INTERPOSER BASED ROOT-OF-TRUST

A Thesis

by

TAPOJYOTI MANDAL

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Paul V. Gratz
Committee Member,	Daniel A. Jiménez
Committee Member,	Jeyavijayan Rajendran
Head of Department,	Miroslav M. Begovic

May 2021

Major Subject: Computer Engineering

ABSTRACT

Modern computing systems try to extract every bit of performance available through various techniques such as hardware speculation and memory de-duplication. Recent Spectre and Meltdown attacks have revealed different types of security vulnerabilities that can arise due to speculative hardware in modern processors. Hardware speculation and complex software applications running on these hardware make it extremely difficult to verify every possible functional aspects of a system, which leads to security vulnerabilities within the system. Attackers tend to take advantage of these unforeseen vulnerabilities through means of both software and hardware attacks.

In this work, we focused on hardware attacks specifically through hardware trojans[1]. Hardware Trojans can be defined as malicious modifications that are made in the design of a processor or any other part of the integrated circuit(IC). These trojans can be inserted into the system during design or fabrication phase. Economic factors have led to the semiconductor industry adopting the approach of chiplet based design, where the functional design logic is distributed across multiple chiplets which are fabricated with high yields and then are connected onto a die using interposers. These chiplets are fabricated in fabrication facilities based in different countries to further streamline the fabrication process. Such an approach reduces the cost of manufacturing while also allowing for the possible introduction of hardware trojans in the system during fabrication stages[2] in untrusted fabs.

The study tackles the issue of security vulnerabilities brought in through these hardware trojans in multi-core multi-chip systems with the use of active interposer technology. Interposers[3] are used for interconnecting multiple chiplets onto a single die. Recent research show that interposers can be active[4][5]. This means that design logic can be embedded into these interposers which can then serve the role of monitoring transactions across chiplets. Interposers, which have a lower technological constraint for fabrication, can be manufactured in trusted fabs and serve as the "root-of-trust" for a system.

In our proposal we take a look into design aspects of a Security Network Interface(SNI)[6][7]

which is a module that is embedded into active interposer layer and monitors cache coherence messages between chiplets in a multi-chip system. We analyze various aspects of network and cache coherence protocol which impact the design of a SNI. We also provide the impact of SNI on performance of a system using SPEC CPU benchmarks.

We intend to contribute to the idea of active interposer as the "root of trust". Past research have focused on VLSI aspects of interposer[6], but in this research we intend to focus on the architectural challenges. These include understanding the network-on-chip and cache coherence system, since the SNI implemented in interposer layer will monitor network packets which carry cache coherence messages. Thus, this research will highlight the architectural factors which ought to be considered for implementing an interposer as "root-of-trust" and the performance impact of such SNI in a network.

DEDICATION

To my family and my friends.

ACKNOWLEDGEMENTS

I would like to extend my most profound gratefulness to my advisor - Dr. Paul V. Gratz. It was a privilege to work with Dr.Gratz who has been a great advisor and mentor through out the period of my research work. Dr. Gratz's patience, guidance, and support was unwavering and kept me going throughout my master's journey. I value the weekly technical discussions with my advisor, which helped me define the direction of my research work. There were numerous steps where I received valuable comments from Dr.Gratz and my other research members, without which I would be making mistakes in my observation of results. I would also like to thank Dr. Daniel A. Jiménez and Dr. Jeyavijayan Rajendran for agreeing to be on my committee and providing me valuable feedback on my work and its documentation.

A special thanks to all the members of the CAMSIN research group; they all were terrific colleagues to work with, and I always enjoyed the technical discussions that happened.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis (or) dissertation committee consisting of Prof. Paul V. Gratz (ECEN Department) as the advisor, Prof. Daniel A. Jiménez (CSCE Department) and Prof. jeyavijayan Rajendran (ECEN Department) as the committe members.

All the work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was partly supported by merit based scholarship from the ECEN Department at Texas A&M University.

NOMENCLATURE

NOC	Network On Chip
VN	Virtual Network
VC	Virtual Channel
IOT	Internet of Things
DRAM	Dynamic Random Access Memory
CAMSIN	Computer Architecture, Memory Systems and Interconnection Networks
IP	Intellectual Property
PCB	Printed Circuit Board
CPU	Central Processing Unit
OS	Operating System
MB	Mega-Bytes
KB	Kilo-Bytes
CPU	Central Processing Unit
IPC	Instructions Per Clock Cycle
NI	Network Interface
MOESI	Modified-Owned-Exclusive-Shared-Invalid Protocol

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION AND MOTIVATION

Security failure incidents have raised concerns regarding the architecture of modern processors where the focus has always been on performance. The increased hardware complexity along with complex software applications being run on these processors have led to vulnerabilities in applications and hardware which are difficult to verify. Hence, considerable research is being done to improve the security of modern architectures through either hardware or software based security features being added to the design. There are various ways in which security vulnerabilities may present itself, but these can be broadly categorized into two prominent sections:

1. **Software attacks**: Software attacks leverage the complexity of code and the underlying hardware to find loop-holes within the system. The attackers target a specific aspect of a software application and then try to deviate from the intended use of application to breach security. Buffer overflow attacks [8] are one of the most popular software attacks that are used by attackers. Use of function calls in unintended use can allow these buffer overflow attacks to extract critical information or cause a system breakdown.

2. **Hardware Attacks**: In modern computing systems multiple applications run on the same hardware. The attacker and victim applications can share the same hardware resources which allows the attacker to obtain sensitive information from victim applications. Hardware attacks can occur through untrusted components such as peripheral IPs or devices on a system's PCB. Hardware attacks can also occur in the form of physical attacks such as in the form of probing of memory bus.

One of the ways in which hardware attacks can occur is through the introduction of hardware trojans[9][2] during the fabrication stages. Modern processors are moving towards multi-core multi-chip systems[10] where the cores are distributed across chiplets. These chiplets are fabricated in overseas fabs and then connected together into a chip using interconnects such as interposers. The availability of advanced fabrication devices with overseas fabs has resulted in the tasks

of fabrication stage being off-loaded to the fabs in different parts of the world. Although this also raises a concern of hardware trojans being inserted into the chips in the fabrication stages.

In this work, we show that active interposers can act as the "root-of-trust" in modern computing systems and provide security against such physical attacks where hardware trojans have been introduced into chiplets. Previous research[4] demonstrates that interposers can be manufactured such that they contain active design components. These interposers can be manufactured in trusted fabs and at lower process technology nodes, thus alleviating the issue of introduction of vulnerabilities in the hardware security modules itself. The objective has been to analyze the design of a security module that would be emplaced in the interposer layer. We call it Security Network Interface(SNI) since it serves as a security interface which sits on the Network-on-Chip(NOC) interconnecting the chiplets. The SNI monitors the packets being exchanged between the cores and memory. In case of an unintended type of transaction the SNI raises a machine check error[11] to warn about a possible security attack within the system.

We implemented a multi-core multi-chip system in an architectural simulator and configured the system to emulate the behavior of an interposer. We analysed various design aspects of SNI through simulation, thus providing insight into the critical components that need to be taken care of in the design of an SNI. We have also analysed the performance impact of the SNI on the system by characterizing it against the SPEC2006 CPU benchmarks. We provide details on the modifications that would be needed in the OS in order to handle the dynamic nature of application flow in a computing system in order to handle the SNI functionality.

To summarize, we make the following contribution through this thesis work :

1. We propose a new, novel interposer-based SoC system architecture.
2. We empirically characterize the performance impact of the hardware based SNI modules introduced in the interposer layer
3. We present an analysis of the design of the SNI from an architectural perspective
4. We analyse the modifications that an OS would require in order to facilitate SNI in its functionality

1.1 Thesis Statement

"Active Interposer as "root-of-trust" for chiplet based computing systems"

Interposer are used to interconnect multiple chiplets onto a die. Interposer technology embedded with active design logic are considered as active interposers. These active interposers can be fabricated in trusted fabs thus acting as "root-of-trust". This "root-of-trust" acts as the root of all security evaluation for the system thus helping prevent hardware based attacks in a multi-chip system.

1.2 Document Organization

In this thesis document, we start with the background and related work in Chapter 2. We look at the types of security vulnerabilities and how hardware trojans can leave a system insecure. Then we discuss some of the previous work on active interposers and SNI and how our approach differs from those. In Chapter 3, we provide details of the architecture of SNI and the modifications required in OS in order for the SNI to work. In Chapter 4, we provide our observations on the performance impact of SNI on a system. In Chapter 5, we conclude and discuss some key points for future work, which is not covered in this work.

2. BACKGROUND AND RELATED WORK

2.1 Security Vulnerabilities

A computing system contains the hardware and the software running on it. Both of these components can be vulnerable to attacks. But the type of attacks that can be carried out on each of these components can vary.

Attacks can be carried out through software or hardware or a combination of both. For example, a malicious software trying to access sensitive information through side-channel attacks will fall under the category of software-on-hardware attacks as it exploits the processor's cache vulnerability. An example of hardware-on-software attack can be a malicious memory controller trying to access the main memory in an unintended way from user's perspective and gain sensitive data stored on main memory which belongs to a different application. A hardware-on-hardware attack can be a presence of any peripheral IC that may try to disable encryption modules or spam the memory bus.

In this work, we focus specifically on hardware attacks. As described, hardware attacks are usually carried out by the introduction of malicious hardware onto the system. Even if the design engineers add security features into the system, there is always a possibility of adding malicious hardware in late stages of fabrication to subvert the security features and testing methodologies. One of the key concerns is the introduction of hardware trojans. In the next section we describe what is meant by the term hardware trojan and what are its implications on the security of a device.

2.2 Hardware Trojans

As the complexity of processors and other peripherals have increased, an IP based methodology has been adopted by the industry, in order to manage the cost and time-period of taping out these chips. The functionalities of processor and peripherals are broken down and then functionalities are designed into smaller blocks called IPs which can be designed by a company itself or obtained from other vendors. Also the manufacturing of the ICs are done in facilities which often are not

owned by the design company itself. All the different parties involved in supplying the IP or during the manufacturing stages can potentially modify the design by adding malicious hardware trojans into the IC. Hence, even if a processor or system is designed to work perfectly with certain security measures in place, research[12] has shown that addition of hardware trojans can render a secure processor vulnerable.

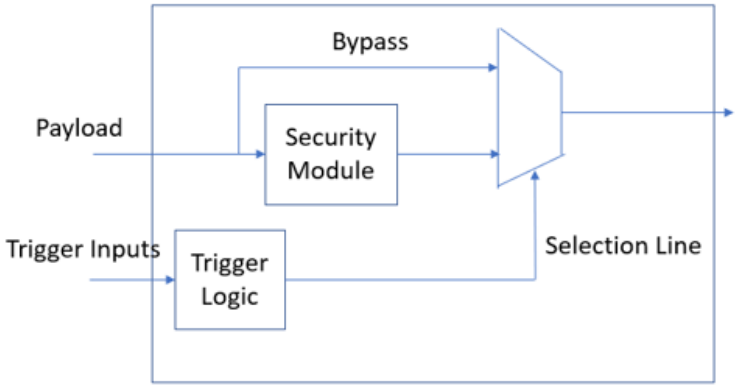

Figure 2.1: Example Hardware Trojan

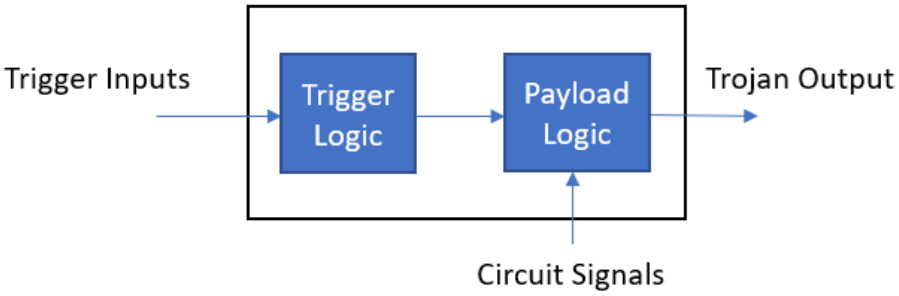

Figure 2.2: Example Hardware Trojan

Figure 2.1 shows a simplistic model of a hardware trojan. We can see that we have a security

module which gets certain inputs that it processes. It might be a hardware module responsible for encryption of data. But as we can observe, there is a mux path which has been implemented such that when the "sel" selection signal is triggered, the security module is bypassed, which leaves the data unencrypted. The change of the selection signal can be triggered by the attacker through a software application which triggers the hardware controls.

Figure 2.2 shows another example of a hardware trojan where the trojan is emplaced on the output ports of the IP. The trigger circuit signals is the intended output, but in this diagram we can observe that the circuit signals are fabricated through a hardware trojan such that when the trojan is triggered through certain trigger inputs it alters the output of the IP.

There have been research on categorization of hardware trojans[13]. Hardware trojans can be inserted into the system during various stages of system design and manufacture. Design engineers during the design phase implement functionality through RTL design and then design is mapped onto the desired process technology. During this stage 3rd party IPs may be used in order to accelerate the time to tape-out by integrating the IPs as black boxes into the design with certain amount of configuration. These 3rd party IPs may contain trojans and it may be tough to verify as well because usually a lot of the IPs are provided as a black box with only specification for the interface and functionality but not the detailed design.

One of the issues with hardware trojans is that it is not trivial to detect hardware trojans[14]. Functional verification performed on designs aims to find bugs in the design functionality but not reveal security vulnerabilities. Also a lot of the security features are built into the design. But that leaves them vulnerable to modification during fabrication stages. Masks are used to fabricate the chips. These masks can be modified by the adversaries to introduce trojans. Hardware trojans can be embedded as RTL into 3rd party IPs which can then be activated when it has been deployed[15].

These cannot be detected using post tape-out testing where the focus is on functional correctness and performance measurements. Modern processors contain design-for-test(DFT) capabilities which allow detection of manufacturing defects in the design. But most of the DFT designs don't take into account that security vulnerabilities need different set of measures in order to be

detected[16]. The objective of DFT is to observe internal signals which are part of the specification, not hardware trojans which were introduced by attackers.

In our work, the focus is on chiplet based systems, where the assumptions is that the in a multi-core multi-chip system, the chiplets that are being fabricated are vulnerable to introduction of hardware trojans. And hence, our objective is to add security measures in the hardware and software to tackle such trojans. Hence, in the next section we define the threat model for our system.

2.3 Interposer as hardware root-of-trust

The idea of root-of-trust is that a part of the system is trusted because of its design and fabrication being under control of the designer. Any other components which have been obtained as 3rd party IPs or have been exposed to the supply chain can be considered as untrusted. In modern systems, we have chiplets and IPs which are being fabricated in overseas fabs[17][18], which adds the possibility of trojans being added to these chiplets and IPs. Over the years, the use of interposers to interconnect chiplet based systems has been in focus.

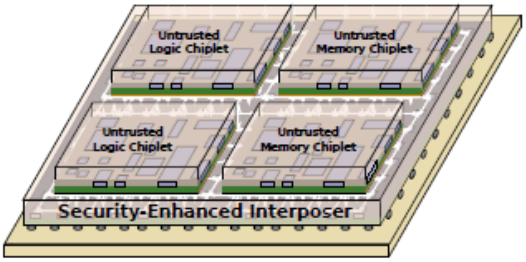

Figure 2.3: A chiplet system where the interposer connects multiple chiplets on the die[6].

In 2D integrated circuits, dies are mounted in a package in a single plane. They are connected using flip-chip or wire-bump technology. One of the drawbacks in such cases is that, usually there is a mismatch in the size of the tracks in the substrate and that of the die, leading to performance concerns and unnecessary power consumption. 2.5D integration instead uses silicon interposer

between the substrate and the die. And the silicon interposer has through silicon vias(TSVs) which are used to connect the metal layers on both its top and bottom surface. But the primary reason that 2.5D IC are being preferred is because of the improved yields by fabricating smaller chiplets and integrating them using 2.5D integration techniques into a single die. Similar approach is used in 3D ICs as well, where the interposer plays a key role in the integration of dies.

While much work to date has assumed passive interposer layers, recent work has shown that interposers can be active[4] [5], and design logic can be implemented in these layers. Active interposers, as the name suggests, can have design logic implemented in them. Passive interposers facilitate communication between chip-to-chip but active interposers provide the benefit of having a NOC within the interposer. A NOC hierarchy can be formed, where at the lowest level we have NOC within the chips, and then we have NOC at the interposer layer connecting the chiplets with each other. Power management features can also be implemented within active interposers allowing robust power delivery to the compute chiplets. Active interposer can support features such as clocking, reset and probes[4]. This provides the motivation that as active interposers get more features added it will be able to support more design logic. And thus we can also have hardware based security checking within the interposer layer. Active interposers can be fabricated using mature technology such as 65nm, thus allowing its fabrication to be carried out by a trusted fab which does not require advanced technologies unlike the advanced process nodes. Memory is one of the primary concerns in security vulnerability scenarios. Active interposers can also integrate memory[4] using system level IOs which allows the memory to be protected against malicious chiplets.

Thus, we have established that active interposers can be relied upon as a hardware root-of-trust because of the control over their fabrication process. This allows addition of security features to the interposer layer without addition of any malicious hardware in the supply chain. There do exist hardware security features such as Intel SGX and AMD TrustZone [19] but they are implemented in the chiplet themselves, again raising the concerns about the situation that an attacker having a full access to a system can manipulate such security features.

3. ARCHITECTURE AND IMPLEMENTATION

In this section we discuss the system architecture and the design implementation. One of the objectives of the work is to analyze the design details of SNI. And discuss in detail how the system architecture, the cache coherence protocol and memory system impacts SNI design. The idea is to model the interposer in an architectural simulator such as gem5[20] and use the complex cache coherence and NOC implementations of gem5 to improve the design of SNI. gem5 has an elaborate set of systems already implemented in place, which can be leveraged to model the interposer layer architecturally and then verify that functionality of SNI is as intended in order to tackle our threat model. Hence, before we delve into the system architecture, it is important to discuss the threat model for our work.

3.1 Threat Model

A threat model is a specification of the threats that a given processor and it's secure architecture protects against. It is difficult to come up with an architecture which tackles all the facets of security threats. Hence, it is important to define a clear and concise view of the threats that our work focuses and tackles. For example, Intel SGX architecture is an implementation for secure architecture but it does not prevent side-channel attacks[21]. Similarly, our work does is not designed to address side channel attacks. Our the threat model is similar to Nabeel et al.[6]. The focus in this work is on a system wherein multiple chiplets have been fabricated in various fabs and then they have been connected together using 2.5D or 3D interposer technologies. The assumption is that the fabrication of the chiplets, either designed by the designer themselves or obtained as 3rd party IPs, cannot be trusted since hardware trojans can be introduced during the fabrication stages.

The introduction of hardware trojans can create unique scenarios in malicious chiplets but they can be broadly categorized into four categories[22]. The threat model attempts to tackle the following types of threats:

- Passive reading or *snooping*, where a core within a chiplet is able to read or gather data which

is meant for other chiplets. Malicious chiplet may observe the cache accesses of a trusted chiplet through hardware trojans which monitor the interface of the cache controller. This will allow the trojan to record the cache blocks which are being accessed by other chiplets based on the state a cache block is in.

- Masquerading, also known as *spoofing* wherein a core within a chiplet maliciously disguises itself as another chiplet in order to gain access to critical data or control. A trojan can modify the requestor ID and embed cache coherence messages which then can trick the directories or other unsuspecting cores to pass critical data to those malicious chiplets.

- Modifying data that the untrusted chiplet does not have write permission for by modifying the outgoing cache coherent messages. Hardware trojans in conjunction with an attacking application can disguise it's access permissions for a memory region. For example, it can try to disguise itself as having write access to a region of memory which it had a read-only access.

- Diverting i.e. a malicious chiplet diverts the data meant for one chiplet to another untrusted chiplet. In shared memory applications, certain chiplets may divert the data from one chiplet to another chiplet through multi-threaded malicious applications.

It is assumed that the attacks are exercised such that a malicious chiplet containing hardware trojan would try to communicate with an unsuspecting chiplet. But the communication between the chiplets and memory would have to pass through the interposer layer. Attacks which occur within a chiplet are considered out of scope for this work. Hence, attacks such as side-channel attacks across cores within the same chip [23] and covert channel across cores [24] are considered out of scope. The focus is on detection of trojans within a chiplet trying to act maliciously by detecting the behaviour at the interface level, which in our case is the interposer. We also focus on attacks from a control perspective. Any kind of attack which tries to modify the data shared between cores across chiplets will not be detected.

In the following section we explain the importance of an interposer as "root-of-trust" which will enable us to implement the security features within the interposer layer in order to handle the discussed threat model.

3.2 Architecture

There are two important regions of the system. One is the chiplet architecture and then the system architecture where all the chiplets are connected to each other through interposer layer. The following sections describe the details of both of these architectures.

3.2.1 System Architecture

The system consists of 8 chiplets and 4 memory controllers. The main memory is distributed across the 4 memory controllers for improved memory bandwidth.

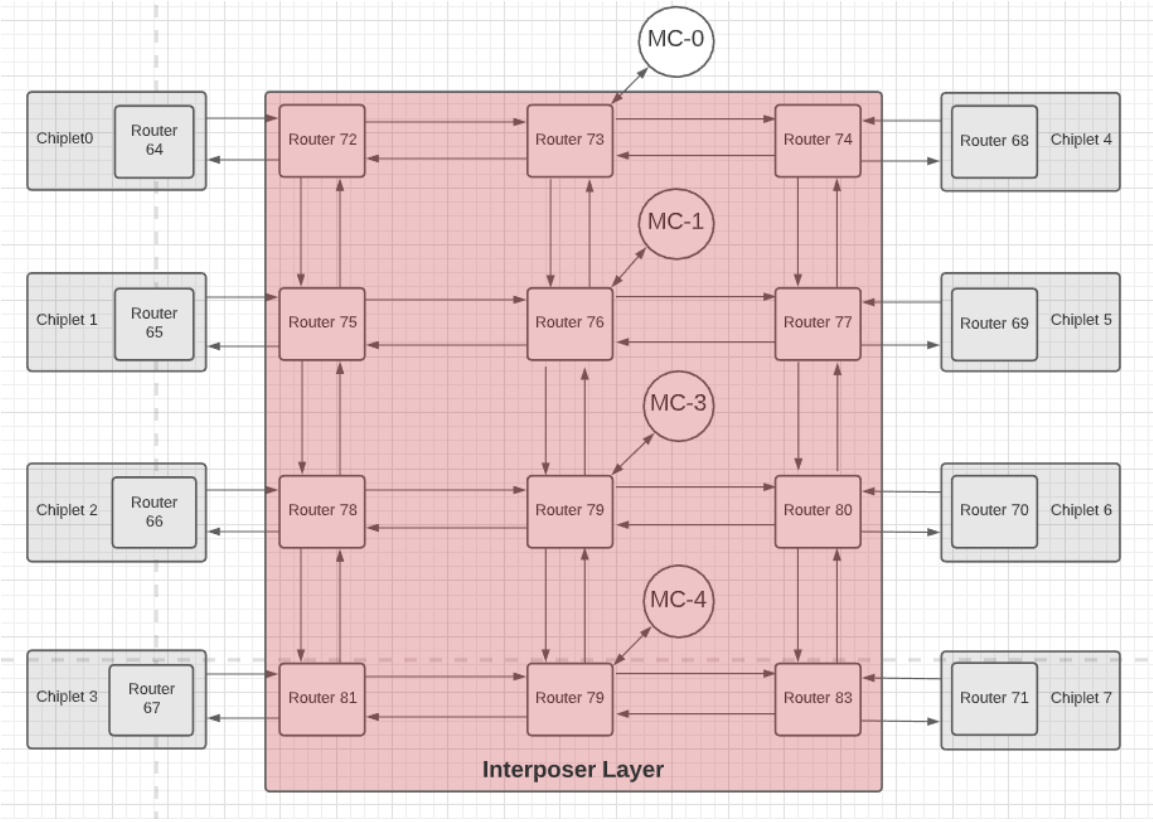

Figure 3.1: System Architecture

Figure 3.1 shows the system architecture used for evaluation in our work. Although this is one of the many different ways in which a system can be configured, the theory of it and how it impacts the SNI placement in the interposer would remain same for any other configuration as well. In this system we have 8 chiplets each containing 8 CPU cores and the chiplets are connected to each other through an *Interposer Layer*. The interposer layer, laid out in the form of a Mesh network interconnect also connects the *Memory Controllers* to the chiplets. The *Routers* depicted inside each chiplet serve as the point of connection between the chiplet and the interposer layer. The main memory has been distributed into 4 sections each of which is controlled by a memory controller numbered 0 to 3. These memory controllers are also connected to the interposer layer.

As we can observe, a key learning is that all the network packets will have to traverse through the interposer layer. If a CPU core in a chiplet wants to read/write data from/to memory, the corresponding cache coherence message, embedded inside flits, will have to traverse the interposer. Similarly, if a core within a chiplet wants to communicate with another core in another chiplet, the flits again will have to traverse the interposer. This leads to a very simple decision of placing the SNI within the interposer layer. The interposer acts as the "root-of-trust" since all message transactions inevitably traverse through interposer thus providing us the confidence, that if needed all the message transactions can be monitored by a SNI emplaced within the interposer layer.

An interesting aspect is the hierarchy of network within the system. Each chiplet has 8 cores which are connected to each other using a NOC, and then we have 8 chiplets which are interconnected by a interposer layer NOC. Thus creating a hierarchy of NOCs in this multi-core multi-chip architecture.

3.2.2 Chiplet Architecture

The interposer acts as the root-of-trust and thus monitors every cache coherence message that crosses the boundary between the chiplets and the interposer. Figure 2 shows the top-level architecture of a chiplet containing multiple cores and respective private cache memories. For the purpose of this study, we examine chiplets containing multiple processor cores, similar to AMD's present day processors[25]. That said, our proposed architecture places no restrictions on the chiplets'

contents, be they cores, accelerators or GPUs as long as they are cache coherent and obey shared memory.

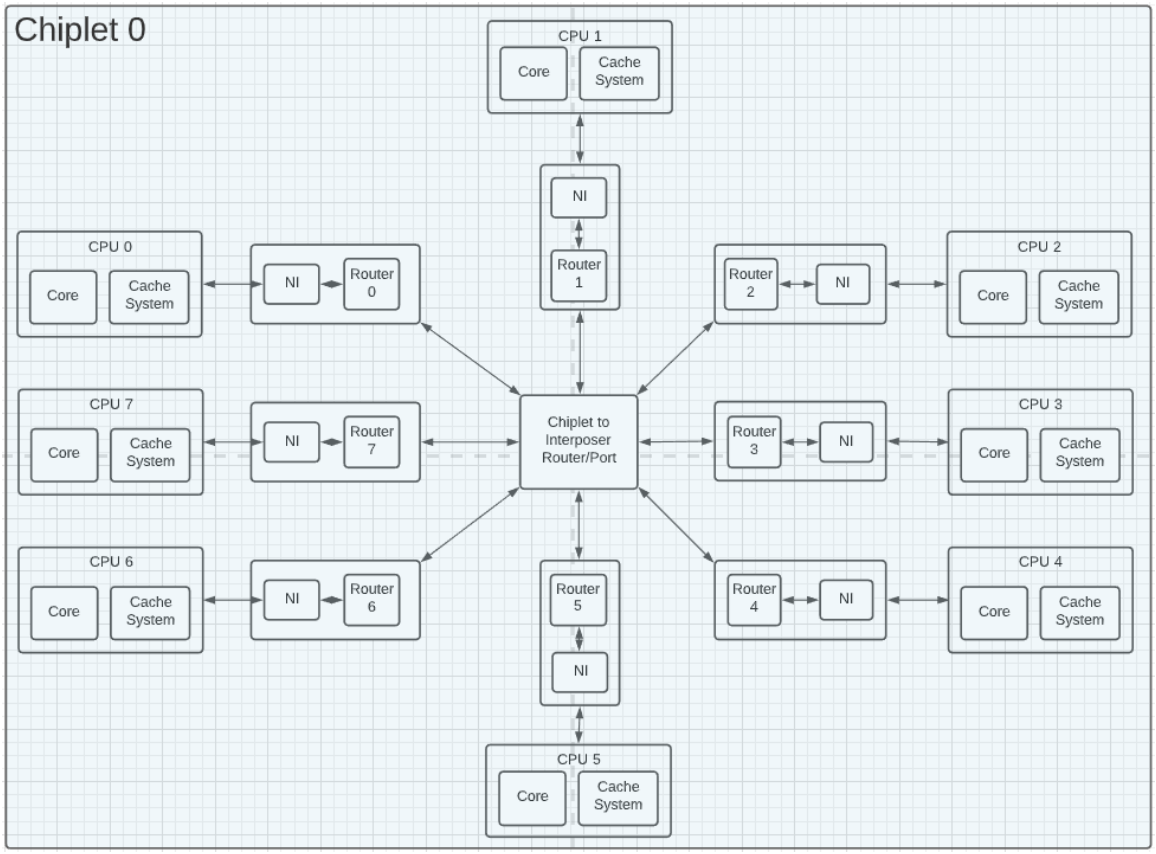

Figure 3.2: Chiplet Architecture

In Figure 3.2, the architecture of a chiplet has been shown. Each chiplet contains 8 CPU cores and each of them contain their own private cache. All of the cores are connected to each other in a star topology through *Network Interface* and *Routers*. The NOC(NOC) connections here depict the way cores are connected to each other in a chiplet. Star topology was chosen to keep the internal chiplet architecture simple. Other topologies such as Mesh, Butterfly[26] can also be chosen. The central router described as "Chiplet to Interposer Router" is the Router which connects the Chiplet to the Interposer layer.

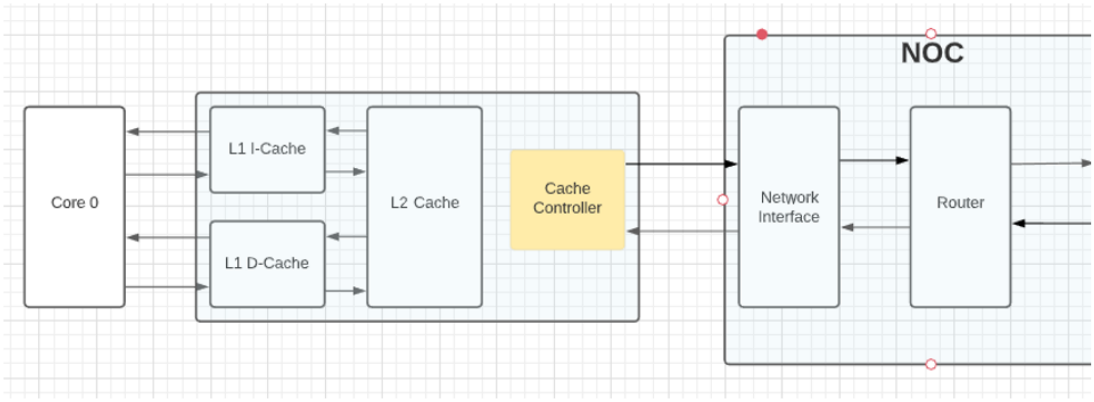

Figure 3.3: Core Architecture

Figure 3.3 depicts the architecture of a CPU core and the manner in which it is connected to the NOC. Each core has a private L1 Instruction cache and L1 Data Cache. It also has a unified L2 cache which is also private to the core and is larger in size compared to L1 caches. These L1 and L2 level caches provide each core with sufficient local memory space needed for good performance. Each of the cache is controlled by a cache controller. The cache controller is responsible for handling the states of each cache block. The CPU core will perform read and write operations, through read and write instructions pertaining to the ISA, which in our case is an X86 ISA. These instructions are then converted into cache coherence messages by the cache controller in order for data to be exchanged between the private cache and the main memory.

The cache coherence messages that are created by the cache controller are then converted into network packets by the *Network Interface*(NI) which is part of the NOC interconnect. At the node of each CPU core a NI is needed, which converts the cache coherent messages into flits as per the NOC protocol. These network packets can then traverse the NOC interconnect and reach their intended destination. In later section we describe the NOC architecture and its components in detail.

3.3 NOC Interconnect

In this section we describe some of the key aspects of the interposer layer implemented as a NOC. A NOC stands for NOC and is used to connect multiple nodes with each other. There are multiple aspects of a NOC which have been described in Natalie et al. [27]. It would be impractical to go in-depth into the various features and components of a NOC. Hence, certain critical aspects of NOC have been discussed which impact the SNI architecture.

There are some important aspects of a NOC which do not really impact the SNI architecture. The network topology is one of them. SNI is an active logic whose primary role is to scan the messages traversing the interposer layer. Hence, the most common place that it should be placed is within the *Routers*. Since, a SNI is emplaced within a router, the topology adopted in the interposer layer to connect the chiplets and other components do not impact the SNI architecture. Another component is the routing algorithm adopted. The routing algorithm of a NOC decides the manner in which network packets traverse through a network. Based on the routing algorithm, a pre-designated path might be followed by a packet. Although this is also irrelevant to the SNI architecture. Routing algorithm decides the direction of traversal of the messages which is of no concern for SNI.

3.3.1 Physical Link width

In our simulation we examine 2 link widths. For physical links within the chiplet the link width is 128-bit whereas for the links within the interposer layer it is 64-bit. Interposer layer links tend to be more costly hence we have chosen half the width of that within the chiplets. This provides us with a better model for analysing the latency introduced by the SNI.

Based on this configuration we have determined that the request type packets fit within 1 Flit within the chiplets. Each flit is of size 128 bits. When the flits enter the interposer layer it is broken down into two flits which is analysed in the SNI over two clock cycles. We dedicate the first clock cycle of logic design within the SNI to extract the requestor ID and cache message type information from the head flit. And then the second clock cycle we extract the address for the cache

block which is being accessed through the cache coherence packet. For any other type of design with different physical link width and different packet size we may need to change the SNI design logic to retrieve the data from flits over multiple pipeline stages and then use that accumulated data to perform security checks. Hence, the physical link width and the flit sizes determine the latency that is observed at the SNIs.

3.3.2 Message Types & Flits

Flits stand for flow control units. A cache coherence message is broken down into packets and then the packet is further broken down into flits. In our work we have opted for MOESI Hammer protocol which we have provided detailed explanation of in later sections. But each protocol has request and response type messages. Here we provide the details of the structure of the message types.

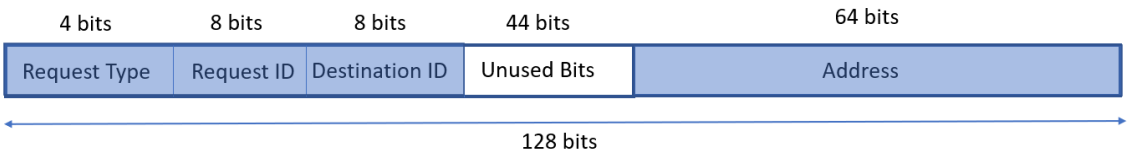

Figure 3.4: Request Message

Figure 3.4 depicts the structure of a request type message in MOESI hammer protocol. Each field within the message contains a specific information required for cache coherence transactions.

- **Request Type**: This specifies the type of cache coherence request. We will describe in detail the various request types in MOESI hammer protocol in later sections. In total we have ten types of request. Hence, 4-bits are allocated for this field.
- **Requestor**: This specifies the requestor identity. As mentioned in the system architecture we have 64 cores and 4 memory controllers. This provides a total of 68 nodes. Hence, 8-bits is allocated which is more than the require 7-bits.

- **Destination**: This specifies the destination. Each request message will have a designated destination. This can be any of the 68 possible nodes. Hence, 8-bits allocated for this field.
- **Address**: This is the 64-bit address for the cache line that is under consideration.

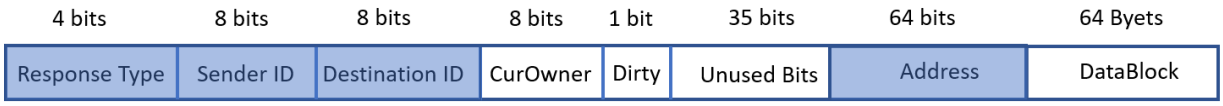

Figure 3.5: Response Message

Figure 3.5 depicts the structure of a response type message in AMD MOESI hammer protocol. Each field within the message contains a specific information required for cache coherence transactions.

- **Response Type**: This specifies the type of cache coherence response. We will describe in detail the various response types in MOESI hammer protocol in later sections. In total we have twelve types of response. Hence, 4-bits are allocated for this field.
- **Sender**: This specifies the sender identity. Same bit allocation as request message.
- **Destination**: This specifies the destination identity. Same bit allocation as request message.
- **CurOwner**: This field specifies the current owner of the cache block. This is required for handling *unblockS* types of responses.
- **Dirty**: This bit specifies if the cache block is dirty or not. Dirty here means that the cache block has been modified.
- **Address**: This is the 64-bit address for the cache line that is under consideration.

- **DataBlock**: This field contains the data. If the response type is of *Control* type then it doesn't have a corresponding DataBlock. If the response type is of *Data* then it will contain a 64 byte data section.

Based on theoretical analysis following parameters related to coherence message structure and flit structure need to be analysed by a SNI in order to provide security:

1. **Coherence Message Type**: The message contains information whether it contains a request or response type coherence request. A request type message means that the core is requesting a cache block which indicates a read/write request for the cache block was issued. Response type messages are the ones which are generated by cores or directory in response to a request message received. Response messages can be for example data response message from memory controller to the requesting core or another core responding to a request by the original requesting core. Every response message originates because of a corresponding request message. That does not mean that response messages should not be monitored. We will show in later sections that because of the complexity of the cache coherence protocol, malicious response messages fabricated by a hardware trojan within a chiplet can lead to a security breach.

2. **Flit type**: In many NOCs a credit-based data flow might be followed[28]. In credit based handshake, once a flit is sent from source to destination router, the source router decrements the credit in its logic, this prevents overflow of packets at the receiving router. When the receiving router has processed the flit it returns credit back to the source router and the source router increments the credit. Now, a credit is also constructed as a flit and exchanged between the routers across the network. Hence, it is important for the SNI to differentiate between a credit type flit and a coherence message type flit. We only want to analyse the coherence message type packets as they contain information regarding data access requests. If there is any violation by any of the cores within a chiplet based on the access permissions determined by the SNI then a security breach fault would be raised in the form of machine

check error.

Another detail to understand is that the flits are basically a packet broken down into chunks based on the size of a packet and bit-width of the physical links. The Control messages are converted into Head and Tail Flits when they traverse from Chiplet into the interposer layer.

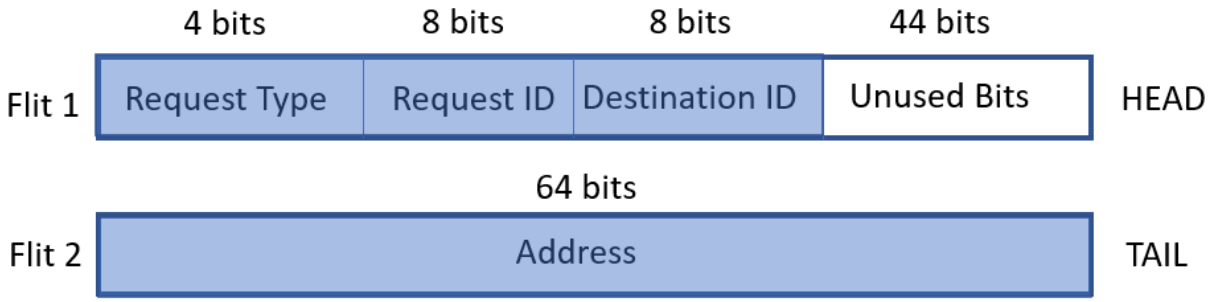

Figure 3.6: Control 128-bit flit within Chiplet

Figure 3.6 depicts the flit structure of a request message when in enters the interposer. A request message fits into a 128-bit physical width, but since we have 64-bit physical links within the interposer layer, the flit is broken into *HEAD* and *TAIL* flits.

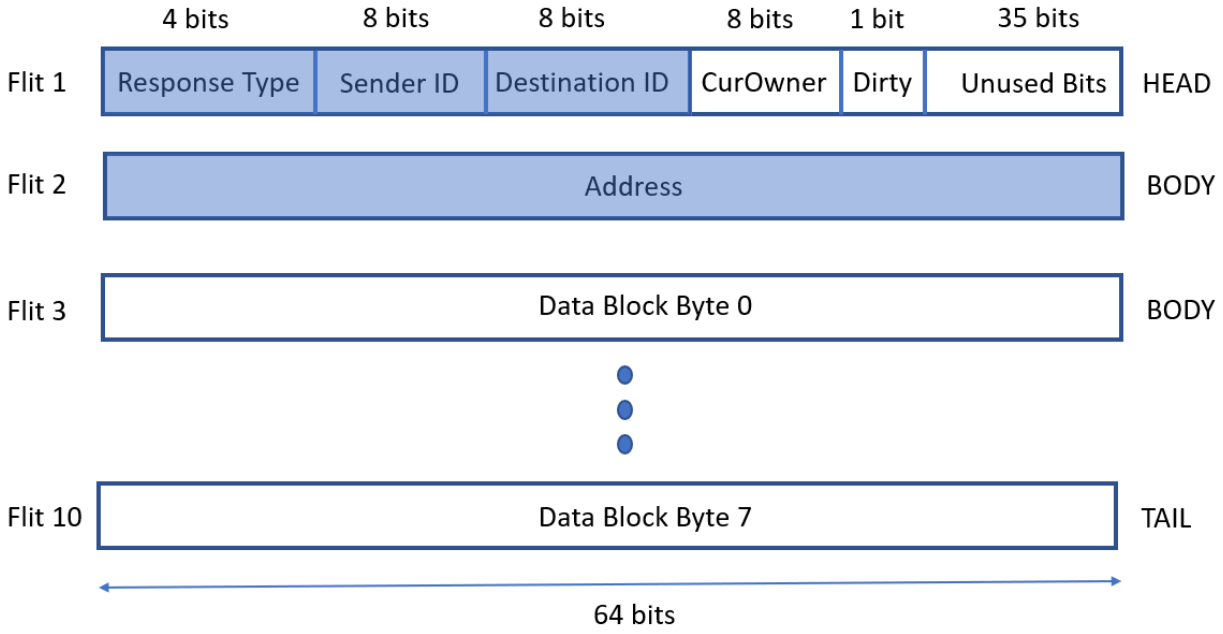

Figure 3.7: Head 64-bit flit within interposer layer

Figure 3.7 represents the flit structure of response message. The difference from request message is that response messages can contain 64 bytes of data as cacheline. Hence, the complete message is distributed over multiple flits as shown in the diagram.

3.3.3 Virtual Network

A key realization in our research is that a NOC is built up such that, the cache coherent messages that can be categorized into certain class, traverse through a designated physical link. Such a link is termed a virtual Network. The number of virtual networks in a NOC is determined by the cache coherence protocol. In our work we used the AMD MOESI hammer cache coherence protocol which has 4 virtual networks and all the different types of cache coherent messages traverse one of these virtual networks based on their type of message.

A SNI need not be placed on all the virtual networks. As per our threat model, the objective is to detect scenarios where certain parameters of the cache coherent messages have been modified by hardware trojans. This can be observed by analysing certain critical cache coherent messages,

which traverse certain particular virtual networks. Through this observation we were able to realize that by understanding the distribution of the cache coherent packet types across the virtual networks the SNI logic can be simplified by placing them on certain virtual networks. The details of which virtual network and what type of cache coherent messages have been taken into consideration are discussed in the cache coherent section. One of the key concerns in our research is that in order to deal with hardware trojans, we may need to place the SNIs in all of the virtual networks in order to monitor all the messages traversing most of the protocol layer. The reason being that in-spite of our analysis of possible security scenarios, there are still possibilities of complex hardware trojans which can lead to security breach if certain cache coherence messages are left out.

3.3.4 Virtual Channels

Virtual Channels can be considered as separate queues for each virtual network. If a virtual network is visualized as a physical link then virtual channels are parallel buffers which help avoid deadlock[29] and head-of-line blocking. Each of the virtual channels arbitrate for the same physical link, but this is resolved within the router on a cycle-by-cycle basis.

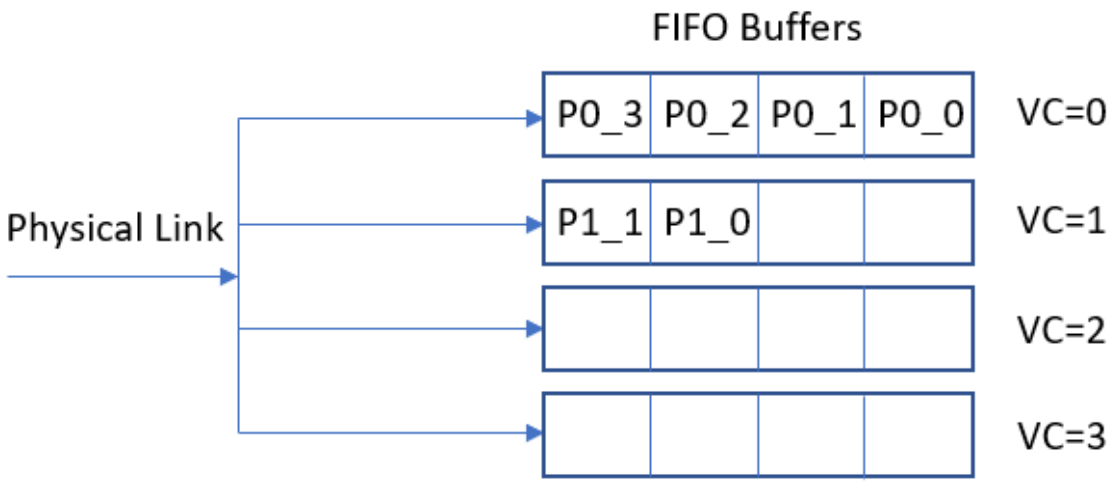

Figure 3.8: Four Virtual Channel per Virtual Network Configuration

Figure 3.8 depicts a configuration of 4 virtual channels per virtual network. As we can see a single physical link feeds into 4 virtual channels each of which has a 4 depth buffer. This means that each of these virtual channels can hold 4 flits before it gets blocked. The advantage of having separate virtual channels can be explained through a simple example. Let's consider a situation wherein two packets need to traverse from one router to another on the same virtual network. Suppose the two packets are named *P0* and *P1* where both P0 and P1 have 8 flits to be transferred. As we can see in figure 3.5, P0 has 4 of its flits occupying VC=0 and for some reason it's unable to proceed because of traffic further ahead. In such case, since VC=1 is available the packet P1 can proceed and start taking space in this virtual channel. This prevents the link from getting blocked just because one packet is stuck.

As far as the impact of virtual channel on SNI architecture, we did not find any situation where it directly impacts the internal architecture of the SNI. Although, the number of VCs per VN definitely determine the latency of the packets in the network. In configurations where the available VCs per VN were low, we observed high queuing latency introduced due to stalling of packets at Network Interface.

3.4 Cache Coherence Protocol

The cache coherence protocol plays a key role in the architecture of the SNI. It's a simple conclusion because it is the cache coherence messages that will be scanned and checked by the SNI. In this section we will focus on the two key aspects of the AMD MOESI Hammer protocol which impact the SNI architecture. One is the type of messages and the distribution of the messages across the virtual networks.

3.4.1 MOESI Hammer

The MOESI hammer[30] is a protocol which has been implemented in AMD's Opteron[31] chiplets. In this protocol each core has a L1-Instruction Cache and a L1-Data Cache. It also has a unified L2 Cache. The L1 and L2 caches are private to each core and are controlled through a shared cache controller. The MOESI hammer protocol has directories which interact with the

memory controller. The directories here are different from traditional directory based protocols in the sense that they do not maintain any directory state. Instead the requests from the requestor core is broadcasted to all the other cores and in parallel the directory issues a data fetch request to the memory. The response data received by the directory from memory is then sent to the requesting core as a response data packet. The reason for broadcast is simple, since no state is maintained with the directory, the only way for sharing cores to maintain the correct cache coherent states is through these broadcasts messages. This leads to a significant amount of network traffic but the benefit is the reduced cost of area of implementation.

Lets take a very simple example where Core 0, issues a GETS to directory 3. Figure 3.9 depicts this scenario. The GETS will traverse the interposer layer NOC as shown in the diagram with green arrows. Depending on the traffic it may or may not get blocked in its path. Once the GETS message is received by directory 3. The directory issues a data request to the MC-3 memory controller. In parallel the same GETS with a requestor ID of Core 0 is broadcasted to all the other cores in the system. This is shown in Figure 3.10.

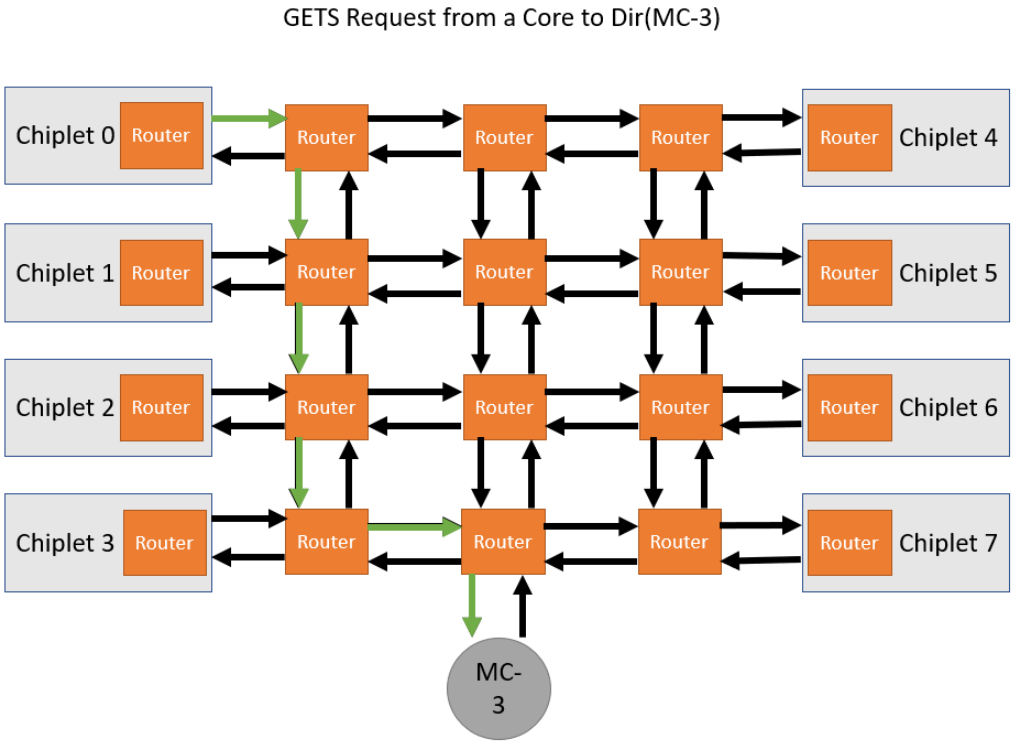

Figure 3.9: GETS issued by Core 0 to Director 3

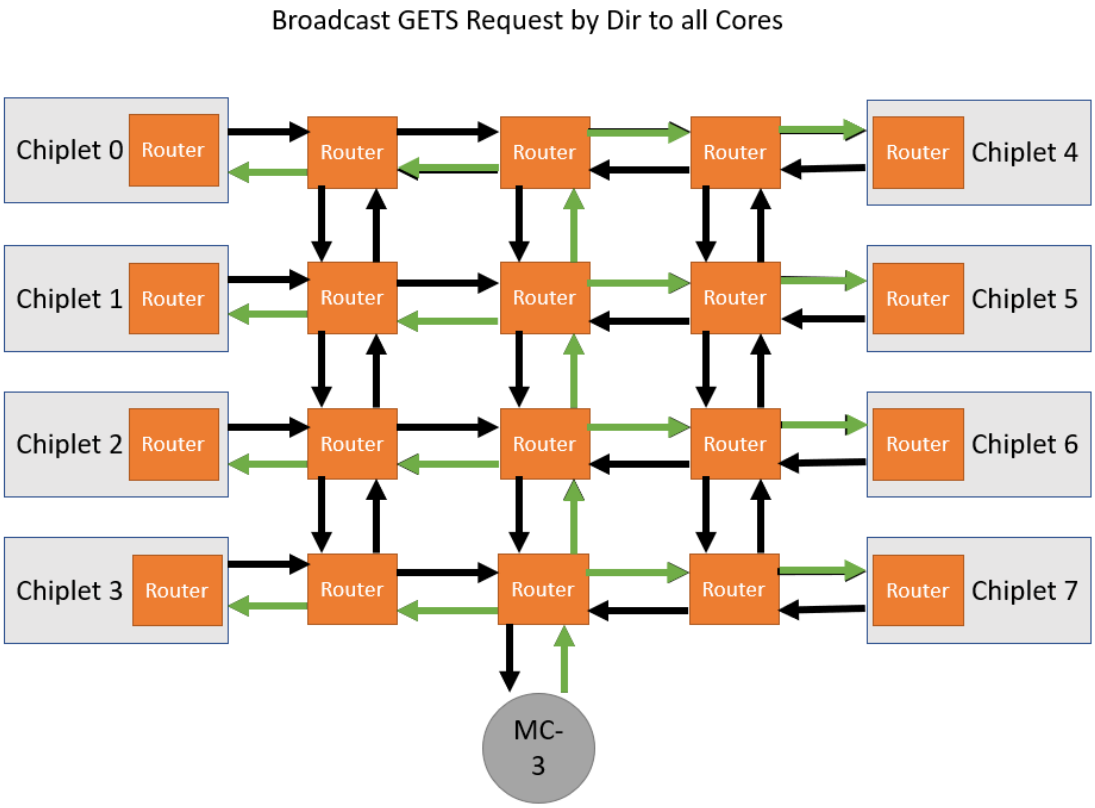

Figure 3.10: GETS being brodcasted by directory 3 to all other cores in chiplets

Once all of the cores receive this broadcast packet, there are two possible responses, ACK or ACK_SHARED where the first message informs the Core 0 that the cache block is not shared while the second one informs that the cache block is shared. Once all the messages are received by Core 0 from all the other cores as shown in Figure 3.11, it can either move to a shared state or an exclusive state based on the responses received from the rest of the cores. And after that it waits for receiving the data from the directory which then eventually finishes the cache transaction for that particular cache block.

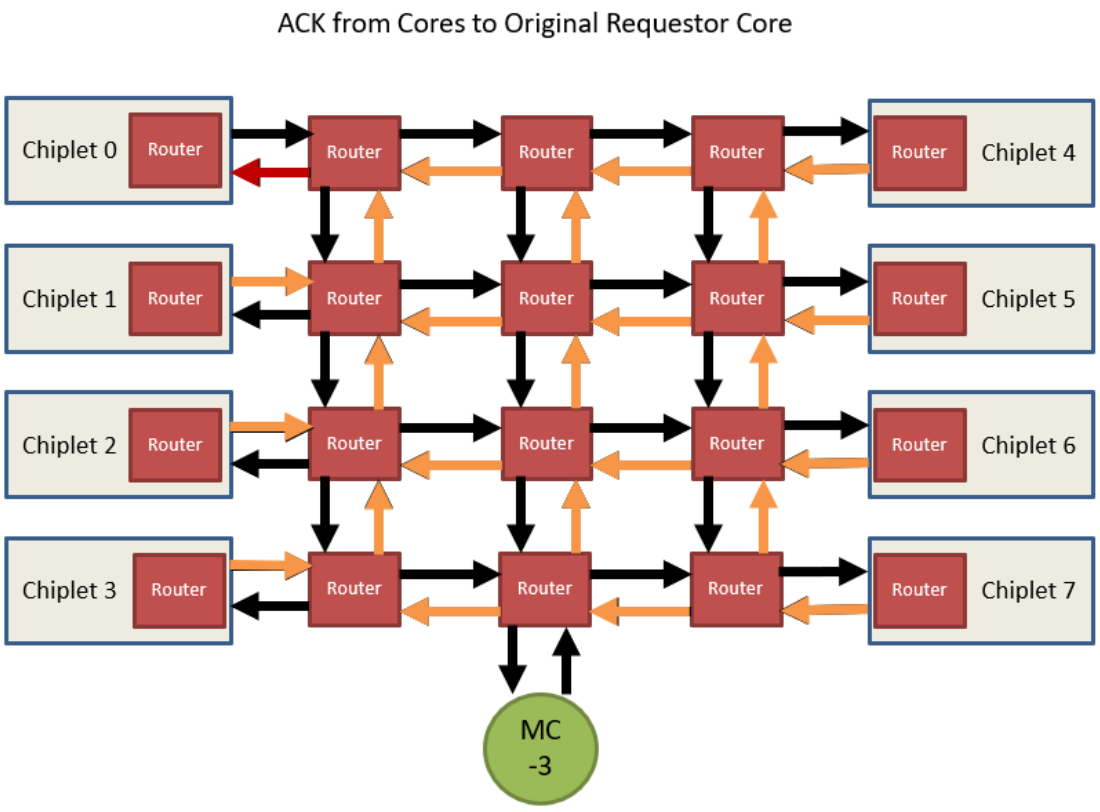

Figure 3.11: ACK being returned from all other cores to the requestor Core 0

We can observe that in a broadcast based protocol there are lots of messages being exchanged as compared to that of a traditional directory based protocol. One of the reasons we think that this protocol is suitable for performance impact measurement of the SNIs is because we want to observe the impact of the SNI in the network in terms of latencies. Hence, more the number of packets that traverse the network, better it is to understand the impact of the additional latencies introduced by the SNIs. Also the MOESI hammer have private caches for each core, this ensures that each chiplet's cache is private to itself which is an important requirement of our evaluation environment.

3.4.2 Message Types

The coherence messages can be broadly categorized into two types:

- Request Type: These messages are generated whenever any node in the cache coherence sys-

tem requests access to a region of memory or a cache block to be specific. GETS and GETX are good examples which are generated by the cache controller of a CPU core whenever it encounters a read or write instruction respectively.

- Response Type: These messages are generated in response to the request type cache coherent messages. ACK message in AMD MOESI hammer is an example which is sent from a responding core to a requesting core indicating that it does not share the cache block.

Request Type	VN	Response Type	VN
GETX	0	ACK	2
GETS	0	ACK_SHARED	2
MERGED_GETS	1	DATA	2
PUT	0	DATA_SHARED	2
WB_ACK	1	DATA_EXCLUSIVE	2
WB_NACK	1	WB_CLEAN	3
PUTF	0	WB_DIRTY	3
GETF	0	WB_EXCLUSIVE_CLEAN	3
BLOCK_ACK	1	WB_EXCLUSIVE_DIRTY	3
INV	1	UNBLOCK	3
		UNBLOCKS	3
		UNBLOCKM	3

Table 3.1: Cache Coherence Messages in AMD MOESI Hammer

Now lets take a look at all of the messages in AMD MOESI Hammer protocol and the corresponding virtual networks that are used by the messages.

Table 3.1 shows all of the cache coherence messages supported in AMD MOESI Hammer protocol and the corresponding VNs which are used to carry those messages.

3.4.3 Message Analysis

We can now analyse the messages based on the type of possible interactions between the cores, directories and memory controllers in the system as follows:

- **GETX**: *GETX* is a message which is generated when a core encounters a store type instruction. This results in the cache state changing from *Invalid* to *Modified*, if the cache block has not been fetched before. The *GETX* is issued from the requesting core to the directory which acts as the home node for the particular cache block. Since, this is an interaction where a core is requesting access to a memory, it is important that GETX messages are monitored by SNI.

- **GETS**: *GETS* is a message which is generated when a core encounters a read type instruction, such as data load or instruction fetch. This results in the cache state changing from

Invalid to *Shared* state, if the cache block was not fetched earlier. The *GETS* is issued from the requesting core to the directory which acts as the home node for the particular cache block. Just like GETX, this GETS message also needs to be monitored.

- **MERGED_GETS**: MERGED_GETS is a message which is forwarded by a directory to a core. A situation may occur, such that a cache block is blocked at the directory, and another core requests access to the same cache block through a GETS message. In that case a *MERGED_GETS* message is constructed and forwarded to the core owning the cache block. Then the owner core responds to the MERGED_GETS with a *DATA_SHARED* to the second sharing core requesting for it. Now, a MERGED_GETS is not like GETS/GETX where it is a direct memory access request. But a malicious core can formulate a MERGED_GETS and forward it to a owner core itself and the unsuspecting owner core can respond back with critical data. Hence, in order to avoid such situation we need to monitor MERGED_GETS messages.

- **PUT**: A *PUT* message is initiated by a core and sent to a directory when it wants to write-back a cache block. The directory usually responds with a *WB_ACK* message letting the core know that it is ready for write-back. The core then sends the cache block along with its data in the form of a write-back message type packet. A PUT is a tricky message which does not directly lead to a security fault in the system. But it is the initiator for a write-back sequence. Hence, it should be monitored.

- **WB_ACK**: As explained in the case for *PUT*, a *WB_ACK* is a response from a directory to the core which wants to write-back. It's very tricky to just use a WB_ACK type message to create some sort of security breach as per our threat model. Worst case, even if a WB_ACK is sent from a malicious core to another core, most likely it will drop the message because it was not the one initiating the request in the first place.

- **WB_NACK**: The *WB_NACK* is a message which sends a negative acknowledge from directory to core when a *PUT* request is placed by a core. This situation occurs because of race

condition at the directory when the directory thinks that it owns the cache block. But to keep it simple, the premise of WB_NACK is similar to WB_ACK and hence this also does not need to be monitored.

- **PUTF**: *PUTF*'s functionality is very similar to that of PUT. It is issued be a core to a directory indication that it wants to write-back a cache block. But here the trigger for the write-back is a *FLUSH* instruction instead of a cache replacement. Similar to PUT we should monitor this packet type to avoid initiation of a fabricated write-back sequence.

- **GETF**: A *GETF* is issued by a core when it has cache block that it wants to write back because of encountering a *FLUSH* instruction. This initiates an exchange of messages between the core and the directory which eventually leads to a write back from the core to the main memory through the directory. But one interaction can be that a malicious core formulates a GETF packet and forwards it to the DIR. The directory then broadcasts the GETF, in order to get the owning core to respond with a fresh copy of the data to the requesting malicious core. In order to avoid this situation we need to monitor GETF.

- **BLOCK_ACK**: This is a request type message sent from the directory to a core which had send a GETF type message. Based on our analysis this packet is not necessary to be monitored since it is a response which even if manufactured by a malicious core does not allow it to extract any kind of important information.

- **INV**: This is an *INVALIDATE* message which is initiated by the directory and sent to cores. Invalidation leads to removal of a cache block from a cores private cache and that's it. It does not necessarily allow a malicious core to extract critical data by accessing memory regions that it does not have permissions for. It may allow the malicious core to hinder the performance, by spamming the coherence network with invalidate messages which it is not supposed to. Hence, for safety reasons it is better to monitor invalidate messages.

- **ACK**: As mentioned earlier, MOESI Hammer is a broadcast based protocol. The directories

do not keep track of which cores share a cache block. Hence, any GETS request from a core is broadcasted to all other cores. If any of the other cores owns the data in its *OWNED* state then it responds back with the data through *DATA_SHARED*. Since ACK is a response type which informs the requestor core that a responding core does not share the cache block, even if a malicious core synthesizes a ACK message it cannot harness any critical data. But a broadcast message to other cores allow snooping of the cache block addresses. We tackle this problem using SNI which is explained in later sections.

- **ACK_SHARED**: This message informs the requestor core that a responding core has a shared copy of the cache block. Thus the requestor core can move to a *SHARED* state instead of a OWNED/EXCLUSIVE state. The only way a malicious core can use this type of message is to spam the system with ACK_SHARED for a cache block that it does not owns and thus trying to hang the cache coherency protocol itself. This may be treated as an effort by the malicious core to shut the system down, hence it is important to monitor this message.

- **DATA**: This message is a response type message from a directory to a requesting core and contains the data originally requested. This is sent in response to a GETS/GETX and hence need not be monitored since the GETS/GETX messages have already undergone checks.

- **DATA_SHARED**: DATA_SHARED is used be a responding core to send a copy of the cache block that it owns to another requesting core. One interesting situation maybe, that a malicious chiplet spams the network with a DATA_SHARED message for a cache block, and thus trying to inject its data into the requesting cores private cache even before the original responder could have responded. Hence, this packet is also being monitored.

- **DATA_EXCLUSIVE**: This packet does the same job as a DATA_SHARED the only difference being that instead of GETS here the triggers is a GETX. Hence, for the same reasons as DATA_SHARED we monitor this message as well.

- **WB_CLEAN**: A message sent from a core to directory trying to write-back a cache block in owned state. Since, write-backs are critical to memory access interaction, all the different write-back type messages are being monitored. The fact that directories can accept write-back from any core irrespective of whether they have permission to write-back a cache block or not makes it risky to not monitor write-back messages.

- **WB_DIRTY**: Same as WB_CLEAN.

- **WB_EXCLUSIVE_CLEAN**: Same as WB_CLEAN. The difference being if the cache block is now in modified state when the core is trying to perform a write-back.

- **WB_EXCLUSIVE_DIRTY**: Same as WB_EXCLUSIVE_DIRTY.

- **UNBLOCK**: An UNBLOCK messages, is sent from a core to a directory indicating the end of transaction for a cache block thus releasing it for subsequent cache transactions. This is required to avoid race conditions. There is no straightforward way for a malicious core to use it to extract any information. But it can be used by malicious core to spam a network and lead to unintentional states of cache blocks leading to protocol malfunction. Hence, even though it is difficult to use UNBLOCK for data extraction it can be used to break the cache coherence protocol.

- **UNBLOCKS**: Same as UNBLOCK.

- **UNBLOCKM**: Same as UNBLOCK.

Although based on our analysis, certain message types need not be analysed, we decided to analyse all the cache coherence messages anyways, to avoid any kind of complex scenario that can arise out of intricate trojan designs. Irrespective of the cache coherence protocol and the complexity of the hardware trojans, if we analyse the "requestor", "destination" IDs along with the "address" of the cache block for every type of message, it is possible to avoid any kind of security vulnerability from a cache coherence perspective. This has made us realize that more research can

be done into the types of hardware trojans which can leverage the various states in cache coherence protocols to breach the security.

3.4.4 Malformed messages

In the previous sections we discussed the different architectural aspects which can be exploited by hardware trojans. However, these are based on the assumption that trojans still fabricate messages by modifying the fields of cache coherence messages. There is a possibility that a message can be created with randomized bits to create denial-of-service attacks. In order to avoid such attacks, SNIs have to analyse the packet structure, to determine that the fields of the messages abide the rules of cache coherence protocol. For example, destination ID for a GETS/GETX request type message can only be the four directories. Hence, if the destination ID contains ID of any other core then it can be considered an illegal message.

VN	VN Name	SNI_Checks
0	request_from_cache	1) Requestor ID range as per Chiplet boundary Placement 2) Destination ID only directory(64,65,66,67)
2	response_from_cache	1) Sender ID range as per Chiplet boundary Placement 2) Destination ID only other cores(0 to 63)
3	unblock_from_cache	1) Sender ID range as per Chiplet boundary Placement 2) Destination ID only other cores(64,65,66,67)

Table 3.2: SNI design logic to detect malformed messages

One observation which we can derive based on our message types and the virtual network analysis, is that there are certain message types which are carried through specific virtual networks. Using this knowledge, we can implement SNI logic as shown in Table 3.2 where certain message types can only have a specific type of requestor or destination ID. The virtual networks shown in Table 3.2, represent the output physical links of cache controllers. SNIs have been placed on these virtual networks at the chiplet-interposer boundary within the interposer layer routers. Then each SNI, as per its placement on a virtual network, ensures that the requestor and destination IDs are as per the rules laid out in the table.

4. SNI AND APU TABLE IMPLEMENTATION

In this chapter we will look at the implementation and functional aspects of SNI and the APU Table. We have already discussed the various factors that decide the SNI architecture and also the parameters that need to be checked by the SNI in order to detect security breaks.

4.1 SNI Architecture

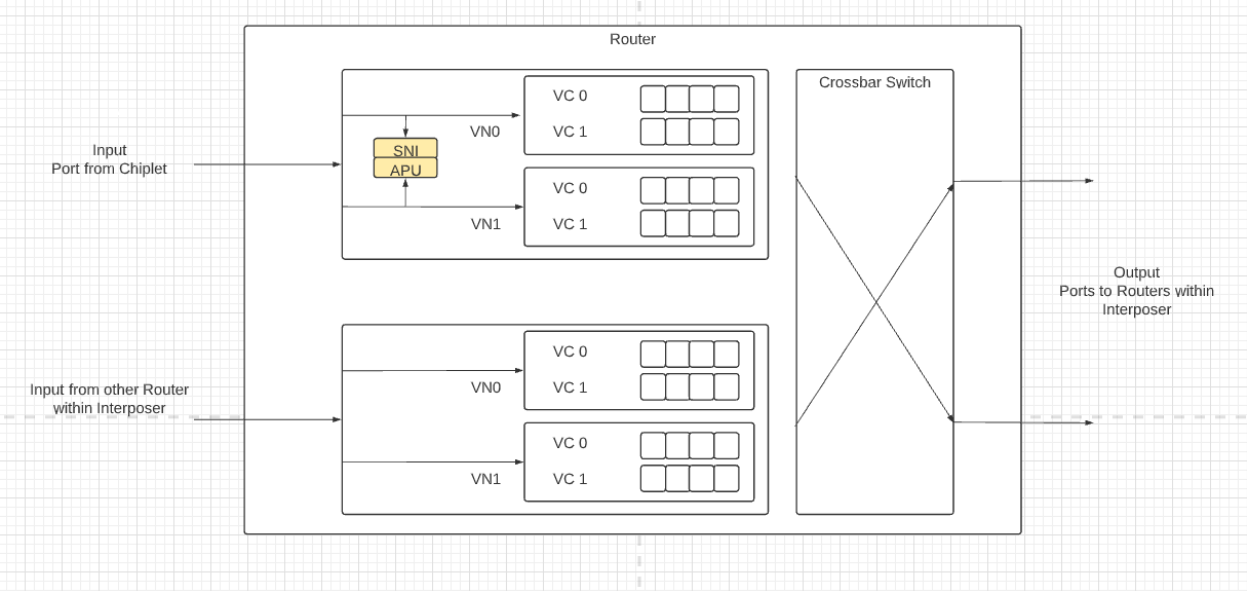

Figure 4.1: Router Placement to only scan packets

Figure 4.1 depicts the first configuration. Here the SNI taps the physical link(virtual network). This configuration allows the SNI to scan the packets without adding additional latency to the packets traversing through the virtual network. Although it may seem that this will help save area on the SNI implementation but that's not the case. The logic responsible for scanning the packets needs to be duplicated for every scanned VN. Each VN is carrying packets independently, hence we need the scan logic to be independently monitoring the packets on the individual VNs. The other disadvantage is that once a insecure message is recognized, the SNI won't be able to stop the

packet from traversing the network. Instead the SNI will record all the information and respond to the OS with a machine check failure message. This will eventually let the OS know that a security breach interaction has been encountered.

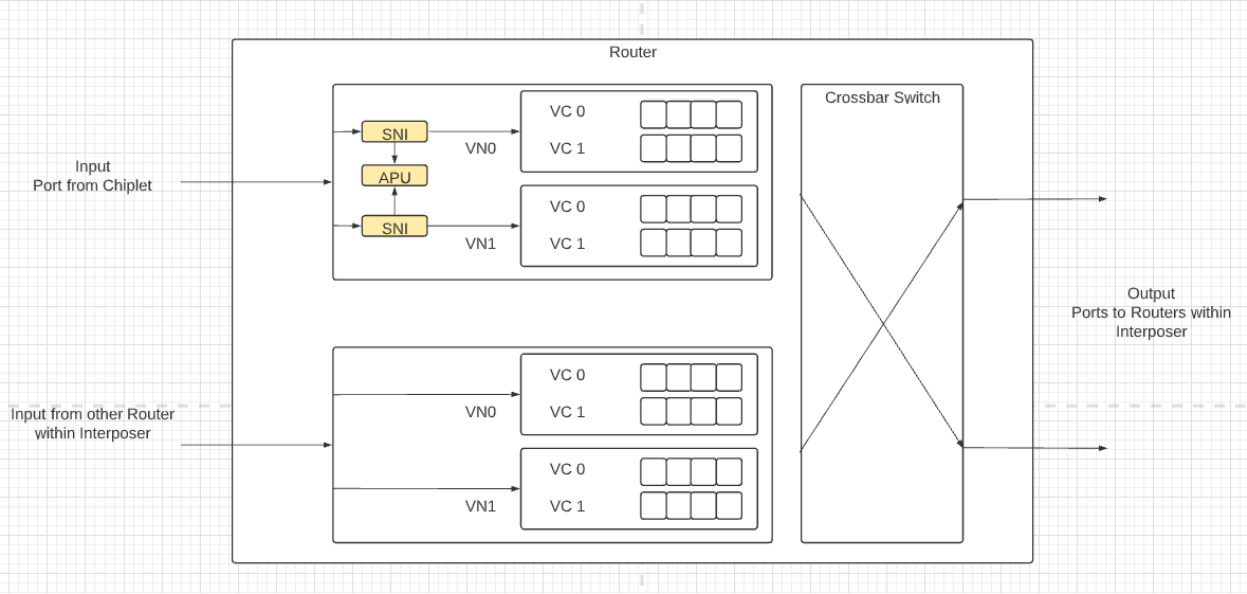

Figure 4.2: Router Placement to scan and handle dropping malicious packets

Figure 4.2 shows a different configuration where the SNI is embedded into the physical link between the input port and that of the internal buffers of the Router. The only difference here is that every network packet that traverses through the SNI undergoes a certain amount of latency. This latency is unavoidable because the SNI will extract the information from the packets and then use it to compare the permissions with the APU_Table. The advantage of this configuration is that, if a malicious packet is recognized, it can be dropped from the network by the SNI before it further proceeds. The SNI will then formulate a message to the OS informing about the malicious packet and raise a machine check failure. The disadvantage being in terms of network latency.

Out of the two configurations, we have went ahead and chosen the second configuration because it is a more robust implementation because a malicious packet is dropped as soon as it enters the SNI.

Another important aspect of SNI which was worked upon was to use the SNI and APU_Table to modify some of the cache coherence messages.

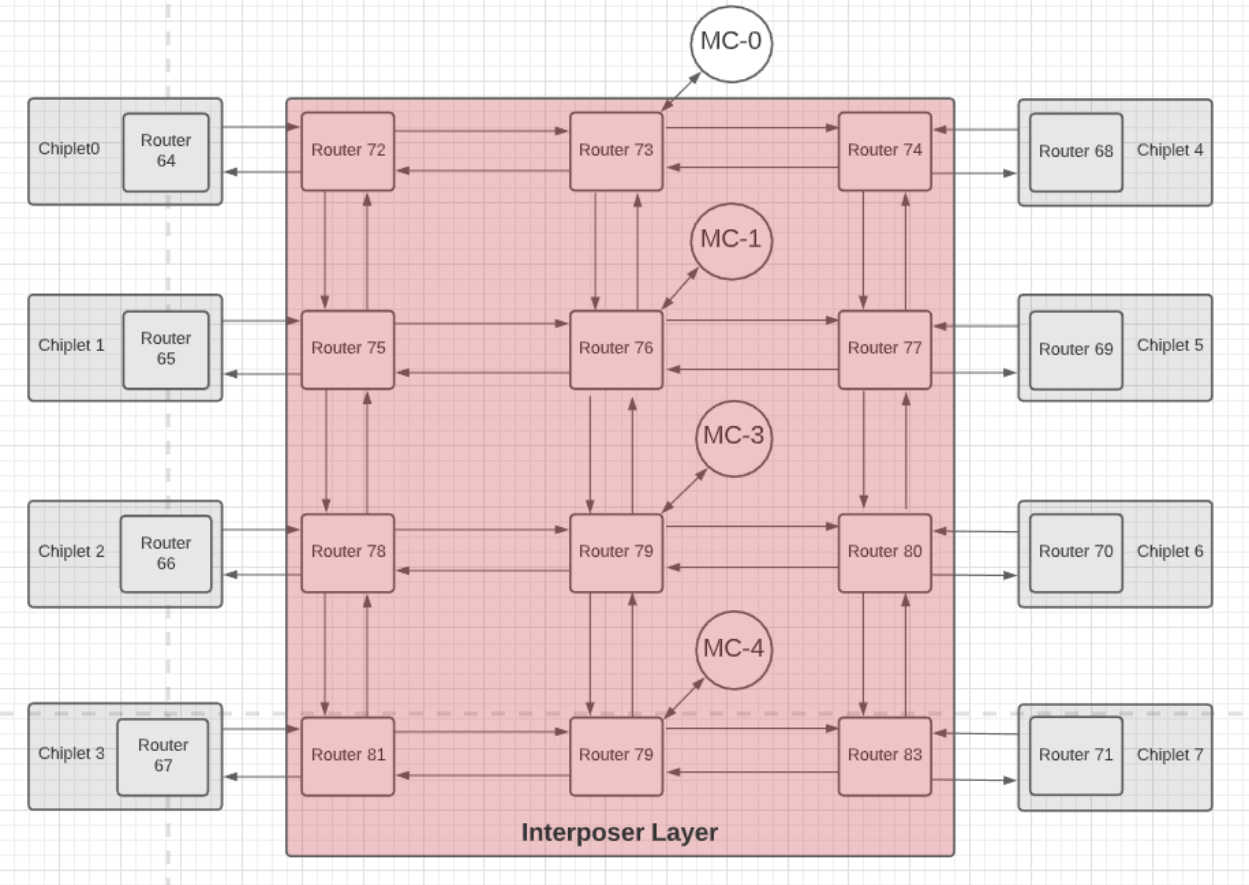

Figure 4.3: Router Placement to scan and handle dropping malicious packets

In Figure 4.3 we can observe the Routers connecting the Memory Controllers to the system through the interposer layer. We observed, that since the MOESI hammer protocol is a broadcast based protocol, for every GETX/GETS request received by the directory, it forwards the GETS/GETX requests to all the other cores in order for the original requestor core to ascertain if the cache block is shared or not. Now a situation may occur where because of these broadcast packets, a malicious core gets to view which core is accessing which regions of memory which is a concern of possible security breach. In order to avoid this, the SNIs were slightly modified

in their functionality and added at the Routers connecting the memory controllers. The primary role for these SNIs is to check if a broadcast message is directed towards a core is necessary or not. This information can be already obtained from the APU_Table. If the SNI determines that the broadcast GETS/GETX packet has a destination core which does not share the cache block based on the information obtained from the APU_Table, it converts those GETS/GETX messages to ACK messages and redirects to the original requestor core. Thus a malicious core that does not share data with another unsuspecting core will never be able to know which cache blocks or regions of memory it is accessing.

In section 3.3.1, we explained the sequence of a read request by a core. The SNIs at the boundary between interposer and memory controller redirect the broadcast GETS/GETX to ACKS as shown in Figure 4.4

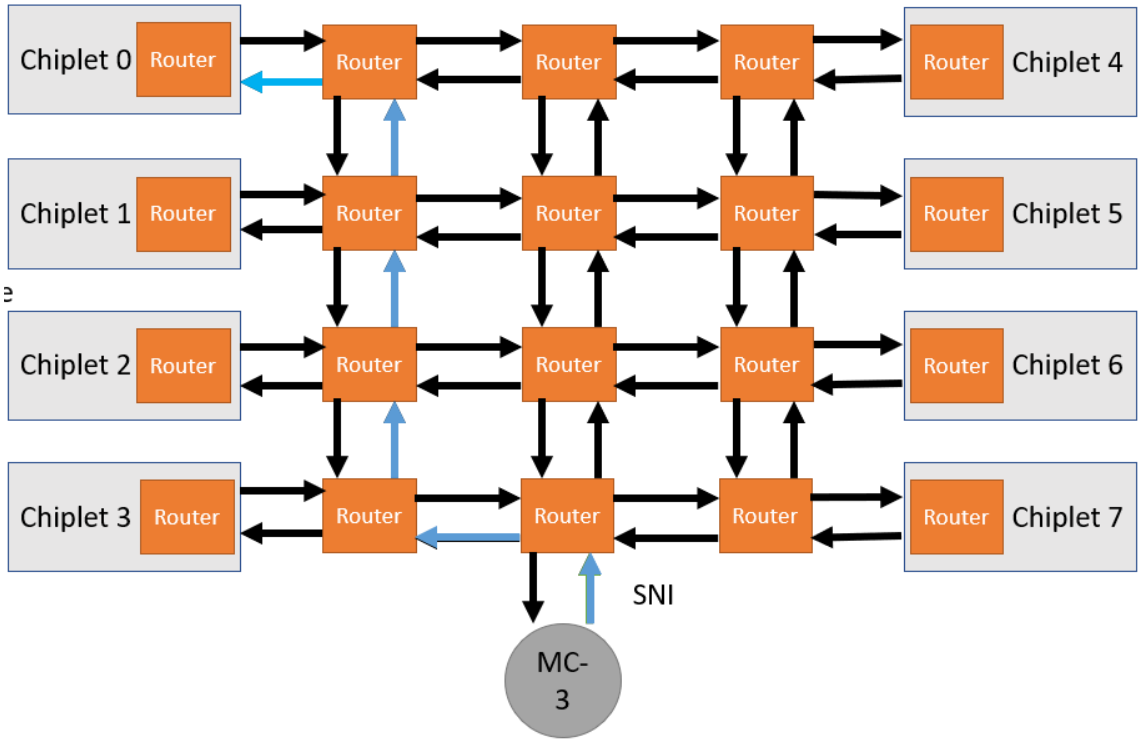

Figure 4.4: Broadcast GETS/GETX to ACK

This does add latency to the packets that are being re-routed and also puts load in the path between the directory and the requestor core. But we are going to provide the data for these in later sections.

4.2 APU_Table

The APU_Table in simple terms is a lookup table that is maintained by the trusted OS. As applications are run on the cores, memory will be accessed by the cores in the form of instruction or data. The OS is responsible for the memory mapping of the virtual to physical memory[32] for these applications. The APU_Table contains the physical address ranges for which a memory has been allocated to a core/chiplet. The APU_Table contains the allocated address ranges on a chiplet basis.

2-bits for each Chiplet								64-bit	64-bit
Chiplet 7	Chiplet 6	Chiplet 5	Chiplet 4	Chiplet 3	Chiplet 2	Chiplet 1	Chiplet 0	Start_Addr	End_Addr
0	0	0	0	0	0	0	3	0	64MB
0	0	0	0	0	0	3	0	64MB	128MB
0	0	0	0	0	3	0	0	128MB	192MB
0	0	0	0	3	0	0	0	192MB	256MB
0	0	0	3	0	0	0	0	256MB	320MB
0	0	3	0	0	0	0	0	320MB	384MB
0	3	0	0	0	0	0	0	384MB	448MB
3	0	0	0	0	0	0	0	448MB	512MB
0	0	0	0	0	0	3	3	512MB	576MB
0	0	0	0	0	1	1	3	576MB	640MB

2-bits for each Chiplet								64-bit	64-bit
Chiplet 7	Chiplet 6	Chiplet 5	Chiplet 4	Chiplet 3	Chiplet 2	Chiplet 1	Chiplet 0	Start_Addr	End_Addr
3	0	0	0	0	0	1	1	832MB	896MB
0	1	3	1	0	0	0	0	896MB	960MB
1	1	0	0	0	1	1	3	960MB	1024MB

Figure 4.5: APU_Table

Figure 4.5 depicts the structure for our APU_Table. You can observe that 2-bits have been

allocated for each chiplet, which defines the permission levels of applications that are being executed within those chiplets. There are 4-permission levels for each chiplet which are described as follows:

- **No Access**: The chiplet has no access to a certain memory address range.
- **Read-Only Access**: The chiplet has read-only access to a certain memory address range.
- **Read-Write Access**: The chiplet has read and write access to a certain memory address range.
- **Execute**: The chiplet has access to execute the code in a given memory range.

These permission levels are determined by the OS and programmed into the APU_Table during run-time.

Apart from that there are two 64-bit spaces for start and end address ranges. One of the issues with multiple cores within a chiplet is the issue that multiple applications might be running at any given time, due to which the applications within a chiplet might be accessing different physical regions of memory. In order to manage it, our approach is that the OS will allocate a chunk of memory to a chiplet, let's say an example of 64MB. Any new memory allocation for stack or heap has to be allocated by the OS from withing the specified memory chunk. All the applications running within a specific chiplet will get new physical memory allocated within the memory chunk initially specified by the OS. Once, the OS runs out of memory space it can use another available slot in the APU_Table or replace a slot which was least recently used. The functionality of the APU_Table is very similar to that of a TLB.

In our configuration for the APU_Table we have allocated 64 entries to the APU_Table and allocated 64MB chunks of memory for every new entry into the APU_Table. The number of entries being allocated for the APU_Table will determine the clock cycle latency of accessing an APU_Table by the SNI in order to check the message parameters. The more the number of entries, more number of clock cycle may be required to access the APU_Table.

4.3 Operating System Management

The OS is responsible for memory management for the applications running in our system. Although our primary focus is not on the operating-system aspects related to security, the OS still has to handle the entries that are allocated in the APU_Table. In order to do that the OS needs to be aware of the memory allocated to chiplets and the applications that are running within those chiplets.

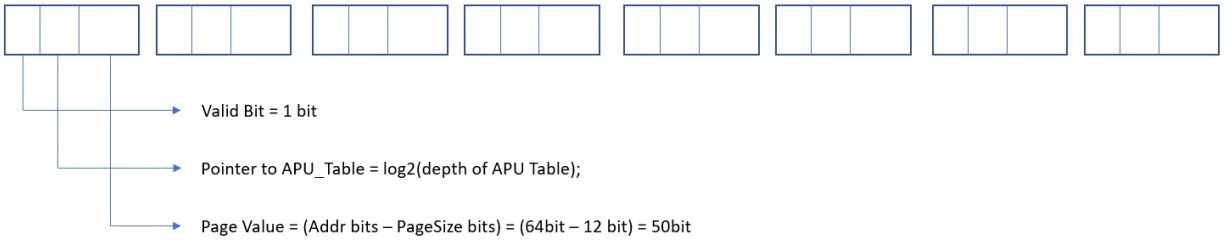

Figure 4.6: Page pointer per Chiplet at the OS

Figure 4.6 shows eight entries that are maintained by the OS for every chiplet. These entries have three fields each:

- **Valid Bit**: This is to maintain that the entry has valid information.
- **Pointer to APU_Table entry**: This points to the next entry in the APU_Table that should be used by the OS for allocating a new entry.
- **Page Value**: This maintains the next physical page address which should be the start page for the new allocated memory chunk.

The OS has a simple algorithm which allocates entries based on the current values entered in the eight entries for each chiplet. These allocations are written into the APU_Table as well. As any application running within a chiplet requests more memory from the OS, it is allocated by OS from the already available memory chunk that has been entered into the APU_Table. Initially

these 8 entries will be empty. But as each chiplet has applications running on it, it will request memory from OS. The OS will then be able to maintain the last allocated physical page address in these entries and based on that it can allocate new pages to the applications and enter the same into the APU_Table. If the amount of memory requested by an application exceeds the 64MB memory chunk initially allocated to a chiplet, another entry within the chiplet is utilized. The replacement algorithm is simplistic and we leave it to future work for a detailed exploration.

5. METHODOLOGY AND RESULTS

5.1 System Configuration

The primary goal of our research is to get an in-depth understanding of the SNI architecture. With that we also want to determine the performance hit of adding the SNIs at the interposer layer. As mentioned in section 4.1 we have the SNI embedded into the physical links. This allows us to get the worst case impact of SNI on the performance of a system.

Table 5.1 depicts the configuration for our system. We are modelling a 64-core system based on the Rocket-chip system [33]. One of the reasons for opting for a 64-core system is we want to simulate a multi-core multi-chip system and Rocket-chip follows such a structure. We have used gem5[34] as our architectural simulator. We can select different types of processor types from gem5 but we opted for *TimingSimpleCPU* because our objective is not to benchmark the performance of the processor but rather measure the performance impact of the SNI from a networking perspective. Hence, a TimingSimpleCPU allows a simpler version of the CPU model which helps improve the run-time of the benchmarks. Each of the chiplets contain 8 cores as shown in Figure 3.1. As per the MOESI Hammer cache coherence protocol, each core has a private cache. We have private L1 Instruction Cache of size 32KB and L1 Data Cache of size 64KB. The main memory is a 4096MB distributed across 4 directories. The directories here share the main memory across them, in order to distribute the memory accesses, they don't maintain any bit vector to know which cores share a cache block. The directories only act as a medium between requestors(cores) and main memory. Another factor that was important for simulation is the clock frequency of operation. The CPU is being simulated at 1GHz clock and as per our model the chiplet and the systems within it are being simulated at a frequency of 1GHz. Since the interposer layer will be manufactured using process nodes which have higher latencies, it was practical to simulate the interposer layer and the systems within it at a slower clock frequency of 250MHz. This will provide much more realistic results when it comes to network latency impacts of the SNI.

Architecture
64-core TimingSimpleCPU; SingleThread at 1.0GHz
8 Chiplets each containing 8 cores
Private L1 I-Cache 32KB
Private L1 D-Cache 64KB
4096MB Main memory distributed across 4 Directories
garnet2.0 network model
Cache Coherence Model
AMD MOESI Hammer
Clock Frequency
Chiplet frequency = 1GHz
Interposer Layer frequency = 250MHz

Table 5.1: System Architecture Configuration

5.2 Observation Parameters

In order to observe the performance impact of the SNI we need to focus on certain key parameters:

- **Speedup**: Instructions per Clock Cycle(IPC) measures the average number of instructions executed over the total clock cycle duration for a specific benchmark run. In our analysis we calculated the ratio of IPCs obtained with SNI enabled and disabled configuration. The speedup in our results is the ratio of IPC obtained with SNI enabled to that with SNI disabled. The ideal speedup value would be 1 but that's not practical. We would obtain a less than 1 speedup because of the latency introduced by the SNIs.

- **Average Packet Queuing Latency**: The average packet queuing latency is the latency that a packet observes at the network interface before it enters the SNI. The Network Interface is the module which is responsible for converting a cache coherence message into a flit and then relay it through the network. But the buffers at the input port of the router connected to the NI may be blocked because of network traffic. This is counted as queuing latency.

- **Average Packet Network Latency**: The average packet network latency is the latency that a

packet observes after entering the network. As a packet enters the network, it has to traverse the network of routers and links in order to reach its destination. This latency that a packet observes till it reaches the destination NI, where it is converted back into cache coherence messages, is counted as the packet network latency.

- **Average Packet Latency**: This is the combined latency that a packet observes. It is a combination of the queuing and network latency.

The above parameters will be used to analyse our SPEC benchmarks and the performance drop we observe when we enable SNIs in our system. Although we will analyse the network latencies, the speedup should provide an overview of the amount of impact that SNIs have in our system.

5.3 Results & Analysis

5.3.1 Performance Impact

In this section we take a look at the results obtained after running the SPEC benchmarks. We ran the simulation with two configurations. One configuration where the SNI is disabled and the other configuration where the SNI is enabled. We performed benchmark simulations for both the configurations and calculated the fractional speedup factor. Here are the results:

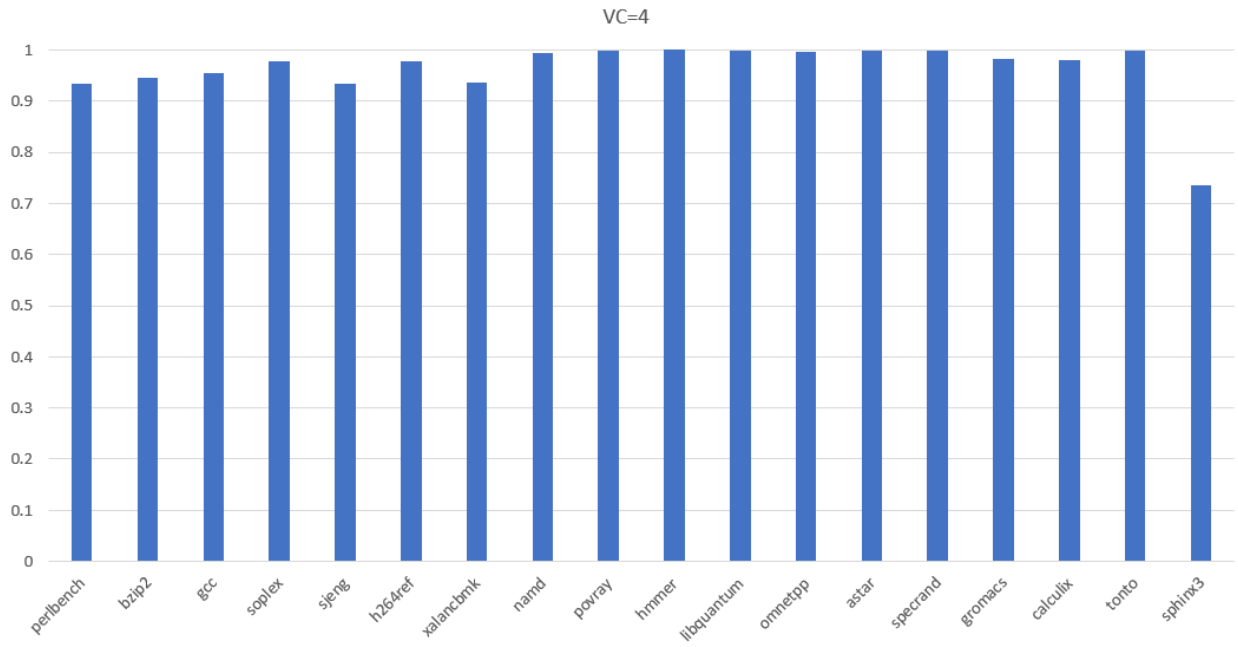

Figure 5.1: Speedup ration between SNI Enabled and SNI Disabled IPC

We can observe in figure 5.1 the speedup values with SNI enabled compared to SNI disabled configuration. The speedup values are less than 1, which is expected since we are expecting a reduction in performance due to the latencies introduced by the SNIs in the network. Some benchmarks such as perlbench, gcc, bzip2 have speedup ratio of 0.93, 0.94 and 0.95 respectively. Benchmarks such as povray, hmmer, libquantum have spedup ratio of around 0.99. The difference in these ratio values is because of difference in the cache hit ratios for the benchmarks. Benchmarks such as povray, hmmer, libquantum have a better cache hit rate compared to perlbench, gcc, bzip2. The geometric mean of the speedup for all the benchmarks is around 0.97, which indicates that the speedup loss due to SNIs is a minor fraction. The key reason being that any performance loss will be dependent on the number of packets injected into the network due to cache misses. We can observe that sphinx3 benchmark has a speedup factor of around 0.73 due to it's high cache miss rate, although that is particular to the case of sphinx3. In modern computing systems, cache hit rates are high [35] which leads to less number of packets being generated to interact with other cores and main memory. Hence, it can be said in modern day computing systems, with high cache hit rates,

SNIs can provide the added benefit of security at the hardware without significant performance loss.

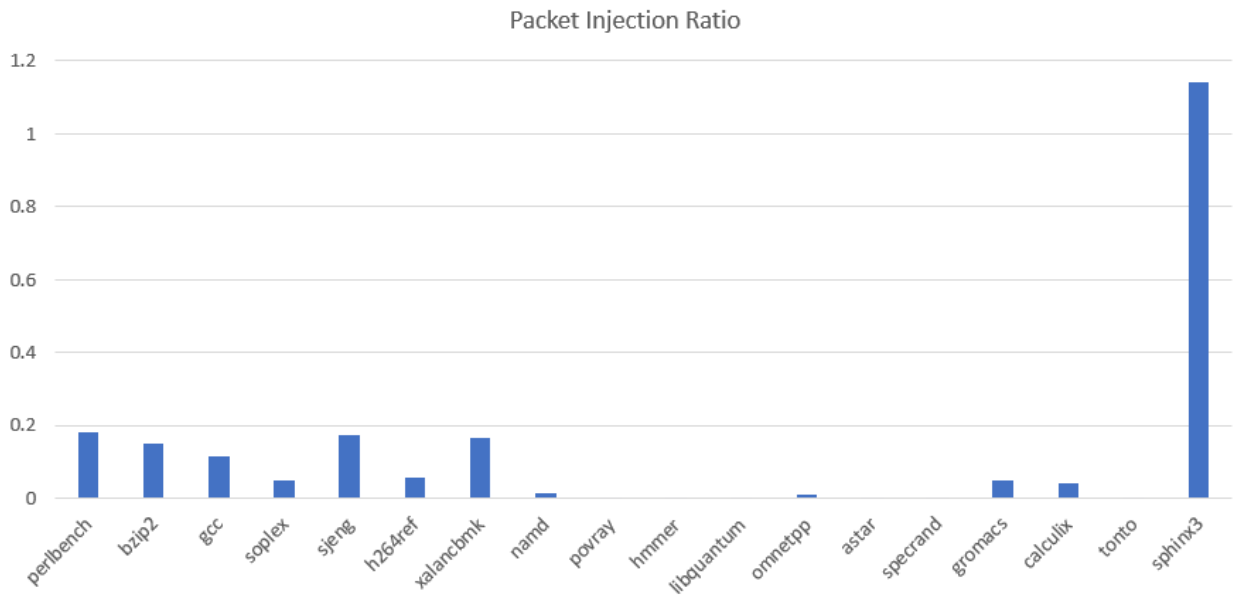

Figure 5.2: Ratio of number of packets injected to number of instructions

Figure 5.2 shows the ratio of number of packets injected to that of number of instructions processed. We can observe that for sphinx3 benchmark the ratio is around 1.14 which is significantly higher than other benchmarks. This explains the reason as to why we observe slightly higher performance loss for sphinx3 benchmark compared to others.

5.3.2 Virtual Channel Configuration

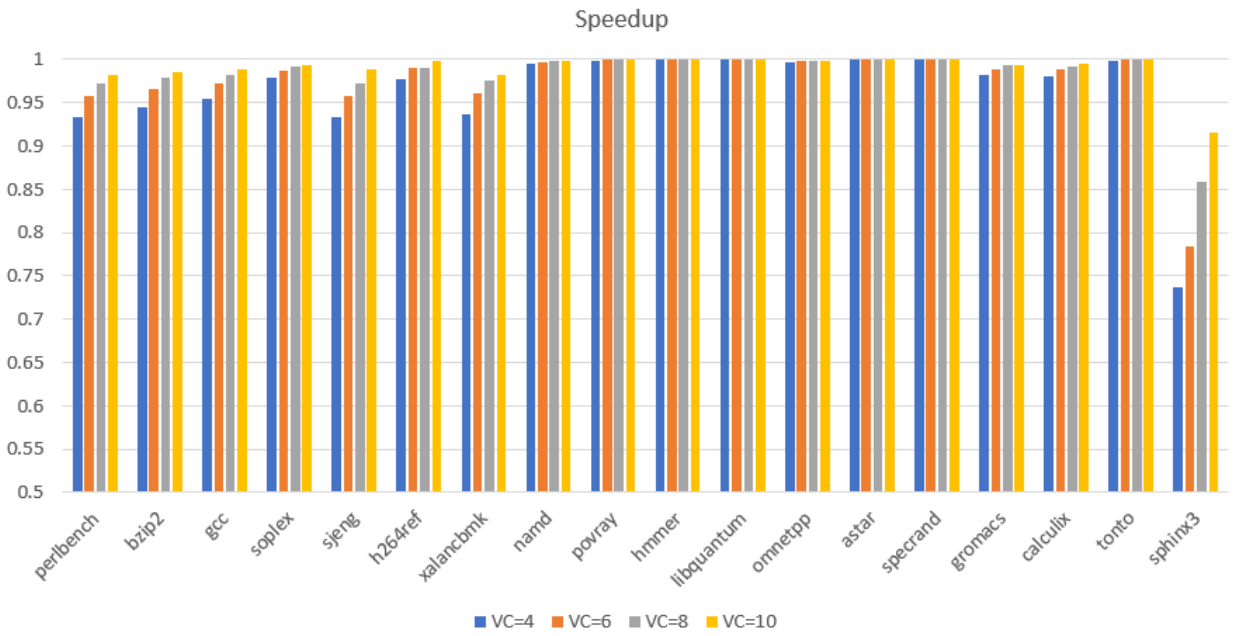

Figure 5.3: Speedup Factor for different vc_per_vnet configurations

Figure 5.3 depicts the change in speedup of the benchmarks with four different configurations of vc_per_vnet. We can observe that with increased number of available VCs the speedup approaches the ideal factor of 1. The change in speedup from VC=4 to VC=8 is significant, while we get diminishing returns for VC=10 configurations. Although, a configuration of VC=10 is not practical, it is presented to show that there is not much performance difference between VC=8 and VC=10 configuration. This observation of improved speedup with more VCs is expected since this helps reduce the queuing latency of packets at the Network Interface.

5.3.3 Latency Changes

In this section we will analyse packet latency parameters for the benchmarks. We have again ran the simulation for two configurations, one where the SNIs are enabled and one where they are disabled. This allows us to measure the latency parameter changes that SNI has impact on.

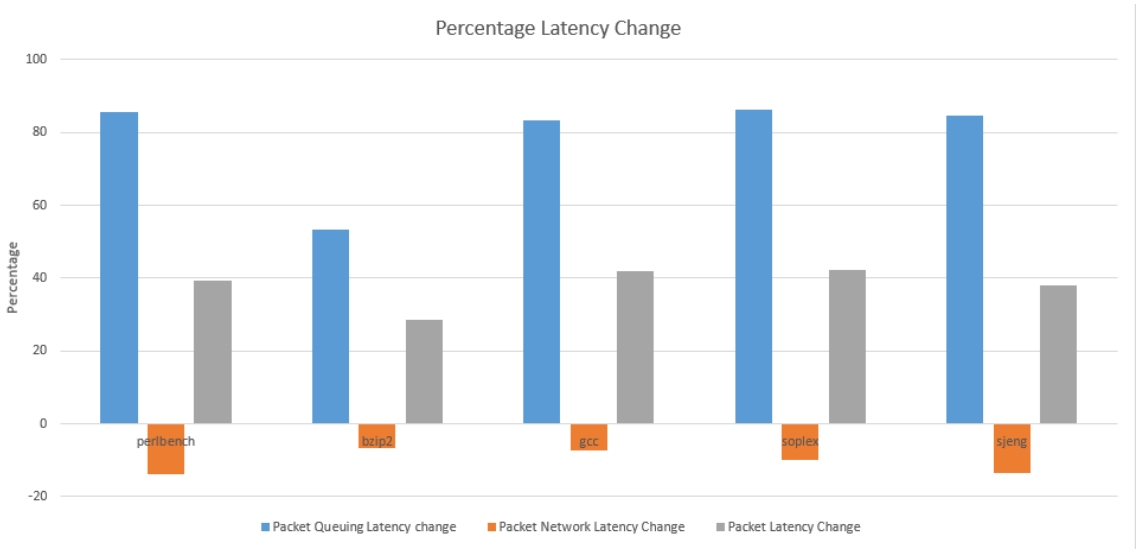

Figure 5.4: Percentage Latency Changes

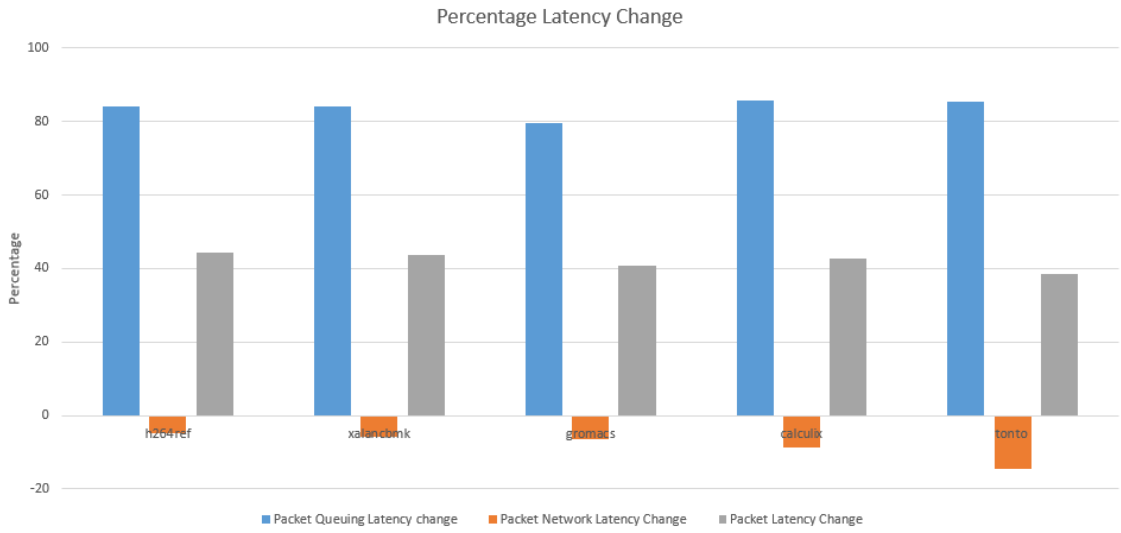

Figure 5.5: Percentage Latency Changes

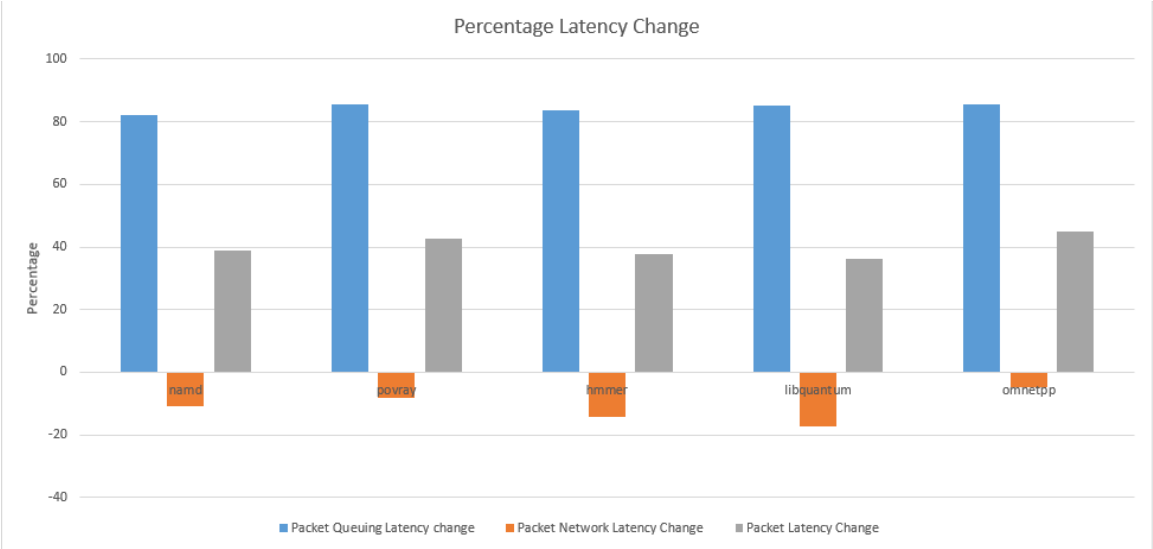

Figure 5.6: Percentage Latency Changes

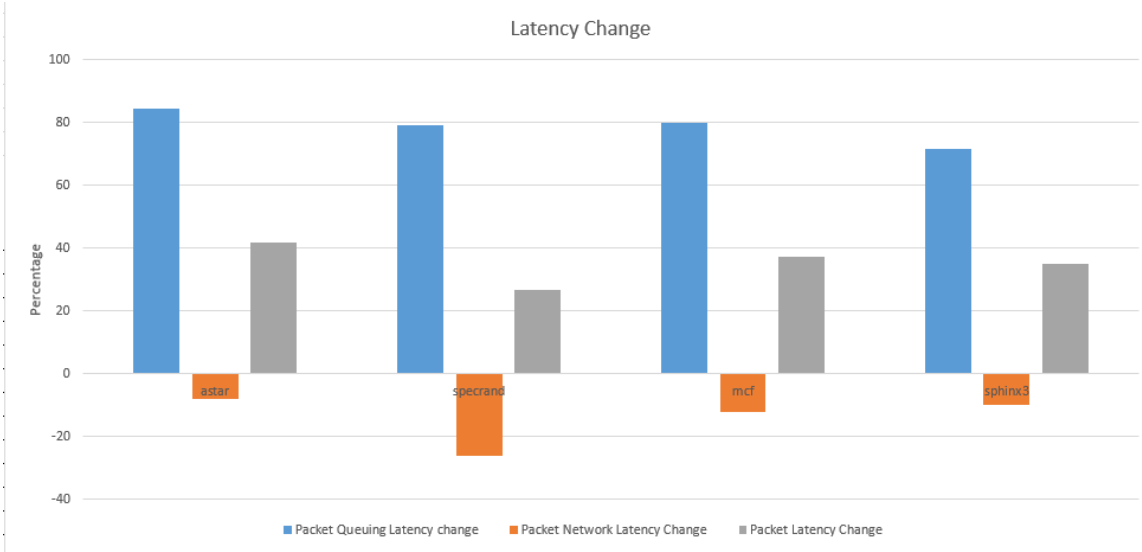

Figure 5.7: Percentage Latency Changes

Figure 5.4, 5.5, 5.6 and 5.7 depict the percentage latency changes. One observation which is unique here is that the latency changes are pretty similar to most of the benchmark runs. This is because, the numbers represented here are average latencies. Average latency is a representation

of the network configuration. For example, any GETS packet, irrespective of which benchmark it is, will undergo network and queuing latency specific to the configuration of the network.

There are couple of observations here:

- **Average Packet Queuing Latency**: The average Queuing latency increases by an average of 81.53% when SNI is enabled compared to when SNI is disabled. One of the reasons for this is the SNIs at the directories which are responsible for converting Broadcast cache coherence messages to ACK response messages. A detailed explanation has been provided later.

- **Average Packet Network Latency**: The average packet network latency decreases by an average of 10.7%. One of the reasons of reduced network latency is due to reduced network traffic. The SNIs at the directories convert a large number of broadcast packets to ACK packets because of which the network traffic reduces. This helps reduce the average network latencies that packets undergo when SNI is enabled.

- **Average Packet Latency**: The total packet latency, i.e. the combination of queuing and network latency, increases by an average of 39.01%.

An interesting observation is that irrespective of the benchmark type and the number of packets injected into the network, the average network and queuing latencies are pretty similar across the benchmark runs. This shows that the average latencies are a reflection of the network configuration of the system.

Now let's take a look at the reason behind the increased queuing latencies. In section 4.1 we explained that SNIs at the directory have the added functionality wherein they scan broadcast type messages and convert them to ACK messages. This conversion is applied if the broadcast messages are being directed towards chiplets which don't have any cores that share a cache block with the original requestor core. The SNI has access to the APU_Table, and thus knows which chiplets share a cache block, and based on that the SNI can modify the broadcast type messages into ACK messages and direct it to the original requestor. This provides the security feature that malicious

chiplets will not be able to scan any of the broadcast type cache coherence messages to snoop on cache lines being accessed by other cores. But the issue with this approach is the traffic burst that is generated from the directory to the original requestor core.

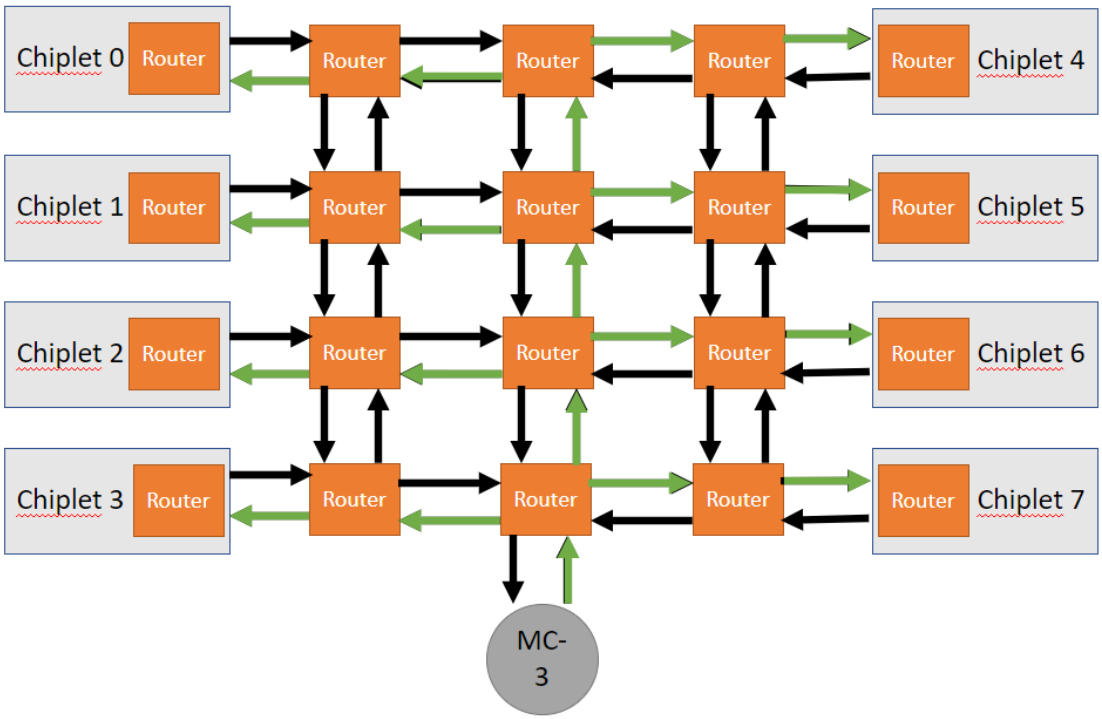

Figure 5.8: Broadcast packet distribution pattern

Figure 5.8 depicts the scenario where after MC-3(directory 3) receives a GETS message from Core 0 in chiplet 0, it then broadcasts the GETS to all the other cores. As we can see the green arrows depict how the packets originate at MC-3 and then reach their destination chiplets by traversing through the network. We can observe that packets are distributed throughout the network and reach their destinations at different latencies from the time they originate. This results in a spaced out load on the virtual network for the broadcast packets.

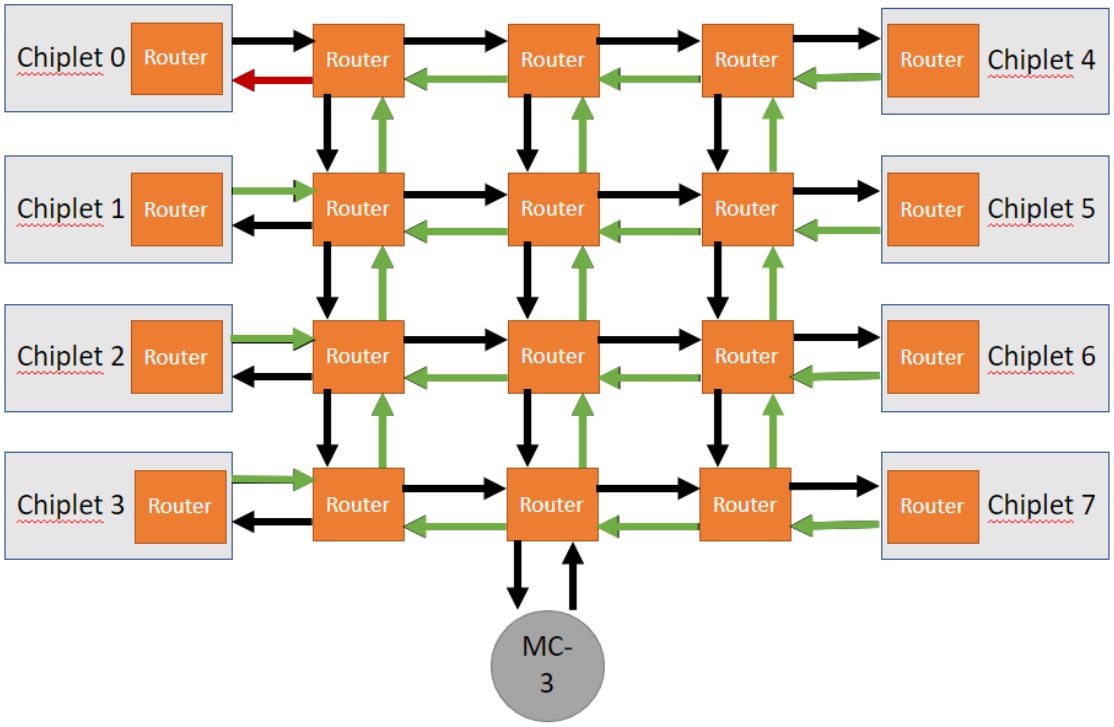

Figure 5.9: ACK response packet distribution pattern

Figure 5.9 depicts the scenario where after each core receives a broadcast GETS, it replies with ACK to Core 0. The cache block is not being shared and thus the 56 cores in rest of the chiplets numbered 1 to 7. Because of the different latencies of broadcast packets reaching each of the cores in the chiplets, the response ACK packets are directed towards the Chiplet 0 but not at the same time. This results in a spaced out load on the virtual network, when ACK response packets are being directed towards Chiplet 0.

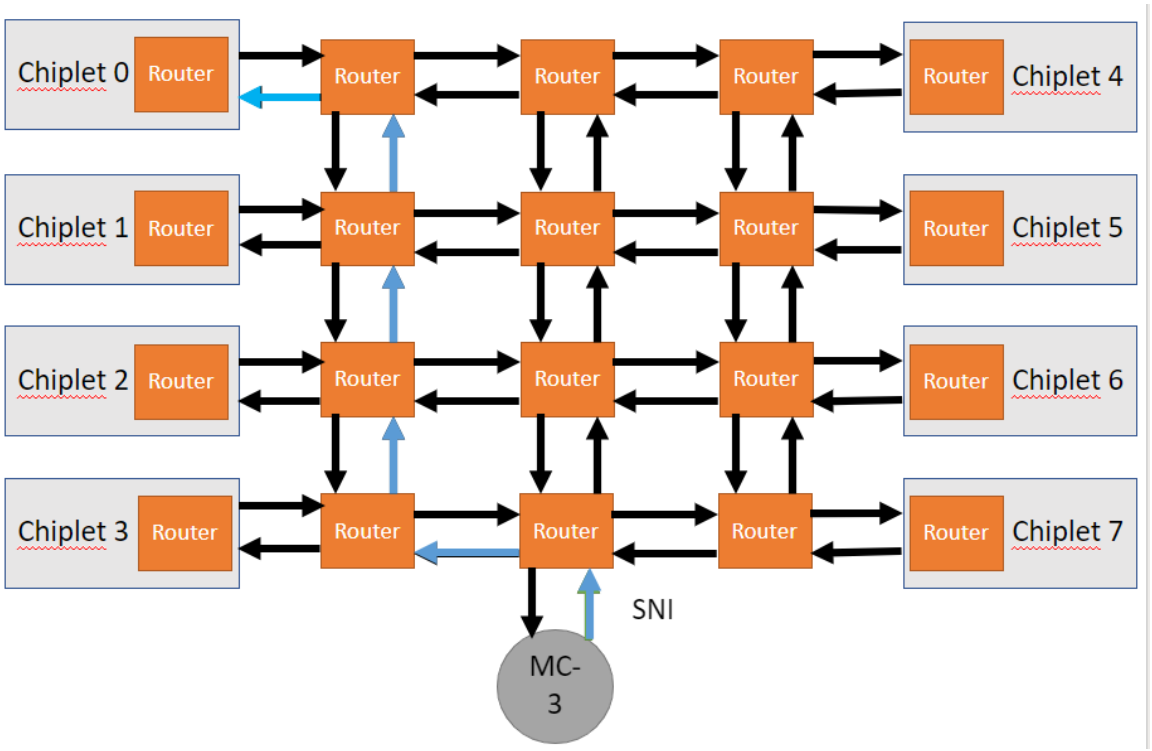

Figure 5.10: SNI at directory enabled response pattern

Figure 5.10 depicts the scenario where the SNI at the directories are enabled. In this scenario, all the 56 broadcast packets supposed to got to the rest of the cores are then converted to ACK response packets and directed towards the Chiplet 0. The blue marked arrows depict the path that is traversed by all of the 56 response ACK packets. In our analysis, it was observed that this burst of traffic, caused significant queuing delays for the ACK responses. And hence, because of this reason we observe the significant increase in the queuing latencies.

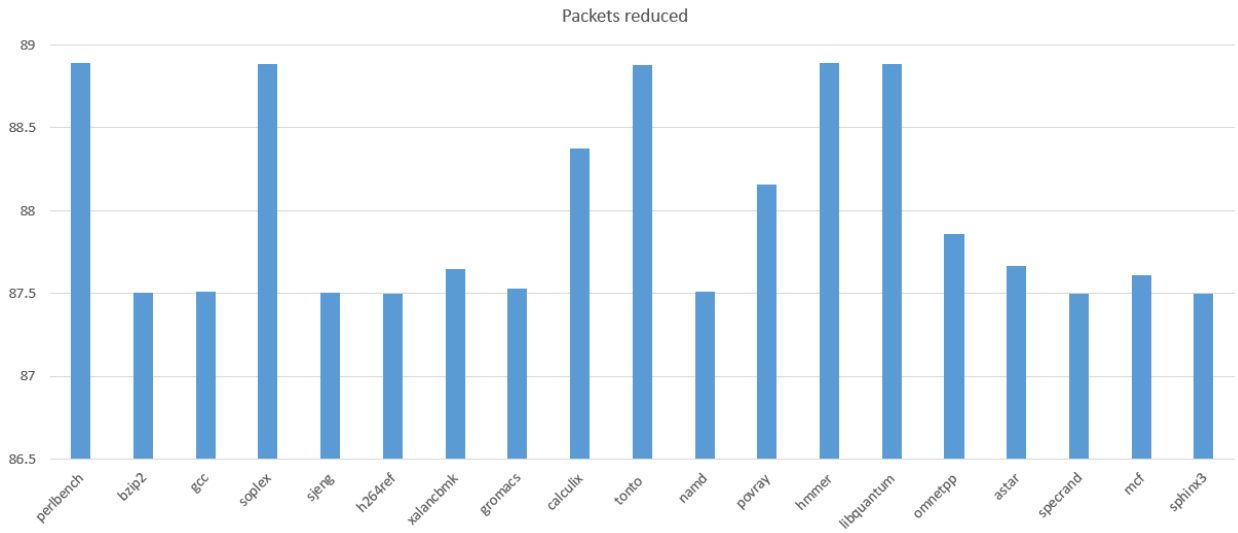

Figure 5.11: Packets reduced with SNIs at directory

Figure 5.11 shows the percentage reduction in packets introduced in virtual network 3. When a directory receives a GETS/GETX message, the corresponding broadcast messages are sent on the VN 3. On comparing the number of packets introduced in VN 3 with and without SNIs, we observed a 87.98% reduction in packets on average. This shows that our initial intention of reducing the packet traffic worked with the SNIs at directory. The only caveat being that the burst of traffic generated in the path from the directory to an original requestor core causes significant increase in the queuing latency.

5.3.4 Dual Benchmark Performance Analysis

We also performed simulation with dual benchmarks, where two benchmarks are being run on two different cores in two different chiplets. The objective being to create more traffic in the network to observe the impact of SNI in such a scenario. We generate random mixes of two benchmarks and run it on two cores in separate chiplets, to observe the impact of SNIs. We run the first 5 billion instructions for each of the applications on each core and they are run simultaneously until each of them reach 5 billion instructions. Here, we report weighted speedup normalized to baseline, i.e. for a multi-programmed run we calculate the IPC of the workload and then also calculate the

isolated IPC of the workload in a single-threaded run, for both SNI enabled and SNI disabled configurations. We then obtain a weighted IPC for the workloads as $\sum(IPC_i/IPC_isolated_i)$ where for an i^{th} workload IPC and IPC_isolated are obtained in multi-program and single-threaded runs respectively. For each of the workload, the sum is then normalized to the weighted IPC calculated similarly for baseline case.

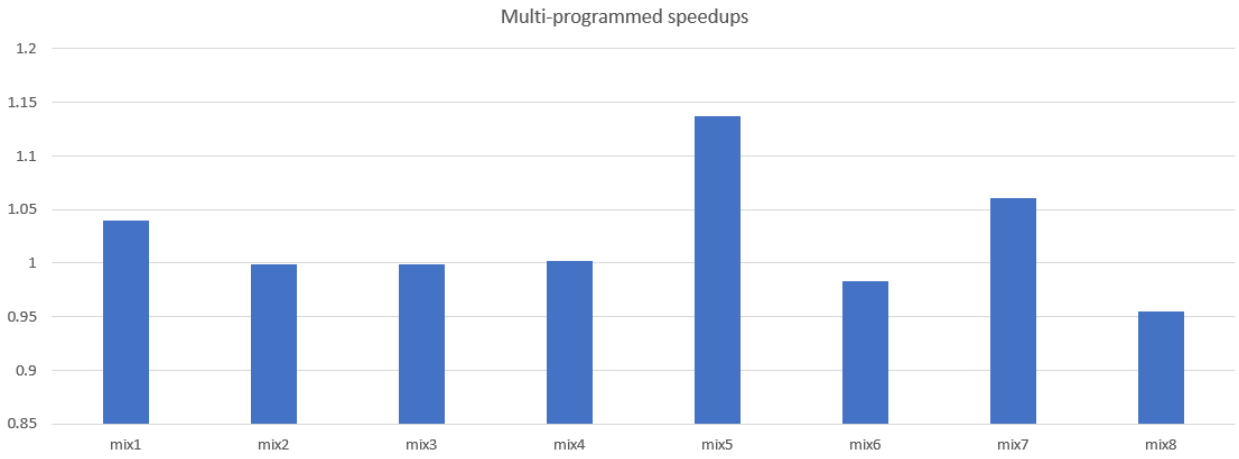

Figure 5.12: Dual benchmark latency changes

Figure 5.12, shows the speedup factor for multi-programmed workloads. We observe that the speedup obtained for these workloads is better than the speedup obtained with the individual speedups obtained by certain workloads. The reason for this observations seems to be the utilization of core for increased processing of instructions per clock cycles. For certain workloads, we observe a speedup greater than 1. It seems to be because of a margin of error or benchmark runs leading to getting slightly better IPC on individual benchmarks when run in multi-programmed mode.

5.3.5 Virtual Channel & Queuing Latency Relation

In this section we present an observation on the relation between the vc_per_vnet configuration and the queuing latency of the packets.

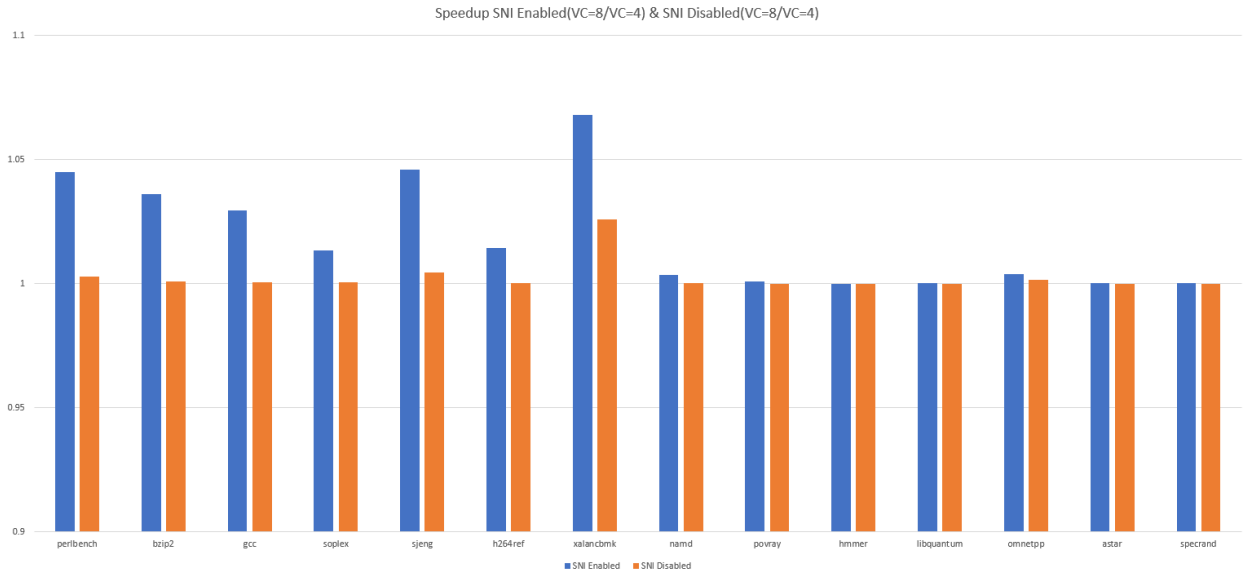

Figure 5.13: Speedup comparison between SNI enabled and SNI disabled configuration

Figure 5.13 shows the speedup factor between SNI enabled and SNI disabled configurations across two configurations of vc_per_vnet. The SNI enabled speedup is obtained as the IPC for VC=4 configuration divided by VC=8 configuration, with SNI enabled in both cases. The SNI disabled speedup is obtained as the IPC for VC=4 configuration divided by VC=8 configuration, with SNI disabled in both cases. An interesting observations is that, with an increase in the number of available virtual channels from four to eight, the speedup obtained in SNI enabled configuration is more than the SNI disabled configuration. This occurs because the queuing latency reduction obtained in SNI enabled configuration is more than SNI disabled configuration. As mentioned earlier, the burst of traffic generated by SNIs at the directories, leading to increased queuing latencies. As the number of available VCs increase, the SNI enabled configuration benefits more in terms of reduction of queuing latencies.

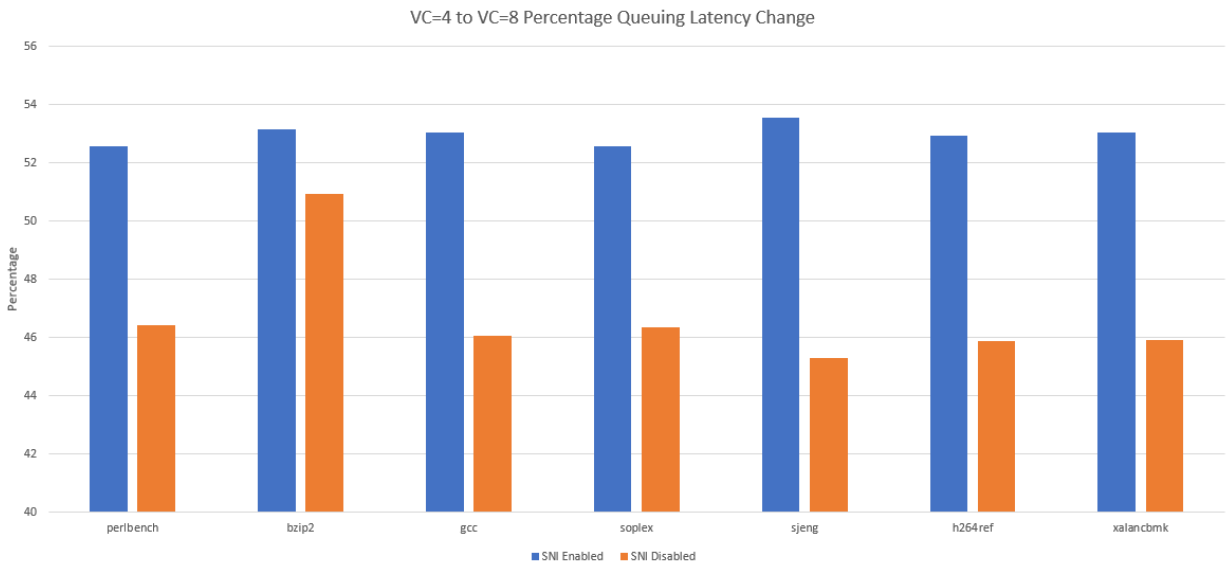

Figure 5.14: Percentage Queuing Latency Change for configurations VC=4 and VC=8

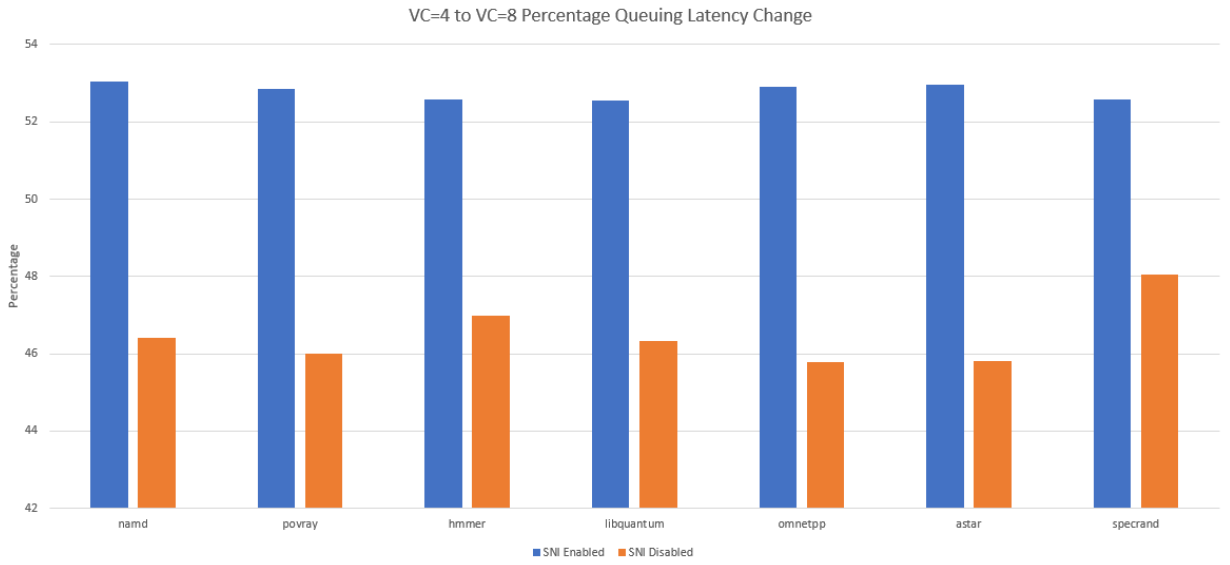

Figure 5.15: Percentage Queuing Latency Change for configurations VC=4 and VC=8

Figure 5.14 and 5.15 show the decrease in queuing latency for SNI enabled and SNI disabled configurations for vc_per_vnet values of 4 and 8. We can observe that as the number of VCs

double, the queuing latency in case of SNI enabled decreases more as compared to that when SNI disabled. This shows that when the SNI is enabled, the burst of traffic that is generated by the SNI at the directories impact the queuing latency of packets to a significant extent as explained earlier. Hence, with availability of more VCs, the SNI enabled configuration benefits more in terms of reduction of queuing latencies compared to the SNI disabled configuration.

6. CONCLUSIONS

6.1 Conclusion

In this work we propose the use of active interposers to implement hardware security features within the interposer layer. Interposers have already gained attraction in modern chip fabrication because of the impact on chip-fabrication costs. The same interposer layer can be used to provide a system wide security interface across different chiplets. Using an interposer as a root-of-trust is an interesting option to interconnect chiplets and simultaneously securing the system.

Through our work, we focus on tackling hardware trojan based attacks from malicious chiplet. We have explored the design details of SNI from an architectural perspective. Unlike previous work which have suggested simple scan of address locations, we have shown that scanning transactions is more complex because of involvement of cache coherency protocol. Messages being exchanged between cores in various chiplets, are in the form of cache coherence packets. These packets are converted into flits. Hence, various parameters of a network, such as flit size, physical link widths, virtual networks, virtual channels etc. determine the design of a SNI. We have also shown statistics through our simulations on the performance impact of SNI over the system. We have compared the IPC changes with SNI enabled and SNI disabled configurations and shown that performance drop due to the latencies introduced by SNIs are reasonable.

We expect that this work can add to the idea of active interposer layer as a "root-of-trust". We have focused on the design aspects of a SNI module which can be implemented in the interposer layer. We have also worked on the idea of how an OS and the SNI will interact in order to dynamically manage the security of the system.

6.2 Future Work

In future work we would like to work on the Operating System and dynamic memory management for more complex applications being run in a multi-chip system. At present our work primarily focuses on the cache coherence aspects of the messages that need to be monitored by

SNI. We have implemented a simplistic model of the OS to simulate a dynamically managed memory. We can work on improving the algorithm for memory management keeping in mind that the APU_Tables have limited number of entries from which memory can be allocated to the applications. In order for lower latencies the interaction between the OS and the SNI/APU_Table has to be streamlined.

REFERENCES

[1] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, "A2: Analog malicious hardware," in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 18–37, 2016.

[2] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan, "Hardware trojan attacks: Threat analysis and countermeasures," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1229–1247, 2014.

[3] M. Matsuo, N. Hayasaka, K. Okumura, E. Hosomi, and C. Takubo, "Silicon interposer technology for high-density package," in *2000 Proceedings. 50th Electronic Components and Technology Conference (Cat. No.00CH37070)*, pp. 1455–1459, 2000.

[4] P. Coudrain, J. Charbonnier, A. Garnier, P. Vivet, R. Vélard, A. Vinci, F. Ponthenier, A. Farcy, R. Segaud, P. Chausse, L. Arnaud, D. Lattard, E. Guthmuller, G. Romano, A. Gueugnot, F. Berger, J. Beltritti, T. Mourier, M. Gottardi, S. Minoret, C. Ribière, G. Romero, P. . Philip, Y. Exbrayat, D. Scevola, D. Campos, M. Argoud, N. Allouti, R. Eleouet, C. Fuguet Tortolero, C. Aumont, D. Dutoit, C. Legalland, J. Michailos, S. Chéramy, and G. Simon, "Active interposer technology for chiplet-based advanced 3d system architectures," in *2019 IEEE 69th Electronic Components and Technology Conference (ECTC)*, pp. 569–578, 2019.

[5] S. Takaya, M. Nagata, A. Sakai, T. Kariya, S. Uchiyama, H. Kobayashi, and H. Ikeda, "A 100gb/s wide i/o with 4096b tsvs through an active silicon interposer with in-place waveform capturing," in *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pp. 434–435, 2013.

[6] M. Nabeel, M. Ashraf, S. Patnaik, V. Soteriou, O. Sinanoglu, and J. Knechtel, "2.5d root of trust: Secure system-level integration of untrusted chiplets," *IEEE Transactions on Computers*, vol. 69, p. 1611–1625, Nov 2020.

[7] H. Mestiri, Y. Salah, and A. A. Baroudi, "A secure network interface for on-chip systems," in *2020 20th International Conference on Sciences and Techniques of Automatic Control and*

Computer Engineering (STA), pp. 90–94, 2020.

[8] A. Zhiyuan and L. Haiyan, "Realization of buffer overflow," in *2010 International Forum on Information Technology and Applications*, vol. 1, pp. 347–349, 2010.

[9] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.

[10] S. K. Moore, "Chiplets are the future of processors: Three advances boost performance, cut costs, and save power," *IEEE Spectrum*, vol. 57, no. 5, pp. 11–12, 2020.

[11] N. Pandit, Z. Kalbarczyk, and R. K. Iyer, "Effectiveness of machine checks for error diagnostics," in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pp. 578–583, 2009.

[12] Y. Wang, P. Chen, J. Hu, G. Li, and J. Rajendran, "The cat and mouse in split manufacturing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 5, pp. 805–817, 2018.

[13] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware trojans," *Computer*, vol. 43, no. 10, pp. 39–46, 2010.

[14] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan detection using ic fingerprinting," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, pp. 296–310, 2007.

[15] M. Abramovici and P. Bradley, "Integrated circuit security: New threats and solutions," in *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, CSIIRW '09, (New York, NY, USA), Association for Computing Machinery, 2009.

[16] Y. Jin, "Design-for-security vs. design-for-testability: A case study on dft chain in cryptographic circuits," in *2014 IEEE Computer Society Annual Symposium on VLSI*, pp. 19–24, 2014.

[17] P. Yang and M. Marek-Sadowska, "Making split-fabrication more secure," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2016.

[18] Y. Wang, P. Chen, J. Hu, and J. J. V. Rajendran, "Routing perturbation for enhanced security in split manufacturing," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 605–510, 2017.

[19] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede, "Hardware-based trusted computing architectures for isolation and attestation," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 361–374, 2018.

[20] Y. M. Qureshi, W. A. Simon, M. Zapater, D. Atienza, and K. Olcoz, "Gem5-x: A gem5-based system level simulation framework to optimize many-core platforms," in *2019 Spring Simulation Conference (SpringSim)*, pp. 1–12, 2019.

[21] Y. Jin, N. Kupp, and Y. Makris, "Experiences in hardware trojan design and implementation," in *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, pp. 50–57, 2009.

[22] A. Basak, S. Bhunia, T. Tkacik, and S. Ray, "Security assurance for system-on-chip designs with untrusted ips," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 7, pp. 1515–1528, 2017.

[23] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Topics in Cryptology – CT-RSA 2006* (D. Pointcheval, ed.), (Berlin, Heidelberg), pp. 1–20, Springer Berlin Heidelberg, 2006.

[24] R. J. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun, "Thermal covert channels on multi-core platforms," in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 865–880, USENIX Association, Aug. 2015.

[25] S. Naffziger, K. Lepak, M. Paraschou, and M. Subramony, "2.2 amd chiplet architecture for high-performance server and desktop products," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 44–45, 2020.

[26] J. Kim, J. Balfour, and W. Dally, "Flattened butterfly topology for on-chip networks," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pp. 172–182, 2007.

[27] N. D. E. Jerger, T. Krishna, and L. Peh, *On-Chip Networks, Second Edition*. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2017.

[28] M. Katevenis, "Buffer requirements of credit-based flow control when a minimum draining rate is guaranteed," in *The Fourth IEEE Workshop on High-Performance Communication Systems*, pp. 168–178, 1997.

[29] W. J. Dally, "Virtual-channel flow control," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 194–205, 1992.

[30] A. Ros, M. E. Acacio, and J. M. Garcia, "Cache coherence protocols for many-core cmps," in *Parallel and Distributed Computing* (A. Ros, ed.), ch. 6, Rijeka: IntechOpen, 2010.

[31] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The amd opteron processor for multiprocessor servers," *IEEE Micro*, vol. 23, p. 66–76, Mar. 2003.

[32] B. Jacob and T. Mudge, "Virtual memory in contemporary microprocessors," *IEEE Micro*, vol. 18, no. 4, pp. 60–75, 1998.

[33] J. Kim, G. Murali, H. Park, E. Qin, H. Kwon, V. C. K. Chekuri, N. Dasari, A. Singh, M. Lee, H. M. Torun, M. Swaminathan, M. Swaminathan, S. Mukhopadhyay, T. Krishna, and S. K. Lim, "Architecture, chip, and package co-design flow for 2.5d ic design enabling heterogeneous ip reuse," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2019.

[34] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera,

M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," 2020.

[35] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.