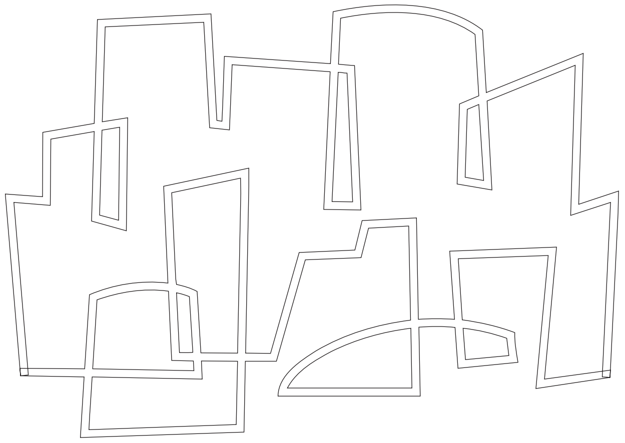
A Methodology for Designing Correct Cache Coherence for DSM Systems

Xiaowei Shen, Arvind

1997, June

Computation Structures Group
Memo 398a



A Methodology for Designing Correct Cache Coherence Protocols for DSM Systems

Computation Structures Group Memo 398 (A)

Xiaowei Shen and Arvind

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
xwshen, arvind@lcs.mit.edu

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.

A Methodology for Designing Correct Cache Coherence Protocols for DSM Systems

Xiaowei Shen and Arvind

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
xwshen, arvind@lcs.mit.edu

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 because these essentially entail multiple directives for each imperative action.

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 to implement Sequential Consistency on a DSM 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 don't 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.

1.2 Formal Verification

The verification of cache coherence protocols has gained considerable attention in recent years [4, 22]. 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.

The biggest problem with the current approaches 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. 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. While proving that a protocol implements the memory model, most of the commonly known invariants systematically show up as lemmas.

1.3 Design Methodology

In spite of the number of publications on cache coherence protocols [12, 26, 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. Is the cache state being maintained correctly? Is there a deadlock due to reordering of messages or lack of buffers in the network? Is it possible that a processor's request may never be satisfied? Answering these questions can be very 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 concern is ensuring 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 of the design process, the main concern is liveness, i.e., ensuring via *directive rules* that the system takes the appropriate imperative action. 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 in distributed systems. As we shall show, within each phase of design, we will successively refine the protocol by injecting more and more implementation concerns. Protocols designed using this methodology are often easy to understand, modify and reason about. The methodology has proved extremely effective in designing adaptive cache coherence protocols [23] because adaptability is only about directives; imperative rules remain unaffected. We illustrate our methodology by successively developing a family of cache coherence protocols to implement Sequential Consistency on a distributed shared memory system with a hierarchy of caches (see Figure 1). The final protocol we present (HCN-base), is to our knowledge, the first precise and complete description of a correct and livelock-free protocol for DSM systems with multi-level caches.

The Organization of the Paper: We begin by giving a brief introduction to our formalism, Term Rewriting Systems (TRS's), and define the notion of a correct implementation of a specification (Section 2). Next we give a TRS, the SC model, to define Sequential Consistency operationally based on a simple multiprocessor system without caches (Section 3). We then define the HC model, a directory-based cache coherence protocol for multiprocessor systems with hierarchical caches (Section 4). The HC model is refined with message passing in the HCN model (Section 5). Both HC and HCN use only imperative actions. After a general discussion of directive messages and other implementation issues (Section 6), we present HCN-base, a complete cache coherence protocol based on HCN (Section 7). The design of

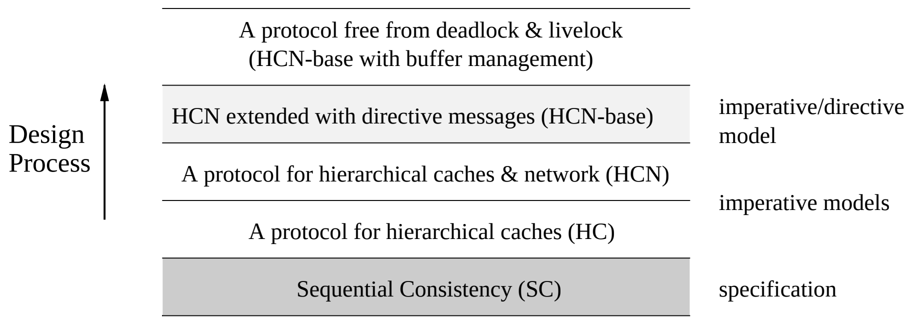

A protocol free from deadlock & livelock
(HCN-base with buffer management)
HCN extended with directive messages (HCN-base)
A protocol for hierarchical caches & network (HCN)
A protocol for hierarchical caches (HC)
Sequential Consistency (SC)
Design Process
imperative/directive model
imperative models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

Figure 1: Design Process: Models to be Discussed (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)

HCN-base is completed by giving a buffer management policy to ensure fairness of message processing (Section 8). Finally we present a summary and briefly discuss research in progress (Section 9).

The length restrictions do not allow us to present the complete proofs. However, we discuss the outline of each proof; reference [24] is a version of this paper with full proofs.

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 is a predicate.

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 applies to a term then the term remains unchanged. 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 zero or more rewriting steps ($s_1 \longrightarrow\!\!\!\!\!\rightarrow s_2$), if either $s_1 = s_2$, or there exists a term s' such that $s_1 \longrightarrow s'$ and $s' \longrightarrow\!\!\!\!\!\rightarrow s_2$.

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 '-' to represent 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 TRS B is a *complete implementation* of TRS A if there exists a pair of mapping functions g $(B \mapsto A)$ and f $(A \mapsto B)$, such that

1. **Soundness:** $s_1 \xrightarrow{B}\!\!\!\!\rightarrow s_2 \implies g(s_1) \xrightarrow{A}\!\!\!\!\rightarrow g(s_2)$;
2. **Completeness:** $s_1 \xrightarrow{A}\!\!\!\!\rightarrow s_2 \implies f(s_1) \xrightarrow{B}\!\!\!\!\rightarrow f(s_2)$;
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.

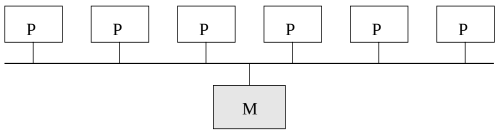
Many real implementations are not complete according to the above definition. Any sound system can be regarded as a *partial implementation* of the specification. However, some partial implementations can be pretty silly in reality: for example, an implementation that has no rewrite rule and thus makes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

3 The SC Model: Operational Semantics 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. There is no question of data replication in such a system. 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. (Notation: We use '$\|$' as meta notation in grammars to separate disjuncts; identifiers Sys, Cell, Proc etc. as constructors). 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 program counter holds the address of the instruction to be executed.

Notations: The connective '$|$' is associative and commutative. 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] the register file that differs from rf only in the content of register r.



SYS	$\equiv$	$\mathsf{Sys}(\mathrm{MEM},\ \mathrm{PG})$	*System*
MEM	$\equiv$	$\epsilon\ [\!]\ \ \mathsf{Cell(a,v)} \mid \mathrm{MEM}$	*Memory*
PG	$\equiv$	$\epsilon\ [\!]\ \ \mathrm{PROC} \mid \mathrm{PG}$	*Processor Group*

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

The processor rules are presented elsewhere [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. Due to lack of space we omit the processor rules except for the memory access operations defined below:

SC-Load Rule

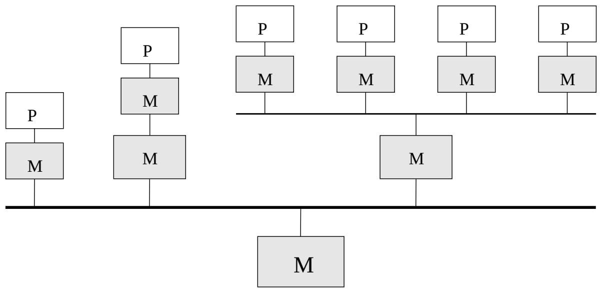
$$\begin{array}{ll} & \mathsf{Sys(Cell(a,v)\,|\,m,\ \ Proc(ia,\ rf,\ prog)\ |\ pg)} \quad \textit{if} \quad \mathsf{prog[ia]} = \ \mathsf{r := Load(r_1)} \quad \textit{and} \quad \mathsf{a} = \mathsf{rf[r_1]} \\ \longrightarrow & \mathsf{Sys(Cell(a,v)\,|\,m,\ \ Proc(ia{+}1,\ rf[r := v],\ prog)\ |\ pg)}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

SC-Store Rule

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

Since connective '|' is associative and commutative, any processor can be brought into the leftmost position. Thus, if two processors intend to execute a Store instruction to the same address, either can be allowed to proceed. However, memory access atomicity is guaranteed because the Load and Store operations are performed directly on the memory and there is no data replication.

We claim that SC is 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 define several cache coherence protocols to implement Sequential Consistency and show that they admit only SC behaviors.



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 initial state of each cell is (Ex,R(ϵ)))

4 The HC Model: A System with Hierarchical Caches

A typical distributed memory system contains a hierarchy of caches and uses different implementation technology and possibly different protocols in different parts of the system. We begin by ignoring some implementation issues associated with the communication among memory sites. In the HC model, we assume that a memory can *atomically* read and update its parent or child memories. We define a directory-based cache coherence protocol for such a system and call it the HC (Hierarchical Cache) model.

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 a set of memory cells and an identifier. The execution unit is either a single processor, or a 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 and children(id) the set of identifiers for its children.

4.1 Cache 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. 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.

Inclusion Invariants: The protocol design process is simplified 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 HC Rewriting Rules

All rewriting rules of the HC model are imperative rules and fall naturally into three categories: the rules for memory access operations, the rules for *caching* operations (i.e., moving data or ownership from parents to children), and the rules for *de-caching* operations (i.e., propagating information from children to parents).

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

HC-Load Rule

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

HC-Store Rule

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

Caching Rules: If the state of a cell in memory id is (Sh,R(dir)) or (Ex,R(dir)) and the directory shows that child id_k has not cached the data, then the parent (id) can give a shared copy to id_k, and record 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 id_k, and change the cell's Hstate to W(id_k) (see Figure 4 (a)).

Sh-Caching Rule

$$\begin{array}{ll} & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{cs},\mathsf{R}(\mathsf{dir}))) \mid \mathsf{m}\rangle,\ \ \mathsf{Sys}(\langle \mathsf{id}_\mathsf{k},\ \mathsf{m}_\mathsf{k}\rangle,\ \mathsf{eu}_\mathsf{k}) \mid \mathsf{sg})\quad \textit{if}\ \ \mathsf{id}_\mathsf{k} \notin \mathsf{dir} \\ \longrightarrow & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{cs},\mathsf{R}(\mathsf{id}_\mathsf{k}|\mathsf{dir}))) \mid \mathsf{m}\rangle,\ \ \mathsf{Sys}(\langle \mathsf{id}_\mathsf{k},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Sh},\mathsf{R}(\epsilon))) \mid \mathsf{m}_\mathsf{k}\rangle,\ \mathsf{eu}_\mathsf{k}) \mid \mathsf{sg}) \end{array}$$

Ex-Caching Rule

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

De-Caching Rules: If the state of a cell in memory id_k is (Ex,R(dir)) then it can write the most up-to-date data back to the parent (id), and change the cell's Cstate from Ex to Sh. If the state of a cell in memory id_k is (Sh,R(ϵ)) then it can invalidate (purge) the cell, and delete id_k from the directory of the corresponding cell in the parent (see Figure 4 (b)).

Writeback Rule

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

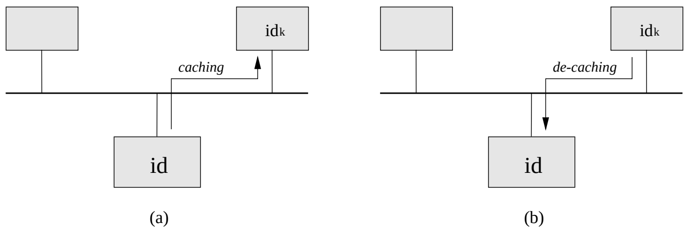


Figure 4: Caching and De-Caching Operations

Invalidate Rule

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

4.3 Correctness of HC w.r.t. SC

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

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

The CF function is easy to define once we notice that if the de-caching rules are applied repeatedly, all non-outermost memories eventually become empty. This follows from the Inclusion Invariants which can be proved by induction on rewriting steps. To show soundness, assume that $s_1 \longrightarrow s_2$ by applying rule α in HC. It can be shown by case analysis on α that 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 the corresponding rule in SC if α is a memory access rule.

The CL function simply introduces empty caches to build any memory hierarchy and sets the state of each cell (Ex,R(ϵ)) in the outermost memory. It is easy to show that each SC rule can be simulated by a sequence of HC rules. Given these mappings, it is trivial to show that CF is the inverse function of CL. (For the full proof, see [24]).

4.4 Some Optimizations as Derived Rules

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 expressive power or the correctness of the system, but may improve the performance by some measure.

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 purge the cell in one rewriting step, if the cell is not cached by any of its children (i.e., the cell is in the $(\mathsf{Ex},\mathsf{R}(\epsilon))$ state).

Pushout Rule

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

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

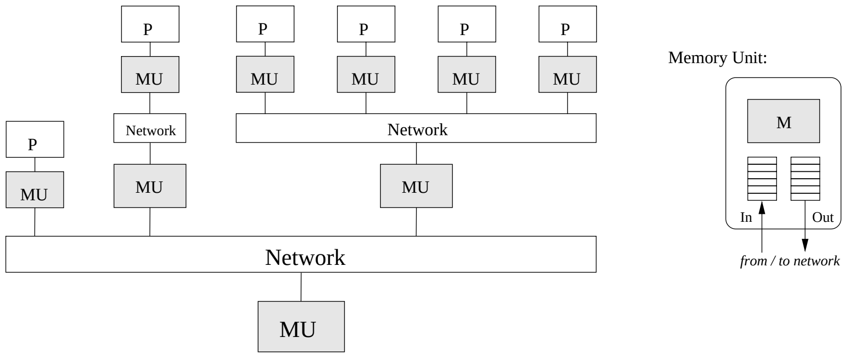
Upgrade: The upgrade operation allows a memory to obtain the exclusive ownership for a shared cell in one rewriting step, if its parent has the exclusive ownership and has not given the data to any other child (i.e., the cell is in the $(\mathsf{Ex},\mathsf{R}(\mathsf{id}_k))$ state). Upgrade is also known as Dclaim.

Upgrade Rule

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

It can be shown that applying the *Upgrade* rule has the same effect as applying *Invalidate* zero or more times, followed by *Ex-Caching*, followed by *Sh-Caching* zero or more times.

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



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 | 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*
INQ	$\equiv$	ϵ $[\!]$ MSG $\odot$ INQ	*Incoming Queue*
OUTQ	$\equiv$	ϵ $[\!]$ MSG $\otimes$ OUTQ	*Outgoing Queue*
MSG	$\equiv$	Msg($\mathrm{id}_{\mathrm{src}}, \mathrm{id}_{\mathrm{dest}}$, CMD, a, v)	*Protocol Message*
CMD	$\equiv$	Sh-rep $[\!]$ Ex-rep $[\!]$ Wb-rep $[\!]$ Inv-rep	*Message Command*

Figure 5: 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 in the (Ex,R(ϵ)) state for each address)

Sh-Forward Rule

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

Ex-Forward Rule

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

It can be shown that applying the *Sh-Forward* rule has the same effect as applying *Writeback* followed by *Sh-Caching*, while applying the *Ex-Forward* rule has the same effect as applying *Writeback* and *Invalidate* followed by *Ex-Caching*.

5 The HCN Model: Refining HC with Message Passing

The HC model assumes that coherence actions involving more than one memory can be performed with one rewriting step. For example, according to the *Ex-Caching* rule, if a memory has a cell in the (Ex,R(ϵ)) state, then it can send an exclusive copy to a child, while in the same rewriting step, the child can receive the data, and cache it in the (Ex,R(ϵ)) state. In distributed systems such rules are considered non-local, and without special hardware support, ensuring the atomicity of a local read followed by a remote write is difficult.

In this section, we derive a local version for HC rules. We define the HCN model (HC with Network) by introducing a message passing network and restricting each rule to examine and update the local memory. The grammar of HCN is given in Figure 5. As can be seen, the memory unit has two new components, an incoming message queue and an outgoing message queues built using the constructors $\odot$ and $\otimes$, respectively.

5.1 Protocol Messages and Queues

A protocol message has five fields: the source and destination memory identifiers, the message command, the address and data value. For messages that carry no data, the data field is marked empty ($\perp$). There are four types of message commands (Sh-rep, Ex-rep, Wb-rep and Inv-rep) and their impact on cache state of the receiving cell is shown in Figure 6. (The suffix "rep" stands for reply; why these commands are called replies will become clear later when we discuss requests or directives).

The constructor '$\otimes$' of the outgoing queue is associative and commutative, because we do not want messages to different destinations to block each other. Commutativity of '$\otimes$' essentially allows us to model any type of non-FIFO network.

For the incoming queue, ideally we would like to process messages in the order in which they are received. However, this may cause deadlocks or livelocks unless messages that cannot be processed temporarily are properly buffered so that other messages can be processed. Protocol design can be simplified by considering the buffer management as a separate issue. This can be achieved simply by assuming that the constructor '$\odot$' of the incoming queue is associative and commutative, which essentially allows us to process any message in the

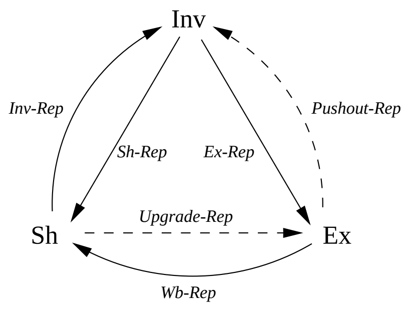


Figure 6: State Transitions Caused by Imperative Messages (Inv represents the state that the address is not cached in the memory; Pushout-rep and Upgrade-rep are potential optimizations for the HCN model)

incoming queue. We will proceed with the design of the HCN rules under this assumption, and later revisit the buffer management issue.

In HCN, all messages are imperative messages that are needed to make the protocol functionally correct. Imperative messages are also called reply messages because usually, though not necessarily, they are used to respond to directive messages as will be discussed in the next two sections.

5.2 HCN Rewriting Rules

The derivation of HCN rules from the HC rules is remarkably simple: HC memory access rules remain unaffected; each caching and de-caching rule in HC becomes a pair of rules for sending and receiving imperative messages; and two new rules for passing messages between parents and children are introduced. As will be seen, a caching or de-caching operation in HCN can be performed in three steps: (i) the source site places a message in the outgoing message queue; (ii) the network transfers the message from the source to the corresponding destination; (iii) the destination site extracts the message from its incoming queue and executes appropriate operations.

Memory Access Rules: Both Load and Store operations are performed on L_1 caches, and the incoming and outgoing queues are not affected.

HCN-Load Rule

$$\begin{array}{ll} & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell(a,v,(cs,R(}\epsilon\mathsf{)))} \mid \mathsf{m,\ in,\ out}\rangle,\ \ \mathsf{Proc(ia,\ rf,\ prog)}) \\ & \qquad\qquad \textit{if}\ \ \mathsf{prog[ia]} = \mathsf{r := Load(r_1)}\ \ \textit{and}\ \ \mathsf{a = rf[r_1]} \\ \longrightarrow & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell(a,v,(cs,R(}\epsilon\mathsf{)))} \mid \mathsf{m,\ in,\ out}\rangle,\ \ \mathsf{Proc(ia{+}1,\ rf[r := v],\ prog)}) \end{array}$$

HCN-Store Rule

$$\begin{array}{ll} & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell(a,u,(Ex,R(}\epsilon\mathsf{)))} \mid \mathsf{m,\ in,\ out}\rangle,\ \ \mathsf{Proc(ia,\ rf,\ prog)}) \\ & \qquad\qquad \textit{if}\ \ \mathsf{prog[ia]} = \mathsf{Store(r_1, r_2)}\ \ \textit{and}\ \ \mathsf{a = rf[r_1]} \\ \longrightarrow & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell(a,v,(Ex,R(}\epsilon\mathsf{)))} \mid \mathsf{m,\ in,\ out}\rangle,\ \ \mathsf{Proc(ia{+}1,\ rf,\ prog)})\quad \textit{where}\ \ \mathsf{v = rf[r_2]} \end{array}$$

Caching Rules: If the state of a cell in memory id is (Sh,R(dir)) or (Ex,R(dir)), and the directory shows that the data is not cached in some child (say $\mathsf{id_k}$), then memory id can send a Sh-rep message with a read-only copy to the child, and record $\mathsf{id_k}$ in the directory. When the Sh-rep message arrives at the destination, memory $\mathsf{id_k}$ caches the data and sets its state as (Sh,R(ϵ)) state.

Send-Sh-Rep Rule

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

Receive-Sh-Rep Rule

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

Similarly, if the state of a cell in memory id is (Ex,R(ϵ)) then it can send an Ex-rep message with a read-write copy to some child (say $\mathsf{id_k}$), and then change the Hstate to W($\mathsf{id_k}$). When the Ex-rep message arrives at the destination, memory $\mathsf{id_k}$ caches the data and sets its state as (Ex,R(ϵ)).

Send-Ex-Rep Rule

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

Receive-Ex-Rep Rule

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

De-Caching Rules: If the state of a cell in memory $\mathsf{id_k}$ is (Ex,R(dir)) then it can send a Wb-rep message with the most up-to-date data to its parent (say id), and change the Cstate of the cell to Sh. When the Wb-rep message is received, memory id updates the memory with the most up-to-date data, and changes the cell's Cstate from W($\mathsf{id_k}$) to R($\mathsf{id_k}$) because the child has given up the write permission.

Send-Wb-Rep Rule

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

Receive-Wb-Rep Rule

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

If state of a cell in memory $\mathsf{id_k}$ is $(\mathsf{Sh},\mathsf{R}(\epsilon))$ then it can invalidate the cell and send an Inv-rep message to notify its parent (say id). When the Inv-rep message is received, memory id removes $\mathsf{id_k}$ from the directory because the cached data in the child has been invalidated.

Send-Inv-Rep Rule

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

Receive-Inv-Rep Rule

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

Message-Passing Rules: Messages passing can happen only between the memories that have parent-child relationship. The *Message-Passing-To-Child* rule delivers a message from the outgoing queue of parent id to the incoming queue of child $\mathsf{id_k}$; while the *Message-Passing-To-Parent* rule delivers a message from the child's outgoing queue to the parent's incoming queue.

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

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

Discussion: Notice because of the associativity and commutativity of $\odot$ and $\otimes$, both message send and receive are non-FIFO and non-blocking. 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 Inv-rep message may arrive at the parent memory before the Wb-rep message. If messages in the input queue were required to be processed in the FIFO order, a deadlock could happen because no rule can apply to the Inv-rep message before the Wb-rep message is processed first. However, since messages in the incoming queue can be examined in any order, this deadlock will not occur.

5.3 Correctness of HCN w.r.t. HC

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

1. **Soundness:** Define the queue-flush function QF ($\mathsf{HCN} \mapsto \mathsf{HC}$), 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 the queue-lift function QL ($\mathsf{HC} \mapsto \mathsf{HCN}$), 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 $QF(QL(s)) = s$.

Analogous to the cache-flush function in the proof of HC discussed earlier, we define the queue-flush function QF to map HCN terms to HC terms. The idea behind the the QF function is that, in HCN, when only message-passing and message-receive rules are applied, all the incoming and outgoing message queues eventually become empty. The detailed proof of this fact is tedious because inclusion invariants are more complicated: for example, if the directory shows that a cell has been cached by a child, it only means that either the child has the cell or a message regarding that cell is on the way to the child or is on the way to the parent from the child.

To show soundness, assume that $s_1 \longrightarrow s_2$ by applying rule α in HCN. It can be shown by case analysis on α that either $QF(s_1) = QF(s_2)$, if α is a message-passing or message-receive rule; or $QF(s_1) \twoheadrightarrow QF(s_2)$ by applying a sequence of rules in HC, if α is a 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. The proofs of completeness and connection are straightforward. (For more details, see [24]).

6 The Directive Design Phase

The protocols presented so far, that is HC and HCN, rely on an oracle to select the rules to be applied to invoke desirable coherence actions. Consider the case when a processor wants to execute a Load operation but its L1 cache does not contain the accessed address. With HCN rules it is possible to bring the desired cell into the L1 cache from the parent, however, it is not clear how the parent memory would know which processor is looking for which particular address. To remedy this problem we introduce the notion of *directive* messages which are requests to invoke imperative actions. Before we explain directive messages, we first explicate some general implementation assumptions.

6.1 General Implementation Issues

For a TRS, one needs to specify a *rewriting strategy* for selecting a rule or a group of rules from the set of rules that are applicable to a term. When TRS's are used in a programming language context, terms represent expressions, and the goal of any rewriting strategy is to produce the value of an expression. Unfortunately, there is no such goal when a TRS represents a protocol or an architecture. But, there is a notion of "making progress" and one should ensure that the strategy is such that every distributed unit (processor, memory unit, etc.) does make progress.

A common strategy to guarantee progress for everyone is not to delay the execution of an applicable rule indefinitely. Such a strategy is known as a *fair* strategy. As we will explain, a fair strategy may not necessarily be efficient or easy to implement. We begin by making the following assumptions about any implementation:

- **Concurrent execution:** processors and memory units are running in a distributed fashion (i.e., no central control);
- **Reliable message passing:** a message is guaranteed to be delivered to the destination in finite time once it is enqueued in the outgoing queue;
- **Fair message processing:** a message in the incoming queue must be processed eventually if it can be processed.

The first two conditions are easily satisfied in most distributed systems. The third condition requires appropriate buffer management for incoming messages to guarantee that a message that cannot be processed temporarily will not block other messages from being processed. We will defer the discussion of the buffer management until Section 8. Here we assume that the messages in the incoming queue can commute with each other to achieve fair processing.

We now show that a fair strategy may not be desirable for the application of some rule. Consider the case of a memory holding a cell in the exclusive state. If the cell has not been passed to any children, then according to HC or HCN rules, the memory can give the cell to any one of its children, regardless of whether they need it or not. This may be wasteful but is not dangerous because the child can always return the cell. It will be more desirable if the caching rules are applied only when *requested* by a child. If we don't rely on fairness then there must be directives to apply certain imperative rules.

6.2 Directive Messages and Transient Records

The intended effect of a directive message is to cause the receiver to take some imperative action, such as issuing certain imperative message. Thus we can introduce request commands for some imperative rules (i.e., caching and de-caching rules) of HCN. The caching directives flow from children to parents while the de-caching directives flow from parents to children. Several issues arise immediately regarding directives:

1. We need to decide under what conditions a directive message should be issued. Furthermore, we may want a mechanism to avoid sending the same request repeatedly.
2. When a directive message is being processed, it is possible that the message needs to be *suspended* to issue more directives and to service their responses. Therefore, we must determine the information that should be maintained to resume a suspended message.
3. When a message cannot be processed temporarily, we need to decide how to handle this message (buffering, retrying, etc.) so that it cannot block other messages that can be processed.

The usual way to avoid repeated sending of the same request is to introduce new temporary states to remember that a request has been issued already. A *transient record* can be used to record all the necessary information regarding a suspended message (or instruction) and the outstanding requests. When a transient record for an address exists, we say the address is in a *transient state*. The specific information maintained in the transient record is dependent on the details of the protocol design.

The Imperative-Directive methodology forbids directive messages to directly affect the states manipulated by the imperative rules. The only way that a directive rule can update those states is by invoking some imperative rules. When a directive message is combined with an imperative rule to form a new rule, the directive message acts like an extra predicate in the imperative rule. This restriction guarantees soundness of the new rules. However, extra predicates can cause deadlocks or livelocks, and we will discuss these issues after presenting a complete protocol in the next section.

7 HCN-base: A Protocol Based on HCN

In this section, we present HCN-base, a complete protocol based on the HCN model, which uses directive messages and is free from deadlocks and livelocks. The design of the HCN-base protocol has been guided by the following properties:

1. *Every coherence action is originally driven by some memory access operation in a processor:* Directive messages can be issued only in cases of cache misses, or while processing other directive messages. A memory cannot send data to a child without receiving a caching request from the child. Similarly, a memory cannot write the most up-to-date data back to its parent or invalidate a shared copy without receiving a de-caching request from the parent.

   Notice this property implies that HCN-base cannot deal with cache line replacement (e.g., due to capacity or associative conflict). Elsewhere [24] we have extended HCN-base to an adaptive protocol that provides much more flexibility without compromising the liveness property. The adaptive protocol not only handles cache line replacement and allows adaptive features including prefetching, but is also tuned with various optimizations such as *Pushout* and *Upgrade*.

2. *A memory only process one request message for the same address at any time:* Except for an invalidation request, a request message for an address is not processed if the address is in a transient state. This restriction simplifies the protocol design, and our experience shows that its impact on performance is negligible.

3. *An invalidation request must be processed even if the address in the transient state to avoid deadlocks.*

4. *No request is ever sent more than once, and no message is ever discarded without being processed.*

We make no assumption about the cache hierarchy depth or the message passing order. HCN-base is free from deadlock in the sense that a request message is guaranteed to be serviced within finite time, and no processor can be stalled forever due to a cache miss.

The grammar of HCN-base is given in Figure 7. Four directive messages, Sh-req, Ex-req, Wb-req and Inv-req, are introduced. ("req" stands for request in these messages). Compared

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 | 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*
INQ	$\equiv$	ϵ [] MSG $\odot$ INQ	*Incoming Queue*
OUTQ	$\equiv$	ϵ [] MSG $\otimes$ OUTQ	*Outgoing Queue*
MSG	$\equiv$	$\mathsf{Msg}(\mathsf{id}_{\mathrm{src}}, \mathsf{id}_{\mathrm{dest}}, \mathrm{CMD}, \mathsf{a}, \mathsf{v})$	*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) | TRECS	*Transient Records*
INITIATOR	$\equiv$	(id,REQUEST) [] (ia,Load) [] (ia,Store)	*Initiator*

Figure 7: The HCN-base Protocol (Initially, all non-outermost memories, all incoming and outgoing message queues, and all transient records are empty; the outermost memory contains a cell in the (Ex,R(ϵ)) state for each address)

with the HCN model, an additional component, the transient records (Trecs), is maintained at each memory unit. A transient record contains the identity of the *initiator* that caused the address to enter the transient state: if it is a Load or Store instruction, the program counter is recorded; if it is a request message, the message source and the request command are recorded.

In order to process some types of requests, multiple actions at a memory may need to be taken. This, in turn, may require remembering several things about the suspended request. It turns out that for the protocol presented here, generally one transient record per address is sufficient. However, if an invalidation request for a transient address needs to be processed then two records for the same address may exist concurrently. We will use the following notation for compactness:

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

Now we present the rewrite rules for the HCN-base protocol. These rules have been derived systematically from the HCN rules and the assumptions discussed above. The reader will find that (1) the processor cache hit-rules and message passing rules remain the same as in

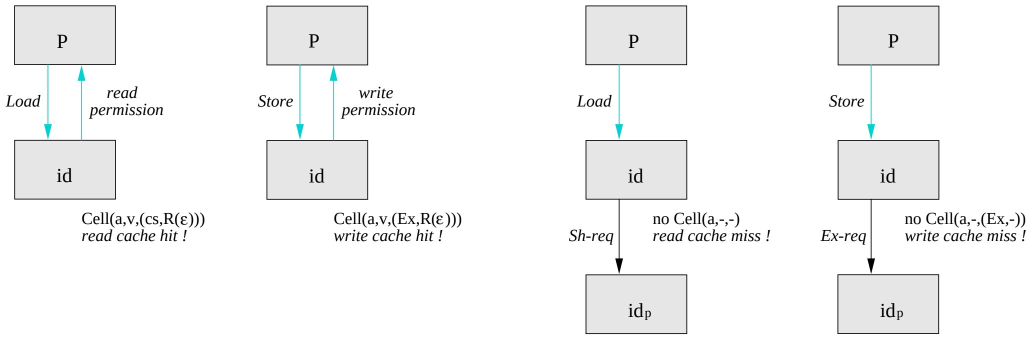


Figure 8: Cache Hit and Cache-Miss Processing

HCN; (2) a new rule for each HCN send rule is derived by adding the corresponding request message in the incoming queue (The request message acts as a predicate); (3) another similar rule is derived for each HCN send rule because the request may be in a transient record; (4) each HCN receive rule is extended to resume a suspended request; (5) new rules are added for propagating requests.

7.1 Memory Access Rules

Cache-Hit Rules: In case of a cache hit, a Load can read the data from the L1 cache, and a Store can write a new value to the L1 cache provided the cell is in the Ex state.

Read-Cache-Hit Rule

$$\begin{array}{ll} & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell(a,v,(cs,R(}\epsilon)))\mid \mathsf{m},\ \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle,\ \ \mathsf{Proc(ia,\ rf,\ prog)}) \\ & \qquad\qquad \textit{if}\ \ \mathsf{prog[ia]} = \mathsf{r}:=\mathsf{Load(r_1)}\ \ \textit{and}\ \ \mathsf{a} = \mathsf{rf[r_1]} \\ \longrightarrow & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell(a,v,(cs,R(}\epsilon)))\mid \mathsf{m},\ \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle,\ \ \mathsf{Proc(ia{+}1,\ rf[r:=v],\ prog)}) \end{array}$$

Write-Cache-Hit Rule

$$\begin{array}{ll} & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell(a,u,(Ex,R(}\epsilon)))\mid \mathsf{m},\ \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle,\ \ \mathsf{Proc(ia,\ rf,\ prog)}) \\ & \qquad\qquad \textit{if}\ \ \mathsf{prog[ia]} = \mathsf{Store(r_1,r_2)}\ \ \textit{and}\ \ \mathsf{a} = \mathsf{rf[r_1]} \\ \longrightarrow & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{Cell(a,v,(Ex,R(}\epsilon)))\mid \mathsf{m},\ \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle,\ \ \mathsf{Proc(ia{+}1,\ rf,\ prog)})\quad \textit{where}\ \ \mathsf{v} = \mathsf{rf[r_2]} \end{array}$$

Cache-Miss Rules: In case of a cache miss, unless the address is already in a transient state, a Sh-req or an Ex-req message is sent to the parent and a transient record for that address is created.

Read-Cache-Miss Rule

$$\begin{array}{ll} & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle,\ \ \mathsf{Proc(ia,\ rf,\ prog)}) \\ & \qquad\qquad \textit{if}\ \ \mathsf{prog[ia]} = \mathsf{r}:=\mathsf{Load(r_1)}\ \ \textit{and}\ \ \mathsf{rf[r_1]} \notin \mathsf{m}\ \ \textit{and}\ \ \mathsf{rf[r_1]} \notin \mathsf{trecs} \\ \longrightarrow & \mathsf{Sys}(\langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg(id,id_p,Sh\text{-}req,a,}\perp),\ \mathsf{Trec(a,(ia,Load))}\mid \mathsf{trecs}\rangle,\ \ \mathsf{Proc(ia,\ rf,\ prog)}) \\ & \qquad\qquad \textit{where}\ \ \mathsf{id_p} = \mathsf{parent(id)}\ \ \textit{and}\ \ \mathsf{a} = \mathsf{rf[r_1]} \end{array}$$

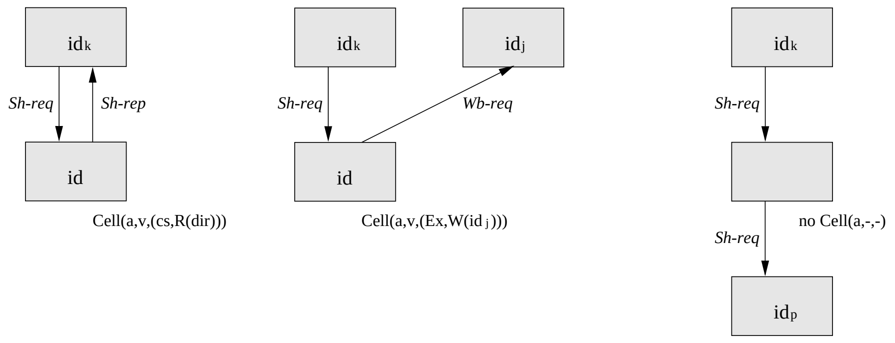


Figure 9: Sh-req Processing (memory id receives a Sh-req message)

Write-Cache-Miss Rule

$$\begin{array}{ll} & \mathsf{Sys}(\langle \mathsf{id, m, in, out, trecs}\rangle,\ \ \mathsf{Proc(ia, rf, prog)}) \\ & \qquad \textit{if}\ \ \mathsf{prog[ia]} = \mathsf{Store}(\mathsf{r}_1, \mathsf{r}_2)\ \ \textit{and}\ \ \mathsf{Cell(rf[r_1],\text{-},(Ex,\text{-}))} \notin \mathsf{m}\ \ \textit{and}\ \ \mathsf{rf[r_1]} \notin \mathsf{trecs} \\ \longrightarrow & \mathsf{Sys}(\langle \mathsf{id, m, in, out} \otimes \mathsf{Msg(id, id_p, Ex\text{-}req, a, \bot)},\ \mathsf{Trec(a, (ia,Store))} \mid \mathsf{trecs}\rangle,\ \ \mathsf{Proc(ia, rf, prog)}) \\ & \qquad \textit{where}\ \ \mathsf{id_p} = \mathsf{parent(id)}\ \ \ \textit{and}\ \ \ \mathsf{a} = \mathsf{rf[r_1]} \end{array}$$

7.2 Child-to-Parent Request Rules

Sh-Request Rules: When memory id receives a Sh-req message from child memory $\mathsf{id_k}$, it processes the request unless the accessed address is in a transient state, in which case it simply lets the message sit in the input queue but without blocking other incoming messages (commutativity of $\odot$ ensures this non blockage). The following three cases arise:

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

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

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

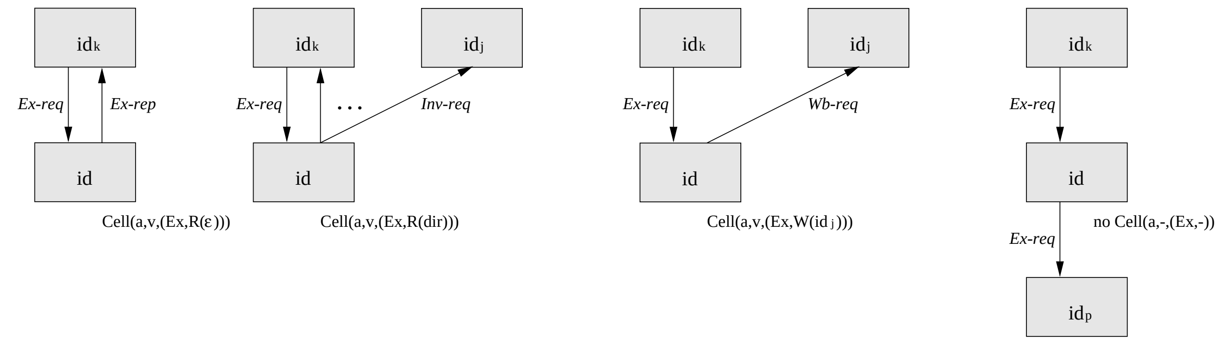


Figure 10: Ex-req Processing (memory id receives a Ex-req message)

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

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

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

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

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

$\langle \mathsf{id},\ \mathsf{m},\ \mathsf{Msg(id_k, id, Sh\text{-}req, a, \bot)} \odot \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle \quad \textit{if}\ \ \mathsf{a} \notin \mathsf{m}\ \ \textit{and}\ \ \mathsf{a} \notin \mathsf{trecs}$

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

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

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

- *Hit:* If id has the data in the $(\mathsf{Ex,R}(\epsilon))$ state, it sends an Ex-rep message to $\mathsf{id_k}$ and changes the cell's state to $(\mathsf{Ex,W(id_k)})$.
- *Hit but outstanding reads:* If id has the data in the (Ex,R(dir)) state, it multicasts Inv-req messages to memories specified by directory dir and creates a transient record for the address.
- *Hit but stale data:* If id has the data in the $(\mathsf{Ex,W(id_j)})$ state, it sends a Wb-req message to child $\mathsf{id_j}$ to request the most up-to-date data, and creates a transient record for the address. Notice that we could also issue the Inv-req command at the this time. We keep the protocol simple by issuing an Inv-req only after the response to Wb-req has been received (see Wb-Reply Rules in Section 7.5).
- *Miss or lack of exclusivity:* If id has not cached the data, or the data is not in the Ex state, it sends an Ex-req message to its parent memory, and creates a transient record for the address.

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

$\langle$id, Cell(a,v,(Ex,R(ϵ))) | m, Msg(id$_k$, id, Ex-req, a, $\perp$) $\odot$ in, out, trecs$\rangle$ *if* a $\notin$ trecs

$\longrightarrow$ $\langle$id, Cell(a,v,(Ex,W(id$_k$))) | m, in, out $\otimes$ Msg(id, id$_k$, Ex-rep, a, v), trecs$\rangle$

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

$\langle$id, Cell(a,v,(Ex,R(dir))) | m, Msg(id$_k$, id, Ex-req, a, $\perp$) $\odot$ in, out, trecs$\rangle$

if dir $\neq \epsilon$ *and* a $\notin$ trecs

$\longrightarrow$ $\langle$id, Cell(a,v,(Ex,R(dir))) | m, in, out $\otimes$ multicast(id, dir, Inv-req, a, $\perp$), Trec(a, (id$_k$,Ex-req)) | trecs$\rangle$

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

$\langle$id, Cell(a,v,(Ex,W(id$_j$))) | m, Msg(id$_k$, id, Ex-req, a, $\perp$) $\odot$ in, out, trecs$\rangle$

if id$_k$ $\neq$ id$_j$ *and* a $\notin$ trecs

$\longrightarrow$ $\langle$id, Cell(a,v,(Ex,W(id$_j$))) | m, in, out $\otimes$ Msg(id, id$_j$, Wb-req, a, $\perp$), Trec(a, (id$_k$,Ex-req)) | trecs$\rangle$

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

$\langle$id, m, Msg(id$_k$, id, Ex-req, a, $\perp$) $\odot$ in, out, trecs$\rangle$

if Cell(a,-,(Ex,-)) $\notin$ m *and* a $\notin$ trecs

$\longrightarrow$ $\langle$id, m, in, out $\otimes$ Msg(id, id$_p$, Ex-req, a, $\perp$), Trec(a, (id$_k$,Ex-req)) | trecs$\rangle$

where id$_p$ = parent(id)

Discussion: It is worth noting that the Sh-req message cannot be processed if the accessed address is in a transient state. There are two reasons for this. First, it effectively prevents multiple requests to be issued for the same coherence action. For example, if memory id receives more than one Sh-req message at the same time (from different child memories), and the accessed data is not cached, then only one Sh-req message will be issued to parent memory id$_p$. Second, it ensures that Sh-req and Ex-req messages are processed in some fair order. Without this fairness a suspended Ex-req may never be processed. As we shall see later, the same issue arises for Inv-req messages.

7.3 Parent-to-Child Request Rules

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

- *Hit:* If id has the data in the (Ex,R(dir)) state, it sends a Wb-rep message to the parent with the most up-to-date data, and changes the cell's Cstate to Sh.
- *Hit but stale data:* If id has cached the data in the (Ex,W(id$_k$)) state, it sends a Wb-req message to child memory id$_k$ to request for the most up-to-date data and creates a transient record for the address.

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

$\langle$id, Cell(a,v,(Ex,R(dir))) | m, Msg(id$_p$, id, Wb-req, a, $\perp$) $\odot$ in, out, trecs$\rangle$ *if* a $\notin$ trecs

$\longrightarrow$ $\langle$id, Cell(a,v,(Sh,R(dir))) | m, in, out $\otimes$ Msg(id, id$_p$, Wb-rep, a, v), trecs$\rangle$

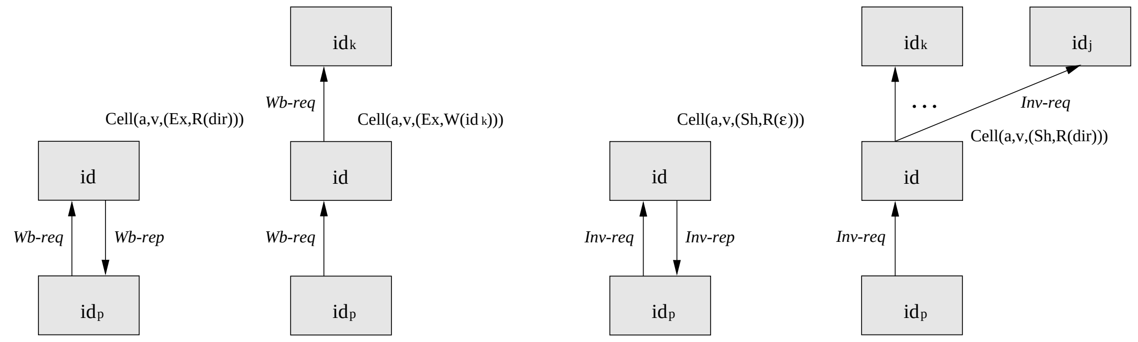


Figure 11: Wb-req and Inv-req Processing (memory id receives a Wb-req / Inv-req message)

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

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

Discussion: Notice that it is safe to block a Wb-req message at any memory if it is in a transient state, because that memory can get out of the transient state without sending requests to the parent. That is, a cache does not need to communicates with its parent if it already has a cell in the Ex state.

Inv-Request Rules: When memory id receives an Inv-req message from parent memory $\mathsf{id_p}$, it processes the message as follows:

- *Hit:* If id has the data in the $(\mathsf{Sh,R(\epsilon)})$ state, it invalidates (purges) the cell and sends an Inv-rep message to the parent to acknowledge the invalidation.
- *Hit but outstanding reads:* If id has the data in the (Sh,R(dir)) state and $\mathsf{dir} \neq \epsilon$, it multicasts Inv-req messages to memories specified by directory dir, and creates a transient record for the address.

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

$$\langle \mathsf{id},\ \mathsf{Cell(a,v,(Sh,R(\epsilon)))} \mid \mathsf{m},\ \mathsf{Msg(id_p, id, Inv\text{-}req, a, \bot)} \odot \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle$$
$$\longrightarrow \quad \langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{Msg(id, id_p, Inv\text{-}rep, a, \bot)},\ \mathsf{trecs}\rangle$$

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

$$\langle \mathsf{id},\ \mathsf{Cell(a,v,(Sh,R(dir)))} \mid \mathsf{m},\ \mathsf{Msg(id_p, id, Inv\text{-}req, a, \bot)} \odot \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle \quad \textit{if} \quad \mathsf{dir} \neq \epsilon$$
$$\longrightarrow \quad \langle \mathsf{id},\ \mathsf{Cell(a,v,(Sh,R(dir)))} \mid \mathsf{m},\ \mathsf{in},\ \mathsf{out} \otimes \mathsf{multicast(id, dir, Inv\text{-}req, a, \bot)},\ \mathsf{Trec(a, (id_p, Inv\text{-}req))} \mid \mathsf{trecs}\rangle$$

Discussion: Notice that an Inv-req message can be processed while the accessed address is in a transient state. This fact is critical to avoid deadlocks. The only reason why Inv-req may not be satisfied or propagated immediately is if the cell is in the Ex state. But then it must

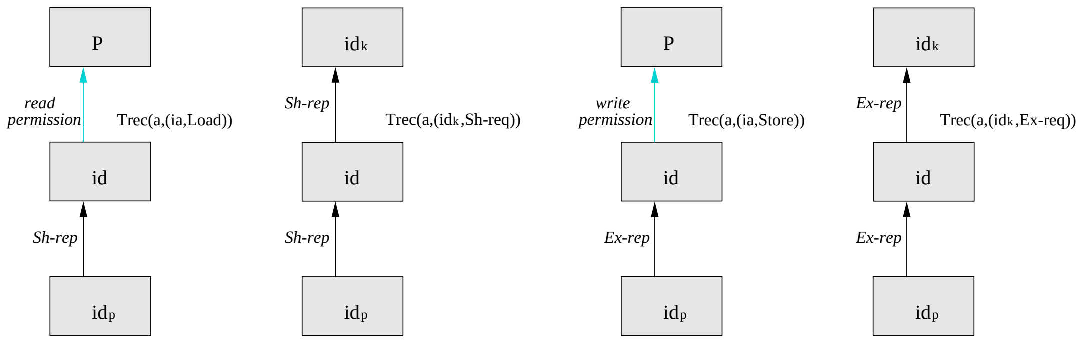


Figure 12: Sh-rep and Ex-rep Processing (memory id receives a Sh-rep / Ex-rep message)

be the case that it has overtaken a Wb-req in the network. Once the Wb-req is executed, Inv-req can be executed. An Inv-req request at an L1 can be processed to completion without further propagation. It can be shown by induction that any memory (not necessarily the innermost one) can process an Inv-req request to completion in a finite number of steps.

7.4 Parent-to-Child Reply Rules

A reply message (Sh-rep or Ex-rep) can be processed immediately. It updates the cache according to the message, processes the suspended request, and deletes the transient record for the address. In the L1 cache the suspended request is from the processor and processing it means resuming the instruction. In non-L1 caches, the suspended request is from another cache, and is processed by sending a reply message.

Sh-Reply Rules: When an L1 cache receives a Sh-rep message, it caches the data in the (Sh,R(ϵ)) state. In all other cases the data in cached in the (Sh,R(id_k)) state, where id_k is the identifier in the suspended request.

Receive-Sh-Rep-And-Execute-Load Rule

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

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

$$\begin{aligned}
& \langle \text{id, m, Msg}(\text{id}_p, \text{id}, \text{Sh-rep}, \text{a}, \text{v}) \odot \text{in, out, Trec(a, (id}_k\text{,Sh-req))} \mid \text{trecs}\rangle \\
\longrightarrow \;\; & \langle \text{id, Cell(a,v,(Sh,R(id}_k))) \mid \text{m, in, out} \otimes \text{Msg(id, id}_k\text{, Sh-rep, a, v), trecs}\rangle
\end{aligned}$$

Ex-Reply Rules: When an L1 cache receives an Ex-rep message, it caches the data in the (Ex,R(ϵ)) state. In all other cases the data in cached in the (Sh,W(id_k)) state, where id_k is the identifier in the suspended request.

Receive-Ex-Rep-And-Execute-Store Rule
$$\begin{array}{ll} & \mathsf{Sys}(\langle \mathsf{id,\ m,\ Msg(id_p, id, Ex\text{-}rep, a, u)} \odot \mathsf{in,\ out,\ Trec(a, (ia,Store))}\ |\ \mathsf{trecs}\rangle,\ \ \mathsf{Proc(ia,\ rf,\ prog)}) \\ & \qquad \textit{if}\ \ \mathsf{prog[ia]} = \mathsf{Store(r_1, r_2)}\ \ \textit{and}\ \ \mathsf{a} = \mathsf{rf[r_1]} \\ \longrightarrow & \mathsf{Sys}(\langle \mathsf{id,\ Cell(a,v,(Ex,R(\epsilon)))}\ |\ \mathsf{m,\ in,\ out,\ trecs}\rangle,\ \ \mathsf{Proc(ia{+}1,\ rf,\ prog)}) \quad \textit{where}\ \ \mathsf{v} = \mathsf{rf[r_2]} \end{array}$$

Receive-Ex-Rep-And-Send-Ex-Rep Rule
$$\begin{array}{ll} & \langle \mathsf{id,\ m,\ Msg(id_p, id, Ex\text{-}rep, a, v)} \odot \mathsf{in,\ out,\ Trec(a, (id_k,Ex\text{-}req))}\ |\ \mathsf{trecs}\rangle \\ \longrightarrow & \langle \mathsf{id,\ Cell(a,v,(Ex,W(id_k)))}\ |\ \mathsf{m,\ in,\ out} \otimes \mathsf{Msg(id, id_k, Ex\text{-}rep, a, v),\ trecs}\rangle \end{array}$$

7.5 Child-to-Parent Reply Rules

Generally a reply message (Wb-rep or Inv-rep) from a child can also be processed immediately in a similar manner: It updates the cache according to the message, processes the suspended request, and deletes the transient record for the address. However, there are several different cases for generating the responses for the suspended requests, and in one case the transient record is not deleted immediately.

Wb-Reply Rules: When memory id receives a Wb-rep message from child $\mathsf{id_k}$, it processes the message depending upon the type of suspended request as following:

- *Parent's* Wb-req *request:* The cell's state is set to $\mathsf{(Sh,R(id_k))}$, and a Wb-rep message is sent to the parent $\mathsf{id_p}$.
- *Child's* Sh-req *request:* If the requesting child is $\mathsf{id_j}$, the cell's state is set to $\mathsf{(Ex,R(id_k|id_j))}$, and a Sh-rep message is sent to $\mathsf{id_j}$.
- *Child's* Ex-req *request:* The request still cannot be satisfied because we need to delete the cell from the child $\mathsf{id_k}$ first. The cell's state is set to $\mathsf{(Ex,R(id_k|id_j))}$, and a Inv-req is sent $\mathsf{id_k}$. The transient record is NOT deleted.

Receive-Wb-Rep-And-Send-Wb-Rep Rule
$$\begin{array}{ll} & \langle \mathsf{id,\ Cell(a,u,(Ex,W(id_k)))}\ |\ \mathsf{m,\ Msg(id_k, id, Wb\text{-}rep, a, v)} \odot \mathsf{in,\ out,\ Trec(a, (id_p,Wb\text{-}req))}\ |\ \mathsf{trecs}\rangle \\ \longrightarrow & \langle \mathsf{id,\ Cell(a,v,(Sh,R(id_k)))}\ |\ \mathsf{m,\ in,\ out} \otimes \mathsf{Msg(id, id_p, Wb\text{-}rep, a, v),\ trecs}\rangle \end{array}$$

Receive-Wb-Rep-And-Send-Sh-Rep Rule
$$\begin{array}{ll} & \langle \mathsf{id,\ Cell(a,u,(Ex,W(id_k)))}\ |\ \mathsf{m,\ Msg(id_k, id, Wb\text{-}rep, a, v)} \odot \mathsf{in,\ out,\ Trec(a, (id_j,Sh\text{-}req))}\ |\ \mathsf{trecs}\rangle \\ \longrightarrow & \langle \mathsf{id,\ Cell(a,v,(Ex,R(id_k|id_j)))}\ |\ \mathsf{m,\ in,\ out} \otimes \mathsf{Msg(id, id_j, Sh\text{-}rep, a, v),\ trecs}\rangle \end{array}$$

Receive-Wb-Rep-And-Send-Inv-Req Rule
$$\begin{array}{ll} & \langle \mathsf{id,\ Cell(a,u,(Ex,W(id_k)))}\ |\ \mathsf{m,\ Msg(id_k, id, Wb\text{-}rep, a, v)} \odot \mathsf{in,\ out,\ Trec(a, (id_j,Ex\text{-}req))}\ |\ \mathsf{trecs}\rangle \\ \longrightarrow & \langle \mathsf{id,\ Cell(a,v,(Ex,R(id_k)))}\ |\ \mathsf{m,\ in,\ out} \otimes \mathsf{Msg(id, id_k, Inv\text{-}req, a, v),\ Trec(a, (id_j,Ex\text{-}req))}\ |\ \mathsf{trecs}\rangle \end{array}$$

Inv-Reply Rules: When memory id receives a Inv-rep message from a child memory $\mathsf{id_k}$, it proceeds as follows:

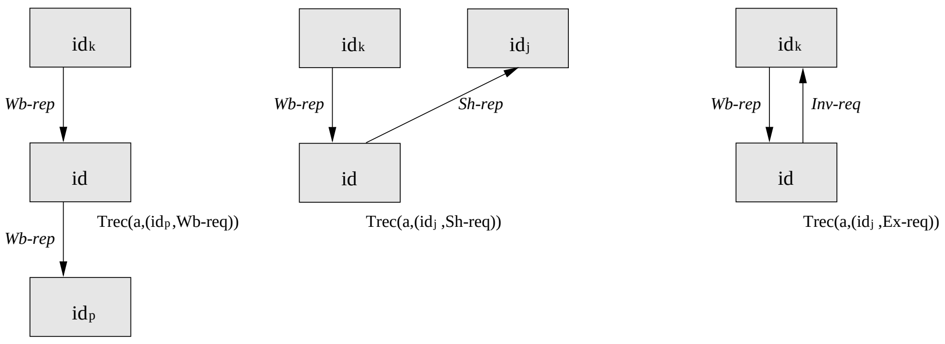


Figure 13: Wb-rep Processing (memory id receives a Wb-rep message)

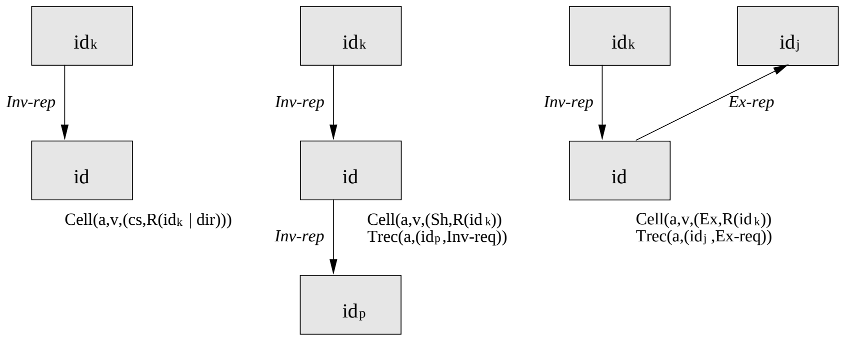


Figure 14: Inv-rep Processing (memory id receives a Inv-rep message)

- *Incomplete invalidation:* If the directory shows that some other children still have the data, the memory simply removes id_k from the directory. The transient record in NOT deleted.
- *Complete invalidation, parent's* Inv-req *request:* The cell is purged and a Inv-rep message is sent to the parent.
- *Complete invalidation, child's* Ex-req *request:* If the suspended message is from child id_j, the cell's state is set to $(\mathsf{Ex},\mathsf{W}(\mathsf{id}_\mathrm{j}))$, and an Ex-rep message is sent to id_j.

Receive-Inv-Rep-Pending Rule

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{cs},\mathsf{R}(\mathsf{id}_\mathrm{k}|\mathsf{dir})))\ |\ \mathsf{m},\ \mathsf{Msg}(\mathsf{id}_\mathrm{k},\mathsf{id},\mathsf{Inv\text{-}rep},\mathsf{a},\perp)\odot \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle \quad \textit{if}\ \ \mathsf{dir}\neq\epsilon \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{cs},\mathsf{R}(\mathsf{dir})))\ |\ \mathsf{m},\ \mathsf{in},\ \mathsf{out},\ \mathsf{trecs}\rangle \end{array}$$

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

$$\begin{array}{ll} & \langle \mathsf{id},\ \mathsf{Cell}(\mathsf{a},\mathsf{v},(\mathsf{Sh},\mathsf{R}(\mathsf{id}_\mathrm{k})))\ |\ \mathsf{m},\ \mathsf{Msg}(\mathsf{id}_\mathrm{k},\mathsf{id},\mathsf{Inv\text{-}rep},\mathsf{a},\perp)\odot \mathsf{in},\ \mathsf{out},\ \mathsf{Trec}(\mathsf{a},(\mathsf{id}_\mathrm{p},\mathsf{Inv\text{-}req}))\ |\ \mathsf{trecs}\rangle \\ \longrightarrow & \langle \mathsf{id},\ \mathsf{m},\ \mathsf{in},\ \mathsf{out}\otimes \mathsf{Msg}(\mathsf{id},\mathsf{id}_\mathrm{p},\mathsf{Inv\text{-}rep},\mathsf{a},\perp),\ \mathsf{trecs}\rangle \end{array}$$

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

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

7.6 Message Passing Rules

Message-Passing-To-Child Rule

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

Message-Passing-To-Parent Rule

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

8 Deadlock Avoidance and Buffer Management

In HCN-base, when a protocol message is received, there are three possible cases regarding how it can be processed:

1. *Reply messages:* If the message is a reply message (Sh-rep, Ex-rep, Wb-rep or Inv-rep), then it can always be processed to completion immediately. If the memory is not an L1 cache, a reply message is issued to the original requesting site.
2. *Invalidation request:* If the message is Inv-req, then it can always be processed to completion in a number of steps by invalidating the descendents. (See the discussion in Section 7.3).
3. *Write-back request:* If the message is Wb-req, then it may be blocked if the accessed address is in a transient state. However, as discussed in Section 7.3, the address will come out of the transient state without communicating with the parent memory.
4. *Shared and Exclusive requests:* If the message is Sh-req or Ex-req, then it may be blocked if the accessed address is in a transient state. The memory may need to communicate with both the parent and children memories to process the request associated with the transient address. Thus, there is a real potential for deadlock. However, this deadlock is avoided because the invalidation requests are always allowed to proceed upwards.

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.

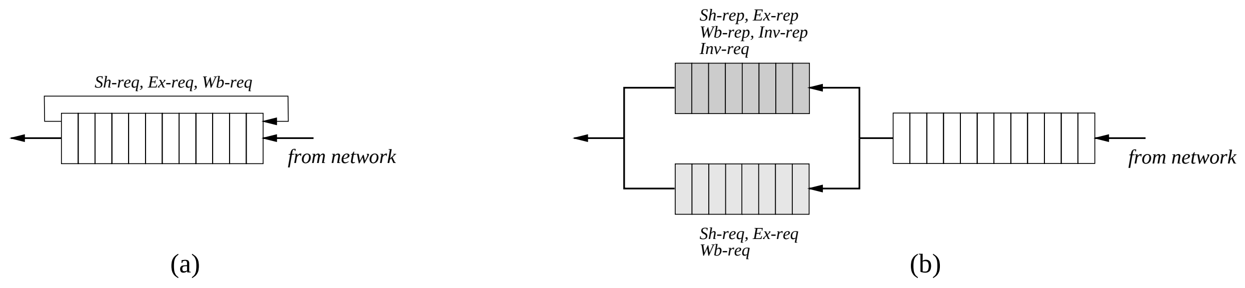


Figure 15: Simple Buffer Management for Incoming Messages

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 15(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 15 (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

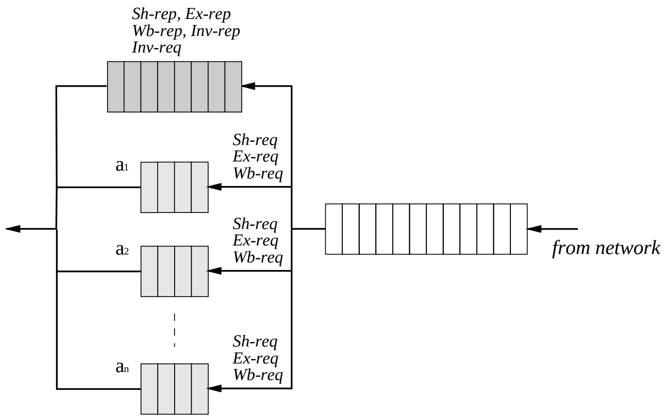


Figure 16: Buffer Management for Incoming Messages

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

9 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 [24].

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

[25] X. Shen and Arvind. Modeling and Verification of ISA Implementations. In *Proceedings of the 1998 Australasian Computer Architecture Conference, Perth, Australia (MIT CSG Memo 400(A))*, Feb. 1998.

[26] I. C. Society. *IEEE Standard for Scalable Coherent Interface*. 1993.

[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.