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, Massachusetts 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 correctness of the abstract protocol in the first place. The main contribution of this paper is to show that 1. it is straightforward to express these protocols in Bluespec SystemVerilog (BSV), a hardware description language based on guarded atomic actions, and 2. it is possible to synthesize an hardware implementation automatically from such a description using the BSV compiler. 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.

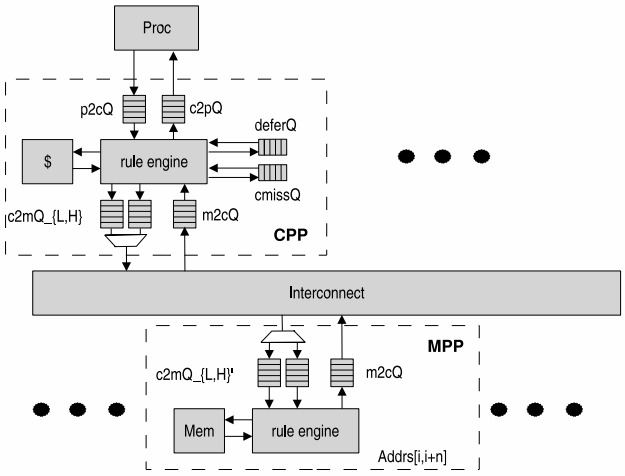

Figure 1: Diagram of DSM System

Cache coherence protocols need to keep large amounts 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, 16]. 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, 10, 11, 13, 14, 15].

For verification purposes, a protocol is often expressed as a table of rules or state-transition diagrams with an

(implicit) model of atomic actions. Many implementation details are left out because they unnecessarily complicate the verification process. Some other details, which are considered important but don't fit the tabular description, are included informally as comments in the protocol description. Furthermore, implementation languages like C or Verilog, in which abstract protocols are eventually implemented do not provide a model of atomic actions. Consequently, the verification of the implementation of a "verified abstract protocol" still poses major verification challenges. 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)[13, 14, 15, 16]. 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.

In this paper, we 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 action. The translation of TRS rules into BSV[3] is straightforward because it relies on the *rule atomicity* of BSV. 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). *The main contribution of this paper is to show that automatic synthesis of high-performance realistic cache coherence protocols expressed in Bluespec is feasible. Via some simple experimental results, we show 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 initial implementation and its correctness in Section 3. We report our synthesis and performance results in Sections 4 and 5, respectively. Lastly, we state out conclusions in Section 6.

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 reference. If the first reference gets a cache miss the next references is automatically considered for processing. Non-blocking caches may respond to processor requests out of order and 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 ordering is enforced between two 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 types of 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 prevent 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 wherever necessary, one to hold messages of each priority.

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 space created by outgoing messages. 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 the `S` state 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 be `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 `S` state, sends Invalidate request (`InvReq`) to any other cache with an outstanding copy. On the other hand, if the address is in `E` state, sends Writeback-and-Invalidate request (`WBIReq`) to the cache exclusively holding the address. For both cases, the state will be modified to `T` after sending the request message.

When MPP receives an Invalidate (`Inv`) message from

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] c2mQ$_H$ 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] c2mQ$_H$ must not be full for this operation

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

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 removing 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).

When a CPP handles a request from memory, it only modifies the cache state and sends a response back to memory if the address is in the appropriate state: it invalidates the `Sh` copy for an `InvReq`, writebacks the `Ex` copy and invalidates the address on a `WBIReq`, and writes back the data and moves from the `Ex` to the `Sh` state on a `WBReq`. Otherwise, the CPP just removes the memory request.

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 preemptively, 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 associate (and synthesize) a policy for invoking voluntary rules.

2.3 Requirements for Correct Implementation

We must ensure 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.

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 of the protocol as a sequence of atomic updates to the state.

2. **Progress:** If there is an action that can be performed, an action must be performed eventually.

3. **Fairness:** A rule cannot be starved. As long as the rule is enabled an infinite number of times, it must be chosen to fire.

As we will show in the next section, this last condition requires us to design an *arbiter*.

3 Implementation in BSV

Suppose we were to implement the TRS or atomic rules as described in the Figures 3, 4, and 5 in a synchronous language such as Verilog, Esterel, or SystemC. For each rule we will have to figure out all the state elements (e.g. Flipflops, Registers, FIFOs, memories) that need to be updated and make sure that either all of them are updated or none of them are. The complication in this process arises from the fact that two different rules may need to update some common state elements and thus, 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. Unless the process of translation from such atomic rules into a synchronous language is automated, it is easy to see that this task is error prone. The fact that we started with a verified protocol can hardly provide 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.

Next we will show that it is straightforward to express the state transition tables shown in Figures 3, 4, and 5 into BSV. Furthermore, BSV has a rich type structure built on top of SystemVerilog and it provides further static guarantees about the correctness of the implementation. Space limitations do not permit even a brief description of BSV but hopefully we will provide enough hints about the syntax and semantics of BSV so that the reader can follow the expression of atomic rules in BSV. A reader can also gain some familiarity with Bluespec by seeing examples in [3], though the syntax is somewhat different.

3.1 BSV Modules

Bluespec System Verilog (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.

A module in Bluespec can be of one of two possible types: a standard module which includes other modules as state, rules, and methods, or a primitive module which is an abstraction layer over an actual Verilog module.

All state elements (e.g. registers, queues, and memories) are explicitly specified in a module. The behavior of a module is represented by a set of rules, each of which consist of an *action*, a change on the hardware state of the module, and a *predicate*, the condition that must hold for the rule to fire. The syntax for a rule is:

```
rule ruleName(predicate);
  action;
endrule
```

The *interface* of a module is a set of methods through which the outside world interacts with the module. Each interface method has a *guard* or predicate which restricts when the method may be called. A method may either be a *read* method (i.e. a combinational lookup returning a value), or an action method, or a combination of the two, an actionValue method.

An actionValue 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:

```
getVal = actionvalue
           fifoVal.deq;
           return(fifoVal.first())
         endactionvalue
```

The action part of a rule can invoke multiple action methods of various modules. To preserve rule atomicity

```
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 c2mQ_Hi <- mkSizedFIFO(2);
FIFO c2mQ_Lo <- mkSizedFIFO(c2mQLsz);

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

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

 <other rules> ...

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

endmodule
```

Figure 6: mkCPP Module

all the affected state elements in various modules have to be updated simultaneously when the rule is applied. BSV compiler ensures that is indeed the case[12].

3.2 The CPP and MPP Interfaces

To effectively translate our rules into a parameterized Bluespec System Verilog design, we need an abstraction for the communication channels between the processors, the caches, and the shared memory modules. We can get this via BSV's interfaces, which are a collection of methods.

One of the most natural ways to describe communication in this system is to have the sending module directly call a method on the receiving module. The other alternative is to make each module a *leaf* module, that does not call external modules and then create glue actions in a super-module to stick together associated methods. The advantage of this second organization is that it allows the modules to be compiled separately. We use the second method because currently, the BSV compiler does not support cyclic call structures.

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 layout, it is nevertheless a useful first step before breaking the module for physical layout.

3.3 State Elements and Queue Sizing

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. 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.

3.4 Rules

A Bluespec rule has the same fundamental structure as a TRS rule; it consists of a predicate and an action. Therefore the transformation from TRS rules to a BSV rule is straightforward. For instance consider the rule represented by the first line of Figure 3. We describe this with the Bluespec rule `ld_defer` shown in Figure 6. ***need to explain syntax here - pattern matching, find, ...

Many of these rules are mutually exclusive and contain similar descriptions. One could desire to merge these rules into a single rule. We can see how the set Load rules from Figure 3 can be merge 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 the `deferQ`, which requires that it not be full. By default, the compiler will blindly concatenate all the implicit conditions onto the explicit condition (in this case "True") we would get the final condition:

```
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

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

This allows the rule to fire, only if the implicit conditions of the actions executed are true. However, it may not let the rule fire when it should be valid. 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.

The BSV compiler provides the `-aggressive-conditions` flag to deal with the this problem. It indicates to the compiler that the firing condition should reflect the conditional nature of actions in the rule. With this flag on, the conditional for `deferQ.enq` would be limited to those cases where the action is actually used (i.e. the implicit condition associated with the enqueue would be):

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

3.5 Fairness Requirement

Next we examine the conditions given in Section for correct implementation.

The atomicity requirement is handled by the semantics of the generated hardware. The compiler will guarantee that all state transitions can be regarded as the composition of a series of atomic actions. The second requirement of forward progress is also easily obtained, because the scheduler will always choose a rule to fire if possible.

The last requirement, strong fairness of rule scheduling is less clearly true. The compiler generates a static scheduler at compile time. This means that if there ever is a place where a choice is made, then the scheduler will always prioritize one over rule over the other. Conversely, if a rule has no conflicts the system trivially meets the strong fairness condition for that rule. If there are no conflicts, the rule will always be fired when its predicate is true. In our design there are only three situations where inter-rule conflicts arise. We need only worry about fairness in these cases.

The first case occurs in the CPP, between rules handling the instructions in `p2cQ` and `deferQ`. We see that since the FIFOs sizes are finite, we will either run out of messages in the `deferQ` or run out of free space in the `deferQ` if we consistently bias towards one rule or the other. This means that the rule that had priority would eventually not be disabled again until the other rule was fired. Thus, the compiler's scheduling provides the necessary fairness requirement. While, either priority is correct, making the compiler enforce requests from the `deferQ` first needlessly reduces the amount of memory reordering of processor requests to the cache. So we prioritize requests in `p2cQ` by simple urgency annotations on the two rules.

The second case occurs in the memory, where we choose the order of high priority message from the caches to handle. Here we notice that since the memory blocks, and handles responses before requests, and that the handling of a cache does not directly cause another cache response to be created, eventually we will run out of responses from other caches. Therefore, the scheduling again provides the necessary fairness.

We are not so fortunate in the last case. This occurs when we are choosing which of the caches' requests we should process. With a static prioritization a single cache could effectively starve another cache by constantly flooding the memory with requests. While we would still make forward progress and maintain correct state, this starvation is undesirable.

To prevent this from happening, we need to explicitly encode some amount of fairness into the rules. The most obvious solution is to add a counter which counts down each cache, and if there is request there, handles it. While this works relatively well, it leads to a substantial drop in performance.

Instead, we add a pointer which points to the next cache

to be given priority. An additional set of rules are made with higher priority than all the other rules which handle cache requests. These rules will handle requests made to the cache to which the pointer is pointing. Whenever any rule completes a request we change the pointer to point to the next cache in sequence.

Since the additional rules only ever do one of the actions associated with other rules we have in the system and its predicate for doing the operation of one of the another rules is more restrictive than that rule (it must be able to handle the rule and the pointer must be pointing the appropriate cache), the addition of this rule does not change the correctness of our implementation. Additionally, because requests will not leave their queues unless they have been handled and the memory is blocking, after processing a request, all the cache request rules which were firable would still be firable. As each time a request is handled, we change priority, a request is guaranteed to be on of the first n requests we process after it becomes handleable, where n is the number of caches in the system. Therefore this additional rule enforces the required fairness.

4 Synthesis Results

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. It only took 1050 lines of BSV code to represent the processor for this protocol. Once the protocol was defined in the tabular form, the design was completed in 3 man-days by two of the authors. Both the designers had expert level understanding of BSV and the MSI protocol presented. 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 in tuning for performance and setting up the test bench.

The final design was tested and synthesized for both 4 and 8 instantiated caches. Compilation of design remained roughly constant, taking 333 seconds for a 4 cache system, and 339 seconds for an 8 cache one.

The design was compiled with the Bluespec Compiler version 3.8.46, available from Bluespec Inc. The generated Verilog was compiled to the TSMC $0.13\mu m$ library using Synopsys Design Compiler version 2004.06 with a timing constraint of $5ns$ The worst case (slow process, low voltage) timing model was used. We divided the area by 5.0922 μm^2 the area of a two input NAND gate (ND2D1) to achieve gate counts. The results of synthesis of the CPP and the MPP for different sized queues is listed in Figure 8.

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.

5 Performance Evaluation

In this section, we briefly study the effect of the degree of non-blocking on the throughput of our protocol engine design with a comparison of three 4-processor systems. 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 `c2mQ_L` because they affect the degree of non-blocking (the functionalities of these queues are explained in Section 2.1). 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.

5.1 Simulation Environment

Figure 9 summarizes the parameters of each system. 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. The testbench consists of 1-million memory instructions for each processor. We collect 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. The access latencies of the cache and the main memory shown in Figure 9 are the minimum times for the cache to service a processor request, and memory to service a cache's request respectively.

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 deferQ of size 4

Parameter	Design (1 / 2 / 3)
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
Non-Blocking Cache	No / Yes / Yes
Cache Latency	4 cycles
Cache Bandwidth	1 word per cycle
deferQ Size	0 / 4 / 16
cmissQ Size	1 / 1 / 4
c2mQ_L Size	4 / 4 / 16
c2mQ_H 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

5.2 Simulation Results

Figure 10 shows some of the statistics collected from the simulation. We can see that even though all systems have comparable miss rates, the systems with non-blocking caches have much lower CPI than the system with blocking caches, indicating that our non-blocking cache design are successful at partially hiding the long latency of cache misses.

Cache Type (deferQ / cmissQ / c2mQ_L)	% Misses	CPI (Improvement)
Blocking (0/1/4)	2.47%	2.35 (0%)
Non-blocking (4/1/4)	2.27%	1.68 (39.43%)
Non-blocking (16/4/16)	2.21%	1.40 (67.83%)

Figure 10: Miss Rate and CPI of Simultion Results

6 Conclusions

In this paper we first described a non-blocking MSI cache coherence protocol for a DSM system. The protocol is both relatively complex and realistic. We showed that it was straightforward to encode the protocol in BSV, as long as the implicit conditions of various method calls were complied using the "aggressive conditionals" in the BSV compiler. 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 have the hardware description match 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 difficulty in this work was guaranteeing fairness between rules. The type of strong fairness that is needed in this design must be programmed explicitly; simple "rule urgency annotations" fail to capture the specific requirements of this design. The correctness arguments for fair scheduling in non-deterministic settings 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.

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.

References

[1] Homayoon Akhiani, Damien Doligez, Paul Harter, Leslie Lamport, Mark Tuttle, and Yuan Yu. Cache-Coherence Verifi cation 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 and Computer Science Department, 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] F. Pong and M. Dubois. A New Approach for the Verification of Cache Coherence Protocols . *IEEE Transactions on Parallel and Distributed Systems*, August 1995.

[11] F. Pong and M. Dubois. Formal Verification of Delayed Consistency Protocols. In *Proceedings of the 10^{th} International Parallel Processing Symposium*, April 1996.

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

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

[14] 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.

[15] 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.

[16] 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.