Cache Coherent Framework for RISC-V Many-core Systems

Zexin Fu
fzx20@mails.tsinghua.edu.cn
Tsinghua University

Mingzi Wang
wmz22@mails.tsinghua.edu.cn
Tsinghua University

Yihai Zhang
zhang-yh22@mails.tsinghua.edu.cn
Tsinghua University

Zhangxi Tan
xtan@rioslab.org
Tsinghua University

ABSTRACT

Given the limitations of single-core processors in achieving performance gains, the industry and academia have shifted towards many-core processors. While many-core processors have many advantages, their design poses several new requirements, such as on-chip interconnect scalability, cache coherence maintenance, and large system verification. Thus, this paper aims to address some of these challenges by proposing a completely verified many-core cache coherent system framework with an efficient and scalable on-chip interconnect design.

This paper presents a comprehensive study of the design, implementation, and verification of a many-core cache coherence system that employs a directory-based MESI coherence protocol and a 2D mesh NoC. The main contributions of the paper are the design and implementation of a MESI cache coherence protocol suitable for package switching NoC, a sparse directory-based Snoop Control Unit (SCU) used to maintain system coherence, and a 2D mesh-based Network-on-Chip (NoC) that enables high bandwidth, low latency, and scalable many-core systems. The cache coherence system is verified under a novel verification flow using both simulation and formal methods. This work is planned to be open-sourced to boost many-core system research and development in the RISC-V community.

KEYWORDS

Computer Architecture, Cache Coherence, Network-on-Chip, Verification, Many-core

1 INTRODUCTION

Recent years have witnessed the core counts of chips both industry chip vendors and academic groups have released growing rapidly, leading to hundreds and even thousands of cores integrated on a single chip. There have already been server chip products with more than 100 cores on-die[1]. Additionally, chiplet technology, enabled by advanced packaging technologies, allows many silicon dies to be integrated into a single package, resulting in higher core counts in the system.

Many-core processors offer several benefits[4], which have contributed to their increasing popularity in recent years. Firstly, many-core processors provide a scalable and versatile means of improving performance beyond the limitations of the underlying technology. This is particularly relevant since it is challenging to exploit more instruction-level parallelism (ILP). Secondly, the replication of cores within a multi-core processor enables the leverage of design investments, which in turn facilitates the modular design and Intellectual Property (IP) reuse, resulting in cost savings. Finally, the sharing of on-chip resources such as system-level cache and peripherals leads to improved resource utilization and enables optimization techniques such as private cache sharing. Consequently, multi-core processors have replaced single-core processors as the mainstream direction for processor design.

To achieve a many-core system that boasts high performance and reliability, several key areas must be addressed, such as on-chip interconnect scalability, cache coherence maintenance, and large system verification.

On-Chip Interconnect Scalability The growth in the number of cores in modern many-core processors has resulted in a significant increase in communication traffic between them, requiring scalable on-chip interconnects. The design of on-chip interconnects needs to provide low-latency, high-bandwidth communication channels while accommodating various traffic patterns and ensuring equitable resource sharing. The Network-on-Chip (NoC) architecture is a promising solution for addressing the challenge. NoC replaces traditional bus-based interconnects with a packet-switched network that can handle multiple communication requests concurrently. NoC can provide scalable, efficient, and predictable communication between cores while also supporting various traffic patterns.

Cache Coherence Maintenance Cache coherence is the mechanism that ensures that all cores in a many-core system have a coherent view of shared memory. With multiple cores accessing the same memory locations, it's crucial to maintain cache coherence to avoid conflicts and inconsistencies. Cache coherence maintenance involves tracking changes to shared memory locations and ensuring that all caches are updated with the latest data. This requires complex hardware and software mechanisms to ensure that data is consistent and up-to-date across all cores. Directory-based coherence, snooping-based coherence, and many other variations are used to solve this problem.

Large System Verification As many-core processors become more complex, verifying the correctness of the design becomes increasingly challenging. Large system verification involves testing the design to ensure that it meets its functional and performance requirements. This requires sophisticated verification tools and techniques to test the system's functionality, performance, and reliability. As many-core processors increase in complexity, verifying the design becomes more challenging, and designers need to use advanced verification techniques, such as simulation, formal verification, and emulation, to ensure the design is correct.

This paper contributes to the field in several significant ways, including the following:

- A Cache coherent system: we present a cache coherent system with MESI protocol, including a private cache controller and a sparse directory-based Snoop Control Unit (SCU).
- A 2D-mesh-based Network-on-Chip: we present a NoC with 2D mesh topology, dimension-ordered routing, and flit-based flow control, as well as its verification environment.
- A cache coherence verification flow: we present a verification framework for a cache coherence system with both simulation-based verification and formal verification, for both model and RTL implementation.

2 RELATED WORK

In this section, we discuss several other cache coherent fabric projects and the key distinguishing features of our work.

Chipyard and TileLink Chipyard is an open-source platform for designing and evaluating advanced SoC designs that use TileLink as its default on-chip interconnect standard. By leveraging the TileLink interconnect, Chipyard provides a communication infrastructure for the different PEs in the SoC, thereby enabling the design of complex, multi-core SoCs with high performance and reliability. However, the TileLink protocol is not designed for packet-switching NoCs, e.g., its flow control is based on a valid-ready handshake instead of the credit-based flow control widely used in NoCs. Additionally, Chipyard is built using the Chisel hardware construction language, which helps speed up the ASIC front-end process but can cause difficulties with back-end processes, such as the renaming of wires.

Open2C Open2C is a tool developed to explore cache-coherent memory subsystems in large-scale computing systems. Open2C provides a library of parameterized components that can be used to build a complex coherent cache memory subsystem, including a missing register, TAG array, replacement policy unit, and others. But it doesn't include a NoC implementation, and since it's built with Chisel, it has the same ASIC backend process difficulties as Chipyard.

OpenPiton OpenPiton is a research framework that enables academic research of many-core systems. It uses a tile-based architecture on top of a multi-plane NoC. Each OpenPiton tile includes a private L1.5 cache and a slice of the distributed L2 cache. The chipset in an OpenPiton chip provides access to external DRAM and other I/O and connects to other OpenPiton chips through the NoC via a chip bridge. OpenPiton can be configured with up to 500 million cores, but the memory bandwidth is limited by the 32-bit wide chip bridge between the tile and the chipset, which results in lower scalability in terms of memory bandwidth.

3 THE SYSTEM ARCHITECTURE

The coherence system architecture is aimed at a scalable many-core design for server applications as Figure 1 illustrates. The system features a coherent network, scalable cache coherence support, and a mesh linker for on-die network scale-up. The system is built with tiles, including Compute Tile, Home Tile, I/O Tile, and PCIe Tile.

Compute Tile As shown in Figure 1(b), each Compute Tile can be configured with multiple cores to form a cluster, each core has its own private L1 cache, each cluster has a cluster level SCU to maintain the cache coherence inside the cluster, and each cluster has

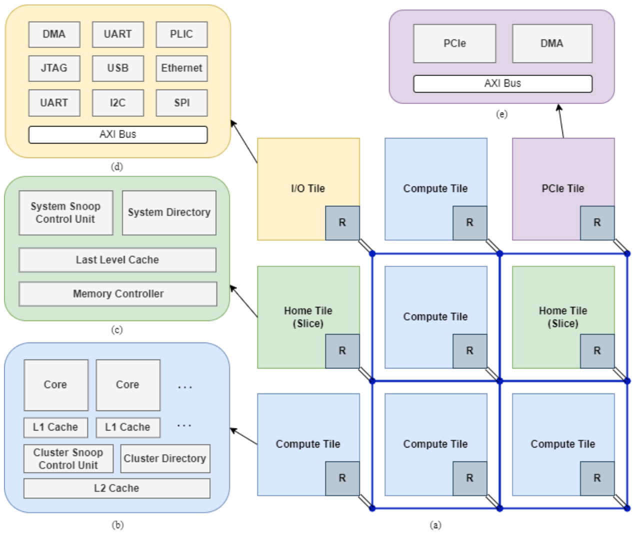

Figure 1: System Structure

a shared L2 cache, which is inclusive with L1 cache, so the cluster level directory can be merged with the shared L2 tag. The cluster-level SCU can quickly and efficiently handle coherence requests inside the cluster, reducing the latency of coherence requests and reducing the number of requests that need to be sent to the NoC. Meanwhile, each cluster is represented as a whole to the outside SoC, so only one bit is needed to represent a cluster for the system-level coherence directory, which leads to better scalability.

Home Tile As shown in Figure 1(c), each Home Tile contains a system-level SCU, a system-level directory, a last-level cache, and a memory controller. The system-level SCU manages cache coherency requests between clusters and tracks valid cache lines in each cluster in the system-level directory, the tracking granularity is each cluster. If there are multiple Home Tiles in the system, each of them will work in the form of a Home Tile slice, which will manage the cache coherence and the read and write access to the memory of a part of the memory addresses through the configured memory address interleaving method.

I/O Tile I/O Tile contains many peripherals, such as UART, I2C, SPI, etc., as well as DMA and platform-level interrupt controller (PLIC). There is also JTAG for test and debug access.

PCIe Tile PCIe Tile includes PCI express controller and DMA controller, and some other peripherals can be drawn out through high-speed PCIe channels to a chipset.

3.1 The Coherence Protocol Design

For the coherence protocol design for the aforementioned multi-core system, we made the following design decisions and the reasons are followed.

3.1.1 Cache line state information tracking mechanism. We choose the sparse directory, which reduces the directory storage requirement compared to traditional directory-based protocols because it only stores directory entries for cache lines that are cached in multiple caches. As a result, it avoids storing redundant information

Table 1: Five Virtual Networks

Channels	Blockable	Description
Request	Yes	Request from private cache to SCU
Evict	Yes	Evict request from private to SCU
Response	No	Response message, both direction
Snoop	No	Snoop request from SCU to private cache
Data	No	Data message, both direction

about single-cached lines, which reduces the overhead of maintaining directory coherence. And as it is a directory-based protocol, it has the advantage of reducing network traffic and energy consumption compared to snoop-based protocols.

3.1.2 Protocol state. We choose the MESI protocol. By introducing Exclusive state, the MESI protocol reduces the coherence traffic when a processor reads a line and then subsequently writes it in MSI protocol. This is a typical sequence of load then store requests in many important applications, not only for multi-threaded applications[7].

3.1.3 Message types. The coherence protocol has five different message types, namely request, evict, response, snoop, and data channel, each forming an independent virtual network. Table 1 shows the message structure design of each channel. It is worth noting that the evict channel is divided from the request channel, the reason why we divide them is that the request channel has the lowest priority, its messages can be stalled by many other messages and architecture states, but evict messages are critical and shouldn't be blocked by any other messages.

3.2 The Coherence Controller Design

In this section, we present the design of a private L1 coherence data cache controller and SCU+LLC controller.

3.2.1 Private Cache Coherence Controller. To handle a snoop request, the cache coherence controller needs to check the tag ram and Line State Table (LST) for line state and generate a snoop response corresponding to the check result. In some cases, snoop transactions may conflict with other transactions, requiring special consideration. During the design, we find some corner cases which require special handling processes, followings are some examples.

- First case: when a conflict happens between a snoop request and an existing valid request in the pipeline, requiring the snoop buffer to stall until the pipeline is empty.
- Second case: when a conflict happens between a snoop request and a valid request in the MSHR, requiring the snoop buffer to stall until the MSHR is available or back-pressing all incoming requests.
- Third case: when a conflict happens between a snoop request and an Eviction Write Request Queue (EWRQ) entry, requiring the snoop transaction to stall until the eviction process is completed before executing.
- Fourth case: if there is a same-line-address write-back request from the upper-level memory that hits an in-flight coherent transaction, the write-back cache line should be forwarded to the hit coherence transaction, and no need to allocate a new MSHR for the write-back transaction.

3.2.2 SCU + LLC Controller Design. The snoop control unit (SCU), and last-level cache (LLC) are integrated together. The LLC is designed to be inclusive of private caches, and the sparse directory to track cache line-sharing information is integrated with the tag in LLC. The inclusion property of the LLC with private caches means that all the data present in the private caches is also present in the LLC. This property provides several advantages, such as reduced latency and improved cache coherence. Since all private caches have a copy of the data in the LLC, when a cache miss occurs in one of the private caches, the data can be retrieved from the LLC instead of going to the next level of memory, which significantly reduces access latency. Additionally, the inclusion property can enable the SCU directory to merge with the LLC tag. This implies that a single tag entry in the LLC can represent both the valid and dirty state and the coherence state for private caches. This also simplifies the directory eviction invalidation by merging it with the LLC eviction invalidation to keep cache inclusion. This approach can reduce the complexity of cache coherence protocols and increase their efficiency by minimizing the number of coherence messages exchanged between caches.

In order to achieve high parallelism, the SCU is designed based on the Snoop State Table (SST), each SST entry can operate its own FSM independently, and in order to serialize some access to the critical resource like ram read and write and message sending, a lot of first in first out (FIFO) queues are employed, namely memory write queue, memory read queue, tag+directory ram write queue, data ram read queue, data ram write queue, snoop message send queue, response message send queue, data message send queue. Although there are a lot of queues, they will not take a lot of extra storage because they only store SST id and replace-SST id, not the actual message and data it involves. Only when it is at the top of the FIFO queue, it will read the SST entry it points to, parse the SST's state and do corresponding pending action. This enables fine-grained control of all the critical resources and helps SCU to improve resource utilization.

3.3 The Network-on-Chip Design

This section describes the design of the NoC, including topology, routing, flow control, router micro-architecture, and Quality of Service (QoS) support.

3.3.1 Topology. The 2D mesh topology was chosen as it provides an efficient, scalable, and straightforward solution for interconnecting the cores. With four neighboring nodes connected bidirectionally through links, the 2D mesh can provide efficient communication between nodes while maintaining a regular and easily implementable structure.

3.3.2 Routing. We choose DOR with X-Y routing and look-ahead routing. In the case of 2D mesh topology, DOR with X-Y routing sends packages along the X-dimension first, and after the value of X-coordinate equals to the target X-coordinate, it then sends packages along the Y-dimension, it is naturally dead-lock free[5]. Look-ahead routing removes the route computation (RC) stage from the critical path by determining the route of the routing packet one hop in advance and encoding it within the head flit. This enables incoming flits to compete for virtual channels (VCs) and switches immediately after the buffer write (BW) stage, and the route computation for the next hop can be performed in parallel with VC/switch allocation.

3.3.3 Flow Control. We choose the wormhole flow control with virtual channel flow control and credit-based buffer backpressure mechanisms due to its ability to minimize packet latency while conserving area and power. The wormhole flow control technique enables a router to send a flit as soon as a buffer of next hop is available, reducing latency compared to packet-based techniques. The virtual channel flow control technique provides additional virtual channels, allowing multiple packets to share a physical link and preventing congestion. Credit-based buffer backpressure provides a simple and efficient mechanism for stalling flits when downstream buffers are full.

3.3.4 Router Micro-architecture. The NoC router consists of a 2-stage pipeline, which includes credit-based flow control and virtual channels (VC) for QoS and switch allocation efficiency. Each VC has a priority level for allocation, which is based on the look-ahead routing result. The switch allocation is performed in two levels, using a fair round-robin algorithm. To improve the timing and VC allocation, the router employs look-ahead routing for the next hop router. Furthermore, the VC selection is decoupled from the VC allocation to improve timing.

As for VC allocation policy, we use Adjustable VC Assignment with Dynamic VC Allocation (AVADA)[11]. AVADA is a VC allocation policy in which VCs are assigned based on the designated output port of a packet to reduce the Head-of-Line (HoL) blocking, and the number of VCs allocated for each output port can be adjusted dynamically for better VC utilization.

3.3.5 Quality of Service. The QoS support includes several features. Firstly, each flit supports a flit QoS value of 0-15, and the larger the value, the higher the priority. Secondly, fair round-robin arbitration is performed on flits with the same priority, which ensures fairness among the same-priority flits. Thirdly, if an input port is not sent out after several cycles by the flit selected by round-robin, another flit with the same priority will participate in the arbitration, reducing head-of-line blocking and improving arbitration efficiency. Fourthly, support is provided to put the flits that need the same router output port on the same virtual channel first, a feature derives from AVADA policy, which reduces HoL blocking and improves arbitration efficiency. Fifthly, we support assigning a dedicated virtual channel to the flit with the highest QoS priority to ensure the lowest latency and end-to-end message ordering, which can be used for I/O devices and real-time applications. Finally, the QoS value of each flit is given by the requesting node when sending the request, the QoS value-setting strategies are dynamic based on transaction latency and requester throughput, respectively.

4 VERIFICATION METHODOLOGY

Cache coherence verification is essential in ensuring that cache coherence protocols are correctly implemented and performing as designed. There are two main verification methods, each with its own advantages. Simulation-based verification can be used to test the functionality of the protocol in realistic scenarios, while formal verification can be used to exhaustively verify the correctness of the protocol under all possible legal input values subject to assumptions and initialization. By using a combination of both techniques and careful planning, the verification process can be performed efficiently and effectively.

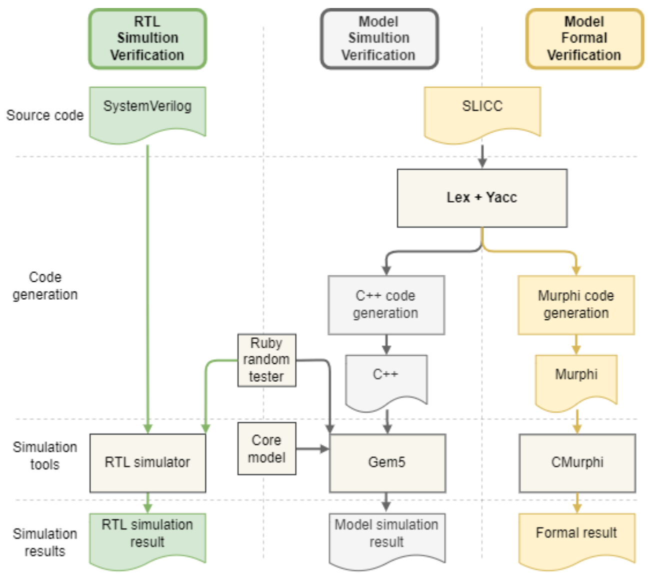

Figure 2: Verification Flow

As shown in Figure 2, the entire verification process consists of three parts, namely RTL simulation verification, model simulation verification and formal verification. In the following of this section, we will introduce the verification framework and the connection of each part.

Build the model and do simulation verification. In the first step, we build our own cache coherence model based on the Gem5 Ruby cache coherence model. Ruby cache coherence uses SLICC Domain-Specific Language (DSL), which mainly describes the finite state machine, input/output, cache replacement algorithm, etc. in the cache coherence controller. In the existing Gem5 flow, the syntax tree will be generated from SLICC through the lexical analyzer Lex and the parser Yacc, and C++ code will be generated and compiled with other parts of Gem5, and the compiled Gem5 simulator will be used for full-system simulation or tested by tools in Gem5.

One of these testing tools is Ruby random tester, which is a cache coherence verification tool that aims to ensure the correctness of the cache coherence protocol implementation. It maintains a check table, each check in the check table maintains a finite state machine that controls its behavior. Each check sends four consecutive stores to a random core to initialize the check and then sends a load instruction to verify the data coherence properties. To detect potential deadlocks, the tester uses a timeout mechanism. If a check's finite state machine cannot change its state for a given time interval, the tester throws a timeout error.

After completing the early verification of our model using Ruby random tester, we can use Gem5's mature system-wide framework to boot Linux and run benchmarks and practical applications.

Formal verification. In our verification flow, we not only use the above simulation verification process for the model, but also hope to perform formal verification on the same code base.

Murphi[3] is a formal verification tool that was originally developed by Professor David Dill's group at Stanford University. It is a state enumeration model checker that is designed to check the correctness of hardware and software systems. The Murphi tool provides a formal verification framework that can be used to prove the correctness of designs, by generating and analyzing all possible system states.

One limitation of Murphi is that it requires a complete and accurate model of the system under test. This means that we need to spend a considerable amount of time and effort in developing a correct and complete model of the cache coherence protocol we want to verify. However, once the model is developed, Murphi can provide powerful insights into the behavior of the system and can help us find and fix potential bugs or issues in the cache coherence protocol.

To address the limitations of Murphi, which requires manual effort to build the model and may have equivalence issues with the original design, some efforts have been done to generate Murphi code from other source code, for example, ProtoGen[9] generates Murphi code from a DSL which specifies directory-based coherence protocols. We found their DSL lack of implementation detail and flexibility, and doesn't have the ability to execute applications. Our approach is to generate the Murphi model from a more detailed and executable specification. We have identified that the core part of both SLICC and Murphi is the description of the finite state machine for each coherence controller, which includes the initial state, state transition, state action and output, and next state decision of the coherence controllers. Thus, it is possible to generate one model from the other.

L1 Cache Store Miss Transition in SLICC Code

```
transition (I, Store, IM_AD) {
        allocateCacheBlock;
        allocateTBE;
        sendGetM;
        popMandatoryQueue;
}
```

L1 Cache Store Miss Transition in Murphi Code

```
Rule "L1Cache_State_I ==> L1Cache_Event_Store ==>
          L1Cache_State_IM_AD"
        ( ( isundefined ( p.tbe.m_TBEState ) ?
         p.entry.m_CacheState : p.tbe.m_TBEState )
         = L1Cache_State_I ) &
        ( L1Cache_GetQueueHead ( p.eventqueue ) =
         L1Cache_Event_Store )
==>
        p.tbe.m_TBEState := L1Cache_State_IM_AD;
        p.entry.m_CacheState := L1Cache_State_IM_AD;
        L1Cache_allocateCacheBlock(n);
        L1Cache_allocateTBE(n);
        L1Cache_sendGetM(n);
        L1Cache_popMandatoryQueue(n);
        L1Cache_Dequeue(p.eventqueue);
 endrule;
```

Figure 3: Example of Murphi Code Generation

Since we already have a Ruby cache coherence model, which is written in SLICC code, we aim to generate Murphi code directly from the same code base. To achieve this, we have modified the SLICC compiler in the Gem5 Ruby model. As shown in Figure 2, the SLICC code is parsed into a syntax tree by Lex + Yacc flow, and other code generation like C++ proceeds on it, so we add a Murphi code generator at this stage, and successfully generate Murphi code from exist SLICC code. This enables us to automatically generate Murphi code from the Ruby cache coherence model, thereby eliminating the need for manual effort and potentially reducing the equivalence problem between the two models. Figure 3 left part is an example of an L1 cache store miss transition, it allocates a cache block and a transition buffer entry (TBE), then it sends a GetM request to the interconnect, and finally it pops the processed message out of the request queue, and Figure 3 right part shows the generated Murphi code from the above SLICC code, it describes the same state transition and called action as the SLICC code.

RTL simulation verification After completing the full verification of the model, we can build our RTL implementation based on this model. We port the C++-based Ruby random tester to SystemVerilog, in order to enable the use of the Ruby tester on the RTL implementation. This allows the tester to be used directly on the RTL implementation, which can be run on both simulators and emulators such as FPGAs and Zebu. Since the process from model to RTL implementation still has to be done manually, in order to strive for equivalence between the RTL implementation and the model in order to migrate our model validation efforts to the RTL implementation, we injected the same sequence of load-store requests into both via the Ruby random tester and compared the logs output from both.

5 EVALUATION

In this section, we present the evaluation of the design described in the previous sections, from the perspective of cache performance and back-end flow result.

5.1 Cache Performance

In our experimental platform, we utilized a core matching Arm A75. This core has the capability of performing up to two load operations and one store operation per cycle to the L1 data cache. The cache configuration used in our platform specified a 32kB size for both L1 data and instruction caches, a 128kB size for the L2 cache, and a 1MB size for the LLC. Additionally, the set-associativity values used were 4 for L1 data, L1 instruction, and L2 caches, and 8 for the LLC.

5.1.1 *Cache Latency and Bandwidth.*

To evaluate the latency and bandwidth characteristics of the cache system, we conducted benchmarking using the "bw mem" and "lat mem rd" programs provided in the lmbench suite. The lmbench benchmark measures the latency and bandwidth of different cache operations, including read and write accesses to the cache.

Figure 4 (a) shows the result of "lat mem rd" with different memory access sizes ranging from 16 KB to 8192 KB. Figure 4 (b) shows the results of the "bw mem" benchmark with different memory access sizes ranging from 32 KB to 8192 KB. From the Figure 4 (a) and (b), we can see there are three major latency increases and bandwidth drops. This is because when the memory access size is small, the data can fit entirely in the higher-level memory, which has lower latency and higher bandwidth. However, as the memory access size increases, the data goes into lower-level memory, leading to higher-level memory misses and leads to lower bandwidth and higher latency.

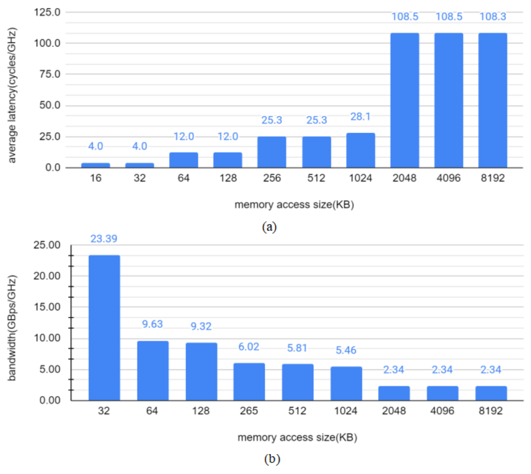

Figure 4: Lmbench Cache Benchmark Results: (a) Latency; (b) Bandwidth

Table 2: Murphi Formal Verification Results

Protocol	Core number	State number	Time	Result
MI	1	121	8.46s	No error
	2	276111	64.85s	No error
	3	123768664	61783.27s	No error
MSI	1	5596	8.89s	No error
	2	22741	10.71s	Deadlocked
	3	99504	21.63s	Deadlocked

5.2 Formal Verification

We generate Murphi code from Gem5 Ruby SLICC code and compile it to an executable using Cmurphi[2]. In our experiment, we performed Murphi code generation and formal verification on two Ruby models that come with Gem5, namely MI and MSI protocol. The number of states, time, and results generated by formal verification are shown in Table 2, which shows that the MSI protocol has a deadlock in the case of multi-core, which exists because the MSI protocol is based on a point-to-point network. When its network is configured as a multi-hop NoC in this paper, the private cache write-back operation and directory snoop operation can easily conflict and cause deadlocks.

In addition, we can see from the verification time that as the number of cores increases, the number of states that formal verification needs to traverse and the corresponding running time increase significantly. The verification time required for systems with more than three cores is unbearable. Therefore, we can consider:

- Simplify the formal verification model, reduce the state, and use the symmetry of the model to prune[8];
- Fractal approach can be used to design our coherence system, and the reliability of the larger system can be deduced from the verification of the subsystems[12];
- Multi-threading and hardware acceleration can be considered to increase the speed of running the formal model[10].

5.3 ASIC Prototype

We have taped out the ASIC prototype on a 6nm process with industry collaborators. As shown in Figure 5, the system configuration of our ASIC prototype is 1 Home Tile with 512 KB LLC and 100 KB system directory, and 8 Compute Tile each with 32 KB L1 instruction cache, 32 KB L1 data cache, and 128 KB L2 cache. Tiles are connected via a 3x3 mesh NoC.

The clock frequency meets 1GHz with minimal effort. The total area is $4.52mm^2$, the area of each compute tile is about $0.51mm^2$ and the area of each home tile is about $0.46mm^2$.

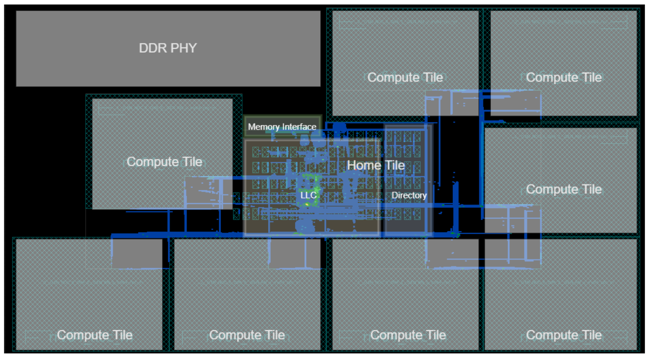

Figure 5: ASIC Prototype Layout

6 CONCLUSION AND FUTURE WORK

This work presents the framework of a verified many-core system consisting of four tiles connected by a 2D mesh NoC, and goes through a verification process combining simulation and formalization from model to RTL.

As future work, we list five major directions: exploring higher performance coherence system implementation and more efficient cache allocation mechanism; analyzing the bottleneck of NoC traffic, optimizing the bandwidth allocation, routing algorithm, and QoS mechanism; conducting large-scale system-level emulation to test the performance of the entire system; exploring ways to accelerate formal tools, including multi-threading and hardware acceleration; trying the backend implementation under the open EDA flow and use the open PDK for tape-out[6]. We plan to open-source the entire work when it is complete.

REFERENCES

[1] Ampere Computing. 2021. Ampere altra max 64-bit multi-core processor features. *Ampere Computing*.
[2] Giuseppe Della Penna, Benedetto Intrigila, Igor Melatti, Enrico Tronci, and Marisa Venturini Zilli. 2004. Exploiting transition locality in automatic verification of finite-state concurrent systems. *International Journal on Software Tools for Technology Transfer*, 6, 320–341.
[3] David L Dill. 1996. The mur ϕ verification system. In *Computer Aided Verification: 8th International Conference, CAV'96 New Brunswick, NJ, USA, July 31–August 3, 1996 Proceedings 8*. Springer Berlin Heidelberg, 390–393.
[4] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.
[5] Natalie Enright Jerger, Tushar Krishna, and Li-Shiuan Peh. 2017. On-chip networks. *Synthesis Lectures on Computer Architecture*, 12, 3, 1–210.
[6] Andrew B Kahng and Tom Spyrou. 2021. The openroad project: unleashing hardware innovation. In *Proc. GOMAC*.

[7] Vijay Nagarajan, Daniel J Sorin, Mark D Hill, and David A Wood. 2020. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 15, 1, 1–294.

[8] C Norris Ip and David L Dill. 1996. Better verification through symmetry. *Formal methods in system design*, 9, 41–75.

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

[10] Ulrich Stern and David L Dill. 2001. Parallelizing the murϕ verifier. *Formal Methods in System Design*, 18, 117–129.

[11] Yi Xu, Bo Zhao, Youtao Zhang, and Jun Yang. 2010. Simple virtual channel allocation for high throughput and high frequency on-chip routers. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–11.

[12] Meng Zhang, Alvin R Lebeck, and Daniel J Sorin. 2010. Fractal coherence: scalably verifiable cache coherence. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 471–482.