Automatic Synthesis of Cache-Coherence Protocol Processors Using Bluespec

Nirav Dave, Man Cheuk Ng, Arvind
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology, Cambridge, MA 02139
Email: {ndave, mcn02, arvind}@csail.mit.edu

Abstract

There are few published examples of the proof of correctness of a cache-coherence protocol expressed in an HDL. A designer generally shows the correctness of a protocol where many implementation details have been abstracted away. Abstract protocols are often expressed as a table of rules or state transition diagrams with an (implicit) model of atomic actions. There is enough of a semantic gap between these high-level abstract descriptions and HDLs that the task of showing the correctness of an implementation of a verified abstract protocol is as daunting as proving the abstract protocol's correctness in the first place. The main contribution of this paper is to show that this problem can be largely avoided by expressing the verified abstract protocol in Bluespec SystemVerilog (BSV), which is based on guarded atomic actions and is synthesizable into efficient hardware. Consequently, once a protocol has been verified at the rules-level, little verification effort is needed to verify the implementation. We illustrate our approach by synthesizing a non-blocking MSI cache-coherence protocol for Distributed Memory Systems and discuss the performance of the resulting implementation.

1 Introduction

A Distributed Shared Memory multiprocessor (DSM) consists of processor and memory modules that communicate via high-bandwidth, low-latency networks designed for cache-line sized messages (see Fig. 1). Each memory module serves a specific range of global addresses. Processor modules contain local caches to mitigate the long latency of global memory accesses. A DSM system that is designed to support shared memory parallel programming models has an underlying *memory model* and relies on a *cache-coherence protocol* to preserve the integrity of the model. In spite of decades of research, cache-coherence protocol design and implementation remains intellectually the most demanding task in building a DSM system. Please refer to Culler-Singh-Gupta textbook[5] for a more detailed description of DSMs and protocols.

Cache-coherence protocols need to keep large amounts

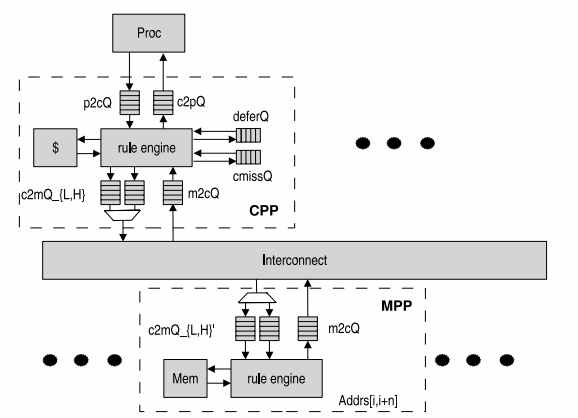

Figure 1: Diagram of DSM System

of frequently accessed data in various caches coherent with each other and with the global memory. People have devised ingenious schemes for this purpose, and as a result, realistic cache-coherence protocols tend to be complex. e.g., see [2, 7, 9, 17]. Nondeterministic or time-dependent behavior is an integral part of cache-coherence protocols which makes the task of formal verification of *real* protocols quite formidable; sufficiently so that the design and verification of cache-coherence protocols continues to be a topic of research both in academia and industry [1, 4, 12, 11, 14, 15, 16, 10].

For verification purposes, a protocol is often expressed as a table of rules or state-transition diagrams with an (implicit) model of atomic actions. These tables are themselves derived by stepwise refinement where a big atomic action, i.e., one involving state information from physically distant elements, is broken down into smaller atomic actions [10, 15]. Any implementation detail that unnecessarily complicates the verification process is left out. Some other details, which are considered important but don't fit the tabular description, are included informally as comments in the protocol description. Formal proofs rarely go beyond showing that the refinement process from big global atomic actions into small local atomic actions is correct.

The next step involves implementing the *verified ab-*

0-7803-9227-2/05/$20.00 ©2005 IEEE

stract protocol using a language like C or Verilog which does not provide a model of atomic actions. Any modification of the protocol during the implementation stage to meet real-world constraints (e.g. timing, area) essentially renders most of the original verification work useless. In fact, the task of proving that the implementation of a verified abstract protocol is correct is often more difficult than the task of verifying the correctness of original abstract protocol.

Shen and Arvind have shown how cache-coherence protocols can be naturally and precisely represented using *Term Rewriting Systems* (TRSs) [14, 15, 16, 17]. Furthermore, they have shown that such TRS descriptions lend themselves to formal verification, especially if the rules are divided into *mandatory* and *voluntary* rules. Mandatory rules are needed to make essential state transitions and must be applied when applicable, while voluntary rules are applied based on some policies which affect only the performance but not the correctness of the protocol. *The main contribution of this paper is to show that such protocol descriptions can be automatically synthesized into hardware when they are expressed in Bluespec SystemVerilog (BSV), a language based on guarded atomic actions*. The translation of TRS rules into BSV[3] is straightforward due to BSV's *rule atomicity*. We show what extra steps have to be taken, especially with regards to fairness of rule firings, to ensure the correctness of implementation.

We illustrate our technique using a non-blocking MSI cache-coherence protocol as our running example. Although the basic elements of this protocol are well known, the specific protocol we present is our own. Our classroom experience shows that it is easy to introduce mistakes if one tries to optimize the protocol. Often, these mistakes go undetected for long periods of time. We give a parameterized hardware description of this protocol in BSV, and report on the synthesis numbers - area and timing - for an ASIC to implement the cache-coherence protocol processor (CPP) and the memory protocol processor (MPP). We quantify the effect of "non-blocking" on performance via a simple experiment to illustrate that such high-level synthesis will dramatically raise our ability to experiment with protocols and processor-memory interfaces in multiprocessor systems.

Paper Organization: In Section 2 we describe the MSI protocol using TRS rules or guarded atomic actions. We discuss our implementation, its correctness, and the synthesis results in Section 3. We discuss Architectural exploration in Section 4. Lastly, we state our conclusions in Section 5.

2 The MSI Protocol

Even though we present a complete protocol in this section, it is not necessary for the reader to understand its every nuance to follow the main issues. The details as noted are only important to appreciate the complexity of the protocol and associated fairness issues.

The MSI protocol we will implement is *non-blocking*, that is, a processor can have many outstanding memory references. If the first reference gets a cache miss the next reference is automatically considered for processing. Non-blocking caches may respond to processor requests out of order. Thus, each request has to be tagged so that the processor can match a response to the request. In this protocol, a pair of memory references has an enforced ordering between them if and only if they are made by the same processor and operate on the same memory address. Without loss of correctness, some extra orderings are enforced between references from a processor if the addresses in both references map onto the same memory module.

Message Type Definition		
p2c_Req	=	Load Addr | Store Addr Value
c2m_Msg	=	ShReq_L | ExReq_L | Inv_H | WBI_H | WB_H
m2c_Msg	=	ShResp_H | ExResp_H | InvReq_L | WBIReq_L | WBReq_L

Sender	**Receiver**	**Information**
Proc	CPP	⟨ Tag, p2c_Req⟩
CPP	Proc	⟨ Tag, Value⟩
CPP	MPP	⟨CID, c2m_Msg, Addr, CLine⟩
MPP	CPP	⟨CID, m2c_Msg, Addr, CLine⟩

Figure 2: Protocol Messages

In the MSI protocol, the memory addresses stored in the caches exist in one of three stable states: read-only amongst some subset of the caches (*Sh*), held in a modified state by one cache (*Ex*), or stored only in main memory (*Inv*). Since, we are not always able to make an *atomic transition* from one stable state to another, there are transient states (e.g., *Pen*) to represent partial transitions. A directory (*MDIR*) is used to keep track of the current state of each address. It records whether each address is held exclusively by a single cache (*E*), shared amongst a set of them (possibly the empty set) (*S*), or transitioning between the two (*T*).

In the following sections, we will describe how the Cache Protocol Processor (CPP) handles processor requests and how the CPP and the memory protocol processor (MPP) together handle each other's requests. Figure 2 summarizes the messages which are used in the system.

Messages are divided into two categories: high and low priority. Messages representing new requests (`InvReq`, `WBReq`, `WBIReq`, `ShReq`, `ExReq`) are labeled as low priority. Responses from these requests (`Inv`, `WB`, `WBI`, `ShResp`, `ExResp`) are labeled as high priority. To pre-

vent deadlock, a low-priority request must not prevent the flow of a high-priority message indefinitely. We accomplish this by using a pair of queues to hold messages of each priority whereever necessary.

2.1 Protocol Description

The following description should be read in conjunction with the summary of rules given in Figures 3, 4, and 5.

Handling Processor Requests at the Cache: A processor makes a `Load` or `Store` request by placing it into the Processor-to-Cache queue (`p2cQ`). In addition to this queue, the CPP maintains two more queues: the *deferred queue* (`deferQ`) and the *cache miss queue* (`cmissQ`) (see Figure 1). `deferQ` holds requests whose processing may have to be deferred for various reasons. If it ever becomes full, the CPP stops processing new requests until the current request from the processor can be processed. The `cmissQ` holds requests that have been forwarded from the cache to the memory for processing. For buffer management, it is necessary to limit the number of such requests and consequently if `cmissQ` becomes full, additional requests cannot be forwarded to the memory. Instead, they can be placed into `deferQ` if desired, or held up in `p2cQ`.

The CPP handles the request at the head of `p2cQ` by first checking whether the address is present in the deferred queue. If the address is found in `deferQ`, the new request is also placed in `deferQ` (see Figure 3). Otherwise, the CPP looks up the address in the cache. For a `Load` request, it returns the value if the address is in the cache (i.e. its state is either `Sh` or `Ex`). A `Store` request is handled immediately only if the address is in the `Ex` state. If either of these cases occurs, the CPP handles the request by appropriately reading or writing the cache and enqueuing a response in `c2pQ`.

If the requested address is in the `Pen` state in the cache, the request is placed in the deferred queue. If the requested address is missing from the cache and the cache is not full, the CPP enqueues an appropriate request (i.e., `ShReq` for a `load`, or a `ExReq` for a `store`) to `c2mQ`. It also marks the current address as pending (`Pen`) in the cache. If the cache is full, the CPP is first forced to evict another memory address from the cache. If the victim address is in the `Sh` state, it sends an Invalidate message (`Inv`); if it is in the `Ex` state, it sends a Writeback-and-Invalidate message (`WBI`) to the directory. Finally, if the request is `Store` and the address is in the `Sh` state, the CPP invalidates the `Sh` copy before it sends the `ExReq`.

When the message at the head of `deferQ` is also available for processing, it is handled just like a new request. If it cannot be completed, it remains in `deferQ`. Usually, incoming requests have priority over the deferred ones for performance reasons.

Handling Cache Messages at the Memory: MPP maintains two queues: A high-priority and low-priority queue. Each incoming message is placed into one of these queues immediately according to its priority. The MPP processes messages in the low-priority queue, only when the high-priority queue is empty. If these queues are full, the MPP stops accepting new messages until there is a space created by an outgoing message. For deadlock avoidance, the size of the low-priority queue has to be large enough to receive the maximum number of low-priority requests it can receive from all of the caches. Since we have limited the number of outstanding requests from each cache, this size is known and related to the size of cache miss queue (`cmissQ`).

When MPP receives a `ShReq` for address a from cache c, it checks if a is in state `S` and c is not already a member of the directory. If so, it sends a `ShResp` to c and adds c to its directory. If a is exclusively held by another cache c'; the MPP stalls on this request and requests c' to write back the value (with a `WBReq`) and change a's state to `T` (see Figure 4).

When MPP gets an `ExReq` for address a from cache c, and no other cache has a copy, it marks the a as exclusively held by c and sends a `ExResp` response to c. Otherwise, if the address is in state `S` it sends an Invalidate request (`InvReq`) to all other caches with outstanding copies. If, on the other hand, the address is in state `E` it sends a Writeback-and-Invalidate request (`WBIReq`) to the cache holding the address exclusively. In both cases, the state is changed to `T` after sending the request.

When MPP receives an Invalidate (`Inv`) message from a cache c, it removes c from directory. If it receives a writeback-and-invalidate (`WBI`) message from cache c, it writes back the data as well removes it from c. If it receives a Writeback (`WB`) message, it writes back the new value into the address and changes the address state to `S`.

Handling Messages from Memory at the Cache: When the CPP receives a response from memory (`ShResp`, `ExResp`), it changes the cache state as appropriate, removes the corresponding entry in `cmissQ` and puts a response in the `c2pQ`. After that, it dequeues the memory response (see Figure 5).

While handling a request from memory, CPP modifies the cache state and sends a response back to memory only if the address is in the appropriate state. Otherwise, it just removes the memory request. That is, for an `InvReq`, it invalidates the `Sh` copy; for a `WBIReq`, it writebacks the `Ex` copy and invalidates the address; and for a `WBReq`, it writes back the data and changes the `Ex` to the `Sh` state.

2.2 Voluntary Rules

In the previous sections, we saw that when the CPP handled a request from the processor it could issue a `ShReq` or `ExReq` to memory. However, there is no reason why the CPP is limited to sending messages only when necessary. It could send a message voluntarily, in effect doing a prefetch of a value. Similarly, the CPP can also preemp-

P2C request[4]	deferQ State	CState	Action	Next CState
Load(a)	$a \in$ deferQ	-	req $\rightarrow$ deferQ[1]	-
	$a \notin$ deferQ	Cell(a,v,Sh)	$retire$[2]	Cell(a,v,Sh)
	$a \notin$ deferQ	Cell(a,v,Ex)	$retire$[2]	Cell(a,v,Ex)
	$a \notin$ deferQ	Cell(a,-,Pen)	req $\rightarrow$ deferQ[1]	Cell(a,-,Pen)
	$a \notin$ deferQ	$a \notin$ cache	if cmissQ.isNotFull $then$ $\langle$ShReq, a, L$\rangle \rightarrow$ Mem, req $\rightarrow$ cmissQ $else$ req $\rightarrow$ deferQ [1]	Cell(a,-,Pen) $a \notin$ cache
Store(a,v)	$a \in$ deferQ		req $\rightarrow$ deferQ[1]	
	$a \notin$ deferQ	Cell(a,-,Sh)	$\langle$Inv, a, H$\rangle \rightarrow$ Mem, Keep req	$a \notin$ cache
	$a \notin$ deferQ	Cell(a,-,Ex)	$retire$[2]	Cell(a,v,Ex)
	$a \notin$ deferQ	Cell(a,-,Pen)	req $\rightarrow$ deferQ[1]	Cell(a,-,Pen)
	$a \notin$ deferQ	$a \notin$ cache	if cmissQ.isNotFull $then$ $\langle$ExReq, a, L$\rangle \rightarrow$ Mem, req $\rightarrow$ cmissQ $else$ req $\rightarrow$ deferQ [1]	Cell(a,-,Pen) $a \notin$ cache
$voluntary\ rule$	-	Cell(a,-,Sh)	$\langle$Inv, a, H$\rangle \rightarrow$ Mem[3]	$a \notin$ cache
	-	Cell(a,v,Ex)	$\langle$WBI, a, v, H$\rangle \rightarrow$ Mem[3]	$a \notin$ cache
	-	Cell(a,v,Ex)	$\langle$WB, a, v, H$\rangle \rightarrow$ Mem[3]	Cell(a,v,Sh)

[1] deferQ must not be full for this operation, otherwise, req will remain in the p2cQ

[2] $retire$ means a response is sent to the requesting processor and the input request is deleted

[3] c2mHiQ must not be full for this operation

[4] The rules for handling deferQ requests are almost identical and not shown

Figure 3: Rules for Handling P2C Requests at Cache-site

C2M Message	Priority	MState	MDIR	Action	Next MState	Next MDIR
ShReq(c,a)	Low	-	$\emptyset$[1]	$\langle$ShResp, a, Mem[a] $\rangle \rightarrow c$, deq c2m Message	S	$\{c\}$
		S	$c \notin$ MDIR	$\langle$ShResp, a, Mem[a] $\rangle \rightarrow c$, deq c2m Message	S	MDIR + $\{c\}$
		E	$\{c'\}$, $c' \neq c$	$\langle$WBReq, a $\rangle \rightarrow c$	T	$\{c'\}$
ExReq(c,a)	Low	-	$\emptyset$[1]	$\langle$ExResp, a, Mem[a] $\rangle \rightarrow c$, deq c2m Message	E	$\{c\}$
		S	$c \notin$ MDIR	$\forall c' \in$ MDIR. $\langle$InvReq, a $\rangle \rightarrow c'$,	T	MDIR
		E	$\{c'\}$, $c' \neq c$	$\langle$WBIReq, a $\rangle \rightarrow c'$	T	$\{c'\}$
Inv(c,a)	High	$mstate$	$c \in$ MDIR	deq c2m Message	$mstate$	MDIR - $\{c\}$
WBI(c,a,v)	High	T | E	$\{c\}$	Mem[a]:=v, deq c2m Message	S	$\emptyset$
WB(c,a,v)	High	T | E	$\{c\}$	Mem[a]:=v, deq c2m Message	S	$\{c\}$

[1] any state with MDIR = $\emptyset$ is treated as S with $\emptyset$

Figure 4: Rules for Handling Cache Messages at Memory-site

M2C Message	CState	Action	Next CState
ShResp(a,v)	Cell(a,-,Pen)	remove Load(a) from cmissQ deq m2cQ, $retire$[1]	Cell(a,v,Sh)
ExResp(a,v_1)	Cell(a,-,Pen)	remove Store(a,v_2) from cmissQ, deq m2cQ, $retire$[1]	Cell(a,v_2,Ex)
InvReq(a)	Cell(a,-,Sh)	$\langle$Inv, a, H$\rangle \rightarrow$ Mem[2], deq m2cQ,	$a \notin$ cache
	$a \notin$ cache	deq m2cQ	$a \notin$ cache
	Cell(a,-,Pen)	deq m2cQ	Cell(a,-,Pen)
WBIReq(a)	Cell(a,v,Ex)	$\langle$WBI, a, v, H$\rangle \rightarrow$ Mem[2], deq m2cQ	$a \notin$ cache
	Cell(a,-,Sh)	$\langle$Inv, a, H$\rangle \rightarrow$ Mem[2], deq m2cQ	$a \notin$ cache
	$a \notin$ cache	deq m2cQ	$a \notin$ cache
	Cell(a,-,Pen)	deq m2cQ	Cell(a,-,Pen)
WBReq(a)	Cell(a,v,Ex)	$\langle$WB, a, v, H$\rangle \rightarrow$ Mem[2], deq m2cQ	Cell(a,v,Sh)
	Cell(a,-,Sh)	deq m2cQ	Cell(a,-,Sh)
	$a \notin$ cache	deq m2cQ	$a \notin$ cache
	Cell(a,-,Pen)	deq m2cQ	Cell(a,-,Pen)

[1] $retire$ means a response is sent to the requesting processor and the input request is deleted

[2] c2mHiQ must not be full for this operation

Figure 5: Rules for Handling Memory Responses at Cache-site

tively, invalidate or writeback a value and the MPP can preemptively request an invalidation from any CPP. We have listed three voluntary rules in Figure 3. These rules are needed to deal with capacity misses. Though we do not give details, one can also associate (and synthesize) a policy for invoking voluntary rules.

2.3 Requirements for Correct Implementation

Even in the verification of the abstract protocol we have to make sure that the buffer sizes are large enough to handle the maximum number of outgoing requests to avoid deadlocks. For the MPP to be able to accept a high-priority message anytime, it must have space to hold the maximum number of low-priority request messages in the system at once plus one. Since the size of `cmissQ` for each CPP is exactly the number of outstanding requests that the CPP can have at any given time, the minimum size of the low-priority queue in the MPP is $ns + 1$ where n is the number of caches and s is the size of the cache miss queue.

Buffer sizes are usually determined in a separate refinement step. Also showing that the system is free of deadlocks involves a different kind of proof technique than showing correctness of the big-atomic-step to small-atomic-step refinement. If the buffer requirement turns out to be too large to avoid deadlocks but the probability of deadlock can be made vanishingly small by much smaller buffer sizes, an implementer may choose to tolerate a small probability of deadlock. Therefore, it is important that cache protocol engines be parameterized by various queue sizes.

Beyond this sizing requirement, the protocol makes the following assumptions for its correct operation:

1. **Atomicity:** All actions of a rule are performed atomically. This property permits us to view the behavior as a sequence of atomic updates to the state.
2. **Progress:** If there are actions that can be performed, an action must be performed.
3. **Fairness:** A rule cannot be starved. As long as the rule is enabled repeatedly, it must be chosen to fire eventually.

As is well known this last condition of *strong fairness* is usually quite difficult to implement[10] and requires us to design an *arbiter*.

3 Implementation

3.1 Language choice and verification

Suppose we were to implement the atomic rules as described in the previous section in a synchronous language such as Verilog, Esterel, or SystemC. For each rule first we will have to determine all the state elements (e.g. Flipflops, Registers, FIFOs, memories) that need to be updated and make sure that either all or none of them are updated when the rule is ready to fire. The complication in this process arises from the fact that two different rules may need to update some common state elements. In such a situation muxes and proper control for muxing will need to be encoded by the implementer. The situation is further complicated by the fact that for performance reasons one

may apply several rules simultaneously if they update different state elements and are conflict free. A test plan for any such implementation would be quite complex because it would have to check for various dynamic combinations of enabled signals. It is easy to see that unless the process of translation from atomic rules into a synchronous language is automated, the translation process is error prone. The fact that we started with a verified protocol can provide little comfort about the correctness of the resulting implementation.

The semantic model underlying BSV is that of guarded atomic actions. That is, a correct implementation of BSV has to guarantee that every observable behavior conforms to some sequential execution of these atomic rules. In fact, at the heart of the BSV compiler is the synthesis of a rule scheduler that selects a subset of enabled rules that do not interfere with each other[8]. The compiler automatically inserts all the necessary muxes and their controls. Therefore, if we assume that the BSV compiler is correct then a test plan for a BSV design does not have to include verification of rule atomicity. We are able to effectively leverage the verification done for the protocol itself when we translate the protocol into BSV rules.

In this section we will show that it is straightforward to express the state transition tables shown in Figures 3, 4, and 5 into BSV. In addition to this ease of translation, BSV's rich type structure built on top of SystemVerilog provides further static guarantees about the correctness of the implementation.

3.2 BSV preliminaries: Modules and Interfaces

BSV is an object-oriented hardware description language which is first compiled into a TRS after type checking and static elaboration and then further compiled into Verilog 95 or a cycle accurate C program. In BSV, a module is the fundamental unit of design. Each module roughly corresponds to a Verilog module. A module has three components: state, rules which modify that state, and methods, which provide an interface for the outside world, including other modules, to interact with the module. All state elements (e.g. registers, queues, and memories) are explicitly specified in a module. A *primitive module* in BSV is simply a Verilog module with an appropriate Bluespec wrapper and does not refer to other BSV modules. (The examples in [3], though in a somewhat different syntax, may help a reader who has had no exposure to Bluespec).

The CPP and MPP Interfaces: BSV's *interfaces*, which are simply a collection of methods, provide an abstraction for the communication channels between the processors, the caches, and the shared memory modules. A natural way to describe communication in this system is to have the sending module directly call a method of the receiving module. The other alternative is to make each module a *leaf* module (i.e., one that does not call external modules) and then create "glue" rules in a super-module to call associated methods of each module. The advantage of this second organization is that it allows the modules to be self contained and separately compilable. We use the second method because currently the BSV compiler does not support cyclic call structures.

```
module mkCPP#(Integer deferQsz,
              Integer cmissQsz,
              Integer c2mQLsz)(CPP_ifc);

FIFO p2cQ <- mkSizedFIFO(2);
FIFO c2pQ <- mkSizedFIFO(2);

FIFO deferQ <- mkSizedFIFO(deferQsz);
FIFO cmissQ <- mkSizedFIFO(cmissQsz);

FIFO c2mHiQ <- mkSizedFIFO(2);
FIFO c2mLoQ <- mkSizedFIFO(c2mQLsz);

Cache cache <- mkCache();
MDir  cdir  <- mkDir(cache);

 <rules> ...

 method put_p2cMsg(x) ...
 method get_c2pMsg()  ...
 method put_m2cMsg(x) ...
 method get_c2mMsg()  ...

endmodule
```

Figure 6: mkCPP Module

To simplify issues with modularity[6], we make the cache modules as a submodule of the Shared memory system. While this design hierarchy does not match our expectation of the physical layout, it is nevertheless a useful first step before breaking the module for physical layout. The `mkCPP` module with its interface methods is shown in Figure 6.

State Elements and Queues: Once the interfaces for our module have been specified, we can define the state elements for each module. We represent all finite queues by FIFOs, and the storages of the caches and the memory by RegFiles (Register Files). There is no restriction that all caches must be the same size, or how they are structured internally. To simplify our design, we use only direct-mapped caches. Sizes of state elements are specified as parameters in Figure 6 and need to be supplied before the `mkCPP` module can be synthesized. In a real design, we will want to buffer messages on both sides of inter-chip communication channels. Therefore, we have FIFOs at both ends of the communication channel. Since messages propagate from the sender's FIFO to the receiver's FIFO in order, the two FIFOs behaves the same as a single FIFO, but with one more cycle of latency in the worst case. However, this additional cycle will be dominated by the communication delay in the system.

```
P2CReq req = p2cQ.first();
Value cVal = cacheVal(req.addr);

rule handle_P2CQReq_Load(True);
 p2cQ.deq();
 if(inDeferQ(req.addr))
   deferQ.enq(req);
 else
   case(lookupDirState(req.addr))
     Valid(Sh) :
      c2pQ.enq(
             C2PResp{req.addr,cVal});
     Valid(Ex) :
      c2pQ.enq(
             C2PResp{req.addr,cVal});
     Valid(Pen): deferQ.enq(req);
     Invalid: begin
      c2mLoQ.enq(shReqMsg(req.addr));
      writeState(req.addr,Pen);
      cmissQ.enq(req);
     end
endrule
```

Figure 7: Load Rule in BSV

3.3 Rules as Behavioral Specifications

The behavior of a module is represented by a set of rules, each of which consist of a *predicate*, the condition that must hold for the rule to fire, and an *action* to change the hardware state of the module. For instance consider the code to represent rule corresponding to the first line of Figure 3:

```
rule ld_defer({Load .a} matches
                  p2cQ.first() &&&
                  deferQ.find(a));
    deferQ.enq(p2cQ.first());
    p2cQ.deq();
endrule
```

A note on syntax: `ld_defer` is the name of the rule; `{Load .a} matches p2cQ.first() &&& deferQ.find(a)` is the rule predicate; and `deferQ.enq(p2cQ.first()); p2cQ.deq()` are the two actions, specified as method calls, to be performed atomically. `{Load .a} matches p2cQ.first()` is a pattern matching statement and returns true if the first entry in `p2cQ` is a `Load` instruction. In that case it also binds `a` in `.a` to whatever address is associated with the `Load`.

`p2cQ.first()` is an example of a *read* method, i.e. a combinational lookup returning a value, while `deferQ.enq` and `p2cQ.deq` are examples of *action* methods which actually change the state of elements inside `deferQ` and `p2cQ` modules. To preserve rule atomicity all the affected state elements in various modules have to be updated simultaneously when the rule is applied. The BSV compiler ensures that this is indeed the case[13].

Each interface method has a *guard* or predicate which restricts when the method may be called. These guards or implicit conditions also help avoid many common coding errors. As an example, the implicit conditions that will affect the `ld_defer` rule are that `p2cQ` should not be empty and `deferQ` should not be full. These implicit conditions would have been coded as part of the FIFO module interface definitions.

The `get` and `put` methods are examples of *ActionValue* methods. An actionValue method is a combination of the read and action methods and is used when we want a value to be made available only when an appropriate action also occurs. Consider a situation where we have a first-in-first-out queue (FIFO) of values and a method that should get a new value on each call. From the outside of the module, we would look at the value only when it is being dequeued from the FIFO. Thus we would write the following:

```
get_c2pMsg = actionvalue
               c2pQ.deq;
               return(c2pQ.first())
             endactionvalue
```

Combining Mutually Exclusive Rules: Many of the rules are mutually exclusive and can be merged into a single rule for brevity. We illustrate this by merging all the Load rules from Figure 3 into one rule as shown in Figure 7.

In this rule, we look at a request at the head of the `p2cQ`. If the address is already in the `deferQ`, or the state of the address in the cache is in the `Pen` state, we move the request to the `deferQ`. Otherwise, if the address is in the cache, we send a response to the processor. If the value is not in the cache we enqueue a request to memory, mark the address as `Pen` in the cache, and enqueue the request into the `cmissQ`.

While there are no explicit conditions for this rule to fire, there are a number of implicit ones. For instance, we may need to enqueue into the `deferQ`, which requires that it not be full. If the compiler blindly concatenated all the implicit conditions onto the explicit condition (in this case "True") we would get the following predicate:

```
p2cQ.notEmpty && deferQ.notFull &&
c2pRsp.notFull && cmissQ.notFull &&
c2mLoQ.notfull
```

This rule firing condition is too conservative; for instance, consider the case where the requested address is in the cache, but `cmissQ` is full. Here, the rule would not be allowed to fire, even though nothing fundamental is blocking the action. Instead, the BSV compiler constructs a firing condition that reflects the conditional nature of actions in the rule. For example, the testing of

implicit condition for `deferQ.enq` would be limited to those cases where this action is actually used:

```
deferQ.notFull ||
(!inDeferQ(req.addr) &&
 (Valid(Pen) /=
   lookupDirState(req.addr)))
```

3.4 Fairness Requirements

Two of three requirements for a correct implementation that were given in Section 2.3 are handled directly by the BSV compiler. The *atomicity* requirement is handled by the semantics of the generated hardware. A correct BSV compiler guarantees that all state transitions can be regarded as the composition of a series of atomic actions [8, 13]. The second requirement of *forward progress* is also easily obtained, because the scheduler always chooses at least one rule to fire among the enabled rules.

The last requirement of *strong fairness* in rule scheduling is not automatically enforced by the BSV compiler. The compiler currently generates only stateless or combinational schedulers which do not remember past decisions. Such a scheduler will always prioritize the same rule over another in case the two rules conflict with each other. Consequently, if two rules are always enabled and conflict then the scheduler will *never* schedule one of the rules. We will analyze the three situations where inter-rule conflicts arise in our design:

1. A choice has to be made between the CPP rules to handle new messages (i.e., in `p2cQ`) and previously deferred messages (i.e., in `deferQ`).
2. Since response messages from different caches share the `c2mHiQ`, a choice has to be made as to which cache's responses should be handled first.
3. Similarly, since response messages from different caches share the `c2mLoQ`, a choice has to be made as to which cache's requests should be handled first.

BSV provides scheduling annotations, known as *urgency*, to prioritize rules. For example, using this annotation we can prioritize rules for processing requests in `p2cQ` over rules for processing requests in `deferQ`, or prioritize responses in `c2mHiQ` over requests in `c2mLoQ`. It is not so easy to select requests from different caches in a queue in this manner. However, as we have already pointed out static priority does not automatically guarantee fairness. We analyze each of the cases for fairness next.

In the first case, neither rule may be executed infinitely often, without the other rule executing sometime. We can see this for the rules for processing requests in `p2cQ`: since FIFOs have finite size and each processor has a limit on the maximum number of outstanding instructions, there is an upper bound on the number of instructions we can handle from the `p2cQ` before we have to handle a request in the `deferQ`. Similarly for the `deferQ` rule, we can only handle as many requests as there are in the `deferQ` before we have to handle a request in the `p2cQ`.

In the second case the fairness is guaranteed because memory blocks low priority messages (i.e., cache requests) before processing messages in the high priority queues (i.e., responses). This will cause memory to run out of response messages eventually because they are all generated as a consequence of processing some request.

We are not so fortunate in the last case. With a static prioritization a single cache could effectively starve other caches. Consider the case where there are two caches, $C1$ and $C2$, with $C1$ having priority over $C2$. $C1$ repeatedly makes a request for an address to memory and then invalidates it. While memory is handling any request, $C1$ will have created another request. This request will have to be processed before any of $C2$'s requests. Thus once $C1$ starts making requests, $C2$ requests will never have enough priority to be handled. This starvation, even though it may have low priority, is undesirable. To prevent this from happening, we need to explicitly encode some fairness into the rules.

One solution is to implement a round-robin scheme by adding a counter `ctr` to indicate which cache's request should be handled next. The predicate of each cache rule can be modified with the condition that $ctr == i$ where i is the cache number. (Notice, this will make the firing condition more restrictive, which is always a safe thing to do because it cannot add any new behavior to the system). We can formally prove that this guarantees fairness, though it is likely to drop the performance substantially as requests now need to wait for their turn even without any contention.

The scheme can be improved by keeping the original copy of each rule as well and giving it a lower "urgency" than the corresponding rule with the counter. The Counter version of the rules will guarantee that no cache will starve while the non-counter versions will guarantee that in case of no contention some rule will get to fire. This should have minimal impact on area and timing because the two versions of the rules can never execute together and will share all the resources.

There are a couple of things worth noting about the discussion in this section. First, the notion of fairness is very problem specific and is not necessarily black and white. One may tolerate unfairness if the probability of starvation is low. Second, it is possible to devise all kinds of ingenious schemes to implement fairness and it can be done in a manner that does not affect the correctness of the system. And lastly, showing that the system is indeed fair is quite involved and should be proved formally. Even though we believe the arguments we have presented are correct, it is easy to make a mistake and overlook some corner cases.

deferQ size (n)	c2mLoQ size ($4n+1$)	CPP		MPP	
		Comb. Gates	# Regs	Comb. Gates	# Regs
4	17	3875	1432	4575	1468
8	33	4835	1742	7251	2575
16	65	6240	2360	13325	4786

Figure 8: Synthesis of CPP and MPP with cmissQ of size 4

3.5 Synthesis Results

Most of the design and all of the BSV coding was done by the first two authors, who have expert level understanding of BSV and the MSI protocol presented. Once the protocol was defined in the tabular form, the design was completed in 3 man-days and took only 650 lines of BSV code to represent. Nearly half of this time was spent finding and correcting typographical errors in the translation. No other functional errors were found or introduced in this encoding phase. The rest of the time was spent tuning for performance and setting up the test bench. The final design is parameterized by the number of caches (CPP), the number of memory modules (MPP), the size of memory in each of these units, and the sizes of various queues. The design can also be modified with some effort so that it can accept various cache organizations (e.g., direct mapped, set associative) as a parameter. This parameterization of the design is important for the architectural exploration that we will discuss in the next section.

The final design was tested and synthesized for both 4 and 8 instantiated caches. The design was compiled with the Bluespec Compiler version 3.8.46, available from Bluespec Inc. Compilation of design remained roughly constant, taking 333 seconds for a 4 cache system, and 339 seconds for an 8 cache one. The generated Verilog was compiled to the TSMC $0.13\mu m$ library using Synopsys Design Compiler v. 2004.06 with a timing constraint of $5ns$. The worst case (slow process, low voltage) timing model was used. The results of synthesis of the CPP and the MPP for different sized queues is listed in Figure 8. We divided the area by 5.0922 $\mu m2$ the area of a two input NAND gate (ND2D1) to achieve gate counts.

In BSV the clock period is determined primarily by the "slowest rule"[8], the rule whose enabling condition or logic to compute the next state has the longest critical path. The actual clock period also includes the time taken by the scheduler. All of the designs were able to meet the $5ns$ timing constraint, however, the model made use of a simplified cache. As a result our timing results should not be used as a quantitative measure of the methodology. However it does lend credence to the reasonability of the generated design, since all results fell within the timing constraint.

4 Architectural Exploration

In this section, we briefly study the effect of the degree of non-blocking on the throughput of our protocol engine design by comparing three different versions of a 4-processor system. The first system has a blocking cache. Meanwhile, the second and third have non-blocking caches. All systems share the same architectural parameters except for the sizes of cmissQ, deferQ and c2mLoQ because they affect the degree of non-blocking (the functionalities of these queues are explained in Section 2.1). Figure 9 summarizes the parameters of each system, where the access latencies of the cache and the main memory are the minimum times for the cache to service a processor request, and memory to service a cache's request, respectively. For the blocking cache design, we modified the rules in Figure 3 and Figure 5 accordingly so that the instruction will remain at the head of the p2cQ and block the execution of the subsequent instructions when the cache is waiting for the reply from the memory during a cache miss.

Simulation Environment: The results presented in this paper are gathered from the Verilog Compiler Simulator (VCS) run on the Verilog files generated from the BSV compiler. The testbench used in the simulation is generated by a random memory instruction generator implemented by us for testing and verification purposes. Each test consisted of 1-million memory instructions for each processor. We collected the results of execution of 10 thousand memory instructions for each cache after a warming up period of 500 thousand memory instructions. Of the memory accesses in each instruction stream, 10% of them are stores. Additionally, 10% of all memory accesses are made to addresses which are shared by all of the processors in the system. These memory streams are fed to the cache engines through fake processor units, each with a reorder-buffer-like structure to limit the instruction window size its cache observes. The processor model feeds one instruction to its cache each cycle assuming adequate space in the reorder buffer.

Simulation Results: The simulation results in Figure 10 show that even though all three systems have comparable miss rates, the system with non-blocking caches have much lower CPI than the system with blocking caches. This indicates that our non-blocking cache protocols are successful at partially hiding the long latency of cache misses. Such experiments can help an architect decide if the price in terms of area and timing is worth the benefit of improved performance.

5 Conclusions

In this paper we first described a non-blocking MSI cache-coherence protocol for a DSM system. The protocol is

Design	Cache Type	deferQ Size	cmissQ Size	c2mLoQ Size	% Misses	CPI (Improvement)
A	Blocking	0	4	16	2.47%	2.35 (0%)
B	Non-Blocking	1	1	4	2.27%	1.68(39.43%)
C	Non-Blocking	4	4	16	2.21%	1.40 (67.83%)

Figure 10: Design Parameters and Resulting Simulation Miss Rate and CPI

Parameter	Value
Number of Processors	4
Address width	32 bit
Data width	32 bit
Processor ROB Size	64
Issue Rate	1 per cycle
p2cQ Size	2
c2pQ Size	2
Cache line width	1 word
Cache Size	128 entries
Cache Latency	4 cycles
Cache Bandwidth	1 word per cycle
c2mHiQ Size	2
m2cQ Size	2
Interconnect Bandwidth	1 request, 1 response per cycle
Memory Size	512 entries
Memory Type	blocking
Memory Latency	34 cycles
Memory Bandwidth	1 word every other cycle

Figure 9: Testing Systems Configurations

both realistic and relatively complex. We showed that it was straightforward to code the protocol in BSV. This description was both modular and parameterized to support an arbitrary number of cache units and queue sizes. We then modified our design slightly, maintaining both parameterization and modularity, to meet the *fair scheduling* requirements of this design. In doing so, we could leverage the complicated proof of correctness for the original protocol in our implementation.

The biggest challenge in this work was guaranteeing fair access to all processors in processing their cache misses. The type of strong fairness must be programmed explicitly; simple *rule urgency annotations* fail to capture the specific requirements of this design. The correctness arguments for fair scheduling in a non-deterministic setting are both difficult and problem specific. However, it may be possible to generate a strongly-fair arbitrator for a set of rules in a mechanical way that captures some common cases of fairness. We propose to investigate this further in future.

This work is part of our project to model various PowerPC microarchitectures for multiprocessor configurations. The goal is to generate a series of modular memory systems, each with its own coherence protocol but with identical interfaces to the processor modules. Our setup would provide a much more realistic setting for both protocol verification and performance measurements than traditional simulators. Additionally, we plan to map these BSV designs onto large FPGAs in the near future.

Acknowledgements: The authors would like to thank the anonymous referees whose comments drastically helped shape the final version of this paper. Funding for this work has been provided by the IBM agreement number W0133890 as a part of DARPA's PERCS Projects.

References

[1] Homayoon Akhiani, Damien Doligez, Paul Harter, Leslie Lamport, Mark Tuttle, and Yuan Yu. Cache-Coherence Verification with TLA+. In *World Congress on Formal Methods in the Development of Computing Systems, Industrial Panel*, Toulouse, France, Sept, 1999.

[2] C. Amza, A. L. Cox, S. Dwarkadas, L-J. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive protocols for software distributed shared memory. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):467–475, 1999.

[3] Bluespec, Inc., Waltham, MA. *Bluespec SystemVerilog Version 3.8 Reference Guide*, November 2004.

[4] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language support for writing memory coherence protocols. In *PLDI*, pages 237–248, 1996.

[5] D. Culler, J. P. Singh, and A. Gupta. *Modern Parallel Computer Architecture*. Morgan Kaufmann, 1997.

[6] Nirav Dave. Designing a Processor in Bluespec. Master's thesis, Electrical Engineering & Computer Science Dept., MIT, Cambridge, MA, Jan 2005.

[7] Babak Falsafi and David A. Wood. Reactive numa: A design for unifying s-coma and cc-numa. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[8] James C. Hoe and Arvind. Operation-Centric Hardware Description and Synthesis. *IEEE TRANSACTIONS on Computer-Aided Design of Integrated Circuits and Systems*, 23(9), September 2004.

[9] D. E. Lenoski. *The Design and Analysis of DASH A Scalable Directorybased Multiprocessor*. PhD thesis, Stanford University, Stanford, CA, 1992.

[10] Ratan Nalumasu and Ganesh Gopalakrishnan. Deriving efficient cache coherence protocols through refinement. In *Formal Methods for Parallel Programming: Theory and Applications (FMPPTA)*, Orlando, FL, 98.

[11] Seungjoon Park and David L. Dill. Protocol verification by aggregation of distributed transactions. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 300–310, London, UK, 1996. Springer-Verlag.

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

[13] Daniel L. Rosenband and Arvind. Modular Scheduling of Guarded Atomic Actions. In *Proceedings of DAC'04*, San Diego, CA, 2004.

[14] Xiaowei Shen. *Design and Verification of Adaptive Cache Coherence Protocols*. PhD thesis, Electrical Engineering & Computer Science Dept., MIT, Cambridge, MA, 2000.

[15] Xiaowei Shen and Arvind. Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols. In *MIT CSAIL CSG Technical Memo 398 (http://csg.csail.mit.edu/pubs/memos/Memo-398/memo398.pdf)*, January 1997.

[16] Joseph E. Stoy, Xiaowei Shen, and Arvind. Proofs of Correctness of Cache-Coherence Protocols. In *Proceedings of FME'01: Formal Methods for Increasing Software Productivity*, pages 47–71, London, UK, 2001. Springer-Verlag.

[17] Xiaowei Shen, Arvind, Larry Rudolph. CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems. In *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*, Rhodes, Greeces, Jan 1999.