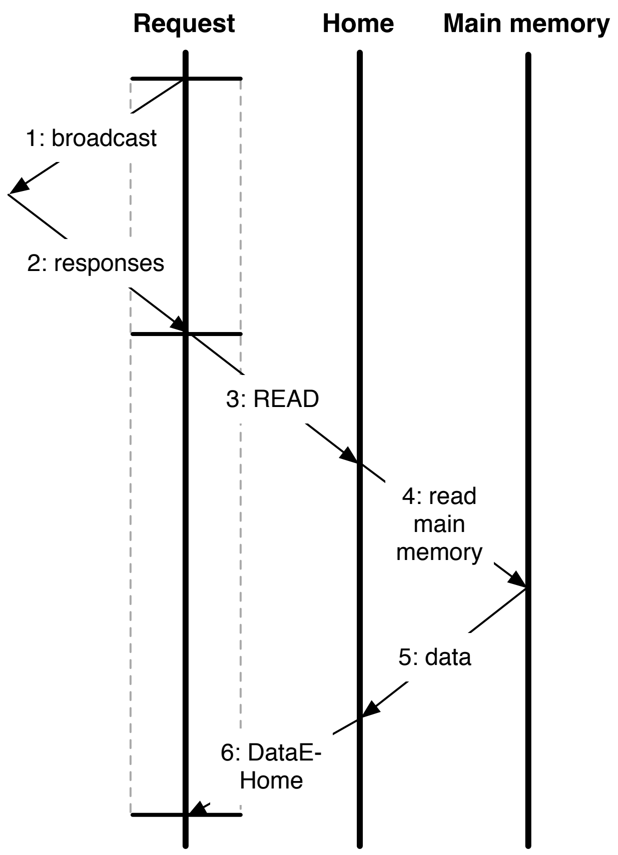


Figure 3.23: *No prefetching of cache block data. Six hops to retire and complete the request.*

end of the forwarding order represented by the *requestor-queue* (the home assumption).

When a request send a *Read* or *Cncl* to home, home checks the message's conflict list, the *requestor-queue*, and the *conflict-table* to decide which requests to add to the end of the *requestor-queue*. This set of requests, R_{add}, is a subset of the requests in the conflict list. Home has complete freedom over the ordering the requests in R_{add}.

One optimization for ordering R_{add} is to minimize the amount of sharer invalidations, resulting more nodes in the *Shared* state for longer. The longer a node has the cache block in shared, the more chance that a read memory *operation* will occur and hit in the cache rather than require a read *request*.

If R_{add} contains any write requests then home can order the write requests in R_{add} before the read requests. We discuss how ordering writes before reads can reduce the number of sharer invalidations.

Consider a read request Req_R and a write request Req_W in R_{add}. These two requests are in conflict with the request which home just received *Read* or *Cncl*. We can make no guarantees at this point about the relationship between Req_R and R_w but home must now decide the order of the forwarding: should Req_R receive the cache block before or

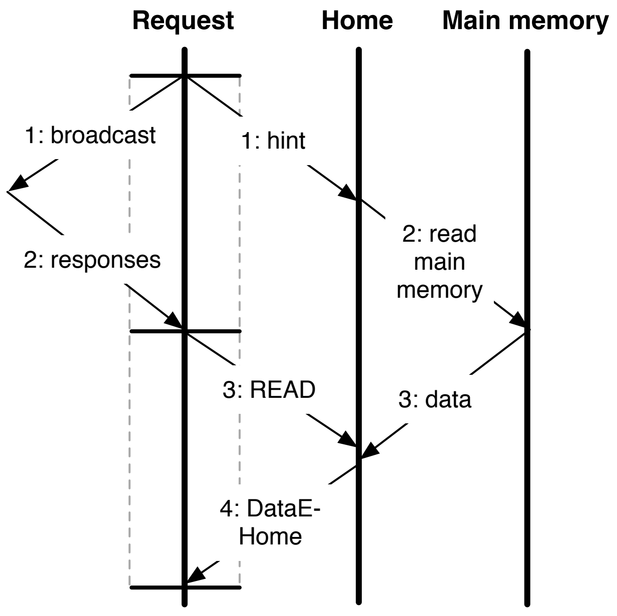


Figure 3.24: *Broadcast message includes prefetching 'hint'. Four hops to retire and complete the request.*

after Req_W.

The requests Req_R and Req_W can either be conflicting or non-conflicting. If non-conflicting then one request is ordered before the other by virtue of sending the broadcast message before or after the other request has completed (or additionally, if the broadcast message is delayed during the other's home phase). Figure 3.25 shows these three possible relationships from the perspective of the read request Req_R.

Recall that this ordering in Figure 3.25 is decided by the requests themselves, home only steps in after one request has sent to home *Read* and *Cncl*.

If the requests are conflicting then Req_R downgrades on forwarding[23]. Otherwise they are non-conflicting in Figure 3.25.

If they are non-conflicting, yet home orders them in the opposite order, then they become in conflict via *virtual* conflicts organized by home and the *Conflict-Update* message. That leaves us with two possibilities, home orders Req_W before Req_R and the requests are in that non-conflicting order. And visa versa with Req_R before Req_W.

If Req_R is before Req_W, then when Req_R commits, the write request Req_W will arrive and invalidate Req_R's node.

On the other hand if Req_W is before Req_R, then when Req_W commits Req_R's node

[23]Note that if Req_R happens to be the tail of the *requestor-queue*, if it stays as the last request, then Req_R does not forward and keeps the forwarding state.

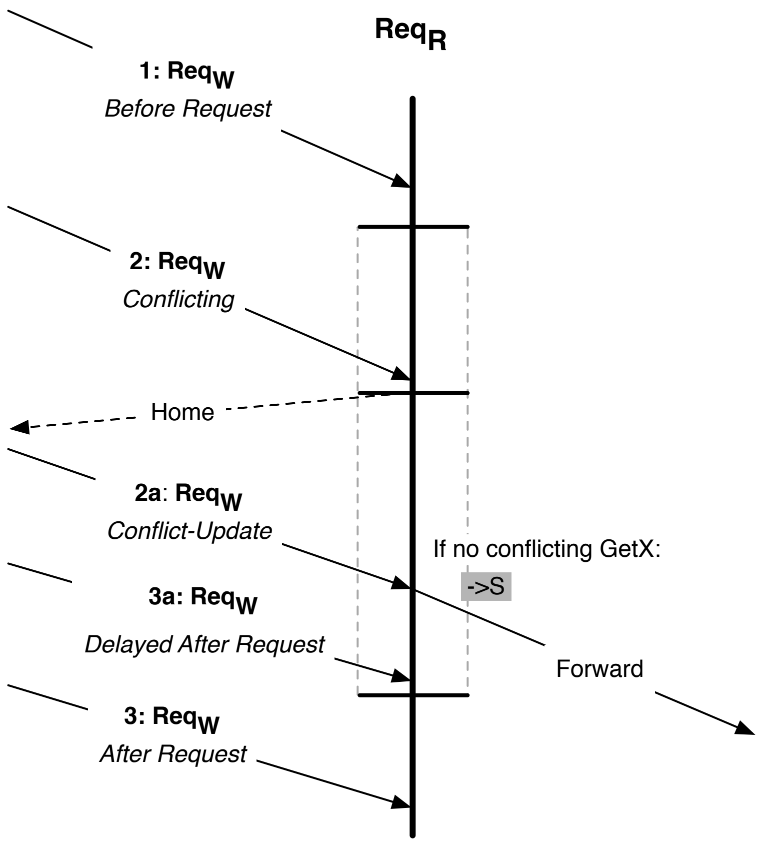


Figure 3.25: *The three possible relationships between the read Req_R and the write Req_W. Either conflicting, non-conflicting with Req_W before Req_R, or non-conflicting with Req_R before Req_W.* Conflict-Update *updates the conflict list for Req_R, so if Req_W is added, it is treated like a conflicting request. Req_W arriving during home phase is delayed, and treated as non-conflicting with Req_W behind Req_R.*

has been invalidated and will stay that way until Req_R is forwarded the cache block. When Req_R is forwarded the cache block, as long as there are no other conflicting write requests, and no write requests after Req_R, then the read request stays as *Shared.*

The best home can do is order all write requests in the set of requests to be added, R_{add}, before the read requests. This will ensure the highest likelihood that the nodes with the read requests will keep the cache block as *Shared* rather than being invalidated to *Invalid.*

In Section 4.3 we show an improvement to *MESIF* such that we can extend this further, where even conflicting write requests with a read requests will not invalidate the *Shared* cache block in the read request's cache.

Chapter 4

MESIF improvements

The previous chapter introduced the *MESIF* cache coherence protocol. While we expanded the high level details that were given in the *MESIF* technical report[6, 7], the original ideas are not our own.

In this chapter, we present four extensions to the *MESIF* cache coherence protocol that we developed.

4.1 Non-conflicting Simultaneous Read Requests

The first improvement is inspired by the *Token Coherence* protocol. *Token Coherence* allows for multiple simultaneous read requests that are non-conflicting. We discuss how *Token Coherence* achieves this. To apply this to *MESIF* requires some modifications, as the four *MESIF* invariants assume that cache block permissions and data are received from the forwarding state, which also transfers this forwarding state.

We introduce the *Reading Phase*, an optional phase the node with the forwarding state can be in. While in the *Reading Phase*, read requests are are sent the cache block data but only *Shared* cache permissions. The forwarding state does not leave the node. Multiple simultaneous read requests are non-conflicting, and can be serviced by the node with the forwarding state with two hop latency. The *Reading Phase* ends when the forwarding state receives a *GetX* message from an active write request.

4.1.1 *Token Coherence* Comparison

When we introduced the *Token Coherence* protocol in Section 2.2.6 we briefly discussed how *Token Coherence* uses an optional *Owned* state to pass a single token and data to a read request. The *Owned* state can be held by at most one cache, and prevents more than one cache forwarding cache block data to the same request.

If we have two read requests reading the value written by the same write, then for coherence it does not matter which read request commits first; either order of read requests are indistinguishable from one another, the reads are simply reordered to appear before the writer-reader period they are reading from[1]. In *MESIF* we defined conflicting requests as requests that exchanged *GetS* or *GetX* message during broadcast phase.

This means that even though simultaneous read requests are logically non-conflicting, as the order of committing does not matter, *MESIF* treats them as conflicting and explicitly orders the reads requests one after the other. This can result in much longer latencies compared to *Token Coherence*.

With *Token Coherence* read requests can be as quick as two hops if the *Owned* state exists, and multiple read request do not affect this. At most, as more and more read requests arrive at the *Owned* state, the *Owned* node can immediately respond with data and one token. Figure 4.1 shows three simultaneous and logically non-conflicting read requests beginning at similar times in *Token Coherence*. Each request takes two hops to execute, minus a few processing delays.

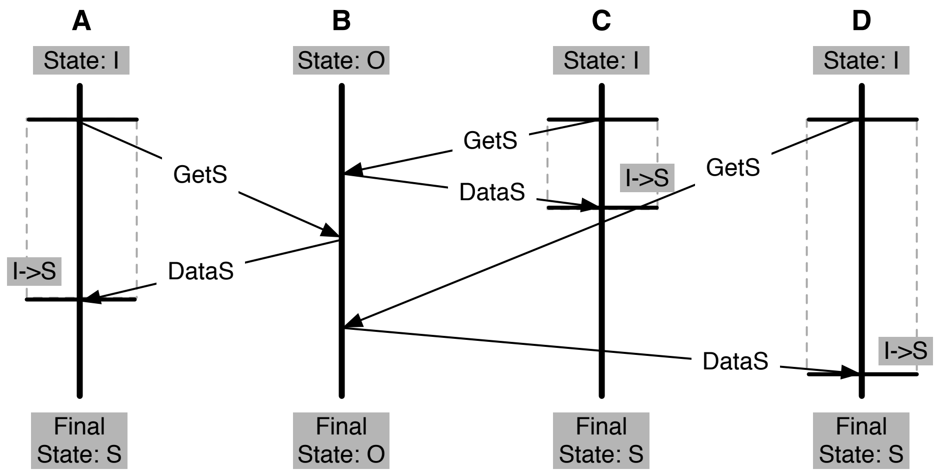


Figure 4.1: *Three simultaneous read requests are not conflicting with the* Owned *state present. Each request takes two hops to complete minus processing delay. Only relevant messages shown. Placement of arrows is not representative of the delay of each message, but for cosmetic purposes: easier to read the diagram.*

In contrast, the same situation of three simultaneous read requests in *MESIF* are now conflicting requests is shown in Figure 4.2. These requests are still logically non-conflicting however the only way to receive permissions and cache block data is via forwarding, and with conflicting requests, requests must send messages to home before they can forward the cache block on.

[1]See Section 2.1 for details on writer-reader and readers periods.

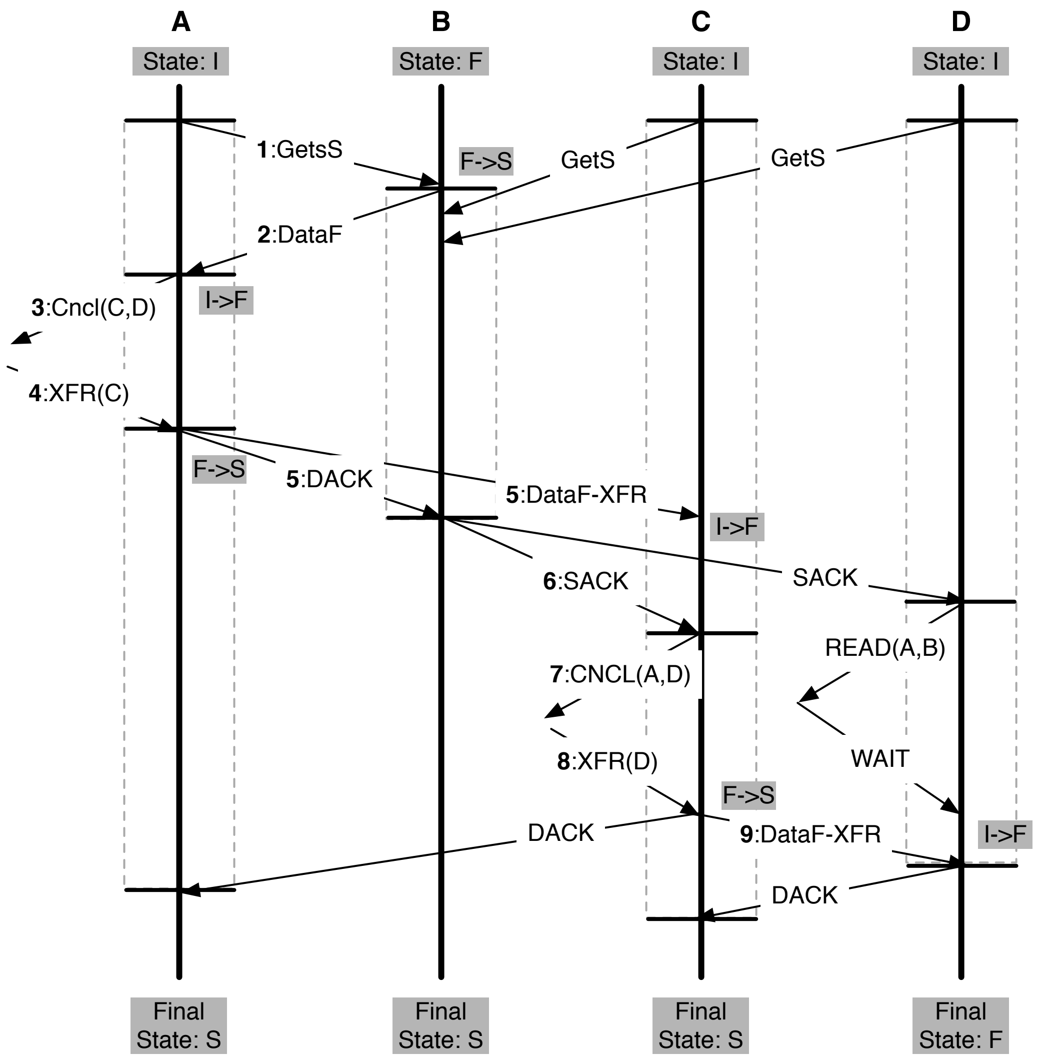


Figure 4.2: *Three simultaneous and conflicting (the* MESIF *definition) read requests. Only the winning request at node A has two hop latency, with the node B requiring (best case) 5 hops and node C requiring 9 hops. The critical path to receiving the forwarding for each node are labeled by hop number as a suffix to the message.*

Only one request, the winning request, actually receives the cache block in two hops. The second request to be forwarded, node *C* requires five hops and the final request requires 9 hops. In the best case, adding another request that begins at the same time as the other requests requires one more additional hop to receive the forwarding state. So the fourth request would need 10 hops, the fifth 11 hops, and so on. The requests require even more hops to complete.

Not only do simultaneous read requests take longer in *MESIF* than *Token Coherence*, they also do not scale well[2].

[2]The conflict handling logic of *MESIF* works the same way whether or not requests are read or write. For the requests in Figure 4.2, and or all could be write requests, and the delay would be the same. Each node would have to downgrade to *Invalid* rather than *Shared*, but that is the only difference.

Inspired by *Token Coherence*, we developed an addition to the original *MESIF* protocol that allows for non-conflicting reads, the *Reading Phase*. In the *Reading Phase* read requests are instead forwarded the cache block data and *Shared* permissions by the forwarding node, instead of the forwarding state being transferred. The complex issue is transitioning from *Reading Phase* state to normal *MESIF* where all request are conflicting. This occurs when a write request's *GetX* message reaches the node with the forwarding state.

4.1.2 *MESIF Reading Phase*

We first describe the actions of the cache in the forwarding state in *Reading Phase*. Next we show that when the forwarding state is not locally cached, request act as they do in original *MESIF*. Finally we show the transition out of *Reading Phase* when a write request begins.

Forwarding State Locally Cached

When the forwarding state is locally cached in node N_F, the first *GetS* message to arrive at N_F triggers the beginning of *Reading Phase*[3]. Home responds to *GetS* with *DataS*, which gives the cache block data and *Shared* cache permissions to the requestor.

Figure 4.3 shows the messages between N_F and a read request. Once the read requestor has receives *DataS* which counts as a reply to the *GetS* sent to N_F. The read requestor waits for all replies back and if none where *Conflict* from write requests, sends *DACK* to N_F.

The forwarding cache N_F continues to respond to read requests with *DataS* and only exists *GetS* once it has received a *DACK* from all outstanding read requests; read requests N_F send *DataS* to.

This works well until a write request begins.

Forwarding State Not Locally Cached

When there the forwarding state is not locally cached, there is no possibility of receiving *DataS*. Read requests then acts as original *MESIF*. Once a request has received all replies, and no node forwarded the forwarding state or *DataS*, then the request sends *Read* to home including the conflict list.

[3]It is not required for N_F to enter *Reading Phase*, passing the cache block as *Shared*. Instead N_F could act as it would in normal *MESIF*, and forward the forwarding state. One case where forwarding the forwarding state may be better is in migratory sharing as the read requestor is probably about to write the cache block. See Section 3.8.2 for more details.

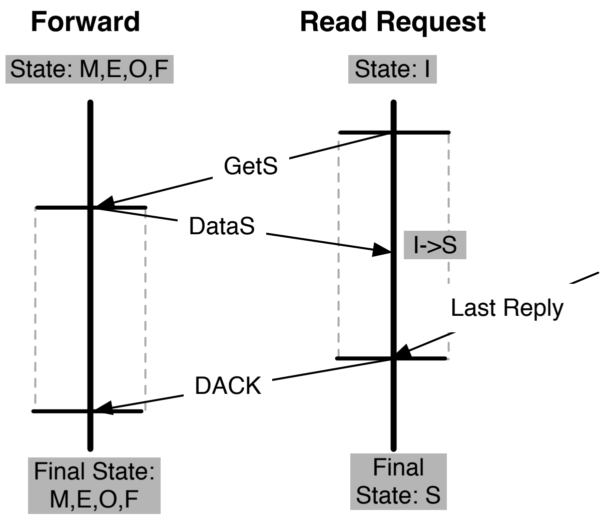


Figure 4.3: *Messages sent between the forwarding state caching node N_F and a read request. N_F enters the* Reading Phase *on receiving the read requests, and exits on receiving the* DACK *response.*

Figure 4.4 shows this situation, with two simultaneous read requests. These read requests are actually conflicting, as there neither request received *DataS*. Home then sends *DataE-XFR(A)* to the winning request, the request at node B, and the timeline is the same as a pair of conflicting requests as discussed in Section 3.6.1.

Transition from *Reading Phase* on Write Request

When a write request begins, it broadcasts *GetX* to all caching nodes. Eventually the *GetX* reaches[4] the forwarding node N_F which is in *Reading Phase*. This triggers N_F to begin moving into the original *MESIF* node; it must clean up.

What we want the forwarding node N_F is to decide which read requests will commit (via receiving *DataS*) before any write request gets the forward state and which read requests will *not* receive *DataS* but must get it forwarded from another request.

The problem here is while N_F can easily allow reads to commit via *DataS* then *DACK* response, the difficulty is updating the write requests and the non-committed read requests (the ones that will not receive *DataS*) conflict lists to *not* include any of those committed read requests.

We need to do this to preserve the **Pairwise Conflict-Aware** invariant. The committed reads have not sent any messages to home. If any of the write requests or the non-committed read requests has one of the committed reads in its conflict list, this will break

[4]If N_F wishes to write the cache block, then N_F must also 'delay' the write just as it does for a *GetX* from another node.

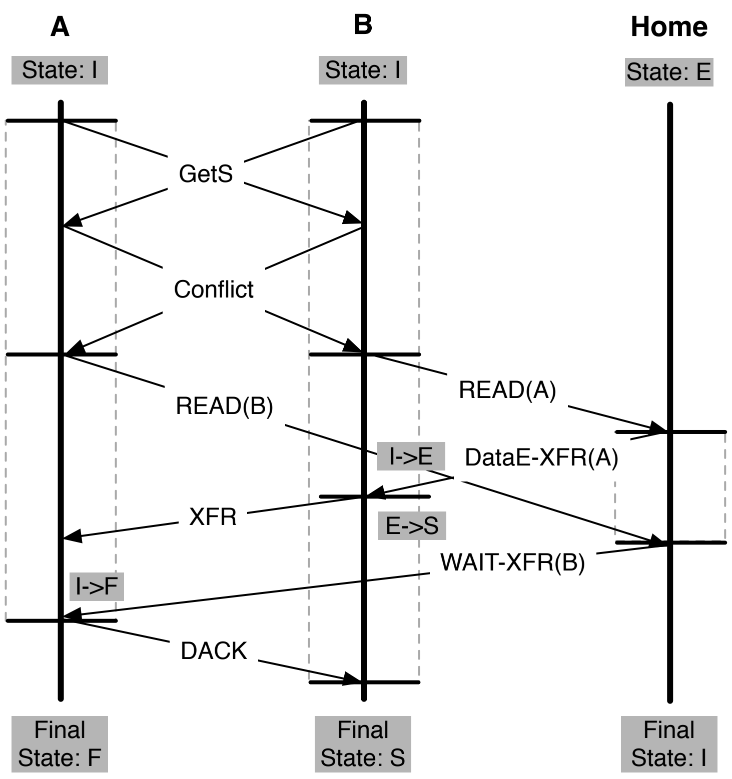


Figure 4.4: *With no forwarding state present as neither requests received* DataS *or was forwarded the forwarding states, the read requests act in the same manner as pairs of conflicting requests.*

Pairwise Conflict-Aware invariant, which is relied by home to ensure all requests receive the forwarding state precisely once.

Recall that a read request Req_R broadcasts *GetS* and receives responses, including *Conflict*. Additionally, *GetS* requests arrive during Req_R's broadcast phase, requiring a *Conflict* response. However, when there are no writing nodes, and there exists a forwarding node to reply with *DataS* the conflict list is not used: there logically is no conflict with just read requests. Read requests must also respond to *GetX* with *Conflict*.

The read requests which N_F will decide to allow to commit, after N_F receives the first *GetX*, are the read requests N_F cannot prevent from committing: requests that have already committed and have sent a *DACK* to N_F or those that will commit as N_F has sent *DataS*. All other requests, both reads and writes, and buffered at N_F until it receives confirmation from the soon to be committed read requests, that they have actually committed. The confirmation is receiving the *DACK*.

Once N_F has received *DACK*s from all outstanding read requests that were sent *DataS*, N_F must now wait for a *GetS* or *GetX* from every request that has one of the committed reads in its conflict list.

Let us assume for now that N_F both knows which requests have a committed read in its conflict list. And that for each request, N_F knows which of the committed reads that request has in its conflict list.

To ensure the **Pairwise Conflict-Aware** invariant holds, we need to remove these committed reads from the request's conflict list.

Note that while N_F is delaying request messages, no request will send *Read* or *Cncl* to home. Every request will still be in the broadcast phase until N_F replies. Once N_F has received all *DACK*s from outstanding read requests, N_F then replies to the first write request as it would if it was a normal forwarding node. Except, in addition to forwarding the cache block and forwarding state, N_F also sends *Conflict-Update*, which when received results in the committed read requests being removed from the write request's conflict list.

After forwarding to the first write request, N_F must continue delaying all other read and write request messages; it is in the equivalent *waiting* phase as in normal *MESIF*. Once the write request (now the winning request) has sent *Cncl* to home and received a response, the write request sends *DACK* to N_F. Now N_F is free to respond to the rest of the delayed messages. It does so by sending *IACK* plus *Conflict-Update*, with the *Conflict-Update* message containing the committed read request that this delayed request is in conflict with.

After replying to all request messages, H_N can exit the *Reading Phase*.

Each request will now only have other conflicting requests in their conflict lists. No committed reads will be in it. Thus the **Pairwise Conflict-Aware** invariant will hold, and the conflicting requests are handled in the original *MESIF* manner.

N_F **Knowledge Assumption:**

We assumed that N_F knows not only which requests are in conflict with committed read requests, but also which committed read requests this request is in conflict with. By knowing this, N_F knows how long to delay responding to the first write request, and how to prevent the non-committed requests from inadvertently telling home that it is in conflict with one of the committed read requests.

To discover this information, N_F uses the **Pairwise Conflict-Aware** invariant and requires that each read request send their conflict list with the final *DACK* message. N_F can use the conflict list in much the same way home does.

The situation is analogues to conflicting requests sending to home *Read* or *Cncl* and including a conflict list. The *DACK* to N_F is equivalent to the *Read* and *Cncl* message. Except in this case, there is no need to order any request to forward the forwarding state on, N_F will hold onto the forwarding state.

Another difference is that with multiple conflicting requests from the same node (see Section 3.6.3) we were limited to two conflicting requests from the same node. Figure 4.5 shows how a request from node *B* can be in conflict with any number of requests from node *A*, there is no mechanism to prevent requests from continuing to commit and begin again; with conflicting requests sent to home (Section 3.6.3) the mechanism was the ordering of the requests by home.

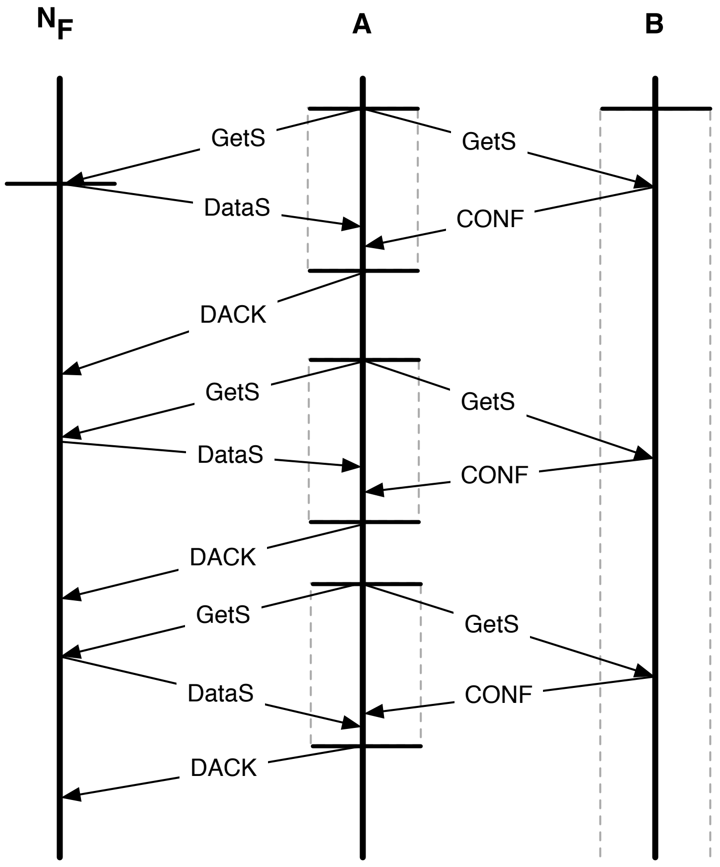


Figure 4.5: *Example of a request at node B being in conflict with multiple requests from the same node, A. There is no limit to the number of conflicts from the same node.*

This can be problematic, but an easy solution is to limit the number of requests that a single request can be in conflict with. In Figure 4.5, node *B*'s request can delay the third *GetS* from node *A* until node *B* has committed, preventing a third conflict with the same node.

Then we can use the same logic used with *conflict-table* and home which was to decide which nodes had not yet been added to the *requestor-queue*. Instead we are deciding which requests are in conflict with committed read requests and also which committed read requests this request is in conflict with.

When a read request R_R commits and sends $DACK$, with a list of conflicting requests, to N_F, N_F now checks to see that for all of the nodes in the conflict list of R_R, has that node's request reported R_R (check out the R_R entry in that requests row for a mark in the *conflict-table*). If so, then remove the mark in the R_R entry and the conflicting requests row: that request has already sent $DACK$ to home with R_R in its conflict list.

Otherwise the R_R entry and the conflicting requests row is empty, mark the conflicting requests entry in the R_R row.

When transitioning out of *Reading Phase*, we do not process any read requests (N_F also does not process any write requests), until all read requests that will eventually commit, the read requests N_F has already sent *DataS*, do commit. We record these outstanding committing read requests by marking the requests own entry in its own row in the *conflict-table* on a *GetS* request and *DataS* response. We wipe the mark once the node sends $DACK$ to N_F[5].

Once N_F has been sent $DACK$ from all read requests that it sent *DataS* from, N_F can now respond to the delayed read and write requests. By looking at the conflict table, we can tell which of the delayed requests are in conflict with a committed read. The delayed requests column (or entry) will have a mark (or two, as the delayed request could be in conflict with at most two read requests) indicating that this delayed request (column) was in conflict with this committed read (row).

We iterate over all rows for each delayed request, and append the rows (committed read requests) to an *Conflict-Update* which accompanies the normal response by the forwarding node. Thus, after all committed read requests have arrive, the first write request is forwarded the data like usual *MESIF*, but with *Conflict-Update* appended.

The now forwarding node N_F delays all requests until it receives a $DACK$ from the first write request. Then it processes each of the delayed requests, appending *Conflict-Update* to the $IACK$ and $SACK$ replies as necessary.

4.1.3 Sharing Patterns for *Reading Phase*

In Section 3.8.2 we discussed the migratory sharing pattern and an optimization to *MESIF* for migratory sharing. The optimization allowed for a node in the *Modified* state to invalidate itself in a read request, passing the *Modified* state to the requestor. We used a heuristic to detect if a cache block was migratory or not to decide if a request should pass as *Modified* for migratory, or pass as *Owned*.

Similarly for the *Reading Phase*. It is not necessary for the forwarding state to *always*

[5]After sending the $DACK$, the requesting node commits and is free to being another read or write request. This request message can race ahead of the $DACK$ and arrive before the $DACK$ at N_F. What N_F does is delay that request until it has received and processed the $DACK$.

respond to a read request by passing *Shared*. We could only do so in the case that writes are rare. One sharing pattern is the mostly read sharing pattern[18] where writes are infrequent but many cores read the cache block.

4.2 Write Updating *MESIF*

We briefly mentioned in Section 2.2 about *write-update* cache coherence protocols. In write-update protocols, a write request does not invalidate *Shared* copies of a cache block[6]. Instead, the updated cache block value is sent to all sharers, which can then read the newly written value immediately.

The advantages of write-updating is that there is no invalidation of sharers on external requests. If a cache block is rarely written but often read[18], write-update can quickly update the values for each of the sharers allowing the sharers to continue reading the cache block. A write-invalidate protocol would have to invalidate all sharers first before committing the write. If the cache block is often read this could result in many read requests, read requests that could have been avoidable with write-updating.

Write-updating in *MESIF* is only allowed for the node that has the cache block in one of the forwarding states: *Forward* or *Owned*[7]. However this fits well with the *producer-consumer* sharing pattern. This pattern occurs where only one core writes to a cache block but many cores read from it[3]. Thus the forwarding state node will be the writer, or *producer*, and the other nodes are the *consumers*.

Whether or not to use write-update does depend on the typical sharing patterns of the programs a particular implementation of *MESIF* would encounter. Write-update can require large amounts of bandwidth as cache data is sent to nodes that are no longer reading the cache block[3, 2, 8], or never were or will be reading the block. Here we only show how write-updating can be optionally done on a write request, so that it can be used if desired.

There are two different options for write-updating in *MESIF* involving a trade off between latency and bandwidth.

The first option is shown in Figure 4.6. The forwarding node, F_N, broadcasts a new message, *Data-Write-Update*, to all caching nodes. When a node receives *Data-Write-Update*, if it has the cache block in *Shared* state it updates the data value and sends *SACK* reply back to F_N. Otherwise the node is *Invalid*, and has the option of ignoring the request and replying with *IACK*. Or the invalid node can upgrade to *Shared* and send an *SACK*. The *Data-Write-Update* does not trigger a conflict if the receiving node

[6]A write-invalidate protocol does this. *MESIF* and *Token Coherence* are write-invalidate protocols.

[7]If it has it in *Modified* or *Exclusive*, then the node has permission to do the write operation without initiating a write request. And there are no sharers to update.

has an active request.

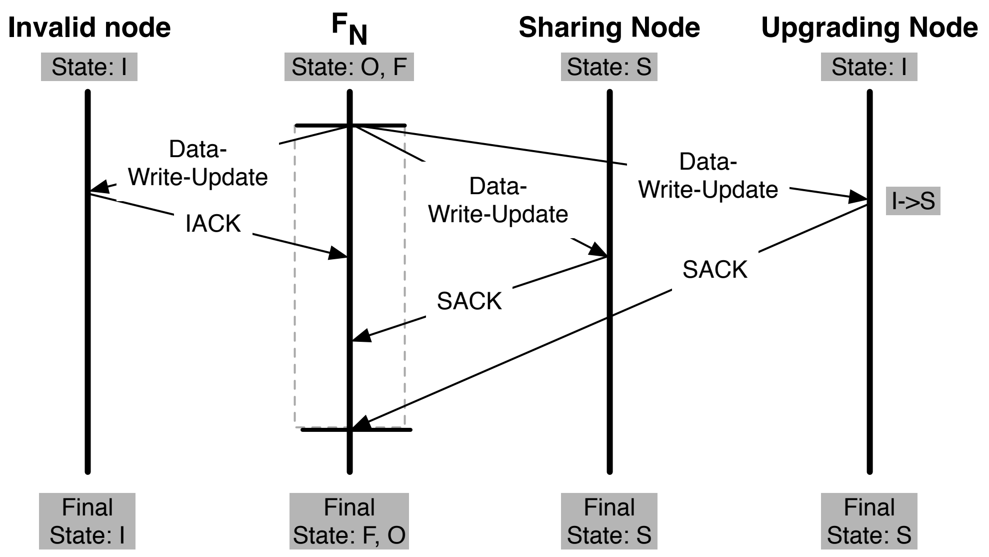


Figure 4.6: *Demonstrates the latency optimized write-update option. This option requires only two hops to complete. Shows the three different possible replies and state changes.*

Once all replies have come back, F_N commits and completes its request. Only when all replies have arrived can F_N be sure that all readers will be reading the new value for the cache block. If *Data-Write-Update* arrives at a node with a request, the request must respond by updating the cache block value (if *Shared*) and replying with *IACK* or *SACK*. All broadcast messages sent by simultaneous requests to F_N during the write update are delayed until completion; F_N is ordering itself before all other requests.

This option is fast, requiring only two hops to complete the write-back. But F_N must send cache block data to all caching nodes, as F_N does not know which nodes have the block as *Shared.* Because R_N cannot miss any sharers, the only option is broadcast.

The second option for write-update is shown in Figure 4.7. This option is slower, requiring four hops to complete. The first two hops are a broadcast of the new message, *Write-Update*, to discover which nodes have this cache block as *Shared.* A node that receives *Write-Update* replies with *IACK* or *SACK* if it has the cache block as *Invalid* or *Shared* respectively; R_N is not added to the conflict list if this node has a request active.

After receiving all responses to *Write-Update*, the second two hops are the *Data-Write-Update* message and response, updating the each sharer's cache block value.

Because F_N holds the forwarding state, no other request can receive cache block data and permissions except via F_N. To ensure *Data-Value* invariant, we only need to update

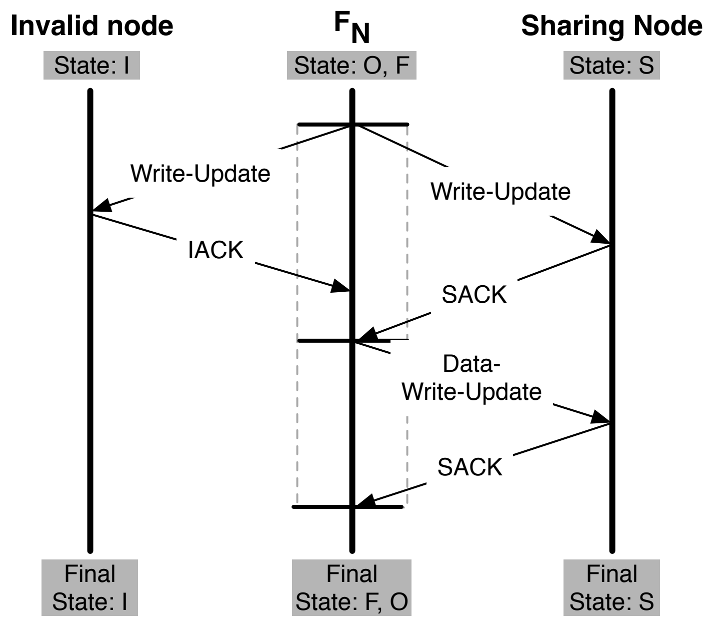


Figure 4.7: *Demonstrates the bandwidth optimized write-update option. While requiring four hops to complete, only sharers are sent the cache block data.*

valid caches (caches with the cache block as *Shared*). Hence, the second step only needs to send *Data-Write-Update* to the *Shared* nodes which were the nodes that responded with *SACK* in the first broadcast.

The second option is more bandwidth efficient than the first, only sending to sharers the cache block data. But it requires four hops to complete. Much like the first option, all broadcast messages sent by simultaneous requests to F_N during the write update are delayed until completion; F_N is ordering itself before all other requests.

Which option is used, and when write-update is used (compared to the usual write-invalidate write request) is a complex issue that we do not discuss further in this thesis.

4.3 Read Request Forwards as Shared

In Section 3.8.4, we showed how home could use the freedom to order newly added requests to minimize sharing invalidation. Here, we show that we can further minimize sharing invalidation in a read request that is in conflict with write requests.

When a read request Req_R sends *Cncl* and *Read* to home with a non-empty conflict list, home checks the *requestor-queue* and *conflict-table* to decide which nodes (if any) home should add to the tail of the *requestor-queue*. By the home assumption, this order represented by the *requestor-queue* is fixed as new requests are only added to the end of the order.

With this information, home can tell which nodes in Req_R's conflict list are ordered before Req_R. Given the set of requests to be added from the conflict list, R_{add}, any request not in R_{add} has already been added to the *requestor-queue* before. Home then checks for each of these requests not in R_{add}, are they ordered before Req_R or after. If the requests not in R_{add} are not present in the *requestor-queue*, then these requests have been removed so we assume the worst: that they were ordered before Req_R.

The reason Req_R invalidates when it forwards the cache block if one of the conflicting nodes is a write, is to not violate *SWMR* for that write. If Req_R does not invalidate, but keeps a *Shared* copy, then if one of the conflicting requests ordered after Req_R attempts to commit, then there still exists sharers; *SWMR* does not hold for the write request.

However, now that home knows which nodes in the Req_R's conflict list are ordered before Req_R, home can check if all *write* requests are ordered before Req_R. If there are no write requests ordered after Req_R, then Req_R is free to keep the cache block as *Shared* when it forwards the cache block and forwarding state on.

Figure 4.8 gives an example of three request all in conflict with one another, one write request and two read requests. The write request is the winning request, reaching the locally cached forwarding state first. Home decides the ordering is from node A to node C finally to node D.

The timeline proceeds as usual in *MESIF* until node C, the request in the middle of the ordering, receives *XFR(D)* with a message *do not downgrade*[8]. Even though node C has received a *GetX* message and downgrades to *Invalid* in *MESIF*, here node C can safely downgrade to *Shared* as the conflicting write in node N has committed.

Once all requests have finished, we are left with one more *Shared* copy than in original *MESIF*. With larger numbers of conflicts, there is opportunity for even more minimizing of sharer invalidation.

4.4 Winning Non-Conflicting Requests Without Home Message

In Section 3.5 and Section 3.6 we showed that the winning request (the request that gets the forwarding state first) that is non-conflicting has one of two orders, the *1b* or *1c* ordering as in Figure 3.10b.

For a winning request that receives all replies and the request is not in conflict with any other node then in *MESIF*, the request sends *Cncl* or *Read* to home with an empty

[8]This could be represented as an extra bit in the response messages from home.

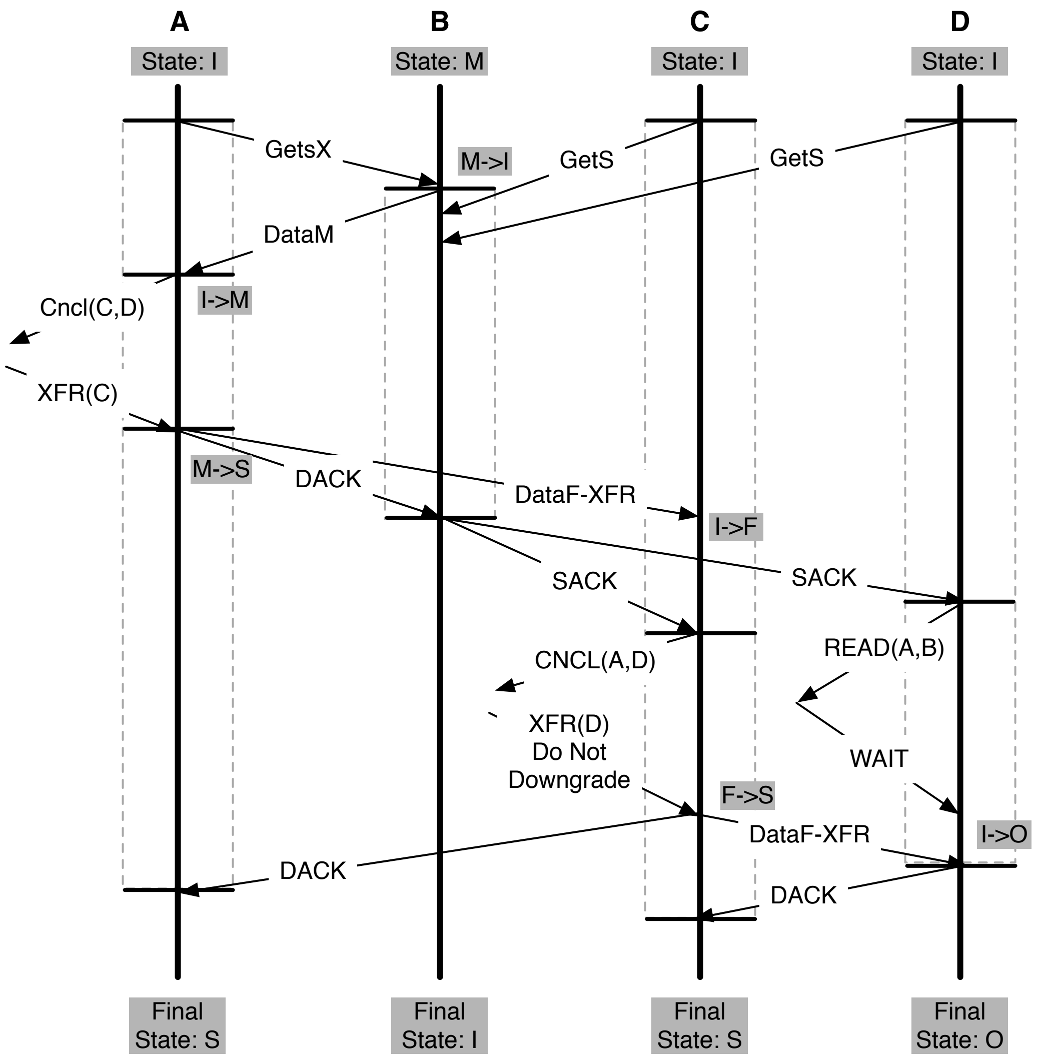


Figure 4.8: *Demonstrating how home uses* do not downgrade *to prevent the invalidation of node C as it forwards to node D. All the requests exchange broadcast messages and* Conflict *responses in their broadcast phases, these messages are omitted for clarity.*

conflict list. Home responds to *Cncl* with *ACK* and to *Read* with *DataE-Home*. The request receives this respond from home and commits, sending *DACK* to the forwarding node if it exists.

Home is only required to sort out orders when there is conflicting nodes and to access main memory. If the forwarding state is locally cached, the winning request is the first request to have its broadcast message reach the forwarding state. When the winning request has all replies and is not conflicting with any request, the request has both data and permissions. The *SWMR* invariant for the winning request will hold as each node will not violate *SWMR* for the request, and there cannot be any upgrading of permissions since that time (the winning request has the forwarding state). It is not necessary to send *Cncl* to home, and the winning request can immediately commit.

The concept is that the transfer of forwarding state from the forwarder to the winning request is silent; it is unobservable by any simultaneous request. Home is only required to sort out orders when there is conflicting nodes and to access main memory. Figure 4.9 shows three nodes, one with the forwarding state, the winning request, and another simultaneous request.

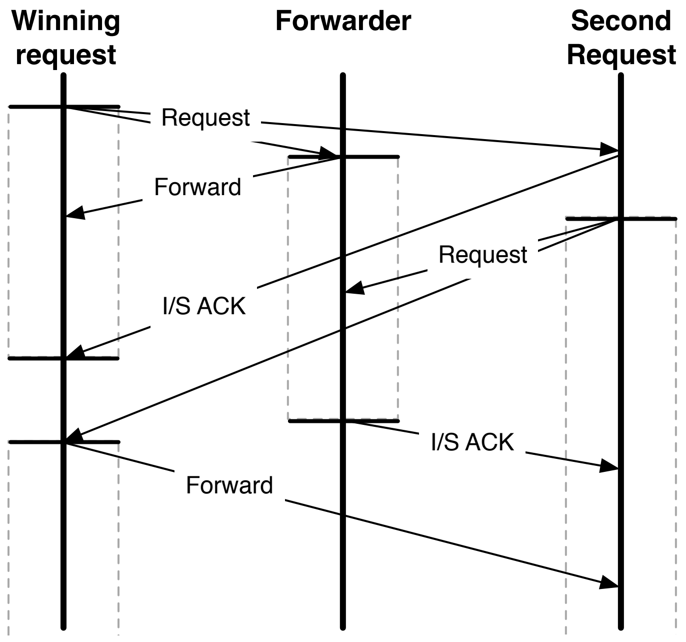


Figure 4.9: *Two simultaneous but non-conflicting requests. Because the winning request has no conflicting requests, can safely commit without sending a* Cncl *to home. The second request does not observe the first request at all.*

The winning request now immediately commits and completes once it has receives the last reply, the home phase does not exist. There are no delayed requests because only in the home phase can a request delay broadcast messages. The second request does not observe the winning request, or the forwarding node, and sees the world as shown in Figure 4.10. To the second request it appears as if there are no simultaneous requests, and that the second request is the winning request.

By eliminating the message to and from home, any simultaneous but non-conflicting requests will now not be delayed. This could result in a saving of two hops for a request, as the winning node sends *DACK* to the forwarding node two hops faster than before.

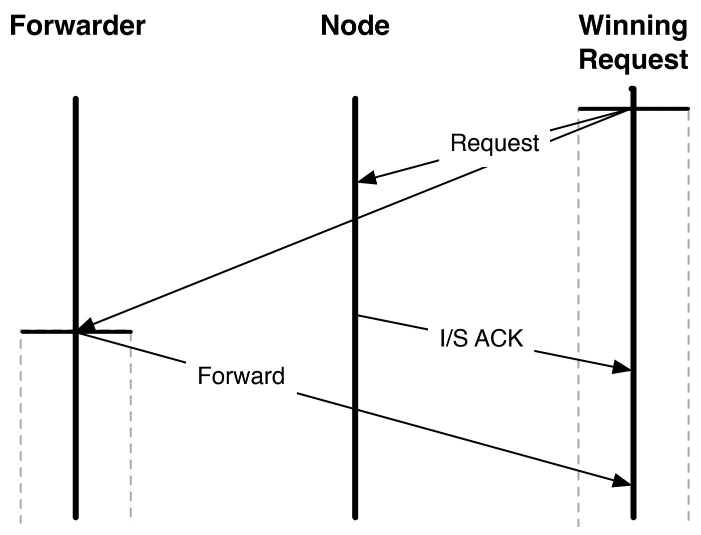


Figure 4.10: *Identical to Figure 4.9 but the world as observed by the second request (now the winning request here). The* node *appears to be delaying the broadcast message, but this is indistinguishable from the message and reply being delayed with* node *replying instantly to the request.*

Chapter 5

Conclusions and Future Work

In this thesis we presented the *MESIF* cache coherence protocol for fast two hop cache-to-cache latency requests, coupled with improved scalability by using a more scalable unordered or point-to-point interconnect. Unlike *Token Coherence*, requests in *MESIF* are guaranteed to succeed without backoff and retry of the requests.

We demonstrated how coherence is achieved with any number of conflicting requests. We discussed some common optimizations such as prefetching cache blocks from memory and home ordering requests intelligently.

Inspired by *Token Coherence* protocol, we extended the *MESIF* protocol to treat simultaneous read requests as non-conflicting which should improve performance especially for cache blocks that are rarely written but frequently read.

Another extension was to allow for write-updating. Even though *MESIF* is a write-invalidate cache coherency protocol, having the freedom to write-update could improve performance. The producer-consumer sharing pattern, where one core writes and other cores read, works well as the producer core can prevent invalidation of readers. Invalidation would require every reading core to initialize another read request, leading to unnecessary request broadcasting. Producer-consumer also works well as only the forwarding state cache can initiate a *MESIF* write update request, so the producer cache would keep the forwarding state.

This leads us to conclude that the *MESIF* cache coherence protocol has great potential as a future protocol. While we have not yet demonstrated this to be the case we anticipate future work, utilizing our implementation of *MESIF* in the Wisconsin Multifacet GEMS SLICC language, to evaluate how *MESIF* compares to other cache coherence protocols such as *Token Coherence* protocol.

Appendix A

Local Cache Controller SLICC Code

```
/*
    Copyright (C) 1999-2005 by Mark D. Hill and David A. Wood for the
    Wisconsin Multifacet Project.  Contact: gems@cs.wisc.edu
    http://www.cs.wisc.edu/gems/

    --------------------------------------------------------------------

    This file is part of the SLICC (Specification Language for
    Implementing Cache Coherence), a component of the Multifacet GEMS
    (General Execution-driven Multiprocessor Simulator) software
    toolset originally developed at the University of Wisconsin-Madison.


    SLICC was originally developed by Milo Martin with substantial
    contributions from Daniel Sorin.

    Substantial further development of Multifacet GEMS at the
    University of Wisconsin was performed by Alaa Alameldeen, Brad
    Beckmann, Jayaram Bobba, Ross Dickson, Dan Gibson, Pacia Harper,
    Derek Hower, Milo Martin, Michael Marty, Carl Mauer, Michelle Moravan,
    Kevin Moore, Manoj Plakal, Daniel Sorin, Haris Volos, Min Xu, and Luke
Yen.

    --------------------------------------------------------------------
    If your use of this software contributes to a published paper, we
    request that you (1) cite our summary paper that appears on our
    website (http://www.cs.wisc.edu/gems/) and (2) e-mail a citation
    for your published paper to gems@cs.wisc.edu.

    If you redistribute derivatives of this software, we request that
    you notify us and either (1) ask people to register with us at our
    website (http://www.cs.wisc.edu/gems/) or (2) collect registration
    information and periodically send it to us.

    --------------------------------------------------------------------

    Multifacet GEMS is free software; you can redistribute it and/or
    modify it under the terms of version 2 of the GNU General Public
    License as published by the Free Software Foundation.

    Multifacet GEMS is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with the Multifacet GEMS; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
    02111-1307, USA

    The GNU General Public License is contained in the file LICENSE.

### END HEADER ###
*/
/*
 * $Id: MOESI_CMP_token-dir.sm 1.6 05/01/19 15:48:35-06:00
mikem@royal16.cs.wisc.edu $
 */

/**

Modified by Andrew Hay (andrewh@cs.auckland.ac.nz), 2011
*/
```

```
machine(L1Cache, "MESIF protocol") {

    // Message buffers: this node TO the network
    MessageBuffer requestFromL1Cache, network="To", virtual_network="0",
ordered="false";
    MessageBuffer responseFromL1Cache, network="To", virtual_network="1",
ordered="false";
    MessageBuffer dataResponseFromL1Cache, network="To", virtual_network="4",
ordered="false";

    // Message buffers: this node FROM the network
    MessageBuffer requestToL1Cache, network="From", virtual_network="0",
ordered="false";
    MessageBuffer responseToL1Cache, network="From", virtual_network="1",
ordered="false";
    MessageBuffer dataResponseToL1Cache, network="From", virtual_network="4",
ordered="false";

    // STATES
    enumeration(State, desc="Cache states", default="L1Cache_State_I") {
        // Base states
        //NP,     "NP",       desc="Not Present";
        I,      "I",        desc="Idle";
        S,      "S",        desc="Shared";
        E,      "E",        desc="Exclusive";
        F,      "F",        desc="Forward";
        M,      "M",        desc="Modified";
        O,      "O",        desc="Forward+Modified, MESIF paper denotes this
as the 'FM' state";

        // Transient states I-M
        IM,     "IM",       desc="Transiting to exclusive (then doing a
store), not all replies nor data";
        IMH,    "IMH",      desc="Transiting to exclusive (then doing a
store), all replies but no data";
        IMF,    "IMF",      desc="Modified forwarded - has data, not all
replies";
        IMFH,   "IMFH",     desc="Modified forwarded - has data, all
```

```
replies";
        // Conflicting states
        IMC,    "IMC",      desc="IM but with conflicts";
        IMHC,   "IMHC",     desc="IMH but with conflicts";
        IMFC,   "IMFC",     desc="IMF but with conflicts";
        IMFHC,  "IMFHC",    desc="IMFH but with conflicts";
        IMWX,   "IMWX",     desc="All replies but home has told to wait for
data -- then transfer";
        IMWC,   "IMWC",     desc="All replies but home has told to wait for
data";

        // Transferring line states
        EI,     "EI",       desc="Forwarded data, awaiting DACK";
        MI,     "MI",       desc="Forwarded data, awaiting DACK";
        FI,     "FI",       desc="Forwarded data, awaiting DACK";
        OI,     "OI",       desc="Forwarded data, awaiting DACK";
        ES,     "ES",       desc="Forwarded data, awaiting DACK";
        MS,     "MS",       desc="Forwarded data, awaiting DACK";
        FS,     "FS",       desc="Forwarded data, awaiting DACK";
        OS,     "OS",       desc="Forwarded data, awaiting DACK";

        WB,     "WB",       desc="Writing data back to memory on L1
replacement";

        // Transient states I-S
        IS,     "IS",       desc="Transiting to shared, not all replies nor
data";
        ISH,    "ISH",      desc="Transiting to shared, all replies, no
data";
        ISF,    "ISF",      desc="Transiting to shared, forwarded data, not
all replies";
        ISFH,   "ISFH",     desc="Transiting to shared, forwarded data, all
replies";
        ISFM,   "ISFM",     desc="ISF but data is modified";
        ISFMH,  "ISFMH",    desc="ISFH but data is modified";
        // Conflicting states - S, shared
        ISC,    "ISC",      desc="IS but with conflicts";
        ISHC,   "ISHC",     desc="ISH but with conflicts";
        ISFC,   "ISFC",     desc="ISF but with conflicts";
        ISFHC,  "ISFHC",    desc="ISFH but with conflicts";
        ISFMC,  "ISFMC",    desc="ISFC but data is modified";
        ISFMHC, "ISFMHC",   desc="ISFHC but data is modified";

        ISWX,   "ISWX",     desc="All replies but home has told to wait for
data -- then transfer as forward";
        ISWC,   "ISWC",     desc="All replies but home has told to wait for
data";
        // Conflicting states - S, invalidating
        ISCI,   "ISCI",     desc="ISC but must invalidate at the end";
        ISHCI,  "ISHCI",    desc="ISHC but must invalidate at the end";
        ISFCI,  "ISFCI",    desc="ISFC but must invalidate at the end";
        ISFHCI, "ISFHCI",   desc="ISFHC but must invalidate at the end";

        ISFMCI, "ISFMCI",   desc="ISFCI but data is modified";
        ISFMHCI,"ISFMHCI",  desc="ISFHCI but data is modified";
        ISECI,  "ISECI",    desc="ISFCI but forwarded in state E";
        ISEHCI, "ISEHCI",   desc="ISFHCI but forwarded in state E";
        ISMCI,  "ISMCI",    desc="ISFCI but forwarded in state M";
        ISMHCI, "ISMHCI",   desc="ISFHCI but forwarded in state M";

        ISWXI,  "ISWXI",    desc="ISWX but must invalidate at the end";

        // Transient states F-M
        FM,     "FM",       desc="Transiting to exclusive, not all replies";
        FMH,    "FMH",      desc="Transiting to exclusive, all replies";
        // Conflicting states - invalidating
        FMC,    "FMC",      desc="FM but with conflicts";
        FMHC,   "FMHC",     desc="FMH but with conflicts";

        // Transient states Owned/Forward to M
        WR,     "WR",       desc="Transiting to exclusive, not all replies --
already had the forward state";
        WRH,    "WRH",      desc="Transiting to exclusive, all replies --
already had the forward state";

    }

    // EVENTS
    enumeration(Event, desc="Cache events") {
        Load,           desc="Load request from the processor";
        Store,          desc="Store request from the processor";
        Ifetch,         desc="Instruction fetch request from the processor";
        L1_Replacement, desc="L1 replacement";
        WB_Ack,         desc="Ack from L2, write back successful";

        Fwd_GETX,       desc="GETX from other processor";
        Fwd_GETS,       desc="GETS from other processor";
        Fwd_GET_INSTR,  desc="GET_INSTR from other processor";

        Fwd_Recycle,    desc="GET from other processor, must be recycled as
this is the second one in the same request";

        IACK,           desc="Acknowledgement of request, invalid state";
        IACK_Final,     desc="Final acknowledgement of request, invalid
state";
        SACK,           desc="Acknowledgement of request, shared state";
        SACK_Final,     desc="Final acknowledgement of request, shared
state";
        DACK,           desc="Data acknowledgement, can now respond to other
requests";

        DataF,          desc="Received shared data from a cache";
        DataF_Final,    desc="Received shared data from a cache, no more acks
needed";
        DataFM,         desc="Received forwarded/modified data from a cache";
        DataFM_Final,   desc="Received forwarded/modified data from a cache,
no more acks needed";
        DataE,          desc="Received exclusive data from a cache";
        DataE_Final,    desc="Received exclusive data from a cache, no more
acks needed";
        DataM,          desc="Received modified data from a cache";
        DataM_Final,    desc="Received modified data from a cache, no more
acks needed";

        //TxnF,           desc="Received transferred shared data from a
cache";
        //TxnFM,          desc="Received transferred forwarded/modified data
from a cache";
        //TxnE,           desc="Received transferred exclusive data from a
cache";
        //TxnM,           desc="Received transferred modified data from a
cache";

        Extra_Fwd_GETX,       desc="GETX from processor we are transferring
to";
        Extra_Fwd_GETS,       desc="GETS from processor we are transferring
to";
        Extra_Fwd_GET_INSTR,  desc="GET_INSTR from processor we are
transferring to";

        DataE_Home,     desc="Received exclusive data from home";
        Ack_Home,       desc="Received Ack from home";

        Conflict,       desc="Conflict message received";
        Conflict_Final, desc="Conflict message received, no more acks
needed";
        Transfer,       desc="Ack from home and Transfer to another node";
        Data_Transfer,  desc="Data from home and Transfer to anther node";
        Wait,           desc="Wait for data";
        Wait_Transfer,  desc="Wait for data and then Transfer to another
node";

        //Extra_Request,  desc="Triggered when a getX/S is received but we
are already transferring to that node";
    }

    // Internal types

    // CacheEntry
    structure(Entry, desc="...", interface="AbstractCacheEntry") {
      State CacheState,        desc="cache state";
      bool Dirty,              desc="Is the data dirty (different than
memory)?";
      DataBlock DataBlk,       desc="data for the block";
    }
    // TBE fields
    structure(TBE, desc="...") {
      Address Address,              desc="Physical address for this TBE";
      State TBEState,        desc="Transient state";
      DataBlock DataBlk,                desc="Buffer for the data block";
      bool Dirty, default="false",   desc="data is dirty";
      PrefetchBit Prefetch,       desc="Set if this was caused by a
prefetch";
      MachineID L1_FwdData,          desc="ID of the L1 cache that forwarded
data to us.";
      MachineID TransferMachine,          desc="ID of the L1 cache that we
should transfer the data to when we get it";
      //int PendingAcks, default="0", desc="number of pending acks";
      NetDest PendingAcks,              desc="The set of machines that has
sent acks";
      AccessModeType AccessMode,    desc="user/supervisor access type";
      MachineIDset ConflictMachs, desc="Stores the first machine to conflict
with this one if in a conflict";
      MachineIDset RequestMach, desc="Stores the machines that have sent a
request to this machine already";
    }

    // External types

    external_type(CacheMemory) {
      bool cacheAvail(Address);
      Address cacheProbe(Address);
      void allocate(Address);
      void deallocate(Address);
      Entry lookup(Address);
      void changePermission(Address, AccessPermission);
      bool isTagPresent(Address);
    }

    external_type(TBETable) {
      TBE lookup(Address);
      void allocate(Address);
      void deallocate(Address);
      bool isPresent(Address);
    }

    // global variables

    TBETable L1_TBEs, template_hack="<L1Cache_TBE>";

    CacheMemory L1IcacheMemory, template_hack="<L1Cache_Entry>",

constructor_hack='L1_CACHE_NUM_SETS_BITS,L1_CACHE_ASSOC,MachineType_L1Cache,in
t_to_string(i)+"_L1I"',
                                abstract_chip_ptr="true";
    CacheMemory L1DcacheMemory, template_hack="<L1Cache_Entry>",

constructor_hack='L1_CACHE_NUM_SETS_BITS,L1_CACHE_ASSOC,MachineType_L1Cache,in
t_to_string(i)+"_L1D"',
                                abstract_chip_ptr="true";

    MessageBuffer mandatoryQueue, ordered="false", abstract_chip_ptr="true";
    Sequencer sequencer, abstract_chip_ptr="true", constructor_hack="i";

    // Functions

    Entry getL1CacheEntry(Address addr), return_by_ref="yes" {
        if (L1DcacheMemory.isTagPresent(addr)) {
            return L1DcacheMemory[addr];
        }
        else {
            return L1IcacheMemory[addr];
        }
    }


    Event mandatory_request_type_to_event(CacheRequestType type) {
        if (type == CacheRequestType:LD) {
            return Event:Load;
        } else if (type == CacheRequestType:IFETCH) {
            return Event:Ifetch;
        } else if ((type == CacheRequestType:ST) || (type ==
CacheRequestType:ATOMIC)) {
```

```
            return Event:Store;
        } else {
            error("Invalid CacheRequestType");
        }
    }

    void changeL1Permission(Address addr, AccessPermission permission) {
        if (L1DcacheMemory.isTagPresent(addr)) {
            return L1DcacheMemory.changePermission(addr, permission);
        }
        else if(L1IcacheMemory.isTagPresent(addr)) {
            return L1IcacheMemory.changePermission(addr, permission);
        }
        else {
            error("cannot change permission, L1 block not present");
        }
    }

    bool isL1CacheTagPresent(Address addr) {
        return (L1DcacheMemory.isTagPresent(addr) ||
L1IcacheMemory.isTagPresent(addr));
    }


    void setState(Address addr, State state) {
        assert((L1DcacheMemory.isTagPresent(addr) &&
L1IcacheMemory.isTagPresent(addr)) == false);

        // MUST CHANGE
        if(L1_TBEs.isPresent(addr)) {
            L1_TBEs[addr].TBEState := state;
        }

        if (isL1CacheTagPresent(addr)) {
            getL1CacheEntry(addr).CacheState := state;

            // Set permission
            if (state == State:I) {
                changeL1Permission(addr, AccessPermission:Invalid);
            }
            //TODO: why is E read only?
            else if (state == State:S || state == State:F || state ==
State:E) {
                changeL1Permission(addr, AccessPermission:Read_Only);
            }
            else if (state == State:M) {
                changeL1Permission(addr, AccessPermission:Read_Write);
            }
            else {
                changeL1Permission(addr, AccessPermission:Busy);
            }
        }
    }

    State getState(Address addr) {
        if((L1DcacheMemory.isTagPresent(addr) &&
L1IcacheMemory.isTagPresent(addr)) == true){
            DEBUG_EXPR(id);
            DEBUG_EXPR(addr);
        }
        assert((L1DcacheMemory.isTagPresent(addr) &&
L1IcacheMemory.isTagPresent(addr)) == false);

        if(L1_TBEs.isPresent(addr)) {
            return L1_TBEs[addr].TBEState;
        }
        else if (isL1CacheTagPresent(addr)) {
            return getL1CacheEntry(addr).CacheState;
        }
        return State:I;
    }

    void registerReply(Address addr, MachineID sender) {
        L1_TBEs[addr].PendingAcks.add(sender);
    }

    // Includes the current ack as well (since registerReply is called first)
    bool acksRemaining(Address addr) {
        return L1_TBEs[addr].PendingAcks.count() < numberOfL1Cache(); // less
than number of other L1's + L2
    }

    void whoSentThis(MachineID m, ResponseMsg msg, int param) {

    }

    bool alreadySeenRequestingNode(Address addr, MachineID m) {
        if (L1_TBEs.isPresent(addr)) {
            return L1_TBEs[addr].RequestMach.contains(m);
        }
        else {
            return false;
        }
    }


    // Returns true if this is a post transfer intermediate state.
    bool transferstate(State s) {
        return s == State:MI || s == State:IMWC || s == State:IMWX || s ==
State:FS || s == State:OS || s == State:ISWC ||
               s == State:ISWX || s == State:ISWXI || s == State:FI || s ==
State:OI || s == State:EI || s == State:MS ||
               s == State:ES;
    }

    // Out ports

    out_port(requestNetwork_out, RequestMsg, requestFromL1Cache);
    out_port(responseNetwork_out, RequestMsg, responseFromL1Cache);
    out_port(dataResponseNetwork_out, RequestMsg, dataResponseFromL1Cache);

    // In ports

    // It's important to handle requests from other caches before own
requests for correctness.
    // Basically, because when going from a MI or similar state to I, those
buffered requests MUST be dealt
```

```
    // with before dealing with a load or store (screws up the conflict
reporting stuff)
    in_port(requestNetwork_in, RequestMsg, requestToL1Cache) {
        if (requestNetwork_in.isReady()) {
            peek(requestNetwork_in, RequestMsg) {
                assert(in_msg.Destination.isElement(machineID));
                // Check if this request is in fact being forfilled by a
transfer, if so, remove it from the queue.
                /*if (L1_TBEs.isPresent(in_msg.Address) &&
                            L1_TBEs[in_msg.Address].TransferMachine !=
machineID &&
                            L1_TBEs[in_msg.Address].TransferMachine ==
in_msg.Requestor) {

                    trigger(Event:Extra_Request, in_msg.Address);
                }*/
                if (L1_TBEs.isPresent(in_msg.Address) &&
                                L1_TBEs[in_msg.Address].TransferMachine !=
machineID &&
                                L1_TBEs[in_msg.Address].TransferMachine ==
in_msg.Requestor) {
                    if (in_msg.Type == CoherenceRequestType:GETX) {
                        trigger(Event:Extra_Fwd_GETX, in_msg.Address);
                    }
                    else if (in_msg.Type == CoherenceRequestType:GETS) {
                        trigger(Event:Extra_Fwd_GETS, in_msg.Address);
                    }
                    else if (in_msg.Type == CoherenceRequestType:GET_INSTR) {
                        trigger(Event:Extra_Fwd_GET_INSTR, in_msg.Address);
                    }
                }
                else if (in_msg.Type == CoherenceRequestType:GETX) {
                    // upgrade transforms to GETX due to race
                    if (alreadySeenRequestingNode(in_msg.Address,
in_msg.Requestor)) {
                        trigger(Event:Fwd_Recycle, in_msg.Address);
                    }
                    else {
                        trigger(Event:Fwd_GETX, in_msg.Address);
                    }
                }
                else if (in_msg.Type == CoherenceRequestType:GETS) {

                    if (alreadySeenRequestingNode(in_msg.Address,
in_msg.Requestor)) {
                        trigger(Event:Fwd_Recycle, in_msg.Address);
                    }
                    else {
                        trigger(Event:Fwd_GETS, in_msg.Address);
                    }
                }
                else if (in_msg.Type == CoherenceRequestType:GET_INSTR) {

                    if (alreadySeenRequestingNode(in_msg.Address,
in_msg.Requestor)) {
                        trigger(Event:Fwd_Recycle, in_msg.Address);
                    }
                    else {
                        trigger(Event:Fwd_GET_INSTR, in_msg.Address);
                    }
                }
                else {
                    error("Invalid forwarded request type");
                }
            }
        }
    }

    in_port(mandatoryQueue_in, CacheMsg, mandatoryQueue, desc="...") {
        if (mandatoryQueue_in.isReady()) {
            peek(mandatoryQueue_in, CacheMsg) {
                if (in_msg.Type == CacheRequestType:IFETCH) {

                    // Check to see if it is in the OTHER L1
                    if (L1DcacheMemory.isTagPresent(in_msg.Address)) {
                        // The block is in the wrong L1, put the request on
the queue to the shared L2
                        trigger(Event:L1_Replacement, in_msg.Address);
                    }
                    if (L1IcacheMemory.isTagPresent(in_msg.Address)) {
                        // The tag matches for the L1, so the L1 asks the L2
for it.
                        trigger(mandatory_request_type_to_event(in_msg.Type),
in_msg.Address);
                    }
                    else {
                        if (L1IcacheMemory.cacheAvail(in_msg.Address)) {
                            // L1 does't have the line, but we have space for
it in the L1 so let's see if the L2 has it

trigger(mandatory_request_type_to_event(in_msg.Type), in_msg.Address);
                        }
                        else {
                            // No room in the L1, so we need to make room in
the L1
                            trigger(Event:L1_Replacement,
L1IcacheMemory.cacheProbe(in_msg.Address));
                        }
                    }
                }
                // Data Access
                else {
                    if (L1DcacheMemory.isTagPresent(in_msg.Address)) {
                        // The tag matches for the L1, so the L1 fetches the
line.  We know it can't be in the L2 due to exclusion
                        trigger(mandatory_request_type_to_event(in_msg.Type),
in_msg.Address);
                    } else {
                        if (L1DcacheMemory.cacheAvail(in_msg.Address)) {
                        // L1 does't have the line, but we have space for it
in the L1

trigger(mandatory_request_type_to_event(in_msg.Type), in_msg.Address);
                        } else {
                            // No room in the L1, so we need to make room
                            trigger(Event:L1_Replacement,
L1DcacheMemory.cacheProbe(in_msg.Address));
```

```
                    }
                }
            }
        }
    }
}



in_port(responseNetwork_in, ResponseMsg, responseToL1Cache) {
    if (responseNetwork_in.isReady()) {
        peek(responseNetwork_in, ResponseMsg) {
            assert(in_msg.Destination.isElement(machineID));


            if (in_msg.SenderMachine == MachineType:L1Cache) {
                if (in_msg.Type == CoherenceResponseType:IACK) {
                    registerReply(in_msg.Address, in_msg.Sender);

                    whoSentThis(in_msg.Sender, in_msg,
L1_TBEs[in_msg.Address].PendingAcks.count());

                    if (acksRemaining(in_msg.Address)) {
                        trigger(Event:IACK, in_msg.Address);
                    }
                    else {
                        trigger(Event:IACK_Final, in_msg.Address);
                    }
                }
                else if (in_msg.Type == CoherenceResponseType:SACK) {
                    registerReply(in_msg.Address, in_msg.Sender);

                    if (acksRemaining(in_msg.Address)) {
                        trigger(Event:SACK, in_msg.Address);
                    }
                    else {
                        trigger(Event:SACK_Final, in_msg.Address);
                    }
                }
```

```
                    else if (in_msg.Type == CoherenceResponseType:Conflict) {
                        registerReply(in_msg.Address, in_msg.Sender);

                        if (acksRemaining(in_msg.Address)) {
                            trigger(Event:Conflict, in_msg.Address);
                        }
                        else {
                            trigger(Event:Conflict_Final, in_msg.Address);
                        }
                    }
                }
                else if (in_msg.SenderMachine == MachineType:L2Cache) {
                    if (in_msg.Type == CoherenceResponseType:DataF_E) {
                        trigger(Event:DataE_Home, in_msg.Address);
                    }
                    else if (in_msg.Type == CoherenceResponseType:ACK) {
                        trigger(Event:Ack_Home, in_msg.Address);
                    }
                    else if (in_msg.Type == CoherenceResponseType:XFR) {
                        trigger(Event:Transfer, in_msg.Address);
                    }
                    else if (in_msg.Type == CoherenceResponseType:DXFR) {
                        trigger(Event:Data_Transfer, in_msg.Address);
                    }
                    else if (in_msg.Type == CoherenceResponseType:Wait) {
                        trigger(Event:Wait, in_msg.Address);
                    }
                    else if (in_msg.Type == CoherenceResponseType:WaitXFR) {
                        trigger(Event:Wait_Transfer, in_msg.Address);
                    }
                    else if (in_msg.Type == CoherenceResponseType:MEMORY_ACK)
{
                        trigger(Event:WB_Ack, in_msg.Address);
                    }
                    // Home, being an L2 cache, also acts as a Node (a
passive one that makes no requests)
                    else if (in_msg.Type == CoherenceResponseType:IACK) {
                        registerReply(in_msg.Address, in_msg.Sender);
                        if (acksRemaining(in_msg.Address)) {
                            trigger(Event:IACK, in_msg.Address);
                        }
                        else {
                            trigger(Event:IACK_Final, in_msg.Address);
                        }
                    }
                    // else if (in_msg.Type ==
CoherenceResponseType:DataF_FM) {
                    //     if (acksRemaining(in_msg.Address)) {
                    //         trigger(Event:DataFM, in_msg.Address);
                    //     }
                    //     else {
                    //         trigger(Event:DataFM_Final, in_msg.Address);
                    //     }
                    // }
                    // else if (in_msg.Type == CoherenceResponseType:DataF_M)
{
                    //      if (acksRemaining(in_msg.Address)) {
                    //          trigger(Event:DataM, in_msg.Address);
                    //      }
                    //      else {
                    //          trigger(Event:DataM_Final, in_msg.Address);
                    //      }
                    //   }

                }
            }
        }
    }

    // Data forward from L1 caches and DACKS
    in_port(dataResponseNetwork_in, ResponseMsg, dataResponseToL1Cache) {
        if (dataResponseNetwork_in.isReady()) {
            peek(dataResponseNetwork_in, ResponseMsg) {

                if (in_msg.Type == CoherenceResponseType:DataF_F ||
                    in_msg.Type == CoherenceResponseType:DataF_E ||
                    in_msg.Type == CoherenceResponseType:DataF_M ||
                    in_msg.Type == CoherenceResponseType:DataF_FM) {

                    registerReply(in_msg.Address, in_msg.Sender);
                }

                if (in_msg.Type == CoherenceResponseType:DataF_F ||
                    in_msg.Type == CoherenceResponseType:TxnF_F) {

                    if (acksRemaining(in_msg.Address)) {
                        trigger(Event:DataF, in_msg.Address);
                    }
                    else {
                        trigger(Event:DataF_Final, in_msg.Address);
                    }
                }
                else if (in_msg.Type == CoherenceResponseType:DataF_E ||
                         in_msg.Type == CoherenceResponseType:TxnF_E) {

                    if (acksRemaining(in_msg.Address)) {
                        trigger(Event:DataE, in_msg.Address);
                    }
                    else {
                        trigger(Event:DataE_Final, in_msg.Address);
                    }
                }
                else if (in_msg.Type == CoherenceResponseType:DataF_M ||
                         in_msg.Type == CoherenceResponseType:TxnF_M) {

                    if (acksRemaining(in_msg.Address)) {
                        trigger(Event:DataM, in_msg.Address);
                    }
                    else {
                        trigger(Event:DataM_Final, in_msg.Address);
                    }
                }
                else if (in_msg.Type == CoherenceResponseType:DataF_FM ||
                         in_msg.Type == CoherenceResponseType:TxnF_FM) {

                    if (acksRemaining(in_msg.Address)) {
```

```
                        trigger(Event:DataFM, in_msg.Address);
                    }
                    else {
                        trigger(Event:DataFM_Final, in_msg.Address);
                    }
                }
                else if (in_msg.Type == CoherenceResponseType:DACK) {
                    trigger(Event:DACK, in_msg.Address);
                }
                else {
                    assert(false);
                }
            }
        }
    }

    // Events

    action(a_issueGETS, "a", desc="Issue GETS") {
        peek(mandatoryQueue_in, CacheMsg) {
            // Make a normal request to the home/L2 cache.
            enqueue(requestNetwork_out, RequestMsg,
latency="L1_REQUEST_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceRequestType:GETS;
                //out_msg.SenderMachine := MachineType:L1Cache;
                out_msg.Requestor := machineID;

out_msg.Destination.add(map_L1CacheMachId_to_L2Cache(address,machineID));
                out_msg.MessageSize := MessageSizeType:Control;
                out_msg.Prefetch := in_msg.Prefetch;
                out_msg.AccessMode := in_msg.AccessMode;
                //out_msg.ConflictMachs := machineID;
            }

            // Broadcast request to L1 caches
            enqueue(requestNetwork_out, RequestMsg,
latency="L1_REQUEST_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceRequestType:GETS;
                //out_msg.SenderMachine := MachineType:L1Cache;
                out_msg.Requestor := machineID;
                out_msg.Destination := getOtherLocalL1IDs(machineID);
                out_msg.MessageSize := MessageSizeType:Control;
                out_msg.Prefetch := in_msg.Prefetch;
                out_msg.AccessMode := in_msg.AccessMode;
            }
        }
    }

    action(b_issueGETX, "b", desc="Issue GETX") {
        peek(mandatoryQueue_in, CacheMsg) {
            // Make a normal request to the home/L2 cache.
            enqueue(requestNetwork_out, RequestMsg,
latency="L1_REQUEST_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceRequestType:GETX;
                //out_msg.SenderMachine := MachineType:L1Cache;
                out_msg.Requestor := machineID;

out_msg.Destination.add(map_L1CacheMachId_to_L2Cache(address,machineID));
                out_msg.MessageSize := MessageSizeType:Control;
                out_msg.Prefetch := in_msg.Prefetch;
                out_msg.AccessMode := in_msg.AccessMode;
                //out_msg.ConflictMachs := machineID;
            }

            // Broadcast request to L1 caches
            enqueue(requestNetwork_out, RequestMsg,
latency="L1_REQUEST_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceRequestType:GETX;
                //out_msg.SenderMachine := MachineType:L1Cache;
                out_msg.Requestor := machineID;
                out_msg.Destination := getOtherLocalL1IDs(machineID);
                out_msg.MessageSize := MessageSizeType:Control;
                out_msg.Prefetch := in_msg.Prefetch;
                out_msg.AccessMode := in_msg.AccessMode;
            }
        }
    }

    action(ai_issueGETINSTR, "ai", desc="Issue GETINSTR") {
        peek(mandatoryQueue_in, CacheMsg) {
            // Make a normal request to the home/L2 cache.
            enqueue(requestNetwork_out, RequestMsg,
latency="L1_REQUEST_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceRequestType:GET_INSTR;
                //out_msg.SenderMachine := MachineType:L1Cache;
                out_msg.Requestor := machineID;

out_msg.Destination.add(map_L1CacheMachId_to_L2Cache(address,machineID));
                out_msg.MessageSize := MessageSizeType:Control;
                out_msg.Prefetch := in_msg.Prefetch;
                out_msg.AccessMode := in_msg.AccessMode;
                //out_msg.ConflictMachs := machineID;
            }

            // Broadcast request to L1 caches
            enqueue(requestNetwork_out, RequestMsg,
latency="L1_REQUEST_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceRequestType:GET_INSTR;
                //out_msg.SenderMachine := MachineType:L1Cache;
                out_msg.Requestor := machineID;
                out_msg.Destination := getOtherLocalL1IDs(machineID);
                out_msg.MessageSize := MessageSizeType:Control;
                out_msg.Prefetch := in_msg.Prefetch;
                out_msg.AccessMode := in_msg.AccessMode;
            }
        }
    }

    action(c_sendCnclHome, "c", desc="Issue cancel to home") {
        enqueue(requestNetwork_out, RequestMsg, latency="L1_REQUEST_LATENCY")
{
```

```
            out_msg.Address := address;
            out_msg.Type := CoherenceRequestType:CNCL;
            //out_msg.SenderMachine := MachineType:L1Cache;
            out_msg.Requestor := machineID;

out_msg.Destination.add(map_L1CacheMachId_to_L2Cache(address,machineID));
            out_msg.MessageSize := MessageSizeType:Control;
            out_msg.AccessMode := L1_TBEs[address].AccessMode;
            out_msg.Prefetch := L1_TBEs[address].Prefetch;
            out_msg.ConflictMachs := L1_TBEs[address].ConflictMachs;
        }
    }



    action(d_sendReadHome, "d", desc="Issue read to home") {
        enqueue(requestNetwork_out, RequestMsg, latency="L1_REQUEST_LATENCY")
{
            out_msg.Address := address;
            out_msg.Type := CoherenceRequestType:READ;
            //out_msg.SenderMachine := MachineType:L1Cache;
            out_msg.Requestor := machineID;

out_msg.Destination.add(map_L1CacheMachId_to_L2Cache(address,machineID));
            out_msg.MessageSize := MessageSizeType:Control;
            out_msg.AccessMode := L1_TBEs[address].AccessMode;
            out_msg.Prefetch := L1_TBEs[address].Prefetch;
            out_msg.ConflictMachs := L1_TBEs[address].ConflictMachs;
        }
    }

    action(df_sendForwardDataToRequestor, "df", desc="send data to requestor
as forward") {
        peek(requestNetwork_in, RequestMsg) {
            enqueue(dataResponseNetwork_out, ResponseMsg,
latency="L2_RESPONSE_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:DataF_F;
                out_msg.SenderMachine := MachineType:L1Cache;
                out_msg.DataBlk := getL1CacheEntry(address).DataBlk;
                out_msg.Dirty := getL1CacheEntry(address).Dirty;
                out_msg.Sender := machineID;
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.MessageSize := MessageSizeType:Response_Data;
                out_msg.SenderMachine := MachineType:L1Cache;
            }
        }
    }

    action(de_sendExclusiveDataToRequestor, "de", desc="send data to
requestor as exclusive") {
        peek(requestNetwork_in, RequestMsg) {
            enqueue(dataResponseNetwork_out, ResponseMsg,
latency="L2_RESPONSE_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:DataF_E;
                out_msg.SenderMachine := MachineType:L1Cache;
                out_msg.DataBlk := getL1CacheEntry(address).DataBlk;
                out_msg.Dirty := getL1CacheEntry(address).Dirty;
                out_msg.Sender := machineID;
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.MessageSize := MessageSizeType:Response_Data;
                out_msg.SenderMachine := MachineType:L1Cache;
            }
        }
    }

    action(dm_sendModifiedDataToRequestor, "dm", desc="send data to requestor
as modified") {
        peek(requestNetwork_in, RequestMsg) {
            enqueue(dataResponseNetwork_out, ResponseMsg,
latency="L2_RESPONSE_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:DataF_M;
                out_msg.SenderMachine := MachineType:L1Cache;
                out_msg.DataBlk := getL1CacheEntry(address).DataBlk;
                out_msg.Dirty := getL1CacheEntry(address).Dirty;
                out_msg.Sender := machineID;
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.MessageSize := MessageSizeType:Response_Data;
                out_msg.SenderMachine := MachineType:L1Cache;
            }
        }
    }

    action(dfm_sendForwardModifiedDataToRequestor, "dfm", desc="send data to
requestor as forwarded/modified") {
        peek(requestNetwork_in, RequestMsg) {
            enqueue(dataResponseNetwork_out, ResponseMsg,
latency="L2_RESPONSE_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:DataF_FM;
                out_msg.SenderMachine := MachineType:L1Cache;
                out_msg.DataBlk := getL1CacheEntry(address).DataBlk;
                out_msg.Dirty := getL1CacheEntry(address).Dirty;
                out_msg.Sender := machineID;
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.MessageSize := MessageSizeType:Response_Data;
                out_msg.SenderMachine := MachineType:L1Cache;
            }
        }
    }

    action(di_sendIACKToRequestor, "di", desc="send IACK to requestor") {
        peek(requestNetwork_in, RequestMsg) {
            enqueue(responseNetwork_out, ResponseMsg,
latency="L1_REQUEST_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:IACK;
                out_msg.SenderMachine := MachineType:L1Cache;
                out_msg.Dirty := false;
                out_msg.Sender := machineID;
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.MessageSize := MessageSizeType:Response_Control;
            }
        }
    }
    action(ds_sendSACKToRequestor, "ds", desc="send SACK to requestor") {
        peek(requestNetwork_in, RequestMsg) {
            enqueue(responseNetwork_out, ResponseMsg,
latency="L1_REQUEST_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:SACK;
                out_msg.SenderMachine := MachineType:L1Cache;
                out_msg.Dirty := false;
                out_msg.Sender := machineID;
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.MessageSize := MessageSizeType:Response_Control;
            }
        }
    }

    action(dd_sendDACKToRequestor, "dd", desc="send DACK to forwarder") {
        enqueue(dataResponseNetwork_out, ResponseMsg,
latency="L1_REQUEST_LATENCY") {
            out_msg.Address := address;
            out_msg.Type := CoherenceResponseType:DACK;
            out_msg.SenderMachine := MachineType:L1Cache;
            out_msg.Dirty := false;
            out_msg.Sender := machineID;
            out_msg.Destination.add(L1_TBEs[address].L1_FwdData);
            out_msg.MessageSize := MessageSizeType:Response_Control;
        }
    }

    action(sc_sendConflict, "sc", desc="Send conflict to requestor") {
        peek(requestNetwork_in, RequestMsg) {
            enqueue(responseNetwork_out, ResponseMsg,
latency="L1_REQUEST_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:Conflict;
                out_msg.SenderMachine := MachineType:L1Cache;
                out_msg.Dirty := false;
                out_msg.Sender := machineID;
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.MessageSize := MessageSizeType:Response_Control;
            }
        }
    }

    action(tm_transferModified, "tm", desc="Transfers the data as modified
state - as ordered by home") {
        enqueue(dataResponseNetwork_out, ResponseMsg,
latency="L2_RESPONSE_LATENCY") {
            out_msg.Address := address;
            out_msg.Type := CoherenceResponseType:TxnF_M;
            out_msg.SenderMachine := MachineType:L1Cache;
            out_msg.DataBlk := getL1CacheEntry(address).DataBlk;
            out_msg.Dirty := getL1CacheEntry(address).Dirty;
            out_msg.Sender := machineID;
            out_msg.Destination.add(L1_TBEs[address].TransferMachine);
            out_msg.MessageSize := MessageSizeType:Response_Data;
            out_msg.SenderMachine := MachineType:L1Cache;
        }
    }

    action(te_transferExclusive, "te", desc="Transfers the data as exclusive
state - as ordered by home") {
        enqueue(dataResponseNetwork_out, ResponseMsg,
latency="L2_RESPONSE_LATENCY") {
            out_msg.Address := address;
            out_msg.Type := CoherenceResponseType:TxnF_E;
            out_msg.SenderMachine := MachineType:L1Cache;
            out_msg.DataBlk := getL1CacheEntry(address).DataBlk;
            out_msg.Dirty := getL1CacheEntry(address).Dirty;
            out_msg.Sender := machineID;
            out_msg.Destination.add(L1_TBEs[address].TransferMachine);
            out_msg.MessageSize := MessageSizeType:Response_Data;
            out_msg.SenderMachine := MachineType:L1Cache;
        }
    }

    action(tfm_transferForwardModified, "tfm", desc="Transfers the data as
forward/modified state - as ordered by home") {
        enqueue(dataResponseNetwork_out, ResponseMsg,
latency="L2_RESPONSE_LATENCY") {
            out_msg.Address := address;
            out_msg.Type := CoherenceResponseType:TxnF_FM;
            out_msg.SenderMachine := MachineType:L1Cache;
            out_msg.DataBlk := getL1CacheEntry(address).DataBlk;
            out_msg.Dirty := getL1CacheEntry(address).Dirty;
            out_msg.Sender := machineID;
            out_msg.Destination.add(L1_TBEs[address].TransferMachine);
            out_msg.MessageSize := MessageSizeType:Response_Data;
            out_msg.SenderMachine := MachineType:L1Cache;
        }
    }

    action(tf_transferForward, "tf", desc="Transfers the data as forward
state - as ordered by home") {
        enqueue(dataResponseNetwork_out, ResponseMsg,
latency="L2_RESPONSE_LATENCY") {
            out_msg.Address := address;
            out_msg.Type := CoherenceResponseType:TxnF_F;
            out_msg.SenderMachine := MachineType:L1Cache;
            out_msg.DataBlk := getL1CacheEntry(address).DataBlk;
            out_msg.Dirty := getL1CacheEntry(address).Dirty;
            out_msg.Sender := machineID;
            out_msg.Destination.add(L1_TBEs[address].TransferMachine);
            out_msg.MessageSize := MessageSizeType:Response_Data;
            out_msg.SenderMachine := MachineType:L1Cache;
        }
    }

    // action(g_issuePUTX, "g", desc="send data to the L2 cache") {
    //     enqueue(requestNetwork_out, RequestMsg,
latency="L2_RESPONSE_LATENCY") {
    //         out_msg.Address := address;
    //         out_msg.Type := CoherenceRequestType:PUTX;
    //         out_msg.DataBlk := getL1CacheEntry(address).DataBlk;
    //         out_msg.Dirty := getL1CacheEntry(address).Dirty;
    //         out_msg.Requestor:= machineID;
    //         out_msg.Destination.add(map_L1CacheMachId_to_L2Cache(address,
machineID));
```

```
    //         //if (getL1CacheEntry(address).Dirty) {
    //         out_msg.MessageSize := MessageSizeType:Writeback_Data;
    //         //}
    //         //else {
    //         //    out_msg.MessageSize :=
MessageSizeType:Writeback_Control;
    //         //}
    //     }
    // }

    action(gf_issueL2FowardModified, "gfm", desc="send data to the L2 cache
as forward modified") {
        enqueue(requestNetwork_out, RequestMsg,
latency="L2_RESPONSE_LATENCY") {
            out_msg.Address := address;
            out_msg.Type := CoherenceRequestType:DataFM;
            out_msg.DataBlk := getL1CacheEntry(address).DataBlk;
            out_msg.Dirty := getL1CacheEntry(address).Dirty;
            out_msg.Requestor:= machineID;
            out_msg.Destination.add(map_L1CacheMachId_to_L2Cache(address,
machineID));
            //if (getL1CacheEntry(address).Dirty) {
            out_msg.MessageSize := MessageSizeType:Writeback_Data;
            //}
            //else {
            //    out_msg.MessageSize := MessageSizeType:Writeback_Control;
            //}
        }
    }

    action(gm_issueL2Modified, "gm", desc="send data to the L2 cache as
modified") {
        enqueue(requestNetwork_out, RequestMsg,
latency="L2_RESPONSE_LATENCY") {
            out_msg.Address := address;
            out_msg.Type := CoherenceRequestType:DataM;
            out_msg.DataBlk := getL1CacheEntry(address).DataBlk;
            out_msg.Dirty := getL1CacheEntry(address).Dirty;
            out_msg.Requestor:= machineID;
            out_msg.Destination.add(map_L1CacheMachId_to_L2Cache(address,
machineID));
            //if (getL1CacheEntry(address).Dirty) {
            out_msg.MessageSize := MessageSizeType:Writeback_Data;
            //}
            //else {
            //    out_msg.MessageSize := MessageSizeType:Writeback_Control;
            //}
        }
    }

    action(h_load_hit, "h", desc="Notify sequencer the load completed.") {
        DEBUG_EXPR(getL1CacheEntry(address).DataBlk);
        sequencer.readCallback(address, getL1CacheEntry(address).DataBlk);
    }

    action(hh_store_hit, "\h", desc="Notify sequencer that store completed.")
{
        DEBUG_EXPR(getL1CacheEntry(address).DataBlk);
        sequencer.writeCallback(address, getL1CacheEntry(address).DataBlk);
        getL1CacheEntry(address).Dirty := true;
    }


    action(i_allocateTBE, "i", desc="Allocate TBE") {
        check_allocate(L1_TBEs);
        L1_TBEs.allocate(address);
        peek(mandatoryQueue_in, CacheMsg) {
            L1_TBEs[address].Prefetch := in_msg.Prefetch;
        }
        L1_TBEs[address].Dirty := getL1CacheEntry(address).Dirty;
        L1_TBEs[address].DataBlk := getL1CacheEntry(address).DataBlk;
        // Signifies that there is no conflict -- yet
        //L1_TBEs[address].ConflictingMachine := machineID;
        // Signifies that we are not transferring to anyone
        L1_TBEs[address].TransferMachine := machineID;
    }

    action(v_recordForwarder, "v", desc="Records which cache sent us the
data, so we can send a DACK") {
        peek(dataResponseNetwork_in, ResponseMsg) {
            L1_TBEs[address].L1_FwdData := in_msg.Sender;
        }
    }

    /*action(v_responseRecordForwarder, "rv", desc="Records which cache sent
us the data, so we can send a DACK (From response network)") {
        peek(responseNetwork_in, ResponseMsg) {
            L1_TBEs[address].L1_FwdData := in_msg.Sender;
        }
    }*/

    action(cr_recordConflictMachineRequest, "cr", desc="Records the
conflicting machine when the conflict is known from a request") {
        peek(requestNetwork_in, RequestMsg) {
            if (L1_TBEs[address].ConflictMachs.contains(in_msg.Requestor) ==
false) {
                L1_TBEs[address].ConflictMachs.add(in_msg.Requestor);
            }
            if (L1_TBEs[address].RequestMach.contains(in_msg.Requestor) ==
false) {
                L1_TBEs[address].RequestMach.add(in_msg.Requestor);
            }
        }
    }

    action(cc_recordConflictMachineResponse, "\c", desc="Records the
conflicting machine when the conflict is known from a response") {
        peek(responseNetwork_in, ResponseMsg) {
            if (L1_TBEs[address].ConflictMachs.contains(in_msg.Sender) ==
false) {
                L1_TBEs[address].ConflictMachs.add(in_msg.Sender);
            }
        }
    }

    action(ct_recordTransferMachine, "ct", desc="Records the machine to
transfer the data to") {
        peek(responseNetwork_in, ResponseMsg) {
            L1_TBEs[address].TransferMachine := in_msg.Transfer;
        }
    }

    action(o_popIncomingResponseQueue, "o", desc="Pop Incoming Response queue
and profile the delay within this virtual network") {
        profileMsgDelay(1, responseNetwork_in.dequeue_getDelayCycles());
    }

    action(od_popIncomingDataResponseQueue, "od", desc="Pop Incoming Data
Response queue and profile the delay within this virtual network") {
        profileMsgDelay(4, dataResponseNetwork_in.dequeue_getDelayCycles());
    }

    action(s_deallocateTBE, "s", desc="Deallocate TBE if it exists") {
        if (L1_TBEs.isPresent(address)) {
            L1_TBEs.deallocate(address);
        }
    }

    action(u_writeDataToL1Cache, "u", desc="Write data to cache from
forwarded L1 data") {
        peek(dataResponseNetwork_in, ResponseMsg) {
          getL1CacheEntry(address).DataBlk := in_msg.DataBlk;
            getL1CacheEntry(address).Dirty := in_msg.Dirty;
        }
    }

    action(ru_responseWriteDataToL1Cache, "ru", desc="Write data to cache
from forwarded L1 data (From response network)") {
        peek(responseNetwork_in, ResponseMsg) {
          getL1CacheEntry(address).DataBlk := in_msg.DataBlk;
            getL1CacheEntry(address).Dirty := in_msg.Dirty;
        }
    }

    action(uu_writeHomeDataToL1Cache, "\u", desc="Write data to cache data
from Home") {
        peek(responseNetwork_in, ResponseMsg) {
          getL1CacheEntry(address).DataBlk := in_msg.DataBlk;
            getL1CacheEntry(address).Dirty := in_msg.Dirty;
        }
    }

    action(k_popMandatoryQueue, "k", desc="Pop mandatory queue.") {
        mandatoryQueue_in.dequeue();
    }

    action(ff_deallocateL1CacheBlock, "\f", desc="Deallocate L1 cache block.
Sets the cache to not present, allowing a replacement in parallel with a
fetch.") {
        if (L1DcacheMemory.isTagPresent(address)) {
            L1DcacheMemory.deallocate(address);
        } else {
            L1IcacheMemory.deallocate(address);
        }
    }

    action(l_popRequestQueue, "l", desc="Pop incoming request queue and
profile the delay within this virtual network") {
        profileMsgDelay(0, requestNetwork_in.dequeue_getDelayCycles());
    }

    action(oo_allocateL1DCacheBlock, "\o", desc="Set L1 D-cache tag equal to
tag of block B.") {
        if (L1DcacheMemory.isTagPresent(address) == false) {
            L1DcacheMemory.allocate(address);
        }
    }

    action(pp_allocateL1ICacheBlock, "\p", desc="Set L1 I-cache tag equal to
tag of block B.") {
        if (L1IcacheMemory.isTagPresent(address) == false) {
            L1IcacheMemory.allocate(address);
        }
    }
    action(rr_recycleRequestQueue, "\r", desc="Send the head of the request
queue to the back of the queue.") {
        requestNetwork_in.recycle();
    }

    action(zz_recycleMandatoryQueue, "\z", desc="Send the head of the
mandatory queue to the back of the queue.") {
        mandatoryQueue_in.recycle();
    }

    // Transitions

    transition({IM, IMF, IMFH, IMH, IS, ISH, ISF, ISFH, EI, MI, FI, OI, ES,
MS, FS, OS,
                IMC, IMHC, IMFC, IMFHC, IMWX, IMWC, ISC, ISFC, ISFHC, ISHC,
ISWX, ISWC,
                ISCI, ISFCI, ISFHCI, ISHCI, ISWXI, FM, FMH, FMC, FMHC, ISFM,
ISFMH, ISFMC, ISFMHC, ISFMCI, ISFMHCI, ISECI, ISMCI, ISEHCI, ISMHCI, WR,
WRH}, L1_Replacement) {
        zz_recycleMandatoryQueue;
    }

    transition({IM, IMH, IMF, IMFH, IMC, IMHC, IMFC, IMFHC, IMWX, IMWC, EI,
MI, FI, OI, ES, MS,
                FS, OS, WB, IS, ISH, ISF, ISFH, ISFM, ISFMH, ISC, ISHC, ISFC,
ISFHC, ISFMC, ISFMHC,
                ISWX, ISWC, ISCI, ISHCI, ISFCI, ISFHCI, ISFMCI, ISFMHCI,
ISECI, ISEHCI, ISMCI,
                ISMHCI, ISWXI, FM, FMH, FMC, FMHC, WR, WRH}, Fwd_Recycle) {
        rr_recycleRequestQueue;
    }

    transition({S, F, E, I}, L1_Replacement, I) {
        ff_deallocateL1CacheBlock;
    }

    // transition({M, O}, L1_Replacement, WB) {
    //     i_allocateTBE;
    //     g_issuePUTX;
    //     ff_deallocateL1CacheBlock;
    // }

    transition(O, L1_Replacement, WB) {
        i_allocateTBE;
        gf_issueL2FowardModified;
        ff_deallocateL1CacheBlock;
    }

    transition(M, L1_Replacement, WB) {
        i_allocateTBE;
        gm_issueL2Modified;
        ff_deallocateL1CacheBlock;
    }

    transition(WB, WB_Ack, I) {
        s_deallocateTBE;
        o_popIncomingResponseQueue;
    }

    transition(WB, {Store, Load, Ifetch}) {
        zz_recycleMandatoryQueue;
    }

    transition(WB, {Fwd_GETX, Fwd_GETS, Fwd_GET_INSTR}) {
        rr_recycleRequestQueue;
    }

    // TODO: This may be blocking other messages in the network from
completing
    // While waiting for a DACK, no other requests are served.
    transition({FI, EI, MI, OI, FS, ES, MS, OS, WR, WRH}, {Fwd_GETX,
Fwd_GETS, Fwd_GET_INSTR}) {
        rr_recycleRequestQueue;
    }

    // Loads and stores

    // Do not yet have permission to do the store/Transferring the line or
```

```
Invalidating
    transition({ISF, ISFH, ISFC, ISFHC, ISFCI, ISFHCI, OI, EI, MI, FI, OS,
ES, MS, FS, ISFM, ISFMH, ISFMC, ISFMHC, ISFMCI, ISFMHCI}, Store) {
        zz_recycleMandatoryQueue;
    }

    // Transferring the line or Invalidating
    transition({OI, EI, MI, FI}, {Load, Ifetch}) {
        zz_recycleMandatoryQueue;
    }

    transition(I, Load, IS) {
        oo_allocateL1DCacheBlock;
        i_allocateTBE;
        a_issueGETS;
        k_popMandatoryQueue;
    }

    transition(I, Ifetch, IS) {
        pp_allocateL1ICacheBlock;
        i_allocateTBE;
        ai_issueGETINSTR;
        k_popMandatoryQueue;
    }

    // States which are legal to read from, can just do the read.
    transition({S, F, E, M, O, IMF, IMFH, ISF, ISFH, IMFC, IMFHC, ISFC,
ISFHC, ISFCI, ISFHCI, FM, FMH, FMC, FMHC, ES, MS, FS, OS,
                ISFM, ISFMH, ISFMC, ISFMHC, ISFMCI, ISFMHCI, ISECI, ISMCI,
ISEHCI, ISMHCI, WR, WRH}, {Load, Ifetch}) {
        h_load_hit;
        k_popMandatoryQueue;
    }

    transition(I, Store, IM) {
        oo_allocateL1DCacheBlock;
        i_allocateTBE;
        b_issueGETX;
        k_popMandatoryQueue;
    }

    transition(S, Store, IM) {
        i_allocateTBE;
        b_issueGETX;
        k_popMandatoryQueue;
    }

    // States which are legal to write from, can just do the write.
    transition({M, IMF, IMFH, IMFC, IMFHC, FM, FMH, FMC, FMHC, ISMCI, ISMHCI,
WR, WRH}, Store) {
        hh_store_hit;
        k_popMandatoryQueue;
    }

    transition(E, Store, M) {
        hh_store_hit;
        k_popMandatoryQueue;
    }


    // F and O are equivalent here, as the Fwd_GetX will invalidate everyone
else, and this node is guaranteed to be the first one to reply to home - not
true!!
    /*transition({F, O}, Store, FM) {
        hh_store_hit;
        i_allocateTBE;
        b_issueGETX;
        k_popMandatoryQueue;
    }*/

    transition(F, Store, WR) {
        i_allocateTBE;
        b_issueGETX;
        hh_store_hit;
        k_popMandatoryQueue;
    }

    // transition(O, Store, WB) {
```

```
    //     i_allocateTBE;
    //     g_issuePUTX;
    // }

    transition(O, Store, WR) {
        i_allocateTBE;
        b_issueGETX;
        hh_store_hit;
        k_popMandatoryQueue;
    }



    // We got the line as exclusive because someother node sent a GetX, and
has not yet gotten the line (to modify it)
    transition(ISECI, Store, ISMCI) {
        hh_store_hit;
        k_popMandatoryQueue;
    }

    transition(ISEHCI, Store, ISMHCI) {
        hh_store_hit;
        k_popMandatoryQueue;
    }

    // Acknowledgement messages

    // IM
    transition(IM, IACK) {
        o_popIncomingResponseQueue;
    }

    transition(IMF, IACK) {
        o_popIncomingResponseQueue;
    }

    transition(IM, IACK_Final, IMH) {
        d_sendReadHome;
        o_popIncomingResponseQueue;
```

```
}

transition(IMF, IACK_Final, IMFH) {
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

transition(IM, SACK) {
    o_popIncomingResponseQueue;
}

transition(IMF, SACK) {
    o_popIncomingResponseQueue;
}

transition(IM, SACK_Final, IMH) {
    d_sendReadHome;
    o_popIncomingResponseQueue;
}

transition(IMF, SACK_Final, IMFH) {
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

// IS

transition({IS, ISF, ISFM}, IACK) {
    o_popIncomingResponseQueue;
}

transition(IS, IACK_Final, ISH) {
    d_sendReadHome;
    o_popIncomingResponseQueue;
}

transition(ISF, IACK_Final, ISFH) {
    c_sendCnclHome;
    o_popIncomingResponseQueue;
```

```
}

transition(ISFM, IACK_Final, ISFMH) {
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

transition({IS, ISF, ISFM}, SACK) {
    o_popIncomingResponseQueue;
}

transition(IS, SACK_Final, ISH) {
    d_sendReadHome;
    o_popIncomingResponseQueue;
}

transition(ISF, SACK_Final, ISFH) {
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

transition(ISFM, SACK_Final, ISFMH) {
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}


// FM

transition(FM, IACK) {
    o_popIncomingResponseQueue;
}

transition(FM, SACK) {
    o_popIncomingResponseQueue;
}

transition(FM, IACK_Final, FMH) {
    c_sendCnclHome;
```

```
        o_popIncomingResponseQueue;
    }

    transition(FM, SACK_Final, FMH) {
        c_sendCnclHome;
        o_popIncomingResponseQueue;
    }

    // Forwarded data from L1/ cache

    transition(IM, {DataE, DataM}, IMF) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        hh_store_hit;
        od_popIncomingDataResponseQueue;
    }

    transition(IM, {DataE_Final, DataM_Final}, IMFH) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        c_sendCnclHome;
        hh_store_hit;
        od_popIncomingDataResponseQueue;
    }

    // These transitions occur when there is a forward state present.
    // May need to distinguish this state as one that cannot yet be written
to
    transition(IM, {DataF, DataFM}, IMF) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        hh_store_hit;
        od_popIncomingDataResponseQueue;
    }

    // May need to distinguish this state as one that cannot yet be written
to
    transition(IM, {DataF_Final, DataFM_Final}, IMFH) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        c_sendCnclHome;
        hh_store_hit;
        od_popIncomingDataResponseQueue;
    }

    transition(IS, DataF, ISF) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        h_load_hit;
        od_popIncomingDataResponseQueue;
    }

    transition(IS, {DataFM, DataM}, ISFM) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        h_load_hit;
        od_popIncomingDataResponseQueue;
    }

    transition(IS, DataF_Final, ISFH) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        c_sendCnclHome;
        h_load_hit;
        od_popIncomingDataResponseQueue;
    }

    transition(IS, {DataFM_Final, DataM_Final}, ISFMH) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        c_sendCnclHome;
        h_load_hit;
        od_popIncomingDataResponseQueue;
    }


    // Response from home
```

```
transition(IMH, DataE_Home, M) {
    uu_writeHomeDataToL1Cache;
    hh_store_hit;
    s_deallocateTBE;
    o_popIncomingResponseQueue;
}

transition(IMFH, Ack_Home, M) {
    dd_sendDACKToRequestor;
    //hh_store_hit;
    s_deallocateTBE;
    o_popIncomingResponseQueue;
}

transition(ISH, DataE_Home, E) {
    uu_writeHomeDataToL1Cache;
    h_load_hit;
    s_deallocateTBE;
    o_popIncomingResponseQueue;
}

transition(ISFH, Ack_Home, F) {
    dd_sendDACKToRequestor;
    s_deallocateTBE;
    o_popIncomingResponseQueue;
}

transition(ISFMH, Ack_Home, O) {
    dd_sendDACKToRequestor;
    s_deallocateTBE;
    o_popIncomingResponseQueue;
}

transition(FMH, Ack_Home, M) {
    s_deallocateTBE;
    o_popIncomingResponseQueue;
}
```

```
// Requests from another cache

transition(I, {Fwd_GETX, Fwd_GETS, Fwd_GET_INSTR}) {
    di_sendIACKToRequestor;
    l_popRequestQueue;
}

transition(S, Fwd_GETX, I) {
    ds_sendSACKToRequestor;
    l_popRequestQueue;
}

transition(S, {Fwd_GETS, Fwd_GET_INSTR}) {
    ds_sendSACKToRequestor;
    l_popRequestQueue;
}

transition(F, Fwd_GETX, FI) {
    df_sendForwardDataToRequestor;
    l_popRequestQueue;
}

transition(F, {Fwd_GETS, Fwd_GET_INSTR}, FS) {
    df_sendForwardDataToRequestor;
    l_popRequestQueue;
}

transition(O, Fwd_GETX, OI) {
    dfm_sendForwardModifiedDataToRequestor;
    l_popRequestQueue;
}

transition(O, {Fwd_GETS, Fwd_GET_INSTR}, OS) {
    dfm_sendForwardModifiedDataToRequestor;
    l_popRequestQueue;
}

transition(E, Fwd_GETX, EI) {
    de_sendExclusiveDataToRequestor;
    l_popRequestQueue;
}

transition(E, {Fwd_GETS, Fwd_GET_INSTR}, ES) {
    df_sendForwardDataToRequestor;
    l_popRequestQueue;
}

transition(M, {Fwd_GETS, Fwd_GET_INSTR}, MS) {
    dfm_sendForwardModifiedDataToRequestor;
    l_popRequestQueue;
}

transition(M, Fwd_GETX, MI) {
    dm_sendModifiedDataToRequestor;
    l_popRequestQueue;
}

// Dack transitions

transition({MS, ES, FS, OS}, DACK, S) {
    s_deallocateTBE;
    od_popIncomingDataResponseQueue;
}

transition({MI, EI, FI, OI}, DACK, I) {
    s_deallocateTBE;
    od_popIncomingDataResponseQueue;
}

// Conflict transitions - GetX

// Recieving conflicting requests in non-conflicting state

transition(IM, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}, IMC) {
    cr_recordConflictMachineRequest;
    sc_sendConflict;
    l_popRequestQueue;
}
```

```
transition(IM, Conflict, IMC) {
    cc_recordConflictMachineResponse;
    o_popIncomingResponseQueue;
}

transition(IM, Conflict_Final, IMHC) {
    cc_recordConflictMachineResponse;
    d_sendReadHome;
    o_popIncomingResponseQueue;
}

transition(IMF, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}, IMFC) {
    cr_recordConflictMachineRequest;
    sc_sendConflict;
    l_popRequestQueue;
}

transition(IMF, Conflict, IMFC) {
    cc_recordConflictMachineResponse;
    o_popIncomingResponseQueue;
}

transition(IMF, Conflict_Final, IMFHC) {
    cc_recordConflictMachineResponse;
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

/*transition(IMH, Data_Transfer, MI) {
    u_writeDataToL1Cache;
    v_recordForwarder;
    hh_store_hit;
    ct_recordTransferMachine;
    tm_transferModified;
    o_popIncomingResponseQueue;
}*/

/*transition(IMH, Wait, IMWC) {
    o_popIncomingResponseQueue;
}*/

/*transition(IMH, Wait_Transfer, IMWX) {
    ct_recordTransferMachine;
    o_popIncomingResponseQueue;
}*/

transition(IMH, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    rr_recycleRequestQueue;
}

transition(IMFH, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    rr_recycleRequestQueue;
}

// Transitions when conflicting states exist

transition(IMC, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    cr_recordConflictMachineRequest;
    sc_sendConflict;
    l_popRequestQueue;
}

transition(IMC, {IACK, SACK}) {
    o_popIncomingResponseQueue;
}

transition(IMC, Conflict) {
    cc_recordConflictMachineResponse;
    o_popIncomingResponseQueue;
}

transition(IMC, {IACK_Final, SACK_Final}, IMHC) {
    d_sendReadHome;
    o_popIncomingResponseQueue;
}

transition(IMC, Conflict_Final, IMHC) {
        cc_recordConflictMachineResponse;
        d_sendReadHome;
        o_popIncomingResponseQueue;
    }

    transition(IMC, {DataF, DataE, DataM, DataFM}, IMFC) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        hh_store_hit;
        od_popIncomingDataResponseQueue;
    }

    transition(IMC, {DataF_Final, DataE_Final, DataM_Final, DataFM_Final},
IMFHC) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        hh_store_hit;
        c_sendCnclHome;
        od_popIncomingDataResponseQueue;
    }

    transition(IMFC, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
        rr_recycleRequestQueue;
    }

    transition(IMFC, {IACK, SACK}) {
        o_popIncomingResponseQueue;
    }

    transition(IMFC, Conflict) {
        cc_recordConflictMachineResponse;
        o_popIncomingResponseQueue;
    }

    transition(IMFC, {IACK_Final, SACK_Final}, IMFHC) {
        c_sendCnclHome;
        o_popIncomingResponseQueue;
    }
```

```
transition(IMFC, Conflict_Final, IMFHC) {
    cc_recordConflictMachineResponse;
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

transition(IMFHC, Ack_Home, M) {
    dd_sendDACKToRequestor;
    s_deallocateTBE;
    o_popIncomingResponseQueue;
}

transition(IMFHC, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    rr_recycleRequestQueue;
}

transition(IMFHC, Transfer, MI) {
    dd_sendDACKToRequestor;
    ct_recordTransferMachine;
    tm_transferModified;
    o_popIncomingResponseQueue;
}

transition(IMFHC, Wait, M) {
    dd_sendDACKToRequestor;
    s_deallocateTBE;
    o_popIncomingResponseQueue;
}

transition(IMFHC, Wait_Transfer, MI) {
    dd_sendDACKToRequestor;
    ct_recordTransferMachine;
    tm_transferModified;
    o_popIncomingResponseQueue;
}

transition(IMHC, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    rr_recycleRequestQueue;
}
```

```
    transition(IMHC, {DataF_Final, DataE_Final, DataM_Final, DataFM_Final},
IMFHC) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        hh_store_hit;
        od_popIncomingDataResponseQueue;
    }

    transition(IMHC, Data_Transfer, MI) {
        ru_responseWriteDataToL1Cache;
        //v_responseRecordForwarder;
        hh_store_hit;
        ct_recordTransferMachine;
        tm_transferModified;
        o_popIncomingResponseQueue;
    }

    transition(IMHC, Wait, IMWC) {
        o_popIncomingResponseQueue;
    }

    transition(IMHC, Wait_Transfer, IMWX) {
        ct_recordTransferMachine;
        o_popIncomingResponseQueue;
    }

    // Wont be DataF since we are going to exclusive
    transition(IMWC, {DataE_Final, DataM_Final, DataFM_Final, DataF_Final},
M) {
        v_recordForwarder;
        dd_sendDACKToRequestor;
        u_writeDataToL1Cache;
        hh_store_hit;
        s_deallocateTBE;
        od_popIncomingDataResponseQueue;
    }

    transition(IMWC, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
        rr_recycleRequestQueue;
    }

    // Wont be DataF since we are going to exclusive
    transition(IMWX, {DataE_Final, DataM_Final, DataFM_Final, DataF_Final},
MI) {
        v_recordForwarder;
        dd_sendDACKToRequestor;
        u_writeDataToL1Cache;
        hh_store_hit;
        tm_transferModified;
        od_popIncomingDataResponseQueue;
    }

    transition(IMWX, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
        rr_recycleRequestQueue;
    }

    // Conflicts - GetS

    // Recieving conflicting requests in non-conflicting state

    transition(IS, {Fwd_GETS, Fwd_GET_INSTR}, ISC) {
        cr_recordConflictMachineRequest;
        sc_sendConflict;
        l_popRequestQueue;
    }

    transition(IS, Fwd_GETX, ISCI) {
        cr_recordConflictMachineRequest;
        sc_sendConflict;
        l_popRequestQueue;
    }

    transition(IS, Conflict, ISC) {
        cc_recordConflictMachineResponse;
        o_popIncomingResponseQueue;
    }
```

```
transition(IS, Conflict_Final, ISHC) {
    cc_recordConflictMachineResponse;
    d_sendReadHome;
    o_popIncomingResponseQueue;
}

transition(ISF, {Fwd_GETS, Fwd_GET_INSTR}, ISFC) {
    cr_recordConflictMachineRequest;
    sc_sendConflict;
    l_popRequestQueue;
}

transition(ISFM, {Fwd_GETS, Fwd_GET_INSTR}, ISFMC) {
    cr_recordConflictMachineRequest;
    sc_sendConflict;
    l_popRequestQueue;
}

transition(ISF, Fwd_GETX, ISFCI) {
    cr_recordConflictMachineRequest;
    sc_sendConflict;
    l_popRequestQueue;
}

transition(ISFM, Fwd_GETX, ISFMCI) {
    cr_recordConflictMachineRequest;
    sc_sendConflict;
    l_popRequestQueue;
}

transition(ISF, Conflict, ISFC) {
    cc_recordConflictMachineResponse;
    o_popIncomingResponseQueue;
}

transition(ISFM, Conflict, ISFMC) {
    cc_recordConflictMachineResponse;
    o_popIncomingResponseQueue;
}
```

```
transition(ISF, Conflict_Final, ISFHC) {
    cc_recordConflictMachineResponse;
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

transition(ISFM, Conflict_Final, ISFMHC) {
    cc_recordConflictMachineResponse;
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

/*transition(ISH, Data_Transfer, FI) {
    u_writeDataToL1Cache;
    v_recordForwarder;
    h_load_hit;
    ct_recordTransferMachine;
    tf_transferForward;
    o_popIncomingResponseQueue;
}*/

/*transition(ISH, Wait, ISWC) {
    o_popIncomingResponseQueue;
}*/

/*transition(ISH, Wait_Transfer, ISWX) {
    ct_recordTransferMachine;
    o_popIncomingResponseQueue;
}*/

transition(ISH, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    rr_recycleRequestQueue;
}

transition(ISFH, Transfer, FS) {
    dd_sendDACKToRequestor;
    ct_recordTransferMachine;
    tf_transferForward;
```

```
        o_popIncomingResponseQueue;
    }

    transition(ISFMH, Transfer, OS) {
        dd_sendDACKToRequestor;
        ct_recordTransferMachine;
        tfm_transferForwardModified;
        o_popIncomingResponseQueue;
    }

    transition({ISFH, ISFMH}, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
        rr_recycleRequestQueue;
    }

    // Transitions when conflicting states exist

    transition(ISC, {Fwd_GETS, Fwd_GET_INSTR}) {
        cr_recordConflictMachineRequest;
        sc_sendConflict;
        l_popRequestQueue;
    }

    transition(ISC, Fwd_GETX, ISCI) {
        cr_recordConflictMachineRequest;
        sc_sendConflict;
        l_popRequestQueue;
    }

    transition(ISC, {IACK, SACK}) {
        o_popIncomingResponseQueue;
    }

    transition(ISC, Conflict) {
        cc_recordConflictMachineResponse;
        o_popIncomingResponseQueue;
    }

    transition(ISC, {IACK_Final, SACK_Final}, ISHC) {
        d_sendReadHome;
        o_popIncomingResponseQueue;
    }

    transition(ISC, Conflict_Final, ISHC) {
        cc_recordConflictMachineResponse;
        d_sendReadHome;
        o_popIncomingResponseQueue;
    }

    transition(ISC, DataF, ISFC) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        h_load_hit;
        od_popIncomingDataResponseQueue;
    }

    transition(ISC, {DataFM, DataM}, ISFMC) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        h_load_hit;
        od_popIncomingDataResponseQueue;
    }

    transition(ISC, DataF_Final, ISFHC) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        h_load_hit;
        c_sendCnclHome;
        od_popIncomingDataResponseQueue;
    }

    transition(ISC, {DataFM_Final, DataM_Final}, ISFMHC) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        h_load_hit;
        c_sendCnclHome;
        od_popIncomingDataResponseQueue;
    }
```

```
transition({ISFC, ISFMC}, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    rr_recycleRequestQueue;
}

transition({ISFC, ISFMC}, {IACK, SACK}) {
    o_popIncomingResponseQueue;
}

transition({ISFC, ISFMC}, Conflict) {
    cc_recordConflictMachineResponse;
    o_popIncomingResponseQueue;
}

transition(ISFC, {IACK_Final, SACK_Final}, ISFHC) {
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

transition(ISFMC, {IACK_Final, SACK_Final}, ISFMHC) {
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

transition(ISFC, Conflict_Final, ISFHC) {
    cc_recordConflictMachineResponse;
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

transition(ISFMC, Conflict_Final, ISFMHC) {
    cc_recordConflictMachineResponse;
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

transition(ISFHC, Ack_Home, F) {
    dd_sendDACKToRequestor;
    s_deallocateTBE;
    o_popIncomingResponseQueue;
```

```
}

transition(ISFMHC, Ack_Home, O) {
    dd_sendDACKToRequestor;
    s_deallocateTBE;
    o_popIncomingResponseQueue;
}

transition({ISFHC, ISFMHC}, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    rr_recycleRequestQueue;
}

transition(ISFHC, Transfer, FS) {
    dd_sendDACKToRequestor;
    ct_recordTransferMachine;
    tf_transferForward;
    o_popIncomingResponseQueue;
}

transition(ISFMHC, Transfer, OS) {
    dd_sendDACKToRequestor;
    ct_recordTransferMachine;
    tfm_transferForwardModified;
    o_popIncomingResponseQueue;
}

transition(ISFHC, Wait, F) {
    dd_sendDACKToRequestor;
    s_deallocateTBE;
    o_popIncomingResponseQueue;
}

transition(ISFMHC, Wait, O) {
    dd_sendDACKToRequestor;
    s_deallocateTBE;
    o_popIncomingResponseQueue;
}

transition(ISFHC, Wait_Transfer, FS) {
    dd_sendDACKToRequestor;
    ct_recordTransferMachine;
    tf_transferForward;
    o_popIncomingResponseQueue;
}

transition(ISFMHC, Wait_Transfer, OS) {
    dd_sendDACKToRequestor;
    ct_recordTransferMachine;
    tfm_transferForwardModified;
    o_popIncomingResponseQueue;
}

transition(ISHC, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    rr_recycleRequestQueue;
}

transition(ISHC, DataF_Final, ISFHC) {
    u_writeDataToL1Cache;
    v_recordForwarder;
    h_load_hit;
    od_popIncomingDataResponseQueue;
}

transition(ISHC, {DataFM_Final, DataM_Final}, ISFMHC) {
    u_writeDataToL1Cache;
    v_recordForwarder;
    h_load_hit;
    od_popIncomingDataResponseQueue;
}

transition(ISHC, Data_Transfer, FS) {
    ru_responseWriteDataToL1Cache;
    //v_responseRecordForwarder;
    h_load_hit;
    ct_recordTransferMachine;
    tf_transferForward;
    o_popIncomingResponseQueue;
}
```

```
transition(ISHC, Wait, ISWC) {
    o_popIncomingResponseQueue;
}

transition(ISHC, Wait_Transfer, ISWX) {
    ct_recordTransferMachine;
    o_popIncomingResponseQueue;
}


transition(ISWC, DataF_Final, F) {
    v_recordForwarder;
    dd_sendDACKToRequestor;
    u_writeDataToL1Cache;
    h_load_hit;
    s_deallocateTBE;
    od_popIncomingDataResponseQueue;
}

transition(ISWC, DataFM_Final, O) {
    v_recordForwarder;
    dd_sendDACKToRequestor;
    u_writeDataToL1Cache;
    h_load_hit;
    s_deallocateTBE;
    od_popIncomingDataResponseQueue;
}

transition(ISWC, DataE_Final, E) {
    v_recordForwarder;
    dd_sendDACKToRequestor;
    u_writeDataToL1Cache;
    h_load_hit;
    s_deallocateTBE;
    od_popIncomingDataResponseQueue;
}

transition(ISWC, DataM_Final, M) {
    v_recordForwarder;
    dd_sendDACKToRequestor;
    u_writeDataToL1Cache;
    h_load_hit;
    s_deallocateTBE;
    od_popIncomingDataResponseQueue;
}

transition(ISWC, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    rr_recycleRequestQueue;
}

transition(ISWX, DataF_Final, FS) {
    v_recordForwarder;
    dd_sendDACKToRequestor;
    u_writeDataToL1Cache;
    h_load_hit;
    tf_transferForward;
    od_popIncomingDataResponseQueue;
}

transition(ISWX, DataFM_Final, OS) {
    v_recordForwarder;
    dd_sendDACKToRequestor;
    u_writeDataToL1Cache;
    h_load_hit;
    tfm_transferForwardModified;
    od_popIncomingDataResponseQueue;
}

transition(ISWX, DataE_Final, ES) {
    v_recordForwarder;
    dd_sendDACKToRequestor;
    u_writeDataToL1Cache;
    h_load_hit;
    tf_transferForward;
    od_popIncomingDataResponseQueue;
}
```

```
transition(ISWX, DataM_Final, MS) {
    v_recordForwarder;
    dd_sendDACKToRequestor;
    u_writeDataToL1Cache;
    h_load_hit;
    tfm_transferForwardModified;
    od_popIncomingDataResponseQueue;
}

transition(ISWX, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    rr_recycleRequestQueue;
}

// Conflicts S-M but invalidating

// Transitions when conflicting states exist

transition(ISCI, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    cr_recordConflictMachineRequest;
    sc_sendConflict;
    l_popRequestQueue;
}
transition(ISCI, {IACK, SACK}) {
    o_popIncomingResponseQueue;
}

transition(ISCI, Conflict) {
    cc_recordConflictMachineResponse;
    o_popIncomingResponseQueue;
}

transition(ISCI, {IACK_Final, SACK_Final}, ISHCI) {
    d_sendReadHome;
    o_popIncomingResponseQueue;
}

transition(ISCI, Conflict_Final, ISHCI) {
    cc_recordConflictMachineResponse;
    d_sendReadHome;
    o_popIncomingResponseQueue;
}

transition(ISCI, DataF, ISFCI) {
    u_writeDataToL1Cache;
    v_recordForwarder;
    h_load_hit;
    od_popIncomingDataResponseQueue;
}

transition(ISCI, DataFM, ISFMCI) {
    u_writeDataToL1Cache;
    v_recordForwarder;
    h_load_hit;
    od_popIncomingDataResponseQueue;
}

transition(ISCI, DataE, ISECI) {
    u_writeDataToL1Cache;
    v_recordForwarder;
    h_load_hit;
    od_popIncomingDataResponseQueue;
}

transition(ISCI, DataM, ISMCI) {
    u_writeDataToL1Cache;
    v_recordForwarder;
    h_load_hit;
    od_popIncomingDataResponseQueue;
}

transition(ISCI, DataF_Final, ISFHCI) {
    u_writeDataToL1Cache;
    v_recordForwarder;
    h_load_hit;
    c_sendCnclHome;
    od_popIncomingDataResponseQueue;
}
```

```
    transition(ISCI, DataFM_Final, ISFMHCI) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        h_load_hit;
        c_sendCnclHome;
        od_popIncomingDataResponseQueue;
    }

    transition(ISCI, DataE_Final, ISEHCI) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        h_load_hit;
        c_sendCnclHome;
        od_popIncomingDataResponseQueue;
    }

    transition(ISCI, DataM_Final, ISMHCI) {
        u_writeDataToL1Cache;
        v_recordForwarder;
        h_load_hit;
        c_sendCnclHome;
        od_popIncomingDataResponseQueue;
    }

    transition({ISFCI, ISFMCI, ISECI, ISMCI}, {Fwd_GETS, Fwd_GET_INSTR,
Fwd_GETX}) {
        rr_recycleRequestQueue;
    }

    transition({ISFCI, ISFMCI, ISECI, ISMCI}, {IACK, SACK}) {
        o_popIncomingResponseQueue;
    }

    transition({ISFCI, ISFMCI, ISECI, ISMCI}, Conflict) {
        cc_recordConflictMachineResponse;
        o_popIncomingResponseQueue;
    }

    transition(ISFCI, {IACK_Final, SACK_Final}, ISFHCI) {
        c_sendCnclHome;
        o_popIncomingResponseQueue;
    }

    transition(ISFMCI, {IACK_Final, SACK_Final}, ISFMHCI) {
        c_sendCnclHome;
        o_popIncomingResponseQueue;
    }

    transition(ISECI, {IACK_Final, SACK_Final}, ISEHCI) {
        c_sendCnclHome;
        o_popIncomingResponseQueue;
    }

    transition(ISMCI, {IACK_Final, SACK_Final}, ISMHCI) {
        c_sendCnclHome;
        o_popIncomingResponseQueue;
    }

    transition(ISFCI, Conflict_Final, ISFHCI) {
        cc_recordConflictMachineResponse;
        c_sendCnclHome;
        o_popIncomingResponseQueue;
    }

    transition(ISFMCI, Conflict_Final, ISFMHCI) {
        cc_recordConflictMachineResponse;
        c_sendCnclHome;
        o_popIncomingResponseQueue;
    }

    transition(ISECI, Conflict_Final, ISEHCI) {
        cc_recordConflictMachineResponse;
        c_sendCnclHome;
        o_popIncomingResponseQueue;
    }

    transition(ISMCI, Conflict_Final, ISMHCI) {
        cc_recordConflictMachineResponse;
        c_sendCnclHome;
        o_popIncomingResponseQueue;
    }

    // The Fwd_GetX that sent us into an IS...I state has been dealt with
already
    transition(ISFHCI, Ack_Home, F) {
        dd_sendDACKToRequestor;
        s_deallocateTBE;
        o_popIncomingResponseQueue;
    }

    transition(ISFMHCI, Ack_Home, O) {
        dd_sendDACKToRequestor;
        s_deallocateTBE;
        o_popIncomingResponseQueue;
    }

    transition(ISEHCI, Ack_Home, E) {
        dd_sendDACKToRequestor;
        s_deallocateTBE;
        o_popIncomingResponseQueue;
    }

    transition(ISMHCI, Ack_Home, M) {
        dd_sendDACKToRequestor;
        s_deallocateTBE;
        o_popIncomingResponseQueue;
    }

    transition({ISFHCI, ISFMHCI, ISEHCI, ISMHCI}, {Fwd_GETS, Fwd_GET_INSTR,
Fwd_GETX}) {
        rr_recycleRequestQueue;
    }

    transition(ISFHCI, Transfer, FI) {
        dd_sendDACKToRequestor;
        ct_recordTransferMachine;
        tf_transferForward;
        o_popIncomingResponseQueue;
    }

    transition(ISFMHCI, Transfer, OI) {
        dd_sendDACKToRequestor;
        ct_recordTransferMachine;
        tfm_transferForwardModified;
        o_popIncomingResponseQueue;
    }

    transition(ISEHCI, Transfer, EI) {
        dd_sendDACKToRequestor;
        ct_recordTransferMachine;
        te_transferExclusive;
        o_popIncomingResponseQueue;
    }

    transition(ISMHCI, Transfer, MI) {
        dd_sendDACKToRequestor;
        ct_recordTransferMachine;
        tm_transferModified;
        o_popIncomingResponseQueue;
    }

    // The Fwd_GetX that sent us into an IS...I state has been dealt with
already
    transition(ISFHCI, Wait, F) {
        dd_sendDACKToRequestor;
        s_deallocateTBE;
        o_popIncomingResponseQueue;
    }

    transition(ISFMHCI, Wait, O) {
        dd_sendDACKToRequestor;
        s_deallocateTBE;
        o_popIncomingResponseQueue;
    }

    transition(ISEHCI, Wait, E) {
```

```
    dd_sendDACKToRequestor;
    s_deallocateTBE;
    o_popIncomingResponseQueue;
}

transition(ISMHCI, Wait, M) {
    dd_sendDACKToRequestor;
    s_deallocateTBE;
    o_popIncomingResponseQueue;
}

transition(ISFHCI, Wait_Transfer, FI) {
    dd_sendDACKToRequestor;
    ct_recordTransferMachine;
    tf_transferForward;
    o_popIncomingResponseQueue;
}

transition(ISFMHCI, Wait_Transfer, OI) {
    dd_sendDACKToRequestor;
    ct_recordTransferMachine;
    tfm_transferForwardModified;
    o_popIncomingResponseQueue;
}

transition(ISEHCI, Wait_Transfer, EI) {
    dd_sendDACKToRequestor;
    ct_recordTransferMachine;
    te_transferExclusive;
    o_popIncomingResponseQueue;
}

transition(ISMHCI, Wait_Transfer, MI) {
    dd_sendDACKToRequestor;
    ct_recordTransferMachine;
    tm_transferModified;
    o_popIncomingResponseQueue;
}
```

```
transition(ISHCI, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    rr_recycleRequestQueue;
}

transition(ISHCI, DataF_Final, ISFHCI) {
    u_writeDataToL1Cache;
    v_recordForwarder;
    h_load_hit;
    od_popIncomingDataResponseQueue;
}

transition(ISHCI, DataFM_Final, ISFMHCI) {
    u_writeDataToL1Cache;
    v_recordForwarder;
    h_load_hit;
    od_popIncomingDataResponseQueue;
}

transition(ISHCI, DataE_Final, ISEHCI) {
    u_writeDataToL1Cache;
    v_recordForwarder;
    h_load_hit;
    od_popIncomingDataResponseQueue;
}

transition(ISHCI, DataM_Final, ISMHCI) {
    u_writeDataToL1Cache;
    v_recordForwarder;
    h_load_hit;
    od_popIncomingDataResponseQueue;
}

transition(ISHCI, Data_Transfer, FI) {
    ru_responseWriteDataToL1Cache;
    //v_responseRecordForwarder;
    h_load_hit;
    ct_recordTransferMachine;
    tf_transferForward;
    o_popIncomingResponseQueue;
}

// Transfer out of IS...I states due to Wait response
transition(ISHCI, Wait, ISWC) {
    o_popIncomingResponseQueue;
}

transition(ISHCI, Wait_Transfer, ISWXI) {
    ct_recordTransferMachine;
    o_popIncomingResponseQueue;
}

transition(ISWXI, DataF_Final, FI) {
    v_recordForwarder;
    dd_sendDACKToRequestor;
    u_writeDataToL1Cache;
    h_load_hit;
    tf_transferForward;
    od_popIncomingDataResponseQueue;
}

transition(ISWXI, DataFM_Final, OI) {
    v_recordForwarder;
    dd_sendDACKToRequestor;
    u_writeDataToL1Cache;
    h_load_hit;
    tfm_transferForwardModified;
    od_popIncomingDataResponseQueue;
}

transition(ISWXI, DataE_Final, EI) {
    v_recordForwarder;
    dd_sendDACKToRequestor;
    u_writeDataToL1Cache;
    h_load_hit;
    te_transferExclusive;
    od_popIncomingDataResponseQueue;
}
```

```
transition(ISWXI, DataM_Final, MI) {
    v_recordForwarder;
    dd_sendDACKToRequestor;
    u_writeDataToL1Cache;
    h_load_hit;
    tm_transferModified;
    od_popIncomingDataResponseQueue;
}

transition(ISWXI, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    rr_recycleRequestQueue;
}

// Conflicts F-M

// Recieving conflicting requests in non-conflicting state

transition(FM, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}, FMC) {
    cr_recordConflictMachineRequest;
    sc_sendConflict;
    l_popRequestQueue;
}

transition(FM, Conflict, FMC) {
    cc_recordConflictMachineResponse;
    o_popIncomingResponseQueue;
}

transition(FM, Conflict_Final, FMHC) {
    cc_recordConflictMachineResponse;
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

transition(FMH, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    rr_recycleRequestQueue;
}

// Transitions when conflicting states exist
```

```
transition(FMC, IACK) {
    o_popIncomingResponseQueue;
}

transition(FMC, SACK) {
    o_popIncomingResponseQueue;
}

transition(FMC, IACK_Final, FMHC) {
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

transition(FMC, SACK_Final, FMHC) {
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

transition(FMC, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    cr_recordConflictMachineRequest;
    sc_sendConflict;
    l_popRequestQueue;
}

transition(FMC, Conflict) {
    cc_recordConflictMachineResponse;
    o_popIncomingResponseQueue;
}

transition(FMC, Conflict_Final, FMHC) {
    cc_recordConflictMachineResponse;
    c_sendCnclHome;
    o_popIncomingResponseQueue;
}

transition(FMHC, {Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX}) {
    rr_recycleRequestQueue;
}
```

```
    transition(FMHC, Ack_Home, M) {
        s_deallocateTBE;
        //s_deallocateTBE;
        o_popIncomingResponseQueue;
    }

    transition(FMHC, Transfer, MI) {
        ct_recordTransferMachine;
        tm_transferModified;
        o_popIncomingResponseQueue;
    }

    // IS forwarded FM states




    // Extra requests! These are GetS/X requests from a L1 cache which this
cache is already transferring too
    // This prevents the request from being recycled until we recieve a DACK,
then being treated as a new request.

    /*transition({MI, EI, FI, OI, MS, ES, FS, OS, IMWX, ISWX, ISWXI},
Extra_Request) {
        l_popRequestQueue;
    }*/
    // TODO: if differentiating between I and S ACK, need to change this
transition
    transition({MI, EI, FI, OI, MS, ES, FS, OS, IMWX, IMWC, ISWX, ISWC,
ISWXI}, {Extra_Fwd_GETS, Extra_Fwd_GET_INSTR}) {
        di_sendIACKToRequestor;
        l_popRequestQueue;
    }

    transition({MI, EI, FI, OI, IMWX, ISWXI}, Extra_Fwd_GETX) {
        di_sendIACKToRequestor;
        l_popRequestQueue;
    }
```

```
transition(MS, Extra_Fwd_GETX, MI) {
    di_sendIACKToRequestor;
    l_popRequestQueue;
}

transition(ES, Extra_Fwd_GETX, EI) {
    di_sendIACKToRequestor;
    l_popRequestQueue;
}

transition(FS, Extra_Fwd_GETX, FI) {
    di_sendIACKToRequestor;
    l_popRequestQueue;
}

transition(OS, Extra_Fwd_GETX, OI) {
    di_sendIACKToRequestor;
    l_popRequestQueue;
}

transition(ISWX, Extra_Fwd_GETX, ISWXI) {
    di_sendIACKToRequestor;
    l_popRequestQueue;
}

transition(ISWC, Extra_Fwd_GETX, ISWXI) {
    di_sendIACKToRequestor;
    l_popRequestQueue;
}

transition(IMWC, Extra_Fwd_GETX, IMWX) {
    di_sendIACKToRequestor;
    l_popRequestQueue;
}


// WR, WRH
```

```
    // TODO: if responding to conflicts, must respond to requests?

    transition(WR, {IACK, SACK}){
        o_popIncomingResponseQueue;
    }

    transition(WR, {IACK_Final, SACK_Final}, WRH){
        c_sendCnclHome;
        o_popIncomingResponseQueue;
    }

    transition(WR, Conflict){
        cc_recordConflictMachineResponse;
        o_popIncomingResponseQueue;
    }

    transition(WR, Conflict_Final, WRH){
        cc_recordConflictMachineResponse;
        c_sendCnclHome;
        o_popIncomingResponseQueue;
    }


    transition(WRH, Transfer, MI){
        ct_recordTransferMachine;
        tm_transferModified;
        o_popIncomingResponseQueue;
    }


    transition(WRH, Ack_Home, M){
        s_deallocateTBE;
        o_popIncomingResponseQueue;
    }

}
```

Appendix B

Home Node and Shared Cache Controller SLICC Code

```
/*
    Copyright (C) 1999-2005 by Mark D. Hill and David A. Wood for the
    Wisconsin Multifacet Project.  Contact: gems@cs.wisc.edu
    http://www.cs.wisc.edu/gems/

    --------------------------------------------------------------------

    This file is part of the SLICC (Specification Language for
    Implementing Cache Coherence), a component of the Multifacet GEMS
    (General Execution-driven Multiprocessor Simulator) software
    toolset originally developed at the University of Wisconsin-Madison.


    SLICC was originally developed by Milo Martin with substantial
    contributions from Daniel Sorin.

    Substantial further development of Multifacet GEMS at the
    University of Wisconsin was performed by Alaa Alameldeen, Brad
    Beckmann, Jayaram Bobba, Ross Dickson, Dan Gibson, Pacia Harper,
    Derek Hower, Milo Martin, Michael Marty, Carl Mauer, Michelle Moravan,
    Kevin Moore, Manoj Plakal, Daniel Sorin, Haris Volos, Min Xu, and Luke
Yen.

    --------------------------------------------------------------------
    If your use of this software contributes to a published paper, we
    request that you (1) cite our summary paper that appears on our
    website (http://www.cs.wisc.edu/gems/) and (2) e-mail a citation
    for your published paper to gems@cs.wisc.edu.

    If you redistribute derivatives of this software, we request that
    you notify us and either (1) ask people to register with us at our
    website (http://www.cs.wisc.edu/gems/) or (2) collect registration
    information and periodically send it to us.

    --------------------------------------------------------------------

    Multifacet GEMS is free software; you can redistribute it and/or
    modify it under the terms of version 2 of the GNU General Public
    License as published by the Free Software Foundation.

    Multifacet GEMS is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with the Multifacet GEMS; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
    02111-1307, USA

    The GNU General Public License is contained in the file LICENSE.

### END HEADER ###
*/
/*
 * $Id: MOESI_CMP_token-dir.sm 1.6 05/01/19 15:48:35-06:00
mikem@royal16.cs.wisc.edu $
 */

/**

Modified by Andrew Hay (andrewh@cs.auckland.ac.nz), 2011
```

```
*/

machine(L2Cache, "MESIF protocol") {

    // Message buffers: this node TO the network
    MessageBuffer requestFromL2Cache, network="To", virtual_network="2",
ordered="false";  // this L2 bank -> Memory
    MessageBuffer responseFromL2Cache, network="To", virtual_network="1",
ordered="false";  // this L2 bank -> a local L1
    MessageBuffer dataResponseFromL2Cache, network="To", virtual_network="4",
ordered="false"; // cheating, simplifies things

    // Message buffers: this node FROM the network
    MessageBuffer L1RequestToL2Cache, network="From", virtual_network="0",
ordered="false";  // a local L1 -> this L2 bank
    MessageBuffer responseToL2Cache, network="From", virtual_network="3",
ordered="false";  // Memory -> this L2 bank
    MessageBuffer dataResponseToL2Cache, network="From", virtual_network="4",
ordered="false"; // cheating, simplifies things

    // STATES
    enumeration(State, desc="Cache states", default="L2Cache_State_I") {
        // Base states
        //NP,     "NP",       desc="Not Present"; // Base Not present?
        I,      "I",        desc="Idle";
        //F,      "F",        desc="Forward";
        FM,     "FM",       desc="Owned (forward but dirty)";
        M,      "M",        desc="Modified";
        W,      "W",        desc="Waiting on reply from L1 and data from
memory";
        WD,     "WD",       desc="Waiting on only data from memory";
        WR,     "WR",       desc="Waiting on only reply from L1";

        // Conflict states
        //WRC,    "WRC",      desc="Waiting on only replies from L1, multiple
conflicting requests";
        WDC,    "WDC",      desc="Waiting on only data from memory, multiple
conflicting requests"; // Should not see a WB event!
        WCF,    "WCF",      desc="Waiting on replies from L1, data in memory
```

```
system, multiple conflicting requests";

        // Write back states
        FW,     "FW",       desc="Forwarding on a GetX/S msg, acts like a WR
state but recycles other GetX/S requests";
        WB,     "WB",       desc="Writing back to memory";
        // WBM,    "WBM",       desc="Writing back M to memory";
        // WBFM,   "WBFM",      desc="Writing back FM to memory";
        // WBWM,   "WBWM",      desc="Writing back M to memory, but we also
have received GetX/S/Inst";
        // WBWFM,  "WBWFM",     desc="Writing back FM to memory, but we also
have received GetX/S/Inst";
        // WBMWCF, "WBMWCF",    desc="WCF but writing back M";
        // WBFMWCF,"WBFMWCF",   desc="WCF but writing back FM";
    }

    // EVENTS
    enumeration(Event, desc="Cache events") {
        GetS,                   desc="GetS from an L1 cache";
        GetX,                   desc="GetX from an L1 cache";
        GetInstr,               desc="GetInstr from an L1 cache";
        Read,                   desc="Read request an L1 cache";
        Read_Final,             desc="Read request an L1 cache(no more
requestors after this)";
        Cncl,                   desc="Cancel memory request an L1 cache
(remaining requestors after this)";
        Cncl_Final,             desc="Final cancel memory request an L1 cache
(no more requestors after this)";
        Data,                   desc="Data from memory - indicates that there
are requests waiting (don't enter I)";
        Data_Final,             desc="Data from memory - no more requests";
        // TODO: we know that a write back only occurs if the forwarding node
is evicting - safe to become forward in
        // L2 cache here.
        //WB_Data,                desc="Write back data from L1 cache";
        //DataF,                  desc="Received shared data from a cache";
        DataFM,                 desc="Received forwarded/modified data from a
cache";
        //DataE,                  desc="Received exclusive data from a
```

```
cache";
        DataM,                  desc="Received modified data from a cache";
        DACK,                   desc="DACK from L1 Cache";

        Mem_Ack,                desc="Ack from memory";

        Cncl_No_Conflict,       desc="Cancel request, BUT we have new
requests which are non-conflicting to handle (cTable is empty, requestor
queue isn't)";
        Read_No_Conflict,       desc="Read request, BUT we have new requests
which are non-conflicting to handle (cTable is empty, requestor queue
isn't)";
        Read_Delay,             desc="Read request that must be delayed";

        L2_Replacement,         desc="L2 cache line replacement";
    }

    // Internal types

    // CacheEntry
    structure(Entry, desc="...", interface="AbstractCacheEntry") {
        State CacheState,        desc="cache state";
        bool Dirty,              desc="Is the data dirty (different than
memory)?";
        DataBlock DataBlk,       desc="data for the block";
    }

    structure(TBE, desc="...") {
        Address Address,            desc="Physical address for this TBE";
        State TBEState,             desc="Transient state";
        DataBlock DataBlk,          desc="Buffer for the data block";
        bool Dirty, default="false",desc="data is dirty";
        PrefetchBit Prefetch,       desc="Set if this was caused by a
prefetch";
        MachineID L1_Read_ID,       desc="ID of the L1 cache we want to
forward the block to when we get data";
        MachineID L1_WB_ID,         desc="ID of the l1 cache that is writing
back the line";
        RequestorQueue requestors,  desc="The queue of nodes which are
```

```
currently requesting this line but don't have the line or are not scheduled
to get the line yet";
        ConflictTable cTable,       desc="The conflict table (nxn array of
boolean values) which helps work out which nodes need the line to be
transferred too";
        int remainingReq,           desc="A counter for the number of
requests remaining to be serviced";
    }



    // External types

    external_type(CacheMemory) {
        bool cacheAvail(Address);
        Address cacheProbe(Address);
        void allocate(Address);
        void deallocate(Address);
        Entry lookup(Address);
        void changePermission(Address, AccessPermission);
        bool isTagPresent(Address);
        void setMRU(Address);
    }

    external_type(TBETable) {
      TBE lookup(Address);
      void allocate(Address);
      void deallocate(Address);
      bool isPresent(Address);
    }

    // global variables

    TBETable L2_TBEs, template_hack="<L2Cache_TBE>";

    CacheMemory L2cacheMemory, template_hack="<L2Cache_Entry>",
constructor_hack='L2_CACHE_NUM_SETS_BITS,L2_CACHE_ASSOC,MachineType_L2Cache,in
t_to_string(i)';
    //CacheMemory L2cacheMemory, template_hack="<L2Cache_Entry>",
constructor_hack='L2_CACHE_NUM_SETS_BITS,L2_CACHE_ASSOC,MachineType_L2Cache,in
```

```
t_to_string(i)+"_L2"';


    // Functions

    // inclusive cache, returns L2 entries only
    Entry getL2CacheEntry(Address addr), return_by_ref="yes" {
        return L2cacheMemory[addr];
    }

    bool isL2CacheTagPresent(Address addr) {
        return (L2cacheMemory.isTagPresent(addr));
    }

    void changePermission(Address addr, AccessPermission permission) {
        if (L2cacheMemory.isTagPresent(addr)) {
            return L2cacheMemory.changePermission(addr, permission);
        }
        else {
            error("cannot change permission, L2 block not present");
        }
    }

    State getState(Address addr) {
        if(L2_TBEs.isPresent(addr)) {
            return L2_TBEs[addr].TBEState;
        }
        else if (isL2CacheTagPresent(addr)) {
            return getL2CacheEntry(addr).CacheState;
        }
        return State:I;
    }

    void setState(Address addr, State state) {
        // MUST CHANGE
        if (L2_TBEs.isPresent(addr)) {
            L2_TBEs[addr].TBEState := state;
        }
        if (isL2CacheTagPresent(addr)) {
            getL2CacheEntry(addr).CacheState := state;

            // Set permission
            // if (state == State:I) {
            //     changePermission(addr, AccessPermission:Invalid);
            // }
            // else {
            //     changePermission(addr, AccessPermission:Busy);
            // }

            if (state == State:I) {
                changePermission(addr, AccessPermission:Invalid);
            }
            else if (state == State:FM || state == State:M ) {
                changePermission(addr, AccessPermission:Read_Write);
            }
            else {
                changePermission(addr, AccessPermission:Invalid);
            }
        }
    }

    // Requestor queue functions

    /*bool isRequestor(Address addr, MachineID element) {
        return L2_TBEs[addr].requestors.contains(element);
    }*/

    bool isLastRequestor(Address addr, MachineID element) {
        return L2_TBEs[addr].requestors.lastElement(element);
    }

    /*MachineID getNextRequestor(Address addr) {
        return L2_TBEs[addr].requestors.get(0);
    }*/

    MachineID popNextRequestor(Address addr) {
        return L2_TBEs[addr].requestors.pop();
    }

    /*void addNode(Address addr, MachineID element) {
        if (isRequestor(addr, element) == false) {
            L2_TBEs[addr].requestors.push(element);
        }
    }*/

    /*bool finalRequestor(Address addr) {
        return (L2_TBEs[addr].requestors.getSize() <= 1) &&
L2_TBEs[addr].cTable.empty();
    }*/

    void removeRequestor(Address addr, MachineID requestor) {
        //if (isRequestor(addr, requestor)) {
        L2_TBEs[addr].requestors.remove(requestor);
        //}
    }

    // Removes the requestor and then pops the head of list
    MachineID popNextTransfer(Address addr, MachineID requestor) {
        removeRequestor(addr, requestor);
        return popNextRequestor(addr);
    }

    // Only works if newHead is inside the requestor queue already
    /*void moveToHead(Address addr, MachineID newHead) {
        if (isRequestor(addr, newHead)) {
            removeRequestor(addr, newHead);
            L2_TBEs[addr].requestors.pushToHead(newHead);
        }
    }*/

    // returns the next on the list that is not the requestor
    /*MachineID popNextTransferNotRequestor(Address addr, MachineID
requestor) {
        if (getNextRequestor(addr) == requestor) {
            moveToHead(addr, L2_TBEs[addr].requestors.get(1));
        }
        return popNextRequestor(addr);

    }*/

    // Must only be run when requestor is not the last element
    MachineID popNextTransferAfterRequestor(Address addr, MachineID
requestor) {
        assert(isLastRequestor(addr, requestor) ==  false);
        return L2_TBEs[addr].requestors.popElementAfter(requestor);
    }

    void clearRequestors(Address addr) {
        L2_TBEs[addr].requestors.clear();
    }

    void allocateTBE(Address addr) {
        if (L2_TBEs.isPresent(addr) == false) {
            check_allocate(L2_TBEs);
            L2_TBEs.allocate(addr);
            //L2_TBEs[address].DataBlk := getL2CacheEntry(address).DataBlk;
            //L2_TBEs[address].Dirty := getL2CacheEntry(address).Dirty;
            L2_TBEs[addr].requestors.clear();
            L2_TBEs[addr].cTable.setSize(numberOfL1Cache());
            if (isL2CacheTagPresent(addr)) {
                L2_TBEs[addr].TBEState := getL2CacheEntry(addr).CacheState;
            }
        }
    }

    // Also allocates the TBE, it is simpler this way as adding the nodes
simplifies the transition logic. Not used
    /*void addRequestorNode(Address addr, MachineID requestor) {
        // Check TBE is allocated first
        allocateTBE(addr);

        // Only add the node if it has not already been added, and then
subsequently removed from the requestor queue by being transferred to!
        if (L2_TBEs[addr].cTable.reportedConflict(requestor) == false) {
            addNode(addr, requestor);
        }
    }*/

    void addConflictNodes(Address addr, MachineID requestor, MachineIDset
conflicting) {
        // Add all the conflicting nodes which have not already gotten the
line

L2_TBEs[addr].requestors.pushAll(L2_TBEs[addr].cTable.addConflictNodes(request
or, conflicting));
    }

    // All delays when we want the read to recycle -- and not be processed in
addConflictNodes
    bool isReadDelay(Address addr, MachineID requestor) {
        return (getState(addr) == State:WDC) || // Already a read, so must
wait
                // When in WCF, we want to delay if this requestor is at the
end of the queue (because if we sent a "wait", other elements may be
                // added to the end of the queue - we are screwed), but if
the queue is empty, we dont want to delay
                (getState(addr) == State:WCF && isLastRequestor(addr,
requestor) && L2_TBEs[addr].requestors.isEmpty() == false);
    }

    Event writeback_request_type_to_event(CoherenceRequestType type) {
        if (type == CoherenceRequestType:DataM) {
            return Event:DataM;
        } else if (type == CoherenceRequestType:DataFM) {
            return Event:DataFM;
        } else {
            error("Invalid CacheRequestType");
        }
    }

    // Whenever a GetX/S/Inst arrives and is not recycled
    void registerRequest(Address addr) {
        allocateTBE(addr);
        L2_TBEs[addr].remainingReq := L2_TBEs[addr].remainingReq + 1;
    }

    // Out ports

    out_port(requestNetwork_out, RequestMsg, requestFromL2Cache);
    out_port(responseNetwork_out, ResponseMsg, responseFromL2Cache);
    out_port(dataResponseNetwork_out, ResponseMsg, dataResponseFromL2Cache);



    // In ports

    in_port(L1requestNetwork_in, RequestMsg, L1RequestToL2Cache) {
        if (L1requestNetwork_in.isReady()) {
            peek(L1requestNetwork_in, RequestMsg) {
                assert(in_msg.Destination.isElement(machineID));

                // Due to the fact that this method does computation ONLY
when the msg is NOT recycled, must bypass the recycling messages!

                // GETS and GETX add the requestor to the requestor queue (if
it should be added)
                if (in_msg.Type == CoherenceRequestType:GETS) {
                    // if (getState(in_msg.Address) != State:FW &&
                    //     getState(in_msg.Address) != State:WB) {
                    //
                    //     // Assuming that the GetX/S/Inst msg always comes
before the Read/Cncl msg
                    //     allocateTBE(in_msg.Address);
                    //     L2_TBEs[in_msg.Address].remainingReq :=
L2_TBEs[in_msg.Address].remainingReq + 1;
                    // }
                    trigger(Event:GetS, in_msg.Address);
                }
                else if (in_msg.Type == CoherenceRequestType:GET_INSTR) {
                    // if (getState(in_msg.Address) != State:FW &&
                    //     getState(in_msg.Address) != State:WB) {
                    //
                    //     allocateTBE(in_msg.Address);
                    //     L2_TBEs[in_msg.Address].remainingReq :=
L2_TBEs[in_msg.Address].remainingReq + 1;
                    // }
                    trigger(Event:GetInstr, in_msg.Address);
                }
                else if (in_msg.Type == CoherenceRequestType:GETX) {
                    // if (getState(in_msg.Address) != State:FW &&
                    //     getState(in_msg.Address) != State:WB) {
                    //
                    //     allocateTBE(in_msg.Address);
                    //     L2_TBEs[in_msg.Address].remainingReq :=
L2_TBEs[in_msg.Address].remainingReq + 1;
                    // }
                    trigger(Event:GetX, in_msg.Address);
                }

                // READ and CNCL adds the conflicting nodes
                else if (in_msg.Type == CoherenceRequestType:READ) {
                    assert(getState(in_msg.Address) != State:WD);

                    // Only if this read is not to be delayed (recycled) do
we add conflictnodes
                    if (isReadDelay(in_msg.Address, in_msg.Requestor) ==
false) {
                        addConflictNodes(in_msg.Address, in_msg.Requestor,
in_msg.ConflictMachs);
                        L2_TBEs[in_msg.Address].remainingReq :=
L2_TBEs[in_msg.Address].remainingReq - 1;
                        // Requestor queue is up to date now
                    }

                    if (isReadDelay(in_msg.Address, in_msg.Requestor)) {
                        trigger(Event:Read_Delay, in_msg.Address);
                    }
                    else if (L2_TBEs[in_msg.Address].cTable.empty()  &&
L2_TBEs[in_msg.Address].remainingReq == 0) {
                        trigger(Event:Read_Final, in_msg.Address);
                    }
                    else if (L2_TBEs[in_msg.Address].cTable.empty() &&
L2_TBEs[in_msg.Address].remainingReq > 0) {
                        trigger(Event:Read_No_Conflict, in_msg.Address);
                    }
                    // Delay only if we are in a conflicting state and we are
the final requestor in the RQ
                    /*else if (isReadDelay(in_msg.Address, in_msg.Requestor))
{
                        trigger(Event:Read_Delay, in_msg.Address);
                    }*/
                    /*else if (L2_TBEs[in_msg.Address].cTable.empty() ==
false
                                &&
L2_TBEs[in_msg.Address].requestors.getSize() == 1 &&
isRequestor(in_msg.Address, in_msg.Requestor)) {
                        trigger(Event:Read_Delay, in_msg.Address);
                    }*/
                    else if (L2_TBEs[in_msg.Address].cTable.empty() == false)
{
                        trigger(Event:Read, in_msg.Address);
                    }
                    else {
                        error("Invalid state for the conflict table on a Read
request");
                    }
                }

                else if (in_msg.Type == CoherenceRequestType:CNCL) {
                    addConflictNodes(in_msg.Address, in_msg.Requestor,
in_msg.ConflictMachs);
                    removeRequestor(in_msg.Address, in_msg.Requestor);
                    L2_TBEs[in_msg.Address].remainingReq :=
L2_TBEs[in_msg.Address].remainingReq - 1;
                    // Requestor queue is up to date now

                    if (L2_TBEs[in_msg.Address].cTable.empty() &&
L2_TBEs[in_msg.Address].remainingReq == 0) {
                        trigger(Event:Cncl_Final, in_msg.Address);
                    }
                    else if (L2_TBEs[in_msg.Address].cTable.empty() &&
L2_TBEs[in_msg.Address].remainingReq > 0) {
                        trigger(Event:Cncl_No_Conflict, in_msg.Address);
                    }
                    else if (L2_TBEs[in_msg.Address].cTable.empty() == false)
{
                        trigger(Event:Cncl, in_msg.Address);
                    }
                    else {
                        error("Invalid state for the conflict table on a
Cancel request");
                    }
                }

                /*else if (in_msg.Type == CoherenceRequestType:PUTX) {
                    trigger(Event:WB_Data, in_msg.Address);
                }*/

                else if (in_msg.Type == CoherenceRequestType:DataM ||
                         in_msg.Type == CoherenceRequestType:DataFM) {

                    if (L2cacheMemory.isTagPresent(in_msg.Address) ||
L2cacheMemory.cacheAvail(in_msg.Address)) {
                        trigger(writeback_request_type_to_event(in_msg.Type),
in_msg.Address);
                    }
                    else {
                        trigger(Event:L2_Replacement,
L2cacheMemory.cacheProbe(in_msg.Address));
                    }
                }

            }
        }
    }

    in_port(responseToL2Cache_in, ResponseMsg, responseToL2Cache) {
        if (responseToL2Cache_in.isReady()) {
            peek(responseToL2Cache_in, ResponseMsg) {
                if (in_msg.Type == CoherenceResponseType:MEMORY_DATA) {
                    if (L2_TBEs.isPresent(in_msg.Address) &&
L2_TBEs[in_msg.Address].remainingReq > 0) {
                        trigger(Event:Data, in_msg.Address);

                    }
                    else {
                        trigger(Event:Data_Final, in_msg.Address);
                    }
                }
                else if (in_msg.Type == CoherenceResponseType:MEMORY_ACK) {
                    trigger(Event:Mem_Ack, in_msg.Address);
                }
            }
        }
    }

    //Dummy, not used in L2
    in_port(dataResponseNetwork_in, ResponseMsg, dataResponseToL2Cache) {
        if (dataResponseNetwork_in.isReady()) {
            peek(dataResponseNetwork_in, ResponseMsg) {
                // Unnessary for L2 cache
                if (in_msg.Type == CoherenceResponseType:DACK){
                    trigger(Event:DACK, in_msg.Address);
                }
            }
        }
    }

    // Events

    action(a_issueFetchToMemory, "a", desc="fetch data from memory") {
        enqueue(requestNetwork_out, RequestMsg, latency="L2_REQUEST_LATENCY")
{
            out_msg.Address := address;
            out_msg.Type := CoherenceRequestType:GETS;
            out_msg.Requestor := machineID;
            out_msg.Destination.add(map_Address_to_Directory(address));
            out_msg.MessageSize := MessageSizeType:Control;
        }
    }

    action(aa_sendAck, "aa", desc="Send acknowledgement message to
requestor") {
        peek(L1requestNetwork_in, RequestMsg) {
            enqueue(responseNetwork_out, ResponseMsg,
latency="L1_REQUEST_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:ACK;
                out_msg.SenderMachine := MachineType:L2Cache;
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.MessageSize := MessageSizeType:Control;
            }
        }
    }

    action(c_writebackData, "c", desc="Write back data memory") {
        enqueue(requestNetwork_out, RequestMsg, latency="L2_REQUEST_LATENCY")
{
            out_msg.Address := address;
            out_msg.Type := CoherenceRequestType:PUTX;
            out_msg.Requestor := machineID;
            out_msg.Destination.add(map_Address_to_Directory(address));
            out_msg.MessageSize := MessageSizeType:Writeback_Data;
            out_msg.DataBlk := getL2CacheEntry(address).DataBlk;
        }

    }

    action(di_sendIACKToRequestor, "di", desc="send IACK to requestor") {
        peek(L1requestNetwork_in, RequestMsg) {
            enqueue(responseNetwork_out, ResponseMsg,
latency="L1_REQUEST_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:IACK;
                out_msg.SenderMachine := MachineType:L2Cache;
                out_msg.Dirty := false;
                out_msg.Sender := machineID;
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.MessageSize := MessageSizeType:Control;
            }
        }
    }

    action(d_storeDataTBE, "d", desc="Temporarily stores the data in the
TBE") {
        peek(responseToL2Cache_in, ResponseMsg) {
            L2_TBEs[address].DataBlk := in_msg.DataBlk;
            L2_TBEs[address].Dirty := in_msg.Dirty;
        }
    }

    action(rd_requestStoreDataTBE, "rd", desc="Temporarily stores the data in
the TBE (taken from WB data)") {
        peek(L1requestNetwork_in, RequestMsg) {
            L2_TBEs[address].DataBlk := in_msg.DataBlk;
            L2_TBEs[address].Dirty := in_msg.Dirty;
        }
    }

    action(f_forwardDataToCache, "f", desc="Forward data to L1 cache - not
conflicting here") {
        removeRequestor(address, L2_TBEs[address].L1_Read_ID);

        if (isL2CacheTagPresent(address)) {

            enqueue(responseNetwork_out, ResponseMsg,
latency="L2_RESPONSE_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:DataF_E;
                out_msg.Sender := machineID;
                out_msg.SenderMachine := MachineType:L2Cache;
                out_msg.Destination.add(L2_TBEs[address].L1_Read_ID);
                out_msg.DataBlk := getL2CacheEntry(address).DataBlk;
                out_msg.Dirty := getL2CacheEntry(address).Dirty;
                DEBUG_EXPR(out_msg.Address);
                DEBUG_EXPR(out_msg.Destination);
                DEBUG_EXPR(out_msg.DataBlk);
                out_msg.MessageSize := MessageSizeType:Response_Data;
            }

        }
        else {
            enqueue(responseNetwork_out, ResponseMsg,
latency="L2_RESPONSE_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:DataF_E;
                out_msg.Sender := machineID;
                out_msg.SenderMachine := MachineType:L2Cache;
                out_msg.Destination.add(L2_TBEs[address].L1_Read_ID);
                out_msg.DataBlk := L2_TBEs[address].DataBlk;
                out_msg.Dirty := L2_TBEs[address].Dirty;
                DEBUG_EXPR(out_msg.Address);
                DEBUG_EXPR(out_msg.Destination);
                DEBUG_EXPR(out_msg.DataBlk);
                out_msg.MessageSize := MessageSizeType:Response_Data;
            }
        }
    }

    action(fm_forwardmodifiedDataToCache, "fm", desc="Forward modified data
to L1 cache - not conflicting here") {
        peek(L1requestNetwork_in, RequestMsg) {
            enqueue(dataResponseNetwork_out, ResponseMsg,
latency="L2_RESPONSE_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:DataF_M;
                out_msg.Sender := machineID;
                out_msg.SenderMachine := MachineType:L2Cache;
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.DataBlk := getL2CacheEntry(address).DataBlk;
                out_msg.Dirty := getL2CacheEntry(address).Dirty;
                DEBUG_EXPR(out_msg.Address);
                DEBUG_EXPR(out_msg.Destination);
                DEBUG_EXPR(out_msg.DataBlk);
                out_msg.MessageSize := MessageSizeType:Response_Data;
            }
        }
    }

    action(ffm_forwardforwardmodifiedDataToCache, "ffm", desc="Forward
forward modified data to L1 cache - not conflicting here") {
        peek(L1requestNetwork_in, RequestMsg) {
            enqueue(dataResponseNetwork_out, ResponseMsg,
latency="L2_RESPONSE_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:DataF_FM;
                out_msg.Sender := machineID;
                out_msg.SenderMachine := MachineType:L2Cache;
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.DataBlk := getL2CacheEntry(address).DataBlk;
                out_msg.Dirty := getL2CacheEntry(address).Dirty;
                DEBUG_EXPR(out_msg.Address);
                DEBUG_EXPR(out_msg.Destination);
                DEBUG_EXPR(out_msg.DataBlk);
                out_msg.MessageSize := MessageSizeType:Response_Data;
            }
        }
    }


    action(wa_sendWBAck, "wa", desc="Send WB acknowledgement message to
requestor") {
        enqueue(responseNetwork_out, ResponseMsg,
latency="L1_REQUEST_LATENCY") {
            out_msg.Address := address;
            out_msg.Type := CoherenceResponseType:MEMORY_ACK;
            out_msg.SenderMachine := MachineType:L2Cache;
            out_msg.Destination.add(L2_TBEs[address].L1_WB_ID);
            out_msg.MessageSize := MessageSizeType:Control;
        }
    }



    action(wb_sendWBAckDirect, "wb", desc="Send WB acknowledgement message to
requestor") {
        peek(L1requestNetwork_in, RequestMsg) {
            enqueue(responseNetwork_out, ResponseMsg,
latency="L1_REQUEST_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:MEMORY_ACK;
                out_msg.SenderMachine := MachineType:L2Cache;
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.MessageSize := MessageSizeType:Control;
            }
        }
    }

    // Conflict only messages

    action(ax_ackAndTransfer, "ax", desc="Send ack and XFR to requester
cache") {
        peek(L1requestNetwork_in, RequestMsg) {

            enqueue(responseNetwork_out, ResponseMsg,
latency="L1_REQUEST_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:XFR;
                out_msg.SenderMachine := MachineType:L2Cache;
                out_msg.Dirty := false;
                out_msg.Sender := machineID;
                out_msg.Transfer := popNextRequestor(address);
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.MessageSize := MessageSizeType:Control;
            }
        }
    }

    action(at_sendAckOrTransfer, "at", desc="Send acknowledgement message to
requestor or transfer if necessary") {
        peek(L1requestNetwork_in, RequestMsg) {
            // No final nodes to transfer too, simply send ack
            if (L2_TBEs[address].requestors.getSize() == 0) {
                enqueue(responseNetwork_out, ResponseMsg,
latency="L1_REQUEST_LATENCY") {
                    out_msg.Address := address;
                    out_msg.Type := CoherenceResponseType:ACK;
                    out_msg.SenderMachine := MachineType:L2Cache;
                    out_msg.Destination.add(in_msg.Requestor);
                    out_msg.MessageSize := MessageSizeType:Control;
                }
            }
            // Final node to be transferred too, send to this
            else {
                enqueue(responseNetwork_out, ResponseMsg,
latency="L1_REQUEST_LATENCY") {
                    out_msg.Address := address;
                    out_msg.Type := CoherenceResponseType:XFR;
                    out_msg.SenderMachine := MachineType:L2Cache;
                    out_msg.Dirty := false;
                    out_msg.Sender := machineID;
                    out_msg.Transfer := popNextRequestor(address);
                    out_msg.Destination.add(in_msg.Requestor);
                    out_msg.MessageSize := MessageSizeType:Control;
                }
            }
        }
    }

    // Using popNextTransfer because this (and tdx_dataAndTransferFromTBE)
are called when home will be sending the data to the node: they should be
removed from
    // the requestor queue.
    action(dx_dataAndTransfer, "dx", desc="Send data and XFR to requester
cache") {
        peek(L1requestNetwork_in, RequestMsg) {

            enqueue(responseNetwork_out, ResponseMsg,
latency="L2_RESPONSE_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:DXFR;
                out_msg.SenderMachine := MachineType:L2Cache;
                out_msg.DataBlk := L2_TBEs[address].DataBlk;
                out_msg.Dirty := false;
                out_msg.Sender := machineID;
                out_msg.Transfer := popNextTransfer(address,
in_msg.Requestor);
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.MessageSize := MessageSizeType:Response_Data;
            }
        }
    }

    action(tdx_dataAndTransferFromTBE, "tdx", desc="Send data and XFR to
requester cache (from TBE)") {

        enqueue(responseNetwork_out, ResponseMsg,
latency="L2_RESPONSE_LATENCY") {
            out_msg.Address := address;
            out_msg.Type := CoherenceResponseType:DXFR;
            out_msg.SenderMachine := MachineType:L2Cache;
            out_msg.DataBlk := L2_TBEs[address].DataBlk;
            out_msg.Dirty := false;
            out_msg.Sender := machineID;
            out_msg.Transfer := popNextTransfer(address,
L2_TBEs[address].L1_Read_ID);
            out_msg.Destination.add(L2_TBEs[address].L1_Read_ID);
            out_msg.MessageSize := MessageSizeType:Response_Data;
        }
    }

    action(w_sendWait, "w", desc="Send wait to requester cache") {
        peek(L1requestNetwork_in, RequestMsg) {
            removeRequestor(address, in_msg.Requestor);

            enqueue(responseNetwork_out, ResponseMsg,
latency="L1_REQUEST_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:Wait;
                out_msg.SenderMachine := MachineType:L2Cache;
                out_msg.DataBlk := L2_TBEs[address].DataBlk;
                out_msg.Dirty := false;
                out_msg.Sender := machineID;
                //out_msg.Transfer := in_msg.Requestor;
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.MessageSize := MessageSizeType:Response_Data;
            }
        }
    }

    action(wx_sendWaitAndTransfer, "wx", desc="Send wait and XFR to requester
cache") {
        peek(L1requestNetwork_in, RequestMsg) {
            //removeRequestor(address, in_msg.Requestor);

            enqueue(responseNetwork_out, ResponseMsg,
latency="L1_REQUEST_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:WaitXFR;
                out_msg.SenderMachine := MachineType:L2Cache;
                out_msg.DataBlk := L2_TBEs[address].DataBlk;
                out_msg.Dirty := false;
                out_msg.Sender := machineID;
                out_msg.Transfer := popNextTransferAfterRequestor(address,
in_msg.Requestor);
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.MessageSize := MessageSizeType:Response_Data;
            }
            // Since this node is ready to complete its memory transaction as
soon as it gets the line, it should be passed the line ASAP
            // Cannot do this now
            //moveToHead(address, in_msg.Requestor);
        }
    }

    // Other actions

    action(ss_allocateTBE, "\s", desc="Allocates the TBE (only for WB)") {
        allocateTBE(address);
    }
    action(s_deallocateTBE, "s", desc="Deallocate external TBE") {
        L2_TBEs.deallocate(address);
    }

    action(dc_deallocateL2CacheBlock, "dc", desc="Deallocate L2 cache block.
Sets the cache to not present.") {
        assert(L2cacheMemory.isTagPresent(address));
        L2cacheMemory.deallocate(address);
    }

    action(ac_allocateL2CacheBlock, "ac", desc="Set L2-cache tag equal to tag
of block B.") {
        assert (L2cacheMemory.isTagPresent(address) == false); //{
            L2cacheMemory.allocate(address);
        //}
    }

    action(wc_writeHomeDataToL1Cache, "wc", desc="Write data from L1 cache")
{
        peek(L1requestNetwork_in, RequestMsg) {
            getL2CacheEntry(address).DataBlk := in_msg.DataBlk;
            getL2CacheEntry(address).Dirty := in_msg.Dirty;
        }
    }

    action(ct_copydataTBE, "ct", desc="Copies data from the L2 cache into the
TBE"){
        L2_TBEs[address].DataBlk := getL2CacheEntry(address).DataBlk;
        L2_TBEs[address].Dirty := getL2CacheEntry(address).Dirty;
    }

    action(o_popL1RequestQueue, "o", desc="Pop L1 request queue.") {
        L1requestNetwork_in.dequeue();
    }

    action(oo_popIncomingResponseQueue, "\o", desc="Pop response to L2
queue.") {
        responseToL2Cache_in.dequeue();
    }
    action(od_popIncomingDataResponseQueue, "od", desc="Pop DACK response to
L2 queue.") {
        dataResponseNetwork_in.dequeue();
    }

    action(rr_recordRead, "\r", desc="Record L1 read") {
        peek(L1requestNetwork_in, RequestMsg) {
            L2_TBEs[in_msg.Address].L1_Read_ID := in_msg.Requestor;
        }
    }

    action(gr_registerGet, "gr", desc="Registers this get request") {
        peek(L1requestNetwork_in, RequestMsg) {
            registerRequest(in_msg.Address);
        }
    }

    action(r_recordWB, "r", desc="Record L1 read") {
        peek(L1requestNetwork_in, RequestMsg) {
            L2_TBEs[in_msg.Address].L1_WB_ID := in_msg.Requestor;
        }
    }

    action(mr_setMRU, "mr", desc="manually set the MRU bit for cache line" )
{
        if(L2cacheMemory.isTagPresent(address)) {
            L2cacheMemory.setMRU(address);
        }
    }

    action(rq_recycleRequestQueue, "rq", desc="Send the head of the request
queue to the back of the queue.") {
        L1requestNetwork_in.recycle();
    }

    action(uu_profileMiss, "\u", desc="Profile the demand miss") {
      peek(L1requestNetwork_in, RequestMsg) {
        // AccessModeType not implemented
        if (in_msg.Type != CoherenceRequestType:GET_INSTR){
            profile_L2Cache_miss(convertToGenericType(in_msg.Type),
in_msg.AccessMode, MessageSizeTypeToInt(in_msg.MessageSize),
in_msg.Prefetch, machineIDToNodeID(in_msg.Requestor));
        }
      }
    }

    // Transitions

    transition(I, {GetS, GetInstr, GetX}, W) {
        gr_registerGet;
        di_sendIACKToRequestor;
        a_issueFetchToMemory;
        uu_profileMiss;
        o_popL1RequestQueue;
    }

    // If we received a cancel request before the line came back.
    transition(I, {Data, Data_Final}) {
        oo_popIncomingResponseQueue;
    }



    transition(I, DataM, M) {
        ac_allocateL2CacheBlock;
        wc_writeHomeDataToL1Cache;
        wb_sendWBAckDirect;
        mr_setMRU;
        o_popL1RequestQueue;
    }

    transition(I, DataFM, FM) {
        ac_allocateL2CacheBlock;
        wc_writeHomeDataToL1Cache;
        wb_sendWBAckDirect;
        mr_setMRU;
        o_popL1RequestQueue;
    }
```

```
// Requests when cached

transition(M, {GetS, GetInstr, GetX}, FW) {
    fm_forwardmodifiedDataToCache;
    //ct_copydataTBE;
    //mr_setMRU;
    dc_deallocateL2CacheBlock;
    gr_registerGet;
    //uu_profileMiss; // TODO: want to change this to an L2 hit profile?
    o_popL1RequestQueue;
}

transition(FM, {GetS, GetInstr, GetX}, FW) {
    ffm_forwardforwardmodifiedDataToCache;
    //ct_copydataTBE;
    //mr_setMRU;
    dc_deallocateL2CacheBlock;
    gr_registerGet;
    //uu_profileMiss; // TODO: want to change this to an L2 hit profile?
    o_popL1RequestQueue;
}

transition({I, FM, M, W, WD, WR, WDC, WCF, FW, WB}, DACK) {
    od_popIncomingDataResponseQueue;
}

// transition({M, FM}, {GetS, GetInstr, GetX}, WB) {
//     ss_allocateTBE;
//     c_writebackData;
//     dc_deallocateL2CacheBlock;
// }

transition(FW, {GetS, GetInstr, GetX}) {
    rq_recycleRequestQueue;
}

transition({FW, FM, M}, {Data, Data_Final}) {
    oo_popIncomingResponseQueue;
}

transition(FW, Cncl, WCF) {
    ax_ackAndTransfer;
    o_popL1RequestQueue;
}

transition(FW, Cncl_Final, I) {
    aa_sendAck;
    s_deallocateTBE;
    o_popL1RequestQueue;
}

transition(FW, Cncl_No_Conflict, WR) {
    at_sendAckOrTransfer;
    o_popL1RequestQueue;
}

// L2_Replacement

transition({M, FM}, L2_Replacement, WB) {
    ss_allocateTBE;
    c_writebackData;
    dc_deallocateL2CacheBlock;
}

// transition(I, L2_Replacement) {
//     dc_deallocateL2CacheBlock;
// }
//
// transition({W, WD, WR, WDC, WCF, FW}, L2_Replacement) {
//     rq_recycleRequestQueue;
// }


transition(WB, Mem_Ack, I) {
    s_deallocateTBE;
    oo_popIncomingResponseQueue;
}
```

```
    transition(WB, {GetS, GetInstr, GetX}) {
        rq_recycleRequestQueue;
    }



    // transition(WB, Mem_Ack, I) {
    //     wa_sendWBAck;
    //     s_deallocateTBE;
    //     oo_popIncomingResponseQueue;
    // }

    // transition(WB, {GetS, GetInstr, GetX}) {
    //     rq_recycleRequestQueue;
    // }



    // transition({W, WR}, WB_Data, WBW) {
    //     c_writebackData;
    //     r_recordWB;
    //     rd_requestStoreDataTBE;
    //     o_popL1RequestQueue;
    // }
    transition({W, WR}, {DataM, DataFM}, WR) {
        rd_requestStoreDataTBE;
        r_recordWB;
        wa_sendWBAck;
        o_popL1RequestQueue;
    }

    // transition(WBW, {GetS, GetInstr, GetX, Read, Read_Final,
Read_No_Conflict, Cncl, Cncl_Final, Cncl_No_Conflict}) {
    //     rq_recycleRequestQueue;
    // }

    // Have the data already from the WB_Data
    // transition(WBW, Mem_Ack, WR) {
    //     wa_sendWBAck;
    //     oo_popIncomingResponseQueue;
    // }
    //
    // transition(WBW, Data) {
    //     oo_popIncomingResponseQueue;
    // }

    // WBWCF

    // transition(WCF, {DataF, DataFM}, WCF) {
    //     rd_requestStoreDataTBE;
    //     r_recordWB;
    //     wa_sendWBAck;
    //     o_popL1RequestQueue;
    // }


    // transition(WBWCF, {GetS, GetInstr, GetX, Read, Read_Final,
Read_No_Conflict, Cncl, Cncl_Final, Cncl_No_Conflict}) {
    //     rq_recycleRequestQueue;
    // }

    // Have the data already from the WB_Data
    // transition(WBWCF, Mem_Ack, WCF) {
    //     wa_sendWBAck;
    //     oo_popIncomingResponseQueue;
    // }
    //
    // transition(WBWCF, Data) {
    //     oo_popIncomingResponseQueue;
    // }

    // Because of the WBW -> WR transition, could get data in WR
    transition(WR, {Data, Data_Final}) {
        oo_popIncomingResponseQueue;
    }

    transition(WB, Data_Final) {
        oo_popIncomingResponseQueue;
    }
```

```
transition(W, {Read_Final, Read_No_Conflict}, WD) {
    rr_recordRead;
    o_popL1RequestQueue;
}

transition(W, Cncl_Final, I) {
    aa_sendAck;
    s_deallocateTBE;
    o_popL1RequestQueue;
}

transition(W, Cncl_No_Conflict) {
   aa_sendAck;
   o_popL1RequestQueue;
}

transition(W, {Data, Data_Final}, WR) {
    d_storeDataTBE;
    oo_popIncomingResponseQueue;
}

// Requests left to be issued, get line agian
transition(WD, Data, W) {
    d_storeDataTBE;
    f_forwardDataToCache;
    a_issueFetchToMemory;
    oo_popIncomingResponseQueue;
}

transition(WD, Data_Final, I) {
    d_storeDataTBE;
    f_forwardDataToCache;
    s_deallocateTBE;
    oo_popIncomingResponseQueue;
}

transition(WR, Read_Final, I) {
    rr_recordRead;
    f_forwardDataToCache;
    s_deallocateTBE;
    o_popL1RequestQueue;
}

// transition(WR, Read_No_Conflict, WD) {
//     rr_recordRead;
//     a_issueFetchToMemory;
//     o_popL1RequestQueue;
// }

transition(WR, Read_No_Conflict) {
    rr_recordRead;
    f_forwardDataToCache;
    o_popL1RequestQueue;
}

transition(WR, Cncl_Final, I) {
    aa_sendAck;
    s_deallocateTBE;
    o_popL1RequestQueue;
}

transition(WR, Cncl_No_Conflict) {
    at_sendAckOrTransfer;
    o_popL1RequestQueue;
}

// Conflicting transitions

transition(WD, Cncl, WCF) {
    ax_ackAndTransfer;
    o_popL1RequestQueue;
}

// Should never occur
/*transition(WD, Read, WDC) {
    rr_recordRead;
    o_popL1RequestQueue;
```

```
}*/



transition(WR, Cncl, WCF) {
    ax_ackAndTransfer;
    o_popL1RequestQueue;
}

transition(WR, Read, WCF) {
    dx_dataAndTransfer;
    o_popL1RequestQueue;
}

// TODO: I think it is fine to NOT transfer to WC here
transition(W, {GetS, GetInstr, GetX}) {
    gr_registerGet;
    di_sendIACKToRequestor;
    uu_profileMiss;
    o_popL1RequestQueue;
}

transition(W, Cncl, WCF) {
    ax_ackAndTransfer;
    o_popL1RequestQueue;
}

transition(W, Read, WDC) {
    rr_recordRead;
    o_popL1RequestQueue;
}

transition(WD, {GetS, GetInstr, GetX}) {
    gr_registerGet;
    di_sendIACKToRequestor;
    uu_profileMiss;
    o_popL1RequestQueue;
}

transition(WR, {GetS, GetInstr, GetX}) {
    gr_registerGet;
    di_sendIACKToRequestor;
    uu_profileMiss;
    o_popL1RequestQueue;
}

/*transition(WRC, {GetS, GetInstr, GetX}) {
    di_sendIACKToRequestor;
    o_popL1RequestQueue;
}*/

transition(WDC, {GetS, GetInstr, GetX}) {
    gr_registerGet;
    di_sendIACKToRequestor;
    uu_profileMiss;
    o_popL1RequestQueue;
}

// A solution here is to sent a wait transfer request
transition(WDC, Read_Delay) {
    rq_recycleRequestQueue;
}

transition(WDC, {Data, Data_Final}, WCF) {
    d_storeDataTBE;
    tdx_dataAndTransferFromTBE;
    oo_popIncomingResponseQueue;
}

/*transition(WRC, Read, WCF) {
    dx_dataAndTransfer;
    o_popL1RequestQueue;
}*/

/*transition(WRC, Cncl, WCF) {
    ax_ackAndTransfer;
    o_popL1RequestQueue;
}*/
```

```
transition(WCF, {GetS, GetInstr, GetX}) {
    gr_registerGet;
    di_sendIACKToRequestor;
    uu_profileMiss;
    o_popL1RequestQueue;
}

transition(WCF, {Data, Data_Final}) {
    oo_popIncomingResponseQueue;
}

transition(WCF, Read) {
    wx_sendWaitAndTransfer;
    o_popL1RequestQueue;
}

transition(WCF, Read_Final, I) {
    w_sendWait;
    s_deallocateTBE;
    o_popL1RequestQueue;
}

transition(WCF, Read_No_Conflict, W) {
    w_sendWait;
    a_issueFetchToMemory;
    o_popL1RequestQueue;
}

transition(WCF, Read_Delay) {
    rq_recycleRequestQueue;
}

transition(WCF, Cncl) {
    ax_ackAndTransfer;
    o_popL1RequestQueue;
}

transition(WCF, Cncl_Final, I) {
    aa_sendAck;
```

```
        s_deallocateTBE;
        o_popL1RequestQueue;
    }

    transition(WCF, Cncl_No_Conflict, W) {
        aa_sendAck;
        a_issueFetchToMemory;
        o_popL1RequestQueue;
    }
}
```

Appendix C

Main Memory Controller SLICC Code

```
/*
    Copyright (C) 1999-2005 by Mark D. Hill and David A. Wood for the
    Wisconsin Multifacet Project.  Contact: gems@cs.wisc.edu
    http://www.cs.wisc.edu/gems/

    --------------------------------------------------------------------

    This file is part of the SLICC (Specification Language for
    Implementing Cache Coherence), a component of the Multifacet GEMS
    (General Execution-driven Multiprocessor Simulator) software
    toolset originally developed at the University of Wisconsin-Madison.


    SLICC was originally developed by Milo Martin with substantial
    contributions from Daniel Sorin.

    Substantial further development of Multifacet GEMS at the
    University of Wisconsin was performed by Alaa Alameldeen, Brad
    Beckmann, Jayaram Bobba, Ross Dickson, Dan Gibson, Pacia Harper,
    Derek Hower, Milo Martin, Michael Marty, Carl Mauer, Michelle Moravan,
    Kevin Moore, Manoj Plakal, Daniel Sorin, Haris Volos, Min Xu, and Luke
Yen.

    --------------------------------------------------------------------
    If your use of this software contributes to a published paper, we
    request that you (1) cite our summary paper that appears on our
    website (http://www.cs.wisc.edu/gems/) and (2) e-mail a citation
    for your published paper to gems@cs.wisc.edu.

    If you redistribute derivatives of this software, we request that
    you notify us and either (1) ask people to register with us at our
    website (http://www.cs.wisc.edu/gems/) or (2) collect registration
    information and periodically send it to us.

    --------------------------------------------------------------------

    Multifacet GEMS is free software; you can redistribute it and/or
    modify it under the terms of version 2 of the GNU General Public
    License as published by the Free Software Foundation.

    Multifacet GEMS is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with the Multifacet GEMS; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
    02111-1307, USA

    The GNU General Public License is contained in the file LICENSE.

### END HEADER ###
*/
/*
 * $Id: MOESI_CMP_token-dir.sm 1.6 05/01/19 15:48:35-06:00
mikem@royal16.cs.wisc.edu $
 */

/*
Code taken from MESI_CMP_directory-mem.sm
Modified by Andrew Hay (andrewh@cs.auckland.ac.nz), 2011
```

```
*/

machine(Directory, "MESIF protocol") {

    MessageBuffer requestToDir, network="From", virtual_network="2",
ordered="false";
    //MessageBuffer responseToDir, network="From", virtual_network="3",
ordered="false";    // Noone is sending on this VN yet
    MessageBuffer responseFromDir, network="To", virtual_network="3",
ordered="false";

    // STATES
    enumeration(State, desc="Directory states", default="Directory_State_I")
{
        // Base states
        I, desc="Owner";
    }

    // Events
    enumeration(Event, desc="Directory events") {
        Fetch, desc="A GETX arrives";
        Data, desc="A GETS arrives";
    }

    // TYPES

    // DirectoryEntry
    structure(Entry, desc="...") {
        DataBlock DataBlk,             desc="data for the block";
    }

    external_type(DirectoryMemory) {
        Entry lookup(Address);
        bool isPresent(Address);
    }


    // ** OBJECTS **
```

```
DirectoryMemory directory, constructor_hack="i";

State getState(Address addr) {
return State:I;
}

void setState(Address addr, State state) {
}

// ** OUT_PORTS **
out_port(responseNetwork_out, ResponseMsg, responseFromDir);

// ** IN_PORTS **

in_port(requestNetwork_in, RequestMsg, requestToDir) {
    if (requestNetwork_in.isReady()) {
        peek(requestNetwork_in, RequestMsg) {
            assert(in_msg.Destination.isElement(machineID));
            if (in_msg.Type == CoherenceRequestType:GETS) {
                trigger(Event:Fetch, in_msg.Address);
            }
            else if (in_msg.Type == CoherenceRequestType:GETX) {
                trigger(Event:Fetch, in_msg.Address);
            }
            else if (in_msg.Type == CoherenceRequestType:PUTX) {
                trigger(Event:Data, in_msg.Address);
            }
            else {
                error("Invalid message");
            }
        }
    }
}

// Never used!
/*in_port(responseNetwork_in, ResponseMsg, responseToDir) {
    if (responseNetwork_in.isReady()) {
        peek(responseNetwork_in, ResponseMsg) {
            assert(in_msg.Destination.isElement(machineID));
```

```
                if (in_msg.Type == CoherenceResponseType:MEMORY_DATA) {
                    trigger(Event:Data, in_msg.Address);
                }
                else {
                    DEBUG_EXPR(in_msg.Type);
                    error("Invalid message");
                }
            }
        }
    }*/

  // Actions
    action(a_sendAck, "a", desc="Send ack to L2") {
        peek(requestNetwork_in, RequestMsg) {
            enqueue(responseNetwork_out, ResponseMsg,
latency="MEMORY_LATENCY") {
            out_msg.Address := address;
            out_msg.Type := CoherenceResponseType:MEMORY_ACK;
            out_msg.Sender := machineID;
            out_msg.Destination.add(in_msg.Requestor);
            out_msg.MessageSize := MessageSizeType:Response_Control;
            }
        }
    }

    action(d_sendData, "d", desc="Send data to requestor") {
        peek(requestNetwork_in, RequestMsg) {
            enqueue(responseNetwork_out, ResponseMsg,
latency="MEMORY_LATENCY") {
                out_msg.Address := address;
                out_msg.Type := CoherenceResponseType:MEMORY_DATA;
                out_msg.Sender := machineID;
                out_msg.Destination.add(in_msg.Requestor);
                out_msg.DataBlk := directory[in_msg.Address].DataBlk;
                out_msg.Dirty := false;
                out_msg.MessageSize := MessageSizeType:Response_Data;
            }
        }
    }
    action(j_popIncomingRequestQueue, "j", desc="Pop incoming request queue")
{
        requestNetwork_in.dequeue();
    }

    /*action(k_popIncomingResponseQueue, "k", desc="Pop incoming request
queue") {
        responseNetwork_in.dequeue();
    }*/

    action(m_writeDataToMemory, "m", desc="Write dirty writeback to memory")
{
        peek(requestNetwork_in, RequestMsg) {
            directory[in_msg.Address].DataBlk := in_msg.DataBlk;
            DEBUG_EXPR(in_msg.Address);
            DEBUG_EXPR(in_msg.DataBlk);
        }
    }

  // TRANSITIONS

    transition(I, Fetch) {
        d_sendData;
        j_popIncomingRequestQueue;
    }

    transition(I, Data) {
        m_writeDataToMemory;
        a_sendAck;
        j_popIncomingRequestQueue;
    }
}
```

References

[1] M. Azimi, F. Briggs, M. Cekleov, M. Khare, A. Kumar, and L. P. Looi. Scalability port: A coherent interface for shared memory multiprocessors. In *Proceedings of the 10th Symposium on High Performance Interconnects HOT Interconnects*, HotI '02, pages 65–70, Washington, DC, USA, 2002. IEEE Computer Society.

[2] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Trans. Comput. Syst.*, 13:205–243, August 1995.

[3] L. Cheng, J. Carter, and D. Dai. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 328 –339, Feb. 2007.

[4] B. Cuesta, A. Robles, and J. Duato. An effective starvation avoidance mechanism to enhance the token coherence protocol. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:47–54, 2007.

[5] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11 – 13, May 2005.

[6] J. Goodman and H. Hum. Mesif: A two-hop cache coherency protocol for point-to-point interconnects. Technical Report 2004-002, Department of Computer Science, University of Auckland, 2004.

[7] J. Goodman and H. Hum. Mesif: A two-hop cache coherency protocol for point-to-point interconnects. Technical Report 2009-002, Department of Computer Science, University of Auckland, 2009.

[8] H. Grahn, P. Stenstrm, and M. Dubois. Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. *Future Generation Computer Systems*, 11:247–271, 1995.

[9] D. Kanter. The common system interface: Intel's future interconnect. `http://www.realworldtech.com/page.cfm?ArticleID=RWT082807020032&p=5`.

[10] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 148–159, New York, NY, USA, 1990. ACM.

[11] R. Maddox, G. Singh, and R. Safranek. *Weaving high performance multiprocessor fabric: architectural insights into the Intel QuickPath Interconnect.* Intel Press, 2009.

[12] M. Martin, M. Hill, and D. Wood. Token coherence: decoupling performance and correctness. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 182 – 193, June 2003.

[13] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 206–217, New York, NY, USA, 2003. ACM.

[14] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacets general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33:2005, 2005.

[15] A. Raghavan, C. Blundell, and M. M. K. Martin. Token tenure: Patching token counting using directory-based cache coherence. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 47–58, Washington, DC, USA, 2008. IEEE Computer Society.

[16] D. J. Sorin, M. D. Hill, and D. A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–197, 2011.

[17] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *Proceedings of the 13th annual international symposium on Computer architecture*, ISCA '86, pages 414–423, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[18] W. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, ASPLOS-III, pages 243–256, New York, NY, USA, 1989. ACM.