Modular Implementation of Directory-based Cache Coherence for Multicore Processing

Ullas Pai, Naorem Akshaykumar, Deepank Grover, Sujay Deb
Department of ECE,
Indraprastha Institute of Information Technology Delhi
Emails: {ullas20255, naorema, deepankg, sdeb}@iiitd.ac.in

2024 IEEE International Symposium on Smart Electronic Systems (iSES) | 979-8-3315-3322-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISES63344.2024.00067

***Abstract*—Modern CPU designs are shifting toward multi-core architectures, as many commercial and scientific workloads benefit significantly from concurrent and multithreaded operations. This leads to increased throughput, performance, and efficiency. The cache coherence protocols are used for communication and coordination in multi-core - to achieve this gain. In the open-source environment, particularly in CVA6, the lack of a coherence-enabled cache controller motivates the need for a modular approach to enable multicore processing using CVA6. To overcome this limitation, we propose an invalidation-based directory protocol to enable cache coherency in multi-core shared memory Network-on-Chip (NoC) architecture.**

I. Introduction

As Moore's Law slows down, CPU designers are shifting from single-core to multi-core designs to boost performance. Many different applications can be divided into multiple independent tasks. The Multi-core processor can exploit this independent task for concurrent execution, thus increasing throughput, efficiency, and performance. The Shared memory computing model is the dominating paradigm in this multi-core processor environment that enables parallel execution [1] [2]. However, in a shared memory architecture, each core has a private cache. The cache coherence protocols ensure the correctness of shared data among the multi-core caches. These protocols help in communication and coordination by managing memory read-write operations and ensuring all the cores observe the latest version of data by either updating or invalidating a cache line to prevent the usage of stale data.

The cache coherence protocol can be implemented in either invalidation or update-based schemes. The invalidation-based design offers several advantages compared to the update-based [3]. Independent of these two implementations, data coherence can be ensured via two approaches, namely, snoopy and directory. In the snoopy-bus-based protocol, all messages are broadcast, and a total order is maintained through a bus-based interconnect. In the case of a directory-based protocol, coherence messages are unicast and the order is maintained at the directory.

In the open-source environment, integrating different cores remains a significant challenge. In this paper, we are proposing an invalidation-based directory protocol to support coherence for RISCV-based CVA6 core, enabling multi-core processing. The CVA6 is an application class 6-stage, single-issue, in-order CPU [4]. The directory-based design is chosen to support scalability in core integration. We designed and verified a quad-core (4-cores) system as shown in Fig. 1, over a mesh Network-on-Chip (NoC) interconnect, leveraging the NoC for its ability to provide a scalable memory hierarchy [5].

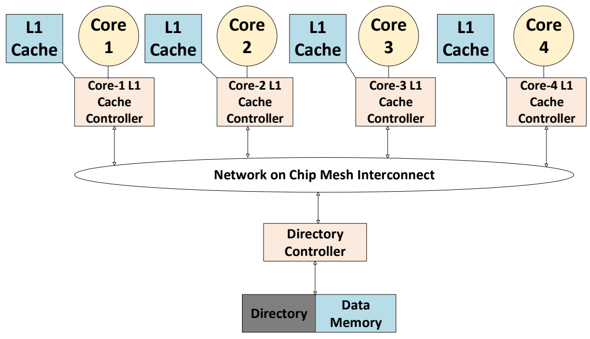

L1 Cache
Core 1
L1 Cache
Core 2
L1 Cache
Core 3
L1 Cache
Core 4
Core-1 L1 Cache Controller
Core-2 L1 Cache Controller
Core-3 L1 Cache Controller
Core-4 L1 Cache Controller
Network on Chip Mesh Interconnect
Directory Controller
Directory
Data Memory

Figure 1: Directory System Model

II. BACKGROUND

A. MSI-Cache Coherence Protocol

Cache coherence protocol enforced two invariants: Single-Writer, Multi-Read Invariants (SWMR), and Data-Value Invariants [6]. It specifies the interaction between the coherence controllers associated with the private cache and LLC/memory to enforce these invariants. In our work, we are implementing the MSI cache coherence protocol. In this protocol, each cache block within the directory can be in either the M, S or I state.

- **Modified (M)**: The cache block/line is valid, owned, exclusive, and potentially dirty. In a directory-based design, a core having the cache line in read/write access is in the Modified state, and there should be only one copy of the particular cache line.
- **Shared (S)**: The cache block/line is read-only, valid, not owned, and not exclusive. In a directory-based design, there can be multiple copies of the cache line in the Shared state. In this state, the directory holds the ownership of the cache line.
- **Invalid (I)**: The cache block/line is invalid; the particular cache line is not present.

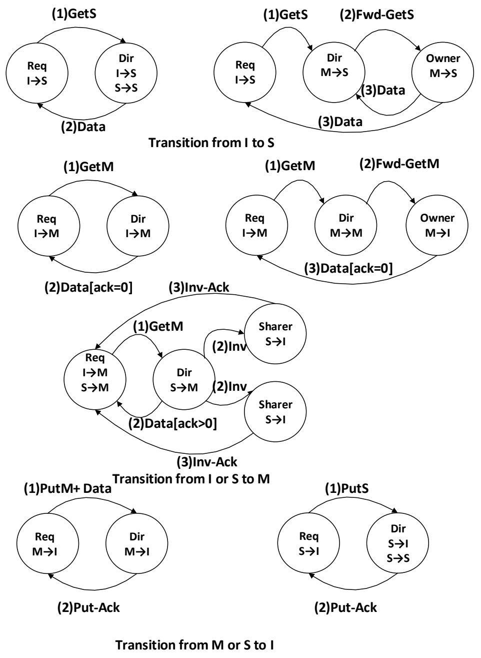
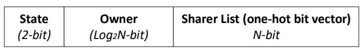


Figure 2: State transition of MSI protocol reproduced from [6].

B. Directory System Model



State (2-bit)	Owner (Log₂N-bit)	Sharer List (one-hot bit vector) N-bit

Figure 3: Directory entry for a cache line.

The directory cache coherence model is introduced to implicitly allow caching and coherence on processors in physically distributed systems, eliminating the need for a common bus. When a processor/core incurs a cache miss, it will communicate to the directory in point-to-point transaction to determine the location of the cache line. A directory entry format for a cache line in a system with N nodes (cores) is shown in Fig. 3.

III. Directory-based MSI Protocol Implementation

The transition between various states of MSI is shown in Fig 2. The details can be found in the works of [6].

A. System level messages

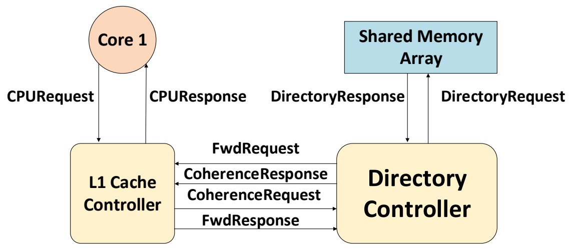


Figure 4: Illustrative list of messages in our design.

The messages in our design that are being routed for various transactions, including the responses generated by the CPU/cores and the controllers, are shown in Fig. 4. Since **CPURequest** and **CPUResponse**, **CoherenceRequest** and **CoherenceResponse, DirectoryRequest** and **DirectoryResponse** are self-explanatory, the details of the remaining are provided below:

- **FwdRequest-** If the cache line is held in a **Modified state** by other cores, the directory forwards the request for either a **Shared** state (load) or an **Invalid** state (store), depending on the CPU request for the cache line.
- **FwdResponse-** After processing the FwdRequest, the latest data and acknowledgment are sent in this response message toward the requesting core.

Table I lists the set of transactions supported in our design in response to the requests from the associated cores:

Transaction	Goal of Requester
GetShared **(GetS)**	Obtained the block in shared state (Read-only)
GetModified **(GetM)**	Obtained the block in modified state (Read-write)
PutShared **(PutS)**	Evict the block in shared state
PutModified **(PutM)**	Evict the block in in modified state

Table I: Transaction used in our design

B. *The Directory Controller*

The directory controller is responsible for handling coherence requests from multiple cores in serial manner. The FSMs of the directory controller and cache controller obey the state transitions shown in Fig. 2. It can handle a transition from **M→S, M→M, S→M, S→S, I→S, I→M** for a particular cache line, apart from the invalidation requests.

C. *The Cache Controller*

The Cache controller is responsible for handling CPURequest and generating either a response or coherence transaction depending on whether it hits or misses in the cache. In our design, the forwarded coherence request has higher priority over CPURequest. The new CPURequest will not be fetched into the cache controller until its previous request has been served completely, including the response from the directory, i.e., when the cache controller handles the Forwarded Coherence request.

IV. The NoC and Network-Interface IPs

The NoC provides a scalable interconnect between processors and memory. It uses data/inst packets to route from the source to the destination through a network fabric that consists of switches and interconnect links. The Network-on-Chip topology can either be - direct network, indirect network, or irregular network [7]. In our design, we are using a direct network topology as it provides point-to-point links. Data through the NoC is transmitted in the form of 32-bit flits packets. The coherence messages are stored in a synchronous FIFO before being processed at the receiver end. The FIFOs are required at every cache controller and the directory to buffer requests.

- The Flitter is used to disassemble the messages into flits of 32 bits. At cache, CoherenceRequest and FwdResponse have to be disassembled into flits. Moreover, the FwdResponse may or may not contain data; only the acknowledgment is sent. In the directory, CoherenceResponse is sent to the requesting core based on the destination node coordinate, and FwdRequest is sent to all sharers or the existing owner.
- The Deflitter is used to reassemble the messages and identify the message. At the cache, FwdRequest and CoherenceResponse are reassembled and stored in their corresponding FIFOs. At the directory, CoherenceRequest and FwdResponse (and acknowledgment) messages are handled. The CoherenceRequest and FwdResponse is reassembled and written to its FIFO. Whereas, the acknowledge (Ack) messages from different nodes (cache controllers) are accumulated and compared with the expected number of acknowledgments.
- The Read Arbiter: Arbitration is required in both the cache and directory. At the cache, simultaneous requests can arrive from the CPU (CPURequest) as well as the directory (FwdRequest or CoherenceResponse). At the directory, simultaneous requests can arrive from different caches (CoherenceRequests or FwdResponse).

V. Simulation and Verification of a Multicore System

We simulated a quad-core (4-core) system using Intel ModelSim and the screenshots of the simulation window are from the same software. The directory controller is placed in the central node of the network, i.e., Node 4, as shown in Fig. 5. The detailed architectural view of our design is shown in Fig. 6. Our NoC implementations are based on the methodology outlined in [8]. The CPU requests are buffered into the CPU Requests Buffer through the use of system verilog tasks.

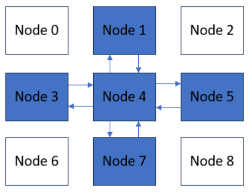


Figure 5: Connection of Nodes in the quad-core system

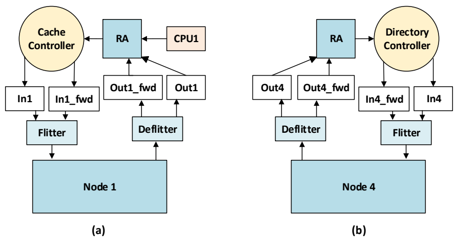


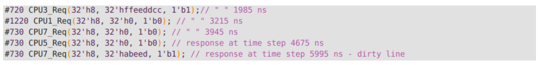
Figure 6: Detailed architecture of (a) CPU node 1 and (b) Directory node 4 - in the quad-core system.

For illustrative purposes, we are presenting the series of events listed below:

- Core 3 requests for a write to address ’8’ with data “ffeeddcc”.
- Core 1 requests for a read of address ’8’.

- Core 7 requests for a read of address ’8’.
- Core 5 requests for a read of address ’8’.
- Core 7 requests for a write to address ’8’ with data “000abeed”.

Fig. 7 depicts a series of requests fed to the CPU requests’ buffer of different cores to test the quad-core system. Initially (Fig. 8), CPU3 requests to store data at address 8, and CPU3 obtains **M** access to it. Then, CPU1 sends load request of address 8, obtaining it with **S** access, causing an **M** −> **S** transition at the directory. Similarly, CPU5 and CPU7 also obtain **S** access to address 8 (Figures omitted). Finally (Fig. 9), when CPU7 wants to write to address 8, it will cause the invalidation of the other copies (Fig. 10) and obtain **M** access for address 8.



```
#720 CPU3_Req(32'h8, 32'hffeeddcc, 1'b1);// " " 1985 ns
#1220 CPU1_Req(32'h8, 32'h0, 1'b0); // " " 3215 ns
#730 CPU7_Req(32'h8, 32'h0, 1'b0); // " " 3945 ns
#730 CPU5_Req(32'h8, 32'h0, 1'b0); // response at time step 4675 ns
#730 CPU7_Req(32'h8, 32'habeed, 1'b1); // response at time step 5995 ns - dirty line
```

Figure 7: The CPU Requests generated by calling above-mentioned tasks

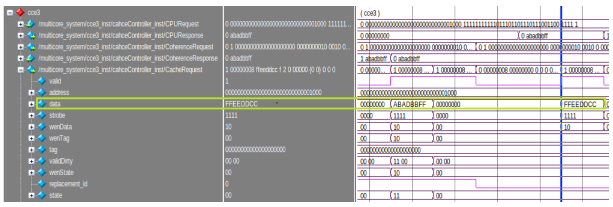


Figure 8: CPU3 receives address 8 with ’M’ access and writes ’ffeeddcc’.

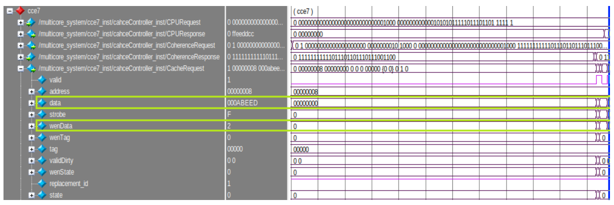


Figure 9: CPU7 obtains ’M’ access to address location 8 and writes ’000abeed’.

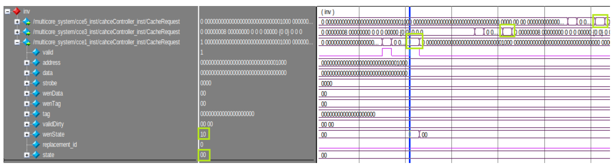


Figure 10: Invalidation of copies of address 8 with sharers.

VI. Optimizations and Future Works

In our design, the Owner(O) and Exclusive(X) states can be added to optimize performance. Such a state can help avoid the write-back of the latest value to memory (by transitioning from M to O) and also enable a silent upgrade to the M state (by transitioning from X to M) without explicitly communicating with the directory. Another optimization could involve having the cache controller respond directly to the requesting cache controller for forwarded requests. This approach would reduce the number of hops needed to maintain coherency by one. The final stage will be integrating the coherence engine (Controllers along with the NoC) to handle the actual CPU workload using CVA6 cores.

VII. Conclusion

We have successfully designed and verified a directory-based cache coherence protocol using invalidation wherein messages are routed via a 3x3 NoC mesh fabric. The objective of this work is to facilitate the support for cache coherence in CVA6 cores, thereby enabling multicore processing with CVA6. This modular approach will significantly enhance the performance and scalability of the CVA6 multi-core system.

VIII. Acknowledgement

This work was partially supported by Thales India.

References

[1] M. M. K. Martin, M. D. Hill, and D. J. Sorin, “Why on-chip cache coherence is here to stay,” *Communications of the ACM*, vol. 55, pp. 78–89, July 2012.

[2] S. H. Gade and S. Deb, “A novel hybrid cache coherence with global snooping for many-core architectures,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 27, Sept. 2021.

[3] D. Glasco, B. Delagi, and M. Flynn, “Update-based cache coherence protocols for scalable shared-memory multiprocessors,” in *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, vol. 1, pp. 534–545, Jan. 1994.

[4] OpenHW Group, “CVA6 RISC-V CPU.” Available at: https://github.com/openhwgroup/cva6 (Accessed: 3 December, 2023).

[5] D. Giri, P. Mantovani, and L. P. Carloni, “Noc-based support of heterogeneous cache-coherence models for accelerators,” in *2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pp. 1–8, 2018.

[6] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture, Cham: Springer International Publishing, 2020.

[7] S. Pasricha and N. Dutt, *On-chip communication architectures: system on chip interconnect*. Amsterdam ; Boston: Elsevier / Morgan Kaufmann Publishers, 2008.

[8] M. Kumar, “Project presentation.” https://bit.ly/noc-des, 2023. Last Accessed: May 10 2024.