A Backend for a Coherence Protocol Generator

Jiawei GU

Master of Science

School of Informatics

University of Edinburgh

2020

Abstract

This paper presents a backend for the conversion from Murphy to SLICC, which can take Stable State Protocol(SSP) to real instantiations of protocols. Reasoning cache coherence protocols can be complicated especially in multi-core systems. To overcome it, ProtoGen was presented to generate protocols automatically. To make it deployable, it is necessary to develop a backend to finish this task, which can output SLICC files for further simulation and research. In this paper, the characteristics of Murphy and SLICC are discussed including their difference and similarity. Based on the discussion, different conversion methods are presented for difference code blocks in the SLICC files. In addition, some potential problems during conversion and their solutions are mentioned as well. We also verified its correctness and simulated generated MSI protocol performance in gem5 simulator.

Acknowledgements

First of all, I would like to thank Dr.Vijay Nagarajan. Thank you for supervising this project, offering your expertise and guidance. Also, I would like to thank Theo and Nicolai who provide the inspiration for this project.

Secondly, I want to extend a sincere 'thank you' to all the close friends and helpful colleagues I have had the privilege of getting to know since coming to the UK.

Finally, I would like to thank my family and loved ones for their unending patience and support. Nothing that I have done in my life would I have been able to do without you.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Alongside technological development, multi-core processor devices have become quite common. Typically, a personal computer has at least four cores and latest mobile phone even has 8 cores[13]. Based on that, a multi-core system gives rise to cache coherence[1] problems. With regard to computer architecture, cache coherence denotes the uniformity of shared memory data that end up being stored in multiple local caches. As clients in a system maintain caches of a common memory resource, problems may arise due to incoherent data, which is particular in CPUs in a multiprocessing system. For example in the figure below[16], processor p1 updated its private cache data to x+1. However, this update is invisible for processor p3. Therefore, processor p3 will definitely read dirty data. Thus, directory-based cache coherence protocols[7] , e.g. MI and MSI, have been presented to resolve such problems.

However, designing those kinds of protocols is not easy and even complicated, because in order to level up processors' performance, it is typical to import transient states[6] based on stable protocols. This means that stable states coherence transactions are non atomic, and messages will be interleaved with others coming from other processors in such systems, which is similar to database transactions. Therefore, practitioners may not somehow be able to reason all possible states or correct orders, which will then cause some serious problems such as deadlock.

To overcome this, ProtoGen[11] was presented as a tool for generating cache coherence protocols with transient states when a stable state protocol (SSP) is passed into it. The ProtoGen output file is written in Murphy that is used for model check[3] in terms of safety and deadlock freedom. It is obvious that practitioners do not have

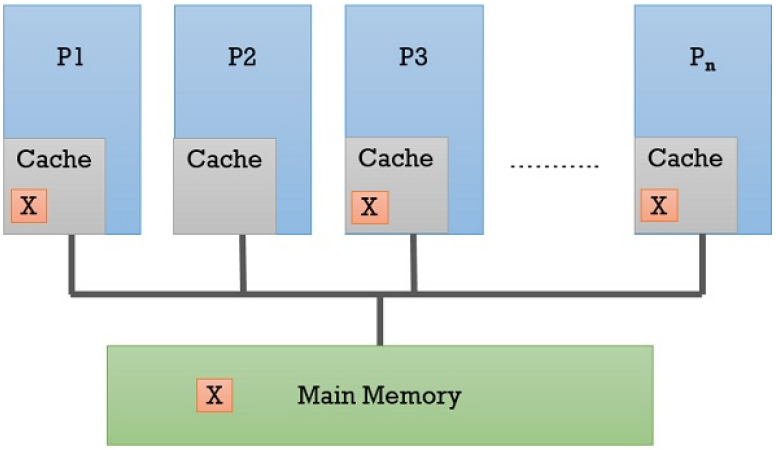

Figure 1.1: Cache coherence overview

to reason complicated protocols and can instead generate them automatically through using ProtoGen. Moreover, the performance of generated protocols is identical or even better than the one manually generated.

Thus, in order to make great use of ProtoGen and analyse the protocols, they shall serve as inputs into the gem5 simulator[2] that receives SLICC[8] files. A backend in this report is presented for this conversion, i.e. from Murphy to SLICC. Likewise, due to the differences between the two programming languages, the conversion is not just a "one-to-one" presentation. Therefore, strategies are going to be developed to finish the conversion task. The whole process flow is illustrated in Figure below.

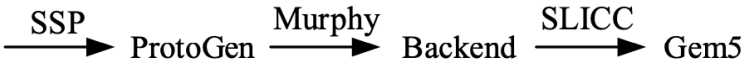

Figure 1.2: Pipeline of the whole process

1.2 Project Goals and Contributions

From the figure above, the backend in this report serves as a connecting link between the preceding and the following in the whole process. Actually, there is no existing work in the literature that takes in SSP state machine representation of protocol to real instantiations of protocols either in simulation or RTL. The backend is a necessary step to make ProtoGen deployable. To develop the backend, this report will address the following problems:

1. The difference between Murphy and SLICC in terms of structure and implementation principle.

2. According to the difference and analysis, specific strategies for the conversion and their influence.

3. Generated files correctness verification and testing.

1.3 Outline

Chapter 2 introduces the ProtoGen. The backend objects – Murphy and SLICC – are also discussed in detail to analyse the differences between them. For the next chapter, based on previous analysis and SLICC structure, the conversion rules for each part of the SLICC code blocks and their reasons are provided including in_port, transition, and action code blocks. The conversion methods of the cache controller and the directory controller are also discussed, including their similarities and differences. In Chapter 4, the generated SLICC files are connected for verification by gem5. After verification for correctness and equivalence of original source code, the protocol is tested through different tasks such as multi-threads and stack printing to validate whether the performance is better than that of a protocol generated manually. The final chapter provides the research summary, as well as some necessary points for future work.

Chapter 2

Background

In this chapter, we will provide the required knowledge of coherence in parallel architecture, Murphy, SLICC, and Ruby system which will provide the foundation of backend development. In first section, we will talk about ProtoGen briefly. The next coming 2 sections will illustrate Murphy, SLICC, and Ruby system in detail. In the last section, the difference will be discussed including their structure and way of implementation, which is the basis of the following work.

2.1 Coherence in Parallel System

As previously mentioned, nowadays the devices are multi-core systems, i.e. parallel system. To overcome the cache incoherence issues, various protocols were presented such as MI and MSI. Also, ProtoGen focuses on directory based protocols. Therefore, in this section, basic knowledge of cache controller and cache coherence protocol will be covered. There are 2 invariants[14] we have to enforce to maintain the coherence.

1. Single-Writer, Multiple Read(SWMR) Invariants. For any memory location A, at any given (logical) time, there exists only a single core that may write to A (and can also read it) or some number of cores that may only read A.

2. Data-Value Invariant. The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch.

To implement these invariants, we associate with each storage structure, i.e. a finite state machine called a coherence controller[14].The collection of these controllers consist of a distributed system where the controllers send messages with each other to

ensure that, for each block, the SWMR and data value invariants are maintained at all times. In the next coming figure, it illustrates the structure of cache controller and memory controller.

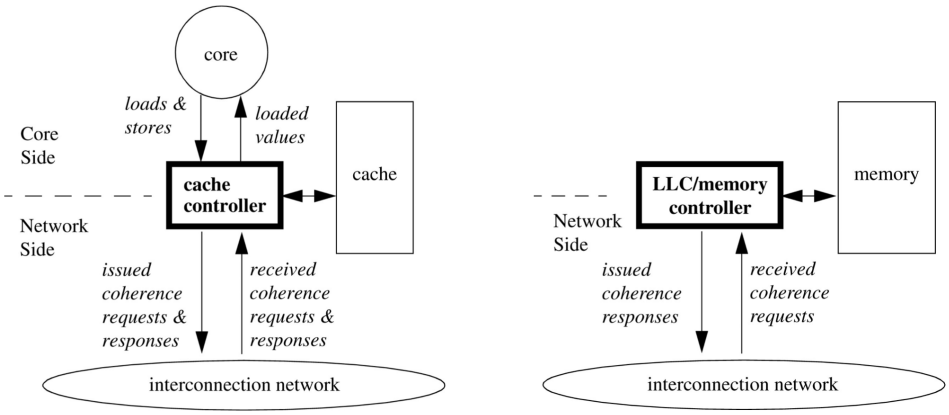

Figure 2.1: Structure of Cache Controller and Memory Controller

In this report, the memory controller is specified as the directory controller that has a list of data shares and list of data owner. Whatever it is described by Murphy or SLICC, the function and structure of cache and directory controller are the same. Also, the next discussion about controller is based on it.

After discussion of controllers' structure and function, the next element of coherence is cache coherence protocols. In the coming figure shows stable state MSI protocol[15][17].

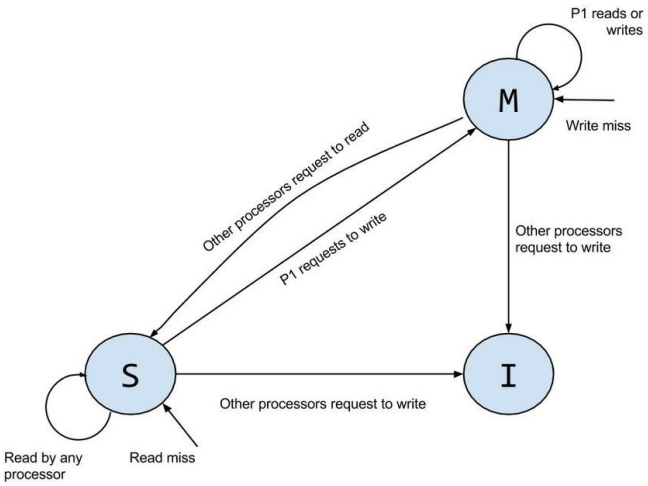

Figure 2.2: State Transition of MSI Protocol

The figure above shows the stable states transitions. However, the protocol has transient state to fit modern multi-core systems. Therefore, the MSI will be much more

complicated. The figure below shows the states transition including transient states and is generated by ProtoGen[10]. Due to its complexity, ProtoGen was presented that is

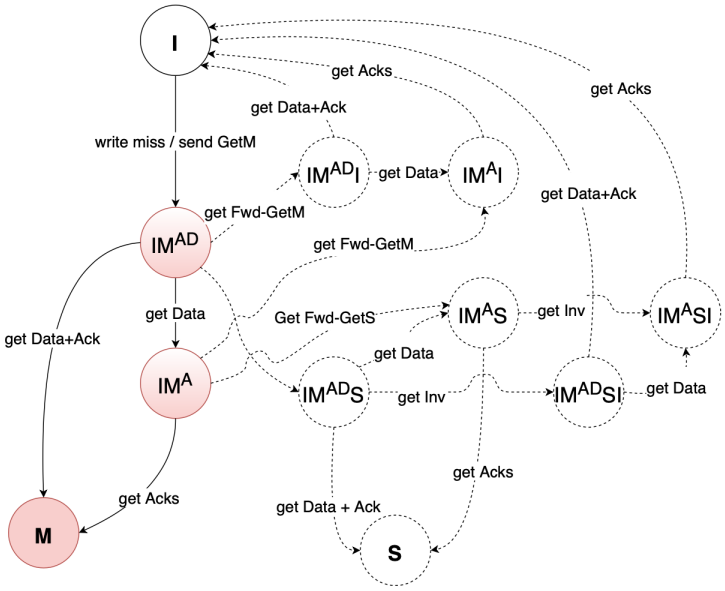

Figure 2.3: MSI Protocol Contains Transient States

the most advanced work towards automating the design of cache coherence protocols currently found in the literature. Its output file is the source code of the backend. Also, the discussion and example will bases on this generated MSI protocol.

2.2 Structure and Characteristic of Murphy

Given a stable state protocol specification, the ProtoGen can generate corresponding non-stalling protocol[14] by using transient states and a output file written in Murphy language(in the next part we will call it Murphy file). This output file is the source code of the conversion. Also a protocol can be regraded as state machines, based on this point, a Murphy file can be divided into sections as follows.

1. Basic Definitions of Variables and Functions. In this part, the Murphy defines basic elements that will be used during the state transition including the enumerations of cache and directory states, the message data structure, the entry of cache and directory, and message related functions such as multi-cast and message receiving. These variables and functions can be regarded as the basic components(attributes) of a state machine. Other operations are based on them.

2. Cache and Directory Transitions. In this part, it can also be divided into 2 parts, i.e. cache and directory part. Because the cache controller has to receive requests from CPUs, however these kind of messages do not belong to regular messages from other cache or directory controllers, we have to deal with it by using rule set. It is similar to handling ”cold start” problem. During this rule set section, for a given access(e.g. store and load), the cache controller will transition to different states and send corresponding(request) messages to directory controller. For example, when a cache block is in Invalid(I) state and there is a store access passed to it. The cache controller will switch to I_store state and then send a related(GetM) message. The meaning and abbreviation of messages are the same with primer book[14]. Despite rule set part, both cache and directory transitions are specified using corresponding functions. Both functions share the same structure, i.e. switch-case structure. For a given message and state, according to controller's condition, e.g. the number of acknowledge it has received, it will trigger different actions and switch to different states.

3. Machine Buffers. In order to communicate with other controllers, a cache/directory controller has to contains these buffers. Also, in order to avoid deadlock[19], we have to assign three channels, i.e. request, response, and forward. Because a message from a cache or directory controller can be divided into these categories, if there are three ways to transfer messages, controllers will not suffer from deadlock. This idea is also introduced in the next SLICC part.

Based on three parts, Murphy file finishes describing cache and directory state machines which contains lots of transient stats and shows better performance than conventional stable states cache coherence protocols. The Murphy file is generated from ProtoGen and used for model checking. Now from this backend's point of view, it is the input of the backend and converted into SLICC files.

2.3 SLICC and Ruby System

Ruby implements a detailed simulation model for the memory subsystem. It models inclusive/exclusive cache hierarchies with various replacement policies such as MRU and LRU, coherence protocol implementations, interconnection networks, memory controllers, various sequencers that initiate memory requests and handle responses. It shows configurability, which means that almost any aspect affecting the memory hi-

erarchy functionality and timing can be controlled, and rapid prototyping ability, i.e. a high-level specification language, SLICC, is used to specify functionality of various controllers. The Ruby system overview can be illustrated as follow[9].

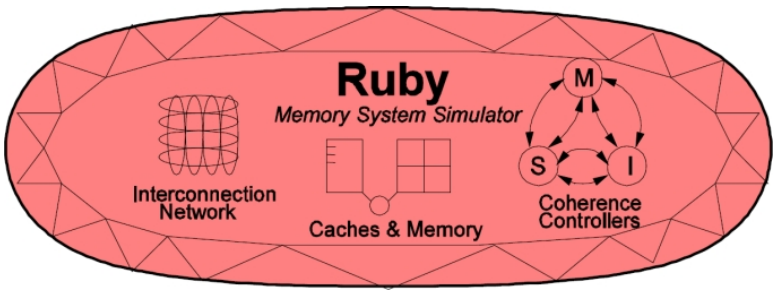

Figure 2.4: Ruby Overview

In addition, in Ruby system, the most of the work comes from editing SLICC files. As for SLICC, it is stands for Specification Language for implementing cache coherence and is a domain specific language(DSL[18]) that is used for specifying cache coherence protocols. Due to it, users do not have to pay too much attention to specific details of cache coherence protocols implementations such as the way of cache memory mapping. The purpose of SLICC is to describe the states transitions process and corresponding actions. In essence, a cache coherence protocol behaves like a state machine. SLICC is used for specifying the behavior of the state machine. This state machine can interact with Ruby system such as message sending through message buffers and fetch request from CPUs. SLICC also imposes constraints on the state machines that can be specified. For instance, SLICC can impose restrictions on the number of transitions that can take place in a single cycle. Apart from protocol specification, SLICC also combines together some of the components in the memory model. The overview of SLICC structure can be illustrated as follow[9]. For a canonical state

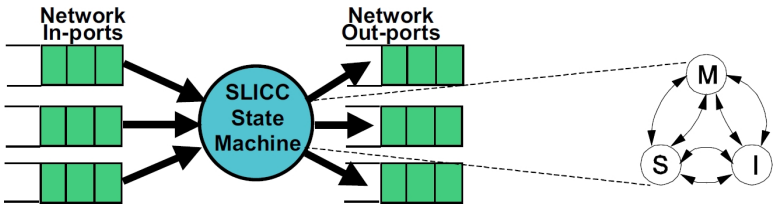

Figure 2.5: SLICC structure Overview

machine(cache or directory state machine), it contains roughly 4 parts which are listed below.

1. State Machine Declaration. In this section, state machine's basic components will be declared including state machine parameters, states and events declaration, user-defines structure, and other definitions. As for state machine parameters, in this subsection, these parameters are directly exported to the machine object attributes that is generated by the state machine.There are main 3 attributes(parameters) of the machine, i.e sequencer, cache memory, and message buffers. Each attribute serves its own function and is listed as below.

   (a) Sequencer. This is a special class that is implemented in Ruby to interface with the rest of gem5. The Sequencer is a gem5 object with a slave port so it can accept memory requests from other objects especially from CPUs and converts the gem5 the packet into a Ruby system request. Finally, the request is pushed onto the message buffer of the state machine. In another words, it is an interface of machine and system.

   (b) Cache/Directory Memory. It holds the date and provides basic functionality such as address looking up. During the state machine implementation, it provides functions for data operations.

   (c) Message Buffers. As figure 2.2 shows, message buffers are the interface between the state machine and the Ruby network. Messages are sent and received via the message buffers. As for a state machine, coming message can be a triggering signal for state transition. Also, in order to prevent deadlock, it is good to build virtual channels for message spreading. The structure of two networks is shown as follow.

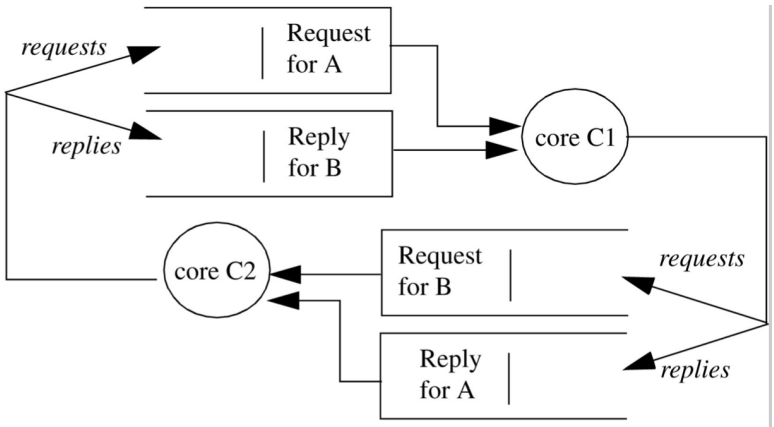

Figure 2.6: Two Virtual Networks

Through this way, on the one hand, the system can avoid deadlock, on the

other hand, it can avoid setting up physical channels. In addition, there is a special buffer for Ruby interaction. The packed messages from sequencer will finally be passed to this special buffer to trigger controllers' transitions.

2. In port code blocks. As for a state machine, with incoming messages and accesses, it should have input ports to handle the incoming information. Thus, these code blocks sever for it. For a given access or a message, the port will trigger different event. For example, when the coming message type is GetM type and it is from forward queue, the port will consequently trigger Forward GetM event(i.e. Fwd_GetM). For a given access, this kind of piece of information is passes through Ruby, thus in the mandatory queue will trigger related event for accesses such as cache load.

3. Actions code blocks. As we all know, for a protocol, during the state transition period, the state machine will take some actions to finish the transition such as sending messages, cache management, and queue management. Thus, these kind of tasks will be implemented in action code blocks.

4. State machine transitions. The main part of the protocol. In these code blocks, it will describe each transition for a given event. For example, if a cacheline is in Invalid(I) state and the coming event is store, it will transition to IS_D, which means from state I to state Share(S) and waiting for data. For this transition, the state machine will finish 4 related actions.

By using these structure, SLICC files describe the state machines i.e. cache and directory state machines. Also, from transition layer to basic configuration layer, the state machine description goes from abstraction to specification. The SLICC files are the output of the backend and will be run in the gem5 simulator.

2.4 Comparison of SLICC and Murphy

Based on the analysis above, both Murphy and SLICC describe the same finite state machines in different ways. Thus, it is essential to compare them to get the difference and summarize the similarity between the objects. From the high level point of view, SLICC is a subsystem of Ruby system, which has to interact with Ruby and call Ruby's interface during the transition. Also, it is a domain specific language, user do not have to concern about the implementation details. The Ruby system, or gem5 system, has

provides interface to use. Due to this, the description of state machine in SLICC files is more object oriented, thus it can be separated into 4 parts easily. Also, SLICC and Murphy look quite different unlike conversion from C++ to Java. The conversion is not a one to one presentation. To be specific, most difference can fall into these categories as follows.

1. Function Difference. As previously mentioned, SLICC does not show implementation details and Murphy shows. Thus, from this point of view, during the conversion, it is important to know the purpose of functions in Murphy and get the corresponding functions. For example, in Murphy, in order to send messages to other controllers, multi-cast is used that is implemented by using loop. However, in SLICC, it does support loop. Thus, we can assign all the data sharers to the message destination and make the output message to enter the queue. Murphy is flexible, SLICC sacrifice the flexibility to get its convenience.

2. Structure Difference. As previously mentioned, a state machine in SLICC file must contain 4 parts. Thus, it is important to parse and categorize the code blocks in Murphy. Because most of the state transitions are finished within switch-case block, we have to parse the file to get the related information for each part.

3. Parameters Difference. As previously mentioned, a cache/directory state machine has parameters that play important roles during transitions such as cache entry allocation that are not shown in Murphy file and common in SLICC files. This difference is also caused the SLICC and Ruby system inclusion relation. During the conversion, we have to add extra codes to meet the requirements.

4. Message Difference. As previously mentioned, both SLICC and Murphy have three channels for communication and avoiding deadlock. However, there is still difference for the two systems. In Murphy, all the messages are processed in one function and sent to different channels. On the other hand, in SLICC system, each message has its own channel and is processed in each own port. This difference is shown in code, i.e. in SLICC, there are different input ports(in port code blocks), to process their own messages. In SLICC, following this view, we have to figure the message source buffer out and put them into correct buffers.

5. SLICC is more objected oriented files than Murphy file, which means that it has clear and obvious layers such as transitions code blocks, action code blocks, and

message buffers. Each field of the state machine has its own role during the transition.

6. Action Difference. In SLICC, there are explicitly action code blocks which are triggered events. However, in Murphy, there is no concept of event or actions. Thus, apart from figuring messages out, we also have to figure events out. Also, because there is no event in Murphy files, it can be difficult to get the useful information to generate events.

7. The grammars of two programming languages are different. For Murphy language, all the transient state transitions are included in one function with given messages and cache/directory objects. However, in SLICC, the whole process are divided into several parts and each transition is written in on code block.Thus, when parsing the Murphy file, we should distinguish which parts of codes belong to corresponding parts in SLICC files.

Chapter 3

Methodology

Having introduced the characteristics of Murphy and SLICC and difference analysis, it is time to demonstrate the methodology of conversion based on previous discussion. Due to the property of SLICC, different strategies of conversion will be applied for different code blocks.

3.1 Overview

As previous mentioned, both Murphy file and SLICC files describe the same state machine for the cache/directory controllers. It is used to describe the behavior of the object in its life cycle. In computer science, finite state machines are widely used in modeling application behavior, hardware circuit system design, software engineering, compiler, network protocol, and computing and language research. For example, the well-known TCP protocol[12] can be described by using state machine. A classical finite state machine is described by a five-element tuple:

$$M = \{Q, \Sigma, \delta, q_0, F\} \tag{3.1}$$

In this equation, Q means a finite set of states, Σ refers to a finite and nonempty input alphabet, δ refers to a series of transition functions, i.e. $\delta(Q \times \Sigma) \in Q$, q_0 means start state, i.e $q_0 \in Q$, F means the set of final states $F \in Q$. The figure 2.2 and 2.3 have shown states transitions of a cache coherence protocol. Based on that, it is safe to say that the first thing we can do is building a medium to implement the conversion[4], i.e. finite state machines one for cache controller and one for directory state machine. The state machine can be regarded as inter-conversion between two programming languages. In addition, according to knowledge of cache coherence, the transition func-

Table 3.1: Examples of Cache Transition Table

start state	final state	input	output	condition	action
I	I_store	store	GetM		
SM_AD	M	GetM_Ack_AD		ack equals	ack operation
SM_AD	S_store_GetM_Ack_AD	GetM_Ack_AD		ack not equals	ack operation

tion is not complicated, which means given a start state, an input, and controller's condition, the final state and related actions are determined. Therefore, an abstract state machine can be demonstrated by using transition tables that is shown above. The collection of all start states and final states consists of Q, the way of mapping between final states and condition under given start state consists of transition functions, i.e. δ. From high level of point, by traversing the tables, we can get the output files i.e. SLICC files. In Ruby system, there are three SLICC files, cache, directory, and messages. Message file is used for describing messages during the transitions. The other two files share similar structure. Therefore, in the next part, the demonstration of conversion will be based on cache controller. Also, cache and directory are still slightly different. In the last section, the conversion of directory will be discussed. The schema of the transition table and examples are shown as follow. The meaning of the messages goes:

1. GetS. Cache sends message to directory controller means that cache issues a request and intends to transition to state S.

2. GetM_Ack_AD. Directory sends message to cache controllers indicating that the cache can continue the transition to M if the number of received acknowledgement equals the number in the message, if not, it will switch to another transient state.

Noticing, some fields are not just string variables. Each field has its own data structure for recording transitions. As for a message, it should contain source address, destination address, transferred data, the number of expected acknowledgement, messages content, and its channel. Also one controller can only process one message at the same time, however, it can output more than one messages. Therefore, for output ones, the data structure is a list of messages based on previous analysis. Fortunately, first 5 fields in transition table can be acquired through ProtoGen by extending its source code easily, because ProtoGen also exploits the table to generates Murphy files. Therefore, we also have to parse the Murphy file to get the related actions. In addition, as for a

controller, actions can fall into 3 categories, i.e. (1) acknowledgement operations, (2) data blocks assignments, and (3) message deferring. The main structure of the Murphy file is switch-case based, which means that it is not difficult for us to parse it and acquire related actions. The algorithm of getting the cache transition table is presented as follow.

Algorithm 1 Getting Transition Table Procedure

```
1:  //extend from ProtoGen
2:  Murphy_table = [start_state, final_state, input, output, condition, action]
3:  pointer = first line of Murphy file
4:  while pointer is not null do
5:      if (*pointer).contains (Fun_Cache) then
6:          for each start_state case block do
7:              for each input case block do
8:                  Murphy_table.get("action").put("ack", ack);
9:                  Murphy_table.get("action").put("data_blocks", data_blocks);
10:                 Murphy_table.get("action").put("msg_deferring", msg_deferring);
11:             end for
12:         end for
13:     else
14:         pointer++;
15:     end if
16: end while
```

As previous analysis, SLICC file has 4 parts to describe the state machine, however, Murphy does not. If we strictly follow the trace of SLICC templates and get the target pieces of code from top layer to bottom layer(from transition code blocks to in port code blocks), it will first break Murphy structure, which means the generated SLICC file will look much different from Murphy. More importantly, we have to finish SLICC events and Murphy messages mapping, which is indirect. In Ruby system, a controller is driven by events that are registered at the initialization stage of the state machine, however, for Murphy, the triggering signals of states transitions are messages. Therefore, one to one presentation is not a good idea for this backend. This is because for a given state and coming message, controller's internal attributes will also affect the final state and output messages. For examples, in Table 3.1, transition 2 and 3 share the same start state and input message, but their final states vary due to the number

of acknowledgements. More importantly, transition code blocks do not support if-else statement, which means we cannot transition to 2 different states based one state and one event.

Based on the analysis above, and according to the characteristics of SLICC and Ruby system, it is a good way to put most of the transitions in "in_port" code blocks. In this way, the style of SLICC and Murphy will be much similar. Moreover, the transition in SLICC transition code block is atomic and a controller receives one message at the same time to handle. Therefore, it is also atomic if we finish transition in "in_port" code block. Following this direction, in the coming sections, different code blocks conversion strategies will be demonstrated and discussed in detail.

3.2 Events Determination

According to previous discussion, the state machine in Ruby system is driven by its events. Therefore, it is a good idea to deal with events conversion firstly. We cannot directly map messages in Murphy into events in SLICC. Because for a given message and start state, according to different conditions, controller will switch to different states. Also an event means that start state and final state are determined and not ambiguous at all. Messages cannot do that. In addition, a state machine runs in Ruby system and has to manage its cache entries and deals with its channels such as cache entry and transaction buffer entry(TBE) allocation. These kind of operations have to be done by system functions that belong to Ruby interface. In another words, we cannot handle them only in one code blocks. Thus, resource allocation and release have to be regarded as events. From this point of view, every cache/directory controller must have these 2 types of events, i.e. resource allocation and resource release. Also, according to the knowledge of cache coherence protocols, the resource allocation and release only happens when the controller's state is Invalid. To be specific, if a controller's start state is Invalid(I), it has to allocate related resource such as transaction buffer entry(TBE) and cache entry, vice versa for resource release. These events are determined regardless types of cache coherence protocols, i.e. every protocol has theses events. In this scenario, the protocol has transient state, which means that if a cache is in Shared(S) state and there is a store access, the cache will not occur a store miss directly, but transitions to a transient state, e.g. SM_AD, and sends related messages.

Based on the analysis above, a protocol's events in Ruby system can be : "deallocate request queue", "deallocate response queue", "deallocate forward queue", "allo-

cation of I load", and "allocation of I store". The reason of different message buffers is to avoid deadlock which will be illustrated in detail in the next section.

3.3 In_Port Code Blocks Conversion

In this section, the "in_port" code blocks conversion methods are going to be discussed in detail. It is the most important one for the backend, because most of the transitions happen this part. As previously mentioned, there are 4 channels, i.e three channels for communicating with other controllers and the rest one for connecting with CPUs that is mandatory. Therefore, according to the types of message buffers, the conversion methods will be demonstrated in 2 parts, i.e. communication ones and mandatory buffer.

3.3.1 Communication Buffers Conversion

Communication buffers are used for messages sending with other controllers including cache controllers and directory controller, which are shown in figure 2.5 and 2.6. Different types of messages will be sent to its own channel. For example, when a cache controller is going to send a request message to directory. It will push this output to controller's request buffer. The directory controller will receive this message in the same channel later on. In another word, the controller has to distinguish messages' types and put them in the correct buffers and a controller has different channels(buffers) for different types messages as well. However, this point is not shown and stressed in Murphy. In Murphy, all the coming messages are processed in one function named "Func_cache" and CPUs' requests are processed in rule set part. Therefore, in order to overcome the difference, we can assign 3 "in port" code blocks in SLICC, i.e. forward buffer, request buffer, and response buffer, because a message will definitely fall into one of these categories. In this subsection, the three buffer conversion method is going to be discussed.

In each of three buffers, e.g. forward buffer, the cache controller does the same thing with Murphy. Also SLICC does not support switch-case, during the conversion, if-else statements will be used. The example in forward buffer is shown below.

Listing 3.1: Snippets from Murphy

```
case cache_SI_A:
switch inmsg.mtype
```

```
        case Inv:
            msg := Resp(adr,Inv_Ack,m,inmsg.src,cle.cl);
            Send_resp(msg);
            cle.State := cache_II_A;
            cle.Perm := none;
         case Put_Ack:
            cle.State := cache_I;
            cle.Perm := none;
        else return false;
    endswitch;
```

Listing 3.2: Snippets from SLICC

```
    else if (st == State:SI_A) {
    if (in_msg.Type == CoherenceMessageType:Put_Ack) {
        setState(tbe,entry,LineAddress,State:I);
        assert(is_valid(entry));
        trigger(Event:deallocfwdfrom_in,address,entry,tbe);
    }
    else if (in_msg.Type == CoherenceMessageType:Inv) {
        enqueue(respto_out, CoherenceMessage, 1) {
        out_msg.LineAddress := address;
        out_msg.MessageSize := MessageSizeType:Data;
        out_msg.Destination.add(in_msg.Sender);
        out_msg.Type := CoherenceMessageType:Inv_Ack;
        out_msg.Sender := machineID;
        out_msg.cl := entry.clL1;
        }
        setState(tbe,entry,address,State:II_A);
        fwdfrom_in.dequeue(clockEdge());
    }
    else {
        //stall
    }
    }
```

From 2 lists, when the cache controller's current state is "SI_A", there are two

options. These pieces of code is representative. If it receives ”Inv_Ack”, the cache state will transition to ”II_A” and send output message. If the message is ”Put_Ack”, it will transition to Invalid. It is obvious that both two lists describe the same transitions. However, there are still some difference.

1. SLICC has to manage messages. In this example, during the transition, the controller has to send the output messages to response buffer. Also, when the transition finishes, the controller has to leave current buffer, i.e. forward buffer.

2. The output messages are sent by pushing them into the corresponding buffer, which can be regarded as queue as well. However, in Murphy, it is implemented by its function.

3. Whatever the final state is, all Murphy needs to do is assignment. On the contrary, as previously mentioned, when the final state is Invalid, it has to trigger an event to finish the transition. If the final state is not Invalid, it can set the state by using SLICC interface function.

The rule of in port code block conversion can be summarized as follows.

1. Message Handling. SLICC has to push messages into related queues. For each output message, as previously mentioned, message data structure has its channel and destination. Therefore, it is easy for a controller to choose correct channel and finish the message assignment. Also, under different conditions, the ways of handling are slightly different.

   (a) Message Sending. It is a normal case of handling. The controller has to choose the corresponding channel of the messages and push them to it.

   (b) Message Deferring. In Murphy, a cache may defer some messages when it receives forwarded messages. Therefore, there is a queue data structure for deferring. However, SLICC/Ruby does not have such flexibility. The way of deferring is to not popping the message immediately after finishing state conversion. Therefore, when the controller receives other messages and is going to send messages, it can first peek buffer queue to get the deferred ones.

2. State Assignment. It still has some three options in SLICC.

(a) Final State is Invalid. As previously mentioned, Invalid state will trigger event. Therefore, if a controller's final state is Invalid. Is has to trigger related event to handle it.

(b) Transition Needs Cache Entry. This scenario happens cache hitting. For example, if a cache controller's start state is "I_store" and it receives message "GetM_Ack_AD". It will switch to stable state M. In this scenario, the controller has to set cache entry. What is more, SLICC is a subsystem of Ruby system, it has to tell the CPU(Ruby calls it sequencer) it finishes. Thus, sequencer function is called.

(c) Others. In this scenario, controller's state switches from one to another by using interface function "setSatte". The example is shown in list3.2.

3. Leaving Message Buffer. As it is shown in list 3.2, after finishing the transition, the controller has to leave message buffer. If it does not do that, this port, e.g. forward buffer, cannot receive other information at all and the controller will stop.

Following these rules, the transition can be converted from Murphy into SLICC. Also, the example is illustrated by using forward buffer. Actually, the other 2 buffer does the same thing. The only difference is the buffer. However, we do not have to concern about the messages conflict. When pushing output messages into buffers, the controller chooses which one is correct. Thus, for instance a response message will never appear in other buffers. But in order to get the similar format of Murphy, in each buffer, the controller will make the judgement.

3.3.2 Mandatory Buffer Conversion

Mandatory buffer is a special one. As for a system, the initial message comes from CPU, therefore, it is mandatory. Also, according to its characteristics, this buffer only offers Ruby request messages, which means the controller's state is stable state. Actually, when a controller is in transient state, it cannot receive access from CPUs. This buffer corresponds to rule set in Murphy. According to the types of Ruby requests, there are 3 types of requests. Also, based on these requests, the conversion is finished.

1. Evict Request. Only when the controller's state is Modified or Shared, it can process them. Also, evicting a cache line into memory needs to communicate

with directory controller. Therefore, the output message's destination is directory. Also, due to cache line hitting, the controller has to response to sequencer as well. Thus, an example of conversion is shown below.

Listing 3.3: Example of Eviction in Murphy

```
rule "cache_M_evict"
cle.State = cache_M
==>
SEND_cache_M_evict(adr, m);
endrule;
```

Listing 3.4: Example of Eviction in SLICC

```
if (st == State:M) {
enqueue(reqto_out,CoherenceMessage,responseLatency){
out_msg.LineAddress := LineAddress;
out_msg.MessageSize := MessageSizeType:Data;
out_msg.Destination.add(mapAddressToMachine(LineAddress,
MachineType:Directory));
out_msg.Type := CoherenceMessageType:PutM;
out_msg.Sender := machineID;
out_msg.cl := entry.clL1;
}
setState(tbe, entry, LineAddress, State:M_evict);
sequencer.evictionCallback(LineAddress);
}
```

Being similar with communication buffers, for CPU access, Murphy also finishes them by using functions. Also, SLICC finishes them in the same way.

2. Load Request. Being different with eviction request, the controller is in any of stable states can process it. More important, when the state is Invalid, the controller has to trigger an event to allocate resources.

3. Store Request. Store request is similar to load request. The controller can response to this request regardless its state. However, for store request, if the controller's state is Shared, it will transition to a transient state rather a cache hitting.

3.3.3 Deadlock Avoidance

During the conversion, a essential issue is deadlock avoidance. Deadlock can be regarded as resource cycle dependency. The figure below shows this relation.

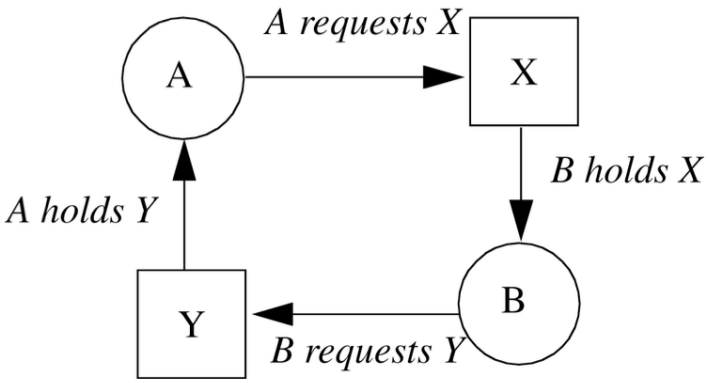

Figure 3.1: Deadlock Demonstration

Assume A holds resource X and B holds resource Y. If A requests Y and B requests X, then unless one node relinquishes the resource it already holds, these two nodes will deadlock. There are three types of deadlock in general, i.e. protocol deadlock, cache resource deadlock, and protocol-dependent network deadlock. Also, the Murphy file has passed Murphy model check and there are three types of messages buffers for messages sending. Therefore, possible deadlock may come from cache resource deadlock. Cache resource deadlock arises when a cache controller must allocate a resource before performing some actions. From this point of view, during the conversion, it is important to first perform actions and then allocate resources to avoid deadlocks. For example, during one transition, the controller has to first send output messages and then do other actions related resource allocations. If it is not done in correct order, the system tends to suffer deadlocks.

3.4 Transition and Actions Code Blocks Conversion

According to SLICC, a triggered transition consists of several actions. For example, transition(I, allocate_load, I_load) has three actions, i.e. entry allocation, action of I to I_load, and leaving mandatory buffer. Thus, in this section, both 2 code blocks are going to be discussed.

3.4.1 Transition Conversion

Based on previous analysis, one triggered event causes one transition. As for Ruby system, events are required when the controller needs to allocate or release resources. Thus, there are 2 types transitions corresponding to events, i.e. resource allocation and release. Also, for each transition, we have to determine start state and final state. If a controller's final state is Invalid, we have set the final state in "in_port" code blocks. Thus, what we have to do is releasing related resources. As for the other option, start state is Invalid, it will switch to another state according to transition table, we can get its final state and output messages easily. For example, the first transition in table 3.1, when a controller receives a "store" access and its state is Invalid, it will switch to "I_store" state and send request messages to directory controller.

3.4.2 Action Conversion

After determination of transition code blocks. It is time to finish action code blocks. Based on previous analysis, controllers' actions are resources allocation and release and output messages sending, e.g. the first transition in table 3.1. Thus, it has to make use of Ruby interface to finish message sending and resource management.The SLICC working flow can be demonstrated as follow.

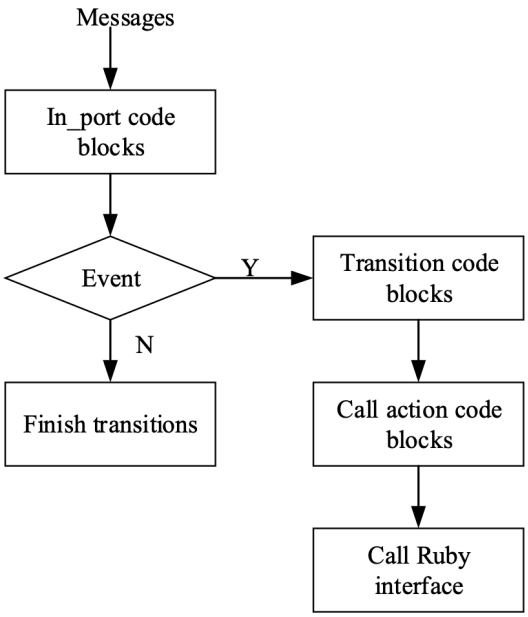

Figure 3.2: SLICC Working Flow

3.5 Directory Conversion

As previously mentioned, the whole SLICC system contains cache controllers and directory controller. Even though the structure of directory controller is similar to cache one. There are still some difference between them.

1. Sequencer. As previously mentioned, a cache controller has to receive Ruby requests through sequencer. However, as for a directory controller, it is not necessary. We can assume that that the directory state and memory data is stored together in main-memory to simplify the protocol. Thus, in this scenario, we do not have to consider about communication with Ruby system. In addition, setting sequencer works as well.

2. Controller entry structure. According to the knowledge of directory based cache coherence protocols, a directory controller has cache owner and sharers. Thus, during the implementation, we have to add these structure to the controller entry to meet the requirement.

3. Multicast messages. Based on (2) illustration, a directory has sharers structure. Thus, it will multicast messages during the transitions. Being different from a cache controller, when a directory controller is going to forward requests, the directory controller will direct the request to all the sharers. The way of multicast implementation is different in Murphy and SLICC. This constraint comes from Ruby interface.

The other parts of a directory controller is similar to a cache controller. Thus, we can keep following the rules.

3.6 Other Related Work

From the description above, it seems that the conversion is somehow straightforward, however, it needs the effort required significant engineering effort and debugging. In this section, we will introduce common potential problems during the conversion and corresponding solutions.

1. Debugging Inconvenience. As we all know, the Ruby is event-driven system. However, we do not follow the trace and finish most of the transitions in ”in_port” code blocks. Therefore, during the debugging process, we have to add extra

functions, e.g. ”DPRINTF” and ”APPEND_TRANSITION_COMMENT”, to print useful information to keep track of transitions.

2. Invalid Transition. This is triggered by invalid transition. Based on trace function mentioned in (1), we can locate what is wrong with the transition. Maybe wrong output messages are sent or incorrect state is set.

3. Message Buffer Dequeue Issue. It is triggered by popping message buffers incorrectly. For example, if we pop forward message buffer when it is transitioning from I to IS_D. Therefore, we should double check related buffers, especially whether the deferred messages are handled correctly.

4. Entry Attributes Management. It is caused by the difference between Murphy and SLICC. In Murphy, the data owner in the directory controller is assigned directly. However, in SLICC, the data structure is similar to queue. Therefore, in order to ensure that there is only one owner, we have to first clear the queue and then add the owner to the queue. This limitation comes from Ruby interface that we have to obey. Also, sharer is also a queue data structure, we have to deal with it in the same way. If not, the program will shut down due to the assertion.

Chapter 4

Evaluation

To experimentally evaluate the backend, it is used to generate several different protocols with different features. Unlike traditional compiler evaluations that seek to show improvements in compiling and running speed, etc. This evaluation is going to verify backend's correctness and explore generated protocols' performance, which means that ProtoGen can successfully generate protocols that are identical and, in some cases, arguably superior—to existing protocols.

4.1 Correctness Verification

4.1.1 Semantics Perseverance

The backend finishes the conversion from Murphy to SLICC. Therefore, to some extent, it can be regarded as a compiler. The goal of compiler verification is to prove that a compiler preserves key properties of the program it is transforming[5]. In specific, preserving the semantics, or behavior of a program.

Both two languages share the same transitions table, if the table contains all elements of a state machine, then its semantics is persevered. From the schema that is shown in table 3.1, the set of start states and final state is finite states set. All the inputs consist input alphabet. Also, each transition in the table is grouped by the start states, which means elements q_0 and F are persevered. As for transition function, the table contains each transition's output messages and related actions, thus, it is also persevered. Based on that, both of 2 languages share the same behavior.

4.1.2 Simulator Testing

Even though semantics is persevered, due to SLICC and Ruby unique properties, we should also test its correctness by using test files. The direct way is running the generated SLICC files in gem5 simulator. In this subsection, MI and MSI protocols are going to be tested.

In order to run generated SLICC files in Ruby(gem5) system smoothly, we have to configure related scripts to connect our state machines and system[8]. There are 2 scripts for the connection, i.e. one for describing controllers including caches and directory and one for the whole machine. In the first script, we assign the number of CPUs that is the same with it is in SSP, the names of each message buffers in caches controllers and directory controller, the order attributes of message buffer. Also, based on the previous analysis, the number of virtual networks is a constant, i.e.3. As for the other script, it connects virtual controllers with the system and call system function to start simulation.

After finishing all the steps, the gem5 simulator will first compile the state machine into binaries. To test its correctness, we use random tester to do the job that issues semi-random requests into the Ruby system and checks to make sure the returned data is correct. In this scenario, the generated SLICC files have passes compiling and tests. Thus, in terms of this, the correctness is verified.

4.2 Latency Statistics and Analysis of Generated Protocols

In this section, the performance of generated protocol will be tested by using different tasks. According to the knowledge of ProtoGen, the generated protocols' performance is identical or better than the ones generated manually. Thus, in this section, the generated non-stalling MSI protocol is going to be tested with 2 tasks, i.e. stack printing and multi-threads. According to ProtoGen, non-stalling MSI protocol is based on stable MSI protocol, adds extra transient states compared with Sorin's. It is more aggressive and shows more concurrency.

The experiments were conducted in my own PC that has 4 CPUs and uses Linux CentOS. The simulated environment is X86 architecture, "syscall emulation"(SE)mode, 3 CPUs with 1 level private cache for each, and 1 directory controller. The baseline of Protocol is an MSI protocol which has three stable states and 8 transient states and is

Table 4.1: Multi-threads Statistics for Two Protocols

Latency	ProtoGen	MSI Non-stalling
Fetch-hit	1607905	1858300
Fetch-miss	3622	4187
Load-hit	294135	339936
Load-miss	1567	14764
Store-hit	12770	136976
Store-miss	2360	2719

Table 4.2: Stack Printing Statistics for Two Protocols

Latency	ProtoGen	MSI Non-stalling
Fetch-hit	180882	206321
Fetch-miss	1088	1389
Load-hit	28869	34643
Load-miss	1567	1780
Store-hit	17308	19038
Store-miss	602	659

generated manually. The target one is the protocol that is generated by ProtoGen and is converted into SLICC, which has 3 stable states and 17 transient states. The results of the simulation are shown as follows. The latency of memory and response in this experiment is 1 for simplicity.

When a processor is going to do some operations, it will come cross data hitting or missing. Therefore, the total latency will be demonstrated in this way. Also, due to the complexity of multi-threads, the total latency is higher than others. From these tables, it is safe to say that the average latency of ProtoGen is less than normal one. Based on knowledge of ProtoGen, it adds extra transient states to decrease the latency. For example, when the cache state is IM_AD, which means it is going to transition from Invalid to Modified ans wait for acknowledge and data[14], and receives a forward message, it will stall. However, for ProtoGen in this scenario, it will change to IM_AD_S. Therefore, it can respond to the input message and change its state immediately. In this way, the latency decreases. Actually, when the cache state is IM_AD and SM_AD, for normal non-stalling one, the cache will occur stall. However, for the one from ProtoGen, it will transition to other transient states to decrease stalling.

Chapter 5

Conclusions

This chapter presents the research summary and provides directions for future work relative to the topic.

5.1 Summary

In this paper, a backend is presented for converting programming languages, i.e. from Murphy to SLICC, based on the finite state machine theory. It can take SSP state machine representation of protocol to real instantiations of protocols either in simulation or RTL, which is meaningful for productive practice.

Chapter 2 introduced basic elements of cache coherence, the objects of conversion, i.e. Murphy and SLICC, and their own characteristics. Also, the difference such as structure and the way of description a protocol between them is analysed for the conversion.

Chapter 3 discussed the conversion methods based on each code blocks and previous analysis. Being different from traditional way, most of the transitions are put in ”in_port” code blocks for a better conversion from Murphy. Also, in the end of this chapter, the potential problems and corresponding solutions are presented.

Chapter 4 verified the correctness of the generated protocol including semantics perseverance. Also, it is compared with the non stalling MSI protocol in terms of latency and we discusses their difference.

5.2 Future Work

For this backend, it finishes the conversion from Murphy to SLICC. However, there are still some work to be done in the future.

1. Protocol Diversity. Whilst the current backend has finished the conversion of MI and MSI protocols, there are still many other protocols that need to be converted, such as MOSI and MESI. These protocols can also be further explored, such as adding more transient states to improve performance.

2. SLICC Debugging Convenience. As mentioned, most of the transitions occur in "in_port" code blocks. However, gem5 or Ruby is an event- triggering system. In this case, during debugging, most of the information comes from the transition code blocks, i.e. event information, and some from action code blocks, i.e. related actions during the events. However, for the backend, the debugging information is limited. To overcome this, extra print functions are added in the "in port" code blocks. Whilst this is not that of a big deal for an experienced user, it is a bit unfriendly for users who are relatively new to SLICC and gem5.

3. Redundancy Reduction. As mentioned, "in port" code blocks have covered most of the transitions. In order to avoid deadlocks, three channels are assigned to a controller. One issue is that these three code blocks look almost the same. As such, a channel will cover all messages regardless of their attributes. Therefore, in the future, the backend should distinguish the messages for each port, although the generated SLICC files may be different from the Murphy one. In this way, the transitional and current ways become balanced, which is also friendly for the readers.

Bibliography

[1] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems (TOCS)*, 4(4):273–298, 1986.

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[3] David L Dill. The mur ϕ verification system. In *International Conference on Computer Aided Verification*, pages 390–393. Springer, 1996.

[4] Dony George, Priyanka Girase, Mahesh Gupta, Prachi Gupta, and Aakanksha Sharma. Programming language inter-conversion. *International Journal of Computer Applications*, 1(20):68–74, 2010.

[5] R Gerber and S Hong. *Semantics-based compiler transformations for enhanced schedulability*. University of Maryland at College Park, 1993.

[6] Erik Hagersten, Anders Landin, and Seif Haridi. Multiprocessor consistency and synchronization through transient cache states. In *Scalable Shared Memory Multiprocessors*, pages 193–205. Springer, 1992.

[7] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. *ACM SIGARCH Computer Architecture News*, 18(2SI):148–159, 1990.

[8] Jason Lowe-Power. Slicc. `https://www.gem5.org/documentation/general_docs/ruby/slicc/`.

[9] m5sim. High level components of ruby. `http://www.m5sim.org/Ruby`.

[10] Vijay Nagarajan. Cache coherence protocols are notoriously easy prof. https://www.youtube.com/watch?v=ZfMyDDsN5u4.

[11] Nicolai Oswald, Vijay Nagarajan, and Daniel J Sorin. Protogen: automatically generating directory cache coherence protocols from atomic specifications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 247–260. IEEE, 2018.

[12] Sushant Rewaskar, Jasleen Kaur, and F. Donelson Smith. A passive state-machine approach for accurate analysis of tcp out-of-sequence segments. *Acm Sigcomm Computer Communication Review*, 36(3):51–64, 2006.

[13] Samsung. Take the latest galaxy phones for a virtual test drive. https://www.samsung.com/hk_en/smartphones/galaxy-s20/models/.

[14] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis lectures on computer architecture*, 6(3):1–212, 2011.

[15] Pradeep Subedi and Wei Zhang. Wcet estimation of multi-core processors with the msi cache coherency protocol. *Proc. Work-in-Progress Session LCTES*, pages 17–20, 2012.

[16] Neha T. What is cache coherence problem? https://binaryterms.com/cache-coherence.html.

[17] Andrei Terechko. Msi-protocol-state-transitions. https://www.researchgate.net/figure/MSI-protocol-state-transitions_fig4_226344933.

[18] Arie Van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of computing and information technology*, 10(1):1–17, 2002.

[19] Stephen R VanDoren, Madhumitra Sharma, and Simon C Steely. Employing multiple channels for deadlock avoidance in a cache coherency protocol, January 11 2000. US Patent 6,014,690.