Design and Verification of a Multichip Coherence Protocol

Shahid Ikram, Isam Akkawi, Richard Kessler, Jim Ellis, David Asher

Cavium, Inc., 600 Nickerson Road, Marlborough, MA 01752

***Abstract*- This paper describes an industrial experience of the development of a multichip coherence interconnect protocol (OCI). The protocol specification was formulated as extended state transition tables. We used formal as well as simulation tools in different protocol development's phases like discovery, design, modeling, validation and maintenance. A number of design bugs as well as holes (missing features) were found in these phases. Furthermore, the protocol models were used to validate micro-architectural protocols and constraints. The tools and processes developed were employed in the design of two major multi-billion transistors chips and helped us in achieving a quicker convergence of verification work by saving many months of effort.**

Categories and Subject Descriptors
C.2.2 [**Network Protocol**]: Protocol verification; B.3.2 [**Memory Structures**]: Design Aids—*Formal verification*; B.5.3 [**RTL Implementation**]: Design Aids—*Verification*

General Terms
Functional coverage, Verification

Keywords
Cache Coherence, Protocols modeling and Formal verification

I. INTRODUCTION

IN this paper, we share our experience in the design, modeling and verification of an industrial multichip cache coherence protocol. It was an interesting and challenging endeavor, as we started with the inception of the protocol and took it all the way to chip tape-out. RTL simulations are the main vehicle for validation of nearly all the chip designs and same was the case in our group. However, a new multichip protocol was too big a challenge to leave to simulation only.

Formal methods have shown promising results in protocol verification [1][2][3][5][6][7][10]. The hope was any effort in addition to simulations will increase the confidence in the protocol design. We decided to employ formal verification with the following key objectives:

1. To create a robust set of high-level specifications, that can be directly used in different parts of chip design.
2. To help the verification team reach its tape-out goals in time.

The challenges to meet these objectives were:

1. The tape-out had firm time constraints (let us say six months).
2. Resources are limited in terms of man-power, software licenses and compute power.
3. Robustness requires formal proofs of the protocol but design is too large (4-nodes, more than 4000 transitions) for formal proofs with the given resources in the given time constraints.

We developed an approach that was a compromise between simulation-only and formal-verification-only paths. The compromised solution proposed here is easy to implement yet bring substantial improvement in correctness over pure simulation results. Quantitatively, the number of errors and missing features discovered during the course of the effort were in the order of tens. Qualitatively, we achieved quite impressive results i.e.

- No protocol errors found during RTL regressions.
- The protocol was operational on first silicon samples.

That means no architectural bugs escaped from our effort and protocol was complete in terms of features.

The requirement to use protocol specification during all stages of chip design created a secondary goal of verifying micro-architectural protocols and constraints in the context of this architectural protocol. We developed methods for composition of architectural and micro-architectural protocols and constraints and validated their interactions.

The rest of the paper goes like this. In section 2, we will provide an overview of our methodology. Section 3, will give an overview the protocol (called OCI: OCTEON Cache coherence Interconnect protocol) developed using this methodology. Sections 3, 4, 5, 6 and 7 will cover how we approached different phases of our development effort.

II. AN OVERVIEW OF OUR METHODOLOGY

The key objectives of our methodology were:

1. Identify the components of the protocol,
2. Create specifications for each of the components,
3. Identify the services provided by each component,
4. Model the protocol,
5. Create a comprehensive set of correctness and coverage properties of the protocol,
6. Create an automated flow of protocol validation that supports protocol revisions.

In rest of this section we will expand on each of the items in this list.

A. Protocol Components

A protocol can be viewed as a set of processes/agents [1] [3]. A process in a protocol maintains a local state and defines its legal behavior through a set of transitions. A transition has an enabling condition called its guard. More than one guard may be enabled in a given state and a process may randomly choose one among them. Processes are connected to other processes through shared variables called channels. A computation of protocol is a sequence of process transitions (Proc1State:Proc1Transition, Proc2State:Proc2Transition…). A computation sequence can be interleaving (processes taking turns to execute enabled transitions [1]) or can be parallel (process executing transitions in parallel [4]). The interleaving description makes the verification problem simpler and with random choice among enabled transitions is assumed to be equivalent to parallel descriptions [3].

B. Protocol Specification

Plenty of research has been done in defining Protocol specification languages like [1][2][3][4][5][6] to mention a few. However, the key idea behind all these specifications is "Finite State Tables", probably extended in some way to facilitate particular domain application. Our view on the protocol specification is that each process can be described as an extended state table and if that table is parseable, then it can be used to generate executable models. Each line in the table represents a potential transition of the process. The columns of the table consist of four sets:

1. Inputs from environment or channels,
2. Current state of the process,
3. Next state of the process and
4. Outputs to channels and/or environment.

The label of each column in the current-state and next-state defines the state variables of the process.

C. Protocol Properties

Once we have a model of the protocol in the chosen language, the next key question is "What to check?" i.e. what properties should hold in all of the protocol computations. We categorized protocol properties as:

- Protocol Completeness: A protocol specification consists of a set of interacting finite state machines i.e. essentially a set of transitions between a given state and next state for a given input. Given this set of transitions we want to avoid:
  - Under-Specification: Any missing transitions will cause the system to enter dead-end states.
  - Over-Specification: Any redundant transition will cause false errors in coverage collection at all levels.
  - Legal States Reachability: All the legal states of the protocol are reachable.
  - Missing/hidden assumptions: Any missing or hidden assumption of machine initial states etc. will cause false errors.

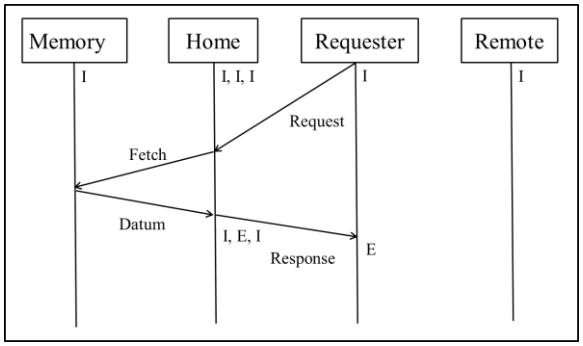

Figure 1: An OCI Sequence

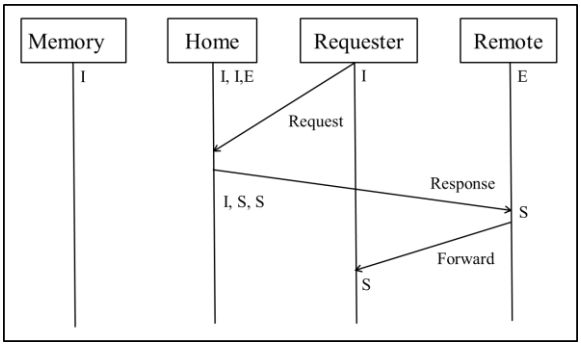

Figure 2: An OCI Sequence with Forwarding

- Protocol Correctness: These requirements arise from two sources, structure of the protocol and functionality of the protocol.
  - o Deadlock/livelock freedom: This is a structural property. Multiple interacting state machines always have the potential of having deadlocks, livelocks etc.
  - o Cache Coherence: This is a functional property, and requires that only one node at one time have writing-rights to datum.
  - o Data Consistent: This is also a functional property that requires that the data in a given memory location is propagated correctly.
  - o Designer Properties: The protocol architect has certain assumptions about the behavior as well as states. These are very protocol-specific but give a high degree of assurance to the architect.

For every revision of the protocol, our protocol model was verified against all these checks. Most of these checks were auto-generated from tables and others were manually crafted.

D. Protocol Constraints

Beyond just the logic required to implement the OCI protocol, the architect must define certain required constraints on the implementation to ensure that all of the assumptions the architect made during protocol definition are valid. One of the most common examples is the architect's definition of channels. The implementation typically must guarantee that lower priority channels do not block higher priority channels.

E. Protocol Validation Flow

Our protocol validation flow is shown in Figure 3. For our case, the protocol was under constant revision. Therefore, one of the key requirements was automation. Any changes in the protocol should be validated quickly. To meet this goal, we created a number of tools around our protocol verification tools (potentially Spin[1], Murphi[5] and Jasper Architecture verification tool[6]). The flow starts with Architects generating a Golden reference model in ASCII formatted tables. These tables are used to create six role/service tables. Verifiers create templates and instrumentations. All these items are merged together to create the protocol model, that is fed into formal/simulation tools for validation. The next few subsections provide details of the flow.

Models

There are four models involved in the flow:

1. Golden Reference Model: These are extended state tables. They are in ASCII format and auto generated by SpecGen. Each of these tables represents one of the basic processes of our protocol. Each table lists all the local variables, inputs, outputs and legal transitions possible by the process.
2. Roles/Service Tables: Auto-generated from the Golden Reference Model by FSpecGen. These are disjoint subsets of the transitions of each process table based upon services provided by the process.
3. Templates/Instrumentation: Manually created and managed by the verifier. Templates are used as a wrapper around raw protocol tables. There is one template per protocol service/role. Instrumentation includes model assumptions, reduction constraints, completeness properties and correctness properties.
4. The Model: This is a formally verifiable set of files, generated as composition of Process state tables, instrumentation and templates. In our case, they were either Promela [1]files for Spin or XML files for JasperArch modeling tool[6].

The Tool Set

There are four tools involved in the flow:

1. SpecGen: The tool used to auto-generate the Golden Reference Model. This tool manages the Protocol in a programmable way and provides a uniform consistent output for any changes in the protocol.
2. FSpecGen: The tool used to generate the six role tables for Formal Verification.
3. Merge: The tool to merge Protocol raw tables with templates and instrumentation to generate a formally verifiable

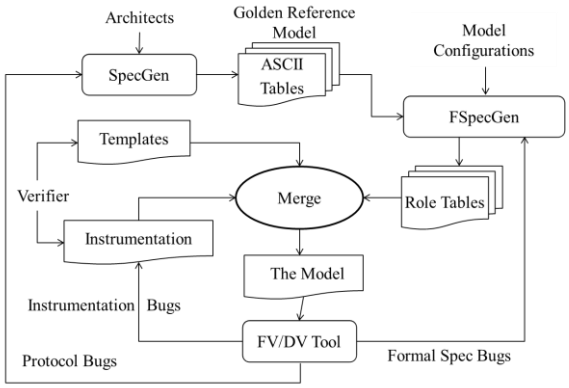

Figure 3: Protocol Validation Flow

specification.

4. FV/DV Tools: The tools to validate the model. We used Spin model checker [1] and Jasper Architectural tool [6] at different times for this purpose.

III. MODELING OCI

The modeling effort of OCI was based upon couple of observations:

- We can divide two main transitions tables (home and remote) into six unique roles/services tables.
- For a given address at a given time, any one of these roles can be picked up randomly and executed atomically without the need for concurrency as long as our search is exhaustive.

Based upon these observations, we devised a generic model as shown in Figure 4. The role/service tables are used to auto-generate sub-models like (LocalAddressRequest, RemoteAddressRequest, etc.). Most of the time, protocol iterations cause changes in these sub-models only and hence the protocol validation loop shown in Figure 3 is only limited by the FV/DV tools speed and progress is not hindered by manual translations and model changes. Each of sub-models like LocalAddressRequest consists of all the transitions of that rule in the form of choices through guards of each transition. If no guard matches then it is a failure of a check called "Full" as described in section V:B. Each statement set after a guard includes a logging statement recording the fact that the transition is reachable and covered. An important characteristic of the generic model is that it permits a non-deterministic choice between different OCI roles for a given address. Through the protocol revisions, the changes required in the instrumentation were minimal.

IV. HANDLING MODEL COMPLEXITY

The total number of possible transitions in the OCI protocol was more than 4000. The verification of the complete protocol for a 4-node configuration for all possible addresses was impossible. Any validation approach with some chance of success has to abstract the protocol in more than one way. In next few sections we will provide a brief overview of some of the abstractions employed in our effort.

A. Parameterization

We believe parameterization of a model is the key to success of any formal verification effort. OCI model is parameterized in terms of Addresses, Nodes and Channel depths. For pure architectural exploration, this address was set to 1 but for micro-architectural explorations, this was set to higher numbers like 4 and 8. Similarly two, three and four node models can be generated on demand. Any new change in the protocol is usually tested on the two-node model for a quick check of the modification before being run on three and four node models.

B. Address Abstraction

The first key observation about OCI is that it is a per-address based protocol. No two addresses interact with each other's protocol space. Using this observation, we totally abstracted away the address from the Model. However, please note that this abstraction does not hold when we validate micro-architectural protocols and abstractions in OCI context. It is because in micro-architectural space, different addresses share queues, channels and other data-structure and cannot be considered in total isolation of each other.

C. Initial State Abstraction

Generally, model verification assumes an all-invalid (or all-zero) initial state. However, our specification also provided a list of all the valid non-transient states of the model. We have proven that all states in this list are reachable and when there is no transaction in flight for a given address, OCI is always in one of these valid states. With these proofs in hand, we can assume any of these states as a starting state for model verification. This technique provides model verification tools multiple initial states. Since for each of these legal states, only a subset of all transitions will be enabled, it divides the verification problem into smaller pieces and results in much faster coverage of the complete protocol transition set.

```
//The Polling Loop, for a given address
start:                                  //Starting  point of the polling loop
                                                    //Choose randomly among
available choices.
case(true)
 (Home node has no pending request):   goto LocalAddressRequest;
(Remote node has no pending request):  goto RemoteAddressRequest;
(There is a request from a remote): goto HomeReceiveRequest;
(There is a response message for remote):  goto RemoteReceiveResponse;

(There is a forwarding message for remote): goto RemoteReceiveForwardRequest;
(There is response message for home): goto HomeReceiveResponse;
Default:   goto deadlockchecker;     //No options for this address; It is a deadlock.
endcase
goto start;
```

Figure 4: The Polling Loop

D. Configurations

The OCI protocol consists of more than 50 core instructions. Validation all of these together is desirable but at times may not be feasible. Generally, for any changes in the protocol, a quicker partial correctness result shows that the model is not broken and then one can run complete correctness tests for long period of time. The important observation for this abstraction is that all of these instructions assume a legal initial state from the state specification table and leave the system in a legal final state again from the state specification state. This implies that we can use a subset of the instructions to create small models for quick assurance of the correctness of the Protocol modifications. With architectural insight of the behavior of these instructions, we can group them in clusters of related instructions and validate group behavior.

V. VALIDATING OCI

We used two different tools for OCI validation. The first one is Spin [1], a free tool that has been around for twenty some years. The second one is a commercial tool called Jasper Architectural modeling tool [6].

A. Spin Model Checker

Our initial exploration of the protocol was based upon free formal tools like Spin [1], TLA [2] [7] and Murphi[5]. We decided to go along with Spin because of: 1) continuous tool development, 2) tool documentation, and 3) most importantly prompt tool support from Gerard Holzmann.

Spin's Modes of Operation

We used Spin in three different modes at three different stages of our work:

- Interactive simulation: This capability of Spin[1] proved to be a great help in the initial model development. We can choose messages going across the channels connecting nodes and see how the model reacts. It helps us greatly in path clearing and feature testing of the protocol.
- Formal Verification: Once the model has end-to-end flows in place, formal verification aspects of the tool helped us find deadlocks, deadends, livelocks in it. Eventually we have full protocol in the model with the exception of the data-part. With the full model, formal verification was running for days without results. At this stage Swarm[8] helped us to run and manage multiple jobs.
- Random Simulation: Spin has built-in random simulation capability with a command line seed option. All verification environments today have compute grids being used for exerciser runs with random seeds. We modified Swarm [7], to run random simulations continuously using Spin with different seeds (like exercisers). It helped us explore the model much deeper.

B. Jasper Architectural Modeling Tool

Jasper [6] is a state of the art commercial formal verification tool set with a host of engines and debugging capabilities. Jasper architectural tool is useful for modeling and validating protocols. The input to the tool is in the form of tables. This ability gave us leverage to use the same table specifications which we were using to generate Spin models. Also, our protocol validation flow depicted in Figure 3, mostly remained intact with the replacement of SPIN with Jasper in FV tool box. It took us just a couple of weeks to align our validation flow around the Jasper Arch tool. Jasper architectural tool provided a number of advantages, like auto-generated cover statements for all lines of the protocol, parallel and full checks for role tables, debugging capabilities etc. This investment has been proven worthy as we were able to achieve our goals in 4 months as compared to the otherwise projected time of 8 months. The details of this work were presented at [9].

VI. RESULTS

The effort was a great success. Tens of bugs and missing features were found. Here is short list of achievements:

1. Found missing actions in the protocol.
2. Found missing/hidden assumptions of the protocol.
3. Found missing transitions causing deadends.
4. Found unreachable transitions.
5. Proved that the protocol is cache coherent and data consistent.
6. Proved that the protocol is deadlock free.

Table 1 provides bugs categories and their relative occurrence rate.

Table 1: Bugs Categorization

Category	Percentage
Missing Cases/Dead-Ends	30%
Unreachable cases	30%
Deadlocks	10%
Missing Assumptions	10%
Data/Masking	10%
Miscellaneous	10%

VII. CONCLUSION AND RELATED WORK

Protocol verification is a well-researched field and few examples can be seen in [1][2][3][5][6][7][10]. A number of academic and commercial tools are available to help the cause. The protocol verified by O'Leary etc. [10], is very similar to OCI but what our works shows is how available tools can be used in an industrial environment under commercial constraints in an efficient way. The OCI protocol has more than 4000 transitions and none of the works we know have tackled such a huge problem in such a short time. We could have compressed the protocol manually by merging similar transitions, creating helper assertions and seeking complete proofs for a 4-node configuration with all 50+ instructions. However, that would have taken the automation away and introduced errors in the translation for each of the protocol revisions. More importantly, we could not have met our time constraints. Instead, we focused on efficiency and quick turn-around time. We used Spin in earlier stages and JasperGold for most of the later work. However our methodology does not restrict itself to particular tool. The tools choice was determined by other constraints like available-time, efficiency, support etc. Finally this work was closely coordinated with our RTL verification efforts. The details on that effort can be found here[11].

We have shown how a combination of automated flows, role identifications, smart abstraction schemes and good FV/DV tools can be orchestrated together to achieve good results in a short time. There is no claim of perfection for this effort as it is all about engineers' tinkering to achieve quick results. The usefulness of this approach was evident in its application to a couple of very complex chip designs where the protocol. We are extending the techniques described here to help in coverage convergence and other verification challenges.

REFERENCES

[1] Holzmann, G.J. 2004. The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, ISBN 0-321-22862-6, 608 pages, 2004.

[2] Plakal, M. Yu, etc. 1998. Lamport Clocks: Verifying a Directory Cache-Coherence Protocol. In Proceedings of the Tenth ACM Symposium on Parallel Algorithms and Architectures. Pages 67-76, June 1998. DOI= http://doi.acm.org/10.1145/277651.277672

[3] Gouda, M. G. 1993. Protocol Verification Made Simple. In Proceedings of Computer Networks and ISDN Systems 25. Pages 969-980, 1993.

[4] Milne, G.J. 1984. A Model for Hardware Description and Verification.. In Proceedings of DAC1984, June 2-6, 1984. 21st Design automation Conference. 1984.

[5] Dill, David L. etc. 1992. Protocol Verification as a Hardware Design Aid. In Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors. Pages 522-525, October 1992.

[6] Weber, Ross. 2011. Modeling and Verifying Cache-Coherent Protocols, VIP, and Designs. Jasper Design Automation, June 2011.

[7] Tasiran, S. Yu, Yuan and Batson, Brannon. 2003. Using a Formal Specification and a Model Checker to Monitor and Direct Simulation. In Proceedings of DAC2003, June 2-6, 2003. 23rd IEEE/ACM Int. Conf. on Automated Software Engineering. L'Auila, Italy, Sept. 2008. DOI= http://dx.doi.org/10.1145/775832.775926

[8] Holzman, G.J. Joshi, R. and Groce, A. 2008. Swarm Verification Techniques. In Proceedings of ASE 2008, 23rd IEEE/ACM Int. Conf. on Automated Software Engineering. L'Auila, Italy, Sept. 2008. DOI= http://dx.doi.org/10.1109/TSE.2010.110

[9] Ikram, Shahid and Akkawi, Isam. 2013. Modeling and Verifying a Coherence Protocol using JG-ARCH. In Proceedings of Jasper Architectural Formal Verification Forum 2014, Santa Clara, CA, Oct. 2013. http://dx.doi.org/10.13140/2.1.1047.4245

[10] O'Leary, John, Talupur, Murali and Tuttle, Mark R. 2009. Protocol Verification Using Flows: An Industrial Experience. In the Proceedings of FMCAD 2009 DOI= http://dx.doi.org/10.1109/FMCAD.2009.5351126

[11] Ikram, Shahid and Akkawi, Isam, etc. 2014. A Framework for Modeling, Specifying, Implementation and verification of SOC Protocols. In IEEE SOCC Conference Proceedings 2014, Las Vegas, NV, September. 2014. http://dx.doi.org/10.13140/2.1.4094.8480