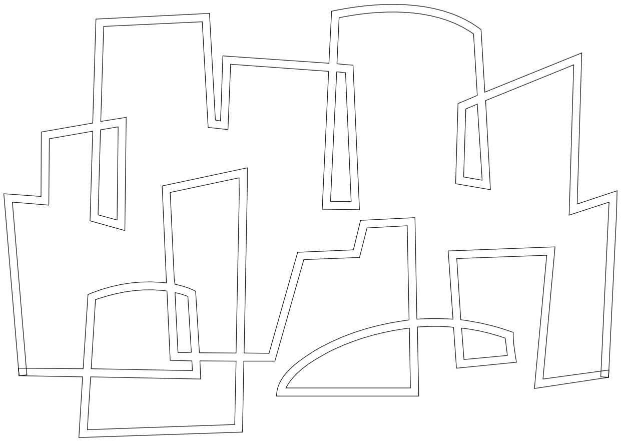
Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols

Xiaowei Shen, Arvind

1997, January

Computation Structures Group
Memo 398



LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols

Computation Structures Group Memo 398
May 2, 1998

Xiaowei Shen and Arvind

This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310 and Ft Huachuca contract DABT63-95-C-0150.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols

Xiaowei Shen and Arvind

May 2, 1998

Abstract

We propose a two-phase Imperative-Directive design methodology for designing cache coherence protocols, and use it to develop a family of protocols to implement Sequential Consistency in a distributed system with hierarchical caches. In the Imperative design phase, actions or state transitions are defined to ensure that the system only exhibits behaviors that are consistent with the memory model. In the Directive design phase one ensures liveness, i.e., the system eventually takes the desired action. In each design phase the protocol can be refined incrementally to accommodate implementation constraints. The separation of correctness and liveness concerns (and successive refinement) greatly simplifies protocol design and verification. The methodology is especially suitable for designing adaptive protocols which essentially allow imperative actions to be invoked adaptively according to program access patterns.

1 Introduction

The design of cache coherence protocols plays an important role in building parallel or distributed systems that support shared memory. Protocols can be implemented completely in hardware or completely in software or using a combination of the both. The performance of shared memory systems largely depends on the cache coherence protocols that are responsible for maintaining a coherent view of replicated data in accordance with a memory model. Over the years, the desire to achieve higher performance has resulted in more and more sophisticated cache coherence protocols, which are difficult to design and verify. In this paper we present a new Imperative-Directive methodology for designing protocols and verifying them against a memory model. The methodology is illustrated through an elaborate protocol that implements Sequential Consistency on DSM (Distributed Shared Memory) systems with hierarchy of caches.

1.1 Memory Models

A memory model is a contract that specifies the memory behavior which the system implementors (architects, compiler writers, etc.) provide to the programmers. Sequential Consistency [17] has been the dominant memory model in parallel computing for decades, but for performance reasons, both architects and compiler writers have been exploring alternative memory models that allow more implementation flexibility. Architects prefer weaker instruction orderings (see, for example, PowerPC [19]), which often give rise to relaxed memory models such as Weak Consistency [8], Release Consistency [10, 11] and Lazy Release Consistency [15]. The language and compiler community have suggested their own relaxed memory models such as Location Consistency [9] and DAG Consistency [5]. One problem with relaxed memory models is that even experts may not agree on their precise definition.

We have chosen Sequential Consistency [17] to demonstrate our methodology for designing protocols. This is not because we believe Sequential Consistency is the most desirable memory model, but rather because there is a consensus on its definition. The correctness of a protocol to implement a memory model can be discussed only if there is a precise specification of the memory model. It is important that the specification be independent of any specific implementation, and thus of caches, write buffers and interconnection networks etc. We will present an operational but fairly abstract view of Sequential Consistency, and then design protocols that admit exactly those behaviors that are permitted by this operational model. Needless to say that the same technique can be applied to designing and verifying coherence protocols for other memory models.

1.2 Design Methodology

In spite of the number of publications on cache coherence protocols [12, 18, 16, 1, 2], it is difficult to discern a methodology that has guided the design of these protocols. A major source of difficulty in protocol design is that designers often try to deal with many different issues simultaneously. Are cache states being maintained correctly? Is deadlock possible due to reordering of messages or lack of buffers in the network? What is the consequence of having write buffers and allowing instructions to be executed out-of-order? Answering these questions can be difficult in asynchronous systems with distributed control. The net result is that protocol design is viewed as black magic, where even the designers are not totally confident of their understanding of the protocol behavior.

We propose a two phase Imperative-Directive design methodology to rectify this problem. The methodology completely separates the *correctness* and the *liveness* concerns in the design process. Correctness ensures that the system can only exhibit behaviors that are allowed by the memory model. The rules that specify such state transitions are called *imperative rules*. The protocol designer initially focuses on developing a complete set of imperative rules. In the second phase, the main concern is liveness, i.e., ensuring via *directive rules* that the system always takes appropriate imperative actions at appropriate times. Improper conditions for invoking imperative rules can cause deadlocks or livelocks but cannot affect the correctness of the system.

By separating the correctness and the liveness concerns, the Imperative-Directive methodology can dramatically simplify the design and verification of cache coherence protocols. Protocols designed using this methodology are often easy to understand, modify and reason about. We illustrate our methodology by successively developing a family of cache coherence protocols to implement Sequential Consistency on DSM systems with hierarchy of caches. The final protocol is, to our knowledge, the first precise description of a provably correct coherence protocol for DSM systems with multi-level caches. The methodology has proved extremely effective in designing adaptive cache coherence protocols [23] because adaptability is only about when and how to invoke imperative actions; imperative rules remain unaffected.

1.3 Formal Verification

The verification of cache coherence protocols has gained considerable attention in recent years [4, 27, 22, 21]. Most methods verify certain invariants for cache coherence protocols, and are based on state enumeration [13, 14] and symbolic model checking [6, 7, 20], which can check correctness of assertions by exhaustively exploring all reachable states of the system. For example, Stern and Dill [27] use the Murφ system to automatically check if all reachable states satisfy certain properties which are attached to protocol specifications. Pong and Dubois [21] exploit the symmetry and homogeneity of the system states by keeping track of whether zero, one or multiple copies have been cached. This reduces the state space and makes the verification independent of the number of processors. Generally speaking, the major difference among these techniques is the representation of protocol states and the pruning method adopted in the state expansion process.

Exponential state explosion has been a serious concern for model checker approaches. Another problem is that it is often difficult to choose the invariants in a systematic manner or to convince oneself that all the important invariants have been considered. While some invariants are obvious (e.g., two L1 caches should not contain the same address in the exclusive state simultaneously), many others are motivated by the specific protocol implementation instead of the memory model. Sometimes it is not even clear if the chosen invariants are necessary or sufficient for the correctness. This means that for the same memory model, we may have to prove very different properties for different implementations. In this sense, these techniques are more like a bag of useful tools for debugging cache coherence protocols, rather than for verifying them.

The difficulty of protocol verification with current approaches can be largely attributed to the fact that protocols are designed and verified separately and sequentially. In our approach, both the memory model and the protocol are expressed in the same formalism, and there is a notion of when one system *completely implements* another system. We begin with a baseline protocol as the operational specification of the memory model, and then refine the protocol successively by incorporating more and more implementation and optimization details. Protocols are designed and verified iteratively throughout the successive process. The invariants that need to be verified are usually straightforward, because the semantics gap between the two models associated with each refinement step is relatively small. Our experience shows that most of the commonly known invariants systematically show up as lemmas and can be verified by case analysis on rewriting rules.

1.4 The Organization of the Paper

We begin with a brief introduction to our formalism, Term Rewriting Systems, and the notion of a complete implementation with respect to a specification. In Section 3 we give a baseline protocol, the SC model, which will be used as the specification of Sequential Consistency. We define the HC model, a directory-based coherence protocol for DSM systems with hierarchical

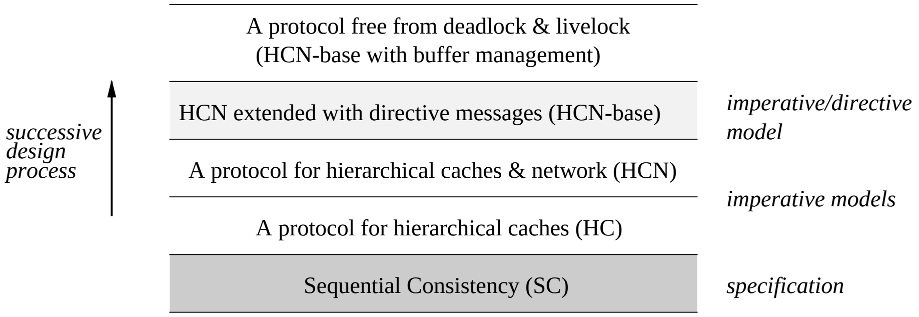

A protocol free from deadlock & livelock
(HCN-base with buffer management)
HCN extended with directive messages (HCN-base)
A protocol for hierarchical caches & network (HCN)
A protocol for hierarchical caches (HC)
Sequential Consistency (SC)
successive design process
imperative/directive model
imperative models
specification

Figure 1: Successive Design Process (SC is the specification of Sequential Consistency, HC defines a protocol for systems with hierarchical caches, HCN is a refined version of HC with message passing, and HCN-base is a HCN-based cache coherence protocol that is free from deadlock and livelock)

caches, and prove that HC is a complete implementation of SC (Sections 4 & 5). Some derived rules of HC are discussed in Section 6. Then we define the HCN model by refining HC with message passing, and prove that HCN is a complete implementation of HC (Sections 7 & 8). This is followed by a discussion of potential optimizations of HCN in Section 9.

In Section 10 we start with a general discussion regarding the liveness issue, and then introduce directive messages and directive rules. We present HCN-base, a simple protocol derived from HCN, and show that the protocol implements Sequential Consistency and guarantees that each processor can always make progress (Sections 11 & 12). The design of HCN-base is completed in Section 13 with a buffer management policy that ensures fair message processing. Finally we present a summary and briefly discuss our research in progress.

2 The Formalism

Our formal framework is based on Term Rewriting Systems (TRS's). We use TRS's to specify the operational behavior of memory models and cache coherence protocols. A TRS consists of a set of terms and a set of rewriting rules. In the architectural context, the terms represent system states and the rules specify state transitions. The general structure of rewriting rules is as follows:

$$\begin{array}{ll} & s_1 \quad \textit{if} \quad p(s_1) \\ \longrightarrow & s_2 \end{array}$$

where s_1 and s_2 are terms, and $p(s_1)$ is an optional predicate about term s_1.

A rule can be used to rewrite a term if its left-hand-side pattern matches the term or one of its subterms, and the corresponding predicate is true. If several rules are applicable, then any one of them may be applied. If no rule is applicable, then the term cannot be rewritten any further and is said to be in *normal form*. Sometimes a rewriting strategy is used to specify which rule among the applicable rules should be applied to a term at every step.

We say term s_1 can be rewritten to term s_2 in one rewriting step ($s_1 \longrightarrow s_2$), if there exist a context C$[\![\ ]\!]$ and terms s_1' and s_2' such that $s_1 = \mathrm{C}[\![s_1']\!]$ and $s_2 = \mathrm{C}[\![s_2']\!]$, and s_1' can be rewritten to s_2' according to some rewriting rule. (A context is a term with a "hole" that can be filled by a term. C$[\![s]\!]$ refers to the term in which the hole is filled by term s). We say term s_1 can be rewritten to term s_2 in zero or more rewriting steps ($s_1 \twoheadrightarrow s_2$), if either $s_1 = s_2$, or there exists a term s' such that $s_1 \longrightarrow s'$ and $s' \twoheadrightarrow s_2$.

A term s is a legal term if there exists $s_0 \in S_0$ such that $s_0 \twoheadrightarrow s$. Since we are only interested in legal terms, we will drop the qualifier "legal" in our discussion.

A TRS is *confluent* if, for any term s_1, if $s_1 \twoheadrightarrow s_2$ and $s_1 \twoheadrightarrow s_3$, then there exists a term s_4 such that $s_2 \twoheadrightarrow s_4$ and $s_3 \twoheadrightarrow s_4$. A TRS is *strongly terminating* if, for any term, it can always be rewritten to a normal form using any rewriting strategy.

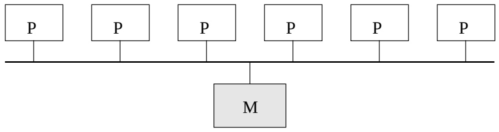
Notations: While pattern matching it is important to distinguish between variables and constants or data-structure constructors. A variable matches any expression while a constant or constructor matches only itself. Throughout the paper, we will follow the convention that variables are represented by identifiers with only lower-case letters, while constants and constructors are represented by either identifiers that begin with a capital letter, or special characters such as '$|$', '$\odot$', and '$\otimes$'. We use 'ϵ' to represent the empty term, and '-' the wild-card term that can match any term.

2.1 Correctness of an Implementation

The use of TRS's allows us to define and prove when a protocol implements a memory model correctly. The proof is based on showing that the TRS for the protocol admits only the observable behaviors that are permitted by the memory model. We say that system B is a *complete implementation* of system A if there exists a pair of mapping functions g ($B \mapsto A$) and f ($A \mapsto B$), such that

1. **Soundness:** $s_1 \overset{B}{\twoheadrightarrow} s_2 \quad \Longrightarrow \quad g(s_1) \overset{A}{\twoheadrightarrow} g(s_2)$;
2. **Completeness:** $s_1 \overset{A}{\twoheadrightarrow} s_2 \quad \Longrightarrow \quad f(s_1) \overset{B}{\twoheadrightarrow} f(s_2)$;
3. **Connection:** $g(f(s)) = s$.

The soundness property states that an implementation cannot take a step that is inconsistent with the specification, while the completeness property states that an implementation can imitate every possible step of the specification. Together these conditions can be interpreted as saying that the two systems can *simulate* each other. However, the correspondence between the implementation and the specification has not been properly confined with just these two conditions. For example, consider a function that maps all implementation terms to the same specification term. The connection property rules out such unreasonable mapping functions. The intuition behind this property is that an implementation term contains enough information to reconstruct the corresponding specification term. It is important to notice that the connection property is asymmetric, i.e., $f(g(s))$ does not necessarily equal to s. This is because an implementation term usually contains extra information that cannot be reconstructed once it is projected to a term in the specification.



SYS	≡	Sys(MEM, PG)	*System*
MEM	≡	ϵ [] Cell(a,v) | MEM	*Memory*
PG	≡	ϵ [] PROC | PG	*Processor Group*

Figure 2: The SC Model (Initially, the memory contains a cell for each address)

The notion of complete implementation is transitive, i.e., if system C is a complete implementation of system B, which is in turn a complete implementation of system A, then system C is a complete implementation of system B. This implies that the proof that a protocol is a complete implementation of a memory model can be carried out step-by-step throughout the successive design process. Needless to say any system is a complete implementation of itself.

Many real implementations are not complete according to the above definition. Any sound system can be regarded as a *partial implementation* of the specification. However, some partial implementations can be pretty silly in reality: for example, an implementation that has no rewrite rule and thus makes no transition is a partial implementation of any specification by the virtue of being sound.

3 The SC Model: Specification of Sequential Consistency

Intuitively, a system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appears in this sequence in the order specified by its program [17]. We take a slightly different approach and define Sequential Consistency operationally using a multiprocessor system based on a simple non-pipelined processor, which has no caches or write buffers and which executes instructions sequentially. The system is defined using a TRS called SC. All the protocols presented in this paper implement only those behaviors that are permitted by SC.

The grammar of the SC model is given in Figure 2. The system has two components, a memory and a processor group. The memory consists of a set of memory cells, where each memory cell has an address and a value. We assume addresses in a memory are pairwise distinct. The processor group consists of a set of processors where each processor has a program counter, a register file, and a program. The processor grammar and rules are presented in [24] and understanding them is not necessary to follow the rest of the paper as long as we remember that instructions are executed strictly according to the program order (the program counter holds the address of the instruction to be executed).

Notations: We use ‘[]’ as meta notation in grammars to separate disjuncts (SYS, MEM and PG are grammar symbols). Identifiers such as Sys, Cell and Proc are constructors. Notation prog[ia] refers to the instruction at instruction address ia in program prog. We use rf[r] to represent the content of register r in register file rf, and rf[r := v] to represent the register file which is the same as rf except that register r contains value v.

We use ‘|’ as an associative and commutative constructor ($s_1 \mid s_2 = s_2 \mid s_1$ and $s \mid \epsilon = s$). As we shall see, it can be used as a connective for terms such as processor groups, system groups and directories, which are intuitively associative and commutative.

The memory access rules are defined as follows:

SC-Load Rule

$$\begin{array}{ll} & \mathsf{Sys(Cell(a,v)} \mid \mathsf{m},\ \ \mathsf{Proc(ia,\ rf,\ prog)} \mid \mathsf{pg)} \quad \textit{if} \quad \mathsf{prog[ia]} = \ \mathsf{r} := \mathsf{Load(r_1)} \quad \textit{and} \quad \mathsf{a} = \mathsf{rf[r_1]} \\ \longrightarrow & \mathsf{Sys(Cell(a,v)} \mid \mathsf{m},\ \ \mathsf{Proc(ia{+}1,\ rf[r := v],\ prog)} \mid \mathsf{pg)} \end{array}$$

SC-Store Rule

$$\begin{array}{ll} & \mathsf{Sys(Cell(a,u)} \mid \mathsf{m},\ \ \mathsf{Proc(ia,\ rf,\ prog)} \mid \mathsf{pg)} \quad \textit{if} \quad \mathsf{prog[ia]} = \ \mathsf{Store(r_1, r_2)} \quad \textit{and} \quad \mathsf{a} = \mathsf{rf[r_1]} \\ \longrightarrow & \mathsf{Sys(Cell(a,v)} \mid \mathsf{m},\ \ \mathsf{Proc(ia{+}1,\ rf,\ prog)} \mid \mathsf{pg)} \quad \textit{where} \quad \mathsf{v} = \mathsf{rf[r_2]} \end{array}$$

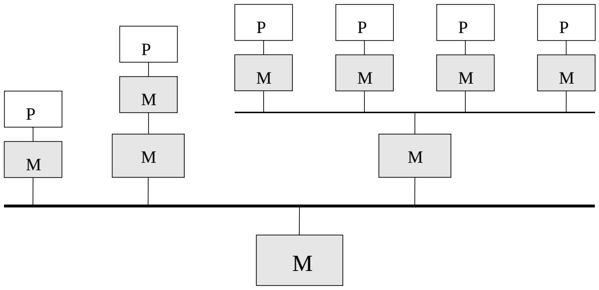
Since the connective ‘|’ is associative and commutative, any processor can be brought into the leftmost position in the processor group. Thus, if two processors intend to write to the same address, either can be allowed to proceed. Non-determinism can happen due to data races, however, memory access atomicity is guaranteed because there is no data replication and the Load and Store operations are performed directly on the memory.

We claim that SC define an operational semantics for Sequential Consistency although it has different flavor from the traditional definition. It is easy to show that a total instruction order, consistent with the program order for each individual processor, exists for all instructions. From now on we identify the range of behaviors admitted by Sequential Consistency as precisely the set of legal terms of SC. In the rest of the paper we will present several cache coherence protocols to implement Sequential Consistency and show that they admit only SC behaviors.

4 The HC Model: A System with Hierarchical Caches

A typical distributed memory system consists of multi-level caches and uses different implementation technologies and possibly different protocols in different parts of the system. In this section, we define a directory-based cache coherence protocol for a system with hierarchy of caches and call it the HC (Hierarchical Caches) model. In HC, we ignore the communication latency between memory sites, and assume that coherence actions involving two or more memory sites (e.g., a local read followed by a remote write) can be performed in one rewriting step.

The grammar of the HC model is given in Figure 3. The system has two components, a memory unit and an execution unit. The memory unit consists of an identifier and a memory. A memory is a set of memory cells, where each memory cell has an address, a value and a state used for coherence maintenance. The execution unit is either a single processor, or a



SYS	$\equiv$	Sys(MU, EU)	*System*
MU	$\equiv$	$\langle$id, MEM$\rangle$	*Memory Unit*
EU	$\equiv$	PROC [] SG	*Execution Unit*
SG	$\equiv$	ϵ [] SYS | SG	*System Group*
MEM	$\equiv$	ϵ [] Cell(a,v,STATE) | MEM	*Memory & Cache*
STATE	$\equiv$	(CSTATE, HSTATE)	*Cell's State*
CSTATE	$\equiv$	Sh [] Ex	*Cell's Cstate*
HSTATE	$\equiv$	R(DIR) [] W(id)	*Cell's Hstate*
DIR	$\equiv$	ϵ [] id | DIR	*Directory*

Figure 3: The HC Model (Initially, all memories except the outermost memory are empty; the outermost memory contains a cell for each address and the state of each cell is (Ex,R(ϵ)))

system group that consists of a set of systems. This recursive definition effectively allows arbitrary levels of cache hierarchy. Notice that although we show each memory as one block pictorially, in implementation addresses can be divided among multiple sites.

In the memory hierarchy, the memory at the root is called the outermost memory, and the memories that directly interface with processors are called innermost memories or L1 caches. Every memory except the innermost and outermost behaves simultaneously as a *cache* and *home*, that is, for its parent a memory is a cache which holds replicated data, and for its children it is the home where all the cells that have been cached by the children reside. Thus, we do not draw a distinction between "cache" and "memory", and use the two words interchangeably. Given a memory id, parent(id) represents its parent's identifier, children(id) the set of identifiers for its children, and siblings(id) the set of identifiers for its siblings.

4.1 State Encoding

Each memory cell contains an address, a value and a state for coherence maintenance. The state in each cell has two components, Cstate (cache state) and Hstate (home state). The Cstate is the "horizontal" state that indicates whether the cell is shared (Sh) or exclusive (Ex) with respect to its sibling caches. The Hstate is the "vertical" state that records which children have cached the data and for which purpose (i.e., for reading or writing). If the Hstate is R(dir), shared copies are cached in the children specified by the directory dir, which is a set of memory identifiers. If the Hstate is W(id), the child memory id has the exclusive copy for the address and can write into the cell.

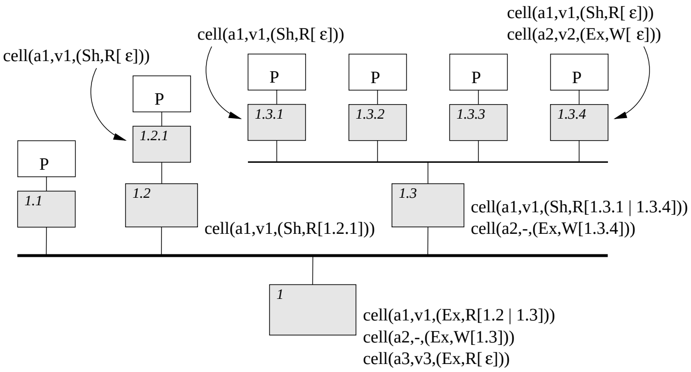


Figure 4: A Snapshot of Coherent States (For example, in memory $m_{1.3}$, the state for address a_1 is (Sh,R($m_{1.3.1} \mid m_{1.3.4}$)), indicating that the cell is a read-only copy and its child memories $m_{1.3.1}$ and $m_{1.3.4}$ have cached shared copies at the time; the state for address a_2 is (Ex,W($m_{1.3.4}$)), indicating that the cell is a read-write copy and the exclusive ownership has been given to child memory $m_{1.3.4}$)

The Hstate is always R(ϵ) for the cells in the innermost memories, because the innermost memories cannot have children. Similarly the Cstate is always Ex for the cells in the outermost memory, because it has no siblings. It is worth noting that (Sh,W(id)) is an illegal state, because a memory cannot give the write permission to any child unless it has obtained the exclusive ownership for that address. A snapshot of hierarchical caches in coherent states is shown in Figure 4.

Inclusion Invariants: The protocol can be implemented efficiently if, by checking a cell's state in a memory, it can be determined whether any further coherence actions need to be taken for its descendant memories. To accomplish this, HC maintains two invariants, namely *shared inclusion* and *exclusive inclusion*. The shared inclusion invariant states that, if a memory has a shared copy, its parent must have the address with the same value. The exclusive inclusion invariant states that, if a memory has an exclusive copy, its parent must have the address exclusively, although the value of the cell can be out-of-date.

4.2 Rewriting Rules

The rewriting rules of the HC model are all imperative rules and fall naturally into three categories: the rules for memory access operations (i.e., Load and Store), the rules for caching operations (i.e., moving data and/or coherence information from parents to children), and the rules for de-caching operations (i.e., moving data and/or coherence information from children to parents).

Memory Access Rules: Memory access operations by a processor are performed on its L1 cache. A Load instruction can execute if the data is cached in the L1 cache. A Store instruction can execute if the L1 cache has cached the address with the exclusive ownership.

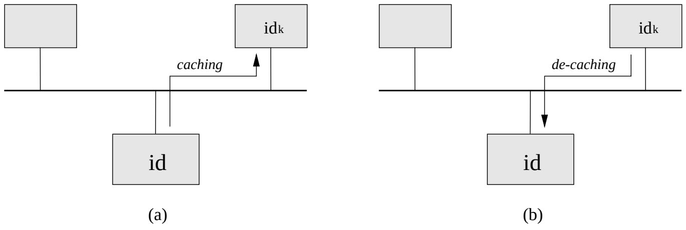


Figure 5: Caching and De-Caching Operations

HC-Load Rule

$\mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(cs,R(}\epsilon)))\mid\mathsf{m}\rangle,\ \ \mathsf{Proc}(\mathsf{ia,\ rf,\ prog}))$

$\quad$ *if* $\ \mathsf{prog[ia]} = \mathsf{r} := \mathsf{Load}(\mathsf{r}_1)$ *and* $\mathsf{a} = \mathsf{rf}[\mathsf{r}_1]$

$\longrightarrow\ \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(cs,R(}\epsilon)))\mid\mathsf{m}\rangle,\ \ \mathsf{Proc}(\mathsf{ia}+1,\ \mathsf{rf}[\mathsf{r} := \mathsf{v}],\ \mathsf{prog}))$

HC-Store Rule

$\mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,u,(Ex,R(}\epsilon)))\mid\mathsf{m}\rangle,\ \ \mathsf{Proc}(\mathsf{ia,\ rf,\ prog}))$

$\quad$ *if* $\ \mathsf{prog[ia]} = \mathsf{Store}(\mathsf{r}_1, \mathsf{r}_2)$ *and* $\mathsf{a} = \mathsf{rf}[\mathsf{r}_1]$

$\longrightarrow\ \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(Ex,R(}\epsilon)))\mid\mathsf{m}\rangle,\ \ \mathsf{Proc}(\mathsf{ia}+1,\ \mathsf{rf},\ \mathsf{prog}))$ *where* $\mathsf{v} = \mathsf{rf}[\mathsf{r}_2]$

Caching Rules: If the state of a cell in memory id is (-,R(dir)) and the directory dir shows that child $\mathsf{id_k}$ has not cached the data, then memory id can give a shared copy to memory $\mathsf{id_k}$ and record $\mathsf{id_k}$ in the directory. If the state of a cell in memory id is (Ex,R(ϵ)), then it can give an exclusive copy to child $\mathsf{id_k}$ and change the cell's Hstate to W($\mathsf{id_k}$). See Figure 5 (a).

Sh-Caching Rule

$\mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(cs,R(dir)))}\mid\mathsf{m}\rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{m_k}\rangle,\ \mathsf{eu_k})\mid\mathsf{sg})$ *if* $\mathsf{id_k} \notin \mathsf{dir}$

$\longrightarrow\ \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(cs,R(id_k|dir)))}\mid\mathsf{m}\rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{Cell}(\mathsf{a,v,(Sh,R(}\epsilon)))\mid\mathsf{m_k}\rangle,\ \mathsf{eu_k})\mid\mathsf{sg})$

Ex-Caching Rule

$\mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(Ex,R(}\epsilon)))\mid\mathsf{m}\rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{m_k}\rangle,\ \mathsf{eu_k})\mid\mathsf{sg})$

$\longrightarrow\ \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(Ex,W(id_k)))}\mid\mathsf{m}\rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{Cell}(\mathsf{a,v,(Ex,R(}\epsilon)))\mid\mathsf{m_k}\rangle,\ \mathsf{eu_k})\mid\mathsf{sg})$

De-Caching Rules: If the state of a cell in memory $\mathsf{id_k}$ is (Ex,R(dir)), then it can write the most up-to-date data back to the parent and change the cell's Cstate to Sh. If the state of a cell in memory $\mathsf{id_k}$ is (Sh,R(ϵ)), then it can invalidate the cell and delete identifier $\mathsf{id_k}$ from the corresponding directory in the parent. See Figure 5 (b).

Writeback Rule

$\mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,u,(Ex,W(id_k)))}\mid\mathsf{m}\rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{Cell}(\mathsf{a,v,(Ex,R(dir)))}\mid\mathsf{m_k}\rangle,\ \mathsf{eu_k})\mid\mathsf{sg})$

$\longrightarrow\ \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(Ex,R(id_k)))}\mid\mathsf{m}\rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{Cell}(\mathsf{a,v,(Sh,R(dir)))}\mid\mathsf{m_k}\rangle,\ \mathsf{eu_k})\mid\mathsf{sg})$

Invalidate Rule

$\mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(cs,R(id_k|dir)))}\mid\mathsf{m}\rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{Cell}(\mathsf{a,v,(Sh,R(}\epsilon)))\mid\mathsf{m_k}\rangle,\ \mathsf{eu_k})\mid\mathsf{sg})$

$\longrightarrow\ \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(cs,R(dir)))}\mid\mathsf{m}\rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{m_k}\rangle,\ \mathsf{eu_k})\mid\mathsf{sg})$

5 Verification of the HC Model

We can prove that the HC model completely implements the SC model. The proof consists of three steps:

1. **Soundness:** Define cache-flush function *CF* (HC $\mapsto$ SC), and show
$$s_1 \overset{\mathsf{HC}}{\twoheadrightarrow} s_2 \quad \Longrightarrow \quad CF(s_1) \overset{\mathsf{SC}}{\twoheadrightarrow} CF(s_2);$$
2. **Completeness:** Define cache-lift function *CL* (SC $\mapsto$ HC), and show
$$s_1 \overset{\mathsf{SC}}{\twoheadrightarrow} s_2 \quad \Longrightarrow \quad CL(s_1) \overset{\mathsf{HC}}{\twoheadrightarrow} CL(s_2);$$
3. **Connection:** For any SC term s, show $CF(CL(s)) = s$.

The *CF* function is easy to define once we notice that we can apply the de-caching rules repeatedly to empty all non-outermost caches and propagate all valid values to the outermost memory. To show soundness, we prove that if $s_1 \longrightarrow s_2$ by applying some rule α in HC, then in SC, either $CF(s_1) = CF(s_2)$ if α is a caching or de-caching rule; or $CF(s_1) \longrightarrow CF(s_2)$ (by applying some memory access rule) if α is a memory access rule.

The *CL* function is based on the simple observation that a SC term can be "lifted" to a HC term by introducing empty caches in the memory hierarchy, setting the state for each outermost memory cell to (Ex,R(ϵ)), and assigning memory identifiers for all memory units. It is easy to show that each SC rule can be simulated by a sequence of HC rules.

5.1 Inclusion Invariants

The HC model maintains two inclusion invariants. The shared inclusion invariant states that, if a memory has a shared copy, the parent must have the address with the same value. The exclusive inclusion invariant states that, if a memory has an exclusive copy, the parent must have the address and the ownership, albeit the value of the cell can be out-of-date. It is because of these invariants that the cache coherence protocol can determine, simply by checking the a cell's state, if further coherence actions need to be taken for its descendant memories.

Lemma 1 (Inclusion Invariants) In any HC term $\mathsf{Sys}(\langle \mathsf{id},\ \mathsf{m}\rangle,\ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{m_k}\rangle,\ \mathsf{eu_k}) \mid \mathsf{sg})$,

- $\mathsf{Cell(a,v,(Sh,\text{-}))} \in \mathsf{m_k} \quad \Longrightarrow \quad \mathsf{Cell(a,v,(\text{-},R(id_k} \mid \text{-} \mathsf{)))} \in \mathsf{m}$
- $\mathsf{Cell(a,v,(Ex,\text{-}))} \in \mathsf{m_k} \quad \Longrightarrow \quad \mathsf{Cell(a,\text{-},(Ex,W(id_k)))} \in \mathsf{m}$

Proof The proof is by induction on rewriting steps. The invariants hold trivially for the initial terms where all non-outermost memories are empty. It can be shown by checking each rewriting rule that, if the invariants hold for a term, then they still hold after the term is rewritten according to that rule. □

It can be shown that no memory can contain two cells that have the same address. This is because only caching rules can create new cells (in the child memory). According to the Inclusion Invariants, if a caching rule is applicable, the child memory cannot have a cell that has the same address as the cell that is to be created.

5.2 Cache Flushing Property

Suppose we define a new rewriting system $\mathsf{R_{CF}}$, which has the same grammar as HC but uses only the de-caching rules.

Definition 2 (TRS for cache flushing) $\mathsf{R_{CF}} \equiv \{$ *Writeback, Invalidate* $\}$

Now we discuss some properties of the $\mathsf{R_{CF}}$ system.

Lemma 3 $\mathsf{R_{CF}}$ is strongly terminating and confluent, i.e., for any HC term, rewriting with respect to $\mathsf{R_{CF}}$ terminates within a finite number of steps and always reaches the same normal form, regardless of the order in which the rules are applied.

Proof The termination is obvious because $\mathsf{R_{CF}}$ includes only the de-caching rules that can either invalidate Sh cells or degrade Ex cells to Sh cells, but can never create cells or upgrade Sh cells to Ex cells. The confluence follows from the fact that the de-caching rules do not interfere each other. □

Definition 4 For any HC term s, $NF_{CF}(s)$ is the normal form of s in $\mathsf{R_{CF}}$.

Lemma 5 (Cache Flushing Property) For any HC term s, all non-outermost memories are empty in $NF_{CF}(s)$.

Proof Suppose not all non-outermost memories are empty. Consider a non-outermost memory which is not empty and whose descendant memories (if any) are all empty. According to the Inclusion Invariants, either the *Writeback* rule or the *Invalidate* rule would apply, and hence the term cannot be a normal form. □

Lemma 6 (Value Preservation Property) For any HC term s, if Cell(a,v,(-,R(-))) is in some cache of s, then Cell(a,v,-) is in the outermost memory of $NF_{CF}(s)$.

Proof This can be shown by checking each de-caching rule.

- When *Writeback* applies, Cell(a,v,(Ex,R(-))) is in the child and Cell(a,u,(Ex,W(-))) is in the parent; after the writeback, Cell(a,v,(Ex,R(-))) is in the parent memory (i.e., value v is preserved).
- When *Invalidate* applies, Cell(a,v,(Sh,R(ϵ))) is in the child and Cell(a,v,(-,R(-))) is in the parent; after the invalidation, Cell(a,v,(-,R(-))) is in the parent memory (i.e., value v is preserved).

By induction, if Cell(a,v,(-,R(-))) is in some cache, Cell(a,v,-) will be in the outermost memory after applying the de-caching rules repeatedly till all non-outermost caches are emptied. □

The Value Preservation Property shows that we can propagate all valid values to the outermost memory by repeatedly applying the de-caching rules. This is the key idea behind the cache-flush function used in the soundness proof. It should be noted that $NF_{CF}(s)$ also preserves all processor states including program counters, register files and programs.

5.3 Soundness of HC

We define function *CF* (cache-flush) that maps HC terms to SC terms as follows:

Definition 7 (Cache-flush function) For any HC term s, $CF(s)$ is the projection of $NF_{CF}(s)$ on the corresponding SC term where all non-outermost memory units have been deleted along with the coherence states of all the cells in the outermost memory and the identifier for the outermost memory.

Lemma 8 (Soundness) $s_1 \xrightarrow{\mathsf{HC}}\!\!\!\!\rightarrow s_2 \quad\Longrightarrow\quad CF(s_1) \xrightarrow{\mathsf{SC}}\!\!\!\!\rightarrow CF(s_2)$.

Proof The proof is by induction on rewriting steps. We give the proof for one rewriting step. Assume $s_1 \longrightarrow s_2$ by applying some rule α in HC. The proof is based on the case analysis on α.

- $\alpha \in \mathsf{R_{CF}}$. Needless to say $CF(s_1) = CF(s_2)$.
- α is *Sh-Caching*. Then $s_2 \longrightarrow s_1$ by applying *Invalidate*, hence $CF(s_1) = CF(s_2)$.
- α is *Ex-Caching*. Then $s_2 \longrightarrow\!\!\!\!\rightarrow s_1$ by applying *Writeback* followed by *Invalidate*, hence $CF(s_1) = CF(s_2)$.
- α is *HC-Load*. Let Cell(a,v,(-,R(ϵ))) be the cell in the L1 cache. Then Cell(a,v) is preserved in both $CF(s_1)$ and $CF(s_2)$; and $CF(s_1)$ differs from $CF(s_2)$ only in the program counter and the register modified by the load operation. Thus $CF(s_1) \longrightarrow CF(s_2)$ by applying *SC-Load*.
- α is *HC-Store*. Let Cell(a,u,(Ex,R(ϵ))) and Cell(a,v,(Ex,R(ϵ))) be the cell in the L1 cache of s_1 and s_2, respectively. Then Cell(a,u) and Cell(a,v) are preserved in $CF(s_1)$ and $CF(s_2)$, respectively; and $CF(s_1)$ is the same as $CF(s_2)$ except for the program counter and the cell modified by the store operation. Thus $CF(s_1) \longrightarrow CF(s_2)$ by applying *SC-Store*. □

One thing that is still missing from the definition of *CF* is that we have not shown that the function always maps legal HC terms to legal SC terms. The soundness of HC guarantees this, noting that *CF* maps the initial HC term to the initial SC term.

5.4 Completeness of HC

We define function *CL* (cache-lift) that maps SC terms to corresponding HC terms. Suppose the memory hierarchy structure of the HC system is known. For any SC term s, $CL(s)$ is defined by introducing empty caches in the memory hierarchy, setting the state for each outermost memory cell to (Ex,R(ϵ)), and assigning memory identifiers for all memories.

It is easy to show that each SC rule can be simulated by a sequence of HC rules. For example, to simulate the *SC-Load* rule, we can apply *Sh-Caching* n times to propagate a shared copy to the corresponding L1 cache, then apply *HC-Load* to read the data from the L1 cache, and then apply *Invalidate* n times to purge the shared copies from the L1 cache and all caches between the L1 cache and the outermost memory. Here n is the L1 cache's depth, counting from the outermost memory whose depth is 0.

Lemma 9 (Completeness) $s_1 \overset{\mathsf{SC}}{\twoheadrightarrow} s_2 \quad \Longrightarrow \quad CL(s_1) \overset{\mathsf{HC}}{\twoheadrightarrow} CL(s_2)$.

It is trivial to show that *CF* is the inverse function of *CL*.

Lemma 10 (Connection) For any SC term s, $CF(CL(s)) = s$.

This completes the proof that HC is a complete implementation of SC.

Theorem 11 The HC model completely implements the SC model.

6 Some Derived Rules of the HC Model

A derived rule is one that can be derived from other rules of the TRS. A derived rule can simply be an existing rule but with more stringent predicate, or a sequential combination of several other rules. Adding derived rules cannot affect the correctness of the system, but may improve the performance by some measure.

6.1 Pushout

The pushout operation allows a memory to write the most up-to-date data of an exclusive cell back to the parent memory and invalidate the cell in one rewriting step, if the cell is not cached by any child (i.e., the cell's state is $(\mathsf{Ex},\mathsf{R}(\epsilon))$).

Pushout Rule

$$\begin{array}{ll} & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{u},(\mathsf{Ex},\mathsf{W}(\mathsf{id}_k))) \mid \mathsf{m}\rangle,\ \ \mathsf{Sys}(\langle \mathsf{id}_k,\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Ex},\mathsf{R}(\epsilon))) \mid \mathsf{m}_k\rangle,\ \mathsf{eu}_k) \mid \mathsf{sg}) \\ \longrightarrow & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Ex},\mathsf{R}(\epsilon))) \mid \mathsf{m}\rangle,\ \ \mathsf{Sys}(\langle \mathsf{id}_k,\ \mathsf{m}_k\rangle,\ \mathsf{eu}_k) \mid \mathsf{sg}) \end{array}$$

Obviously applying *Pushout* has the same effect as applying *Writeback* and *Invalidate* consecutively. We can define another model which is the same as HC except that the *Writeback* rule is replaced by the *Pushout* rule. In this new model, *Writeback* can be treated as a derived rule (*Pushout* followed by *Sh-Caching*). It can be shown that these two TRS's are equivalent.

6.2 Upgrade

The upgrade operation allows a memory to obtain the exclusive ownership for a shared cell in one rewriting step, if its parent has the exclusive ownership and has not given the data to any other child. Upgrade is also known as Dclaim.

Upgrade Rule

$$\begin{array}{ll} & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Ex},\mathsf{R}(\mathsf{id}_k))) \mid \mathsf{m}\rangle,\ \ \mathsf{Sys}(\langle \mathsf{id}_k,\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Sh},\mathsf{R}(\mathsf{dir}))) \mid \mathsf{m}_k\rangle,\ \mathsf{eu}_k) \mid \mathsf{sg}) \\ \longrightarrow & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Ex},\mathsf{W}(\mathsf{id}_k))) \mid \mathsf{m}\rangle,\ \ \mathsf{Sys}(\langle \mathsf{id}_k,\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Ex},\mathsf{R}(\mathsf{dir}))) \mid \mathsf{m}_k\rangle,\ \mathsf{eu}_k) \mid \mathsf{sg}) \end{array}$$

It can be shown that applying *Upgrade* has the same effect as applying *Invalidate* $n+1$ times, followed by *Ex-Caching*, followed by *Sh-Caching* n times, where n is the number of shared copies (of the same address) cached in the descendant memories ($n \geq 0$). Here the *Invalidate* rule is applied repeatedly to invalidate the shared copies in the descendant memories before the shared copy in the memory itself can be invalidated, and the *Sh-Caching* rule is applied repeatedly to propagate shared copies to those descendant memories in which shared copies have just been invalidated.

6.3 Forward

If the state of a cell in a memory is $\mathsf{(Ex,R(dir))}$, then it can write the most up-to-date data back to its parent and forward a shared copy to some sibling memory. Similarly, if the state of a cell in a memory is $\mathsf{(Ex,R(}\epsilon\mathsf{))}$, then it can invalidate the cell and forward the exclusive copy to some sibling memory. Forward is also known as Intervention.

Sh-Forward Rule

$$\begin{aligned}
&\qquad \mathsf{Sys(\langle id,\ Cell(a,u,(Ex,W(id_k)))\mid m\rangle,}\\
&\qquad\qquad\qquad \mathsf{Sys(\langle id_k,\ Cell(a,v,(Ex,R(dir)))\mid m_k\rangle,\ eu_k)\ \mid\ Sys(\langle id_j,\ m_j\rangle,\ eu_j)\ \mid\ sg)}\\
&\longrightarrow\ \ \mathsf{Sys(\langle id,\ Cell(a,v,(Ex,R(id_k|id_j)))\mid m\rangle,}\\
&\qquad\qquad\qquad \mathsf{Sys(\langle id_k,\ Cell(a,v,(Sh,R(dir)))\mid m_k\rangle,\ eu_k)\ \mid\ Sys(\langle id_j,\ Cell(a,v,(Sh,R(}\epsilon\mathsf{)))\mid m_j\rangle,\ eu_j)\ \mid\ sg)}
\end{aligned}$$

Ex-Forward Rule

$$\begin{aligned}
&\qquad \mathsf{Sys(\langle id,\ Cell(a,u,(Ex,W(id_k)))\mid m\rangle,}\\
&\qquad\qquad\qquad \mathsf{Sys(\langle id_k,\ Cell(a,v,(Ex,R(}\epsilon\mathsf{)))\mid m_k\rangle,\ eu_k)\ \mid\ Sys(\langle id_j,\ m_j\rangle,\ eu_j)\ \mid\ sg)}\\
&\longrightarrow\ \ \mathsf{Sys(\langle id,\ Cell(a,v,(Ex,W(id_j)))\mid m\rangle,}\\
&\qquad\qquad\qquad \mathsf{Sys(\langle id_k,\ m_k\rangle,\ eu_k)\ \mid\ Sys(\langle id_j,\ Cell(a,v,(Ex,R(}\epsilon\mathsf{)))\mid m_j\rangle,\ eu_j)\ \mid\ sg)}
\end{aligned}$$

Obviously applying *Sh-Forward* has the same effect as applying *Writeback* followed by *Sh-Caching*, while applying *Ex-Forward* has the same effect as applying *Writeback* and *Invalidate* followed by *Ex-Caching*.

For exclusive forwarding, it is unnecessary to update the cell's value in the parent memory with the most up-to-date data. The cell's value cannot be used before it is overwritten later (maybe with the same value). Based on this, we can optimize *Ex-Forward* as follows:

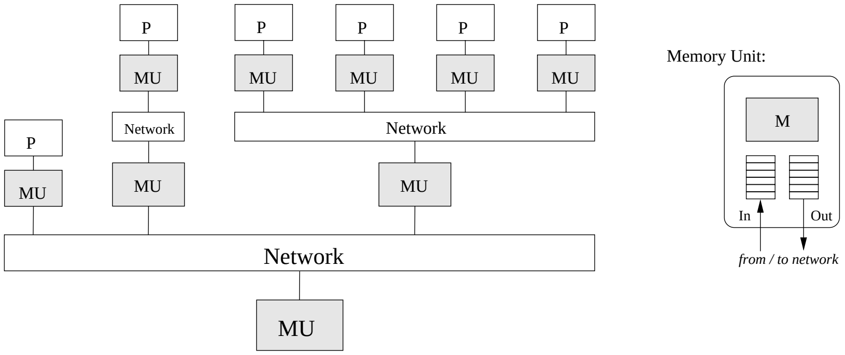
Ex-Forward-Without-Writeback Rule

$$\begin{aligned}
&\qquad \mathsf{Sys(\langle id,\ Cell(a,u,(Ex,W(id_k)))\mid m\rangle,}\\
&\qquad\qquad\qquad \mathsf{Sys(\langle id_k,\ Cell(a,v,(Ex,R(}\epsilon\mathsf{)))\mid m_k\rangle,\ eu_k)\ \mid\ Sys(\langle id_j,\ m_j\rangle,\ eu_j)\ \mid\ sg)}\\
&\longrightarrow\ \ \mathsf{Sys(\langle id,\ Cell(a,u,(Ex,W(id_j)))\mid m\rangle,}\\
&\qquad\qquad\qquad \mathsf{Sys(\langle id_k,\ m_k\rangle,\ eu_k)\ \mid\ Sys(\langle id_j,\ Cell(a,v,(Ex,R(}\epsilon\mathsf{)))\mid m_j\rangle,\ eu_j)\ \mid\ sg)}
\end{aligned}$$

It is worth noting that *Ex-Forward-Without-Writeback* is not a derived rule, since it allows terms that are illegal in HC. However, extending HC with this new rule cannot result in any illegal program behavior that violates Sequential Consistency. It can be proved that the extended system is still a complete implementation of SC.

7 The HCN Model: Refining HC with Message Passing

The HC model assumes that coherence actions involving two or more memory units can be performed with one rewriting step. For example, the *Ex-Caching* rule states that if a memory has a cell whose state is $\mathsf{(Ex,R(}\epsilon\mathsf{))}$, it can send an exclusive copy to some child and



SYS	$\equiv$	Sys(MU, EU)	*System*
MU	$\equiv$	$\langle$id, MEM, INQ, OUTQ$\rangle$	*Memory Unit*
EU	$\equiv$	PROC $[\!]$ SG	*Execution Unit*
SG	$\equiv$	ϵ $[\!]$ SYS $\mid$ SG	*System Group*
MEM	$\equiv$	ϵ $[\!]$ Cell(a,v,STATE) $\mid$ MEM	*Memory & Cache*
STATE	$\equiv$	(CSTATE, HSTATE)	*Cell's State*
CSTATE	$\equiv$	Sh $[\!]$ Ex	*Cell's Cstate*
HSTATE	$\equiv$	R(DIR) $[\!]$ W(id)	*Cell's Hstate*
DIR	$\equiv$	ϵ $[\!]$ id $\mid$ DIR	*Directory*
INQ	$\equiv$	ϵ $[\!]$ MSG $\odot$ INQ	*Incoming Queue*
OUTQ	$\equiv$	ϵ $[\!]$ MSG $\otimes$ OUTQ	*Outgoing Queue*
MSG	$\equiv$	Msg($\mathsf{id}_{\mathrm{src}}$, $\mathsf{id}_{\mathrm{dest}}$, CMD, a, v)	*Protocol Message*
CMD	$\equiv$	Sh-rep $[\!]$ Ex-rep $[\!]$ Wb-rep $[\!]$ Inv-rep	*Message Command*

Figure 6: Grammar of the HCN Model (Initially, all non-outermost memories, and all incoming and outgoing message queues are empty; the outermost memory contains a cell for each address and the state of each cell is (Ex,R(ϵ)))

the child will receive the data and cache the cell in the (Ex,R(ϵ)) state. All this happens atomically with respect to other components of the system. Such rules are considered non-local in DSM systems where memory units communicate each other via message passing and the communication latency cannot be ignored. The caching and de-caching rules of HC are all non-local rules. Without special hardware support, it is expensive and difficult to ensure the atomicity of coherence actions such as a local read followed by a remote write.

In this section, we define the HCN model (HC with Network) by incorporating a message passing network to HC. The grammar of HCN is given in Figure 6. Each memory unit has two new components, an incoming message queue and an outgoing message queue. The queue constructors '$\odot$' and '$\otimes$' are associative and commutative, allowing non-FIFO message passing and non-blocking message processing. In HCN, each rewriting rule except message passing rules can examine and/or update only the local memory unit.

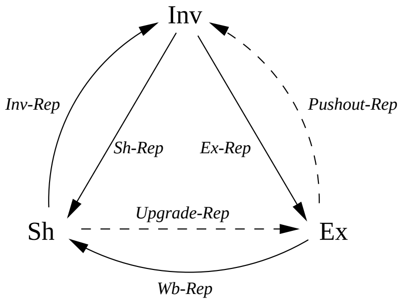


Figure 7: Relationship of Cache State Transitions and Reply Messages (Inv represents the state that the address is not cached; Pushout-rep and Upgrade-rep are potential optimizations)

7.1 Messages and Message Queues

In HCN, all protocol messages are imperative messages. A message has five fields: source, destination, command, address and data (which can be '$\perp$' if the message carries no data). There are four types of message commands: Sh-rep, Ex-rep, Wb-rep and Inv-rep, where the suffix '-rep' stands for 'reply', because such messages are usually, although not necessarily, issued upon requests (this will become clear later when request messages are introduced). Figure 7 shows the cache state transition that can happen when a reply message is received and processed.

We use outgoing queues to characterize certain properties of the network. The constructor '$\otimes$' is associative and commutative, indicating any message in an outgoing queue to be brought to the front of the queue. This effectively models general non-FIFO networks which enforce no restriction on the order in which messages are delivered. With this associativity and commutativity, messages issued from the same source can arrive at their destinations in arbitrary order, even when the destination is also the same.

Ideally we would like to treat incoming queues as FIFOs and process incoming messages in the order in which they are received. However, this may cause deadlock or livelock unless messages that cannot be processed temporarily are properly buffered so that other messages can be processed first. To avoid this complication, we assume that the constructor '$\odot$' is associative and commutative, thus incoming messages can be processed in arbitrary order since any message in an incoming queue can be brought to the front of the queue if necessary.

The adoption of associative and commutative queues for incoming messages allows us to treat buffer management as a separate issue that is transparent to imperative and directive rules. This simplifies the protocol design, because messages always appear to arrive in the order in which they are to be processed, and scenarios involving message reordering become irrelevant. Our experience shows that exposing buffer management at early design stages is inappropriate, since it could give rise to a bloated set of rewriting rules and dramatically complicate the protocol verification. Throughout the successive design process, buffer management should not be considered until all imperative and directive rules are defined; and its goal is purely to reorder incoming messages whenever necessary. We will revisit the buffer management issue in Section 13.

7.2 Rewriting Rules

The derivation of HCN rules from the HC rules is quite straightforward: each caching and de-caching rule in HC becomes a pair of rules for sending and receiving messages; and two message passing rules are introduced for passing messages between parent and child memories. The memory access rules remain unaffected (they are considered local rules because the processor and its L1 cache are tightly coupled and coherence actions involving the both appear to be atomic).

In HCN, a caching or de-caching operation is performed in three steps: the source site sends a message to its outgoing queue; the network transfers the message to the corresponding destination; and the destination site receives the message from its incoming queue.

Memory Access Rules: Memory access operations by a processor are performed on its L1 cache, and the message queues are not affected.

HCN-Load Rule

$$\begin{array}{ll} & \text{Sys}(\langle\text{id, Cell(a,v,(cs,R}(\epsilon)))\,|\,\text{m, in, out}\rangle,\ \ \text{Proc(ia, rf, prog)}) \\ & \qquad\qquad \textit{if}\ \ \text{prog[ia]} = \text{r} := \text{Load}(\text{r}_1)\ \ \textit{and}\ \ \text{a} = \text{rf}[\text{r}_1] \\ \longrightarrow & \text{Sys}(\langle\text{id, Cell(a,v,(cs,R}(\epsilon)))\,|\,\text{m, in, out}\rangle,\ \ \text{Proc(ia+1, rf[r}:=\text{v], prog)}) \end{array}$$

HCN-Store Rule

$$\begin{array}{ll} & \text{Sys}(\langle\text{id, Cell(a,u,(Ex,R}(\epsilon)))\,|\,\text{m, in, out}\rangle,\ \ \text{Proc(ia, rf, prog)}) \\ & \qquad\qquad \textit{if}\ \ \text{prog[ia]} = \text{Store}(\text{r}_1,\text{r}_2)\ \ \textit{and}\ \ \text{a} = \text{rf}[\text{r}_1] \\ \longrightarrow & \text{Sys}(\langle\text{id, Cell(a,v,(Ex,R}(\epsilon)))\,|\,\text{m, in, out}\rangle,\ \ \text{Proc(ia+1, rf, prog)}) \quad \textit{where}\ \ \text{v} = \text{rf}[\text{r}_2] \end{array}$$

Sh-Caching Rules: If the state of a cell in memory id is (-,R(dir)), and the directory dir shows that the data is not cached in child id_k, then memory id can send a Sh-rep message to the child to give it a shared copy. When the Sh-rep message arrives, memory id_k caches the data in the (Sh,R(ϵ)) state.

Send-Sh-Rep Rule

$$\begin{array}{ll} & \langle\text{id, Cell(a,v,(cs,R(dir)))}\,|\,\text{m, in, out}\rangle \quad \textit{if}\ \ \text{id}_\text{k} \in \text{children(id)} \quad \textit{and}\ \ \text{id}_\text{k} \notin \text{dir} \\ \longrightarrow & \langle\text{id, Cell(a,v,(cs,R(id}_\text{k}|\text{dir)))}\,|\,\text{m, in, out} \otimes \text{Msg(id,id}_\text{k}\text{,Sh-rep,a,v)}\rangle \end{array}$$

Receive-Sh-Rep Rule

$$\begin{array}{ll} & \langle\text{id}_\text{k},\ \text{m}_\text{k},\ \text{Msg(id,id}_\text{k}\text{,Sh-rep,a,v)} \odot \text{in}_\text{k},\ \text{out}_\text{k}\rangle \\ \longrightarrow & \langle\text{id}_\text{k},\ \text{Cell(a,v,(Sh,R}(\epsilon)))\,|\,\text{m}_\text{k},\ \text{in}_\text{k},\ \text{out}_\text{k}\rangle \end{array}$$

Ex-Caching Rules: If the state of a cell in memory id is (Ex,R(ϵ)), then it can send an Ex-rep message to child id_k to give it an exclusive copy. When the Ex-rep message arrives, memory id_k caches the data in the (Ex,R(ϵ)) state.

Send-Ex-Rep Rule

$$\begin{array}{ll} & \langle\text{id, Cell(a,v,(Ex,R}(\epsilon)))\,|\,\text{m, in, out}\rangle \\ \longrightarrow & \langle\text{id, Cell(a,v,(Ex,W(id}_\text{k})))\,|\,\text{m, in, out} \otimes \text{Msg(id,id}_\text{k}\text{,Ex-rep,a,v)}\rangle \quad \textit{where}\ \ \text{id}_\text{k} \in \text{children(id)} \end{array}$$

Receive-Ex-Rep Rule

$$\begin{array}{ll} & \langle\text{id}_\text{k},\ \text{m}_\text{k},\ \text{Msg(id,id}_\text{k}\text{,Ex-rep,a,v)} \odot \text{in}_\text{k},\ \text{out}_\text{k}\rangle \\ \longrightarrow & \langle\text{id}_\text{k},\ \text{Cell(a,v,(Ex,R}(\epsilon)))\,|\,\text{m}_\text{k},\ \text{in}_\text{k},\ \text{out}_\text{k}\rangle \end{array}$$

Writeback Rules: If the state of a cell in memory $\mathsf{id_k}$ is $(\mathsf{Ex},\mathsf{R(dir)})$, then it can send a $\mathsf{Wb\text{-}rep}$ message to its parent id to write the most up-to-date data back to the home. When the $\mathsf{Wb\text{-}rep}$ message is received, memory id updates the cell's value and changes the cell's Hstate from $\mathsf{W(id_k)}$ to $\mathsf{R(id_k)}$.

Send-Wb-Rep Rule

$$\begin{array}{ll} & \langle \mathsf{id_k},\ \mathsf{Cell(a,v,(Ex,R(dir)))} \mid \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k} \rangle \\ \longrightarrow & \langle \mathsf{id_k},\ \mathsf{Cell(a,v,(Sh,R(dir)))} \mid \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k} \otimes \mathsf{Msg(id_k,id,Wb\text{-}rep,a,v)} \rangle \quad \textit{where} \quad \mathsf{id} = \mathsf{parent(id_k)} \end{array}$$

Receive-Wb-Rep Rule

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{Cell(a,u,(Ex,W(id_k)))} \mid \mathsf{m},\ \mathsf{Msg(id_k,id,Wb\text{-}rep,a,v)} \odot \mathsf{in},\ \mathsf{out} \rangle \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{Cell(a,v,(Ex,R(id_k)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out} \rangle \end{array}$$

Invalidate Rules: If state of a cell in memory $\mathsf{id_k}$ is $(\mathsf{Sh},\mathsf{R}(\epsilon))$, then it can purge the cell and send an $\mathsf{Inv\text{-}rep}$ message to its parent id to notify the home of the invalidation. When the $\mathsf{Inv\text{-}rep}$ message is received, memory id removes identifier $\mathsf{id_k}$ from the directory.

Send-Inv-Rep Rule

$$\begin{array}{ll} & \langle \mathsf{id_k},\ \mathsf{Cell(a,v,(Sh,R(}\epsilon\mathsf{)))} \mid \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k} \rangle \\ \longrightarrow & \langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k} \otimes \mathsf{Msg(id_k,id,Inv\text{-}rep,a,}\perp\mathsf{)} \rangle \quad \textit{where} \quad \mathsf{id} = \mathsf{parent(id_k)} \end{array}$$

Receive-Inv-Rep Rule

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{Cell(a,v,(cs,R(id_k|dir)))} \mid \mathsf{m},\ \mathsf{Msg(id_k,id,Inv\text{-}rep,a,}\perp\mathsf{)} \odot \mathsf{in},\ \mathsf{out} \rangle \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{Cell(a,v,(cs,R(dir)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out} \rangle \end{array}$$

Message Passing Rules: Messages passing can happen only between the memories that have the parent-child relationship. The *Message-Passing-To-Child* rule transfers a message from the parent's outgoing queue to the child's incoming queue; while the *Message-Passing-To-Parent* rule transfers a message from the child's outgoing queue to the parent's incoming queue. Since the constructor '⊗' is associative and commutative, messages in an outgoing queue can be chosen to deliver in any order.

Message-Passing-To-Child Rule

$$\begin{array}{ll} & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{Msg(id,id_k,cmd,a,v)} \otimes \mathsf{out} \rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k} \rangle,\ \mathsf{eu_k}) \mid \mathsf{sg}) \\ \longrightarrow & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out} \rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{in_k} \odot \mathsf{Msg(id,id_k,cmd,a,v)},\ \mathsf{out_k} \rangle,\ \mathsf{eu_k}) \mid \mathsf{sg}) \end{array}$$

Message-Passing-To-Parent Rule

$$\begin{array}{ll} & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out} \rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{Msg(id_k,id,cmd,a,v)} \otimes \mathsf{out_k} \rangle,\ \mathsf{eu_k}) \mid \mathsf{sg}) \\ \longrightarrow & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{m},\ \mathsf{in} \odot \mathsf{Msg(id_k,id,cmd,a,v)},\ \mathsf{out} \rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k} \rangle,\ \mathsf{eu_k}) \mid \mathsf{sg}) \end{array}$$

Still one scenario deserves a bit more discussion. Suppose a cell in $(\mathsf{Ex},\mathsf{R}(\epsilon))$ state performs a writeback operation followed by an invalidate operation. With non-FIFO message passing, the $\mathsf{Inv\text{-}rep}$ message may arrive at the parent before the $\mathsf{Wb\text{-}rep}$ message. If incoming messages were required to be processed in the FIFO order, deadlock could happen because the $\mathsf{Inv\text{-}rep}$ message cannot be processed before the $\mathsf{Wb\text{-}rep}$ message is processed. However, this will not occur, since messages in the incoming queue can be examined in any order, which allows the $\mathsf{Wb\text{-}rep}$ message to be processed first.

8 Verification of the HCN Model

It can be shown that the HCN model completely implements the HC model. The proof consists of three steps:

1. **Soundness:** Define queue-flush function QF (HCN ↦ HC), and show
$$s_1 \overset{\mathsf{HCN}}{\twoheadrightarrow} s_2 \quad\Longrightarrow\quad QF(s_1) \overset{\mathsf{HC}}{\twoheadrightarrow} QF(s_2);$$
2. **Completeness:** Define queue-lift function QL (HC ↦ HCN), and show
$$s_1 \overset{\mathsf{HC}}{\twoheadrightarrow} s_2 \quad\Longrightarrow\quad QL(s_1) \overset{\mathsf{HCN}}{\twoheadrightarrow} QL(s_2);$$
3. **Connection:** For any HC term s, show $QF(QL(s)) = s$.

The QF function is defined based on the intuition that, in HCN, when only message passing and message receive rules are applied, all message queues will eventually become empty. To show soundness, we prove that if $s_1 \longrightarrow s_2$ by applying some rule α in HCN, then in HC, either $QF(s_1) = QF(s_2)$ if α is a message passing or message receive rule; or $QF(s_1) \longrightarrow QF(s_2)$ by applying an appropriate HC rule if α is a memory access or message send rule.

The queue-lift function QL maps HC terms to HCN terms by simply introducing empty incoming and outgoing queues for each memory unit. It is easy to show that each HC rule can be simulated by a sequence of HCN rules.

8.1 Inclusion Invariants

As the Inclusion Invariants for the HC model, the HCN model maintains analogous invariants. However, the invariants are more complicated, because caching and de-caching operations involve two memory units that can communicate only via sending and receiving messages. In other words, a coherence action that is atomic in HC may have to be carried out in multiple steps in HCN. For example, if the directory shows that a shared copy has been cached by a child regarding some address, it means three possible cases: the child has the shared copy at the time; a Sh-rep message is on the way to the child (the child will receive the shared copy); or an Inv-rep message is on the way to the parent (the child has just invalidated the shared copy).

In our notation, we use '∪' to represent the operator that merges messages from two queues, and '−' the operator that deletes a message from a queue. The invariants fall into two categories: the situation when a cell is cached in a memory; and the situation when a message is in transient in the network. The meaning of the invariants is straightforward. For example, if a child has a shared cell, then either the parent has the address with the same value while the directory shows that the child has a shared copy; or there is a Wb-rep message on the way to the parent in which the Hstate indicates that an exclusive copy has been given to the child.

Lemma 12 (Inclusion Invariants) In any HCN term
$$\mathsf{Sys}(\langle \mathsf{id}, \mathsf{m}, \mathsf{in}, \mathsf{out}\rangle, \mathsf{Sys}(\langle \mathsf{id_k}, \mathsf{m_k}, \mathsf{in_k}, \mathsf{out_k}\rangle, \mathsf{eu_k}) \mid \mathsf{sg})\ ,$$

Sh-Inclusion

$\mathsf{Cell(a,v,(Sh,R(\text{-})))} \in \mathsf{m_k} \implies$

$\mathsf{Cell(a,v,(\text{-},R(id_k|\text{-})))} \in \mathsf{m}$

$\mathsf{Msg(id, id_k, \text{-}, a, \text{-})} \notin \mathsf{out} \cup \mathsf{in_k}$

$\mathsf{Msg(id_k, id, \text{-}, a, \text{-})} \notin \mathsf{out_k} \cup \mathsf{in}$

or

$\mathsf{Cell(a,\text{-},(Ex,W(id_k)))} \in \mathsf{m}$

$\mathsf{Msg(id_k, id, Wb\text{-}rep, a, v)} \in \mathsf{out_k} \cup \mathsf{in}$

$\mathsf{Msg(id, id_k, \text{-}, a, \text{-})} \notin \mathsf{out} \cup \mathsf{in_k}$

$\mathsf{Msg(id_k, id, \text{-}, a, \text{-})} \notin \mathsf{out_k} \cup \mathsf{in} - \mathsf{Msg(id_k, id, Wb\text{-}rep, a, v)}$

Ex-Inclusion

$\mathsf{Cell(a,v,(Ex,\text{-}))} \in \mathsf{m_k} \implies$

$\mathsf{Cell(a,\text{-},(Ex,W(id_k)))} \in \mathsf{m}$

$\mathsf{Msg(id, id_k, \text{-}, a, \text{-})} \notin \mathsf{out} \cup \mathsf{in_k}$

$\mathsf{Msg(id_k, id, \text{-}, a, \text{-})} \notin \mathsf{out_k} \cup \mathsf{in}$

Sh-rep-Inclusion

$\mathsf{Msg(id, id_k, Sh\text{-}rep, a, v)} \in \mathsf{out} \cup \mathsf{in_k} \implies$

$\mathsf{a} \notin \mathsf{m_k}$

$\mathsf{Cell(a,v,(\text{-},R(id_k|\text{-})))} \in \mathsf{m}$

$\mathsf{Msg(id, id_k, \text{-}, a, \text{-})} \notin \mathsf{out} \cup \mathsf{in_k} - \mathsf{Msg(id, id_k, Sh\text{-}rep, a, v)}$

$\mathsf{Msg(id_k, id, \text{-}, a, \text{-})} \notin \mathsf{out_k} \cup \mathsf{in}$

Ex-rep-Inclusion

$\mathsf{Msg(id, id_k, Ex\text{-}rep, a, v)} \in \mathsf{out} \cup \mathsf{in_k} \implies$

$\mathsf{a} \notin \mathsf{m_k}$

$\mathsf{Cell(a,v,(Ex,W(id_k)))} \in \mathsf{m}$

$\mathsf{Msg(id, id_k, \text{-}, a, \text{-})} \notin \mathsf{out} \cup \mathsf{in_k} - \mathsf{Msg(id, id_k, Ex\text{-}rep, a, v)}$

$\mathsf{Msg(id_k, id, \text{-}, a, \text{-})} \notin \mathsf{out_k} - \mathsf{in}$

Wb-rep-Inclusion

$\mathsf{Msg(id_k, id, Wb\text{-}rep, a, v)} \in \mathsf{out} \cup \mathsf{in_k} \implies$

$\mathsf{Cell(a,v,(Sh,R(\text{-})))} \in \mathsf{m_k}$

$\mathsf{Cell(a,\text{-},(Ex,W(id_k)))} \in \mathsf{m}$

$\mathsf{Msg(id, id_k, \text{-}, a, \text{-})} \notin \mathsf{out} \cup \mathsf{in_k}$

$\mathsf{Msg(id_k, id, \text{-}, a, \text{-})} \notin \mathsf{out_k} \cup \mathsf{in} - \mathsf{Msg(id_k, id, Wb\text{-}rep, a, v)}$

or

$\mathsf{a} \notin \mathsf{m_k}$

$\mathsf{Cell(a,\text{-},(Ex,W(id_k)))} \in \mathsf{m}$

$\mathsf{Msg(id_k, id, Inv\text{-}rep, a, \bot)} \in \mathsf{out_k} \cup \mathsf{in}$

$\mathsf{Msg(id, id_k, \text{-}, a, \text{-})} \notin \mathsf{out} \cup \mathsf{in_k}$

$\mathsf{Msg(id_k, id, \text{-}, a, \text{-})} \notin \mathsf{out_k} \cup \mathsf{in} - \mathsf{Msg(id_k, id, Wb\text{-}rep, a, v)} - \mathsf{Msg(id_k, id, Inv\text{-}rep, a, \bot)}$

Inv-rep-Inclusion

$\mathsf{Msg}(\mathsf{id_k}, \mathsf{id}, \mathsf{Inv\text{-}rep}, \mathsf{a}, \bot) \in \mathsf{out} \cup \mathsf{in_k} \quad \Longrightarrow$

$\mathsf{a} \notin \mathsf{m_k}$
$\mathsf{Cell(a,\text{-},(\text{-},R(id_k|\text{-})))} \in \mathsf{m}$
$\mathsf{Msg}(\mathsf{id}, \mathsf{id_k}, \text{-}, \mathsf{a}, \text{-}) \notin \mathsf{out} \cup \mathsf{in_k}$
$\mathsf{Msg}(\mathsf{id_k}, \mathsf{id}, \text{-}, \mathsf{a}, \text{-}) \notin \mathsf{out_k} \cup \mathsf{in} - \mathsf{Msg}(\mathsf{id_k}, \mathsf{id}, \mathsf{Inv\text{-}rep}, \mathsf{a}, \bot)$

or

$\mathsf{a} \notin \mathsf{m_k}$
$\mathsf{Cell(a,\text{-},(Ex,W(id_k)))} \in \mathsf{m}$
$\mathsf{Msg}(\mathsf{id_k}, \mathsf{id}, \mathsf{Wb\text{-}rep}, \mathsf{a}, \mathsf{v}) \in \mathsf{out_k} \cup \mathsf{in}$
$\mathsf{Msg}(\mathsf{id}, \mathsf{id_k}, \text{-}, \mathsf{a}, \text{-}) \notin \mathsf{out} \cup \mathsf{in_k}$
$\mathsf{Msg}(\mathsf{id_k}, \mathsf{id}, \text{-}, \mathsf{a}, \text{-}) \notin \mathsf{out_k} \cup \mathsf{in} - \mathsf{Msg}(\mathsf{id_k}, \mathsf{id}, \mathsf{Wb\text{-}rep}, \mathsf{a}, \mathsf{v}) - \mathsf{Msg}(\mathsf{id_k}, \mathsf{id}, \mathsf{Inv\text{-}rep}, \mathsf{a}, \bot)$

Proof The proof is by induction on rewriting steps. The invariants hold trivially for the initial terms where all caches and queues are empty. It can be shown by checking each rewriting rule that, if the invariants hold for a term, then they still hold after the term is rewritten according to that rule. □

Based on the Inclusion Invariants, we can show that no memory can contain two cells that have the same address. This is because only the *Receive-Sh-Rep* and *Receive-Ex-Rep* rules can create new cells; when a Sh-rep or Ex-rep is received, the memory cannot have a cell regarding the same address.

8.2 Queue Flushing Property

Suppose we define a new rewriting system $\mathsf{R_{QF}}$, which has the same grammar as HCN but uses only a subset of the HCN rules.

Definition 13 (TRS for queue flushing)

$\mathsf{R_{QF}} \equiv$ { *Message-Passing-To-Child, Message-Passing-To-Parent, Receive-Sh-Rep, Receive-Ex-Rep, Receive-Wb-Rep, Receive-Inv-Rep* }

Now we discuss some properties of the $\mathsf{R_{QF}}$ system.

Lemma 14 $\mathsf{R_{QF}}$ is strongly terminating and confluent, i.e., for any HCN term, rewriting with respect to $\mathsf{R_{QF}}$ terminates within a finite number of steps and always reaches the same normal form, regardless of the order in which the rules are applied.

Proof The termination is obvious because $\mathsf{R_{QF}}$ includes only the message passing and message receive rules. The message passing rules move messages from sources to destinations, while the message receive rules process messages extracted from incoming queues. However, none of these rules can generate new messages. The confluence follows from the fact that message passing and message receive rules do not interfere each other. □

In fact $\mathsf{R_{QF}}$ rules not only do not interfere each other, but also do not interfere with non-$\mathsf{R_{QF}}$ rules. This can be trivially verified by checking each pair of $\mathsf{R_{QF}}$ / non-$\mathsf{R_{QF}}$ rules.

Lemma 15 $\mathsf{R_{QF}}$ rules do not interfere with non-$\mathsf{R_{QF}}$ rules, i.e., if $s_1 \longrightarrow s_2$ by applying rule $\alpha \notin \mathsf{R_{QF}}$, and $s_1 \longrightarrow s_3$ by applying rule $\beta \in \mathsf{R_{QF}}$, then there exists s_4 such that $s_2 \longrightarrow s_4$ by applying β and $s_3 \longrightarrow s_4$ by applying α.

Definition 16 For any HCN term s, $NF_{QF}(s)$ is the normal form of s in $\mathsf{R_{QF}}$.

Lemma 17 (Queue Flushing Property) For any HCN term s, all incoming and outgoing message queues are empty in $NF_{QF}(s)$.

Proof Suppose not all message queues are empty. If an outgoing queue is not empty, some message passing rule would apply; if an incoming queue is not empty, according to the Inclusion Invariants, some message receive rule would apply. Thus the term cannot be a normal form. □

8.3 Soundness of HCN

We define function QF (queue-flush) that maps HCN terms to HC terms as follows:

Definition 18 (Queue-flush function) For any HCN term s, $QF(s)$ is the projection of $NF_{QF}(s)$ on the corresponding HC term where all the message queues have been deleted.

The queue-flush function builds a relationship between HCN terms and HC terms. Based on this mapping function, HC can simulate HCN in the following sense: if HCN applies a message passing or message receive rule, then HC applies no rule (no action is taken); if HCN applies a memory access or message send rule, then HC applies an appropriate rule defined as follows.

the HCN rule	the corresponding HC rule
HCN-Load	*HC-Load*
HCN-Store	*HC-Store*
Send-Sh-Rep	*Sh-Caching*
Send-Ex-Rep	*Ex-Caching*
Send-Wb-Rep	*Writeback*
Send-Inv-Rep	*Invalidate*

Lemma 19 (Soundness) $s_1 \overset{\mathsf{HCN}}{\twoheadrightarrow} s_2 \quad \Longrightarrow \quad QF(s_1) \overset{\mathsf{HC}}{\twoheadrightarrow} QF(s_2)$.

Proof We give the proof for one rewriting step; proof for multiple steps follows from induction. Assume $s_1 \longrightarrow s_2$ by applying some rule α in HCN. The proof is based on the case analysis on α:

- $\alpha \in \mathsf{R_{QF}}$. Needless to say $QF(s_1) = QF(s_2)$.
- α is a memory access rule. Since α cannot interfere with any $\mathsf{R_{QF}}$ rule and applying α cannot generate any new message, it can be shown by induction that $NF_{QF}(s_1) \longrightarrow NF_{QF}(s_2)$ by applying α. Notice all message queues are empty in $NF_{QF}(s_1)$ and $NF_{QF}(s_2)$, thus $QF(NF_{QF}(s_1)) \longrightarrow QF(NF_{QF}(s_2))$ by applying the corresponding memory access rule. Hence $QF(s_1) \longrightarrow QF(s_2)$. See Figure 8 (a).

$$\begin{tikzcd}
s_1 \arrow[r, "R_{QF}"] \arrow[d, "\alpha"] & s_3 \arrow[r, dashed, no head] \arrow[d, "\alpha"] & {} \arrow[r, "R_{QF}"] & NF_{QF}(s_1) \arrow[r, dashed, no head, "QF"] \arrow[d, "\alpha"] & QF(s_1) \arrow[d, "\textit{the corresponding memory access rule in HC}"] \\
s_2 \arrow[r, "R_{QF}"] & s_4 \arrow[r, dashed, no head] & {} \arrow[r, "R_{QF}"] & NF_{QF}(s_2) \arrow[r, dashed, no head, "QF"] & QF(s_2)
\end{tikzcd}$$

(a) α is a memory access rule

$$\begin{tikzcd}
s_1 \arrow[r, "R_{QF}"] \arrow[d, "\alpha"] & s_3 \arrow[r, dashed, no head] \arrow[d, "\alpha"] & {} \arrow[r, "R_{QF}"] & NF_{QF}(s_1) \arrow[rr, dashed, no head, "QF"] \arrow[d, "\alpha"] & & QF(s_1) \arrow[d, "\textit{the corresponding caching or de-caching rule in HC}"] \\
s_2 \arrow[r, "R_{QF}"] & s_4 \arrow[r, dashed, no head] & {} \arrow[r, "R_{QF}"] & s_n \arrow[r, two heads, "R_{QF}"] & NF_{QF}(s_2) \arrow[r, dashed, no head, "QF"] & QF(s_2)
\end{tikzcd}$$

(b) α is a message-send rule

Figure 8: Simulate HCN in HC

- α is a message send rule. Since α cannot interfere with any R_{QF} rules, it can be shown by induction that there exists s_n such that $NF_{QF}(s_1) \longrightarrow s_n$ by applying α, and $s_2 \longrightarrow s_n$ by applying R_{QF} rules. Since applying α generates a new message, $s_n \twoheadrightarrow NF_{QF}(s_2)$ by applying a message passing rule followed by a message receive rule. Notice all message queues are empty in $NF_{QF}(s_1)$ and $NF_{QF}(s_2)$, thus $QF(NF_{QF}(s_1)) \longrightarrow QF(NF_{QF}(s_2))$ by applying the corresponding caching or de-caching rule. Hence $QF(s_1) \longrightarrow QF(s_2)$. See Figure 8 (b). □

8.4 Completeness of HCN

We define function QL (queue-lift) that maps HC terms to corresponding HCN terms. For any HC term s, $QL(s)$ is defined by adding empty incoming and outgoing message queues in each memory unit. Based on this mapping function, it is easy to show that each HC rule can be simulated by a sequence of HCN rules. For example, to simulate the *Sh-Caching* rule, we can apply *Send-Sh-Rep*, followed by *Message-Passing-To-Child*, and followed by *Receive-Sh-Rep*. The table below gives the sequence of HCN rules for each HC rule.

the HC rule	the sequence of HCN rules to simulate the HC rule
HC-Load	*HCN-Load*
HC-Store	*HCN-Store*
Sh-Caching	*Send-Sh-Rep* + *Message-Passing-To-Child* + *Receive-Sh-Rep*
Ex-Caching	*Send-Ex-Rep* + *Message-Passing-To-Child* + *Receive-Ex-Rep*
Writeback	*Send-Wb-Rep* + *Message-Passing-To-Parent* + *Receive-Wb-Rep*
Invalidate	*Send-Inv-Rep* + *Message-Passing-To-Parent* + *Receive-Inv-Rep*

Lemma 20 (Completeness) $s_1 \overset{\mathsf{HC}}{\twoheadrightarrow} s_2 \quad\Longrightarrow\quad QL(s_1) \overset{\mathsf{HCN}}{\twoheadrightarrow} QL(s_2)$.

It is trivial to show that QF is the inverse function of QL.

Lemma 21 (Connection) For any HC term s, $QF(QL(s)) = s$.

This completes the proof that HCN is a complete implementation of HC.

Theorem 22 The HCN model completely implements the HC model.

9 Some Optimizations of the HCN Model

A memory model can be implemented with different protocols. In designing cache coherence protocols, the designer often faces various design options. While soundness and liveness should always be guaranteed, different protocols can result in different performance, complexity and implementation cost. In this section, we discuss some optimization techniques for the HCN model. Although the optimization rules cannot be derived from the existing HCN rules, they can be employed safely to optimize certain common scenarios. We can extend HCN with one or more such optimizations while ensuring that the extended model remains a complete implementation of HC. All the optimization rules are imperative rules.

9.1 Pushout

In HCN, when an exclusive cell is pushed out, the memory sends two messages to its parent, a Wb-rep followed by an Inv-rep. The pushout optimization uses just one message, Pushout-rep, to notify the parent of the pushout operation. The Pushout-rep message can be considered as a combination of the Wb-rep and Inv-rep, and can be used to reduce the number of messages. It can also simplify the protocol design, noting in HCN the Inv-rep can arrive before the Wb-rep with non-FIFO message passing.

Send-Pushout-Rep Rule

$$\begin{array}{ll} & \langle \mathsf{id_k},\ \mathsf{Cell(a,v,(Ex,R(}\epsilon\mathsf{)))} \mid \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k} \rangle \\ \longrightarrow & \langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k} \otimes \mathsf{Msg(id_k, id, Pushout\text{-}rep, a, v)} \rangle \quad \textit{where} \quad \mathsf{id} = \mathsf{parent(id_k)} \end{array}$$

Receive-Pushout-Rep Rule

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{Cell(a,u,(Ex,W(id_k)))} \mid \mathsf{m},\ \mathsf{Msg(id_k, id, Pushout\text{-}rep, a, v)} \odot \mathsf{in},\ \mathsf{out} \rangle \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{Cell(a,v,(Ex,R(}\epsilon\mathsf{)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out} \rangle \end{array}$$

We can extend the HCN model with the pushout rules. It is trivial to show that the extended system is a complete implementation of HCN (the projection function replaces each Pushout-rep message with a Wb-rep followed by an Inv-rep).

9.2 Upgrade

The motivation of the upgrade optimization is to allow a shared copy to be upgraded with the exclusive ownership without being invalidated first. If the state of a cell in memory id is (Ex,R($\mathsf{id_k}$)), then memory id can send an Upgrade-rep message to child $\mathsf{id_k}$; when the Upgrade-rep message is received, memory $\mathsf{id_k}$ changes the cell's Cstate from Sh to Ex.

Send-Upgrade-Rep Rule

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{Cell(a,v,(Ex,R(id_k)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out} \rangle \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{Cell(a,v,(Ex,W(id_k)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg(id, id_k, Upgrade\text{-}rep, a, }\perp\mathsf{)} \rangle \end{array}$$

Receive-Upgrade-Rep Rule

$$\begin{array}{ll} & \langle \mathsf{id_k},\ \mathsf{Cell(a,v,(Sh,R(dir)))} \mid \mathsf{m_k},\ \mathsf{Msg(id, id_k, Upgrade\text{-}rep, a, \bot)} \odot \mathsf{in_k},\ \mathsf{out_k} \rangle \\ \longrightarrow & \langle \mathsf{id_k},\ \mathsf{Cell(a,v,(Ex,R(dir)))} \mid \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k} \rangle \end{array}$$

Consider the scenario in which the shared copy in the child has just been invalidated when the Upgrade-rep message arrives (an Inv-rep message is on its way to the parent). Deadlock can happen because the Upgrade-rep and Inv-rep messages cannot be processed (in order for one message to be processed, the other has to be processed first). To We can rely on directive rules to deal with the dilemma, noting that deadlock in an imperative model does not necessarily mean deadlock in the final protocol. For example, the deadlock cannot happen if we can somehow ensure that the shared copy in the child cannot be invalidated in this case. This can be achieved by using directive messages to properly coordinate the parent and child memories.

Relying on directive rules to avoid such deadlock can put unnecessary constraints on how directive messages must be used, and performance can also be sacrificed. A better solution is to provide proper rules so that the protocol can recover from a potential deadlock situation by cancelling certain operation effects. For example, we can use a negative acknowledgment to notify the parent or the child that the Upgrade-rep or Inv-rep message cannot be processed. This technique can be adopted to eliminate deadlocks for many similar scenarios throughout the protocol design.

Using Upgrade-neg-rep: When the Upgrade-rep message arrives, if the child does not have the shared copy, there are two possible cases: either the shared copy has been invalidated (an Inv-rep is on the way to the parent), or the shared copy has not been received yet (a Sh-rep is on the way to the child). Notice that the child cannot decide which case is true based on its local information. To prevent deadlock, the child can discard the Upgrade-rep message and send an Upgrade-neg-rep message to the parent to report the failure of the upgrade operation. When the Upgrade-neg-rep is received, the parent sets the cell's Hstate to Sh (the cell's value is still valid).

Receive-Upgrade-Rep-And-Send-Upgrade-Neg-Rep Rule

$$\begin{array}{ll} & \langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{Msg(id, id_k, Upgrade\text{-}rep, a, v)} \odot \mathsf{in_k},\ \mathsf{out_k} \rangle \quad \textit{if} \quad \mathsf{a} \notin \mathsf{m_k} \\ \longrightarrow & \langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out} \otimes \mathsf{Msg(id_k, id, Upgrade\text{-}neg\text{-}rep, a, \bot)} \rangle \end{array}$$

Receive-Upgrade-Neg-Rep Rule

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{Cell(a,v,(Ex,W(id_k)))} \mid \mathsf{m},\ \mathsf{Msg(id_k, id, Upgrade\text{-}neg\text{-}rep, a, \bot)} \odot \mathsf{in},\ \mathsf{out} \rangle \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{Cell(a,v,(Ex,R(id_k)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out} \rangle \end{array}$$

Using Inv-neg-rep: Instead of using the Upgrade-rep message, we can have the parent send an Inv-neg-rep message to the child to negatively acknowledge the invalidate operation while letting the Upgrade-rep message wait at the child. Notice that the Inv-neg-rep message carries the data read from the cell in the parent. When the Inv-neg-rep is received, the child caches the data in the $(\mathsf{Sh},\mathsf{R}(\epsilon))$ as if the invalidate operation were never performed.

Receive-Inv-Rep-And-Send-Inv-Neg-Rep Rule

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{Cell(a,v,(Ex,W(id_k)))} \mid \mathsf{m},\ \mathsf{Msg(id_k, id, Inv\text{-}rep, a, \bot)} \odot \mathsf{in},\ \mathsf{out} \rangle \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{Cell(a,v,(Ex,W(id_k)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg(id, id_k, Inv\text{-}neg\text{-}rep, a, v)} \rangle \end{array}$$

Receive-Inv-Neg-Rep Rule

$\quad\langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{Msg(id, id_k, Inv\text{-}neg\text{-}rep, a, v)} \odot \mathsf{in_k},\ \mathsf{out_k} \rangle$

$\longrightarrow\quad \langle \mathsf{id_k},\ \mathsf{Cell(a,v,(Sh,R(\epsilon)))} \mid \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k} \rangle$

Discussion: The rewriting rules of an imperative model can be classified as two categories, the *memory access rules* that perform memory access operations such as load and store, and the *coherence maintenance rules* that move coherence information such as data and ownership in the memory hierarchy. Intrinsically the imperative model is not confluent due to potential data access races. However, we can always make the coherence maintenance rules confluent by introducing extra rules if necessary to undo improper operations. The confluence of the coherence maintenance rules eliminates deadlock in the imperative model, thereby providing more flexibility for the directive design phase.

9.3 Forward

The forward optimization accelerates the processing when a memory wants to have a shared or exclusive copy while the most up-to-date data resides in some sibling memory. It allows the memory that exclusively owns the data to send a copy to a sibling memory directly. If a memory has a cell in the $(\mathsf{Ex},\mathsf{R}(\epsilon))$ state, it can send a Sh-fwd-rep message to a sibling memory to give it a shared copy; if a memory has a cell in the $(\mathsf{Ex},\mathsf{R}(\epsilon))$ state, it can send an Ex-fwd-rep message to a sibling memory to give it an exclusive copy.

The parent memory (home) should be informed when a forward operation happens so that the cell can be updated properly. Depending on which site is responsible for notifying the home (the memory that receives the forwarded data, or the memory that sends the forwarded data), the forward optimization can be implemented in the lazy or eager manner (see Figure 9).

Lazy home notification: When a memory receives a Sh-fwd-rep or Ex-fwd-rep message from a sibling, it sends a Sh-fwd-home-rep or Ex-fwd-home-rep message to the parent. Notice the Sh-fwd-home-rep also contains the most up-to-date data.

Send-Sh-Fwd-Rep Rule

$\quad\langle \mathsf{id_k},\ \mathsf{Cell(a,v,(Ex,R(dir)))} \mid \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k} \rangle$

$\longrightarrow\quad \langle \mathsf{id_k},\ \mathsf{Cell(a,v,(Sh,R(dir)))} \mid \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k} \otimes \mathsf{Msg(id_k, id_j, Sh\text{-}fwd\text{-}rep, a, v)} \rangle$

$\qquad\qquad$ *where* $\mathsf{id_j} \in \mathsf{siblings(id_k)}$

Receive-Sh-Fwd-Rep-And-Send-Sh-Fwd-Home-Rep Rule

$\quad\langle \mathsf{id_j},\ \mathsf{m_j},\ \mathsf{Msg(id_k, id_j, Sh\text{-}fwd\text{-}rep, a, v)} \odot \mathsf{in_j},\ \mathsf{out_j} \rangle$

$\longrightarrow\quad \langle \mathsf{id_j},\ \mathsf{Cell(a,v,(Sh,R(\epsilon)))} \mid \mathsf{m_j},\ \mathsf{in_j},\ \mathsf{out_j} \otimes \mathsf{Msg(id_j, id, Sh\text{-}fwd\text{-}home\text{-}rep, a, v)} \rangle$

$\qquad\qquad$ *where* $\mathsf{id} = \mathsf{parent(id_j)}$

Receive-Sh-Fwd-Home-Rep Rule

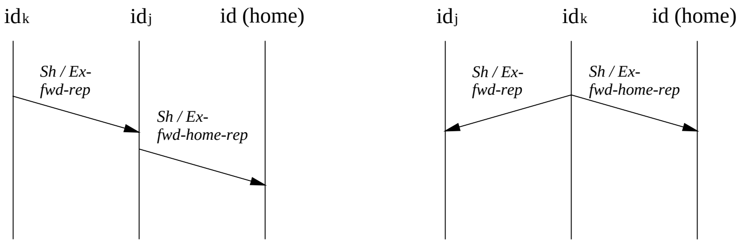
$\quad\langle \mathsf{id},\ \mathsf{Cell(a,u,(Ex,W(id_k)))} \mid \mathsf{m},\ \mathsf{Msg(id_j, id, Sh\text{-}fwd\text{-}home\text{-}rep, a, v)} \odot \mathsf{in},\ \mathsf{out} \rangle$

$\longrightarrow\quad \langle \mathsf{id},\ \mathsf{Cell(a,v,(Ex,R(id_k|id_j)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out} \rangle$

Send-Ex-Fwd-Rep Rule

$\quad\langle \mathsf{id_k},\ \mathsf{Cell(a,v,(Ex,R(\epsilon)))} \mid \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k} \rangle$

$\longrightarrow\quad \langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k} \otimes \mathsf{Msg(id_k, id_j, Ex\text{-}fwd\text{-}rep, a, v)} \rangle \quad$ *where* $\mathsf{id_j} \in \mathsf{siblings(id_k)}$



(a) lazy home notification

(b) eager home notification

Figure 9: The Forward Optimization

Receive-Ex-Fwd-Rep-And-Send-Ex-Fwd-Home-Rep Rule

$$\langle \mathsf{id_j},\ \mathsf{m_j},\ \mathsf{Msg(id_k, id_j, Ex\text{-}fwd\text{-}rep, a, v)} \odot \mathsf{in_j},\ \mathsf{out_j}\rangle$$
$$\longrightarrow\quad \langle \mathsf{id_j},\ \mathsf{Cell(a,v,(Ex,R(\epsilon)))} \mid \mathsf{m_j},\ \mathsf{in_j},\ \mathsf{out_j} \otimes \mathsf{Msg(id_j, id, Ex\text{-}fwd\text{-}home\text{-}rep, a, \perp)}\rangle$$
$$\textit{where}\quad \mathsf{id} = \mathsf{parent(id_j)}$$

Receive-Ex-Fwd-Home-Rep Rule

$$\langle \mathsf{id},\ \mathsf{Cell(a,u,(Ex,W(id_k)))} \mid \mathsf{m},\ \mathsf{Msg(id_j, id, Ex\text{-}fwd\text{-}home\text{-}rep, a, \perp)} \odot \mathsf{in},\ \mathsf{out}\rangle$$
$$\longrightarrow\quad \langle \mathsf{id},\ \mathsf{Cell(a,u,(Ex,W(id_j)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out}\rangle$$

Eager home notification: When a memory sends a Sh-fwd-rep or a Ex-fwd-rep message to a sibling memory, it also sends a Sh-fwd-home-rep or Ex-fwd-home-rep message to the parent to notify the home of the forward operation. Notice that the home notification messages contain a new field that tells which memory receives the forwarded data.

Send-Sh-Fwd-Rep-And-Sh-Fwd-Home-Rep Rule

$$\langle \mathsf{id_k},\ \mathsf{Cell(a,v,(Ex,R(dir)))} \mid \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k}\rangle$$
$$\longrightarrow\quad \langle \mathsf{id_k},\ \mathsf{Cell(a,v,(Sh,R(dir)))} \mid \mathsf{m_k},\ \mathsf{in_k}$$
$$\mathsf{out_k} \otimes \mathsf{Msg(id_k, id_j, Sh\text{-}fwd\text{-}rep, a, v)} \otimes \mathsf{Msg(id_k, id, Sh\text{-}fwd\text{-}home\text{-}rep, a, v, id_j)}\rangle$$
$$\textit{where}\quad \mathsf{id_j} \in \mathsf{siblings(id_k)} \quad \textit{and} \quad \mathsf{id} = \mathsf{parent(id_k)}$$

Receive-Sh-Fwd-Rep Rule

$$\langle \mathsf{id_j},\ \mathsf{m_j},\ \mathsf{Msg(id_k, id_j, Sh\text{-}fwd\text{-}rep, a, v)} \odot \mathsf{in_j},\ \mathsf{out_j}\rangle$$
$$\longrightarrow\quad \langle \mathsf{id_j},\ \mathsf{Cell(a,v,(Sh,R(\epsilon)))} \mid \mathsf{m_j},\ \mathsf{in_j},\ \mathsf{out_j}\rangle \quad \textit{where}\quad \mathsf{id} = \mathsf{parent(id_j)}$$

Receive-Sh-Fwd-Home-Rep Rule

$$\langle \mathsf{id},\ \mathsf{Cell(a,u,(Ex,W(id_k)))} \mid \mathsf{m},\ \mathsf{Msg(id_k, id, Sh\text{-}fwd\text{-}home\text{-}rep, a, v, id_j)} \odot \mathsf{in},\ \mathsf{out}\rangle$$
$$\longrightarrow\quad \langle \mathsf{id},\ \mathsf{Cell(a,v,(Ex,R(id_k|id_j)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out}\rangle$$

Send-Ex-Fwd-Rep-And-Ex-Fwd-Home-Rep Rule

$$\langle \mathsf{id_k},\ \mathsf{Cell(a,v,(Ex,R(\epsilon)))} \mid \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k}\rangle$$
$$\longrightarrow\quad \langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k} \otimes \mathsf{Msg(id_k, id_j, Ex\text{-}fwd\text{-}rep, a, v)} \otimes \mathsf{Msg(id_k, id, Ex\text{-}fwd\text{-}home\text{-}rep, a, \perp, id_j)}\rangle$$
$$\textit{where}\quad \mathsf{id_j} \in \mathsf{siblings(id_k)} \quad \textit{and} \quad \mathsf{id} = \mathsf{parent(id_k)}$$

Receive-Ex-Fwd-Rep Rule

$$\langle \mathsf{id_j},\ \mathsf{m_j},\ \mathsf{Msg(id_k, id_j, Ex\text{-}fwd\text{-}rep, a, v)} \odot \mathsf{in_j},\ \mathsf{out_j}\rangle$$
$$\longrightarrow\quad \langle \mathsf{id_j},\ \mathsf{Cell(a,v,(Ex,R(\epsilon)))} \mid \mathsf{m_j},\ \mathsf{in_j},\ \mathsf{out_j}\rangle \quad \textit{where}\quad \mathsf{id} = \mathsf{parent(id_j)}$$

Receive-Ex-Fwd-Home-Rep Rule

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{Cell(a,u,(Ex,W(id_k)))} \mid \mathsf{m},\ \mathsf{Msg(id_k, id, Ex\text{-}fwd\text{-}home\text{-}rep, a, \perp, id_j)} \odot \mathsf{in},\ \mathsf{out}\rangle \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{Cell(a,u,(Ex,W(id_j)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out}\rangle \end{array}$$

Discussion: With lazy notification, it is guaranteed that the forwarded data has been received when the home is notified. One the other hand, eager notification takes just one message passing hop to notify the home. Given a FIFO network, lazy notification is often preferred, because eager notification can prohibit us from taking advantage from the FIFO order preserved by the message passing.

Message passing between siblings: In addition to the message passing rules that transfer messages between the parent and the child, the forward operation also requires the following rule that transfers messages between siblings.

Message-Passing-To-Sibling Rule

$$\begin{array}{ll} & \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{Msg(id_k, id_j, cmd, a, v)} \otimes \mathsf{out_k}\rangle,\ \mathsf{eu_k}) \mid \mathsf{Sys}(\langle \mathsf{id_j},\ \mathsf{m_j},\ \mathsf{in_j},\ \mathsf{out_j}\rangle,\ \mathsf{eu_j}) \\ \longrightarrow & \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k}\rangle,\ \mathsf{eu_k}) \mid \mathsf{Sys}(\langle \mathsf{id_j},\ \mathsf{m_j},\ \mathsf{in_j},\ \mathsf{Msg(id_k, id_j, cmd, a, v)} \otimes \mathsf{out_j}\rangle,\ \mathsf{eu_j}) \end{array}$$

It is also worth pointing out that for eager home notification, the *Message-Passing-To-Parent* rule need to be modified slightly to accommodate messages that have the extra field.

10 Directive Messages and Directive Rules

The imperative models presented so far rely on an oracle to cause appropriate coherence actions at appropriate times. Consider the scenario that a processor intends to execute a Load instruction while the accessed data is not cached in the L1 cache. In HCN, if the parent has the most up-to-date data, it can send a copy to the L1 cache, however, it is not clear how it would know that the child needs the data for the particular address.

To remedy this problem, we introduce directive messages and directive rules. While imperative messages can be used to transfer data and other coherence information between memory sites, the intended effect of directive messages is to invoke desirable coherence actions. For example, when a read cache miss happens, a directive message can be sent to the parent to request the accessed data; when the directive message is received, the parent can send a shared copy to the child.

Directive rules are forbidden to modify any state manipulated by imperative rules. This guarantees that correctness cannot be compromised when directive rules are added. When imperative and directive rules are integrated, directive messages usually act as extra predicates to indicate when the imperative actions should be invoked. Improper conditions for invoking imperative rules can cause deadlocks or livelocks but cannot affect the correctness of the system. Before we delve into directive messages and directive rules, we first discuss some implementation assumptions and explicate the protocol liveness.

10.1 Rewriting Fairness

While rewriting a term, there may exist more than one rules that can be applied. A rewriting strategy can be used to select a rule or a group of rules from the set of applicable rules. If a rule can be applied on different redexes, the strategy also specifies the redex or redexes that can be rewritten. When TRS's are used to describe programming languages, terms represent expressions and rewriting rules represent computations. The goal of a rewriting strategy is to produce the value or one of the values of an expression.

Unfortunately, there is no such obvious goal for distributed systems such as cache coherence protocols. Instead, there is a notion of *fair rewriting*, which requires that all concurrent components execute in a distributed fashion and hence make progress in parallel. The concurrent components of a typical DSM system may include processors, network and message queues, memory units, and so on. Without losing generality, we make the following assumptions:

- **Concurrent processor execution:** A processor cannot be stalled indefinitely while it has executable instructions;
- **Reliable message passing:** An outgoing message is guaranteed to be delivered to the destination in finite time;
- **Fair message processing:** An incoming message is guaranteed to be processed sooner or later if it can be processed.

While the first two assumptions are intuitive and can be satisfied in most implementations, the fair message processing assumption deserves some explanation. Fair message processing implies that messages that cannot be processed temporarily should not block other messages from being processed. Moreover, no message should remain in an incoming queue indefinitely, unless it cannot be processed *at all times* from certain time. In other words, when more than one messages can be processed, the selection of which message is to be processed is fair so that any message that can be processed will be processed eventually.

These implementation assumptions manifest the necessity of rewriting fairness. A fair rewriting strategy guarantees that each concurrent component periodically obtains an execution opportunity as the system makes progress. If the system does not terminate, an infinite number of opportunities must be given to each component throughout its execution. This fairness prevents the execution of an applicable rule from being delayed indefinitely. If a term contains a redex, the redex must be rewritten in finite number of steps, unless such rewriting becomes illegal (noting the redex can be destroyed, and the context in which the redex resides can be changed).

10.2 Protocol Liveness

The correctness and the liveness concerns are separate issues in the Imperative-Directive design methodology. While correctness ensures that the protocol can only exhibit behaviors allowed by the memory model (i.e., bad things cannot happen), liveness ensures that desirable coherence actions will eventually be invoked (i.e., good things will happen). The definition of correctness is unambiguous given a precisely defined memory model. However, liveness can have very different implications for different protocols.

We say a protocol enforces *weak liveness*, if it ensures that the system cannot enter a deadlock situation in which no further action can be invoked. Weak liveness does not eliminate potential livelocks, although in certain implementations, the probability of livelocks may be negligible. For example, some protocols [18, 1] implemented in simple hardware avoid deadlocks by bouncing messages that cannot be processed back to their senders. In such protocols, it is possible that messages are passed back and forth in the system while no processor makes any progress.

If a protocol ensures that the system as a whole can always make progress, we say it enforces *moderate liveness*. Moderate liveness implies that when multiple processors access the same memory address, at least one of them will succeed eventually. However, starvation can happen because no fairness is guaranteed among different processors. For example, it is possible that a cache miss is never serviced while memory accesses from other processors are always satisfied.

We say a protocol enforces *strong liveness*, if it ensures that each individual processor and memory unit can always make progress. This guarantees that a cache miss on any processor can always be serviced, and a cell in any cache can always be purged whenever necessary. As we shall see, the HCN-base protocol we will present provides such liveness; the protocol is free from all sorts of deadlock, livelock and starvation.

10.3 Directive Messages and Directive Rules

Directive messages are needed to coordinate memory sites so that appropriate coherence actions can be invoked whenever necessary. It is usually straightforward to determine what directive messages should be used for an imperative model. For HCN, we introduce four directive messages: Sh-req, Ex-req, Wb-req and Inv-req, where the suffix '-req' stands for 'request', since such messages are often used as requests to invoke desirable coherence actions. Notice a caching request flows from child to parent, while a de-caching request from parent to child.

Request	Request description	Expected reply
Sh-req	Request a shared copy from the parent	Sh-rep
Ex-req	Request an exclusive copy from the parent	Ex-rep
Wb-req	Request a child to write back the most up-to-date data	Wb-rep
Inv-req	Request a child to invalidate a shared copy	Inv-rep

Directive rules deal with directive messages. Without affecting correctness, directive messages can be generated or discarded at any time. However, directive rules are not allowed to produce or consume any imperative message or modify any memory cell. This guarantees that correctness cannot be compromised when directive messages and directive rules are incorporated. In other words, we can safely extend a correct imperative model with arbitrary directive messages and directive rules; the extended model remains a correct implementation.

Directive Generation Rules: At any time, a memory can send a Sh-req or Ex-req message to its parent; or a Wb-req or Inv-req message to a child.

Generate-Req-To-Parent Rule

$$
\begin{array}{ll}
 & \langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out} \rangle \\
\longrightarrow & \langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg}(\mathsf{id}, \mathsf{id}_{\mathrm{p}}, \mathsf{cmd}, \mathsf{a}, \perp) \rangle \\
 & \qquad \qquad \textit{where} \ \ \mathsf{id}_{\mathrm{p}} = \mathsf{parent}(\mathsf{id}) \quad \textit{and} \ \ \mathsf{cmd} = \mathsf{Sh\text{-}req/Ex\text{-}req}
\end{array}
$$

Generate-Req-To-Child Rule

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out}\rangle \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg}(\mathsf{id}, \mathsf{id}_\mathrm{k}, \mathsf{cmd}, \mathsf{a}, \perp)\rangle \\ & \qquad\qquad \textit{where} \ \ \mathsf{id}_\mathrm{k} \in \mathsf{children}(\mathsf{id}) \quad \textit{and} \ \ \mathsf{cmd} = \mathsf{Wb\text{-}req/Inv\text{-}req} \end{array}$$

Directive Discard Rules: At any time, a directive message can be discarded from an incoming or outgoing queue.

Discard-Incoming-Req Rule

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{m},\ \mathsf{in} \odot \mathsf{Msg}(\text{-}, \text{-}, \mathsf{cmd}, \text{-}, \text{-}),\ \mathsf{out}\rangle \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out}\rangle \quad \textit{where} \ \ \mathsf{cmd} = \mathsf{Sh\text{-}req/Ex\text{-}req/Wb\text{-}req/Inv\text{-}req} \end{array}$$

Discard-Outgoing-Req Rule

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg}(\text{-}, \text{-}, \mathsf{cmd}, \text{-}, \text{-})\rangle \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out}\rangle \quad \textit{where} \ \ \mathsf{cmd} = \mathsf{Sh\text{-}req/Ex\text{-}req/Wb\text{-}req/Inv\text{-}req} \end{array}$$

Directive rules alone are not interesting because they allow directive messages to be generated and discarded arbitrarily. However, it is unnecessary for a cache coherence protocol to use all the imperative and directive rules in their raw form. To ensure both correctness and liveness, directive and imperative rules are often integrated to give rise to derived rules. In such derived rules, a cell's coherence state can act as an extra predicate for desired directive actions, while an incoming directive message can act as an extra predicate for desired imperative actions. For example, when a memory receives a Sh-req message from a child, if it has the most up-to-date data, it sends an Sh-rep message to the requesting site; if the accessed address is cached but the data is stale, the memory sends a Wb-req message to the child that has the most up-to-date data. Notice the incoming Sh-req message is used as a predicate for specifying when the *Send-Sh-Rep* or *Generate-Req-To-Child* rule should be invoked.

We extend the HCN model with the directive messages and directive rules, and call it HCNr (HCN with Requests). Obviously HCNr is a complete implementation of HCN.

10.4 Transient Records

When a memory receives a directive message, it may or may not be able to process the message at the time. An incoming message that cannot be processed should not block other messages from being processed. This is guaranteed by allowing any message in an incoming queue to be brought to the front end of the queue whenever necessary.

If an incoming message can be processed, then either the message can be processed to completion; or the message must be *suspended* before it can be processed to completion. Consider the scenario that a memory receives an Ex-req message from a child. If the memory has the exclusive ownership and the most up-to-date data, it simply sends an Ex-rep message to the requesting site. However, if the memory has the exclusive ownership but the data is shared by a number of child memories, the Ex-req message cannot be processed to completion before all the outstanding shared copies have been invalidated. In this case, the Ex-req message is suspended and an Inv-req message is multicast to the child memories that have the shared copies. Later the suspended Ex-req message will be resumed when all the invalidation requests are acknowledged.

For each suspended message, a *transient record* is created to maintain necessary information so that the suspended message can be resumed. Since a message has to be first extracted from the incoming queue before it can be processed, all useful information regarding a suspended message must be maintained in its transient record. Typically a transient record contains the command and source of the suspended message. It may also contain information relevant to the outstanding messages caused by the suspension, noting when a message is suspended, appropriate directive messages are often issued to invoke certain remote coherence actions. When a suspended message is resumed, the corresponding transient record is discarded.

Transient records and coherence states are conceptually orthogonal. While the coherence state of a memory cell is used for coherence maintenance, the transient record of an address helps enforce protocol liveness. Some bits can be saved by encoding the coherence states and transient records together in implementation.

When a transient record exists for an address, we say the address is in a *transient state*, otherwise we say the address is in a *stable state*. Also notice that a memory access instruction may also need to be suspended when the instruction execution is stalled due to a cache miss.

10.5 A Systematic Design Procedure

Based on an imperative model, a cache coherence protocol can be developed systematically according to the following design procedure. The protocol ensures both correctness and liveness by integrating imperative and directive actions appropriately. With this 6-step procedure, the protocol design becomes more tractable and less error-prone. Almost all the rewriting rules can be derived systematically.

1. **Define directive message types.** We begin by defining a directive message type for each imperative message type. Later it may become necessary that a directive message type should convey more information to help the receiving site determine the action to be taken. By encoding such information in the message command, we effectively divide the directive message type to several sub-types. On the other hand, it is also possible that certain directive message types can be merged to one super-type, which may reduce the number of bits needed to encode message commands, as well as the number of rewriting rules.

2. **Define transient record structures.** A transient record contains the initiator that caused the address to enter the transient state. When a message is suspended, the initiator contains the source and command of the suspended message. Sometimes a transient record also needs to include certain information regarding the outstanding messages caused by the suspended message.

3. **Define rules to generate original directive messages**. A directive message can be either original, or transitively generated while processing another directive message. It is often obvious to decide when an original directive message should be issued. For example, when a cache miss happens, a directive message is sent from the L1 cache to the parent memory to request the data.

4. **Define rules to generate original imperative messages**. An imperative message can be either original, or issued as the response to some directive message. It is usually straightforward to determine when an original imperative message should be issued. For example, when a shared cell is replaced (purged) from a cache, an imperative message is sent to the parent so that the directory can be modified accordingly.

5. **Define rules to process directive messages.** Generally speaking, a memory site processes at most one directive message for the same address at a time. Our experience shows that this restriction can dramatically simplify the protocol design without noticeably affecting the performance. However, certain directive messages cannot be blocked because otherwise deadlocks can happen. Such directive messages must be processed in time, regardless of whether the address is in a transient state or not.

   If a directive message can be processed, then either it can be processed to completion, or it should be suspended before it can be processed to completion. When a directive message is processed to completion, a corresponding imperative message is sent to the requesting site. When a directive message is suspended, necessary directive messages are issued in order to service the suspended message later.

6. **Define rules to process imperative messages**. If an imperative message can be processed, it can always be processed to completion. When an imperative message is processed, if a suspended message exists regarding the address, it is resumed immediately if such resumption is possible.

   Notice an imperative or directive message that cannot be processed remains in the incoming queue. The fair message passing guarantees that the message will be processed eventually if it can be processed.

It is important to realize that the design procedure represents an *iterative* design process. As the design proceeds, we have a better understanding of the protocol behavior and may revise some previous design decisions.

11 HCN-base: A Simple Protocol Derived from HCN

In this section, we present HCN-base, a simple cache coherence protocol that implements Sequential Consistency and guarantees that each individual processor can always make progress. HCN-base employs transients records to maintain information for suspended messages and instructions. The protocol is free from all sorts of deadlock, livelock and starvation in the sense that any cache miss can be serviced in finite time and hence no processor can be indefinitely stalled.

The rewriting rules of HCN-base can be systematically derived from the HCN model. Memory access instructions can be executed in case of cache hits. When a cache miss happens, a directive message is sent to the parent memory. When a directive message is received and chosen to be processed, there are two possibilities: (1) if the directive message can be processed to completion, an appropriate imperative message is sent to the requesting site; (2) if it cannot be processed to completion, it is suspended while necessary directive messages are sent to the parent or child memories to invoke certain desired coherence actions. The suspended message is resumed when all expected imperative messages are received.

The protocol has the following properties:

- The memory hierarchy can be any tree structure with arbitrary depth.
- The message passing is non-FIFO, i.e., messages can be received in any order.
- All coherence actions are originally driven by memory access instructions. A memory cannot send data to a child without a caching request from the child, or invalidate a cell or write a cell's value back to its parent without a de-caching request from the parent. This implies that the protocol cannot handle cache line replacement caused by capacity or associativity conflict.

  In [23] we extend HCN-base to HCN-adpt, a fully adaptive cache coherence protocol in which coherence actions can happen *voluntarily* without being requested.
- A memory site generally processes only one request message for the same address at a time. A request message cannot be processed if the address is already in a transient state (unless such delay can cause deadlocks). This restriction dramatically simplifies the protocol design, and its impact on performance is negligible.

  The only exception is Inv-req message, which must be processed regardless of whether the address is in a transient state or not. As we shall see, this is necessary in order to avoid deadlocks.

The grammar of HCN-base is given in Figure 10. Compared with HCN, a new component, the transient records, is maintained in each memory unit. A transient record contains an address, and an initiator that caused the address to enter the transient state. When a message is suspended, the initiator records its source and command; when an instruction is suspended, the initiator records its instruction address and opcode.

The HCN-base rules can be classified into several categories: (1) memory access rules that deal with cache hits and cache misses; (2) child-to-parent request rules that process caching requests; (3) parent-to-child request rules that process de-caching requests; (4) parent-to-child reply rules that process caching replies; (5) child-to-parent reply rules that process de-caching replies; and (6) message passing rules.

We use the following shorthand notation for message multicast:

$$\begin{array}{lcl} \mathsf{multicast}(\mathsf{id},\epsilon,\mathsf{cmd},\mathsf{a},\mathsf{v}) & \equiv & \epsilon \\ \mathsf{multicast}(\mathsf{id},\mathsf{id}_{\mathrm{k}}|\mathsf{dir},\mathsf{cmd},\mathsf{a},\mathsf{v}) & \equiv & \mathsf{Msg}(\mathsf{id},\mathsf{id}_{\mathrm{k}},\mathsf{cmd},\mathsf{a},\mathsf{v}) \otimes \mathsf{multicast}(\mathsf{id},\mathsf{dir},\mathsf{cmd},\mathsf{a},\mathsf{v}) \end{array}$$

11.1 Memory Access Rules

Cache-Hit Rules: Memory access operations by a processor are performed on its L1 cache. A Load instruction can read from the L1 cache if the accessed address is cached. A Store

SYS	$\equiv$	Sys(MU, EU)	*System*
MU	$\equiv$	$\langle$id, MEM, INQ, OUTQ, TRECS$\rangle$	*Memory Unit*
EU	$\equiv$	PROC $[\!]$ SG	*Execution Unit*
SG	$\equiv$	ϵ $[\!]$ SYS $\mid$ SG	*System Group*
MEM	$\equiv$	ϵ $[\!]$ Cell(a,v,STATE) $\mid$ MEM	*Memory & Cache*
STATE	$\equiv$	(CSTATE, HSTATE)	*Cell's State*
CSTATE	$\equiv$	Sh $[\!]$ Ex	*Cell's Cstate*
HSTATE	$\equiv$	R(DIR) $[\!]$ W(id)	*Cell's Hstate*
DIR	$\equiv$	ϵ $[\!]$ id $\mid$ DIR	*Directory*
INQ	$\equiv$	ϵ $[\!]$ MSG $\odot$ INQ	*Incoming Queue*
OUTQ	$\equiv$	ϵ $[\!]$ MSG $\otimes$ OUTQ	*Outgoing Queue*
MSG	$\equiv$	$\text{Msg}(\text{id}_{\text{src}}, \text{id}_{\text{dest}}, \text{CMD}, \text{a}, \text{v})$	*Protocol Message*
CMD	$\equiv$	REPLY $[\!]$ REQUEST	*Message Command*
REPLY	$\equiv$	Sh-rep $[\!]$ Ex-rep $[\!]$ Wb-rep $[\!]$ Inv-rep	*Reply Command*
REQUEST	$\equiv$	Sh-req $[\!]$ Ex-req $[\!]$ Wb-req $[\!]$ Inv-req	*Request Command*
TRECS	$\equiv$	ϵ $[\!]$ Trec(a, INITIATOR) $\mid$ TRECS	*Transient Records*
INITIATOR	$\equiv$	(id,REQUEST) $[\!]$ (ia,Load) $[\!]$ (ia,Store)	*Initiator*

Figure 10: The HCN-base Model (Initially, all non-outermost memories, all message queues, and all transient records are empty; the outermost memory contains a cell for each address and the state of each cell is (Ex,R(ϵ)))

instruction can write a new value to the L1 cache if the accessed address is cached with the exclusive ownership.

Read-Cache-Hit Rule $\surd$

$$\begin{aligned}
&\text{Sys}(\langle\text{id, Cell(a,v,(cs,R}(\epsilon)))\mid\text{m, in, out, trecs}\rangle,\ \ \text{Proc(ia, rf, prog)})\\
&\qquad\qquad \textit{if}\ \ \text{prog[ia]} = \ \text{r} := \text{Load}(\text{r}_1)\ \ \textit{and}\ \ \text{a} = \text{rf}[\text{r}_1]\\
\longrightarrow\ &\text{Sys}(\langle\text{id, Cell(a,v,(cs,R}(\epsilon)))\mid\text{m, in, out, trecs}\rangle,\ \ \text{Proc(ia+1, rf[r} := \text{v], prog)})
\end{aligned}$$

Write-Cache-Hit Rule $\surd$

$$\begin{aligned}
&\text{Sys}(\langle\text{id, Cell(a,u,(Ex,R}(\epsilon)))\mid\text{m, in, out, trecs}\rangle,\ \ \text{Proc(ia, rf, prog)})\\
&\qquad\qquad \textit{if}\ \ \text{prog[ia]} = \ \text{Store}(\text{r}_1, \text{r}_2)\ \ \textit{and}\ \ \text{a} = \text{rf}[\text{r}_1]\\
\longrightarrow\ &\text{Sys}(\langle\text{id, Cell(a,v,(Ex,R}(\epsilon)))\mid\text{m, in, out, trecs}\rangle,\ \ \text{Proc(ia+1, rf, prog)})\quad \textit{where}\ \ \text{v} = \text{rf}[\text{r}_2]
\end{aligned}$$

Cache-Miss Rules: If a processor intends to execute a Load instruction while the accessed address is not cached in the L1 cache, the instruction is suspended and a Sh-req message is sent to the parent to request a shared copy. Similarly, if a processor intends to execute a Store instruction while no exclusive copy is cached in the L1 cache for the accessed address, the instruction is suspended and an Ex-req message is sent to the parent to request an exclusive copy.

Read-Cache-Miss Rule $\surd$

$$\begin{aligned}
&\text{Sys}(\langle\text{id, m, in, out, trecs}\rangle,\ \ \text{Proc(ia, rf, prog)})\\
&\qquad\qquad \textit{if}\ \ \text{prog[ia]} = \ \text{r} := \text{Load}(\text{r}_1)\ \ \textit{and}\ \ \text{rf}[\text{r}_1] \notin \text{m}\ \ \textit{and}\ \ \text{rf}[\text{r}_1] \notin \text{trecs}\\
\longrightarrow\ &\text{Sys}(\langle\text{id, m, in, out} \otimes \text{Msg}(\text{id}, \text{id}_\text{p}, \text{Sh-req}, \text{a}, \bot),\ \text{Trec}(\text{a}, (\text{ia,Load}))\mid\text{trecs}\rangle,\ \ \text{Proc(ia, rf, prog)})\\
&\qquad\qquad \textit{where}\ \ \text{id}_\text{p} = \text{parent(id)}\quad \textit{and}\ \ \text{a} = \text{rf}[\text{r}_1]
\end{aligned}$$

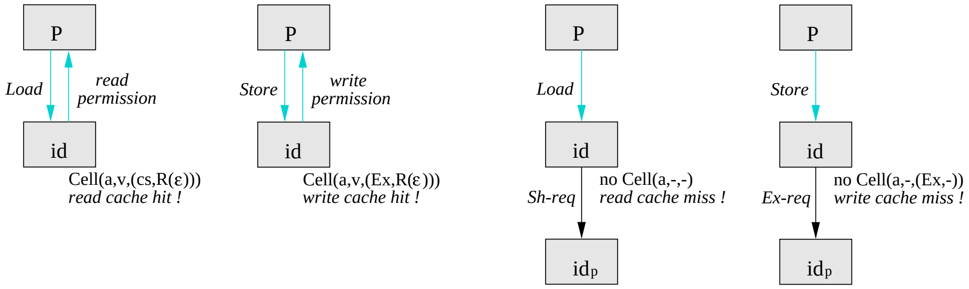


Figure 11: Cache-hit and Cache-miss

Write-Cache-Miss Rule $\surd$

$$\begin{array}{ll}
 & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle,\ \ \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{prog})) \\
 & \qquad \textit{if}\ \ \mathsf{prog}[\mathsf{ia}] = \mathsf{Store}(\mathsf{r}_1,\mathsf{r}_2)\ \ \textit{and}\ \ \mathsf{Cell}(\mathsf{rf}[\mathsf{r}_1],\text{-},(\mathsf{Ex},\text{-})) \notin \mathsf{m}\ \ \textit{and}\ \ \mathsf{rf}[\mathsf{r}_1] \notin \mathsf{trecs} \\
\longrightarrow & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg}(\mathsf{id},\mathsf{id}_\mathrm{p},\mathsf{Ex\text{-}req},\mathsf{a},\perp),\ \mathsf{Trec}(\mathsf{a},(\mathsf{ia},\mathsf{Store}))\mid\mathsf{trecs}\rangle,\ \ \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{prog})) \\
 & \qquad \textit{where}\ \ \mathsf{id}_\mathrm{p} = \mathsf{parent}(\mathsf{id})\ \ \ \textit{and}\ \ \mathsf{a} = \mathsf{rf}[\mathsf{r}_1]
\end{array}$$

Discussion: When an instruction is suspended, a transient record is created for the suspended instruction. This effectively prevents the same request to be issued more than once, noting that the same rule cannot be applied again after the address enters a transient state.

11.2 Child-to-Parent Request Rules

Sh-Request Rules: When memory id receives a Sh-req message from child id_k, it processes the request as follows, provided the address is not in a transient state:

- (Hit) If memory id has the data and the cell's Hstate is R(dir), it sends a Sh-rep message to child id_k and records identifier id_k in its directory.
- (Hit but stale data) If memory id has the data and the cell's state is $(\mathsf{Ex},\mathsf{W}(\mathsf{id}_\mathrm{j}))$, it suspends the request and sends a Wb-req message to child id_j.
- (Miss) If memory id does not have the data, it suspends the request and sends a Sh-req message to its parent.

Receive-Sh-Req-And-Send-Sh-Rep Rule $\surd$

$$\begin{array}{ll}
 & \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{cs},\mathsf{R}(\mathsf{dir})))\mid\mathsf{m},\ \mathsf{Msg}(\mathsf{id}_\mathrm{k},\mathsf{id},\mathsf{Sh\text{-}req},\mathsf{a},\perp)\odot\mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle \\
 & \qquad \textit{if}\ \ \mathsf{id}_\mathrm{k} \notin \mathsf{dir}\ \ \textit{and}\ \ \mathsf{a} \notin \mathsf{trecs} \\
\longrightarrow & \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{cs},\mathsf{R}(\mathsf{id}_\mathrm{k}|\mathsf{dir})))\mid\mathsf{m},\ \mathsf{in},\ \mathsf{out}\otimes\mathsf{Msg}(\mathsf{id},\mathsf{id}_\mathrm{k},\mathsf{Sh\text{-}rep},\mathsf{a},\mathsf{v}),\ \mathsf{trecs}\rangle
\end{array}$$

Receive-Sh-Req-And-Send-Wb-Req Rule $\surd$

$$\begin{array}{ll}
 & \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Ex},\mathsf{W}(\mathsf{id}_\mathrm{j})))\mid\mathsf{m},\ \mathsf{Msg}(\mathsf{id}_\mathrm{k},\mathsf{id},\mathsf{Sh\text{-}req},\mathsf{a},\perp)\odot\mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle \\
 & \qquad \textit{if}\ \ \mathsf{id}_\mathrm{k} \neq \mathsf{id}_\mathrm{j}\ \ \textit{and}\ \ \mathsf{a} \notin \mathsf{trecs} \\
\longrightarrow & \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Ex},\mathsf{W}(\mathsf{id}_\mathrm{j})))\mid\mathsf{m},\ \mathsf{in},\ \mathsf{out}\otimes\mathsf{Msg}(\mathsf{id},\mathsf{id}_\mathrm{j},\mathsf{Wb\text{-}req},\mathsf{a},\perp),\ \mathsf{Trec}(\mathsf{a},(\mathsf{id}_\mathrm{k},\mathsf{Sh\text{-}req}))\mid\mathsf{trecs}\rangle
\end{array}$$

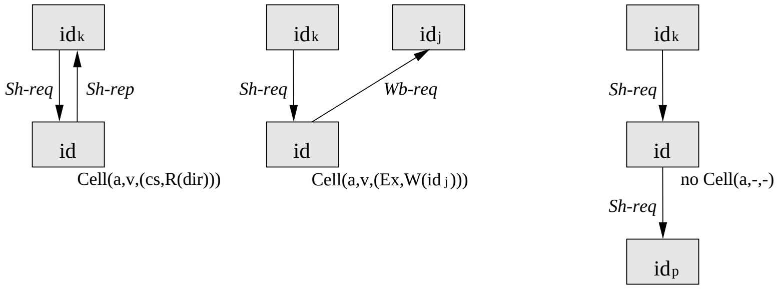


Figure 12: Memory id receives a $\mathsf{Sh\text{-}req}$ message from child $\mathsf{id_k}$

Receive-Sh-Req-And-Send-Sh-Req Rule

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{m},\ \mathsf{Msg}(\mathsf{id_k},\mathsf{id},\mathsf{Sh\text{-}req},\mathsf{a},\perp)\odot\mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle \quad \textit{if}\ \ \mathsf{a}\notin\mathsf{m}\ \ \textit{and}\ \ \mathsf{a}\notin\mathsf{trecs} \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out}\otimes\mathsf{Msg}(\mathsf{id},\mathsf{id_p},\mathsf{Sh\text{-}req},\mathsf{a},\perp),\ \mathsf{Trec}(\mathsf{a},(\mathsf{id_k},\mathsf{Sh\text{-}req}))\mid\mathsf{trecs}\rangle \\ & \qquad\qquad \textit{where}\ \ \mathsf{id_p}=\mathsf{parent}(\mathsf{id}) \end{array}$$

Ex-Request Rules: When memory id receives an $\mathsf{Ex\text{-}req}$ message from child $\mathsf{id_k}$, it processes the request as follows, provided the address is not in a transient state:

- (Hit) If memory id has the data in the $(\mathsf{Ex},\mathsf{R}(\epsilon))$ state, it sends an $\mathsf{Ex\text{-}rep}$ message to child $\mathsf{id_k}$ and changes the cell's Hstate to $\mathsf{W}(\mathsf{id_k})$.
- (Hit but outstanding reads) If memory id has the data and the cell's state is $(\mathsf{Ex},\mathsf{R}(\mathsf{dir}))$ where $\mathsf{dir}\neq\epsilon$, it suspends the request and multicasts an $\mathsf{Inv\text{-}req}$ message to the child memories specified by directory dir.
- (Hit but stale data) If memory id has the data and the cell's state is $(\mathsf{Ex},\mathsf{W}(\mathsf{id_j}))$, it suspends the request and sends a $\mathsf{Wb\text{-}req}$ message to child $\mathsf{id_j}$.

  Notice that an $\mathsf{Inv\text{-}req}$ message must be sent to memory $\mathsf{id_j}$ sooner or later in order to obtain the exclusive ownership. To keep the protocol simple, this message is not issued until the $\mathsf{Wb\text{-}rep}$ message requested by the $\mathsf{Wb\text{-}req}$ message is received (see the *Receive-Wb-Rep-And-Send-Inv-Req* rule).
- (Miss) If memory id does not have an exclusive copy for the accessed address, it suspends the request and sends an $\mathsf{Ex\text{-}req}$ message to its parent.

Receive-Ex-Req-And-Send-Ex-Rep Rule $\surd$

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Ex},\mathsf{R}(\epsilon)))\mid\mathsf{m},\ \mathsf{Msg}(\mathsf{id_k},\mathsf{id},\mathsf{Ex\text{-}req},\mathsf{a},\perp)\odot\mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle \quad \textit{if}\ \ \mathsf{a}\notin\mathsf{trecs} \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Ex},\mathsf{W}(\mathsf{id_k})))\mid\mathsf{m},\ \mathsf{in},\ \mathsf{out}\otimes\mathsf{Msg}(\mathsf{id},\mathsf{id_k},\mathsf{Ex\text{-}rep},\mathsf{a},\mathsf{v}),\ \mathsf{trecs}\rangle \end{array}$$

Receive-Ex-Req-And-Multicast-Inv-Req Rule $\surd$

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Ex},\mathsf{R}(\mathsf{dir})))\mid\mathsf{m},\ \mathsf{Msg}(\mathsf{id_k},\mathsf{id},\mathsf{Ex\text{-}req},\mathsf{a},\perp)\odot\mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle \\ & \qquad\qquad \textit{if}\ \ \mathsf{dir}\neq\epsilon\ \ \textit{and}\ \ \mathsf{a}\notin\mathsf{trecs} \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Ex},\mathsf{R}(\mathsf{dir})))\mid\mathsf{m},\ \mathsf{in},\ \mathsf{out}\otimes\mathsf{multicast}(\mathsf{id},\mathsf{dir},\mathsf{Inv\text{-}req},\mathsf{a},\perp),\ \mathsf{Trec}(\mathsf{a},(\mathsf{id_k},\mathsf{Ex\text{-}req}))\mid\mathsf{trecs}\rangle \end{array}$$

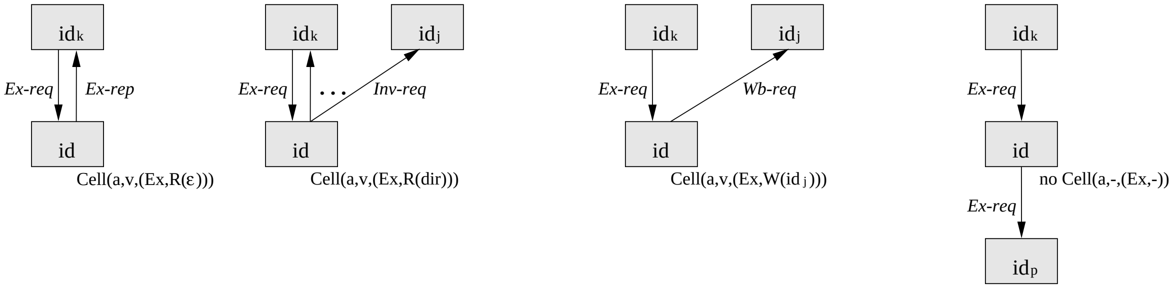


Figure 13: Memory id receives an Ex-req message from child $\mathsf{id_k}$

Receive-Ex-Req-And-Send-Wb-Req Rule $\surd$

$\langle \mathsf{id},\ \mathsf{Cell(a,v,(Ex,W(id_j)))} \mid \mathsf{m},\ \mathsf{Msg(id_k, id, Ex\text{-}req, a, \bot)} \odot \mathsf{in},\ \mathsf{out},\ \mathsf{trecs} \rangle$

if $\mathsf{id_k} \neq \mathsf{id_j}$ *and* $\mathsf{a} \notin \mathsf{trecs}$

$\longrightarrow$ $\langle \mathsf{id},\ \mathsf{Cell(a,v,(Ex,W(id_j)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg(id, id_j, Wb\text{-}req, a, \bot)},\ \mathsf{Trec(a, (id_k, Ex\text{-}req))} \mid \mathsf{trecs} \rangle$

Receive-Ex-Req-And-Send-Ex-Req Rule

$\langle \mathsf{id},\ \mathsf{m},\ \mathsf{Msg(id_k, id, Ex\text{-}req, a, \bot)} \odot \mathsf{in},\ \mathsf{out},\ \mathsf{trecs} \rangle$

if $\mathsf{Cell(a,\text{-},(Ex,\text{-}))} \notin \mathsf{m}$ *and* $\mathsf{a} \notin \mathsf{trecs}$

$\longrightarrow$ $\langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg(id, id_p, Ex\text{-}req, a, \bot)},\ \mathsf{Trec(a, (id_k, Ex\text{-}req))} \mid \mathsf{trecs} \rangle$

where $\mathsf{id_p} = \mathsf{parent(id)}$

Discussion: It is worth pointing out that certain predicates in the caching request rules are always true. These predicates are not omitted purely for clarity reason: (1) when memory id receives a Sh-req message from child $\mathsf{id_k}$, the cell's Hstate cannot be R(dir) ($\mathsf{id_k} \in \mathsf{dir}$) or $\mathsf{W(id_k)}$, since the child cannot have a shared or exclusive copy at the time; (2) when memory id receives an Ex-req message from child $\mathsf{id_k}$, the cell's Hstate cannot be $\mathsf{W(id_k)}$, since the child has no exclusive copy at the time. These invariants can be easily verified once we notice that in HCN-base, a memory cannot send data to a child without first receiving a request from the child, and a child cannot send the same caching request more than once.

An incoming Sh-req or Ex-req message cannot be processed if the address is already in a transient state. For example, if memory id receives several Sh-req and Ex-req messages (from different child memories) regarding the same address, it selects one to process. Suppose the accessed data is not cached, the selected request needs to be suspended before it can be processed to completion. Other request messages cannot be processed before the suspended message is resumed. This greatly simplifies the protocol design while the performance is not compromised.

11.3 Parent-to-Child Request Rules

Wb-Request Rules: When memory id receives a Wb-req message from parent $\mathsf{id_p}$, it processes the message as follows, provided the address is not in a transient state:

- (Hit) If memory id has the data in the (Ex,R(dir)) state, it sends a Wb-rep message to parent $\mathsf{id_p}$ and changes the cell's Cstate to Sh.

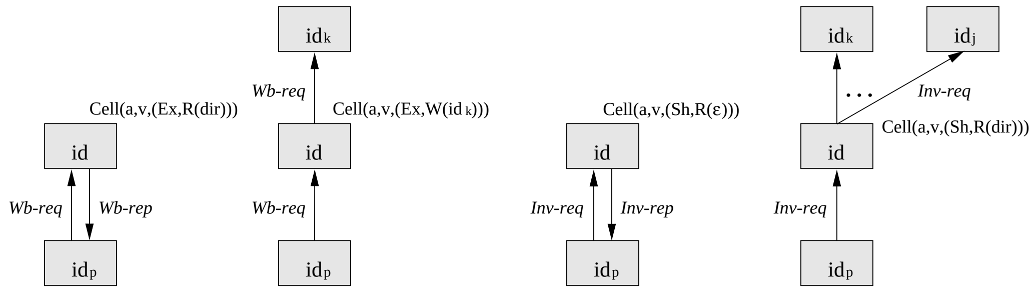


Figure 14: Memory id receives a $\mathsf{Wb\text{-}req/Inv\text{-}req}$ message from parent $\mathsf{id_p}$

- (Hit but stale data) If memory id has the data and the cell's state is $(\mathsf{Ex},\mathsf{W}(\mathsf{id_k}))$, it suspends the request and sends a $\mathsf{Wb\text{-}req}$ message to child $\mathsf{id_k}$.

Receive-Wb-Req-And-Send-Wb-Rep Rule $\surd$

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{Cell(a,v,(Ex,R(dir)))} \mid \mathsf{m},\ \mathsf{Msg}(\mathsf{id_p},\mathsf{id},\mathsf{Wb\text{-}req},\mathsf{a},\perp) \odot \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle \qquad \textit{if} \ \ \mathsf{a} \notin \mathsf{trecs} \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{Cell(a,v,(Sh,R(dir)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg}(\mathsf{id},\mathsf{id_p},\mathsf{Wb\text{-}rep},\mathsf{a},\mathsf{v}),\ \mathsf{trecs}\rangle \end{array}$$

Receive-Wb-Req-And-Send-Wb-Req Rule

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(Ex,W(id_k))}) \mid \mathsf{m},\ \mathsf{Msg}(\mathsf{id_p},\mathsf{id},\mathsf{Wb\text{-}req},\mathsf{a},\perp) \odot \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle \qquad \textit{if} \ \ \mathsf{a} \notin \mathsf{trecs} \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(Ex,W(id_k))}) \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg}(\mathsf{id},\mathsf{id_k},\mathsf{Wb\text{-}req},\mathsf{a},\perp),\ \mathsf{Trec}(\mathsf{a},(\mathsf{id_p},\mathsf{Wb\text{-}req})) \mid \mathsf{trecs}\rangle \end{array}$$

Discussion: In HCN-base, a memory cannot write an exclusive cell's data back to its parent without first receiving a request from the parent, and the parent cannot send the same Wb-req request more than once. This indicates that when memory id receives a Wb-req message from its parent, the memory must have an exclusive cell for the accessed address.

An incoming Wb-req message cannot be processed if the address is already in a transient state. Notice that the resumption of the suspended message (which caused the address to enter the transient state) cannot depend on the processing of the blocked Wb-req message. This is simply because a memory never needs to send any request message to its parent regarding an address that it has the exclusive ownership (it may need to send request messages to its child memories). In other words, it is safe to block the Wb-req message and such delay cannot cause deadlocks.

Inv-Request Rules: When memory id receives an Inv-req message from parent $\mathsf{id_p}$, it processes the message as follows, *regardless of whether the address is in a transient state*:

- (Hit) If memory id has the data in the $(\mathsf{Sh},\mathsf{R}(\epsilon))$ state, it purges the cell from the memory and sends an Inv-rep message to parent $\mathsf{id_p}$.

- (Hit but outstanding reads) If memory id has the data and the cell's state is $(\mathsf{Sh},\mathsf{R}(\mathsf{dir}))$ where $\mathsf{dir} \neq \epsilon$, it suspends the request and multicasts an Inv-req message to the child memories specified by directory dir.

Receive-Inv-Req-And-Send-Inv-Rep Rule ✓

$\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Sh},\mathsf{R}(\epsilon))) \mid \mathsf{m},\ \mathsf{Msg}(\mathsf{id}_\mathrm{p}, \mathsf{id}, \mathsf{Inv\text{-}req}, \mathsf{a}, \perp) \odot \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle$ ~~$\mathit{if}\ \ \mathsf{a} \notin \mathsf{trecs}$~~

$\longrightarrow\ \langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg}(\mathsf{id}, \mathsf{id}_\mathrm{p}, \mathsf{Inv\text{-}rep}, \mathsf{a}, \perp),\ \mathsf{trecs}\rangle$

Receive-Inv-Req-And-Multicast-Inv-Req Rule

$\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Sh},\mathsf{R}(\mathsf{dir}))) \mid \mathsf{m},\ \mathsf{Msg}(\mathsf{id}_\mathrm{p}, \mathsf{id}, \mathsf{Inv\text{-}req}, \mathsf{a}, \perp) \odot \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle$

$\mathit{if}\ \ \mathsf{dir} \neq \epsilon$ ~~$\mathit{and}\ \ \mathsf{a} \notin \mathsf{trecs}$~~

$\longrightarrow\ \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Sh},\mathsf{R}(\mathsf{dir}))) \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{multicast}(\mathsf{id}, \mathsf{dir}, \mathsf{Inv\text{-}req}, \mathsf{a}, \perp),\ \mathsf{Trec}(\mathsf{a}, (\mathsf{id}_\mathrm{p}, \mathsf{Inv\text{-}req})) \mid \mathsf{trecs}\rangle$

Discussion: In HCN-base, a memory cannot invalidate a shared cell without first receiving a request from the parent, and the parent cannot send the same Inv-req more than once. Therefore, when memory id receives an Inv-req message from its parent, the memory must have a shared cell for the accessed address.

Notice that an Inv-req message can be processed regardless of whether the accessed address is in a transient state or not. This is crucial to ensure the protocol liveness. Consider the scenario that a memory receives two Ex-req messages from two child memories. Either of the two request messages can be selected for processing. Suppose the memory has the exclusive ownership of the address but the data is shared by the two child memories. In order to process the selected Ex-req message, an Inv-req message is sent to both the child memories (see the *Receive-Ex-Req-And-Multicast-Inv-Req* rule). Deadlock can happen if the Inv-req message is blocked in either child memory, since the Ex-req message can be processed to completion before the shared copies are invalidated. Detecting such dependences is extremely important to eliminate all potential deadlock.

Generally speaking, there can be at most one transient record for each address at any time. However, when an Inv-req message is processed while the address is already in a transient state, another transient record can be created since the Inv-req message may also need to be suspended before it can be processed to completion. In HCN-base, this is the only case that two transient records exist for the same address.

11.4 Parent-to-Child Reply Rules

A caching reply message can always be processed immediately. When a caching reply is received, the corresponding transient record must contain a suspended memory access instruction or caching request message (from some child).

Sh-Reply Rules: When memory id receives a Sh-rep message from its parent, it processes the message as follows:

- If memory id is an L1 cache, it caches the data in the $(\mathsf{Sh},\mathsf{R}(\epsilon))$ state. The suspended Load instruction is resumed and executed.
- If memory id is a non-innermost memory, it caches the data in the $(\mathsf{Sh},\mathsf{R}(\mathsf{id}_\mathrm{k}))$ state, where id_k is the source of the suspended Sh-req message. The suspended Sh-req is resumed and a Sh-rep message is sent to memory id_k.

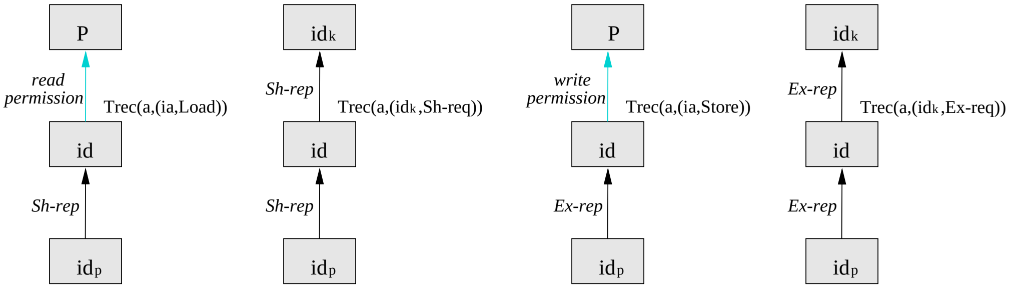


Figure 15: Memory id receives a Sh-rep/Ex-rep message from parent $\mathsf{id_p}$

Receive-Sh-Rep-And-Execute-Load Rule $\surd$

$$\mathsf{Sys}(\langle \mathsf{id},\ \mathsf{m},\ \mathsf{Msg}(\mathsf{id_p}, \mathsf{id}, \mathsf{Sh\text{-}rep}, \mathsf{a}, \mathsf{v}) \odot \mathsf{in},\ \mathsf{out},\ \mathsf{Trec}(\mathsf{a}, (\mathsf{ia},\mathsf{Load})) \mid \mathsf{trecs}\rangle,\ \ \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{prog}))$$

$$\textit{if}\ \ \mathsf{prog[ia]}\ =\ \mathsf{r} := \mathsf{Load}(\mathsf{r_1})\ \ \textit{and}\ \ \mathsf{a} = \mathsf{rf}[\mathsf{r_1}]$$

$$\longrightarrow\ \ \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(Sh,R(}\epsilon\mathsf{)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle,\ \ \mathsf{Proc}(\mathsf{ia}{+}1,\ \mathsf{rf}[\mathsf{r} := \mathsf{v}],\ \mathsf{prog}))$$

Receive-Sh-Rep-And-Send-Sh-Rep Rule

$$\langle \mathsf{id},\ \mathsf{m},\ \mathsf{Msg}(\mathsf{id_p}, \mathsf{id}, \mathsf{Sh\text{-}rep}, \mathsf{a}, \mathsf{v}) \odot \mathsf{in},\ \mathsf{out},\ \mathsf{Trec}(\mathsf{a}, (\mathsf{id_k},\mathsf{Sh\text{-}req})) \mid \mathsf{trecs}\rangle$$

$$\longrightarrow\ \ \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(Sh,R(id_k)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg}(\mathsf{id}, \mathsf{id_k}, \mathsf{Sh\text{-}rep}, \mathsf{a}, \mathsf{v}),\ \mathsf{trecs}\rangle$$

Ex-Reply Rules: When memory id receives an Ex-rep message from its parent, it processes the message as follows:

- If memory id is an L1 cache, it caches the data in the $(\mathsf{Ex,R(}\epsilon\mathsf{)})$ state. The suspended Store instruction is resumed and executed.
- If memory id is a non-innermost memory, it caches the data in the $(\mathsf{Ex,W(id_k)})$ state, where $\mathsf{id_k}$ is the source of the suspended Ex-req message. The suspended Ex-req is resumed and an Ex-rep message is sent to memory $\mathsf{id_k}$.

Receive-Ex-Rep-And-Execute-Store Rule $\surd$

$$\mathsf{Sys}(\langle \mathsf{id},\ \mathsf{m},\ \mathsf{Msg}(\mathsf{id_p}, \mathsf{id}, \mathsf{Ex\text{-}rep}, \mathsf{a}, \mathsf{u}) \odot \mathsf{in},\ \mathsf{out},\ \mathsf{Trec}(\mathsf{a}, (\mathsf{ia},\mathsf{Store})) \mid \mathsf{trecs}\rangle,\ \ \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{prog}))$$

$$\textit{if}\ \ \mathsf{prog[ia]}\ =\ \mathsf{Store}(\mathsf{r_1}, \mathsf{r_2})\ \ \textit{and}\ \ \mathsf{a} = \mathsf{rf}[\mathsf{r_1}]$$

$$\longrightarrow\ \ \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(Ex,R(}\epsilon\mathsf{)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle,\ \ \mathsf{Proc}(\mathsf{ia}{+}1,\ \mathsf{rf},\ \mathsf{prog}))\quad \textit{where}\ \ \mathsf{v} = \mathsf{rf}[\mathsf{r_2}]$$

Receive-Ex-Rep-And-Send-Ex-Rep Rule

$$\langle \mathsf{id},\ \mathsf{m},\ \mathsf{Msg}(\mathsf{id_p}, \mathsf{id}, \mathsf{Ex\text{-}rep}, \mathsf{a}, \mathsf{v}) \odot \mathsf{in},\ \mathsf{out},\ \mathsf{Trec}(\mathsf{a}, (\mathsf{id_k},\mathsf{Ex\text{-}req})) \mid \mathsf{trecs}\rangle$$

$$\longrightarrow\ \ \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a,v,(Ex,W(id_k)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg}(\mathsf{id}, \mathsf{id_k}, \mathsf{Ex\text{-}rep}, \mathsf{a}, \mathsf{v}),\ \mathsf{trecs}\rangle$$

Discussion: For other memory units, it is an atomic operation to process an incoming reply message and resume the suspended instruction or request message. The atomicity is needed to ensure the protocol liveness. We can break this atomicity by simply splitting each caching reply rule to two separate rules, one to receive the incoming reply message and cache the data, the other to resume the suspended instruction or request. However, it is then possible that before the suspended instruction or request is resumed, the cached data is invalidated

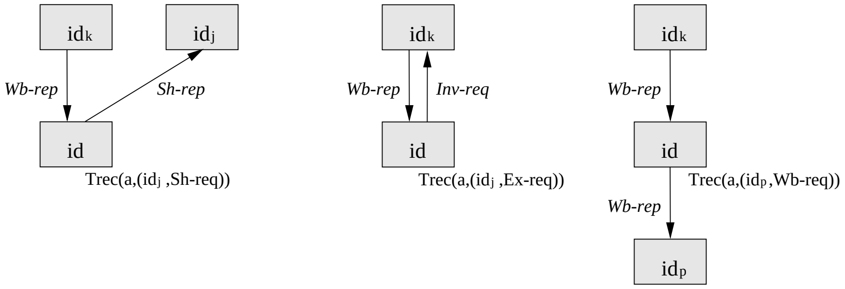


Figure 16: Memory id receives a Wb-rep message from child id_k

or written back to its parent (because a de-caching request from the parent is received). This can result in livelock situations that no cache miss can be serviced in finite time.

From the point of view of implementations, it may not be practical to enforce this atomicity. For modern microprocessors, we have little control when an L1 cell will be accessed by a stalled memory access instruction, or when an L2 cell will be cached to an L1 cache that has been waiting for the data. Fortunately, the latency is relatively small and usually can be neglected. For example, a stalled Load instruction is often repeatedly *retried* and will be executed immediately once the data is available in the L1 cache.

11.5 Child-to-Parent Reply Rules

As caching reply messages, a de-caching reply message can also be processed immediately. When a de-caching reply is received, the corresponding transient record must contain a suspended caching request (from some child) or de-caching request (from the parent).

Wb-Reply Rules: When memory id receives a Wb-rep message from child id_k, it processes the message as follows:

- If a Sh-req message is suspended, memory id updates the cell's value and sets the cell's state to (Ex,R($\text{id}_\text{k}|\text{id}_\text{j}$)), where id_j is the source of the suspended message. The suspended Sh-req message is resumed and a Sh-rep message is sent to memory id_j.

- If an Ex-req message is suspended, memory id updates the cell's value, sets the cell's state to (Ex,R(id_k)), and sends an Inv-req message to memory id_j, which is the source of the suspended Ex-req message.

  Notice that the suspended message cannot be resumed since the shared copy in memory id_k must be first invalidated. It will be resumed when the corresponding Inv-rep message is received (see the *Receive-Inv-Rep-And-Send-Ex-Rep* rule).

- If a Wb-req message is suspended, memory id updates the cell's value and sets the cell's state to (Sh,R(id_k)). The suspended Wb-req message is resumed and a Wb-rep message is sent to the parent.

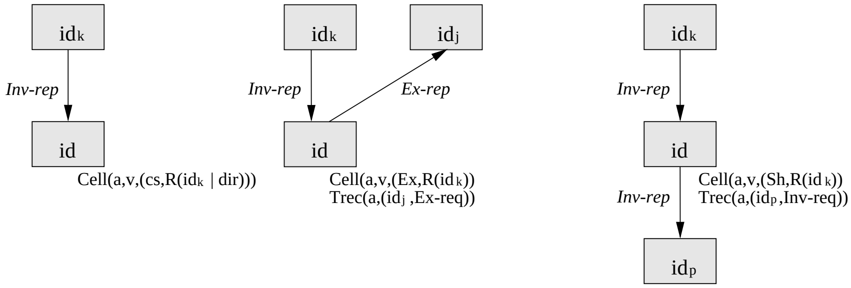


Figure 17: Memory id receives an $\mathsf{Inv\text{-}rep}$ message from child $\mathsf{id_k}$

Receive-Wb-Rep-And-Send-Sh-Rep Rule $\surd$

$$\langle \mathsf{id,\ Cell(a,u,(Ex,W(id_k)))\ |\ m,\ Msg(id_k, id, Wb\text{-}rep, a, v) \odot in,\ out,\ Trec(a, (id_j,Sh\text{-}req))\ |\ trecs} \rangle$$
$$\longrightarrow \quad \langle \mathsf{id,\ Cell(a,v,(Ex,R(id_k|id_j)))\ |\ m,\ in,\ out \otimes Msg(id, id_j, Sh\text{-}rep, a, v),\ trecs} \rangle$$

Receive-Wb-Rep-And-Send-Inv-Req Rule $\surd$

$$\langle \mathsf{id,\ Cell(a,u,(Ex,W(id_k)))\ |\ m,\ Msg(id_k, id, Wb\text{-}rep, a, v) \odot in,\ out,\ Trec(a, (id_j,Ex\text{-}req))\ |\ trecs} \rangle$$
$$\longrightarrow \quad \langle \mathsf{id,\ Cell(a,v,(Ex,R(id_k)))\ |\ m,\ in,\ out \otimes Msg(id, id_k, Inv\text{-}req, a, v),\ Trec(a, (id_j,Ex\text{-}req))\ |\ trecs} \rangle$$

Receive-Wb-Rep-And-Send-Wb-Rep Rule

$$\langle \mathsf{id,\ Cell(a,u,(Ex,W(id_k)))\ |\ m,\ Msg(id_k, id, Wb\text{-}rep, a, v) \odot in,\ out,\ Trec(a, (id_p,Wb\text{-}req))\ |\ trecs} \rangle$$
$$\longrightarrow \quad \langle \mathsf{id,\ Cell(a,v,(Sh,R(id_k)))\ |\ m,\ in,\ out \otimes Msg(id, id_p, Wb\text{-}rep, a, v),\ trecs} \rangle$$

Inv-Reply Rules: When memory id receives an $\mathsf{Inv\text{-}rep}$ message from child $\mathsf{id_k}$, it processes the message as follows:

- If the directory shows that more $\mathsf{Inv\text{-}rep}$ messages (from other child memories) are impending, memory id simply removes identifier $\mathsf{id_k}$ from the directory. The suspended message is not resumed.
- If all invalidation requests (regarding the address) have been acknowledged, and an $\mathsf{Ex\text{-}req}$ message is suspended, memory id sets the cell's state to $\mathsf{(Ex,W(id_j))}$, where $\mathsf{id_j}$ is the source of the suspended message. The suspended $\mathsf{Ex\text{-}req}$ message is resumed and an $\mathsf{Ex\text{-}rep}$ message is sent to memory $\mathsf{id_j}$.
- If all invalidation requests (regarding the address) have been acknowledged, all an $\mathsf{Inv\text{-}req}$ message is suspended, memory id purges the cell from the memory. The suspended $\mathsf{Inv\text{-}req}$ message is resumed and an $\mathsf{Inv\text{-}rep}$ message is sent to the parent.

Receive-Inv-Rep-Pending Rule $\surd$

$$\langle \mathsf{id,\ Cell(a,v,(cs,R(id_k|dir)))\ |\ m,\ Msg(id_k, id, Inv\text{-}rep, a, \perp) \odot in,\ out,\ trecs} \rangle \qquad \textit{if} \quad \mathsf{dir} \neq \epsilon$$
$$\longrightarrow \quad \langle \mathsf{id,\ Cell(a,v,(cs,R(dir)))\ |\ m,\ in,\ out,\ trecs} \rangle$$

Receive-Inv-Rep-And-Send-Ex-Rep Rule $\surd$

$$\langle \mathsf{id,\ Cell(a,v,(Ex,R(id_k)))\ |\ m,\ Msg(id_k, id, Inv\text{-}rep, a, \perp) \odot in,\ out,\ Trec(a, (id_j,Ex\text{-}req))\ |\ trecs} \rangle$$
$$\longrightarrow \quad \langle \mathsf{id,\ Cell(a,v,(Ex,W(id_j)))\ |\ m,\ in,\ out \otimes Msg(id, id_j, Ex\text{-}rep, a, v),\ trecs} \rangle$$

Receive-Inv-Rep-And-Send-Inv-Rep Rule

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{Cell(a,v,(Sh,R(id_k)))} \mid \mathsf{m},\ \mathsf{Msg(id_k, id, Inv\text{-}rep, a, \bot)} \odot \mathsf{in},\ \mathsf{out},\ \mathsf{Trec(a, (id_p, Inv\text{-}req))} \mid \mathsf{trecs} \rangle \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg(id, id_p, Inv\text{-}rep, a, \bot)},\ \mathsf{trecs} \rangle \end{array}$$

11.6 Message Passing Rules

Message passing rules remain unchanged. Message passing can happen only between the memories that have the parent-child relationship. Since the constructor of outgoing queues is associative and commutative, any message in an outgoing queue can be brought to the front of the queue. Therefore, messages can be delivered in any order. Even for messages regarding the same address and between the same source and destination, their arrival order can be arbitrary.

Message-Passing-To-Child Rule $\surd$

$$\begin{array}{ll} & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{Msg(id, id_k, cmd, a, v)} \otimes \mathsf{out},\ \mathsf{trecs} \rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k},\ \mathsf{trecs_k} \rangle,\ \mathsf{eu_k}) \mid \mathsf{sg}) \\ \longrightarrow & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out},\ \mathsf{trecs} \rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{in_k} \odot \mathsf{Msg(id, id_k, cmd, a, v)},\ \mathsf{out_k},\ \mathsf{trecs_k} \rangle,\ \mathsf{eu_k}) \mid \mathsf{sg}) \end{array}$$

Message-Passing-To-Parent Rule $\surd$

$$\begin{array}{ll} & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out},\ \mathsf{trecs} \rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{Msg(id_k, id, cmd, a, v)} \otimes \mathsf{out_k},\ \mathsf{trecs_k} \rangle,\ \mathsf{eu_k}) \mid \mathsf{sg}) \\ \longrightarrow & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{m},\ \mathsf{in} \odot \mathsf{Msg(id_k, id, cmd, a, v)},\ \mathsf{out},\ \mathsf{trecs} \rangle,\ \ \mathsf{Sys}(\langle \mathsf{id_k},\ \mathsf{m_k},\ \mathsf{in_k},\ \mathsf{out_k},\ \mathsf{trecs_k} \rangle,\ \mathsf{eu_k}) \mid \mathsf{sg}) \end{array}$$

12 Verification of the HCN-base Protocol

The HCN-base protocol implements Sequential Consistency and guarantees that each individual processor can always make progress. When a program is executed on a DSM system that maintains cache coherence by employing the HCN-base protocol, the program behaves exactly the same as if it were running on a sequentially consistent machine. Moreover, no processor in the DSM system can be stalled indefinitely due to cache misses.

12.1 Soundness of HCN-base

The soundness is obvious because all coherence actions in HCN-base are performed according to the HCN rules. It can be shown that all HCN-base rules can be derived from the HCNr rules, which include the HCN rules and the directive rules. The table below gives the sequence of HCNr rules that are applied while deriving each HCN-base rule (if a rule can be applied more than once in the derivation, it is marked with '+'). Notice the transient records are disregarded in the derivation because they behave as extra predicates for the imperative and directive actions.

The HCN-base **Rule**	**The Sequence of** HCN **Rules**
Read-Cache-Hit	*HCN-Load*
Write-Cache-Hit	*HCN-Store*
Read-Cache-Miss	*Generate-Req-To-Parent* (Sh-req)
Write-Cache-Miss	*Generate-Req-To-Parent* (Ex-req)
Receive-Sh-Req-And-Send-Sh-Rep	*Send-Sh-Rep* + *Discard-Incoming-Req*
Receive-Sh-Req-And-Send-Wb-Req	*Generate-Req-To-Child* (Wb-req) + *Discard-Incoming-Req*
Receive-Sh-Req-And-Send-Sh-Req	*Generate-Req-To-Parent* (Sh-req) + *Discard-Incoming-Req*
Receive-Ex-Req-And-Send-Ex-Rep	*Send-Ex-Rep* + *Discard-Incoming-Req*
Receive-Ex-Req-And-Multicast-Inv-Req	*Generate-Req-To-Child*$^+$ (Inv-req) + *Discard-Incoming-Req*
Receive-Ex-Req-And-Send-Wb-Req	*Generate-Req-To-Child* (Wb-req) + *Discard-Incoming-Req*
Receive-Ex-Req-And-Send-Ex-Req	*Generate-Req-To-Parent* (Ex-req) + *Discard-Incoming-Req*
Receive-Wb-Req-And-Send-Wb-Rep	*Send-Wb-Rep* + *Discard-Incoming-Req*
Receive-Wb-Req-And-Send-Wb-Req	*Generate-Req-To-Child* (Wb-req) + *Discard-Incoming-Req*
Receive-Inv-Req-And-Send-Inv-Rep	*Send-Inv-Rep* + *Discard-Incoming-Req*
Receive-Inv-Req-And-Multicast-Inv-Req	*Generate-Req-To-Child*$^+$ (Inv-req) + *Discard-Incoming-Req*
Receive-Sh-Rep-And-Execute-Load	*Receive-Sh-Rep* + *HCN-Load*
Receive-Sh-Rep-And-Send-Sh-Rep	*Receive-Sh-Rep* + *Send-Sh-Rep*
Receive-Ex-Rep-And-Execute-Store	*Receive-Ex-Rep* + *HCN-Store*
Receive-Ex-Rep-And-Send-Ex-Rep	*Receive-Ex-Rep* + *Send-Ex-Rep*
Receive-Wb-Rep-And-Send-Sh-Rep	*Receive-Wb-Rep* + *Send-Sh-Rep*
Receive-Wb-Rep-And-Send-Inv-Req	*Receive-Wb-Rep* + *Generate-Req-To-Child* (Inv-req)
Receive-Wb-Rep-And-Send-Wb-Rep	*Receive-Wb-Rep* + *Send-Wb-Rep*
Receive-Inv-Rep-Pending	*Receive-Inv-Rep*
Receive-Inv-Rep-And-Send-Ex-Rep	*Receive-Inv-Rep* + *Send-Ex-Rep*
Receive-Inv-Rep-And-Send-Inv-Rep	*Receive-Inv-Rep* + *Send-Inv-Rep*
Message-Passing-To-Child	*Message-Passing-To-Child*
Message-Passing-To-Parent	*Message-Passing-To-Parent*

Lemma 23 All HCN-base rules can be derived from the HCNr rules.

The HCN-base protocol is *not* a complete implementation of HCN, because coherence actions that can happen in HCN are not all allowed to happen in HCN-base. While HCN allows memory cells to be cached or de-cached at any time provided that memory coherence is not violated, such operations can happen in HCNr only when they are necessary to service cache misses. For example, in HCN, a memory can send a shared copy to a child if it has the valid data. However, in HCN-base, this cannot happen unless the child has issued a request. The child cannot issue such request unless it has received a request from one of its children, or it is an L1 cache and a cache miss happens in the L1 cache.

Since all HCN-base rules can be derived from HCNr rules, HCN-base is a partial implementation of HCN. The soundness alone is sufficient to ensure that the protocol behavior conforms to Sequential Consistency. It is worth noting that although HCN-base is just a partial implementation of HCN and HC, it is a complete implementation of SC.

12.2 Liveness of HCN-base

Theorem 24 HCN-base enforces strong liveness, i.e., a cache miss on any processor can be serviced in finite time.

This part will be added soon. The buffer management section will also be rewritten.

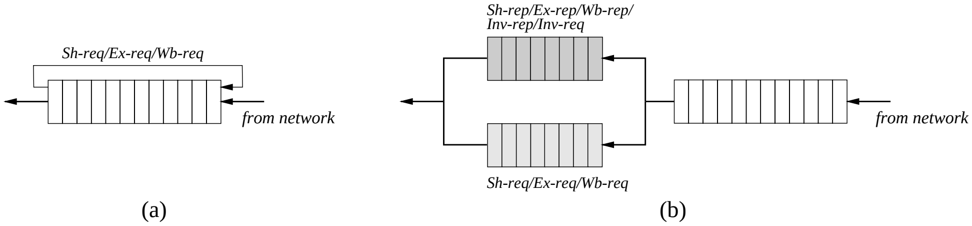


Figure 18: Two Simple Buffer Management Strategies

13 Buffer Management

A rigorous proof that HCN-base is deadlock free is based on the case analysis of the relative positions of requesters and the location of the data in the memory hierarchy, and is quite tedious. Next we discuss two interrelated issues: incoming queue management and liveness.

Both reply and request messages may be present in an incoming queue. To avoid deadlocks, it is essential that reply messages not be blocked by the request messages, and an enabled request message not be blocked by other blocked requests. We did not have to pay attention to this problem so far because we assumed that the messages in the queue could commute with each other. Now we develop a concrete buffer management strategy that is fair, deadlock free and implementable.

Figure 18(a) gives a simple buffer management strategy involving a single FIFO queue. In HCN-base, a reply message or an Inv-req message can be processed immediately upon its arrival. The other request messages (i.e. Sh-req, Ex-req or Wb-req), if they cannot be processed when at the head of the queue, are simply put at the end of the incoming queue. This ensures that the memory cannot go idle as long as there is an enabled request in the queue. This implies that if there are cache misses in the system, then within a finite amount of time, one of the cache misses will be serviced. However, this simple buffer management strategy does not ensure the liveness for each processor. In theory, it is possible that a certain unlucky Sh-req, Ex-req or Wb-req message never gets an opportunity to be processed because the requests from other processors always beat it. The probability of this type of starvation may be very small in practice. A deadlock can also result if the queue cannot accommodate all the outstanding requests. The worst case for the queue length is determined by the number of processors and is usually not a serious issue.

Figure 18 (b) ensures the liveness for each process by employing two buffer queues for incoming messages, one for reply messages and Inv-req messages, and the other for Sh-req, Ex-req and Wb-req messages. This organization puts all the blockable requests in a separate queue and processes them in the FIFO order. This organization guarantees fairness for all requests.

An obvious drawback of the buffer management described above is that a blocked request message may unnecessarily prevent the processing of different addresses. Figure 19 shows the organization used in the protocol we designed for the Start-Voyager machine: blocked request messages for different addresses are maintained in different queues so that they cannot block each other. This strategy can result in better performance. This completes the description

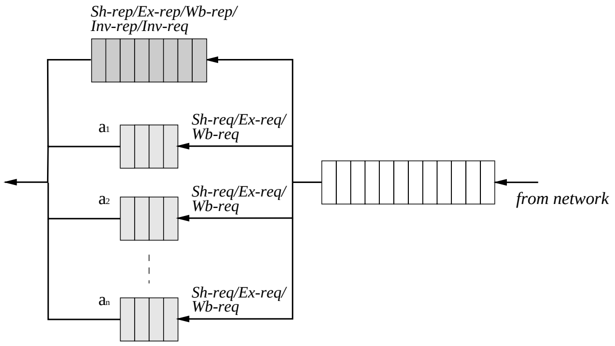


Figure 19: HCN-base Buffer Management Strategy

of a realistic protocol for DSM's with a hierarchy of caches.

14 Summary and Research-in-Progress

This paper has made the following contributions:

A new two-phase Imperative-Directive methodology for designing cache coherence protocols: This methodology separates the correctness and the liveness concerns in the design process. In the imperative design phase, we ignore the liveness issues and design a preliminary protocol by giving a set of rules that can only cause state transitions that are consistent with the memory model. In the directive design phase, we specify the precise conditions for invoking the imperative rules by incorporating directive messages and transient records. The key point is that improper additional conditions for invoking imperative actions cannot affect the correctness of the system although they may cause deadlocks or livelocks. Protocols designed with this methodology are often easier to understand, modify and reason about. For example, the final protocol presented here in 27 rules is far more tractable than its 3000 line implementation in C for StarT-Voyager [3].

Successive refinement of protocols to incorporate implementation issues: The HC model ignored the DSM issues but made it easy to derive the rules for the HCN model, which had a network and distributed control. Similarly, in the directive design phase, we separated the message buffer management issue by first assuming that messages in the input queue could commute and thus avoid blocking enabled messages. The separation of buffer management results in protocols with better modularity.

Protocol verification against a memory model: We specify both the memory model and the protocol using the same formalism. TRS's are well suited to describe asynchronous computations, and allow us to formulate the correctness question precisely. The designer has to prove three conditions (soundness, completeness and connection) with respect to the memory model to show that a protocol implements the memory model correctly. Our successive refinement approach to protocol design makes these proofs much easier to develop and understand. In fact for us the design and verification process is totally intermingled.

Our approach to verification is different from others [27, 21] because they concentrate on proving certain invariants. Generally, it is difficult to determine if one has a sufficient set of invariants to ensure that the behaviors are consistent with the memory model. In the course of our proofs one ends up proving many similar invariants but their need is derived in a systematic way. It is important to point out that, for sophisticated protocols, the tedious part of the correctness proof (e.g., case analysis) can be automated using a model checker tool.

A complete protocol to implement Sequential Consistency on a DSM with a hierarchical caches: The protocol we have presented to illustrate our methodology is a simpler version of one of the protocols implemented on StarT-Voyager. The final version of the protocol is free from deadlock, and ensures that every processor makes progress. Some potential optimizations have been excluded from the protocol for the sake of clarity. An optimized version of the HCN-base protocol along with all the proofs can be found in [25].

Related Research-in-Progress: Needless to say, the Imperative-Directive methodology can be applied to designing other more sophisticated cache protocols. Cachet [23] is a toolbox containing cache-coherence primitives that can be used to build protocols on-the-fly. Cachet implements a relaxed memory model and employs two critical techniques, instant-writes (to reduce write latency) and lazy-flushes (to decrease the effect of false sharing). Cachet defines a set of coherence primitives for each state for both the cache and the home memory engines. Memory consistency and protocol liveness are guaranteed regardless of how the primitives are chosen to execute, although a smart selection can result in better performance.

We have applied the TRS framework to modeling and verification of out-of-order and speculative microprocessors [26]. We are also exploring hardware synthesis from the type of TRS's presented in this paper. The preliminary results based on hand compilation of TRS rules into synthesizable Verilog look promising. Our goal is to produce an architecture description language and a compiler that will dramatically reduce the design effort required to implement complex systems.

Acknowledgment: We are thankful to Larry Rudolph, Boon Ang, Alex Caro, Derek Chiou and Keith Randall for reading the draft of this paper. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Ft. Huachuca contract DABT63-95-C-0150.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22th International Symposium On Computer Architecture*, 1995.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[3] B. S. Ang and D. Chiou. Start-Voyager: Hardware Engineering Specification. CSG Memo 385, Laboratory for Computer Science, MIT, June 1997.

[4] J. K. Archibald. The Cache Coherence Problem in Shared-Memory Multiprocessors. Phd thesis, Department of Computer Science, University of Washington, Feb. 1987.

[5] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 297–308, Padua, Italy, June 1996.

[6] M. Browne, E. Clarke, D. Dill, and B. Mishra. Automatic Verification of Sequential Circuits Using Temporal Logic. *IEEE Transaction on Computers*, pages 1035–1044, Dec. 1986.

[7] E. Clarke, E. Emerson, and A. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, Apr. 1986.

[8] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13rd International Symposium On Computer Architecture*, pages 434–442, June 1986.

[9] G.-R. Gao and V. Sarkar. Location Consistency – Stepping Beyond the Barriers of Memory Coherence and Serializability. Technical Memo 78, ACAPS Laboratory, School of Computer Science, McGill Univerisity, Dec. 1993.

[10] K. Gharachorloo. Memory Consistency Models for Shared-Memory Multiprocessors. Phd. thesis, Stanford University, 1995.

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

[12] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proceedings of the 15th International Symposium On Computer Architecture*, pages 422–431, May 1988.

[13] C. Ip and D. Dill. Better Verification Through Symmetry. In *Proceedings of the 11th International Symposium on Computer Hardware Description Languages and Their Applications*, pages 87–100, Apr. 1993.

[14] C. Ip and D. Dill. Efficient Verification of Symmetric Concurrent Systems. In *International Conference on Computer Design: VLSI in Computers and Processors*, Oct. 1993.

[15] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th International Symposium On Computer Architecture*, pages 13–21, May 1992.

[16] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, Apr. 1994.

[17] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[18] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92–103, May 1992.

[19] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification for A New Family of RISC Processors*. Morgan Kaufmann, 1994.

[20] K. McMillan. Symbolic Model Checking: An Approach to the State Explosion Problem. Ph.d dissertation, Carnegie Mellon University, May 1992.

[21] F. Pong and M. Dubois. A New Approach for the Verification of Cache Coherence Protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6, Aug. 1995.

[22] F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois. Verifying Distributed Directory-based Cache Coherence Protocols: S3.mp, a Case Study. In *EuroPar'95*, 1995.

[23] X. Shen. Cachet: A Cache Coherence Primitive Toolkit (in preparation). CSG Memo 404, Laboratory for Computer Science, MIT, Nov. 1997.

[24] X. Shen and Arvind. Processor Models. CSG Memo 400, Laboratory for Computer Science, MIT, June 1997.

[25] X. Shen and Arvind. Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols. CSG Memo 398, Laboratory for Computer Science, MIT, June 1997.

[26] X. Shen and Arvind. Modeling and Verification of ISA Implementations. In *Proceedings of the Australasian Computer Architecture Conference, Perth, Australia*, Feb. 1998.

[27] U. Stern and D. L. Dill. Automatic Verification of the SCI Cache Coherence Protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

Contents