A framework for Directed Test Generation and Validation for Cache Coherence Protocol Implementations

Abhinaba Chakraborty

A framework for Directed Test Generation and Validation for Cache Coherence Protocol Implementations

Dissertation submitted in partial fulfillment of the requirements for the degree of

Master of Technology
in
Computer Science

by

Abhinaba Chakraborty
[Roll No: CS2109]

under the guidance of

Ansuman Banerjee
Professor
Advanced Computing and Microelectronics Unit
Indian Statistical Institute, Kolkata

Arindam Mallik
Department Director
Computer System Architecture Group, IMEC, Belgium

June 2022

CERTIFICATE

This is to certify that the dissertation titled **"A Framework for Directed Test Generation and Validation for Cache Coherence Protocol Implementations"** submitted by **Abhinaba Chakraborty** to Indian Statistical Institute, Kolkata, in partial fulfillment for the award of the degree of **Master of Technology in Computer Science** is a bonafide record of work carried out by him under our supervision and guidance. The dissertation has fulfilled all the requirements as per the regulations of this institute and, in our opinion, has reached the standard needed for submission

Date: July 5, 2023

Ansuman Banerjee
Advanced Computing and Microelectronics Unit(ACMU)
Indian Statistical Institute, Kolkata

Arindam Mallik
IMEC, Belgium
Leuven, Belgium

Acknowledgement

I would like to express my heartfelt gratitude to my research supervisor, **Professor Ansuman Banerjee**, Advanced Computing and Microelectronics Unit, Indian Statistical Institute, Kolkata, and **Dr. Arindam Mallik**, IMEC, Leuven, Belgium, for their unwavering support, guidance, and mentorship throughout my research journey. Without their constant support, it would not have been possible for me to complete this thesis. I am grateful for their valuable feedback, insightful discussions, and encouragement that helped me in refining my research ideas and make progress toward achieving my research goals.

I would also like to extend my sincere thanks to the faculty members of the Computer Science Department at ISI, Kolata and members of the CSA group of IMEC, Belgium especially **Vinay B. Y. Kumar** and **Philipp S. Käsgen**, who have provided me with an excellent academic and industrial foundation and imparted the knowledge and skills necessary for my research. Their dedication to teaching and research has been an inspiration to me, and I am grateful for their continuous support, advice, and encouragement. I would also like to thank Professor Prabhat Mishra from the University of Florida for the initial discussions that we had on this work.

I would like to thank my family, including my parents, for their unwavering love and support throughout my academic journey. Their encouragement and motivation have been a constant source of inspiration, and I am grateful for their understanding, patience, and sacrifices. I would also like to acknowledge the invaluable support and guidance I received from my friends in the Computer Science Department. Their valuable feedback, discussions, and insights have been instrumental in shaping my research ideas and helping me overcome the many challenges I encountered during my research. In conclusion, I am deeply grateful to everyone who has supported and guided me throughout my academic journey. I hope that this research will contribute to the field of computer science.

Abhinaba Chakraborty
Roll No. CS2109
Indian Statistical Institute
Kolkata - 700108, India.

Abstract

Multi-core processors have revolutionized computer system design by offering significant improvements in performance. However, as the number of cores increases, the need for an efficient memory management system becomes more critical. Caching has been the most effective approach to reducing memory access time and overcoming the trade-off of space, performance, and energy consumption.

To ensure cache coherence and consistency in a multi-core system, various cache coherence protocols have been introduced. These protocols are designed to keep track of changes made to shared data in the local caches and ensure that all processors have the same view of memory. The increasing complexity of cache coherence protocols and the growing number of processing units in modern architectures has led to a significant challenge for verification teams to verify complex coherence protocol implementations within a tight timeline. In recent years, there has been a growing need for efficient and effective verification methods to ensure the correctness of these protocols.

Verification using simulation and random / constrained-random tests is widely used in industry due to its scalability. However, these tests come at a high cost in terms of time and resources required to cover all possible configurations / states and scenarios that the design under test may undergo. Directed tests are promising to achieve high coverage with a much smaller number of tests. Directed testing involves designing test cases that are specifically targeted towards uncovering particular behaviors or states of the system under test. This approach can be highly effective in reducing the number of tests required to achieve full coverage of the state space.

Recent research on verification of cache coherence has shown that it is possible to achieve full coverage of the state space with a small number of test sequences using directed testing. This approach involves generating a set of directed tests that cover all possible states of the system, while also reducing the number of repeated scenarios. This method has been shown to be highly effective in achieving full coverage of the state space in a much shorter time than traditional simulation-based testing methods. However, a significant limitation of contemporary state of the art methods used for cache coherence verification is that they assume basic memory models that are not suitable for modern architectures and NoC models. Modern architectures and NoCs often have complex memory hierarchies and interconnect topologies that require more sophisticated memory models. Therefore, there is a need for further research to develop directed testing methods that can effectively handle these more complex memory models.

In this dissertation, we propose efficient test generation methods for verification of cache coherence protocol implementations. Our methods can cover all possible protocol transitions and identify potential bugs. This approach can significantly reduce the time and effort required for testing while improving the reliability of the verification process. We also address the problem of modeling of multi-core memory models with two or more levels of private caches and the associated verification problem. Our proposed approach provides a comprehensive framework for verifying cache coherence protocols in complex memory models, which is essential for ensuring the correct and efficient operation of modern computing systems.

To demonstrate the effectiveness of our proposed approach, we conduct experiments using the Structural Simulation Toolkit (SST) and verify the results with our expected outcomes. Experiments show that our approach is capable of identifying potential bugs and ensuring the correctness and efficiency of cache coherence protocols in complex memory models.

Overall, we believe this dissertation makes an important contribution to the field of verification of cache coherence protocols. As the complexity of memory models continues to increase, the proposed test generation and verification approach can be further extended to cover more complex scenarios and ensure reliable operation of modern multi-core systems with complex cache coherence protocols.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The introduction of cache memory has been one of the best architectural innovations to address latency issues in computing systems. Caches work on the principle of exploiting temporal and spatial locality, where frequently accessed data values are stored in small but fast cache memory to reduce the access time to the main memory.

However, as the demand for higher processing speeds and the need for lower power consumption grew, multi-core systems were introduced where each core will have its private-level caches and shared memory. This approach proved to be very effective in reducing memory latency and improving performance further.

Despite the increased complexity, these modern memory models have shown exceptional results in terms of high processing speed, low power consumption, and low space requirement. They have become the backbone of modern computing systems, powering everything from smartphones and laptops to high-performance computing clusters and data centers.

As computing systems continue to evolve, the need for more efficient and complex memory models increases. Researchers and engineers continue to work on developing new and innovative memory architectures that can handle the growing demand for faster and more efficient computing systems while minimizing power consumption and space requirements.

1.1 Cache Coherence Protocols

As the complexity of modern computing systems continues to increase, so does the challenge of ensuring memory consistency. In multi-core systems, multiple cores can access the same memory address space simultaneously, which can lead to data inconsistency and other issues.

To address this challenge, a proper protocol is needed to ensure that each core accessing the same address space receives the most recently updated values. This is where cache coherence protocols come into play.

In multi-processor systems, multiple processors may have access to the same memory or cache, and when one processor modifies a particular memory location or cache line, other processors must be notified of the change to maintain data consistency. Cache coherence protocols are designed to ensure that all processors see a consistent view of memory and cache.

A ***cache coherence protocol*** is a set of rules and mechanisms that ensure that all caches in a multi-core system have consistent and up-to-date data. The protocol ensures that each cache knows about the changes made by other caches in the system and that each core always accesses the most recent version of the data.

The **snoopy protocol** is one of the most widely used cache coherence protocols. In this protocol, each cache line has a state associated with it, and every cache controller monitors the state of every cache line in the system. When a processor wants to read or write a particular memory location, it first checks its cache. If the cache line is present in the cache, the processor can directly access the data. However, if the cache line is not present, the processor sends a request to all other processors in the system. If any processor has the requested cache line in the modified or exclusive state, it must invalidate the cache line and send it to the requesting processor. Once the requesting processor receives the cache line, it can modify it and update its state to the modified state, which is then broadcast to all other processors in the system.

Directory-based coherence protocol is another type of cache coherence protocol that is used in large-scale multiprocessor systems. In this protocol, a centralized directory maintains the coherence state of each cache line in the system. Each cache controller has a copy of the directory and uses it to track the coherence state of its cache lines. When a processor wants to read or write a particular memory location, it sends a request to the directory. The directory checks the coherence state of the cache line and grants the request if the cache line is in the shared state. If the cache line is in the modified or exclusive state, the directory sends a message to the processor holding the cache line to invalidate it and transfer it to the requesting processor.

Hybrid coherence protocols combine the features of both snoopy and directory-based protocols. In this protocol, a small directory is used to maintain the coherence state of frequently accessed cache lines, while the snoopy protocol is used to maintain coherence for the remaining cache lines. The hybrid protocol provides the benefits of both protocols, including low latency and high scalability.

1.2 Verification of Cache Coherence Protocols

Cache coherence protocols are critical components of modern computer systems, especially those that have multiple processors. In such systems, multiple processors can access the same memory locations, which can result in inconsistent data if cache coherence protocols are not implemented correctly. Hence, verification of cache coherence protocols is crucial to ensure that the system maintains data consistency. For the verification task, the protocol is typically modeled as a finite state machine, one for each cache block, for each private cache. A global state machine arises from the composition of the state machines across the processors.

Formal verification is a rigorous mathematical approach that is used to verify the correctness of a system's design. In the context of cache coherence protocols, formal verification techniques use mathematical models to verify the correctness of the protocol's behavior.

Simulation-based testing is another approach to verify cache coherence protocols. In this approach, the protocol is implemented in a simulation environment, and a set of test scenarios is executed to check the correctness of the protocol's behavior. The test scenarios should cover all possible transactions that the protocol may encounter during run-time. The simulation results are analyzed to ensure that the protocol correctly maintains data consistency in all scenarios.

Directed testing is a type of testing that involves analyzing the state space model of the possible transitions allowed by a cache coherence protocol and performing graph traversal on those state space models. Directed testing typically involves the use of automated tools to traverse the global state machine of the cache coherence protocol and identify potential issues.

Hardware implementation testing involves testing the cache coherence protocol on actual hardware. This approach is useful to verify that the protocol functions correctly on the target platform. In hardware implementation testing, the protocol is implemented on a test chip, and a set of test scenarios is executed to check the correctness of the protocol's behavior. The test scenarios should cover all possible transactions that the protocol may encounter during run-time. The hardware implementation is then tested to ensure that it correctly maintains data consistency in all scenarios.

It is worth noting that the verification process for cache coherence protocols is a challenging task due to the complexity of the protocols and the large number of possible scenarios that they may encounter during run-time. Furthermore, the verification process must consider the impact of various system parameters, such as the number of processors, the size of the cache, the coherence granularity, and the access patterns on the protocol's behavior. To ensure that the protocol functions correctly and guarantees data consistency in all possible scenarios, verification techniques are necessary.

1.3 Motivation of this dissertation

Cache coherence protocols have become increasingly complex as multi-processor systems have become more prevalent. The challenge in cache coherence protocol verification lies in the fact that these protocols are inherently distributed and concurrent, making it difficult to test all possible scenarios. Directed testing is a popular method for verifying cache coherence protocols, but it has limitations in terms of scalability and completeness. As a result, there is a need for a robust test generation framework that can handle complex cache coherence protocols efficiently.

In this dissertation, we primarily focus on directed testing for each cache coherence protocol. Recent research in directed testing has shown that it is possible to generate test sequences that cover every aspect of the protocol's behavior. However, these test sequences are based on a simplistic view of the memory models and caching mechanisms, which may not accurately reflect the behavior of modern complex systems.

To successfully incorporate directed testing into cache coherence protocol verification for modern complex systems, we need to have a robust test generation method that can handle the complexity of these systems while ensuring that testing is complete in a reasonable time, using moderate space for test storage and management. This is one of the problems that we address here.

Another area where research is needed is multilevel caching verification. In modern multi-processor systems, it is common to have multiple levels of caching, each with its cache coherence protocol. The interactions between these protocols can be complex, and verifying their correctness can be challenging. There is a need for research to develop verification techniques that can handle these interactions efficiently and accurately.

1.4 Contributions of this dissertation

The objective of this thesis is to propose efficient test-generation algorithms for validating the implementation of cache coherence protocols for complex memory models. Our contributions to this thesis are described below.

- *Efficient Test Generation for the SI Protocol*: Considering the most simplistic cache coherence protocol SI, we proposed, implemented, and tested a method that can cover the entire state space of the global state machine using few test sequences without compromising the space requirement. We then extended the framework to other complex coherence protocols (e.g. MSI, MESI) that are built on top of SI.
- *Verification of coherence protocols in multi-level caching*: Considering complex memory models, we have proposed, implemented, and tested a framework that can successfully verify these complex memory hierarchy models.
- *Comparative Analysis of Random Testing and Directed Testing*: Considering different memory models, we show a comparative analysis of the framework proposed by us and a random testing framework. Also, we show a case study of a bug where directed testing shows good promise to trigger the bug with shorter traces as compared to a random tester.

1.5 Organization of the dissertation

This dissertation is organized into 5 chapters. A summary of the contents of the chapters is as follows:

Chapter 1: This chapter contains an introduction and a summary of the major contributions of this work.

Chapter 2: This chapter corresponds to the background and prerequisites of the work for all the topics discussed.

Chapter 3: This chapter presents a framework for verifying SI, MSI, and MESI for a single level cache hierarchy.

Chapter 4: This chapter presents a framework for verifying SI, MSI, and MESI for multiple cache levels in the memory hierarchy.

Chapter 5: We summarize with conclusions of this dissertation and possible future work.

Chapter 2

Background and Related Work

In this chapter, we aim to provide readers with an overview of the key concepts and background knowledge necessary for cache coherence discussed in this dissertation. This includes a discussion on the fundamental principles of caches and their underlying technologies, as well as the methods and approaches used to design, implement, and evaluate them.

To begin, we provide an overview of the core concepts of the systems, including their basic structure, operation, and purpose. We also discuss the key technologies that underpin these systems. Finally, we discuss the key related work done till now by other researchers and the limitations of their work.

2.1 Caches

The main memory (physical memory) is a large DRAM array which is slow. It is impractical to use it in modern systems as it is. To address the latency issue we create fast SRAM *caches* to keep frequently accessed data close to the processor. Utilizing the concept of temporal locality, we can store a subset of the program's address space in the cache, where most of the data can be found. This structure is small, fast, and power efficient.

Temporal locality is a concept that states that if a resource is accessed at some point, most likely it will be accessed again within a short period of time. *Spatial locality* is a concept that states that if a resource is accessed at some point in time, most likely similar resources will be accessed again shortly. By running different benchmarks[*] on different memory systems, designers established the above-stated memory access patterns. This information can then be used to design memory systems that are optimized for these access patterns, such as by using different memory technologies, cache configurations, or memory controllers. A common design for a cache hierarchy is shown in Figure 2.1.Caches deal with data in the form of blocks instead of individual bytes. Each block is fetched or evicted as a single unit to take advantage of the principle of spatial locality.

2.2 Cache Hierarchy

A typical early organization of cache was to keep a fast SRAM L1 cache between the processor and memory, one each for instruction and data. As the benefits of caches became obvious, hierarchies of

[*]They are widely used to evaluate the performance of computer systems, e.g. SPEC2006 benchmark

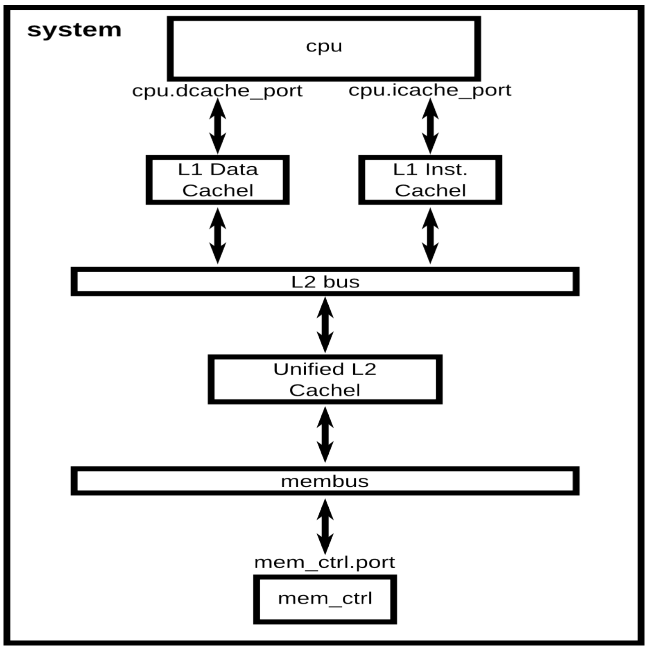

Figure 2.1: Cache Hierarchy

each cache came into proposal. Figure-2.1 shows one popular organization [4] with the L1 and L2 cache. The L2 cache can contain both data as well as instructions. At this level, we do not keep a separate cache for data and instructions. The L2 cache typically is larger (roughly about 256 KB to 8 MB) and slower than the L1 cache.

The cache access mechanism involves the initial access of the processor to the L1 caches, which consist of the instruction cache (i-cache) and the data cache (d-cache). Due to the principle of temporal locality, a significant portion of memory accesses is expected to result in cache hits. In case of a cache miss, data needs to be fetched from lower levels of the memory hierarchy. The L1 caches are designed to be very fast and can often provide data in 1-3 clock cycles. On the other hand, L2 caches are slower and can take 10-50 cycles to execute an operation. This is still significantly faster than accessing data from the main memory, which typically takes 200-400 cycles to complete an operation. Modern computers often include another level of cache in their hierarchy, reducing memory access latency even further.

2.3 Organisation of Cache

The cache holds a subset of addresses that are being used frequently. A cache is organized into S sets which can hold one or more than one block of data and each memory address maps to exactly one set in the cache. Some of the address bits are used to determine the set containing the data. If more than one block is contained in a set, the data may be kept in any of the blocks in the set.

2.3.1 Direct Mapped Cache

A direct mapped cache consists of a set of cache blocks, where each block contains one block of data from the main memory. In a direct mapped cache, each memory address is mapped to a single cache block, determined by the address's lower-order bits. The number of sets in a direct mapped cache is

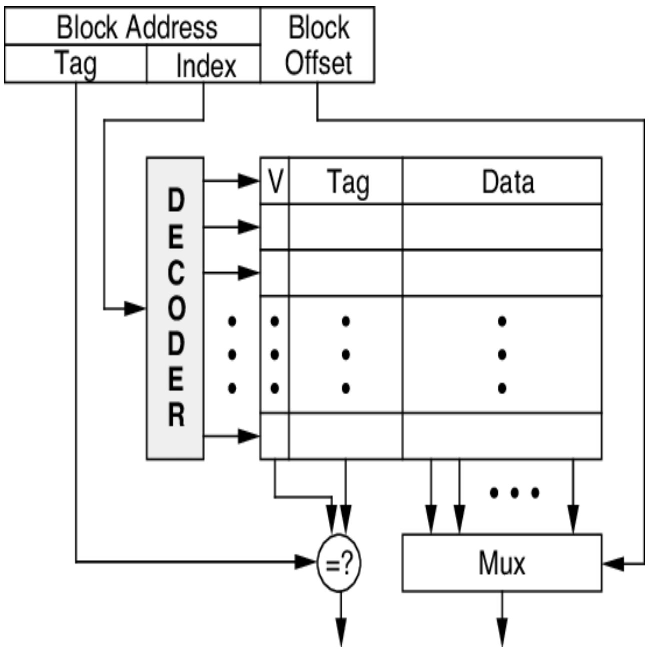

Figure 2.2: Direct Mapped Cache

equal to the number of cache blocks. As a result, the mapping of memory addresses to cache blocks wraps around after S memory addresses, where S is the number of sets in the cache.

Since multiple memory addresses can be mapped to the same set, the actual address of the data in each set must be tracked. This is achieved by dividing the address into two parts: the tag and the set index. The set index, which comprises of the lower-order bits of the address, is used to determine which set the data is located in. The tag, which consists of the remaining higher-order bits of the address, is used to specify the actual memory address stored in the cache block.

The use of a direct mapped cache offers a simple implementation with low hardware overhead. However, it suffers from the problem of conflicts, where multiple memory addresses map to the same set, leading to frequent cache misses and reduced cache hit rates. We have to search all entries or *cache blocks* or *cache lines*.[†] But searching all the lines is too slow. Consider an example with 1024 entries. 1024 cache lines can be identified by $\log_2 1024 = 10$ bits, so we extract 10 least significant bits (LSB) from the block address bits and use them to access the corresponding entry. The remaining 16 MSB bits of the block address are used to distinguish between the separate addresses. Figure-2.2 illustrates this [10]. For a 32-bit address space, each address is divided into three parts,

1. First 16 bit **tags** which are used to distinguish between two memory addresses.

2. Next 10 bit **index** which are used to extract the cache line number into which the memory address will map.

3. Last 6 bit **Offset** which are required for uniquely indexing a byte within a block.

This type of cache organization is simple. Here accessing data and tags arrays can be done in parallel. But here the miss rate is quite high.

[†]Each entry in the cache is called a cache line. The terms cache block and cache line are used interchangeably.

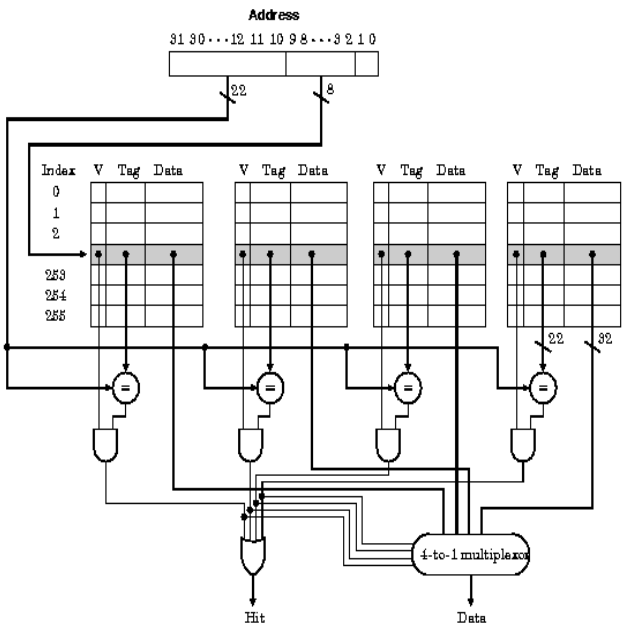

Figure 2.3: Set Associative Caches

2.3.2 Fully Associative Cache

A fully associative cache consists of a single set with B ways, where each way contains a block of data from the main memory. In a fully associative cache, any memory address can be mapped to any of the B cache blocks, and a tag comparison is made for each way in the set to determine if the requested data is present in the cache.

To perform a tag comparison, the address is divided into two parts: the tag and the index. The tag specifies the actual memory address, and the index is used to select the appropriate setting in the cache. If a cache hit occurs, a multiplexer chooses the appropriate data from the selected way.

Fully associative caches typically have the fewest conflict misses for a given cache capacity because each block can be placed in any cache location. However, fully associative caches require more hardware for tag comparisons, making them best suited for small caches where the number of ways is limited. Additionally, fully associative caches are more complex to implement than direct-mapped or set-associative caches due to the need for tag comparison across all ways.

2.3.3 Set Associative Cache

Direct mapped Caches have low access time but low hit ratio[‡] but fully associative caches have a high hit ratio at the expense of higher access time. To use the best of both designs, a K-way Set Associative Caches are introduced. The tag array is divided into groups of equal-sized sets which often contain 2, 4, and 8 blocks. Figure 2.3 shows an example. Consider a 1024-entry cache organization.

Let us assume each set has 4 cache lines. Total number of sets $= \frac{1024}{4} = 256$. The total number of bits required to index each set is $= \log_2 256 = 8$. 8 LSB bits of the block address are used to find the index of sets. The remaining $26 - 8 = 18$ bits are used for tags. To read/write a data block, at first,

[‡] *Hit Ratio* is the ratio between the total number of cache hits to the total number of requests incoming to the cache

the address is sent to the tag array. After computing the set index, we access all entries belonging to that set in parallel. In our case, we read out 4 tags and compare each of them. If there is no match we have a cache miss, otherwise we can read the data array.

2.4 Basic Cache Operations

The basic operations that the cache has to support are mentioned below.

1. ***Lookup*** operation is used to find if the cache contains a data or not. If not, the cache sends a *cache miss* signal to the lower level of the memory hierarchy.
2. ***Replacement*** operation replaces an existing address from the cache line in the case some other memory block maps to the same cache line or in case the cache is full.
3. ***Eviction*** happens when an existing address is forced to give up the ownership[§] of the cache line.
4. ***Write*** operation helps to write new data at a particular cache line. If there is a cache miss, the cache has to fetch the address first and then write on it. One of the two schemes to follow when it comes to writing into the cache line,
   (a) **Write through scheme:** Whenever the processor performs a write operation, the updated data is propagated to the lower level of the memory.
   (b) **Write-back scheme:** Whenever the processor performs a write operation, the updated data does not propagate to the lower level. An *modified bit* is kept to keep track of whether the cache line has been updated or not. In case of an eviction[¶] the updated value is propagated to the lower level of the memory hierarchy. In a modern computing system, write-back caches are used as these caches reduce a lot of traffic in the system bus.

2.5 Cache Replacement Strategies

The most commonly used cache replacement strategies are discussed now. A Random Replacement strategy is simplest but not effective. When a conflict occurs in a direct mapped cache, it means that two or more memory addresses map to the same cache line. As each cache line in a direct mapped cache can only hold a single memory address, there is no other option but to evict the existing cache line and replace it with the new address. This is known as a conflict miss. The process of eviction in a direct mapped cache is simple, as there is only one cache line per set. When a conflict miss occurs, the cache controller replaces the existing cache line with the new data and updates the tag associated with the set to reflect the new address. For K-way set associative or fully associative caches, in a case when the set is full (for fully associative caches the cache needs to be full) a block is chosen by the *LRU (Least Recently Used)* policy. Due to the implementation difficulty of true LRU, Caches employ a *pseudo-LRU* policy. This policy is not perfect but approximates LRU in the best possible way.

[§]Giving up on the ownership simply means another address is mapped to the same cache line and so the former address has to leave the cache line

[¶]An eviction can happen due to various reasons, mainly because of replacement

2.6 Multi core Systems and Shared Memory Multiprocessors

Advancements in technology have pushed processor designers to integrate multiple processor cores on a single chip over the last decade. While in the past, multi-processor systems were only found in high-end computing environments, chip multiprocessors (CMPs) are now prevalent in most modern publicly available computers.

The rise of CMPs can be attributed to several technological barriers that existed in the past, such as power consumption and heat dissipation. As clock speeds approached physical limits, increasing the number of cores on a single chip became a more viable way of improving computational performance. Additionally, improvements in manufacturing processes and materials science have allowed for more efficient and compact chip designs, enabling the integration of multiple cores on a single chip.

By incorporating multiple processor cores on a single chip, CMPs offer several benefits over traditional single-core processors, such as increased parallelism, improved performance, and reduced power consumption. Furthermore, CMPs can be designed to scale to meet the demands of a wide range of applications, from low-power mobile devices to high-performance computing systems.

In multicore systems, some individual processing cores share a large memory space. Each unit has its own private cache. A subset of memory blocks can be stored by each unit. This type of organization may lead to data inconsistency. Let us consider a memory location X which has a value 10 located in the main memory. Two cores $C1$ and $C2$ load this value from main memory into their respective caches. Now after some time, core $C1$ updates the value of the block to 15. Now two cores have copies of the same memory address but with different values which should not happen. Maintaining coherence across caches is the challenge that cache coherence protocols aim to address, and verification of such protocols is what we aim for in this work.

2.7 Consistency and Coherence

Shared memory systems offer benefits such as high performance, low power consumption, and low cost. However, ensuring memory correctness is crucial for their hardware implementation. The problem of achieving memory correctness can be divided into two sub-problems: consistency and coherence.

2.7.1 Consistency

The consistency memory model ensures correctness in shared memory systems by providing rules for memory reads and writes. In single-core processors, these rules guarantee that the execution of a thread produces a single correct output state. However, in multi-core systems, multiple threads running concurrently can result in many correct interleavings of instructions, leading to many correct executions and many more incorrect ones.

To ensure that programmers know what to expect and system designers know the limits of what they can provide, correctness must be defined to specify the allowed behavior of multi-threaded programs executing with shared memory.

Several consistency models, cited from stronger to weaker, are found today, as mentioned below

1. Sequential consistency is a memory model that specifies that a multi-threaded execution should

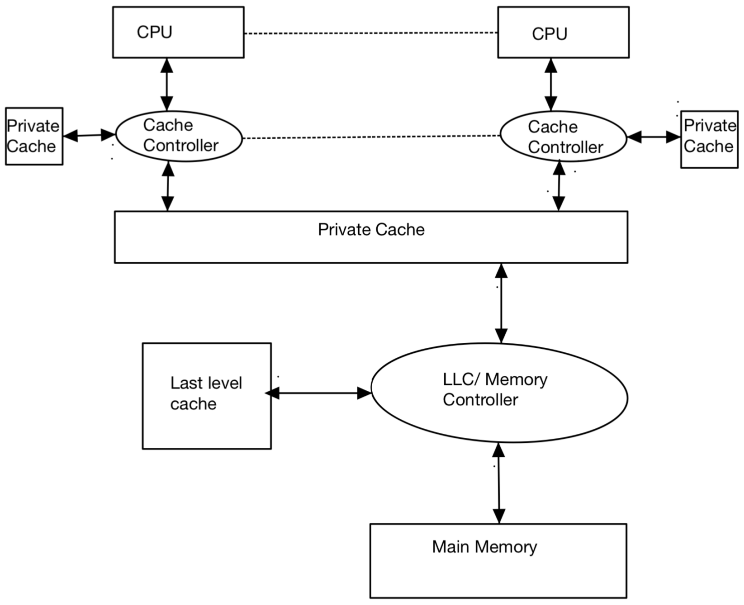

Figure 2.4: Reference Memory Model

appear as an interleaving of the sequential executions of the individual threads as if they were multiplexed on a single-core processor. It is implemented, for example, in the MIPSR10000 architecture.

2. Total Store Order (TSO) is a memory model used in computer architecture that allows for the use of FIFO write buffers to hold the results of committed stores before the effective write to the caches. This optimization ensures that stores are executed in order but does not guarantee the order of loads. TSO is implemented on x86 and SPARC processors, among others.

3. Relaxed consistency allows for relaxed constraints on memory orderings by using specific instructions for load/store barriers and writes buffer flushes. The programmer only needs to ensure that the synchronization flag is properly set after data is updated, rather than enforcing a specific ordering of updates.

2.7.2 Coherence

Coherence is not mandatory but is often implemented in shared memory systems to ensure that multiple processor cores accessing the same data do not cause conflicts. These conflicts can occur when at least one core is overwriting the data.

Coherence protocol prevents accessing stale data in shared memory systems. It consists of rules implemented by different actors within the system to make caches functionally invisible to programmers. The goal of coherence is to ensure that caches do not enable any new or different functional behavior in a shared memory system.

2.8 Cache Coherence Protocols

In a shared memory system, multiple processors, DMA engines, and external devices may have access to the same data simultaneously. If one of these actors modifies the data, other actors may not immediately see the update, leading to an incoherent view of the memory. This is because each actor has its cache or buffer, which may contain a stale copy of the data.

To provide a formal definition of coherence two invariants may be used, based on what is observed as incorrect: [7]

- The *Single Write Multiple Read* (SWMR) invariant states that at any given time, a memory location can be written by only one processor, but several processors can read it. It only needs to be maintained in logical time and can be divided into epochs.
- The data invariant guarantees that the value of a memory location remains consistent throughout its lifetime, from its last write epoch to its end, ensuring all processors read the same value during read epochs.

The majority of coherence protocols, called ”invalidate protocols” are designed to maintain these invariants.

Coherence is often maintained at the cache block level, even though processors perform reads and writes with different sizes. This ensures that two processors cannot write to different parts of the same block simultaneously, following the Single-Writer-Readers invariant.

Cache Coherence applies to all storage components holding shared address space, such as caches, main memory, instruction caches, and TLB. It is not dictated by the consistency model, which instead relies on coherence for correctness.

Two coherence invariants are maintained through a distributed system of storage components, each associated with a Finite State Machine (FSM) called a coherence controller. These components exchange messages to ensure that the invariants are always maintained for each block.

The coherence protocol specifies the interactions between the finite state machines. The coherence controllers have some responsibilities. The cache controller processes requests from two sources,

1. The coherence controller in a cache coherence system is associated with each storage component, and it communicates with other coherence controllers to maintain the two coherence invariants. When a processor performs a load or store operation, it sends the request to its cache, which then checks if the requested data is already present in the cache. If the data is not present, the cache coherence controller starts a coherence transaction by issuing a coherence request, such as requesting read permission, for the desired block to one or more coherence controllers responsible for another storage component
2. The coherence controller receives coherence requests and coherence responses from other coherence controllers through the interconnection network.

A transaction consists of a request and all the messages needed to satisfy the request. The type of transactions and messages depends on the protocol specifications.

The memory controller usually processes requests from the interconnect network. Other components may act as a cache controller or a memory controller depending on their requirements. Figure-2.5 shows one possible design.

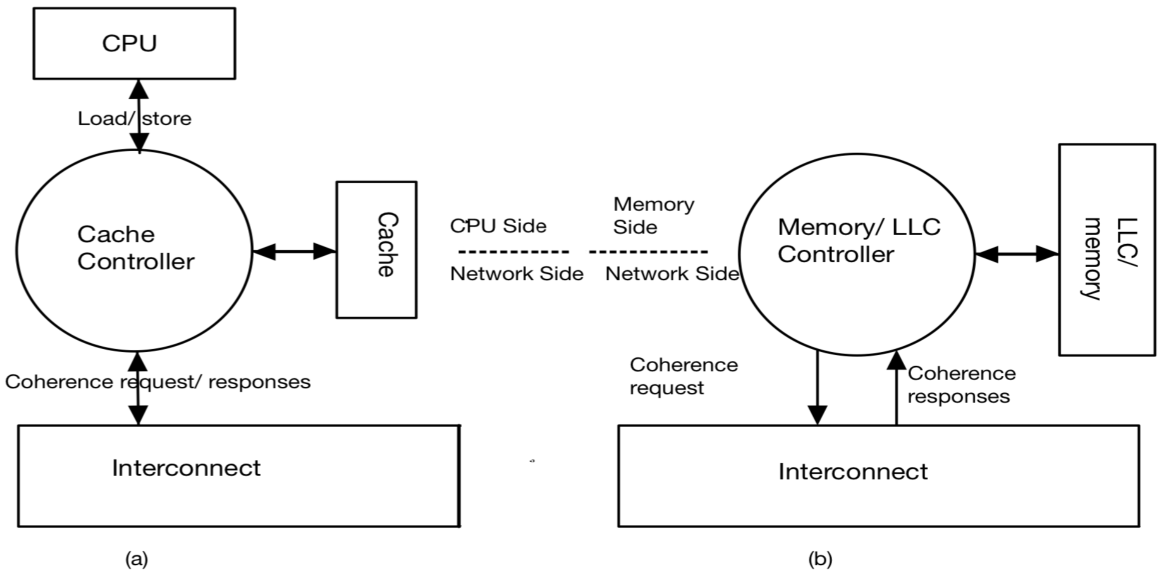

Figure 2.5: Cache Controllers and Memory Controllers

A coherence controller is a finite state machine that receives and processes events depending on the block state. For an event type X to block Y, the controller takes actions that are functions of X and of block Y state and may change the state of Y.

2.8.1 Coherence Protocol Design Space

Cache coherence protocol designers choose the states and transactions for each coherence controller, which are protocol-dependent, while stable states and transactions are protocol-independent. Events, transitions, and transient states are highly dependent on the chosen protocol. Cache blocks have four characteristics that are worth encoding in their states,

1. ***Validity***: A valid block contains an up-to-date data value and can be read by any processor. However, it can only be written by a processor if it has exclusive ownership of the block.

2. ***Dirtiness***: A dirty block has a data value that differs from the value in memory and has not been updated in memory yet. The cache controller is responsible for ensuring that the updated value is eventually written back to memory. A block that has the same value in the cache as in memory is called a clean block.

3. ***Exclusivity***: A block is exclusive when it is the only privately cached copy of that block in the system.

4. ***Ownership***: A block is owned by the cache controller if it is responsible for responding to coherence requests for that block. This block cannot be evicted without giving ownership to another coherence controller.

2.8.2 Cache Stable States

There are three main defined states for any coherence protocol, [7] as below.

1. ***M(modified)***: The block is valid, exclusive, owned, may be dirty, and may be written or read. The cache has only a valid copy of the block and it is potentially stale in the memory. The cache is responsible for the requests for the block.
2. ***S(hared)***: The block is valid but not exclusive, not dirty, not owned, and is read-only. The other caches may hold valid, read-only copies of the block.
3. ***I(nvalid)***: The block is invalid. Either the cache doesn't hold the block or it holds a stale copy that should not read or write.

Other than those states, in 1986, Sweazy and Smith [11] introduced a five-state MOESI model with two additional stable states,

1. ***O(owned)***: A shared block is valid, owned, and possibly dirty, but not exclusive. It can be read but not written to, and other caches may have a read-only copy. It may be potentially stale in memory and is the responsibility of the cache for requests.
2. ***E(xclusive)***: The block is valid, owned, exclusive, clean, and read-only. It is up-to-date in memory.

2.8.3 Cache Transient States

In cache coherence protocols, stable states are the states in which there is no ongoing coherence activity for a particular block, and they are typically used to describe the behavior of a protocol. On the other hand, transient states occur during the transition from one stable state to another stable state, and they are denoted as AB^C, where A and B represent the stable states, and C is the event that is required to complete the transition. These transient states are often implementation-dependent and vary between different coherence protocols. As transient states are numerous and do not concern all blocks but only those that have pending transactions, they are maintained with specific hardware structures, miss status handling registers (MSHR)[*], used to track pending transactions.

2.8.4 Last Level Cache(LLC)/Memory States

Block states in the LLC and memory can be named by two general approaches, whose choice doesn't affect functionality or performance:

1. ***Cache-centric***: The state of a block in the last-level cache (LLC) or memory is determined by the aggregation of the block states in all caches that hold a copy of that block. If a block is exclusively present in all caches in the Invalid(I) state, then it is also considered to be in the Invalid state in the memory. For this thesis, only the states of the block in the caches are considered.

[*]The size of MHSR is 1 to 2 entries

2. ***Memory-centric***: The block state reflects the permission status of the memory controller for that block. If a block is present in all caches in the "Invalid" state, then it is considered to be in the memory in the "Owned" state, because the memory is responsible for providing the up-to-date value of the block.

As it is not feasible to track the memory status of every block in the system, many multi-processor systems use an inclusive Last Level Cache (LLC), which maintains a copy of every block cached anywhere in the system. Therefore, the memory does not need to maintain states, and the state of a block in memory is the same as that in the LLC.

Many possible protocols can be designed from the same set of states and transactions as it is not possible to provide an exhaustive list of possible events and transitions for each coherence controller. However, two main decisions have a major impact on the rest of the protocol. For both these designs, making hybrids are possible. These are mentioned below.

2.9 Snooping Coherence Protocols

Snooping protocols were the first widely deployed class of coherence protocols and offer several attractive features such as low-latency transactions and a simple design. In such protocols, all the coherence controllers snoop the requests and process them in the same order. Thus, they all observe the same scenario and they can update correctly their finite state machines. They order broadcast networks such as buses, which are thus required to ensure this property. These protocols create a total order of coherence requests across all blocks, even though only a per-copy order is required. However, total order implements some consistency models simply.

Requiring total order has implications on the interconnection network as it must determine the order of requests. This is done by the serialization point which broadcast the messages to all controllers. The issuing controller then learns where the requests have been ordered, maybe several cycles after the issuing of the request. In the case of a shared bus, an arbitration logic can be used to ensure that only a single request is issued at a time, which acts as a serialization point.

The key aspect of snooping protocols revolves around the requests. Response messages can travel on a separate network that does not need broadcast or ordering capabilities. As they carry data, they are longer than requests and there are benefits to sending them on a simpler lower-cost network. The time interval between the request appearing on the bus and the response reception does not affect the serialization of the transaction.

2.9.1 SI Snooping Protocol

The simplest coherence protocol that can be designed keeps each block in two possible stable states: *Invalid (I)* and *Valid (S)*. On the memory side, the I state means that the block is absent from all caches, and S means that one of the caches holds the block in the S state. It should be noted that here the cache is a write-through scheme[†].

[†]Because of the write-through scheme we don't need to store if the block is modified or not.

2.9.2 MSI Snooping Protocol

The network is supposed to be shared bus/ connected via a router, caches are supposed to be write-back and cache-centric notion is used. When the CPU makes a read request the transition from the I state to the S state happens. In this case, the cache controller sends a read miss message and gets a copy of the block.

In the S state, the CPU can read the block as many times as required. In those cases that will be a cache hit. But the CPU is not allowed to write into the block. To write into the block, the cache controller has to put a write miss message (or write permission message) and wait to get the permission. In the meantime, other cores which share / modify the block have to invalidate and make a transition from S or M state to I state respectively. Note that only one core is allowed to be in M state for the particular block but multiple cores can be in S state.

2.9.3 MESI Snooping Protocol

The MESI protocol adds an extra E(*exclusive*) state. The state transitions for M, S, and I states remain mostly the same. The E states indicate that if a cache has a valid copy of the block in S mode and that cache is the only cache that is sharing that, it should be a E state. The advantage of adding an E state is in case of writing into the block and if the cache is in E for that block, the cache will know it is the only sharer, so it can silently upgrade to M state without informing others. MESI protocol reduces the traffic on the shared channel.

2.9.4 Other Snooping Protocols

Due to the limitations of MSI and MESI (like a high number of write-back messages and the problem of arbitration[‡]), there are other protocols introduced. Two popular ones are:

1. MOSI Protocol
2. MOESI Protocol

To implement these protocols at the hardware level some bits from *tag bits* are borrowed and are to be used to encode the states based on a particular protocol.

2.10 Directory-based Coherence Protocols

Snooping protocols are simple and offer good performance because each transaction can be completed with two messages. However, ordered broadcast networks are costly, and hardly scale to a large number of processors. To address this lack of scalability, directory protocols were introduced. These avoid both the ordered broadcast network and having each coherence controller process every request.

Directory protocols use a directory that maintains the list of caches that hold each block and the state of each block. A cache controller issues its requests to the directory which determines what

[‡]The process of choosing one entity among a plurality of interested entities is known as arbitration. [9]

action to take according to the block state. It either directly responds or forwards the request to the cache controller. Transactions typically involve two or three steps: a unicast request, a response or a forwarded request, and the response for the forwarded request. In contrast, snooping protocols do not centralize the information and require requests to be broadcast but can complete a transaction in two steps.

Transactions are generally serialized in the directory. If two requests contend for the same location in the directory, the interconnection network chooses which request the directory will process first. In this case, the second request might either be processed directly after the first one or held in the directory awaiting the first request to complete. In some designs, negative acknowledgments can be directly sent.

Since there is no total order, all the controllers do not observe the same things and the requests must be individually serialized concerning all the caches concerned by them, The requester is thus notified when the cache controller has serialized its request.

2.10.1 MSI Directory Protocol

An unoptimized version of the MSI directory protocol is introduced here. For simplicity, it is assumed that point-to-point ordering is enforced. A directory is added right next to the LLC and the LLC controller is also the directory controller. A block is owned by the directory controller unless a cache holds it in a M state. For each block in memory, there is a corresponding directory entry. It includes the stable state, the identity of the owner, and the identities of the sharers in a one-hot-encoding manner. The I to S transition can be divided into three cases when a cache controller issues a $GetS$ request to the directory and changes its state to IS^D.[§].

1. If no cache controller holds the block in M and the directory is the owner, the directory responds with a data message, changes the block state to S, and adds the requester to the sharer list. The transactions are complete when the data arrives at the cache and the states of the block change to S inside the cache block.

2. If the directory is not the owner, the request is forwarded to the owner and the block state changes to S^D. The owner responds to the $fwd - GetS$, sends a data message to the requester, and changes the block state to S. It then sends the data to the directory which must have an up-to-date copy. When the data arrives at both the cache and the directory, they both transition the block to state S and the transactions are complete.

3. If the issuing cache controller receives an invalidation message, the directory first sends a data message due to the former request and then an invalidate message due to the later request. As data and invalid messages are of different classes, they can arrive out of order at the coherence controller.

I or S to M Transition

When a cache controller issues a $GetX$ request to the directory, it changes the block state to IM^{AD}. The directory responses can be divided into several scenarios,

[§]AB^C is a transient state which means a state is changing from I to S for block C

1. If the directory is in I state, it sends a data message with a *Ack_count* of zero, transitions to state M, and updates the block owner.
2. If the directory is in M state, it has to forward the request $fwd - GetM$ to the owner and update the block owner. The now-previous owner responds by sending a data message with *Ack_count* of zero.
3. Races might happen when, for instance, a cache controller receives $fwd-GetS$ message while in IM^A, because the directory has already sent data to the controller, sent invalidate to sharers and changed the block state to M. Thus, it simply forwards the *GetS* message and the $fwd - GetS$ message, traveling on a different network, can arrive before the $Inv - Acks$.

M to I Transitions

When a cache controller evicts a block in M, it sends a *PutM* request to the directory and changes its state to MI^A. The directory updates the LLC, responds with a $Put - Ack$ message, and transitions to I. Meanwhile, the block remains effectively in M at the cache, and if $fwd - GetS$ or $fwd - GetX$ messages are received, the controller responds and changes the block state to SI^A or II^A, respectively, denoting that the block state is effective S or I, but the controller must wait for the *PutAck* to complete the transaction.

S to I Transition

In this type of protocol, a shared block is more silently evicted which means the information is no more broadcast. When a cache controller evicts a block in S, it sends a *PutS* request to the directory and changes its states to SI^A. The directory updates the LLC, responds with a $Put - Ack$ message, and transitions to I when all *PutS* messages from all sharers are received. Meanwhile, the block remains effectively in S at the cache, and if an invalidation request is received, the controller changes the block state to II^A, denoting that the block state is effective I, but the controller must wait for the *PutAck* to complete the transaction.

2.10.2 MESI Directory Protocol

The MESI protocol enables a processor to issue a *GetS* request to obtain a block in-state E if no other cache holds the block. It allows the cache controller to silently upgrade the block state from E to M issuing a coherence request.

As the cache holding the block in-state E or M is the owner of the block, the directory will forward requests to it. Moreover, as for the M state in the previous MSI protocol, the eviction of a block in E must notify the directory with a *PutE* message to inform it that it is a new block owner.

2.11 Cache Coherence Verification

Cache coherence is an essential aspect of modern computer architecture that ensures consistency between multiple copies of the same data held in different caches. As the number of processors in a system increases, maintaining cache coherence becomes more challenging, and verification becomes critical.

Cache coherence verification is the process of ensuring that a system's cache coherence protocol correctly maintains the consistency of shared data across multiple processors. This verification process involves rigorous testing of the protocol's design, implementation, and operation to ensure that it correctly handles all possible scenarios that may arise in a multi-processor system.

To verify cache coherence, engineers use various techniques, including simulation, formal verification, and hardware emulation. Simulation involves modeling the cache coherence protocol in software and running a suite of tests to ensure that it behaves correctly. Formal verification, on the other hand, uses mathematical reasoning to prove the correctness of the protocol's design. Hardware emulation involves implementing the protocol on specialized hardware to simulate its behavior accurately.

One of the most significant challenges in cache coherence verification is detecting and handling potential race conditions, where multiple processors attempt to access the same data simultaneously. These situations can lead to inconsistent data, and it is critical to ensure that the cache coherence protocol handles them correctly.

Another challenge is ensuring that the cache coherence protocol can handle changes to the system's topology, such as adding or removing processors. This requires testing the protocol's scalability and flexibility to ensure that it can handle various system configurations without sacrificing performance or correctness.

Cache coherence verification is essential because any errors in the protocol can lead to incorrect data, crashes, or even system failure. It is critical to ensure that the cache coherence protocol is rigorously tested and verified to avoid these issues.

In conclusion, cache coherence verification is a critical aspect of modern computer architecture. It involves testing and verifying the cache coherence protocol's design, implementation, and operation to ensure that it correctly maintains the consistency of shared data across multiple processors. To achieve this, engineers use various techniques, including simulation, formal verification, and hardware emulation, and address challenges such as race conditions and changes to the system's topology. Ensuring cache coherence verification is thorough is essential to avoid issues that can lead to incorrect data, crashes, or system failure.

2.12 Related Work

2.12.1 Formal Verification

Formal methods are mathematical techniques for verification and proving the correctness of systems. In cache coherence protocols, formal methods can be applied to verify the correctness of the protocol's design and implementation. One approach is *model checking*, which involves analyzing a formal model of the system to verify that it satisfies a set of properties. Another approach is *theorem proving*, which involves using mathematical proofs. For example, researchers have applied formal methods to verify the correctness of cache coherence protocols in multi-core systems [3] and to prove the correctness of a hardware implementation of the MESI coherence protocol [14]. These formal methods provide a rigorous and systematic approach to ensure the correctness of cache coherence protocols, which is crucial for ensuring the reliability and stability of modern computer systems. The main problem with these techniques is that the memory needed to verify the whole system is huge and often unrealistic.

2.12.2 Random Testing

Random testing is the most used technique used to verify cache coherence protocols in modern computer systems. Random testing involves generating random input data to stress the cache coherence protocol and to detect any issues related to coherence and consistency in shared memory locations. According to [2] random testing can be used to detect errors in cache coherence protocols that may be missed by directed testing methods. The paper describes a framework for generating random test cases that ensures complete coverage of the protocol's state space. Random testing is a valuable technique for ensuring the correctness of cache coherence protocols and can provide a more comprehensive test coverage than other testing methods.

The main problem is that the coverage goal is not 100%. The longer we run the tester the better result we can get. For 100% coverage, we need a lot of test cases which may be unrealistic sometimes.

2.12.3 Directed Testing

Directed testing involves designing test cases that target specific behaviors or properties of the system being tested. In the context of cache coherence protocols, directed testing can be used to verify that the protocol behaves correctly under different scenarios, such as when multiple processors are accessing shared data simultaneously.

One study conducted by [16] explored the use of directed testing to verify the correctness of a cache coherence protocol. The authors designed test cases that targeted specific behaviors of the protocol, such as write operations and read operations, to ensure that the protocol maintained data consistency.

Another study conducted by [15] focused on the use of directed testing to verify the correctness of a cache coherence protocol for heterogeneous multi-core systems. The authors designed test cases that targeted specific behaviors of the protocol, such as atomicity and coherence, to ensure that the protocol maintained data consistency across multiple cores. The study found that directed testing was effective in identifying defects in the cache coherence protocol and that it was an essential component of the verification process.

Another paper [5] explored the use of directed testing for validating cache coherence protocols. The author proposed a directed test generation method that utilizes a finite-state machine of a cache coherence protocol to generate directed test cases that target specific behaviors of the protocol. The proposed method was evaluated using several cache coherence protocols, and the results demonstrated about 100% coverage. The paper provides valuable insights into the use of directed testing for validating cache coherence protocols and presents a practical method for generating directed test cases.

2.13 Structural Simulation Toolkit

SST is an open-source, cross-platform simulation platform used to simulate cache coherence protocols in multiprocessor configurations with different memory hierarchies. SST is divided into two libraries,

1. *SST-Core*: It is entirely responsible for simulation creation and teardown. It tracks simulation time and helps to send messages between components. It also provides services like debugging simulation, statistics tracking, memory management, etc.

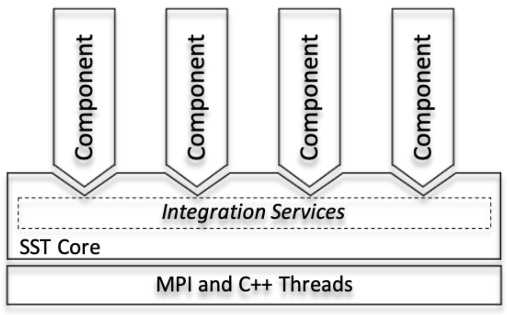

Figure 2.6: SST Component Diagram

2. *SST Elements*: It is a library of components, using which we can simulate different models, such as CPU models (trace and execution driven), memory models (Cache, DRAM, etc), and interconnect models (NoC, etc). Figure-2.6 shows a high-level design of the SST architecture.

2.14 SST Configuration and Execution

Users have to construct simulation inputs using Python or JSON and instantiate SST with the simulation input. SST internally determines the required components and partitions the components into threads and MPI ranks. The dynamic loader loads the components with appropriate data, subsequently SST time vortex is initialized. The message/events are exchanged and after all of that, SST completes the simulation.

2.15 Simulation using SST

SST can accept Python and JSON script as inputs. In our work, we used Python for simulation purposes as it has the best support for the SST-Elements. To run an SST simulation in Python we need to import a specific SST Python library and instantiate one or more components and their associated parameters. Optionally clock configuration and statistics setup can be instantiated for better simulation.

2.16 SST Statistics

SST Statistics can be utilized to get information from a simulated component. By default, the statistics are not enabled but can be enabled for different components and also for user-defined levels. Different statistics are defined for different purposes,

1. Accumulator Statistics, which is analogous to traditional CPU counter
2. Histogram Statistics, which is useful for getting the range of data. etc.

We use SST as the backend tool for our work. Our framework is built on top of SST.

Chapter 3

Directed Testing of Cache Coherence for a Single Level Cache Hierarchy

In this chapter, we present our proposal for a directed testing framework for SI, MSI, and MESI cache coherence protocols for the single-level cache hierarchy. Directed testing has offered a scalable verification technique for analysis, and verification of a wide range of hardware and software systems and is being extensively used in verifying cache coherence protocols. Our proposal intends to improve the state of the art in cache coherence.

3.1 Introduction

In modern computing systems, each processor core maintains its cache for fast access. One major problem with caching is when the same data or memory block is cached in two or more different places, any modification should be propagated to all the cached copies. To do that we need to make sure that, at one point in time only one unit is writing and the updated data is propagated properly. Every cache line reserves some bits to store the states. These states indicate the status of that particular cache line. Depending on the request from the core, the cache controller checks the cache line and checks some more things. First, it checks whether the requested block is in the cache or not. If the block is in the cache (cache hit), the controller checks its corresponding cache line, and depending on the request it decides the next operation. For example, for a write request, the cache controller needs to notify the other caches. If the block is not in the cache, the cache controller requests for the block from the lower level of the memory. This is a very simplistic model of behavior and there are a lot of things that can go wrong. We need a concrete set of protocols to ensure that the processor core reads the most updated version of a block. Cache coherence protocols are those sets of protocols to ensure these kinds of safety. With the increasing demand for complex memory systems, the cache coherence protocols are becoming very complex.

From a protocol specification, we get the idea of how a cache controller should behave depending on different requests from the core and the cache line states. The specification for a core for each cache block is modeled as a finite state machine (FSM). It is needless to state that a cache behavior is affected by another cache. So instead of considering a single cache behavior, we considered all the possible behaviors of all caches for that block across all cores. A global finite state machine can define all possible behaviors of the cache blocks in a system with n-cores, and the entire state space is the product of the n cache block-level FSMs. The FSM of each cache controller is easy to understand, the structure of the product FSM for modern cache coherence protocols usually has quite obscure structures that are hard to analyse [6]. As the cache coherence protocols are becoming very complex, the reachable state space grows exponentially with the number of processing units and states. The verification teams are facing significant challenges to achieve exhaustive verification coverage within a tight time-to-window market. The challenge of exhaustive verification of cache coherence protocol implementation lies in being able to generate tasks covering all transactions and configurations of the different cores for a particular cache block. As the number of cores increases, so does the number of states in the global FSM for a block. Also, the complexity of modern cache coherence protocols being proposed makes this even harder since we now have multiple states for a block.

The global state machine, though large in size, gives us an artifact, which encodes all the possible states and transactions corresponding to the different configurations that the different cores can be in. We can traverse the global FSM to generate transactions that can adequately test a given protocol implementation. This is the standard approach taken in literature and by us as well. The novelty is in managing the traversal scalably and efficiently. One traversal can lead us to one or many test cases that can be used for protocol validation.

One possible solution to carry out the FSM traversal of the global FSM is to use breadth-first-search (BFS). Given the protocol specification, it is possible to compose a test suite that can activate all tests and transitions using two steps.

1. For each state, determine the instruction sequence to reach it by performing a BFS on the global FSM.
2. For each transition, we create the test by appending the required instructions after the instruction sequence to reach the initial state of this transition.

Clearly, the approach is naive and the transitions to or from the initial state will be visited repeatedly. Also, the states in the state space need to be remembered and as the number of states in the state space is large, this approach is memory inefficient.

Industry-level testers are using simulation using random and constrained-random approaches because of their scalability. However, because of the random nature, verification engineers don't have control over the covered state space and these testers require an unacceptable amount of time to cover the entire state space. On the other end of the spectrum, directed testing is promising to achieve higher coverage with a drastically lower number of tests [10]. Therefore these tests can be applied in addition to random tests. [5]. Due to memory and time requirements sometimes directed testing also creates issues. So it was desirable to use an on-the-fly test generator with a space and time-optimized test generation algorithm. In the paper [5], authors proposed an on-the-fly test generation technique for cache coherence protocols by analyzing the state space structure of their corresponding global FSMs. The authors also showed that complex state-space structures can be broken down into simpler structures. Formulating a path-searching problem, the authors exploited those structures for efficient test generation.

Even though on-the-fly algorithms generate a much lower number of test cases than random or constrained-random tests, for a higher number of cores, the number of cores and the number of test sequences are still high and sometimes not feasible to execute. In our work, we attempt to reduce the test sequences. Our contributions for this chapter are,

1. We present a different composition of the state space structures of the protocols.
2. We develop an efficient test generation approach to address the scalability concerns in existing test generation techniques.

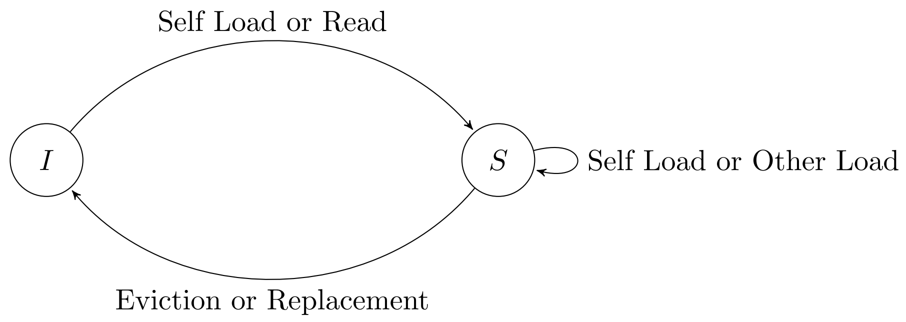

Figure 3.1: State Transitions for a cache block in SI protocol for a particular core.

The rest of the chapter is organized as follows. Section-3.2 provides background and a motivating example for our work. Section-3.3 proposes our eviction-based approach and the required algorithms. Section-3.4 demonstrates our simulation environment. Section-3.5 presents the results.

3.2 Background and Motivating Example

One of the simplest cache coherence protocols is the Shared Invalid (SI) Protocol [9]. The behavior of the cache controller in a processing core is modeled as a finite state machine for each block. As shown in Figure-3.1, the state I changes to state S when the core issues load or read requests. S stays the same when it or another core issues a load request. The state S changes to I when the cache block is evicted because of invalidation by some other core or cache replacement. To generate the on-the-fly tests, authors of [5] constructed an FSM for the entire state space of different protocols. For the SI Protocol every cache line has two states $Shared(S)$ and $Invalid(I)$. When a load request (self-load or self-read) arrives, the cache controller requests the data from the main memory and switches to a shared state. When the block is replaced by another block, the state changes to invalid. This protocol is very simple and a cache has to be write through to make the protocol work.

In [5], the authors defined all possible behaviors of the cache blocks in a system by n cores with private caches employing the SI protocol, by a global finite state machine (FSM). The entire state space is the product of n cache block-level FSMs. For the SI protocol for each block, each core can have two states. For n-cores, there will be 2^n states in the globally composed FSM obtained by a synchronous product of the individual FSMs. These states are labeled by a vector of length n, where each index denotes the unique core index and each element in the vector denotes the states of the corresponding cache. From each state, we have exactly n-transitions, i.e. one of the n caches can change its state totaling n possible transitions from a state. From a graph-theoretic point of view, we have a graph G, G has 2^n nodes, and every node has degree n and two nodes have an edge if their node labeling differs in exactly one position. The resultant graph should be a n-dimensional hypercube as shown in Figure-3.2. From an architectural point of view, the edges in that graph are bi-directional. The total number of edges for an undirected n-dimensional hypercube is $n2^{n-1}$. The total number of transitions for the global finite-state machine of the SI protocol is $n2^n$.

In [5, 8], the authors proposed a graph-based technique to verify the SI cache coherence protocol. It was proven that a n-hypercube can be broken into n isomorphic trees. [13]. For the SI protocol, all the nodes of these isomorphic trees have even degree nodes, which means an efficient Euler tour can be performed. The authors proposed their efficient test generation process based on that. They offered an algorithm (Algorithm-1) that produces a test sequence of length $n2^n$ instructions, which equals the number of transitions possible in the entire FSM of the SI protocol.

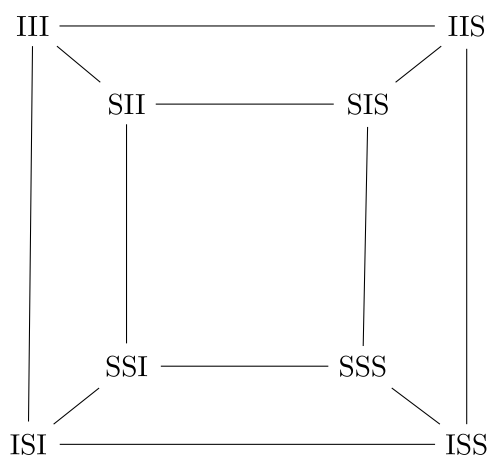

Figure 3.2: State Space of SI Protocol for 3 cores

Algorithm 1 Test Generation for SI Protocol [5]

```
Input: User to define the number of cores $N$
Output: Test sequences
1: for $i = 0$ to $N - 1$ do
2:     visitHypercubeUtil(N,i,N)
3: end for
```

As an example, let us take the number of cores as 3. At first, all cores at the beginning are in the I state. In the first round of the for loop in line 2, $visitHypercubeUtil(N, 0, N)$ is called. The system performs the III-SII transition by exciting a load operation by the 0^{th}-core, followed by two recursive calls with $i = 1$ and $i = 2$. When $visitHypercubeUtil(1, 0, N)$ is called the transitions IIS-ISS and ISS-IIS are visited by executing a load operation and followed by an evict operation by the 1^{st} core without any further recursion. When $visitHypercubeUtil(2, 0, N)$ is called, IIS-SIS is visited by a load operation by the 2^{nd} core, then $visitHypercubeUtil(1, 0, N)$ is invoked to activate two transitions SIS-SSS and SSS-SIS, and at last SIS-IIS is covered by executing an evict operation by the 2^{nd} core. Finally, the global state goes to III via the transition IIS-III after the evict operation by the 0^{th}-core. In the same way, the remaining transitions are covered. It may be noted that for every transaction issued, a transition in the global state space is covered.

The execute path seems complicated for a larger number of cores. The essential idea of this algorithm is based on the fact a n-hypercube that can be decomposed into n isomorphic trees. To show the correctness of the algorithm, the authors proved the following results.

1. There are $n2^n$ transitions within the state space of the SI protocol with n cores and the number of sequences produced by the above algorithm is also $n2^n$. So, instead of issuing an eviction request, we replace the instruction with a load on another block, that maps to the same cache line of the previous block.

2. No transition is covered more than once while applying the algorithm.

Even though the result is useful, there are major issues while applying the above algorithm, which motivated us to propose a new approach for the test generation task. We mention a couple of them below.

1. A core just cannot evict a cache block. The cache block can be replaced or invalidated by

Algorithm 2 visitHypercubeUtil

Input: M, r, N
Output: Test Sequences

```
1: $p \leftarrow (M + i)\%N$
2: print(load $B0$ block by $p^{th}$-core)
3: for $i = 1$ to $M - 1$ do
4:     visitHypercubeUtil(i, r, N)
5: end for
6: print(evict $B0$ block by $p^{th}$-core)
```

another processor. [5] achieved the eviction operation by loading another block that maps to the same cache line.

2. The number of test sequences by which the entire state space is covered is not optimal. By issuing one instruction we sometimes can cover two transitions.

For our example, let us consider a memory system with two cores and two blocks $B0$ and $B1$. The important point to note is that both addresses map to the same cache line. The transaction sequences generated by the above algorithm will be

```
1. Load B0 block by 0th core
2. Load B0 block by 1st core
3. Load B1 block by 0th core
4. Load B1 block by 1st core
5. Load B0 block by 1th core
6. Load B0 block by 0th core
7. Load B1 block by 0th core
8. Load B1 block by 1st core
```

While analyzing the above transaction, we consider the transitions for both $B0$ and $B1$ blocks. At first, both the blocks are in a II state. After the first two instructions for the $B0$ block, the transitions II-SI and SI-II are covered. In the third instruction, the $B1$ block is loaded by the 0^{th} core, which forces the $B0$ block to be evicted from the 0^{th} core. In that way, II-SI transition and SS-IS transitions are covered. This type of behavior can be seen for the rest of the transactions as shown in Figure-3.5.

3.3 Our Approach

3.3.1 SI Protocol

In this section, we present our eviction-based approach for verifying the SI protocol. Algorithm-1 outlines our test generation procedure for SI protocol, which decomposes the n-hypercube in two isomorphic halves. Whenever we invoke a print function with a load or read operation, the core is asked to perform a load operation for a specified block. This becomes part of a testcase through which we can cover one or more transitions of the global state machine.

Our proposed algorithm adopts the algorithm proposed by [5] which is mentioned earlier. The only difference is that we replace the eviction operation by loading another block into the same cache line.

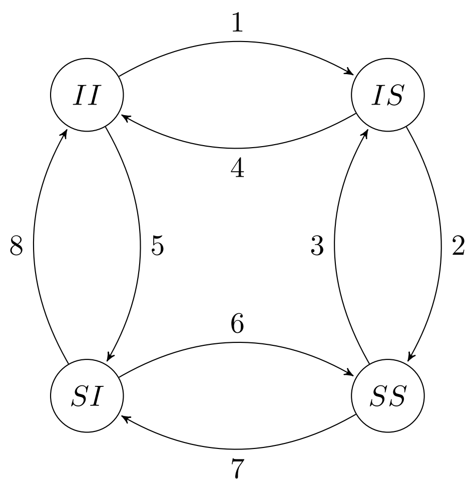

Figure 3.3: Transitions for $B0$ block

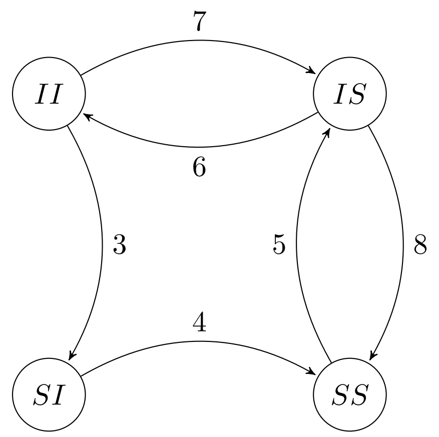

Figure 3.4: Transitions for $B1$ block

Figure 3.5: Example of how two transitions can be covered with one single instruction

Additionally, we use a function *_visitHypercube* that does Gray code traversal and some additional load instructions upon reaching a state.

In our algorithm, we take two blocks and cover the states and transitions on those two blocks. Each block covers different transitions of the n-hypercube that models the global FSM. As an example, we consider the number of cores as 4 and the state space of Figure-3.6 to show the execution of our algorithm. Before going into the analysis, we define the following concepts: *state_vector*, *complementary state vector* and *complementary transitions*.

Definition 3.1: A state vector for an address is a one-dimensional vector used to label the states of the global FSM. ■

A state vector has a length of N, which is the same as the number of cores. Each index in the state vector indicates the states of a particular core. For the SI protocol, the state vector can have only S or I values.

Definition 3.2: Two state vectors X and Y encoding 2 states of the finite state machine are said to be complementary if they satisfy the following condition, $\forall i \in [0, n-1]$ *and* $i \in \mathbb{Z}$, $X[i] \neq Y[i]$. ■

For example, IIS and SSI are complementary states.

Definition 3.3: Two transitions $A \hookrightarrow B$ and $C \hookrightarrow D$ are said to be complementary transitions if state vectors A and C are complementary and B and D are complementary. ■

For example, IIS-SIS and SSI-ISI are complementary transitions.

The main idea that drives our algorithm is the fact that every n-hypercube can be decomposed into 2 isomorphic halves, hence, a traversal of the transitions of the hypercube can be achieved by covering the transitions of the two halves individually. Additionally, as we observe in the case of the SI protocol, these halves are isomorphic and contain complementary states and transitions. Thus, if we cover one state / transition in 1 half, a corresponding state / transition gets covered in another half. This is what we utilize in our approach, by associating one block with one of the halves and another block that maps to the same cache line with another half. Overall, as we cover states / transitions with one

Algorithm 3 visitHypercube: Algorithm for Test Generation of SI Protocol with Reduced Test Sequences

```
Input: User to define the number of cores $N$
Output: Test sequences
 1: if $N \leq 2$ then
 2:     for $i = 0$ to $i = N - 1$ do
 3:         visitHypercubeUtil(N,i,N)          ▷ This function is adopted from [5, 8].
 4:     end for
 5:     return
 6: end if
 7: for $i = 0$ to $i = N - 1$ do
 8:     print(Read $B1$ block by $i^{th}$ core)
 9: end for
10: for $j = 0$ to $j = N - 3$ do
11:     visitHypercubeUtil(N-2,i,N-2)          ▷ traverses the hypercube of dimension $N - 2$
12: end for
13: _visitHypercube(N-2)
14: print(Write $B0$ block by $(N - 1)^{th}$ core)
15: for $j = 0$ to $j = N - 3$ do
16:     visitHypercube(N-2,i,N-2)              ▷ Adopted from [5, 8]. It traverses the hypercube
                                                 of dimension $N - 2$
17: end for
```

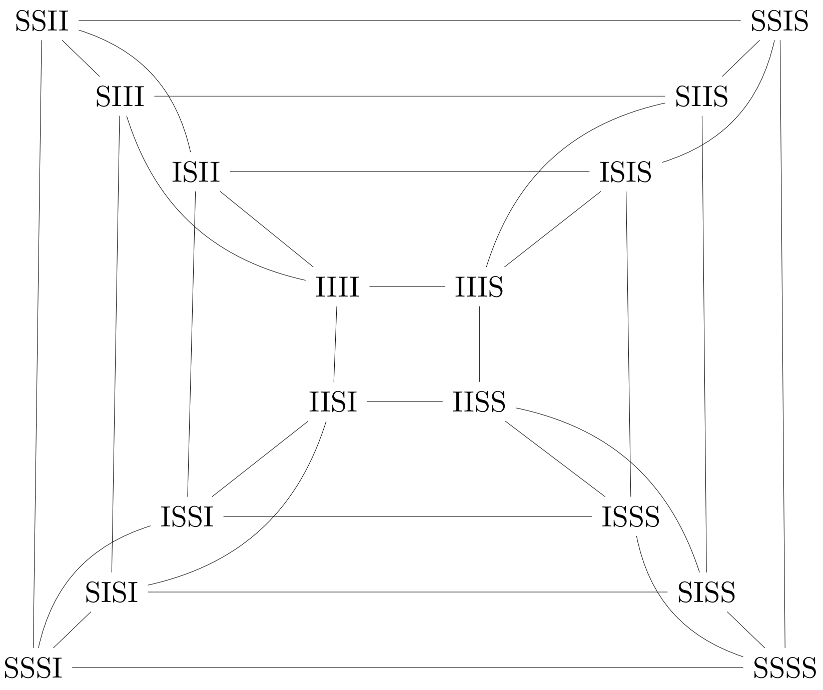

Figure 3.6: State Space of SI protocol for 4 cores

block, the complementary states / transitions in the other half for the other cache block gets covered. This is where the eviction helps us. We now explain the working of our algorithm. On line-2 of Algorithm-3, we check a boundary case. If the number of cores is less than 2, we apply Algorithm-1. For our example, the number of cores is 4. In the beginning, the state vectors of two blocks are $IIII$. We execute the **for loop** on line-7. After this for_loop, the state vector of block $B0$ remains at $IIII$ but for block $B1$ it becomes $SSSS$. Two of the blocks have complementary state vectors. The **for loop** from line-10 to line-12 is the functionality adopted from [5]. It covers all the transitions and states in the $N-2$ dimension. For block $B0$, the transitions with respect to the states $IIII$, $ISII$, $SIII$, and $SSII$ are covered corresponding to the FSM for block $B0$. In turn as complementary transitions, all transitions related to $IISS$, $ISSS$, $SISS$, and $SSSS$ states are covered in block $B1$. we now explain the Gray code traversal.

Gray codes are a system where two successive values differ in only position or index. For the binary numerical system, they differ by one bit. For our case the states in the state space which differ by one position are adjacent. For example, the sequence of Gray codes for 3-bit numbers is 000,001,011,010,110,111,101,100. It is a well-known fact that the Gray code of n bits forms a Hamiltonian cycle on a hypercube, where each bit corresponds to one dimension. Hamiltonian Cycle is a graph cycle where each node is visited exactly once. From this fact, we designed the function $\_visitHypercube$. The $\_visitHypercube$ helps in Gray code traversal and on reaching a state, we execute some additional load operations as shown in Algorithm-4.

Example 3.1: If we are in the $ISII$ state with 4 load instructions as stated earlier, we cover $ISII - ISIS - ISSS - ISIS - ISII$ transitions for block $B0$. In turn for block $B1$, we cover all the complementary transitions, i.e. $SISS - SISI - SIII - SISS$.

Considering the previous example of 4 cores, the transaction sequences generated from the $\_visitHypercube$ function are given below,

```
1. Read B0 block by 3rd core
2. Read B1 block by 2nd core
3. Read B1 block by 2nd core
4. Read B1 block by 3rd core
5. Read B0 block by 1st core
...
10. Read B0 block by 0th core
...
15. Read B1 block by 1st core
...
20. Read B1 block by 0th core.
...
```

It may be noted that we have omitted some of the transitions. The omitted transitions are the same as the transitions (1), (2), (3), and (4) as shown in the above testcase. As mentioned earlier on reaching a state, these transitions are issued. The detailed transitions are shown in Figure-3.7. Next, we issue another instruction as shown in line-14. At this point, block $B0$ and block $B1$ have state vectors $IIIS$ and $SSSI$. Next, we again execute the hypercube traversal from line-15 to line 17 (Algorithm-3), and the rest of the transitions are covered. To show the correctness of our algorithm, we prove the following claims.

1. There exists a decomposition by which the n-hypercube can be decomposed into two isomorphic halves. We construct the two isomorphic halves by choosing the states and transitions as

Algorithm 4 _visitHypercube: Special Gray code traversal for X dimension

Input: X, N

Output: Test Sequences

```
 1: $prev \leftarrow$ ””
 2: print(Read $B0$ block by $(N-1)^{th}$-core)
 3: print(Read $B0$ block by $(N-2)^{th}$-core)
 4: print(Read $B1$ block by $(N-2)^{th}$-core)
 5: print(Read $B1$ block by $(N-1)^{th}$-core)
 6: for $i = 0$ to $i \leq (1 \ll X) - 1$ do
 7:     $val \leftarrow (i \oplus (i \gg 1))$
 8:     $bitset < 32 > r(val)$        ▷ bitset is used in C++. It represents a fixed-sized sequence of some bits and stores values of either 0 or 1
 9:     $s \leftarrow r.to\_string()$        ▷ to_string() is a function inside C++. It converts a bitset into string
10:     if $prev$ is an empty string then
11:         $prev \leftarrow s$
12:         continue
13:     end if
14:     $s \leftarrow s.substr(32 - X)$
15:     $a \leftarrow find\_pos(prev, s)$        ▷ Find the position where there is a change mismatch between $s$ and $t$
16:     $prev \leftarrow s$
17:     $b \leftarrow s[a]$
18:     if the value of $b$ is 0 then
19:         print(Read $B1$ block by $a^{th}$ core)
20:     else
21:         print(Read $B0$ block by $a^{th}$ core)
22:     end if
23:     print(load $B0$ block by $(N-1)^{th}$ core)
24:     print(Read $B0$ block by $(N-2)^{th}$ core)
25:     print(Read $B1$ block by $(N-2)^{th}$ core)
26:     print(Read $B1$ block by $(N-1)^{th}$ core)
27: end for
28: print(Read $B1$ block by $0^{th}$-core)
```

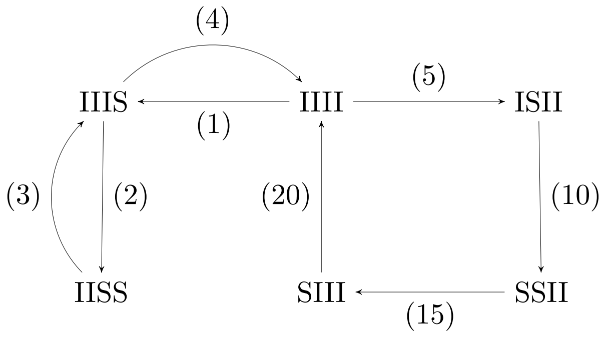
$$\begin{tikzcd}
\text{IIIS} \arrow[d, "(2)"] \arrow[rr, bend left, "(4)"] & & \text{IIII} \arrow[ll, "(1)"] \arrow[rr, "(5)"] & & \text{ISII} \arrow[d, "(10)"] \\
\text{IISS} \arrow[u, bend left, "(3)"] & & \text{SIII} \arrow[u, "(20)"] & & \text{SSII} \arrow[ll, "(15)"]
\end{tikzcd}$$

Figure 3.7: Transactons due to *_visitHypercube*. The transitions are in accordance with the traces stated. The transitions shown in this figure are for the B0 block. The complementary transitions are covered by B1 block.

.

complementary. The isomorphism ensures that a state / transition covered for one block covers a corresponding one in the other half.

2. The total number of transactions in the test sequence produced by Algorithm-3 is $n + n2^{n-1} + 2^{n-2} + 1$ and it covers the entire state space of the SI protocol.

Theorem 3.1: There exists a decomposition such that any n-hypercube ($n \geq 2$) can be decomposed into 2 isomorphic halves.

Proof. We model the finite state machine as a graph $G(V, E)$ where nodes or vertices are labeled with a vector of length n as stated earlier. The goal is to decompose G into two isomorphic graphs, $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$. The following procedure is applied to get G_1 and G_2,

1. While choosing the vertices $v \in V_1$ for $G_1(V_1, E_1)$, apply the following rule, $\forall i \in [0, n-3], v[i] \in \{I, S\}$,
   (a) $if\ v[n-2] = I,\ then\ v[n-1] \in \{I, S\}$,
   (b) $if\ v[n-2] = S,\ then\ v[n-1] = S$
2. While choosing the vertices $v \in V_2$ for $G_2(V_2, E_2)$, apply the following rule, $\forall i \in [0, n-3], v[i] \in \{I, S\}$,
   (a) $if\ v[n-1] = I,\ v[n-2] \in \{I, S\}$
   (b) $if\ v[n-1] = S,\ v[n-1] = S$
3. While choosing edges $e(u_1, u_2) \in E_1$ for graph $G_1(V_1, E_1)$, connect those edges from the original graph G which satisfy the following,
   (a) $u_1[n-2:] \in \{II, IS\}\ and\ u_2[n-2:] \in \{II, IS\}$
   (b) $u_1[n-2:] = IS\ and\ u_2[n-2:] = SS$ and vice-versa.
4. While choosing the edges in $e(u_1, u_2) \in E2$ for $G_2(V_2, E_2)$, connect those edges from original graph G which satisfy the following,
   (a) $u_1[n-2:] \in \{SS, SI\}\ and\ u_2[n-2:] \in \{SS, SI\}$
   (b) $u_1[n-2:] = SI\ and\ u_2[n-2:] = II$ and vice-versa

[*]

To prove two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are isomorphic, a bijection needs to be formulated such that, $f : V_1 \rightarrow V_2$ and the following condition should be satisfied, $if\ \{u_1, u_2\} \in E_1,\ \{f(u_1), f(u_2)\} \in E_2$ and vice versa. Let us consider a function $f(x) = \overline{x}$ that is upon getting one vector x, the function computes the complement of x.[†]. To find the complement, we replace I in every index of state vector x with S and vice-versa.

[*]vec[s: t] is a vector slicing method. **s** indicates the starting position and **t** indicates the last position of the vector *vec*. Point to note that this method returns a part of a vector from position **s** to **t-1**, considering 0-based indexing. Sometimes the values of **s** and **t** are omitted. In that case, we use the default value of **s**, which is 0, and **t**, which is the length of the original vector.

[†]+ operator is defined on the vector, which is vector concatenation. As function f outputs a vector, + operator also works on function f.

Consider a vertex in $u_1 \in V_1$

$$f(u_1) = f(u_1[: n-2] + u_1[n-2 :]) = f(u_1[: n-2]) + f(u_1[n-2 :]) = u_2$$

Following the rule-1 and rule-2 it can be concluded that $f(u_1[: n-2]) \in \{I, S\}$.

Case 1: If $u_1[n-2 :] = II \rightarrow f(u_1[: n-2 :]) = SS, \rightarrow u_2 \in V_2$

Case 2: If $u_1[n-2 :] = IS \rightarrow f(u_1[n-2 :]) = SI, \rightarrow u_2 \in V_2$

Case 3: If $u_1[n-2 :] = SS \rightarrow f(u_1[n-2 :]) = II, \rightarrow u_2 \in V_2$

The edges are connected in both the sub-graphs according to the rules stated above. Now it has been shown earlier that both the graphs $G1$ and $G2$ contain complementary vertices. For any edge $\{u_1, u_2\} \in E_1$, the corresponding complementary vertices of u_1 and u_2 will be in graph $G2$ and the corresponding edge should be in graph $G2$, satisfying the second condition. □

Theorem 3.2: Algorithm-3 satisfies the coverage goal, i.e. all the edges of the hypercube are covered bi-directionally.

Proof. We provide a constructive argument to prove this.

Case 1: When $n \leq 2$, we proceed with the algorithm adopted from [5]. For $n = 2$, our framework provides 1 more transaction than the number of transitions in the global FSM.

Case 2: Let us consider a memory model with $n \geq 3$ number of cores. Depending on the size of the L1 cache size, we choose two blocks that map to the same cache line. Consider two such blocks as $B0$ and $B1$.

1. First the $B1$ block is loaded into every core. At this point, the state vector for $B1$ block is S^n, and for the $B0$ block, the state vector will be I^n.[‡]. After the execution of this step, two blocks are in complementary states. The number of transactions issued in this step is n.

2. *visitHypercube* function is adopted from [5, 8]. The entire for loop from line-10 to line-12 in algorithm-3 is used to traverse the hypercube of dimension $n-2$. Before this for loop, the state vectors for $B0$ and $B1$ blocks were I^n and S^n respectively. Due to the execution of this for loop, the states with label $\{S, I\}^{(n-2)}II$ are visited and all edges corresponding to those states are visited for the $B0$ block. As a consequence, all the states with label $\{S, I\}^{(n-2)}II$ and their corresponding edges are covered for the $B1$ block. The number of test sequences issued here is $(n-2) \times 2^{n-2}$. It may be noted here that after execution of this for loop the state vectors of $B0$ and $B1$ blocks come back to I^n and S^n respectively.

3. Line-13 shows a function call to a Gray code traversal method. Gray code traversal guarantees that all the states are visited exactly once in the hypercube. By calling this function in line no-13 of algorithm-3, all the states with label $\{I, S\}^{(n-2)}II$ are visited exactly once for the $B0$ block. Upon visiting every state, 4 *load/read* instructions are issued, as shown in Algorithm-4.
   (a) load $B0$ block to $(n-1)^{th}$ core, $\rightarrow$ transitions $\{I, S\}^{(n-2)}II\ to\ \{I, S\}^{(n-2)}IS$ are covered.
   (b) load $B0$ block to $(n-2)^{th}$ core, $\rightarrow$ transitions $\{I, S\}^{(n-2)}IS\ to\ \{I, S\}^{(n-2)}SS$ are covered.
   (c) load $B0$ block to $(n-2)^{th}$ core, $\rightarrow$ transitions $\{I, S\}^{(n-2)}SS\ to\ \{I, S\}^{(n-2)}SI$ are covered.
   (d) load $B0$ block to $(n-2)^{th}$ core, $\rightarrow$ transitions $\{I, S\}^{(n-2)}SI\ to\ \{I, S\}^{(n-2)}II$ are covered.

[‡]The notation X^a means a vector with element X repeated a times

Similarly, all the complementary transitions are covered by this block $B1$. The implementation of Gray-code traversal is a bit tricky and our implementation involves some constructs of the C++ Standard Template Library (STL).[§] In the for loop of Algorithm-4 for every index i, we generate the next Gray code. We start with 0 and generate the next code in line-7. Then we convert them to binary form and then to string. This is done to optimize the space requirements. After that we have two choices,

(a) If the *prev* string is empty, this indicates we are in the $i = 0$ step. We just update the prev value with the current Gray code.

(b) If the *prev* string is not empty, we find the substring of length n as shown on line-14 of Algorithm-5. An important point to note is that while executing line-14, we use length $32 - n$. We can use 64 too if the number of cores exceeds 32.

$find\_pos$ function helps to find the position where the previous Gray code string and current Gray code string mismatch. As these are Gray codes, it is guaranteed that these strings will have a mismatch in exactly one index say, $\boldsymbol{a}$. If in the current Gray code, at index, a, the value is *0*, a load of $B1$ block for a^{th} core is issued, else a load of $B0$ block for a^{th} core is issued. After every iteration of the for loop, 4 specific load instructions are issued as shown from line-23 to line-26. For the $n-2$ dimension hypercube, there are 2^{n-2} states. For every state, we issue 4 loads. For Gray code traversal, the number of transitions is 2^{n-2}. Thus, the length of the transaction of the test sequence issued is $2^{n-2} \times 4 + 2^{n-2} = 2^n + 2^{n-2}$. After execution of the $\_visithypercube$ function, the state vectors of $B0$ and $B1$ blocks come back to I^n and S^n respectively.

4. Next a load instruction for the $(n-1)^{th}$ core for the $B0$ block is issued. For $B0$ block and $B1$ block the state vectors will be $I^{(n-1)}S$ and $S^{(n-1)}I$. The length of the test sequence issued here is 1.

5. The for loop from line-15 to line-17 of algorithm-3 is executed next. Due to the execution of this for loop, the states with label $\{S, I\}^{(n-2)}IS$ are visited and all the edges corresponding to those states are visited for the $B0$ block. As a consequence all the states with label $\{S, I\}^{(n-2)}SI$ and their corresponding edges are being covered for $B1$ block. The total number of test sequences issued here is $(n-2) \times 2^{n-2}$. After execution of the for loop, the state vectors of $B0$ and $B1$ blocks come back to $I^{(n-1)}S$ and $S^{(n-1)}I$ respectively.

The total number of test sequences issued is, $n + n2^{n-1} + 2^{n-2} + 1$. $\square$

Lemma 3.1: The time complexity of our algorithm is $\mathcal{O}(n2^n)$ and the space complexity is $\mathcal{O}(n)$. An outline of the proof is included in the Appendix.

3.3.2 MSI Protocol

MSI is another cache coherence protocol that is used extensively in write-back caches. The behavior of the cache controller unit is modeled as an FSM as shown in Figure-3.8. For n cores, each with a private L1 cache, the global state space of MSI becomes a bit complicated.

For the MSI protocol, we keep a modified state (M) in every cache line. When a core issues a store request (self-store or self-write) the cache controller first broadcasts an invalidate request on the bus and then changes to a modified state. Such an invalidation request informs all cores to change the

[§]The C++ STL (Standard Template Library) is a collection of algorithms, data structures, and other components that are used to simplify C++ development.

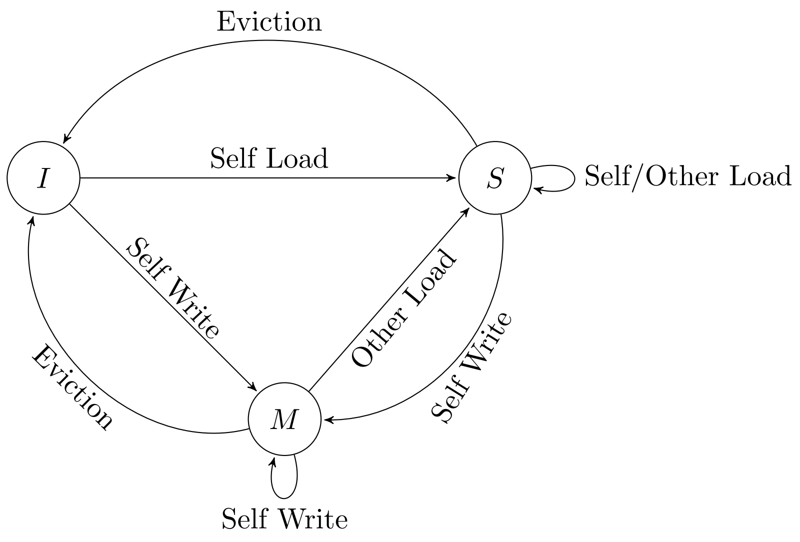

Figure 3.8: FSM of Cache Controller for MSI Protocol

state to I. Now in any cache coherence protocol, there should be only one cache that should be allowed to modify the address. So we have n global modified states in the state space diagram of MSI, hence, the number of states in MSI is $n + 2^n$. We make the following observations.

1. For the global shared and global invalid states [¶], the total number of transitions is $n2^n$. These states form a hypercube which is the same as the one for the SI protocol, as discussed in the previous chapter.
2. The global invalid state and the global modified states[‖] form a clique. The number of transitions, in that case, is $2 \times {}^{n+1}C_2 = n(n + 1)$.[**]
3. Some transitions between every shared state to all modified states are bidirectional, the rest of them are uni-directional. We have the following cases.

   Case 1: The number of unidirectional transitions is $n(2^n - 1)$.

   Case 2: Consider the state vector of a modified state I^xMI^{n-x-1}, where $x \in [0, n - 1]$. Now from this state, we can have a transition to the global shared state if and only if the global shared state has S in the $(x + 1)^{th}$ index and another S in another index other than $x + 1$.

   Example 3.2: For 3 cores, a transition from IIM to ISS is possible but SSI or IIS or not possible.

   Total number of such transitions is $n \times {}^{n-1}C_1 = n(n - 1)$ for n cores.

The total number of transitions possible for MSI is $n2^n+n(n+1)+n(2^n-1)+n(n-1) = n2^{n+1}+2n^2-n$. The state space of MSI for 3 cores is shown in Figure-3.9. For ease of understanding, we only show the hypercube and the clique part. The transitions from shared states to modified states are ignored. In [8], the authors proposed an algorithm to verify the MSI algorithm. In the state space of MSI, there is an SI component. Hence they used the algorithm for verifying the SI algorithm in MSI as well. Our algorithm for verifying MSI protocol is shown in Algorithm-5.

[¶]The global invalid state indicates the state having state vector all-I and the global shared states means the states having a state vector that contains at least one S

[‖]Global modified states means the state's labeling containing exactly one M

[**]We multiplied by 2 as the transitions are bidirectional.

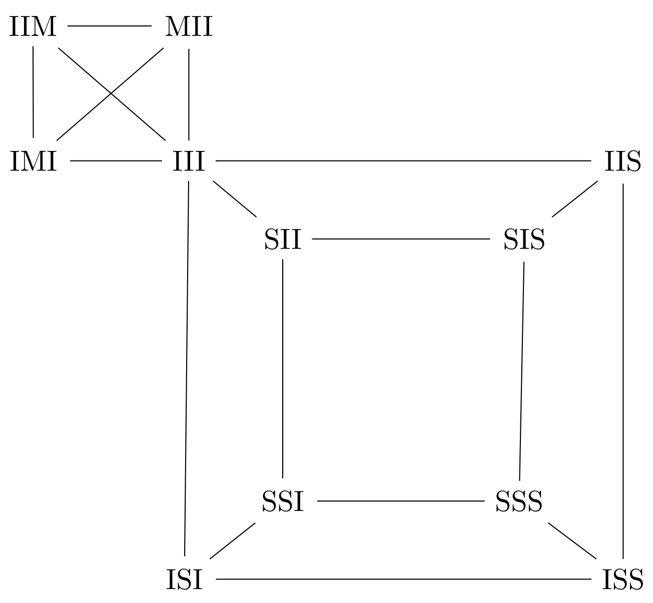

Figure 3.9: State Space of MSI Protocol

Algorithm 5 Test Generation for MSI Protocol [5]

Input: User to define the number of cores N
Output: Test sequences

```
1: createTestsSI(N)
2: visitClique(0)
3: for for every $st \in$ $set\ of\ global\ shared\ states$ do
4:     for $i = 0$ to $n - 1$ do
5:         print(write $B0$ block by $i^{th}$-core)
6:         print the transaction for the shortest path from the current state to $st$
7:     end for
8: end for
```

The *visitClique* function helps to visit all the global modified and global invalid states in one go. The function first covers the III-IIM and IIM-IMI. In the recursive call, IMI-MII and MII-IMI are covered. Next, the transition IMI-IIM is covered. In the next iteration, IIM-MII and MII-IIM are visited. To improve efficiency, we also visit the bidirectionally visited states. Finally in line-3 of algorithm-5, the unidirectional transitional is covered.

3.3.3 MESI Protocol

For the MESI protocol, we have an extra $Exclusive(E)$ state in the cache lines. Figures-3.10 and 3.11 show the transitions between the states in the MESI protocol. The baseline model is a bit different than the MSI protocol. Getting a read request, the state moves to the E or S state depending on the state of the LLC or the memory [7]. Only one cache can be in the E state. The FSM of such a cache controller is shown in Figure-3.10. So number of such states is n and these states are called global-exclusive states. So the total number of states for the MESI protocol is $2n + 2^n$.

1. For all global shared states and global invalid states, the number of transitions is $n2^n$. However, from a global invalid state, the states whose state vector contains a single S are not possible, since for the MESI protocol, when a core reads an address that is not shared by any core, it

Algorithm 6 visitClique:

```
Input: x, N
Output: Test Sequences
1: print(write $B0$ block by $x^{th}$-core)
2: print instructions to visit all bidirectionally reachable global shared states or global invalid state
3: for $i = x + 1$ to $i = N - 1$ do
4:     print(Write $B0$ block by $i^{th}$-core)
5:     if $i == x + 1$ then
6:         visitClique(i, N)
7:     end if
8:     print(Write $B0$ block by $p^{th}$-core)
9: end for
```

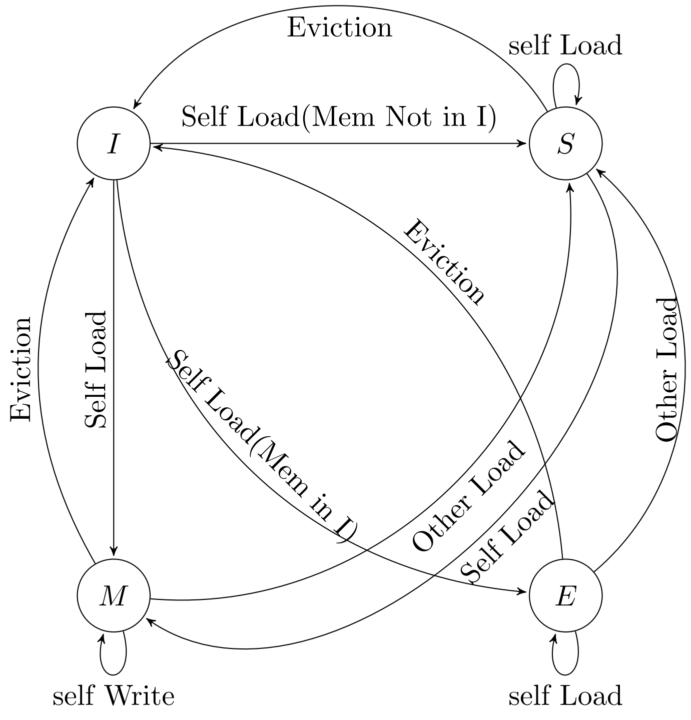

Figure 3.10: Transitions in Cache Controller for MESI Protocol

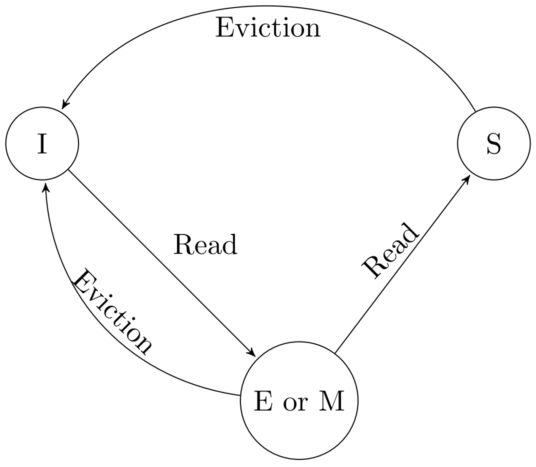

Figure 3.11: Transitions in Memory Controller for MESI Protocol

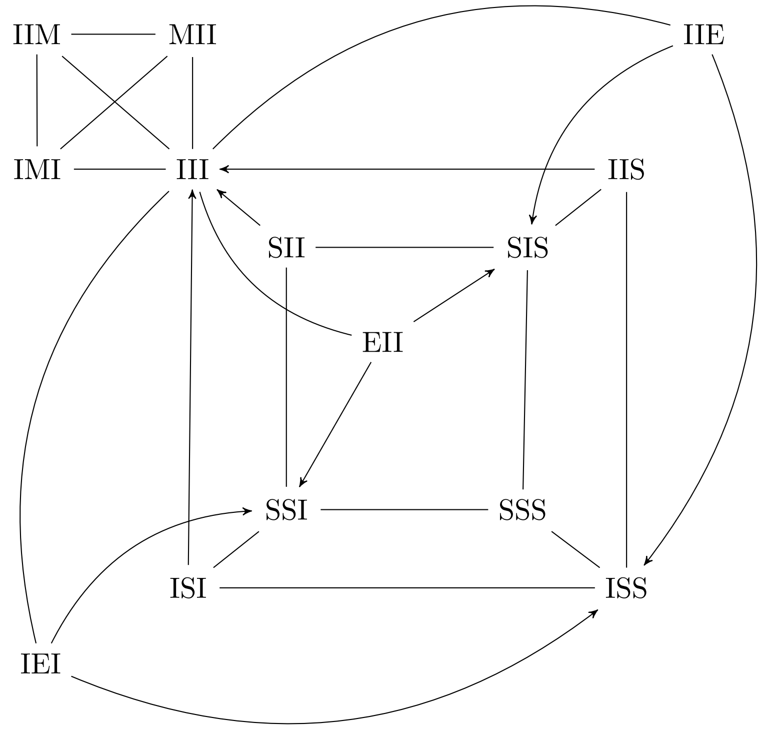

Figure 3.12: State Space of MESI Protocol. Some transitions are omitted.

moves to the E state, and not the S state.

Example 3.3: For a 3 core processor, the transition III to IIS is not possible. But IIS state to III is possible.

The total number of transitions, in that case, is $n2^n - n$.

2. The number of transitions from global invalid states to global exclusive states is $2n$, as these are bidirectional states.
3. As discussed earlier the number of transitions between global invalid and global modified states is $n(n + 1)$.
4. From every global exclusive state to every global modified state, there will be transitions. The number of such transitions is $n \times n = n^2$.
5. As discussed earlier the number of transitions between all shared states and all modified states is $n(2^n - 1) + n(n - 1)$.

The total number of transitions for the MESI protocol is $n2^n - n + 2n + n(n + 1) + n^2 + n(2^n - 1) + n(n - 1) = n2^{n+1} + 3n^2$. The global FSM of n-cores in the MESI protocol is shown in Figure-3.12. In the figure, the bi-directional edges are shown with arrows on them, but the uni-directional edges are shown with arrows in the respective direction. For ease of understanding, we omitted some uni-directional transitions with respect to the global shared state and the global modified states.

In [5], the authors provided algorithms for verifying the MESI protocol. The state space of MESI has some components similar to SI and MSI, so the methods used for SI and MSI protocols can be used for verifying MESI. The algorithm for verifying MESI proposed is given below. The *createTestsSI*

Algorithm 7 Test Generation for MESI Protocol [5]

```
Input: User to define the number of cores $N$
Output: Test sequences
1: createTestsSI(N)
2: visitClique(0)
3: visitE()
4: for for every $st \in$ $set\ of\ global\ shared\ states\ and\ exclusive\ states$ do
5:     for $i = 0$ to $n - 1$ do
6:         print(write $B0$ block by $i^{th}$-core)
7:         print the transaction for the shortest path from the current state to $st$
8:     end for
9: end for
```

Algorithm 8 visitE

```
Input: Index, N
Output: Test Sequences
1: for $i = 0$ to $N - 1$ do
2:     for $i = 0$ to $N - 1$ do
3:         print(Read $B0$ block by $i^{th}$-core)
4:         print(Read $B0$ block by $j^{th}$-core)
5:         if $i \neq j$ then
6:             print(Read $B1$ block by $j^{th}$-core)
7:             print(Read $B1$ block by $i^{th}$-core)
8:         end if
9:     end for
10: end for
```

function works a bit differently than the ones for SI or MSI. The first iteration of the inner loop of *visitE* covers the transitions III-IIE. The second iteration of the inner loop covers the transitions IIE-ISS, ISS-IIS, and IIS-III. The third iteration of the inner loop covers the transitions IIE-SIS, SIS-IIS, and IIS-III.

3.4 Experimental Setup

All experiments and simulations were done on a Windows Subsystem On Linux (WSL) on an Intel(R) Core(TM) i5-103110U CPU with 16GB RAM and Intel(R) UHD 8GB GPU using the SST toolkit.

3.4.1 Simulation Components

We used SST (Structural Simulator Toolkit) for our simulation purpose. For simulation we used three components of SST,

1. *Prospero*: It is a simple processor model which reads memory operations from a trace file and then sends/receives these into a memInterface-based cache system (e.g. memHierarchy) for simulation. The **prospero** implements the trace-based core which issues memory operations into a memEvent-based memory system.

2. *memHierarchy*: MemHierarchy is a highly configurable, flexible, Object-Oriented memory hierarchy simulator that models intra- and inter-node directory-based cache coherency architecture with MSI and MESI protocols.
3. *merlin*: It is a network simulation environment. It allows users to craft scalable simulations of core switching/routing infrastructure connected to various flavors of network interface controllers.

We implemented the proposed algorithm in C++ and generated N (user-defined) different trace files named $c\{i\}.trace$ and wrote a script in SST to simulate a memory hierarchy with only L1 private cache. Each L1 cache is connected to different ports of a router. At the last level of memory, we had a directory controller and a memory controller, both are connected to different ports in that router.

3.4.2 Memory Configuration

For a full simulation of SST, we need to adjust a number of parameters, we talk about only the important parameters.

1. *The clock*: 1GHz
2. *L1 cache size*: 4KB
3. *L1 Coherence protocol*: MSI or MESI
4. *Memory Controller clock*: 500MHz
5. *Main Memory Capacity*: 4GB
6. *Main Memory back-end*: memHierarchy.simpleMem
7. *debug for all components*: 1
8. *debug_level for all components*: 10

There is no inbuilt support for SI inside the SST *memHierarchy* module. But we can get around that if we set the coherence protocol as MSI and do not issue any write requests. The number of cores can be set during the simulation.

3.4.3 Trace File Format

Each trace file is for a core specifying if the core is generating the read or write requests. The format of each trace file is:

```
<time stamp> <r or w> <block address> <number of bytes>
```

Each entry should be *space* separated. To feed the trace files into prosperoCPU or SST simulation we used the *prospero.ProsperoTextTraceReader* module.

The timestamp difference between any two cores issuing a request to is set to 150μs. We have to give sufficient time to execute an issuing request. Here we only deal with stable states and stable transitions. It may be noted that this timestamp difference may change if the clock speed and other related parameters are changed.

3.4.4 Directory Configuration

The directory holds the blocks and their states and the respective shares, it makes sense to set to clock synchronizing with the main memory. Configuration of the directory is listed below,

1. *clock*: 500 MHz
2. *coherence_protocol*: MSI or MESI
3. *debug*: 1
4. *debug_level*: 10

It may be noted that the *coherence_protocol* for L1 should be the same as the one for the directory controller.

3.4.5 Router Configuration

All the private L1 caches are connected to different ports of the router. We used the *merlin.hr_router* API. The configuration of the router is:

1. *debug*:1
2. *debug_level*: 10
3. *number of ports*: $(number\ of\ cores)\ +\ 2$

3.4.6 Statistics, Debug and Simulation

To extract the statistics (refer to chapter-2) and debugging information for the simulation, we had to set the *stat_level* as 7 and set debug as 1 as shown in an earlier section. *debug_level* 10 is the highest level of debugging we can get out of SST. It shows all message passing, state changes, etc.

3.4.7 Test Case Generation

According to the above algorithms, the test cases are generated using C++. Two blocks are taken into account based on the set configuration. Consider the L1 cache size is $2^A KB = 2^{A+10}B$. Let us take 0 as the first block. Any block which is a multiple of 2^{10+A} will map to the same location as the 0^{th} block. For our simulation, since the L1 cache size is taken as $4KB = 2^{12}B$, two such blocks are 0 and 2^{12}.

3.4.8 Simulation

The trace files containing the transactions generated out of our algorithm were fed into the SST simulation script. We executed the Python script using the *sst* command. After simulation, a log

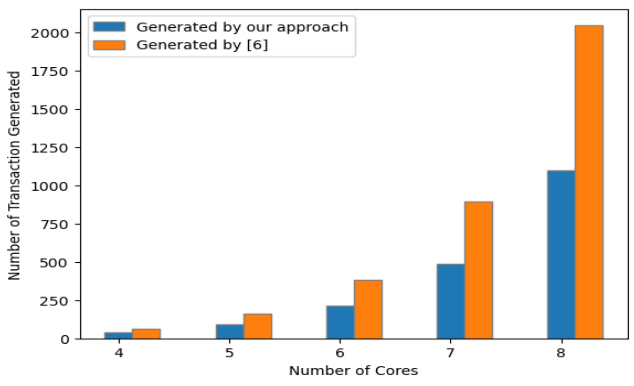

Figure 3.13: Comparison of the number of transactions generated by two approaches for SI protocol.

file was generated. The log file contained timing information, message passing between different components, changes of states inside caches, and a directory controller.

3.4.9 Validation

We examined the logs and all the stable states for the two blocks and their transitions were retrieved before each transaction to validate with our expected global states and transitions. Indeed, all states and transitions were covered as we discuss below.

3.5 Results and Discussion

For this experiment, we chose the number of cores to be small (4 / 8). We experimented with state coverage, transition coverage, and code coverage for SI, MSI, and MESI protocols. Lastly, we measured the memory usage for each algorithm.

3.5.1 State Coverage

From the experiments, it was clear that the framework covered all the states possible in the global FSM. Some of the states like I^N or S^N were visited multiple times.

3.5.2 Transition Coverage

The algorithm covered all transitions in the global FSM. In fact, to achieve the 100% transition coverage for SI protocol our algorithm needs nearly 50% fewer test sequences as compared to [5]. The improvement factor is 2 with respect to the algorithm proposed by [8]. For this comparison, we experimented with different numbers (up to 8) of cores for SI and compared the number of transactions generated with the number of transactions by [5]. The bar graph (Figure-3.13) supports our claim. From the figure, it is clear that the improvement is almost 2 for higher number of cores.

3.5.3 Code Coverage

For code coverage, we checked the messages passed through different components. These statistics can be found in the accumulator statistics for the L1 cache component. For this analysis, we only needed to monitor a 4-core simulation with MESI and look for the statistics of L1 caches *AckInv*, *FetchResp*, *FetchResp*, *FlushLine*, *FlushLineResp*, *GetS*, *GetSResp*, *PutE*, *WriteResp*, *FetchInvX*. We found that the *FetchXResp* of *mesi_l*1 module is covered.[††] This is a significant amount of code coverage.

3.5.4 Memory Usage

The peak memory usage of our algorithm implementation is gathered with Valgrind-3.15.0. We performed the experiment for different cache coherence protocols and different numbers of cores. The memory usage is shown in Table-3.1. The memory requirement for our algorithm depends on

Protocol	4	6	8
SI	72.25KB	72.69KB	74.24KB
MSI	72.125KB	72.125KB	72.125KB
MESI	72.47KB	72.85KB	74.40KB

Table 3.1: Memory Usage for our approach for different protocols

the number of recursions and does not increase much over time. Our algorithm does not keep any additional information about states / transactions except for the stacks due to recursive function calls.

3.5.5 Tests and Simulation Time

Table-3.2 depicts the test generation and simulation time and the number of messages generated. During test generation, we needed to generate the trace file to feed into the SST Python script. This is why our test generation time includes the file read and write time, which gives a slightly higher value than expected. The simulation time can be extracted from the log file generated after the successful simulation of the script. The total number of messages is the count of the total number of messages inside the memory hierarchy to successfully simulate the script.

	Test Generation Time	**Simulation Time**	**Number of Messages**
SI 4 Cores	0.023s	6.05 μs	215
SI 8 Cores	0.48s	164.45 μs	6487
MSI 4 Cores	0.17s	50.63 μs	1907
MSI 8 Cores	7.53s	2.03 ms	80107
MESI 4 Cores	0.26s	85.1 μs	2676
MESI 8 Cores	15.62s	2.58 ms	18233

Table 3.2: Table for Test Generation time, simulation time, and Number of Messages with respect to core and coherence protocol

The test generation time is high because it requires a considerable time to store the instructions in the *trace* files to feed those into *prospero*.

[††]The *mesi_l*1 module contains all the necessary functions to implement MESI and MSI protocol inside SST.

3.6 Conclusion

In this chapter, we propose a novel eviction-based approach to obtain an efficient framework for verifying cache coherence protocols. Other complex protocols like MSI, and MESI are extensions of SI. Having an efficient algorithm for verifying SI protocols certainly helps in complex protocols as well. We demonstrate the effectiveness of the algorithm using the SST simulator.

However, in this chapter, we assume a very simple memory model. In the case of multilevel caches, our algorithm does not fare well. In the next chapter, we extend the framework for multi-level caching by solving these limitations.

Chapter 4

Directed Testing of Cache Coherence for a Multi-Level Cache Hierarchy

In this chapter, we discuss the drawbacks of applying state-of-the-art directed testing directly for the multi-cache hierarchy model. Following that we propose a new framework for test generation for a multi-level cache hierarchy. Finally, we show the validation of the algorithms in SST Simulation as shown in Chapter-3.

4.1 Introduction

The algorithm, discussed in the previous chapter assumes that the underlying memory model is simple, that is every core has its own L1 private cache. The cache coherence protocol we assumed is also very simple. Modern computing systems do not use a SI protocol. Nowadays the caches are write-back type and in the memory hierarchy, each core has at least two levels of private caching. For example, Intel i7 processors have two levels of private caches and one level of shared caches, and L1 is broken into two parts (data and instruction cache). With the advancement of technologies, the cache coherence protocol became very complicated. The simpler protocols are MSI and MESI. Other complex cache coherence protocols like MOSI, MOESI, MESIF [12], and MEUSI are based on these protocols only.

For exclusive verification, a verification engineer attempts to execute every possible scenario to find a potential bug. With increasing complexity, the state space under test increases exponentially. Directed testing, which shows promises to reduce the test sequence by a great margin also suffers from state space explosion problems. To tackle these problems researchers proposed quotient space [5], which is a symmetry reduction technique. By defining some equivalence classes of states and restricting state space representatives, verification techniques can be used to deal with a large number of states. Clarke et al [1] exploited symmetry reduction techniques in model checking.

As of now, directed testing methods have been developed for memory hierarchies with single-level private cache. To the best of our knowledge, directed testing has not been extensively used for verifying cache coherence protocols because of its limitations in multi-level caching due to state space exploration. Our contribution in this chapter is a framework for test generation and validation of cache coherence protocols in a multi-level cache hierarchy.

The remaining chapter is organized as follows. Section-4.2 provides some background and motivation for this chapter. Section-4.3 proposes the approach. Experimental setup and results are presented in

Sections-4.4, 4.5 and 4.6. Finally, in Section-4.7 we conclude the chapter.

4.2 Background and Motivation

In the previous chapter, we explained how we can apply directed testing for a single-level cache hierarchy. In this chapter, we talk about applying directed testing methods for SI, MSI, and MESI protocols for a multi-level cache hierarchy. To the best of our knowledge, no work has been done to extend directed testing in the multi-level cache hierarchy. While designing our framework we need to keep following things in mind, as discussed below.

State Space Explosion Problem

Previously it was shown that the state space of the SI protocol for n cores is a n-hypercube. Here we need to consider the states for L2 caches too. For the SI protocol, L2 caches will have 2 states for each block inside each core. For n cores the number of states will be 2^n. Considering all the possible states of the L1 and L2 cache, the number of possible states in the global state space for SI protocol would be $2^n \times 2^n = 4^n$. However, due to the fact that caches are inclusive[*], an address which is in S for some core's $L1$ cannot be in I state for $L2$ of that core, but for an address which is in I state in L1, can be in S or I state at $L2$. The states in the state space diagram/ graph in multi-level caching can be labeled as a tuple $(l1, l2)$, where $l1 \in$ $set\ of\ state\ vectors\ of\ L1$ and $l2 \in$ $set\ of\ state\ vectors\ of\ L2$. It may be noted that the length of $l1$ and $l2$ will be the same, both being equal to the number of cores. As stated above, $l1$ and $l2$ should follow the below conditions,

1. $\forall\ i\ \in\ [0, n-1]\ if\ l1[i] = I\ and\, l2[i]\ \in\ \{I, S\}$
2. $\forall\ i\ \in\ [0, n-1]\ if\ l1[i] = S\ and\ l2[i] = S$

Without loss of generality, assume a tuple $(l1, l2)$, where $l1 = I^x S^{(n-x)}$. Following the conditions stated above, $l2$ can have the labels $\{I, S\}^x S^{(n-x)}$. The number of such labels is 2^x. For every $l1$ where number of I is x where $x \in [0, n]$, the possible number of $l2$ states is 2^x. The number of possibilities that $l1$ state has x number of I in its labeling is ${}^nC_x$. Total number of states where $l1$ state has x number of I in its labelling is ${}^nC_x \times 2^x$. The total number of states in the state space of the SI protocol for L1 L2 cache memory hierarchy is $\sum_{x=0}^{n} {}^nC_x \times 2^x = 3^n$. In MSI, caches have an extra state beside I and S. Whenever a cache wants to write to some address, the cache gets that address in M state. Naturally, all the caches who were sharing that address should evict that address. The state should follow the following condition, there exists just one $i \in\ [0, n-1], l1[i] \in \{I, M\}, l2[i] = M$ & $\forall j\ \in\ [0, n-1]\ , j \neq\ i,\ l1[i] = l2[i] = I.$

The number of states with M is $2n$. The total number of states in the state space of MSI for L1, L2 cache memory hierarchy is $3^n + 2n$. In the case of the MESI protocol, caches have an extra state beside M, S, and I. When a cache wants to read some particular address and that cache is the only one to share that address/read that address, the cache reads the address in E mode. The labeling of the state should follow the following condition, there exists just one $i \in\ [0, n-1], l1[i] \in\ \{I, E\}, l2[i] = E$ & $\forall j\ \in\ [0, n-1]\ , j \neq\ i\ l1[i] = l2[i] = I$. The total number of such states with E is $2n$. The total number of states in the state space of MESI for L1, L2 memory hierarchy is $3^n + 2n + 2n = 4n + 3^n$.

[*]An inclusive caching scheme is where the same data can be present in both the L1 and L2 caches

$$\begin{tikzcd}
\text{II,II} \arrow[r, dash] \arrow[d, dash] & \text{IS, IS} \arrow[r, dash] \arrow[d, dash] & \text{II, IS} \arrow[d, dash] \\
\text{SI, SI} \arrow[r, dash] \arrow[d, dash] & \text{SS, SS} \arrow[r, dash] \arrow[d, dash] & \text{SI, SS} \arrow[d, dash] \\
\text{II, SI} \arrow[r, dash] & \text{IS, SS} \arrow[r, dash] & \text{II, SS}
\end{tikzcd}$$

Figure 4.1: State Space of SI protocol for two levels of Private Cache

In this type of memory hierarchy, in every state of the global FSM, we can issue 2 types of instructions for every core. So for every state, we have roughly exponentially many transitions[†]. The total number of transitions is also exponential. For example for $n = 2$, the number of states in the state space of SI protocol is 9. The transitions are shown in Figure-4.1

Problem of Missing States

To understand this problem we use an example of a memory system with 4 cores and two levels of private caches for each core. We consider the underlying cache coherence protocol is MESI. We consider Algorithm-. In the second *for loop*, our algorithm will execute the *visitHypercubeUtil* function twice. For $i = 0$ and $i = 1$, the transactions generated are as follows,

```
1. Load B0 block by 0th-core
2. Load B0 block by 1st-core
3. Load B1 block by 0th-core
4. Load B1 block by 1st-core
5. Load B0 block by 1th-core
6. Load B0 block by 0th-core
7. Load B1 block by 0th-core
8. Load B1 block by 1st-core
```

For ease of our understanding, we look into the state transition of the $B0$ block only. For the first 4 instructions in the above trace, the address-0 will cover the transitions, $IIII - EIII - SSII - ISII$. But when we execute the fifth instruction, we expect the 1^{st} core to go to E state but it goes to S state as shown in Figure-4.2. As we introduced another level of cache (L2), after one round($i = 0$), the address got cached into L2. So even though the states in the L1 were changing, from the memory perspective, it was still cached by the cores because memory always keeps track of the last level of private cache, in our case L2. With this simple example, we have shown that due to L2, our existing methods will be missing some states of L1 as well. In conclusion, to tackle the state space explosion problem and missing states issue, we need a method that is scalable and can cater to the needs of ever-increasing complexity.

[†]Edges are bidirectional

$$\begin{tikzcd}
\text{IIII} \arrow[r, "(1)"] \arrow[d, bend right, "(5)"'] & \text{EIII} \arrow[d, "(2)"] \\
\text{ISII} \arrow[u, "(4)"'] & \text{SSII} \arrow[l, "(3)"]
\end{tikzcd}$$

Figure 4.2: Abnormalities in State Transitions, transitions labeled in accordance with the traces.

4.3 Our Approach

It is clear that with the increase in levels of cache, the state space increases exponentially. Also, note that in the above section, we assume that the caches are inclusive and each core has private L1 and L2 caches. Depending on the manufacturer this memory hierarchy changes and so does the state space of the hierarchy. Building different frameworks for different memory models is not a feasible solution as the time required to develop a framework and validate the memory model is a time-consuming task.

Instead of covering the entire state space, we wanted to reduce the state space to a reasonable amount. So instead of focusing on the coverage of the entire L1 L2 cache state space, we focused on covering all the possible states for L1 caches only. Two things can happen for accessing a block inside L1: a cache hit or cache miss. In case of a cache miss, the processor core needs to request the block to the lower level of memory. Our hypothesis in both cases is, if a bug occurs during execution, it will reflect in L1 states and its corresponding data value.

To tackle the issue of missing state, we issue a $flush\_address$ operation whenever we are about to make a transition from any state to all-I state. $flush\ address$ instruction will flush the blocks from all the cores, forcing the system to evict the blocks from all the sharing cores. The $flush\_address$ operation will invalidate any copy inside any L1 and most importantly L2 caches.

4.3.1 SI and MSI Protocol

The problem of missing states occurs only when the coherence protocol is MESI, since on getting a read request, a block changes its state depending on memory controller states. For SI and MSI, the states of the memory controller do not come into play as the state always changes to S. For MSI and SI, we can apply Algorithm-3 and Algorithm-10 to cover all the states in the state space of L1.

4.3.2 MESI Protocol

Our framework for the MESI protocol is based on the concept of covering only L1 states and the corresponding transitions as shown in Algorithm-9. Our proposal is based on the optimized framework of SI cache coherence test generation, as stated in Chapter-3 and other algorithms inspired from [5]. As stated earlier, two blocks are chosen such that they are mapped to the same cache line at L1. Let us call these blocks $B0$ and $B1$. The algorithm is divided into several function calls as shown in [5]. The function calls indicate how we decompose the state space of the MESI protocol. This type of decomposition is adopted from [8]. We explain each procedure in detail below.

Algorithm 9 Algorithm for Test Generation of MESI Protocol

```
Input: User to define the number of cores $N$
Output: Test sequences
1: createTestsSI(N)
2: visitClique(0)
3: visitE()
4: $states \leftarrow I^N$
5: visitOtherMESI(0, states)
6: visitOtherE()
```

4.3.3 createTestsSI

This function is adopted from the previous chapter. It traverses the hypercube just as in the case of SI and MSI, even though the hypercube is somewhat different because of the E state in the MESI protocol. However, instead of evicting every time, sometimes we do the $flush\_address$ operation to reset the memory system into an all-I state. When we are about to reach the all-I state, instead of eviction by another block, we do the flush operation. This modification needs to be done to Algorithm-2. For this, we introduce a *depth* variable in the *visitHypercubeUtil* method. Whenever

Algorithm 10 createTestsSI: Modified Test Generation for SI Protocol

```
Input: User to define the number of cores $N$
Output: Test sequences
1: for $i = 0$ to $N - 1$ do
2:     visitHypercubeUtil(N,i,N, 0)
3: end for
```

Algorithm 11 visitHypercubeUtil

```
Input: M, r, N, depth
Output: Test Sequences
1: $p \leftarrow (M + i)\%N$
2: print(load $B0$ block by $p^{th}$-core)
3: for $i = 1$ to $M - 1$ do
4:     visitHypercubeUtil(i, r, N, depth+1)
5: end for
6: if $depth \neq 0$ then
7:     print(Read $B1$ block by $p^{th}$-core)
8: else
9:     print(Flush $B0$ block by $p^{th}$-core
10: end if
```

this function is called recursively, the variable *depth* increases for the next call. When *depth* is 0, we are in a global shared state, where a single core is sharing the B0 block, following which we issue a $flush\ operation$.

4.3.4 visitClique

All the global-modified states and the global-invalid state will form a clique. On reaching one modified state, we also visit some global-shared states which can be visited from global-modified states as well.

Algorithm 12 visitClique: Method to visit the Clique between modified states

```
Input: Index, N
Output: Test Sequences
1: print(write $B0$ block by $Index^{th}$-core)
2: print(Flush $B0$ block by $Index^{th}$-core)
3: print(write $B0$ block by $Index^{th}$-core)
4: for $i = 0$ to $i = N - 1$ do
5:     if $i \neq p$ then
6:         print(Read $B0$ block by $i^{th}$-core)
7:         print(Write $B0$ block by $p^{th}$-core)
8:     end if
9: end for
10: for $i = Index + 1$ to $i = N - 1$ do
11:     print(Write $B0$ block by $i^{th}$-core)
12:     if $i == Index + 1$ then
13:         traverseClique(i)
14:     end if
15:     print(Write $B0$ block by $p^{th}$-core)
16: end for
```

Consider an example of $n = 3$ for the *visitClique* function. We first cover the transition IIS-MII, MII-III, and III-MII transition in lines-1,2,3 of Algorithm-12. We start the execution from IIS since after the end of the function call to *visitHypercube*, the state vector of block $B0$ will be IIS. We can go back to the III state but it is unnecessary. In the recursive calls to the function, the transitions MII-IMI, and IMI-IIM are visited. In the next iteration, IIM-MII and MII-IIM are visited. To improve efficiency, we also traverse all global shared states that are bidirectionally reachable from the current global-modified state. Below are the transactions generated from the above algorithm. Some of the transactions are omitted for ease of understanding.

```
1. write B0 block by 0th core
2. Flush B0 block by 0th core
3. Write B0 block by 0th core
4. Read B0 block by 1st core
5. Write B0 block by 0th core
6. Read B0 block by 2nd core
7. Write B0 block by 0th core
8. Write B0 block by 1st core
...
16. Write B0 block by 2nd core
...
24. Write B0 block by 1st core
25. Write B0 block by 0th core
26. Write B0 block by 2nd core
27. Write B0 block by 0th core
```

The steps are shown in Figure-4.3.

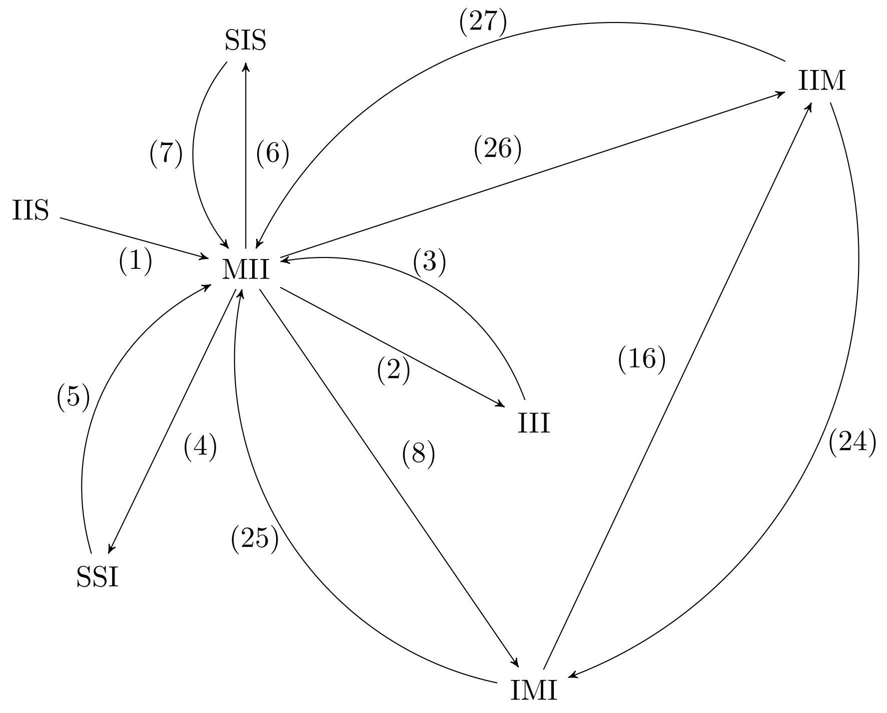

Figure 4.3: Transitions during visitClique function in MESI protocol. For ease of understanding, we showed only the first few transitions, later transitions are omitted. The transitions are labeled by the transaction issued.

4.3.5 visitE

The *visitE*() function helps to visit all the transitions between global-exclusive states and the hypercube. This function is also adopted from [5]. We incorporate our *flushaddress* concept into this function.

Algorithm 13 Method to visit the Clique between modified states

```
Input: N
Output: Test Sequences
 1: print(Flush $0^{th}$-address by $0^{th}$-core)
 2: for $i = 0$ to $i = N - 1$ do
 3:     for $j = 0$ to $j = N - 1$ do
 4:         print(Read $B0$ block by $i^{th}$-core)
 5:         print(Read $B0$ block by $j^{th}$-core)
 6:         if $i \neq j$ then
 7:             print(Read $B1$ block by $j^{th}$-core)
 8:             print(Flush $B0$ block by $i^{th}$-core)
 9:         end if
10:     end for
11: end for
```

After the execution of the *visitClique* function, the state vector for $B0$ block does not come back to the all-I state. To do that in the *visitE* function we first flush the blocks forcing the state vector of block $B0$ to be all-I.

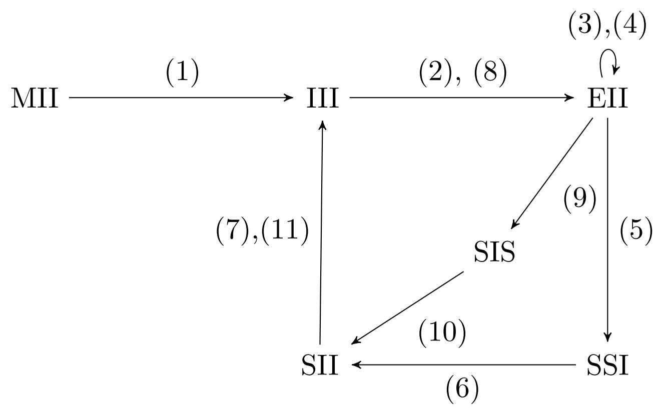

Figure 4.4: Execution of the visitE function for first iteration $i = 0$. The transitions are labeled by the traces. Notice that some of the transitions are repeated.

Considering the previous example with 3 cores. In the first iteration of the inner loop, $j = 0$ covers the transition III-EII. The second iteration of the inner loop, $j = 1$ covers the transitions EII-SSI, SSI-SII, and SII-III. The last iteration $j = 2$ covers the transitions, EII-SIS, SIS-SII, and SII-III. The traces, generated by Algorithm-13 are shown below. For ease of understanding, we only show the transaction for $i = 0$.

```
1. Flush B0 block by 0th core
2. Read B0 block by 0th core
3. Read B0 block by 0th core
4. Read B0 block by 0th core
5. Read B0 block by 1st core
6. Read B1 block by 1st core
7. Flush B0 block by 0th core
8. Read B0 block by 0th core
9. Read B0 block by 2nd core
10. Read B1 block by 2nd core
11. Flush B0 block by 0th core
...
```

It is important to note that we cannot apply the previous *visitClique* method for global-exclusive states as they do not form a clique as global-modified states. Figure-4.4 shows the detailed steps.

4.3.6 visitOtherMESI

visitOtherMESI is called to traverse all uni-directional transitions that occur between global-shared states and global-modified states. We cannot cover them by the above procedures. This algorithm is a brute force algorithm in the sense that, we go to each shared state, and from that, we visit the modified states. We cannot come back from modified states to those states directly. We chose the shortest path from a modified state to that shared state and repeat the same.

To choose the shortest path, we can use Dijkstra's algorithm or simple BFS but to do that we need to store the entire global FSM. A nice observation is that the shortest path from a modified state to a global-shared state will be the transition from a modified state to a global invalid state, a path from the invalid state to the global-shared state. To get the path from an invalid state to a global-shared state we can perform the corresponding load operations to reach the expected global-shared state.

Algorithm 14 visitOtherMESI: Method to traverse all global shared states to global modified states

Input: Index, states
Output: Test Sequences

```
if $Index == sizeof(temp)$ then
    if $states \leftarrow I^N$ then
        Return
    end if
    for $i = 0$ to $i = N - 1$ do
        print(write $B0$ block to $i^{th}$-core)
        print(Flush $B0$ block from $i^{th}$-core)
        if states has exactly one $S$ then
            $pos \leftarrow$ position of single $S$ in states
            $auxPos \leftarrow pos + 1$
            if $pos \geq sizeof(states) - 1$ then
                $auxPos \leftarrow 0$
            end if
            print(Read $B0$ block by $pos^{th}$-core)
            print(Read $B0$ block by $auxPos^{th}$-core)
            print(Read $B1$ block by $auxPos^{th}$-core)
        else
            for $j = 0$ to $j = sizeof(states) - 1$ do
                if $temp[j] == S$ then
                    print(Read $B0$ block by $j^{th}$-core)
                end if
            end for
        end if
        print(Write $B0$ block by $i^{th}$-core)
        print(Flush $B0$ block by $i^{th}$-core)
    end for
    Return
end if
$temp[Index] = I$
visitOtherMESI(Index+1, temp)
$temp[Index] = S$
visitOtherMESI(Index+1,temp)
```

$$\begin{tikzcd}
\text{MII} \arrow[r, bend left, "{(2),(7)}"] & \text{III} \arrow[l, "(1)"'] \arrow[r, "(3)"] & \text{EII} \arrow[d, "(4)"] \\
\text{SII} \arrow[u, "(6)"] & & \text{SIS} \arrow[ll, "(5)"]
\end{tikzcd}$$

Figure 4.5: Transitions due to visitOtherMESI function for SII state vector. The transitions are labeled by the corresponding transactions that cover them.

In Algorithm-14, we take as input a string of length the same as the number of cores. The string contains all I characters initially. We generate all possible combinations of characters I and S by the recursive structure of the function. We discard the string with all-I and consider the strings which have at least one S. Based on the string, we issue the corresponding load instructions for those cores for block $B0$. But in the case of a single S in the string, we cannot simply issue a single load for the core. Since in every iteration, we are starting from the all-I state, issuing a single load changes the state to a global-exclusive state. To avoid that we issue two additional loads (load block $B0$ for another core, followed up by another load for block $B1$ for the former core). Consider the previous example of 3 cores.

Case 1: Consider the string is SII. We first write and flush the block $B0$ to be sure the state vector of block $B0$ is III. Then lines-14,15,16 are executed and at this point, we are in the state SII. Next, we issue a write instruction for core 0 ($i = 0$). We can see that the SII-MII transition is covered. We do it for the rest of the iterations. Below is a snapshot of transactions generated by our approach.

```
1. write B0 block by 0th core
2. Flush B0 block by 0th core
3. Read B0 block by 0th core
4. Read B0 block by 2nd core
5. Read B1 block by 2nd core
6. write B0 block by 0th core
7. Flush B0 block by 0th core
...
```

The detailed transitions are shown in Figure-4.5. It may be noted that some of the transitions are repeated.

Case 2: Consider the string SSS. As done in case-1, we at first issue a write and a flush operation. Then we issue three load instructions to reach the state SSS. We then issue a write instruction for core-0 (in the first iteration, $i = 0$) to cover the transition SSS-MII. The snapshot of the traces is shown below.

```
1. Write B0 block by 0th core
2. Flush B0 block by 0th core
3. Read B0 block by 0th core
4. Read B0 block by 1st core
```

$$\begin{tikzcd}
III \arrow[r, "(3)"] \arrow[d, "(1)"] & EII \arrow[dd, "(4)"] \\
MII \arrow[u, bend left, "{(2),(7)}"] & \\
SSS \arrow[u, "(6)"] & SSI \arrow[l, "(5)"]
\end{tikzcd}$$

Figure 4.6: Transition due to visitOtherMESI function for SSS. The transitions are labeled by the corresponding transactions that cover them.

```
5. Read B0 block by 2nd core
6. Write B0 block by 0th core
7. Flush B0 block by 0th core
...
```

The detailed transition is shown in Figure-4.6. Here also we can see some of the transitions are being repeated.

4.3.7 visitOtherE

The *visitOtherE* function covers the remaining transitions corresponding to the exclusive states to modified states as shown in Algorithm-15. These transitions are also uni-directional. All exclusive states to all modified states are possible but the reverse states are not possible.

Algorithm 15 visitOtherE: Method to traverse all exclusive to modified states

```
Input: N
Output: Test Sequences
1: for $i = 0$ to $i = N - 1$ do
2:     print(Read $B0$ block by $i^{th}$-core)
3:     for $j = 0$ to $j = N - 1$ do
4:         print(Write $B0$ block by $j^{th}$-core)
5:         print(Flush $B0$ block by $j^{th}$-core)
6:         print(Read $B0$ block by $i^{th}$-core)
7:     end for
8:     print(Flush $B0$ block by $i^{th}$-core)
9: end for
```

After the execution of *visitOtherMESI*, the state vector of block $B0$ will be III. Consider the previous example with 3 cores. For the first iteration $i = 0$, we first cover transition III-EII. Then for the first iteration of the inner loop ($j = 0$), the transitions EII-MII and MII-III are covered. In the next two iterations of the inner loop, EII-IMI and EII-IIM transitions are covered. Similarly, other transitions are covered in the next iterations of the outer loop. Below is a snapshot of the traces for the first iteration of the outer loop ($i = 0$).

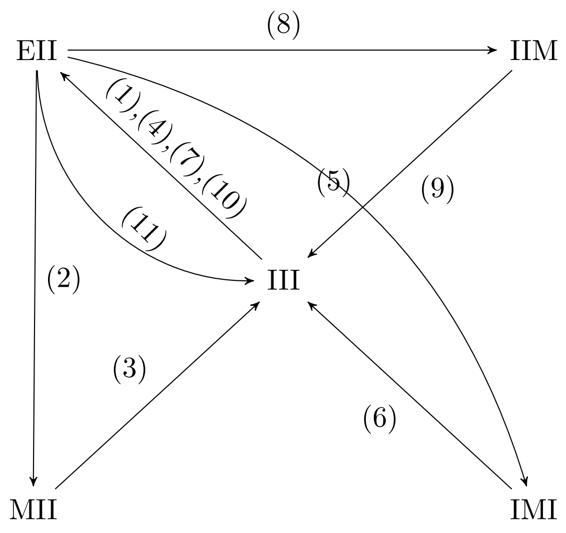

Figure 4.7: Transition due to visitOtherE function for SSS state vector. The transitions are labeled by the corresponding transactions that cover them.

```
1. Read B0 block by 0th core
2. Write B0 block by 0th core
3. Flush B0 block by 0th core
4. Read B0 block by 0th core
5. Write B0 block by 1st core
6. Flush B0 block by 1st core
7. Read B0 block by 0th core
8. Write B0 block by 2nd core
9. Flush B0 block by 2nd core
10. Read B0 block by 0th core
11. Flush B0 block by 0th core
...
```

The detailed transitions are shown in Figure-4.7. The time complexity of Algorithm-9 is $\mathcal{O}(n^2 2^n)$ and space complexity is $\mathcal{O}(n)$. The detailed discussion on the number of transactions generated by the algorithm and the proofs for the time and space complexity are included in the Appendix.

4.4 Experiments

We performed two sets of experiments. The first one (Experiment-1) was done to validate the proposed framework and in the second experiment (Experiment-2), we implemented a directed testing framework to extend an existing SST component developed by the Computer System Architecture (CSA) team at the Interuniversity Microelectronics Centre (IMEC), Belgium where this dissertation was carried out. During the course of a 4-month internship, we built the directed tester and integrated it with their in-house tester. All experiments and simulations were done on a Windows Subsystem on Linux (WSL), on an Intel(R) Core(TM) i5-103119U CPU with 16GB RAM and Intel(R) UHD 8GB GPU.

4.5 Experiment-1

4.5.1 Simulation Components

We used SST for our simulation purpose. For simulation, we used three components of SST, *memHierarchy*, *prospero*, and *merlin*. *Prospero* component does not support flush operation by default. We developed a similar component as *prospero* with *flush* operation. It interfaces with the *standardInterface* API of SST-Core.

4.5.2 Memory Hierarchy Configuration

Our configuration for L1, directory controller, memory controller, and router configuration remains the same as in Chapter-3 except for the *coherence_protocol*. Our L2 configuration is as mentioned below.

1. *cache_size*: 2MB
2. *Associativity* : 2
3. *debug*: 1
4. *debug_level* : 10
5. *coherence_protocol* : MSI or MESI

Note that the *coherence_protocol* for L1, L2, and, directory controller should be the same for the experiment. According to the configuration, two blocks were taken into account as done in Chapter-3. The trace files were generated based on the proposed framework and fed into the SST Simulation script. After the simulation, a log file was generated. We analyzed the log files and the messages passing between the caches, directory controller, router and memory controller, and most importantly the change of states. We found that all the transitions were covered.

4.5.3 Results and Discussions

- *State Coverage:* The results demonstrate that the framework covered all possible L1 states in the global FSM. It is worth noting that some states like I^N or S^N were visited multiple times which is consistent with the expectations.
- *Transition Coverage:* As this method focuses only on the transitions of the L1, all transitions in the FSM of the L1 and L2 state machine were not covered, however, all transitions of the global FSM of L1 were covered.
- *Code Coverage:* Using the proposed framework, we were able to cover the following methods in the *mesi_l*1 module of the memory hierarchy: *AckInv*, *FetchResp*, *FetchResp*, *FlushLine*, *FlushLineResp*, *GetS*, *GetSResp*, *PutE*, *WriteResp*, *FetchInvX*. *FetchXResp*. This achieved significant code coverage.
- *Number of Messages:* While calculating the number of messages, we considered the messages through the entire memory system. For ease of understanding, we only provide the transitions for the lower number of cores in Table-4.1.

- *Test Generation and Simulation Time:* Table-4.1 depicts the test generation and simulation time and the number of generated messages.
- *Memory Usage:* We also show the memory usage incurred by our approach.

No. of Cores	**Test Generation Time**	**Simulation Time**	**# Messages Generated**
SI 4 Core	0.021s	6.07 μs	138
SI 8 Core	0.48s	164.472 μs	4419
MSI 4 Core	0.21s	50μs	2445
MSI 8 Core	10.89s	2.03 ms	10842
MESI 4 Core	0.28s	91.27 μs	4019
MESI 8 Core	14.51s	2.73487ms	135843

Table 4.1: Table for test generation time, simulation time, and total number of messages generated

4.6 Experiment-2

CSA Group at IMEC, Belgium had built an SST component to test the cache coherence protocol, named *memHierarchyTester*. It uses the random testing framework to test the cache coherence protocol. We incorporated our directed testing framework into that component. We then built components with the existing SST 12.1.0 setup. The SST-12.1.0 is built using g++9.4.0. We do not need a *prospoero* module in this experiment since the framework will be generating test cases on-the-fly and issuing requests simultaneously with the simulation. The most important parameters are given in Table-4.2. Some of the parameters are inbuilt into the open-source version of the *memHierarchy* module of SST. Random Seed is a parameter specific to our memory hierarchy tester. If the random seed is set to zero, the tester will behave as a directed tester otherwise it will act as a random tester. For our experiment, we used 331 as a seed for random testing. We carried out the experiment for the MESI protocol only. We monitored the cache behaviors, data read and write, and message passing. The generated log file contained a lot of intermediate states. For our analysis, we did not consider those states and transitions. We now compare our approach with that of the random tester according to certain metrics.

Parameters	**value**
Verbose	7
Number of Check	(user-specific)
Number of Cores	(user-specific)
CPU Clock	1GHz
L1 cache Size	8KB
L1 associativity	1
Main Memory Size	1GB
Coherence Protocol	MESI
Debug	1
Debug_level	10
Random Seed	0, 331

Table 4.2: SST Simulation Parameters

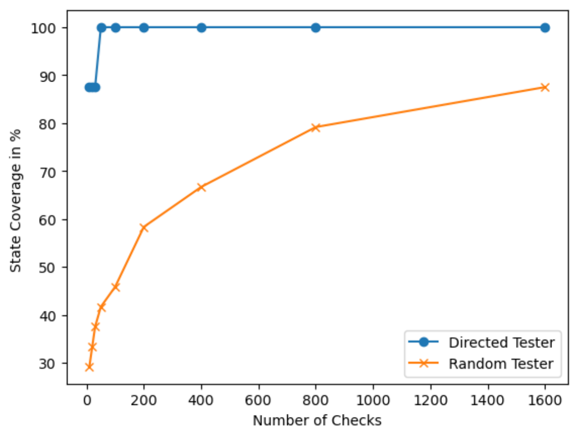

Figure 4.8: State Coverage Comparison of Random and Directed Tester for 4 cores

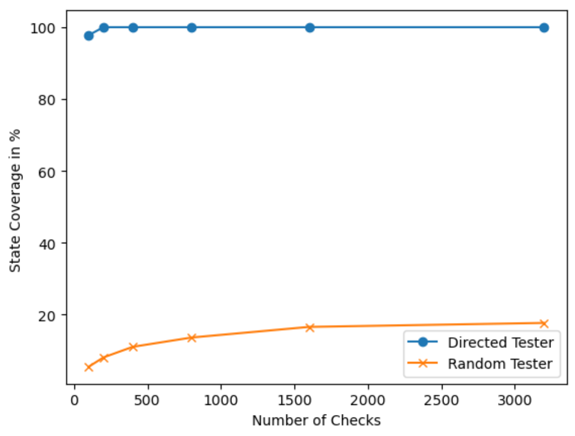

Figure 4.9: State Coverage Comparison of Random and Directed Tester for 8 cores

Figure 4.10: State Coverage Comparison for different cores with different testers

4.6.1 Coverage

Directed testing heavily relies on the tester's knowledge and ability to identify critical scenarios. It may not cover all possible execution paths and edge cases, leaving some untested areas of the cache coherence protocol. But directed testing is intended to cover all possible states and transitions between the stable states in the global FSM. Random Testing on the other hand, does not depend on the tester's knowledge. It may cover some uncertain areas that a directed tester is unable to cover.

We experimented with $n = 4$ and $n = 8$ cores with a different numbers of checks. The number of checks in the memory hierarchy tester that we developed is different from the number of transactions that we generate. We generated the transactions and the tests were simulated inside SST one by one. Whenever we compare the data value received with the golden reference, we conclude one check is complete. For one check we may need to issue more than one instruction.

State Coverage

To compare the state coverage we experimented with $n = 4,\ and\ 8$ cores. We plot two graphs of the number of checks and the percentage of the states covered for two different numbers of cores. Ideally, MESI has $2n + 2^n$ number of states in a global state machine for n cores. For $n = 4$, we tested with 10, 20, 30, 50, 100, 200, 400, 800, and 1600 checks. For $n = 8$ we tested with 100, 200, 400, 800, 1600 checks. In the two plots (Figure-4.10) we can see the state coverage in directed testing is quick. For 4 cores, the random tester took 4000 checks to cover all the states whereas it took around 10000 checks to cover all the states for 8 cores. The corresponding numbers were 200 and 500 for the directed tester.

Transition Coverage

We experimented with $n = 4\ and\ 8$ cores as we did for state coverage. We plot two graphs of the number of checks and the percentage of the transitions covered for two different numbers of cores. We experimented with 10, 40, 100, 200, 400, 800, 1600, and 3200 checks.

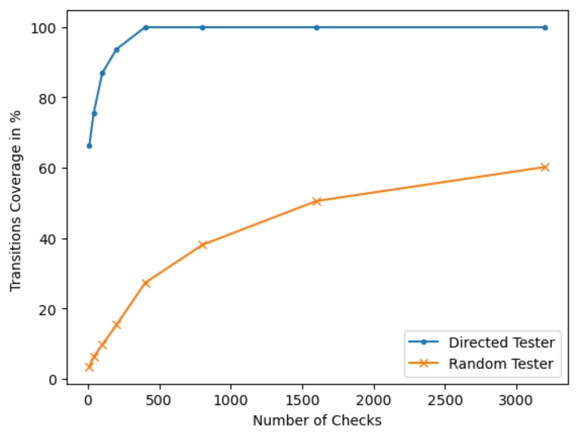

Figure 4.11: Transition Coverage Comparison of Random and Directed Tester for 4 cores

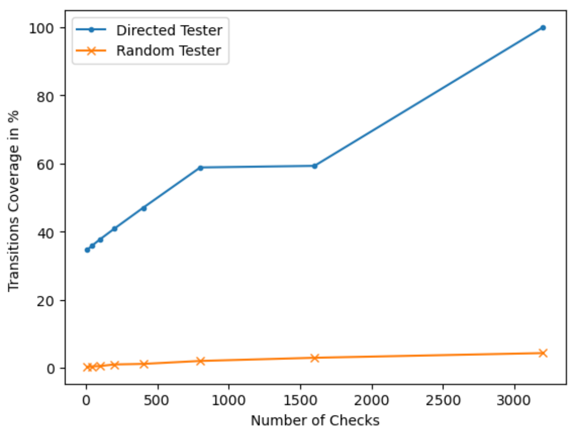

Figure 4.12: Transition Coverage Comparison of Random and Directed Tester for 8 cores

Figure 4.13: Transition Coverage Comparison for different core with different testers

In the two plots (Figure-4.13), we see the transition coverage in directed testing is quick. In directed testing, for 4 cores the entire state space was covered within 500 checks and for 8 cores the entire state space was covered in 3200 checks. In random testing, within 3200 checks we hardly completed 50% of the state space and for 8 cores we completed just 15% of the entire set of transitions.

Code Coverage

To compare the code coverage, we set the number of cores to 4. We did not need to vary the number of cores since the code coverage does not depend on this. However, we needed to vary the types of caches used. We set the number of checks to 3200.

To analyze the code coverage, we set the stats level to 7 and stored the statistics in a CSV file. The format of the CSV file is given below,

```
ComponentName, StatisticName, StatisticSubId, StatisticType, SimTime, Rank, Sum.u64, SumSQ.u64, Count.u64, Min.u64, Max.u64
```

We individually checked the component name, and examined the StatisticName starting with *eventSent_* which have *Count.u64* greater than 0. For both cases, we set the L1 cache controller module to be *mesi_l*1 and changed the L2 cache controller module. Our analysis revealed the following.

1. For L1, *handleGetS*, *handleGetX*, *handleFlushLine*, *FetchResp*, *handleFetchXResp*, *handleGetSResp*, *handleWriteResp*, *handleFlushLineResp* were covered.

2. For L2 cache, *handleGetS*. *handleGetX*, *handleFlushLine*, *handleFetchResp*, *handleFetchXResp*, *handleFetchXResp*, *handleAckInv*, *handleGetSResp*, *handleGetXResp*, *handleFlushLineResp*, *handleFetchInv*, *handleFetchInvX*, *handleInv* were covered.

4.6.2 Scalability

As the complexity of the cache coherence protocols increase, designing comprehensive directed test cases becomes challenging. The number of possible scenarios grows exponentially, making it difficult to achieve complete coverage. Random testing can handle complex cache coherence protocols more effectively since it does not rely on human-designed test cases. It can generate a large number of test cases rapidly, providing broader coverage.

4.6.3 Reproducible Results

Directed testing follows a predefined set of tests, it allows for reproducible test results, which is useful for debugging and fixing issues. The same cannot be said for random testing. The transition coverage, and state coverage changes for different seed values.

4.6.4 Debugging

Due to the lack of a predefined pattern or scenario, debugging random test failures can be challenging. It can be difficult to trace the root cause and reproduce the exact conditions that led to the failure. Due to shorter traces, debugging in the case of directed testing is easier.

4.6.5 Case Study

We finish this section by showing a small case study (triggering a bug inside SST code) where we showed the directed testing framework worked better.

There is a well-known bug inside the SST memory hierarchy module. The information about the bug can be found in the official GitHub repository of SST Elements (issue no 1934). The bug is called *Read after Write Bug*. Random Tester was able to catch that bug after 1000+ checks.

```
MemHierarchyTester[performCallBack:314]: Action/check failure: proc 1444 at the
address: 464, byte_number: 0, the expected value is 3f but received 99 instead.
Time: 6002000
```

Our directed tester was able to catch the bug in less than 100 checks. The output is given below,

```
MemHierarchyTester[performCallBack:314]: Action/check failure: proc 47 at the
address: 0, byte_number: 0, the expected value is 75 but received 5d instead.
Time: 4373000
```

Also, the time required to find the bug for the directed testing framework is less.

4.7 Conclusions

In this chapter, we have proposed a novel approach towards an efficient and scalable framework for verifying MESI and other MESI-based protocols in a multi-level cache hierarchy. Other protocols like MSI and SI can be tested using our framework though, for MSI or SI, we will have some redundant tests. We also incorporated our directed testing framework into the SST component to test cache coherence as well as memory hierarchy in general.

Even though for our simulation purpose and testing purposes we used a memory hierarchy with two levels of caching, our framework can be extended for other types of memory hierarchies with more than two levels of caching.

There are other cache coherence protocols like MOSI, MOESI, etc, but we were not able to extend to those because SST does not have support for those coherence protocols.

Chapter 5

Conclusions and Future Work

In this dissertation, we present a framework for cache coherence verification. In the first contributory chapter, we present our method for verifying cache coherence protocols in a single-level cache hierarchy. We propose a novel approach to reduce the number of test sequences for validating the implementation of the SI cache coherence protocol. Considering the fact that the SI protocol can be further extended to MSI or MESI, we believe our contribution can significantly reduce the transactions needed to verify these protocols.

In the second contributory chapter, we build on the method above to create a setup for verifying cache coherence protocols considering a multi-level cache hierarchy. We propose an approach and validate its applicability by integrating it with the cache coherence modules of SST. Our method is quite generic, in other words, it can be extended beyond two levels of the cache hierarchy. As an additional contribution, we present a discussion on how we implemented a framework to extend the memory-hierarchy-tester built by a team of IMEC, Belgium. We conclude with a case study on the effectiveness of directed testing by triggering a bug inside the SST *memHierarchy* module.

This work leaves open a number of interesting directions. Going ahead, we intend to take up the following activities.

1. In this work, we do not consider the effect of associativity and cache replacement policies. For our ease of understanding, we have taken the cache associativity as 1. This may not be always the case for modern set-associate caches. Going forward, we intend to incorporate associativity in our analysis.

2. We do not consider transient or intermediate states. As a result, the transient transitions and states are not covered by our directed testing. The global state space changes when we introduce these intermediate tests. A good area of research would be to find a way to cover these states and transitions too.

We believe that this thesis will open up new and useful avenues in cache coherence protocol verification going forward. The directed testing framework developed by us can be extended in multiple directions to advance the state of the art in the intersection of verification and computer architecture research.

Bibliography

[1] Clarke, E. M., Enders, R., Filkorn, T., and Jha, S. Exploiting symmetry in temporal logic model checking. *Form. Methods Syst. Des. 9*, 1–2 (aug 1996), 77–104.

[2] Gulzar, M. A., Malik, A., Jamil, F., and Mahmood, A. Random testing of cache coherence protocols. In *2018 International Conference on Frontiers of Information Technology (FIT)* (2018), IEEE, pp. 46–51.

[3] Leroy, X. Formal verification of a realistic compiler. *Commun. ACM 52*, 7 (jul 2009), 107–115.

[4] Lowe-Power, J. Adding cache to the configuration script, 2023-04-10.

[5] Lyu, Y., Qin, X., Chen, M., and Mishra, P. Directed test generation for validation of cache coherence protocols. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 38*, 1 (2019), 163–176.

[6] Mishra, P., and Dutt, N. Graph-based functional test program generation for pipelined processors. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition* (2004), vol. 1, pp. 182–187 Vol.1.

[7] Nagarajan, V., Sorin, D. J., Hill, M. D., Wood, D. A., and Jerger, N. E. *A Primer on Memory Consistency and Cache Coherence*, 2nd ed. Morgan amp; Claypool Publishers, 2020.

[8] Qin, X., and Mishra, P. Automated generation of directed tests for transition coverage in cache coherence protocols. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)* (2012), pp. 3–8.

[9] Sarangi, S. R. *Advanced Computer Architecture*, 1st edition ed. McGrawHill.

[10] Shirvani, P., and McCluskey, E. Padded cache: a new fault-tolerance technique for cache memories. pp. 440 – 445.

[11] Sweazey, P., and Smith, A. J. A class of compatible cache consistency protocols and their support by the ieee futurebus. *SIGARCH Comput. Archit. News 14*, 2 (may 1986), 414–423.

[12] Thomadakis, M. The architecture of the nehalem processor and nehalem-ep smp platforms. *JFE Technical Report* (03 2011).

[13] Wagner, S., and Wild, M. Partitioning the hypercube qn into n isomorphic edge-disjoint trees. *Discrete Mathematics 312* (01 2012), 1819–1823.

[14] Wang, X., Dong, W., Li, X., and Zhang, Y. Formal verification of a hardware implementation of the mesi cache coherence protocol. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)* (2016), IEEE, pp. 1388–1395.

[15] Wang, Y., Wang, J., Yang, G., and Wu, Q. Directed testing of cache coherence protocols for heterogeneous multi-core systems. *Journal of Parallel and Distributed Computing 74*, 9 (2014), 2963–2975.

[16] Yang, J., Yang, Z., and Xu, C. Directed testing for cache coherence protocols. *Journal of Software 6*, 4 (2011), 677–684.

Appendix A

Proofs for Chapter 4

Theorem A.1: The number of test sequences generated by the Algorithm-9 in Chapter-4 is $n2^{n+2} + n^2 2^{n-1} + n2^{n-1} + 2^{n-2} + 12n^2 - 3n + 1$.

Proof. To prove thi, we need to consider different parts of the proposed algorithm and analyze them separately.

1. The *createTestsSI* function uses the same algorithm as shown in Chapter-3. The number of test sequences issued by the algorithm is $n + n2^{n-1} + 2^{n-2} + 1$.

2. The *visitClique* function(refer algorithm-12) traverses all the transitions possible between invalid and modified states and also upon reaching the modified states, it also visits the bidirectionally reachable shared states. The find out the number of test cases generated by the *visitClique* function, we form a recurrence relation. Consider $VC(i)$ to be the number of test cases generated when given input i to the function. Note that another input n has no effect in forming the recurrence relation other than forming the stopping condition. Also it is trivial to understand that $i \in [0, n-1]$ and $VC(n) = 0$. Now in every function call we have 3 transactions issued as shown in line-1 to line-3 of Algorithm-12. Next in the for loop as shown from line-4 to line9 we issue $2 \times (n-1)$ test cases irrespective of the value of i. Next depending on the value of i, $2 \times (n - 1 - i + 1 + 1) = 2n - 2i + 2$ test cases are generated as shown in line-10 to line-16. Lastly, we have $VC(i + 1)$ number of test cases generated. This is required as there is a recursive function call at line-13 which is called when i becomes i+1 in the for loop of line-10. The final recurrence relation would be

$$VC(i) = 3 + \sum_{j=0 \ \& \ j \neq i}^{n-1} [2] + \sum_{j=i+1}^{n-1} [2] + VC(i+1) = 4n - 1 + 2i + VC(i+1)$$

   We need to solve the recurrence relation with $VC(0)$ known, which is consistent with our main function call in Algorithm-9. Solving the recurrence relation for $VC(0)$, we get,

$$VC(0) = 3n^2$$

3. The $visitE()$ function has been completely adopted from [5]. Observing closely we can derive that the number of test sequences generated by that function is $4n^2 - 2n$.

4. The $visitOtherMESI$ traverses from all the shared states to all the modified states, the point to remember here is that some of the transitions are unidirectional.

The basic idea of the algorithm is, after reaching a shared state, we issue the corresponding write instructions for all cores. Now after reaching a modified state. we should come back to the original shared state to cover another *shared* to *modified* state transition. The process shown in Algorithm-14 is a bit different but the idea is the same.

To find out the number of test sequences generated, we first analyze the algorithm. It takes a string input *states* and a variable *index* which is initialized to 0. From line-29 to line-32 we can see that all possible combinations of the *states* string (where every character in the string can take a value I or S) are generated. The $Index$ variable helps to keep track of the length upon which the modification of string is done. If the string *states* contains all-I we do not do any operation and return. If the string *states* contains a single S, we issue a total of 7 operations. If the *states* string contain more than one S, we generate $4 + (number\ of\ \mathbf{S}\ in\ states)$ for every $i \in [0, n-1]$. The length of *states* is n.

Case 1: If *states* contains no S, we do not issue any instruction.

Case 2: If *states* has a single S, we issue 7 instructions for every $i \in [0, n-1]$. The total number of possible *states* where there is only a single S is n. In those cases, the total number of test sequences is $7n^2$.

Case 3: If *states* has more than one S, we issue $4 + (number\ of\ \mathbf{S}\ in\ states)$. Consider a *state* where the number of S is j. Total number of such *states* is ${}^nC_j$. For each *state* string we issue $n \times (4 + j)$ number of instructions. The total number of such instructions for such a string is ${}^nC_j(n \times (4+j)$. For this case, the total number of instructions should be $\sum_{j=2}^{j=n} {}^nC_j(n \times (4+j)$.

The total number of transactions issued will be $7n^2 + \sum_{j=2}^{j=n} {}^nC_j(n \times (4+j) = n2^{n+2} + n^2 2^{n-1} + 2n^2 - 4n$.

5. For the last function *visitOtherE*, the total number of transactions issued will be $2n + 3n^2$.

The total number of transactions issued by Algorithm-9 is

$$= n + n2^{n-1} + 2^{n-2} + 1 + 3n^2 + 4n^2 - 2n + n2^{n+2} + n^2 2^{n-1} + 2n^2 - 4n + 2n + 3n^2$$

$$= 2^n[\frac{1}{4}n^2 + \frac{9}{2}n + \frac{1}{4}] + 12n^2 - 3n + 1$$

□

Theorem A.2: The time and space complexity of Algorithm-9 are $\mathcal{O}(n^2 2^n)$ and $\mathcal{O}(n)$ respectively.

Proof. As shown in Chapter-3, the time and space complexity of the *createTestsSI* function are $\mathcal{O}(n2^n)$ and $\mathcal{O}(n)$ respectively. The time complexity of the function *visitHypercube* is $\mathcal{O}(n^2)$. The space complexity is $\mathcal{O}(n)$ because of the stack space required for the recursive function call. The *visitE* function has time complexity of $\mathcal{O}(n^2)$ and requires $\mathcal{O}(1)$ space to execute. $visitOtherMESI$ creates all possible combinations of the string that contains the S and I characters. The number of such combinations is 2^n. Now for each combination, the function does at most n^2 amount of work. So the overall time complexity of the function is $\mathcal{O}(n^2 2^n)$. The space complexity is $\mathcal{O}(n)$ because of the buffer space. The time and space complexity of $visitOtherE$ is $\mathcal{O}(n^2)$ and $\mathcal{O}(1)$. So the overall time complexity of Algorithm-9 is $\mathcal{O}(n2^n) + \mathcal{O}(n^2) + \mathcal{O}(n^2) + \mathcal{O}(n^2 2^n) + \mathcal{O}(n^2))$ which is $\mathcal{O}(n^2 2^n)$ (dominating term) and the space complexity is $\mathcal{O}(n)$. □