Deriving Efficient Cache Coherence Protocols through Refinement*

Ratan Nalumasu and Ganesh Gopalakrishnan
University of Utah, Salt Lake City, UT 84112, USA,
{ratan,ganesh}@cs.utah.edu,
WWW page: http://www.cs.utah.edu/~{ratan,ganesh}

Abstract. We address the problem of developing efficient cache coherence protocols implementing distributed shared memory (DSM) using message passing. A serious drawback of traditional approaches to this problem is that designers are required to state the desired coherence protocol at the level of asynchronous message interactions. We propose a method in which designers express the desired protocol at a high-level using *rendezvous* communication. These descriptions are much easier to understand and computationally more efficient to verify than asynchronous protocols due to their small state spaces. The rendezvous protocol can also be synthesized into efficient asynchronous protocols. We present our protocol refinement procedure, prove its soundness, and provide examples of its efficiency.

1 Introduction

With the growing complexity of concurrent systems, automated procedures for developing protocols are growing in importance. In this paper, we are interested in protocol *refinement* procedures, which we define to be those that accept high-level specifications of protocols, and apply provably correct transformations on them to yield detailed implementations of protocols that run efficiently and have modest buffer resource requirements. Such procedures enable correctness proofs of protocols to be carried out with respect to high-level specifications, which can considerably reduce the proof effort. Once the refinement rules are shown to be sound, the detailed protocol implementations need not be verified.

In this paper, we address the problem of producing correct and efficient cache coherence protocols used in *distributed shared memory (DSM)* systems. DSM systems have been widely researched as the next logical step in parallel processing [2,5,12,14]. A central problem in DSM systems is the design and implementation of distributed coherence protocols for shared cache lines using *message passing* [9]. The present-day approach to this problem consists of specifying the detailed interactions possible between the nodes in terms of low-level requests, acknowledges, negative acknowledges, and dealing with "unexpected" messages. Difficulty of designing these protocols is compounded by the fact that verifying such low-level descriptions invites state explosion (when done using model-checking [6,7]) or tedious (when done using theorem-proving [18]) even for simple configurations. Often these low-level descriptions are model-checked for specific resource allocations (e.g. buffer sizes); it is often not known what would happen when these allocations are changed. Protocol refinement can help alleviate this situation

* Supported in part by ARPA Order #B990 under SPAWAR Contract #N0039-95-C-0018 (Avalanche), DARPA under contract #DABT6396C0094 (UV).

considerably. Our contribution in this paper is a protocol refinement procedure which can be applied to derive a large class of DSM cache protocols.

Most of the problems in designing DSM cache coherence protocols are attributable to the apparent lack of atomicity in the implementation behaviors. Although some designers of these protocols begin with a simple atomic-transaction view of the desired interactions, such a description is seldom written down. Instead, what gets written down as the "highest level" specification is a detailed protocol implementation which was arrived at through *ad hoc* reasoning of the situations that can arise. In this paper, we choose CSP [10] as our specification language to allow the designers to capture their initial atomic-transaction view (*rendezvous protocol*). The rendezvous protocol is then subjected to syntax-directed translation rules to modify the rendezvous communication primitives of CSP into asynchronous communication primitives yielding an efficient detailed implementation (*asynchronous protocol*). We empirically show that the rendezvous protocols are several orders of magnitude more efficient to model-check than their corresponding detailed implementations. In addition, we also show that in the context of a state of the art DSM machine project called the Avalanche [2], our procedure can automatically produce protocol implementations that are comparable in *quality* to hand-designed asynchronous protocols, where quality is measured in terms of (1) the number of *request*, *acknowledge*, and *negative acknowledge* (nack) messages needed for carrying out the rendezvous specified in the given specification, and (2) the buffering requirements to guarantee a precisely defined and practically acceptable progress criterion.

The rest of the paper is organized as follows. In this section, we review related past work. Section 2 presents the structure of typical DSM protocols in distributed systems. Section 3 presents our syntax-directed translation rules, along with an important optimization called *request/reply*. Section 4 presents an informal argument that the refinement rules we present always produce correct result, and also points to a formal proof of correctness done using PVS [17]. Section 5 presents an example protocol developed using the refinement rules, and the efficiency of model-checking the rendezvous protocol compared to the efficiency of model-checking the asynchronous protocol. Finally, Section 6 presents a discussion of buffering requirements and concludes the paper.

Related Work

Chandra *et al* [3] use a model based on continuations to help reduce the complexity of specifying the coherency protocols. The specification can then be model checked and compiled into an efficient object code. In this approach, the protocol is still specified at a low-level; though rendezvous communication can be modeled, it is not very useful as the transient states introduced by their compiler cannot adequately handle unexpected messages. In contrast, in our approach, user writes the rendezvous protocol using only the rendezvous primitive, verifies the protocol at this level with great efficiency and compiles it into an efficient asynchronous protocol or object code.

Our work closely resembles that of Buckley and Silberschatz [1]. Buckley and Silberschatz consider the problem of implementing rendezvous using message passing when the processes use generalized input/output guard to be implemented in software. Their solution is too expensive for DSM protocol implementations. In contrast, we focus on a star configuration of processes with suitable syntactic restrictions on the high-

level specification language, so that an efficient asynchronous protocol can be automatically generated.

Chandy and Misra [4] showed that under a strict condition called *asynchrony*, shared variables can be implemented by message passing. Unfortunately, this condition is not met in many practical protocols. Gribomont [8] explored the protocols where the rendezvous communication can be simply replaced by asynchronous communication without affecting the processes in any other way. In contrast, we show how to *change* the processes when the rendezvous communication is replaced by asynchronous communication. Lamport and Schneider [13] have explored the theoretical foundations of comparing atomic transactions (*e.g.*, rendezvous communication) and split transactions (*e.g.*, asynchronous communication), based on left and right movers [15], but have not considered specific refinement rules.

2 Cache Coherency in Distributed Systems

In directory based cache coherent multiprocessor systems, the coherency of each line of shared memory is managed by a CPU node, called *home* node, or simply *home*[1]. All nodes that may access the shared line are called *remote* nodes. The home node is responsible for managing access to the shared line by all nodes without violating the coherency policy of the system. A simple protocol used in Avalanche, called migratory, is shown in Figures 2. The remote nodes and home node engage in the following activity. Whenever a remote node R wishes to access the information in a shared line, it first checks if the data is available (with required access permissions) in its local cache. If so, R uses the data from the cache. If not, it sends a request for permissions to the home node of the line. The home node may then contact some other remote nodes to revoke their permissions in order to grant the required permissions to R. Finally, the home node grants the permissions (along with any required data) to R. As can be seen from this description, a remote node interacts only with the home node, while the home node interacts with all the remote nodes. This suggests that we can restrict the communication topology of interest to a *star* configuration, with the home node as the hub, without losing any descriptive power. This decision helps synthesizing more efficient asynchronous protocols, as we shall see later.

2.1 Complexity of Protocol Design

As already pointed out, most of the problems in the design of DSM protocols can be traced to lack of atomicity. For example, consider the following situation. A shared line is being read by a number of remote nodes. A remote node, say R1, wishes to modify the data, hence sends a request to the home node for write permission. The home node then contacts all other remote nodes that are currently accessing the data to revoke their read permissions, and then grants the write permission to R1. Unfortunately, it is incorrect to *abstract* the entire sequence of actions consisting of contacting all other remote nodes to revoke permissions and granting permissions to R1 as an atomic action. This is because when the home node is in the process of revoking permissions, a different remote node,

[1] The home for different cache lines can be different. We will derive protocols focusing on one cache line, as is usually done.

say R2, may wish to obtain read permissions. In this case, the request from R2 must be either nacked or buffered for later processing. To handle such *unexpected* messages, the designers introduce intermediate states, also called *transient* states, leading to the complexity of the protocols. On the other hand, as we will show in the rest of the paper, if the designer is allowed to state the desired interactions using an atomic view, it is possible to *refine* such a description using a refinement procedure that introduces transient states appropriately to handle such unexpected messages.

2.2 Communication Model

We assume that the network that connects the nodes in the systems provides *reliable, point-to-point in-order delivery* of messages. This assumption is justified in many machines, e.g., DASH [14], and Avalanche [2]. We also assume that the network has infinite buffering, in the sense that the network can always accept new messages to be delivered. Without this assumption, the asynchronous protocol generated may deadlock. If the assumption is not satisfied, then the solution proposed by Hennessy and Patterson in [9] can be used as a post-processing step of the refined protocol. They divide the messages into two categories: *request* and *acknowledge*. A *request* message may cause the recipient to generate more messages in order to complete the transactions, while an *acknowledge* message does not. The authors argue that if the network always accepts *acknowledge* messages (as opposed to all messages in the case of a network with infinite buffer), such deadlocks are broken. As we shall see in Section 3, asynchronous protocol has two *acknowledge* messages: ack and nack.

We use rendezvous communication primitives of CSP [10] to specify the home node and the remote nodes to simplify the DSM protocol design. In particular, we use direct addressing scheme of CSP, where every input statement in process `Q` is of the form `P?msg(v)` or `P?msg`, where `P` is the identity of the process that sent the message, `msg` is an *enumerated constant* ("message type") and `v` is a variable (local variable of `Q`) which would be set to the contents of the message, and every output statement in `Q` is of the form `P!msg(e)` or `P!msg` where `e` is an expression involving constants and/or local variables of `Q`. When P and Q rendezvous by P executing `Q!m(e)` and Q executing `P?m(v)`, we say that P is the active process and Q is the passive process of the rendezvous.

The rendezvous protocol written using this notation is verified using either a theorem prover or a model checker for desired properties, and then refined using the rules presented in Section 3 to obtain an *efficient* asynchronous protocol that can be implemented directly, for example in microcode.

2.3 Process Structure

We divide the states of processes in the rendezvous protocol into two classes: *internal* and *communication*. When a process is in an internal state, it cannot participate in rendezvous with any other process. However, we assume that such a process will eventually enter a communication state where rendezvous actions are offered (this assumption can be syntactically checked). The refinement process introduces *transient* states where all unexpected messages are handled. We denote the ith remote node by r_i and the home node by h. For simplicity, we assume that all the remote nodes follow the same protocol and that the only form of communication between processes (in both asynchronous and

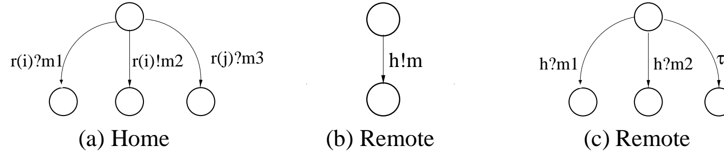


Fig. 1. Examples of communication states in the home node and remote nodes

rendezvous protocols) is through messages, i.e., other forms of communication such as global variables are not available.

As discussed before, we restrict the communication topology to a star. Since the home node can communicate with all the remote nodes and behaves like a *server* of remote-node requests, it is natural to allow generalized input/output guards in the home node protocols (e.g., Figure 1(a)). In contrast, we restrict the remote nodes to contain only input non-determinism, i.e., a remote node can either specify that it wishes to be an active participant of a single rendezvous with the home node (e.g., Figure 1(b)) or it may specify that it is willing to be a passive participant of a rendezvous on a number of messages (e.g., Figure 1(c)). Also, as in Figure 1(c), we allow τ guards in the remote node to model autonomous decisions such as cache evictions. These decisions, empirically validated on a number of real DSM protocols, help synthesize more efficient protocols. Finally, we assume that no fairness conditions are placed on the non-deterministic communication options available from a communication state, with the exception of the forward progress restriction imposed on the entire system (described below).

2.4 Forward Progress

Assuming that there are no τ loops in the home node and remote nodes, the refinement process guarantees that at least one of the refined remote nodes makes forward progress, if forward progress is possible in the rendezvous protocol. Notice that forward progress is guaranteed for some remote node, not for every remote node. This is because assuring forward progress for each remote node requires allocating too much buffer space at the home node. If there are n remote nodes, to assure that every remote node makes progress, the home node needs a buffer that can hold n requests. This is both impractical and non-scalable as n in DSM machines can be as high as a few thousands. If we were to guarantee progress only for some remote node, a buffer that can hold 2 messages suffices, as we will see in Section 3.

3 The Refinement Procedures

We systematically refine the communication actions in h and r_i by inspecting the syntactic structure of the processes. The technique is to split each rendezvous into two halves: a request for the rendezvous and an acknowledgment (ack) or negative acknowledgment (nack) to indicate the success or failure of the rendezvous. At any given time, a refined process is in one of three states: *internal*, *communication*, and *transient*. Internal and communication states of the refined process are same as in the corresponding unrefined process in the rendezvous protocol. Transient states are introduced by the refinement process in the following manner. Whenever a process P has `Q!m(e)` as one

Row	State	Buffer contents	Action
C1	Communication (Active)	empty	(a) Request for rendezvous (b) goto transient state
C2	Communication (Active)	request	(a) delete the request (b) Request home for rendezvous (c) goto transient state
C3	Communication (Passive)	request	Ack/nack the request
T1	Transient	ack	Successful rendezvous
T2	Transient	nack	go back to the communication state
T3	Transient	request	Ignore the request

Table 1. The actions taken by the remote node when it enters a communication state or a transient state. After each action, the message in the buffer is removed.

of the guards in a communication state, P sends a request to Q and awaits in a transient state for an ack/nack or a request for rendezvous from Q. In the transient state, P behaves as follows:

R1: If P receives an ack from Q, the rendezvous is successful, and P changes its state appropriately.

R2: If P receives a nack from Q, the rendezvous has failed. P goes back to the communication state and tries the same rendezvous or a different rendezvous.

R3: If P receives a request from Q, the action taken depends on whether P is the home node or a remote node. If P is a remote node (and Q is then the home node), P simply ignores the message. (This is because, as discussed in the next sentence, P "knows" that Q will get its request that is tantamount to a nack of Q's own request.) If P is the home node, it goes back to the communication state as though it received a nack ("implicit nack"), and processes the Q's request in the communication state. The rules R1–R3 govern how the remote node and home node are refined, as will now be detailed.

3.1 Refining the Remote Node

Every remote node has a buffer to store one message from the home node. When the remote node receives a request from the home node, the request would be held in the buffer. When a remote node is at a communication or transient state, its actions are shown in Table 1. The rows of the table are explained below.

C1: When the remote node is in a communication state, and it wishes to be an active participant of the rendezvous, and no request from home node is pending in the buffer, the remote node sends a request for rendezvous to home, goes to a transient state and awaits for an ack/nack or a request for rendezvous from home node.

C2: This row is similar to C1, except that there is a request from home is pending in the buffer. In this case also, the remote sends a request to home and goes to a transient state. In addition, the request in the buffer is deleted. As explained in rule R3, when the home receives the remote's request, it acts as though a nack is received (implicit nack) for the deleted request.

C3: When the remote node is in a communication state, and it is passive in the rendezvous, it waits for a request for rendezvous from home. If the request satisfies any guards of the communication state, it sends an ack to the home and changes state to

reflect a successful rendezvous. If not, it sends a nack to home and continues to wait for a matching request. In both cases, the request is removed from the buffer.

T1, T2: If the remote node receives an ack, the rendezvous is successful, and the state of the process is appropriately changed to reflect the completion of the rendezvous. If the remote node receives a nack from the home, it is because the home node does not have sufficient buffers to hold the request. In this case, the remote node goes back to communication state and retransmits the request, and reenters the transient state.

T3: As explained in the rule R3, if the remote node receives a request from home, it simply deletes the request from buffer, and continues to wait for an ack/nack from home.

3.2 Refining the Home Node

The home node has a buffer of capacity k messages ($k \geq 2$). All incoming messages are entered into the buffer when there is space, with the following exception. The last buffer location (called the *progress buffer*) is reserved for an incoming request for rendezvous that is known to complete a rendezvous in the current state of the home. If no such reservation is made, a livelock can result. For example, consider the situation when the buffer is full and none of the requests in the buffer can enable a guard in the home node. Due to lack of buffer space, any new requests for rendezvous must be nacked, thus the home node can no longer make progress. In addition, when the home node is in a transient state expecting an ack/nack from r_i, an *additional* buffer need to be reserved so that a message (ack, nack, or request for rendezvous) from r_i can be held. We refer to this buffer as *ack buffer*. When the home is in a communication or transient state, the actions taken are shown in Table 2. The rows of this table are explained below.

C1: When the home is in a communication state, and it can accept one or more requests pending in the buffer, the home finishes rendezvous by arbitrarily picking one of these messages.

C2: If no requests pending in the buffer can satisfy any guard of the communication state, and one of the guards of the communication state is $r_i!m_i$, home node sends a request for rendezvous to r_i, and enters a transient state. As described above, before sending the message, it also reserves ack buffer, to assure hold the messages from r_i. This step may require the home to generate a nack for one of the requests in the buffer in order to free the buffer location. Also note that condition (c) states that no request from r_i is pending in the buffer. The rationale behind this condition is that, if there is a request from r_i pending, then r_i is at a communication state with r_i being the active participant of the rendezvous. Due to the syntactic restrictions placed on the description of the remote nodes, r_i can't satisfy any requests for rendezvous in this communication state. Hence it is wasteful to send any request to r_i in this case.

T1: A reception of an ack while in transient state indicates completion of a pending rendezvous.

T2: A reception of an ack while in transient state indicates failure to complete a rendezvous. Hence the home goes back to the communication state. From that state, it checks if any new request in the buffer can satisfy any guard of the communication state. If so, an ack is generated corresponding to that request, and that rendezvous is completed. If not, the home tries the next output guard of the communication state. If there are no more output guards, it starts all over again with the first output guard. The

Row	State	Condition	Action
C1	Communication	buffer contains a request from r_i that satisfies a rendezvous	(a) an ack is sent to r_i (b) delete request from buffer
C2	Communication	(a) no request in the buffer satisfies any required rendezvous (b) home node can be active in a rendezvous with r_i on m_i (i.e. $r_i!m_i$ is a guard in this state) (c) no request from r_i is pending in buffer	(a) ack buffer is allocated (if not enough buffer space a nack may be generated) (b) a request for rendezvous is sent to r_i (c) goto transient state
T1	Transient	ack from r_i	rendezvous is completed
T2	Transient	nack from r_i	rendezvous failed. Go back to the communication state and send next request. If no more requests left, repeat starting with the first guard.
T3	Transient	(a) request from r_i (b) waiting for ack/nack from r_i	treat the request as a a nack plus a request
T4	Transient	(a) request from $r_j \neq r_i$ has arrived (b) waiting for ack/nack from r_i (c) buffer has > 2 free entries	enter the request into buffer
T5	Transient	(a) request from $r_j \neq r_i$ has arrived (b) waiting for ack/nack from r_i (c) buffer has 2 free entries (d) the request can satisfy a guard in the communication state	enter the request into progress buffer
T6	Transient	request from r_j has arrived (all cases not covered above)	nack the request

Table 2. Home node actions when it enters a communication or transient state.

reason for this is that, even though a previous attempt to rendezvous has failed, it may now succeed, because the remote node in question might have changed its state through a τ guard in its communication state.

T3: When the home is expecting an ack/nack from r_i, if it receives a request from r_i instead, it uses the implicit nack rule, R3. It first assumes that a nack is received, hence it goes to the communication state, where all the requests, including the request from r_i, are processed as in row T2.

T4: If the home receives a request from r_j while expecting an ack/nack from a different remote r_i, and there is sufficient room in the buffer, the request is added to the buffer.

T5: When the home is in a transient state, and has only two buffer spaces, if it receives a message from r_j, it adds the request to buffer according to the buffer reservation scheme, i.e., the request is entered into the progress buffer iff the request can satisfy one of the guards of the communication state. If the request can't satisfy any guards, it would be handled by row T6.

T6: When a request for rendezvous from r_j is received, and there is insufficient buffer space (all cases not covered by T4 and T5), home nacks r_j. r_j retransmits the message.

3.3 Request/Reply Communication

The generic scheme outlined above replaces each rendezvous action with two messages: a request and an ack. In some cases, it is possible to avoid ack message. An example is when two messages, say `req` and `repl` are used in the following manner: `req` is sent from the remote node to home node for some service. The home node, after receiving the `req` message, performs some internal actions and/or communications with other remote nodes and sends a `repl` message to the remote node. In this case, it is possible to avoid exchanging ack for both `req` and `repl`. If statements `h!req(e)` and `h?repl(v)` always appear together as `h!req(e); h?repl(v)` in remote node, and r_i`!repl` always appears *after* r_i`?req` in the home node, then the acks can be dropped. This is because whenever the home node sends a `repl` message, the remote node is always ready to receive the message, hence the home node doesn't have to wait for an ack. In addition, a reception of `repl` by the remote node also acts as an ack for `req`. Of course, if the remote node receives a nack instead of `repl`, the remote node would retransmit the request for rendezvous. This scheme can also be used when `req` is sent by the home node and the remote node responds with a `repl`.

4 Correctness

We argue that the refinement is correct by analyzing the different scenarios that can arise during the execution of the asynchronous protocol. The argument is divided into two parts: (a) all rendezvous that happen in the asynchronous protocol are allowed by the rendezvous protocol, and (b) forward progress is assured for at least one remote node.

The rendezvous finished in the asynchronous protocol when the remote node executes rows C1, C3, or T1 of Table 1 and the home node executes rows C1 or T1 of Table 2. To see that all the rendezvous are in accordance with the rendezvous protocol, consider what happens when a remote node is the active participant in the rendezvous (the case when the home node is the active participant is similar). The remote node r_i sends out a request for rendezvous to the home h and starts waiting for an ack/nack. There are three cases to consider. (1) h does not have sufficient buffer space. In this case the request is nacked, and no rendezvous takes place. (2) h has sufficient buffer space, and it is in either an internal state or a transient state where it is expecting an ack/nack from a different remote node, r_j. In this case, the message is entered into the h's buffer. When h enters a communication state where it can accept the request, it sends an ack to r_i, completing the rendezvous. Clearly, this rendezvous is allowed by the rendezvous protocol. If h sends a nack to r_i later to make some space in buffer (row C2), r_i would retransmit the request, in which case no rendezvous has taken place. (3) h has sent a request for rendezvous to r_i and is waiting for an ack/nack from r_i in a transient state (this corresponds to the rule R3). In this case, r_i simply ignores the request from h. h knows that its request would be dropped. Hence it treats the request from r_i as a combination of nack for the request it already sent and a request for rendezvous. Thus, this case becomes exactly like one of the two cases above, and h generates an ack/nack accordingly; hence if an ack is generated it would be allowed by the rendezvous protocol. An ack is generated only in case 2 which completes a rendezvous in asynchronous protocol. This rendezvous is also allowed by the rendezvous protocol.

We formalized above argument in PVS [17] and proved that the refinement rules are *safety preserving*; i.e., we showed that if the a transition is taken in the refined protocol, then it is allowed in the original rendezvous protocol. We constructed an abstraction function, abs, based on [18]. abs maps a state in the asynchronous protocol to a state in the rendezvous protocol, and showing that for every sequence of states in the asynchronous protocol, there is an equivalent sequence of states in the rendezvous protocol. Of course, since the asynchronous protocol implements a communication action in multiple steps while the rendezvous protocol implements the same communication in a single step, abs must allow stuttering steps. Let S_a be the set of states in the asynchronous protocol, $q \rightarrow_a q^{'}$ indicate a state transition from q to $q^{'}$ in the asynchronous protocol, and $q \rightarrow_r q^{'}$ indicate a state transition from q to $q^{'}$ in the rendezvous protocol. Then, using PVS, we established the following equation:

$$\forall q_a {\in} S_a \; \forall q_a^{'} {\in} S_a : \quad q_a \rightarrow_a q_a^{'} \Rightarrow abs(q_a) = abs(q_a^{'}) \; \vee \; abs(q_a) \rightarrow_r abs(q_a^{'}). \quad (1)$$

The abstraction function abs behaves as follows. If a process is in a communication state, and there is an *ack* towards it, then the state of the process is changed to reflect the completion of the rendezvous. If a process is in a communication state, and there is no *ack* towards it, then the state is changed back to the communication state. Finally, all messages in the medium and buffers are removed. Using the higher-order functions, and showed that $\rightarrow_a$ as defined by Tables 1 and 2, along with the above abs function satisfies Equation 1. The PVS theory files and proofs can be obtained from the first authors WWW home page.

Proof of forward progress

To see that at least one of the remote nodes makes forward progress, we observe that when the home node h makes forward progress, one of the remote nodes also makes forward progress. Since we disallow any process to stay in internal states forever, from every internal state, h eventually enters a communication state from which it may go to a transient state. Note that because of the same restriction, when h sends a request to a remote node, the remote would eventually respond with an ack, nack, or a request for rendezvous. If any forward progress is possible in the rendezvous protocol, we show that h would eventually leave the communication or the transient state by the following case analysis.

1. h is in a communication state, and it completes a rendezvous by row C1 of Table 2. Clearly, progress is being made.
2. h is in a communication state, and conditions for row C1 and C2 of Table 2 are not enabled. h continues to wait for a request for rendezvous that would enable a guard in it. Since a buffer location is used as progress buffer, if progress is possible in the rendezvous protocol, at least one such request would be entered into the buffer, which enables C1.
3. h is in a communication state, row C2 of Table 2 is enabled. In this case, h sends a request for rendezvous, and goes to transient state. Cases below argue that it eventually makes progress.
4. h is in a transient state, and receives an ack. By row T1 of Table 2, the rendezvous is completed, hence progress is made.

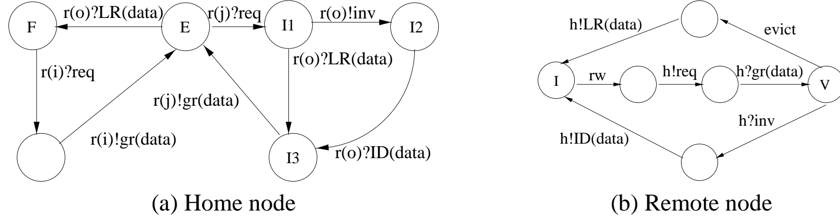
5. h is in a transient state, and receives a nack (row T2 of Table 2) or an implicit nack (row T3 of Table 2). In response to the nack, the home goes back to the communication state. In this case, the progress argument is based on the requests for rendezvous that h has received while it was in the transient state, and the buffer reservation scheme. If one or more requests received enable a guard in the communication state, at least one such request is entered into the buffer by rows T4 or T5. Hence an ack is sent in response to one such request when h goes back to the communication state (row C1), thus making progress. If no such requests are received, h sends request for rendezvous corresponding to another output guard (row C2) and reenters the transient state. This process is repeated until h makes progress by taking actions in C1 or T1. If any progress is possible, eventually either T1 would be enabled, since h keeps trying all output guards repeatedly, or C1 would be enabled, since h repeatedly enters communication state repeatedly from T2 or T3, and checks for incoming requests for rendezvous. So, unless the rendezvous protocol is deadlocked, the asynchronous protocol makes progress.

5 Example Protocol

We take the rendezvous specification of migratory protocol of Avalanche and show how the protocol can be refined using the refinement rules described above. (The architectural team of Avalanche had previously developed the asynchronous migratory protocol without using the refinement rules described in this paper.) The protocol followed by the home and remote nodes is shown in Figure 2. Initially the home node starts in state F (free) indicating that no remote node has access permissions to the line. When a remote node r_i needs to read/write the shared line, it sends a `req` message to the home node. The home node then sends a `gr` (grant) message to r_i along with data. In addition, the home node also records the identity of r_i in a variable `o` (owner) for later use. Then the home node goes to state E (exclusive). When the owner no longer needs the data, it may relinquish the line (`LR` message). As a result of receiving the `LR` message, the home node goes back to F. When the home node is in E, if it receives a `req` from another remote node, the home node revokes the permissions from the current owner and then grants the line to the new requester. To revoke the permissions, it either sends an `inv` (invalidate) message to the current owner `o` and waits for the new value of data (obtained through `ID` (invalid done) message), or waits for a `LR` message from `o`. After revoking the permissions from the current owner, a `gr` message is sent to the new requester, and the variable `o` is modified to reflect the new owner.

The remote node initially starts in state I (invalid). When the CPU tries to read or write (shown as `rw` in the figure), a `req` is sent to the home node for permissions. Once a `gr` message arrives, the remote node changes the state to V (valid) where the CPU can read or write a local copy of the line. When the line is evicted (for capacity reasons, for example), a `LR` is sent to the home node. Or, when another remote node attempts to access the line, the home node may send an `inv`. In response to `inv`, an `ID` (invalid done) is sent to the home node and the line reverts back to the state I.

To refine the migratory protocol, we note that the messages `req` and `gr` can be refined using the request/reply strategy, where the remote node sends `req` and the home node sends `gr` in response. Similarly, the messages `inv` and `ID` can be refined using



(a) Home node (b) Remote node

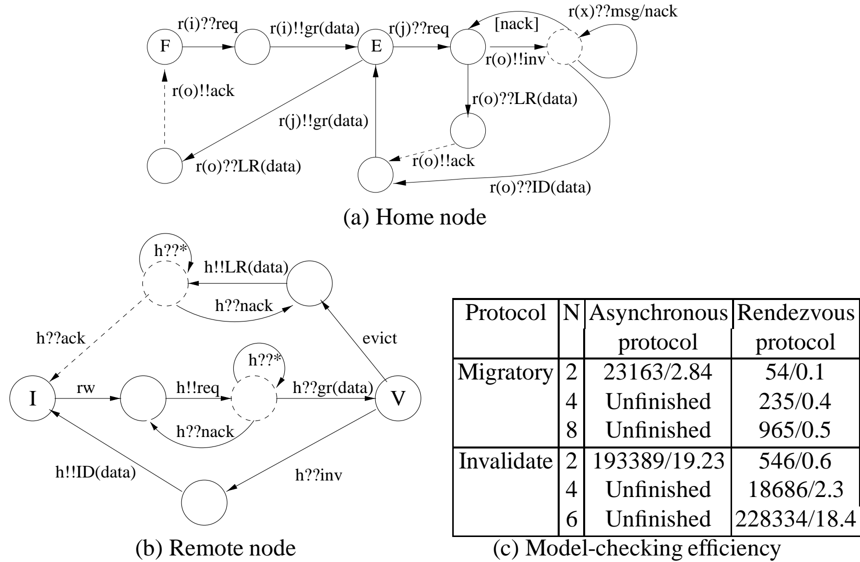
Fig. 2. Rendezvous migratory protocol

request/reply, except that in this case `inv` is sent by the home node, and the remote node responds with an `ID`. By following the request/reply strategy, a pair of consecutive rendezvous such as r_i?`req`; r_i!`gr` or r_i!`inv`; r_i?`ID (data)` takes only 2 messages as shown in Figures 3.

The refined home and remote nodes are shown in Figure 3. In these figures, we use "??" and "!!" instead of "?" and "!" to emphasize that the communication is asynchronous. In both these figures, transient states are shown as dotted circles (the dotted arrows are explained later). As discussed in Section 3.2, when the refined home node is in a transient state, if it receives a request from the process from which it is expecting an ack/nack, it would be treated as a combination of a nack and a request. We write `[nack]` to imply that the home node has received the nack as either an explicit nack message or an implicit nack. Again, as discussed in Section 3.2, when the home node doesn't have sufficient number of empty buffers, it nacks the requests, irrespective of whether the node is in an internal, transient, or communication state. For the sake of clarity, we left out all such nacks other than the one on transient state (labeled `r(x)??msg/nack`). As explained in Section 3.1, when the remote node is in a transient state, if it receives a message from the home node, the remote node ignores the message; no ack/nack is ever generated in response to this request. In Figure 3, we showed this as a self loop on the transient states of remote node with label `h??*`.

The asynchronous protocol designed by the Avalanche design team differs from the protocol shown in Figure 3 in that in their protocol the dotted lines are τ actions, i.e., no ack is exchanged after an `LR` message. We believe that the loss of efficiency due to the extra ack is small. We are currently in the process of quantifying the efficiency of the asynchronous protocol designed by hand and the asynchronous protocol obtained by the refinement procedure.

Verification: As can be expected, verification of the rendezvous protocols is much simpler than verification the asynchronous protocols. We model-checked the rendezvous and asynchronous versions of the migratory protocol above and invalidate, another DSM protocol used in Avalanche, using the SPIN [11]. The number of states visited and time taken in seconds on these two protocols are shown in Figure 3(c). The complexity of verifying the hand designed migratory or invalidate is comparable to the verification of asynchronous protocol. As can be seen, verifying of the rendezvous protocol generates far fewer states and takes much less run time than verifying the asynchronous protocol. In fact, the rendezvous migratory protocol could be model checked for up to 64 nodes in 32MB of memory, while the asynchronous protocol can be model checked



(a) Home node

(b) Remote node

Protocol	N	Asynchronous protocol	Rendezvous protocol
Migratory	2	23163/2.84	54/0.1
	4	Unfinished	235/0.4
	8	Unfinished	965/0.5
Invalidate	2	193389/19.23	546/0.6
	4	Unfinished	18686/2.3
	6	Unfinished	228334/18.4

(c) Model-checking efficiency

Fig. 3. Refined remote node of the Migratory protocol

for only two nodes in 64MB. Currently we are developing a simulation environment to evaluate the performance of the various asynchronous protocols. Both asynchronous and rendezvous versions of the migratory and invalidate protocols can be obtained from the first authors WWW home page.

6 Conclusions

We presented a framework to specify the DSM protocols at a high-level using rendezvous communication. These rendezvous protocols can be efficiently verified, for example using a model-checker. The protocol can be translated into an efficient asynchronous protocol using the refinement rules presented in this paper. The refinement rules add transient states to handle unexpected messages. The rules also address buffering considerations. To assure that the refinement process generates an efficient asynchronous protocol, some syntactic restrictions are placed on the processes. These restrictions, namely enforcing a star configuration and restricting the use of generalized guard, are inspired by domain specific considerations. We are currently studying letting two remote nodes communicate in *asynchronous* protocol so to obtain better efficiency. However, relaxing the star configuration requirement for the rendezvous protocol does not add much descriptive power. However, relaxing this constraint for the asynchronous protocol can improve efficiency.

The refinement rules presented guarantee forward progress per each line, but not per remote node. Forward progress per node can be guaranteed with modest buffer as follows. Every home manages the buffer pool as a shared resource between all the cache lines. However, instead of using a progress buffer per line, a progress buffer *per node* is used. A request from a node is entered into the shared buffer pool if there is enough

buffer space, or into the progress buffer associated that node if it satisfies a progress criterion. This strategy guarantees forward progress per node, but not per line. However, virtually all modern processors have a bounded instruction issue window. Using this property, and the observation that the protocol actions of a line do not interfere with each other (as in the case of the DSM controller in [16]), one can show that forward progress is guaranteed per each line as well as each remote node.

References

1. G. N. Buckley and A. Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM TOPLAS*, 5(2):223–235, April 1983.
2. J. B. Carter, C. Kuo, and R. Kuramkote. A comparison of software and hardware synchronization mechanisms for distributed shared memory multiprocessors. Technical Report UUCS-96-011, University of Utah, Salt Lake City, UT, USA, September 1996.
3. S. Chandra, B. Richards, and J. R. Larus. Teapot: Language support for writing memory coherency protocols. In *SIGPLAN Conference on Programming Language Design and Implementation*, May 1996.
4. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
5. Cray Research, Inc. *CRAY T3D System Architecture Overview*, hr-04033 edition, September 1993.
6. D. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. Protocol verification as a hardware design aid. In *ICCD*, pages 522–525, 1992.
7. A. Th. Eiriksson and K. McMillan. Using formal verification/analysis methods on the critical path in system design: A case study. In *CAV*, pages 367–380, 1995. Springer LNCS 939.
8. E. P. Gribomont. From synchronous to asynchronous communication. In C. Rattay, editor, *Specification and Verification of Concurrent Systems*, pages 368–383. Springer-Verilog, University of Stirling, Scotland, 1990.
9. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Apporach*. Morgan Kaufmann, 1996. Second Edition, Appendix E.
10. C. A. R. Hoare. Communicating sequential processes. *CACM*, 21(8):666–677, 1978.
11. G. J. Holzmann and D. Peled. The state of SPIN. In *CAV*, pages 385–389, New Brunswick, New Jersey, July 1996.
12. J. Kuskin and D. Ofelt et al. The Stanford FLASH multiprocessor. In *21st Annual International Symposium on Computer Architecture*, pages 302–313, May 1994.
13. L. Lamport and F. B. Schneider. Pretending atomicity. In Research Report 44*, Digital Equipment Corporation Systems Research Center*, Palo Alto, CA, May 1989.
14. D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.
15. R. J. Lipton. Reduction: A method of proving properties of parallel programs. *CACM*, 18(12):717–721, December 1975.
16. A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp scalable shared memory multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Processing*, 1995.
17. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking and model checking. In *CAV*, pages 411–414, New Brunswick, NJ, USA, 1996.
18. S. Park and D. L. Dill. Protocol verification by aggregation of distributed transactions. In *CAV*, pages 300–309, New Brunswick, NJ, USA, July 1996.