PROCESSOR MEMORY SYSTEM DESIGN FOR PERFORMANCE AND SECURITY

A Dissertation

by

GINO AUGUSTO CHACON

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

DOCTOR OF PHILOSOPHY

Chair of Committee,	Paul V. Gratz
Committee Members,	Daniel Jiménez
	Jeyavijayan Rajendran
	Ulisses Braga-Neto
Head of Department,	Miroslav Begovic

May 2023

Major Subject: Computer Engineering

ABSTRACT

The benefits of Moore's Law are waning and effects of Dennard Scaling are ending, resulting in modern computing designs improving performance through new and creative designs. These designs target modern performance barriers but introduce new threat vectors that bad actors may exploit to compromise the design and manufacturing process.

Server and database computing system performance can improve by increasing machines' ability to facilitate fast instruction memory accesses to overcome the front-end bottleneck, generally through prefetching. Prefetching is a technique to predict future instruction accesses and place them in the caches before use. In this dissertation, we propose a framework for combining pre-existing hardware prefetchers into a single composite prefetcher to target different instruction stream behaviors during execution.

Software prefetching has recently resurfaced as an alternative to hardware prefetching. While promising, recent proposals do not model industry-standard front-ends in their evaluation. In this dissertation we identify the potential states the front-end can be in and software instruction prefetching's effect on the front-end that degrades performance.

Industry is moving toward chiplet-based designs, which divide systems into chiplets and integrate them onto an interposer via 2.5D integration. This design flow disaggregates the manufacturing and design process between multiple vendors with varying trustworthiness, increasing hardware Trojans' potential insertion and threat. These systems rely on cache coherence for data communication, making coherence an attractive target. Trojan attacks exploiting coherence can modify data in memory that the compromised chiplet never touched or owned. Further, the Trojan need not be physically between the victim and the memory controller to attack a victim's memory transactions. This dissertation explores the fundamental coherence attack vectors possible in chiplet-based systems. Further, we provide an example Trojan implementation capable of directly modifying victim data in memory without disrupting system execution.

To counter coherence threats, we propose a defense mechanism leveraging an active interposer

to produce a generic, secure-by-construction platform forming a physical root of trust for 2.5D systems. The scheme has limited overhead, restricted to the active interposer, allowing the chiplets and the coherence system to remain unmodified. This scheme prevents coherence attacks with little impact on system performance, $\sim$4%, which reduces as workloads increase, ensuring scalability.

DEDICATION

To my family and friends that offered unending support.

ACKNOWLEDGMENTS

First and foremost, I wish to express my gratitude to my advisor, Paul V. Gratz. He has provided endless support and advice along my Ph.D. journey. As an eager undergraduate looking for research, Paul allowed me to work on exciting problems alongside great researchers and people. This work was only possible with his insight and encouragement to explore and tackle challenging problems, even if it did not always work out. I also thank Daniel Jimènez for his advice and encouragement throughout the years. Furthermore, I would like to thank my committee for their feedback: Ulisses Braga-Neto, J. V. Rajendran, and Krishnendra Nathella.

I also thank my former and present colleagues at the Computer Architecture, Memory Systems and Interconnection Networks (CAMSIN) and Texas Architecture and Compiler Optimization (TACO) research groups. Eric Garfinkle, Luke McHale, Elvira Teran, Nathan Gober, Jinchun Kim, Charles Williams Jr., and Elba Garza. It has been a wonderful experience working with all of you and fighting through deadlines, bugs, and lockdowns. Special thanks to my collaborators in industry and abroad: Chris Wilkerson (Intel), Alaa Alameldeen (Simon Fraiser University), Seth Pugsley (Intel), and Krishnendra Nathella (Microsoft).

Finally, I would like to thank my family and friends, who have been overwhelmingly supportive during my time as a student. Without their kindness, understanding, and willingness to listen to me rant about broken simulations for hours, I would not be in this position. I'd also like to thank Becka for her seemingly endless patience when dealing with a stressed-out engineer. Thank you to Fabian, Larissa, and Russel for giving me some social life in lockdown and some of the best backyard barbequing I've had.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Paul V. Gratz, Professor Jeyavijayan Rajendran, and Professor Ulisses Braga-Neto of the Department of Electrical and Computer Engineering and Professor Daniel A. Jimenéz of the Department of Computer Science and Engineering and external committee member Krishnendra Nathella.

Chapter II was collaborated with Elba Garza and Daniel Jimenéz of the Department of Computer Science and Engineering, Paul V. Gratz of the Department of Electrical and Computer Engineering, Alberto Ros and Alexandra Jimborean of the Computer Engineering Department at University of Murcia, and Samira Mirbagher-Ajorpaz of the Department of Electrical and Computer Engineering at North Carolina State University. Chapter III was collaborated with Nathan Gober of the Department of Electrical and Computer Engineering, Daniel Jimenéz of the Department of Computer Science and Engineering, and Krishnendra Nathella while he was employed at ARM Inc. Chapter IV was collaborated with Tapojyoti Mandal of NVIDIA while he was a graduate student of the Department of Electrical and Computer Engineering, and Johann Knechtel and Ozgur Sinanoglu of the Division of Engineering at New York Unversity Abu Dhabi, and Vassos Soteriou of the Department of Electrical and Computer Engineering at Cyprus University of Technology. Chapter V was collaborated with Charles Williams and Paul V. Gratz of the Department of Electrical and Computing Engineering, and Johann Knechtel and Ozgur Sinangolu of the Division of Engineering at New York University Abu Dhabi.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by a College of Engineering fellowship from Texas A&M University, by the Michael W. Powell Electrical Engineering Graduate Fellowship, by the National Science Foundation which supports this work through grant FoMR-1823403, and a generous gift

from Intel.

NOMENCLATURE

APU	Address Protection Unit
AsmDB	Assembly Database
BER	Backward Error Recovery
BTB	Branch Target Buffer
CFG	Control Flow Graph
CMC	Coherence Message Checker
CVP	Championship Value Prediction
ECC	Error Correction Codes
EIP	Entangling Instruction Prefetcher
FDP	Fetch Directed Prefetching
FER	Forward Error Recovery
FNL	Footprint Next Line
FTQ	Fetch Target Queue
GHR	Global Histroy Buffer
IPC	Instructions Per Cycle
L1-I	First Level Instruction Cache
L2C	Second Level Cache
LBR	Last Branch Record
LLC	Last-Level Cache
LRU	Least Recently Used
MC	Memory Controller
MOESI	Modified-Owner-Exclusive-Shared-Invalid

MPKI	Misses-per-Thousand-Instructions
MSHR	Miss Status Handling Register
NoC	Network on Chip
NI	Network Interface
PCM	Packet Checker/Modifier
PFC	Post-Fetch Correction
PC	Program Counter
RAS	Return Address Stack
RMT	Redundant Multithreading
SPFB	Subprefetcher Buffer
TAP	Temporal Ancestry Prefetcher
TEE	Trusted Execution Environment
SWIP	Software Instruction Prefetch
TMR	Triple Modular Redundancy

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Moore's Law has slowed in recent years while the demand for computing continues to increase. The commercial demand for computing leads to more creative design solutions. Memory system performance and operations heavily impact the performance and security of a system. Creative design solutions improve emerging designs' cost and manufacturing efficiency but must address specific memory performance and security concerns. In this section, we provide an introduction to the performance and security challenges associated with modern systems.

First, we provide an overview of the performance challenges associated with modern memory system design, explicitly concerning instruction memory. We then discuss emerging systems based on 2.5D integration technology, which divides designs into chiplets from various vendors of various levels of trust that could potentially insert hardware Trojans targeting these systems' communication fabric (i.e., coherence protocol). Following this, we provide the goal and organization of this dissertation.

1.1 Memory System Performance Challenges

An ongoing challenge in modern systems is the "Memory Wall" [3], which results from the disparity between the processor and memory speeds. This speed gap presents a bottleneck for processors at the memory level. To alleviate this bottleneck, modern processors incorporate large memory hierarchies with various capacities and speeds to maintain frequently accessed memory in faster memory caches closer to the processor and prevent memory requests from being serviced by slower main memory devices. However, the Memory Wall continues to pose a challenge in modern systems despite the prevalence of deep, fast cache hierarchies. In particular, it forms a bottleneck in the delivery and execution of instructions by a processor's front-end. In the following sections, we introduce the Memory Wall's effect on the front-end, or fetch stage, of modern processors and provide background on two forms of instruction prefetching, which attempt to bring instructions into the fastest cache level before the front-end demands them.

1.1.1 Front-End Bottleneck

The Memory Wall [3] manifests in the superscalar core front-end as a lack of instructions available for fetch, as the front-end's resources are encumbered by waiting for the memory hierarchy to service instruction requests [4, 5, 6]. Recent work demonstrates that the front-end has become a significant bottleneck for modern server workloads due to the availability of instructions [7, 8]. This bottleneck results from increased instruction memory footprint in server workloads with ever-deeper software stacks where even simple user requests traverse multiple software layers, touching megabytes of code in the process [9]. Large footprints further exacerbate the disparity between processor and memory performance as the L1 instruction (L1-I) cache of modern processors cannot capture an application's behavior resulting in significant critical-path stalls.

Recent studies have found that large footprints are endemic to modern server workloads and will only grow more significant over time—at a rate of 20% per year—as software stacks deepen to accommodate more complex applications [10]. This overwhelming use of front-end resources is a well-known phenomenon dubbed the *front-end bottleneck* [9]. The front-end bottleneck challenges hardware architects as programs' long-term behaviors can no longer be fully mapped in the L1-I cache nor captured by branch prediction structures [11]. Thus, microarchitects must limit the effects of the front-end bottleneck while keeping to strict timing, area, and power constraints.

Instruction prefetching has been a solution to memory bottlenecks for decades, with a plethora of prior art in instruction prefetching. Instruction prefetching techniques have been proposed at the L1 instruction cache level, as a form of speculative run ahead based on existing prediction structures in the front-end, and at the software level based on some profile of a target application. We further elaborate on each of these instruction prefetching techniques below.

1.1.1.1 Hardware Instruction Prefetching

Prefetching has been heavily studied to alleviate data and instruction memory bottlenecks. A large body of work has emerged to explore different mechanisms to improve the instruction memory performance to address this performance bottleneck. Prefetching a technique that can

effectively mask memory access latencies by speculating on future memory reference streams, to reduce costly cache misses [12, 13]. Prefetching is a viable mechanism for reducing misses in both instruction and data caches. Prefetching requires predicting upcoming memory accesses and issuing preemptive memory requests before explicit requests by memory are necessary [14, 12, 15, 16, 17].

Most instruction accesses are sequential as a program is iterated through, making next-line prefetchers fairly effective [18]. However, discontinuities exist as control-flow instructions. Hardware instruction prefetcher designs commonly use existing control-flow related microarchitectural structures to provide a context to the application's current behavior and prefetch along a speculative execution path.

We find that, due to their independent implementations, recently proposed instruction prefetchers [19, 20, 21, 22, 23, 24, 25, 26] often behave differently dependent on workload and program phase, leading to different prefetch suggestions at different times, and thus they vary in terms of timeliness, coverage, and accuracy. Interestingly, we also find that as the hardware budget of each prefetcher is reduced to make the prefetcher more practical in a production environment, they behave more distinctively and complementarily. This finding argues for the benefits of leveraging multiple prefetchers *in combination*, particularly when implemented in reasonable, buildable hardware budgets. In Chapter II we will propose a methodology for selecting prexisting hardware instruction prefetchers with complementary behavior to combine into composite prefetcher capable of outperforming its subcomponents even at a lower hardware budget.

1.1.1.2 Decoupled Front-Ends

Fetch-Directed Prefetching (FDP) [27] is an important form of instruction prefetching, heavily used in current, aggressive, superscalar processor cores [1]. FDP speculates using the branch prediction structures to aggressively fill the Fetch Target Queue (FTQ) ahead of the instruction stream with the predicted execution path. Entries in the FTQ are sent to the L1-I as soon as their addresses are known to fill the cache ahead of the demand request. The larger the FTQ, the more and earlier requests can be issued to the L1-I, increasing instruction throughput. An important

aspect of FDP is that it allows a decoupling of the front-end from the rest of the processor, allowing the front-end to run as far ahead as the branch predictor allows by holding multiple outstanding instruction requests to the L1-I in the FTQ [28, 29, 30, 31]. We discuss decoupled front-ends in more detail in Chapter III.

1.1.1.3 Software Instruction Prefetching

To avoid the higher overheads and constraints of learning instruction stream access patterns at the L1-I, software instruction prefetchers are also a potential solution to alleviate the front-end bottleneck. Software instruction prefetchers perform some of profiling of an application's instruction stream behavior to build a model of the instruction stream behavior.

Recent works in alleviating the front-end bottleneck have proposed using software prefetching techniques to profile a workload's behavior and identify regions or accesses with a high impact on performance [32, 33, 34, 35, 36], prefetching these high-impact memory accesses ahead of time. Profiling ahead of execution allows for the hardware to have prior knowledge of what accesses are compulsory or have long-term reuse without obvious short-term locality. These techniques do not require hardware overhead other than an ISA implementation of a prefetching instruction. Software instruction prefetching is an attractive solution as it can improve production binaries that can be modified and updated. Recent work in software instruction prefetching demonstrates high potential in implementing schemes such as the state-of-the-art AsmDB [32] prefetcher, seeing benefits as high as $\sim$15 % performance improvement.

Despite the potential benefit of software instruction prefetching, we note that prior work in this area evaluates their proposals in systems without FDP or a very conservative FDP implementation. Recent work emphasizes the importance of modeling a realistic FDP implementation to accurately evaluate a proposal [2]. A conservative implementation of FDP limits the ability of the prefetcher to runahead of the instruction stream, inflating the reported benefit of an L1-I prefetcher.

In Chapter III we evaluate a modern software instruction prefetching technique in an industry-standard front-end, finding that it causes a degradation in performance. We characterize the front-end's behavior and identify the effects of inserting new instructions into the instruction stream by

a software instruction prefetcher that results in the degradation.

1.2 Security Challenges in Cache Coherent 2.5D Chiplet Environments

A recent trend in computing systems is the adoption of hardware organization based on chiplets and interposers [37, 38, 39, 40]. Instead of implementing a monolithic system-on-chip (SoC), this approach disaggregates the functional components across multiple smaller chips, i.e., *chiplets*, which are designed and manufactured separately. These chiplets serve as hard intellectual property (IP) modules, possibly sourced from a variety of vendors, and consolidated on an integration and interconnects carrier, i.e., the *interposer* [37, 38, 39, 40, 41, 42]. This approach is also known as 2.5D integration.

The adoption of chiplet and interposer integration raises design reuse to the level of the physical system, optimizing yields and streamlining time to market, resulting in significant cost benefits. Such 2.5D integration is already adopted by industry in products such as the AMD Epyc processors [39, 40] or the Intel Embedded Multi-Die Interconnect Bridge technology [43]. Recent industry talks herald this design style as the next iteration of Moore's law [44]. While such 2.5D designs provide many benefits, they also increase the exposure risk to Trojan attacks, explicitly targeting the coherence scheme of the overall system.

1.2.0.1 Cache Coherence as a Vulnerable Subsystem

Coherence is an essential mechanism that ensures all components maintain a consistent view of the system's memory, not only for interposer-based systems but interconnected SoCs in general. Coherence protocols ensure updates to cached copies of data are visible to all cores and other IP blocks in modern multi-core designs [45, 37, 40]. Coherence schemes can be broadly categorized as broadcast (or snooping) protocols [46, 47, 48] and directory protocols [49, 50, 51]. While simple to implement, broadcast protocols suffer from high traffic due to the amount of messages multi-core systems require to maintain coherence. Directory protocols allow for fine-grained state tracking and unicast messages, making them highly scalable but difficult to implement and have higher access latencies.

Given that coherence protocols act only based on rules for how memory is updated across multiple parties, attackers may exploit the protocol's low-level behavior. Coherence is a hardware-managed, micro-architectural feature which is neither influenced by, nor exposed to, the software executing on the system, rendering software-based defenses ineffective.

1.2.0.2 Hardware Trojans

Hardware Trojans, or Trojans for short, are a hardware-centric threat in which an attacker infiltrates some level of the design or fabrication process to insert malicious circuitry into a design. Trojans can cause disastrous system failures via confidentiality, integrity, and/or availability violations. Prior work has shown that Trojans can leak data from memory [52], disrupt cryptographic security features [53], and induce denial-of-service attacks [54]. Trojans can be difficult to detect; an ongoing area of research is to create hardened systems resilient to Trojan effects.

While the industry moves towards 2.5D designs, specific IPs used in building these systems may not be trustworthy. Even when considering an IP vendor as trustworthy, the manufacturing processes involved are open to infiltration and the insertion of Trojans. For chiplet designs, memory coherence is a crucial design point, as large designs require each component to maintain an up-to-date view of the system's memory. Thus, coherence systems are an attractive target for Trojans. Despite their attractiveness, there has yet to be a deeper exploration of coherence exploits beyond instigating a system deadlock [54].

In Chapter IV, we explore the threat vectors available to a bad actor attempting to exploit the cache coherence mechanisms in 2.5D chiplet environments. Furthermore we use the fundamental attacks available to a hardware Trojan designer as stages within a more complex hardware Trojan attack capable of modifying memory that the hardware Trojan's compromised chiplet would not have access to during execution.

In Chapter V, we propose using the active interposer as the root of trust due to its properties as the underlying interconnect between chiplets. We assume that the active interposer can be fabricated at a trusted and controlled facility, not only guarenteeing that the active interposer is uncompromised but that any security features embedded within it are also uncompromised by

a hardware Trojan. Based on this assumption, we propose a security monitoring system which verifies coherence messages traversing the network on chip against the memory regions chiplets are permitted to access and verify the integrity of coherence messages.

1.3 Dissertation Statement

In this dissertation, we explore hardware instruction prefetching as a solution to the front-end bottleneck and characterize its behavior for future software instruction prefetcher designs. We then identify the threat of hardware Trojans to coherent 2.5D chiplet-based designs and propose countermeasures against these attacks.

First, we propose a composite prefetching framework to identify state-of-the-art instruction prefetchers with different, but complementary, prefetching behaviors to combine into a better performing prefetcher. Each subprefetcher present in the composite prefetcher takes an equal portion of the hardware budget and allows for each prefetcher to operate in a *black-box* to reduce the design complexity overhead and permit each prefetcher to cover cache misses that its counterparts are unable to identify. We then identify that previously proposed software instruction prefetching solutions do not evaluate their techniques in decoupled environments standard in industry. Our evaluation shows that the evaluated technique, AsmDB [32], degrades performance in a realistic front-end model. We perform a characterization of the state the front-end can be in at a given time and identify the sources of degredation due to the software instruction prefetcher introducing additional instructions into the instruction stream.

Following this, we recognize that industry is moving towards a disagregated design flow based on 2.5D integrated designs that source multiple hard IPs as chiplets that are then integrated through an interposer medium. We identify this design flow as a potential threat vector for bad actors to insert hardware Trojans that may violate the confidentiality, integrity, and avaialability of the entire system by targeting the vulnerable cache coherence subsystem. We propose fundamental coherence-oriented attacks a hardware Trojan may mount and further emphasize this threat by using the fundamental attacks to build a complex attack capable of accessing and modify memory the compromised chiplet would otherwise be unable to access. Once we have identified the potential

threats hardware Trojans pose to coherence subsystems in 2.5D chiplet environments, we propose leveraging the active interposers as the root of trust to embed security features to harden and secure the coherence communication within the system.

1.4 Dissertation Organization

In the following chapters, we demonstrate, evaluate, and characterize our work through a range of experiments. Chapter II proposes a methodology and analysis to select state-of-the-art instruction prefetchers with complementary prefetching behavior to compose a composite prefetcher. In Chapter III we perform a characterization of the software instruction prefetching technique in ChampSim with a modified front-end representative of an industry-standard frontend. Chapter IV explores the types of attacks a hardware Trojan can mount in chiplet-based environments if it compromises and exploits the coherence subsystem, specifically demonstrating the ability of a hardware Trojan to modify data its compromised device does not normally have access to. Chapter V addresses the threat of coherence-targeting hardware Trojans by establishing the chiplet system's active interposer as the root of trust to embed security monitors to enforce correct and secure coherence communications.

2. SELECTING AND EVALUATING INSTRUCTION PREFETCHERS WITH COMPLEMENTARY BEHAVIOR FOR THE DESIGN OF A COMPOSITE INSTRUCTION PREFETCHER [1]

This chapter presents a methodology for combining multiple instruction prefetchers that capture different instruction stream behaviors into a single composite prefetcher capable of identifying instruction stream behavior better than its subcomponents. We first introduce a background of modern hardware instruction prefetching and recently proposed instruction prefetchers' behavior. We then present the design of a composite prefetcher composed of multiple prefetchers. Following this, we discuss the methodology for selecting orthogonal subprefetchers capable of complementing one another's instruction prefetching behavior. The evaluation of a composite prefetcher, and our selection methodology, compares the performance of the best-performing combinations of instruction prefetchers to their individual performance to demonstrate that a composite prefetcher composed of prefetchers outperforms its subcomponents at the same amount of metadata overhead.

2.1 Introduction

Due to their independent implementations, we find that recently proposed instruction prefetchers [19, 20, 21, 22, 23, 24, 25, 26] often behave differently dependent on workload and program phase, leading to different prefetch suggestions at different times. Thus they vary in terms of timeliness, coverage, and accuracy. Interestingly, as we reduce the hardware budget of each prefetcher to make the prefetcher more practical in a production environment, they behave more distinctively and complementarily. This finding argues for the benefits of leveraging multiple prefetchers *in combination*, specifically when implemented in reasonable, buildable hardware budgets.

While prior work has meticulously integrated simple prefetchers to create composite prefetchers for both instruction [55] and data [17, 56], such component-based composite prefetchers do not

[1] Reprinted with permission from "Composite Instruction Prefetching," G. Chacon, E. Garza, A. Jimborean, A. Ros, P. V. Gratz, D. A. Jiménez, and S. Mirbagher-Ajorpaz. 2022 IEEE 40th International Conference on Computer Design (ICCD), Olympic Valley, CA, USA, 2022, pp. 471-478, doi: 10.1109/ICCD56317.2022.00076.

allow for *interchangeability of components*. Each prefetcher must be tuned extensively to ensure each component captures specific application behaviors. This tuning requires in-depth knowledge of the prefetching behavior of each component and an idea of how they may complement each other regarding coverage and precision. These works also require a mechanism to identify whether recent accesses fit a particular pattern attributed to a specific component, increasing the likelihood of the prefetcher misidentifying the application's behavior. By contrast, this work seeks to coordinate and interchange multiple complex instruction prefetchers, allowing for broader coverage to overcome the timing constraints of traditional composite prefetchers.

Based on the increasing instruction footprint size of modern server workloads and the success of individual prefetchers, we propose a new approach to composite prefetching that *requires no knowledge of the component prefetchers* and allows for interchangeable components to enable prefetching design space exploration. Our approach integrates preexisting complex prefetchers as components within a single composite prefetcher. Each prefetcher design may capture different instruction streams, and their combined prefetch behavior results in higher coverage and performance than an individual prefetcher at an equivalent size. Aside from varying the metadata storage required by each prefetcher, each component is considered a black box. This approach mitigates the burden of creating tailor-made components to capture specific application behavior when creating a composite prefetcher. As a practical design methodology, our approach would enable industry to easily interchange any desired prefetcher from the academic literature and study them in combination without spending valuable design time evaluating each component's advantages or shortcomings. In summary, our contributions in this chapter are as follows:

- We characterize prior work, a selection of complex instruction prefetchers, to understand their complementary nature and composability at differing sizes.
- We study the feasibility of combining a set of state-of-the-art hardware prefetchers to better capture instruction stream behavior.
- We identify a simple hybridization scheme for combining existing prefetchers to leverage

their complementary behavior. Our scheme can integrate multiple complex prefetchers with no prior knowledge of function.

- We perform a full design-space exploration for the set of hardware prefetchers to identify their best performing combinations at various hardware budgets.

- Using our scheme, we demonstrate that a combination of multiple state-of-the-art prefetchers can outperform its components at the same hardware budget.

2.2 Background and Motivation

This section provides a background on the component prefetchers we consider for generating composite prefetchers, and our motivation for exploring composite prefetching as a solution to the front-end bottleneck.

2.2.1 Modern Instruction Prefetchers

Recently proposed instruction prefetchers speculate on future references using varying underlying mechanisms. Due to their differences, they are more or less efficient at predicting future references for particular workloads and program phases. By combining these existing prefetchers, we can create a single unified prefetcher better than the sum of its parts. Here we examine several recently proposed prefetchers and classify them based on function.

Each class of prefetcher operates based on specific principles surrounding an application's behavior to predict future instructions. Recently proposed prefetchers can be classified based on how they train, represent instruction stream behavior, and select prefetch candidates. This is similar to recent classifications of data prefetchers [57]. Recent work we consider for our composite prefetchers falls into the following classes of prefetchers:

2.2.1.1 Control-Flow-Graph Recreation

These prefetchers recreate an application's control-flow graph (CFG). Nodes represent basic blocks within a graph, with edges representing control-flow (branch) instructions. The prefetcher uses the address of L1-I accesses to find a starting point to traverse the CFG to find prefetch

candidates. Confidence is generally assigned based on observations of the control flow to indicate the application's likelihood to take a particular execution path.

Barça: The Branch Agnostic Region Searching Algorithm, or Barça [21], creates a control-flow graph to map regions of instruction blocks and their relative control flow.

PIPS: Prefetching Instructions with Probabilistic Scouts, or PIPS [26] recreates an application's CFG using a Line-History Table to connect cache lines recently accessed together, tracking the probability of traversing a particular edge.

2.2.1.2 Temporal Prefetchers

These prefetchers attempt to predict the future instruction stream by identifying accesses that cause cache misses, recording the following misses, and replaying them when a triggering access is seen. This style of prefetcher emphasizes timeliness by prefetching misses in an instruction stream well before the front-end requests them. We classify the following prefetchers as temporal prefetchers:

EIP: The Entangling Instruction Prefetcher [19, 58] "entangles" instructions together to provide prefetch timelines, accounting for the prefetch latency to identify the suitable instruction to trigger the prefetch.

FNL-MMA: FNL-MMA [22] combines a Footprint Next Line Prefetcher (FNL) to predict the "not so distant" future, while the Multiple Miss Ahead Predictor (MMA) takes advantage of predictable cache miss sequences.

TAP: The Temporal Ancestry Prefetcher [23] augments a next-line prefetcher with temporal-based histories leading to a program counter (PC) based on the observation that most cache lines are not rereferenced once they fill the cache.

MANA: MANA [24] creates spatial regions in a set-associative table, tracking a triggering address and a footprint to indicate which blocks within a region are accessed. MANA traverses its table when prefetching, loading a stream address buffer with prefetch candidates.

2.2.1.3 Branch-Oriented Prefetchers

Unlike CFG-based prefetchers, these prefetchers do not recreate the CFG but rather use branch-related information to make predictions about future accesses and cover branch targets. This information includes the branch-target-buffer, the return-address-stack (RAS), or other branch structures. We use the following branch-oriented prefetchers in our work:

D-JOLT: D-JOLT [20] consists of multiple simple prefetchers of varying characteristics. It uses a long-range prefetcher to cover the distant future with higher coverage, a short-range prefetcher to cover the near future with higher accuracy, and a "fall-back" prefetcher.

JIP: JIP [25] is composed of multiple prefetchers that target specific instruction stream behavior, such as sequential accesses within basic-blocks, branches with a single target, and branches to multiple targets.

2.2.2 Complementary Prefetchers

While some prefetchers have high performance at lower hardware budgets, they cannot leverage more storage to achieve higher performance. Each prefetcher's performance varies and operates on different principles, which can be broadly classified (Sec. 2.4.2), but their classification does not predict their performance at various hardware budgets. Figure 2.1 illustrates this by showing the overlap of unique addresses targeted by each prefetcher at sizes of 10KB and 128KB. Ideally, complementary prefetchers have lower overlap to facilitate different instruction stream behaviors. Lower hardware budgets limit the misses each prefetcher learns and targets, resulting in a low overlap between most prefetchers except for FNL+MMA and JIP, PIPS, and TAP. At higher storage budgets, the prefetchers are less constrained, capture more misses, and thus converge towards similar behavior. *This indicates that at different hardware budgets, a combination of prefetchers have vastly different prefetching behavior and potentially capture different instruction streams.* However, if prefetch streams are too dissimilar, there is a potential for destructive interference between the prefetchers as they could each aggressively prefetch different instruction streams, resulting in a high amount of thrashing in the already encumbered L1-I. To this end, we propose a

composite prefetching framework and methodology for searching for the best-performing combination of prefetchers at various hardware budgets.

2.2.3 Composite Prefetching

Prior work in composite prefetching has been mainly in the data prefetching domain, with little work applying it to instruction prefetching. Division of Labor, or DOL [17], attempts to exploit both simple and complex access patterns using a collaboration of specialized subcomponents for each pattern. DOL is extendable with additional components as more access patterns are identified. Note that the hardware designer must identify missing or necessary access patterns, making DOL limited by the designer's knowledge.

Bouquet of Instruction Pointers [56] creates a composite L1 data prefetcher that uses a "bouquet" of pointers to classify instruction pointers and issue data requests based on the classification. This technique covers and identifies a handful of memory access patterns that drive prefetches.

The above works focus on data prefetching that relates specific instructions to data they access. While instruction prefetching and data prefetching are similar, as they attempt to hide access latencies, their access patterns and relationships to data diverge. Instruction prefetchers target instructions themselves, causing control flow to be an important factor. Divide and Conquer Frontend Bottleneck [55] warns against BTB-directed instruction prefetches, presenting the "harmful effects" of making instruction prefetchers dependent on BTB content. Instead, it proposes dividing the front-end bottleneck into a sequential prefetcher to cover sequential misses, a discontinuity prefetcher, and pre-decoding prefetch blocks to reduce BTB misses. This divide-and-conquer method has the same area overhead as a BTB-directed prefetcher but outperforms it by 5% on average for their selected workloads.

As seen by the works described above, the concept of combining prefetchers, both in data and instruction prefetching, is not novel in itself. However, these component prefetchers are *non-interchangeable* and tuned for hardware size and prefetch specialty by the designer, requiring in-depth knowledge of each component. In contrast, our proposition requires no knowledge of the prefetcher components and allows for previously unexplored component interchangeability.

	Barca	D-JOLT	FNL-MMA	EIP	JIP	PIPS	TAP	Mana
Barca	100.0%	33.5%	34.2%	34.1%	34.2%	34.2%	34.2%	34.2%
D-JOLT	33.5%	100.0%	42.9%	42.4%	42.9%	42.9%	42.8%	42.7%
FNL-MMA	34.2%	42.9%	100.0%	48.6%	69.2%	68.8%	66.8%	51.5%
EIP	34.1%	42.4%	48.6%	100.0%	48.6%	48.6%	48.6%	47.5%
JIP	34.2%	42.9%	69.2%	48.6%	100.0%	74.4%	78.5%	51.6%
PIPS	34.2%	42.9%	68.8%	48.6%	74.4%	100.0%	71.7%	51.6%
TAP	34.2%	42.8%	66.8%	48.6%	78.5%	71.7%	100.0%	51.4%
Mana	34.2%	42.7%	51.5%	47.5%	51.6%	51.6%	51.4%	100.0%

(a) Percent overlap of unique prefetch targets between two prefetchers sized at roughly 10KB each for a subset of CVP traces.

	Barca	D-JOLT	FNL-MMA	EIP	JIP	PIPS	TAP	Mana
Barca	100.0%	62.7%	62.7%	62.6%	63.2%	63.2%	63.0%	62.6%
D-JOLT	62.7%	100.0%	66.9%	68.1%	71.0%	70.9%	69.9%	69.1%
FNL-MMA	62.7%	66.9%	100.0%	67.1%	68.8%	68.8%	67.9%	67.4%
EIP	62.6%	68.1%	67.1%	100.0%	71.7%	71.7%	70.3%	70.0%
JIP	63.2%	71.0%	68.8%	71.7%	100.0%	99.6%	77.8%	77.2%
PIPS	63.2%	70.9%	68.8%	71.7%	99.6%	100.0%	77.8%	77.2%
TAP	63.0%	69.9%	67.9%	70.3%	77.8%	77.8%	100.0%	73.3%
Mana	62.6%	69.1%	67.4%	70.0%	77.2%	77.2%	73.3%	100.0%

(b) Percent overlap of unique prefetch targets between two prefetchers sized at roughly 128KB each for a subset of CVP traces.

Figure 2.1: The overlap between individual prefetchers at the smallest (10KB) and largest (128KB) hardware budgets. The higher the percentage, the more the two prefetchers overlap.

2.3 Design and Implementation

This section describes our proposed design of a composite prefetcher, consisting of two or more prefetchers. We begin by describing the hardware framework that enables the integration of multiple prefetchers. We then discuss how the composite prefetcher generates and issues prefetch candidates to the L1-I.

2.3.1 Composite Prefetcher Organization

A key design goal of our composite prefetcher is that it should be comparable to a single prefetcher using the same hardware budget. Thus, we scale down the budget used in the various prefetchers described in Section 2.2 to use multiple prefetchers with a comparable total budget of

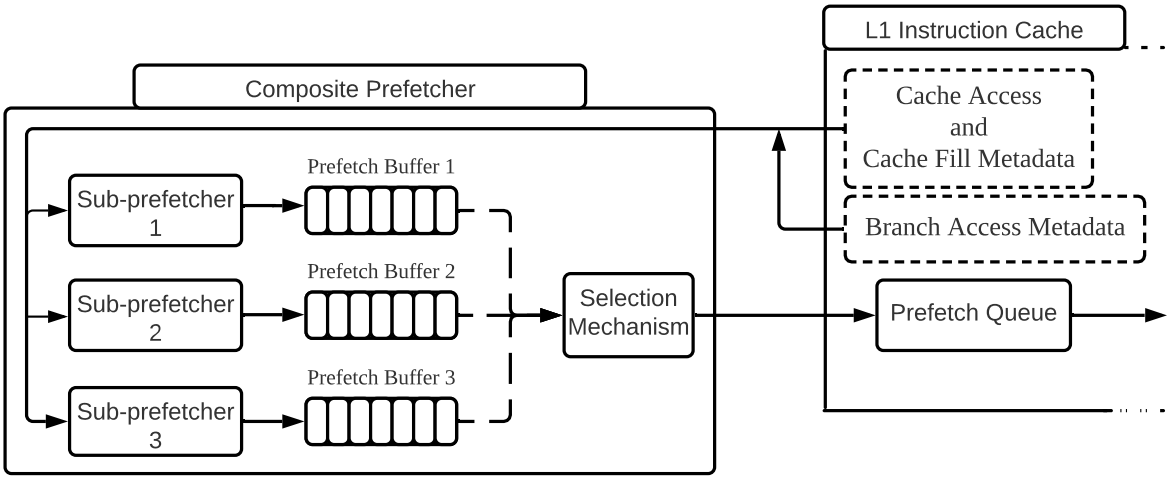


Figure 2.2: Overview of the composite prefetcher's organization.

a single prefetcher. For instance, integrating two 10KB prefetchers will have comparable hardware overhead as a single 20KB prefetcher. A goal when integrating prefetcher components is for their individual operation to remain intact. Thus, other than modifying their structures to reduce their metadata state storage, we make no further changes to any prefetcher component.

In this chapter, we explore integrating mixes of two to four prefetchers. Figure 2.2 illustrates the hardware framework for prefetchers. Each complex prefetcher component, or ***subprefetcher***, receives metadata from the L1-I cache regarding cache accesses, misses, and fills. Following the subprefetchers, a ***subprefetcher buffer (SPFB)*** is available to each prefetcher that is the same size as the L1-I prefetch queue allowing the prefetchers to operate as they would individually without being affected by each other's prefetch queue's bandwidth pressure. Finally, a selection mechanism is placed at the head of the buffers to select a prefetcher's buffer based on a selection policy. Beyond the state required per prefetcher, this buffering and selection mechanism requires 32 entries per SPFB to hold 56-bit prefetch cache line addresses (assuming 64B cache lines), resulting in a 224B overhead per subprefetcher. Further discussion on the operation of the composite prefetcher is provided below in Section 2.3.3.

An essential property of a composite prefetcher organization is that although the subprefetchers

are technically not aware of each other when run in tandem, they adapt to each other's behavior through the misses that are covered versus uncovered during execution. Since most prefetchers are trained only on cache misses, once one prefetcher learns to cover a given miss, other subprefetchers can quickly adapt to disregard the metadata state needed to cover that miss since it is no longer considered a miss from their perspective. As we will show, this is desirable since this allows subprefetchers to use their limited storage to focus on the misses not covered by the accompanying subprefetchers.

2.3.2 L1-I Cache Metadata and Subprefetcher Training

We provide each prefetcher information on cache demand accesses, prefetch hits, branch information and results, and the effects of cache fills, such as the filling cache address and the victim cache line's address. Each prefetcher is treated as a "black box" with regards to the metadata it receives and is not tuned to cover specific instruction stream behavior, as opposed to prior composite prefetcher work [55, 17, 56]. Each subprefetcher trains based on their individual training policy and may disregard any provided information.

2.3.3 Composite Prefetcher Operation

Each prefetcher generates a set of prefetch candidates on a cache access that it places in the SPFBs. Each cycle following generation, the prefetch selection mechanism transfers the head of a particular SPFB into the L1-I cache's prefetch queue using a round-robin selection mechanism. We explored other selection mechanisms but found that round-robin was sufficient because much of the time, only one prefetcher is filling its SPFB while the other SPFBs are empty. If the prefetch queue is full or there is no available Miss Status Handling Register (MSHR), the selection mechanism does not continue to move prefetches into the queue. When generating a new stream of prefetches, the head of the SPFB is set to the first free buffer entry, and new prefetches fill the buffer. Newly generated prefetches overwrite the current contents of the SPFB if it is full, removing stale prefetches from the previous generation that could pollute the L1-I.

2.4 Evaluation

This section describes the design space exploration and evaluation of a composite prefetcher. We begin by describing our simulation environment and evaluation methodology. Next, we evaluate composite prefetching schemes composed of two, three, and four subprefetchers and the design space surrounding subprefetcher selection at different hardware budgets. We then describe the evaluation of individual subprefetchers' contribution to performance. Finally, we discuss the best performing combination of subprefetchers at hardware budgets of 20KB, 30KB, 40KB, 64KB, and 128KB.

2.4.1 Methodology

We perform design-space exploration and evaluation of the composite-prefetcher framework using the ChampSim simulator [59]. Featuring an aggressive front-end similar to Fetch-Directed Prefetching [60, 61] and models a Branch Target Buffer (BTB) that includes an indirect BTB and return address stack. We configure the simulator to reflect recent Intel's Sunny Cove microarchitecture with the parameters in Table 2.1.

For our evaluation, we employ a subset of the traces from the 1st Championship Value Prediction (CVP-1) [62], provided by Qualcomm Datacenter Technologies and ported to the ChampSim format. We select CVP traces that show at least one MPKI (miss per kilo-instruction) at the L1-I and L2C in our baseline configuration and demonstrated high performance potential beyond a next-line L1-I prefetcher regarding the maximum performance as measured by an Oracle L1-I prefetcher. The selected CVP traces demonstrate MPKIs ranging from 3 to 48 at the L1-I cache. All benchmarks maintain low MPKIs in the L1-D, indicating that L1-I miss limits these workloads' performance. As many recent works point to the instruction cache misses becoming more critical in cloud and server workloads, we chose this subset to represent emerging, high instruction cache pressure applications. Each benchmark shown is executed for 50M instructions to warm up the predictors and caches, with another 50M instructions executed to measure performance. Despite the short simulation lengths, we emphasize that these traces demonstrate high L1-I miss rates

Table 2.1: Simulated Baseline System Configuration

Processor Configuration	
Clock Frequency	4GHz
Fetch Queue	64 entries
Decode Queue	32 entries
Dispatch Queue	32 entries
Reorder Buffer	352 entries
Load Queue	128 entries
Store Queue	72 entries
Fetch width	6 instructions
Decode width	6 instructions
Dispatch width	6 instructions
Memory Configurations	
L1 I-Cache	32KB, 8 ways, 64 sets, no prefetcher
L1 D-Cache	48KB, 12 ways, 64 sets, next line prefetcher
L2 Cache	512KB, 8 ways, 1024 sets, spp
LLC Cache	2MB, 16 ways, 2048 sets

Table 2.2: Best performing prefetcher combinations for Composite-2 for hardware budgets of 20KB, 30KB, 40KB, 64KB, and 128KB divided evenly between subprefetchers.

Hardware Budget	Subprefetcher-1	Subprefetcher-2
20KB	Barça	FNL+MMA
30KB	FNL+MMA	MANA
40KB	FNL+MMA	EIP-ISCA
64KB	D-JOLT	FNL+MMA
128KB	FNL+MMA	EIP-ISCA

analogous to the high miss rates seen in modern data center workloads.

2.4.2 Hardware Constraints and Instruction Prefetcher Performance

Existing academic instruction prefetchers have significant coverage when their metadata storage state is unconstrained. However, when implemented with more realistic amounts of storage than industrial designs expect, they struggle. Figure 2.3 shows the individual performance of the prefetchers described in Section 2.2.1 as the storage budget for the prefetcher increases. We compare each prefetcher's performance against a baseline system with no instruction prefetching, an

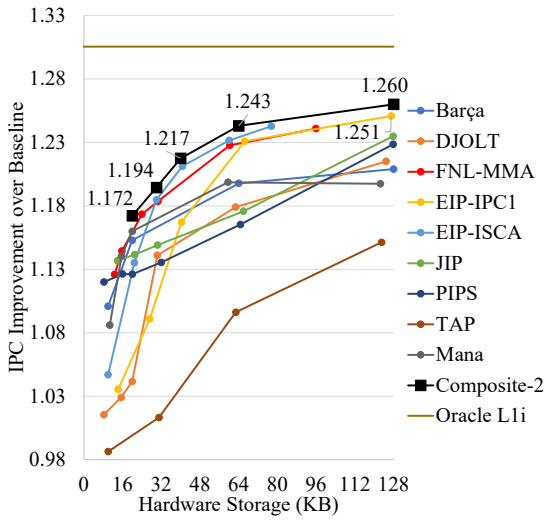


Figure 2.3: Prefetcher performance versus hardware budget, including the best performing combined prefetcher at each hardware budget. At each budget, the composite prefetcher outperforms not only its subprefetcher components but the best performing single-prefetcher.

SPP [63] data prefetcher in the L2 cache, and a least-recently-used (LRU) replacement policy used in all cache levels.

We include an Oracle instruction prefetcher that covers all non-compulsory L1-I misses. The Oracle also fills the unified L2 and L3 caches to mimic the data interference an L1-I prefetch stream would have. The oracle represents a reasonably tight upper bound on the attainable performance from instruction prefetching. As Figure 2.3 shows, at the maximum hardware budget evaluated, EIP comes within 5% of Oracle's performance. As expected, all prefetchers suffer lower performance benefits at lower hardware storage budgets and enjoy increased performance benefits as the hardware budget increases. Performance tends to fall significantly at hardware budgets of 64KB and less, and several prefetchers take turns showing the best performance at different points in the space. In general, we observe three specific trends in Figure 2.3 based on the prefetchers' performance at various hardware budgets: low-budget friendly, budget-sensitive, or budget indifferent.

2.4.2.1 Budget Sensitive Prefetchers

These prefetchers (i.e., EIP, D-JOLT, and TAP) experience heavy performance degradation at lower hardware budgets, with performance substantially increasing with the hardware budget. In

particular, EIP sees ~20% performance improvement from increasing the hardware budget from 15KB to 128KB. These prefetchers are ideal for high-budget designs but may not be reasonable for smaller designs.

2.4.2.2 Low-Budget Friendly Prefetchers

These prefetchers experience a drop in performance at lower hardware budgets while still exhibiting high performance gains from increased hardware budgets. Barça, FNL+MMA, JIP, and PIPS follow this trend, seeing moderate performance benefits at sizes of 10-30KB and scaling as the hardware budget increases. Though their improvements from increasing hardware budgets are not as drastic as EIP, these prefetchers provide consistent performance benefits as their design scales, indicating they are viable options for low-budget *and* high-budget designs.

2.4.2.3 Budget Indifferent Prefetchers

This trend is observed when prefetchers perform well at lower hardware budgets, but only experience modest benefits from an increase in hardware budget. MANA follows this trend, being less affected by a lower hardware budget of 10-15KB. This prefetcher is resilient to lower storage but does not benefit significantly from an increased hardware budget, seeing only a 5% performance benefits from an increased budget of 15KB to 128KB, making it better suited for low hardware budget designs with consistent performance as the prefetcher's hardware budget scales.

2.4.3 Selecting Composite Prefetcher Subprefetchers

The design space of exploring an N-composite scheme, with the possibility of 8 different prefetchers filling any one slot within an N-composite scheme, with prefetchers of various hardware budgets to meet an overall hardware budget is exceedingly large. Given 8 possible subprefetchers of various sizes, the design space increases exponentially as the overall hardware budget increases. For our experiments, each subprefetcher's size is determined by the overall hardware budget divided evenly between the subprefetchers. For example, a 30KB composite prefetcher may be composed of two 15KB prefetchers or three 10KB prefetchers. The results of Figure 2.3 direct the design exploration for the composite design based on a prefetcher's performance relative to its

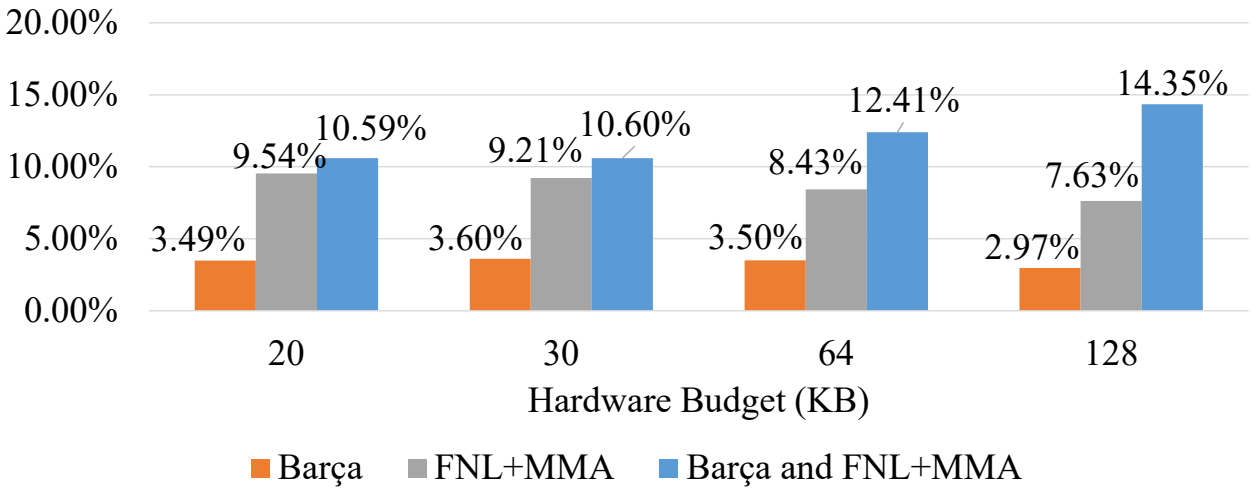


Figure 2.4: Miss coverage, in a Composite-2 of Barça & FNL+MMA covered by Barça, FNL+MMA or both at various hardware budgets

hardware budget.

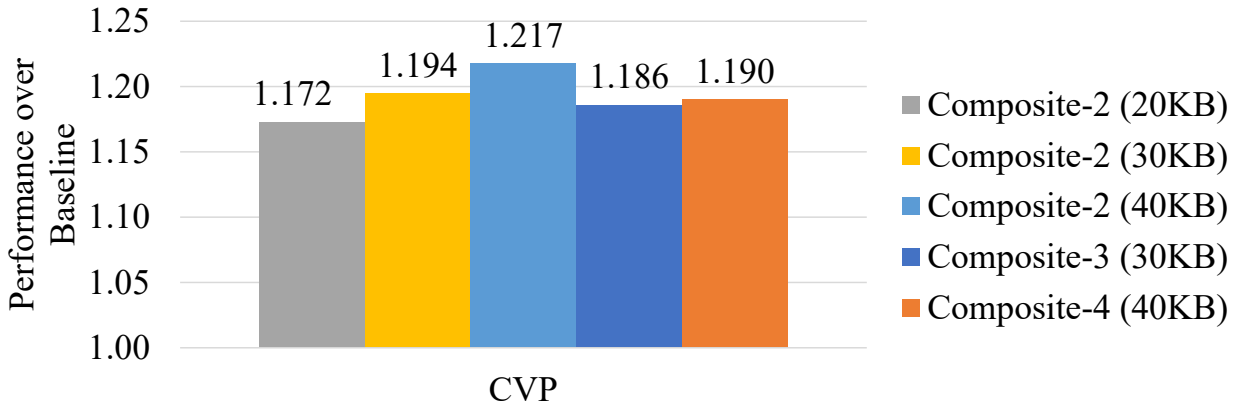


Figure 2.5: Performance comparison of best composites of 2, 3 and 4 prefetchers for different metadata storage state.

We perform a design space exploration for each hardware budget to build composite prefetchers composed of two, three, and four subprefetchers. Our design space exploration combines N of the 8 possible prefetchers and then evaluates the resultant composite prefetcher. Each N-composite prefetcher has a range of performance results, with the best-performing combination varying based

on the hardware budget. Figure 2.5 shows the performance of the best-performing composite of 2, 3, and 4 prefetchers at 20KB, 30KB, and 40KB.

Here, the hardware budget of the composites is divided evenly between each subprefetcher, i.e. for Composite-2 at 20KB, each subprefetcher has roughly 10KB of state. In the figure, we see that composites of 2 significantly outperform composites of 3 and composites of 4 prefetchers. Even the smaller 20KB Composite-2 prefetcher approaches the performance of the Composite-3 and Composite-4 prefetchers within ~2%. The allocated budget per prefetcher is reduced as the number of prefetchers increases to keep the overall budget within limits. Thus, the effectiveness of each prefetcher diminishes. We conclude that composing a high number of prefetchers with a reasonable budget is not promising. As a result, we focus on Composite-2 prefetchers for the remainder of this chapter.

Table 2.2 lists the subprefetchers from the best performing composite-2 prefetcher found in our design space exploration at each storage size. Interestingly, the best performing composite changes significantly at each hardware budget.

2.4.4 Full Results Comparison

Figure 2.3 shows the results of all prefetchers examined, scaled to different sizes, along with the best performing Composite-2 prefetcher combination discussed in Section 2.4.3 and listed in Table 2.2. In general, the Composite-2 prefetcher outperforms all single prefetchers at every metadata storage size, often by significant margins. Composite-2's performance increases as its subprefetchers' can capture more of the workloads' behavior but see the highest performance benefits at low hardware budgets.

Figure 2.4 shows the misses that Barça and FNL+MMA cover individually and the overlap in their access coverage when combined at varying hardware budgets in a Composite-2 prefetcher. As Composite-2 is scaled, the coverage overlap between the two prefetchers increases. Interestingly, for small budgets, each subprefetcher covers a more significant fraction of misses than both cover together. This result illustrates that at smaller budgets, each prefetcher tends to focus on misses it is better able to cover, yielding greater metadata storage efficiency than can be achieved by a single

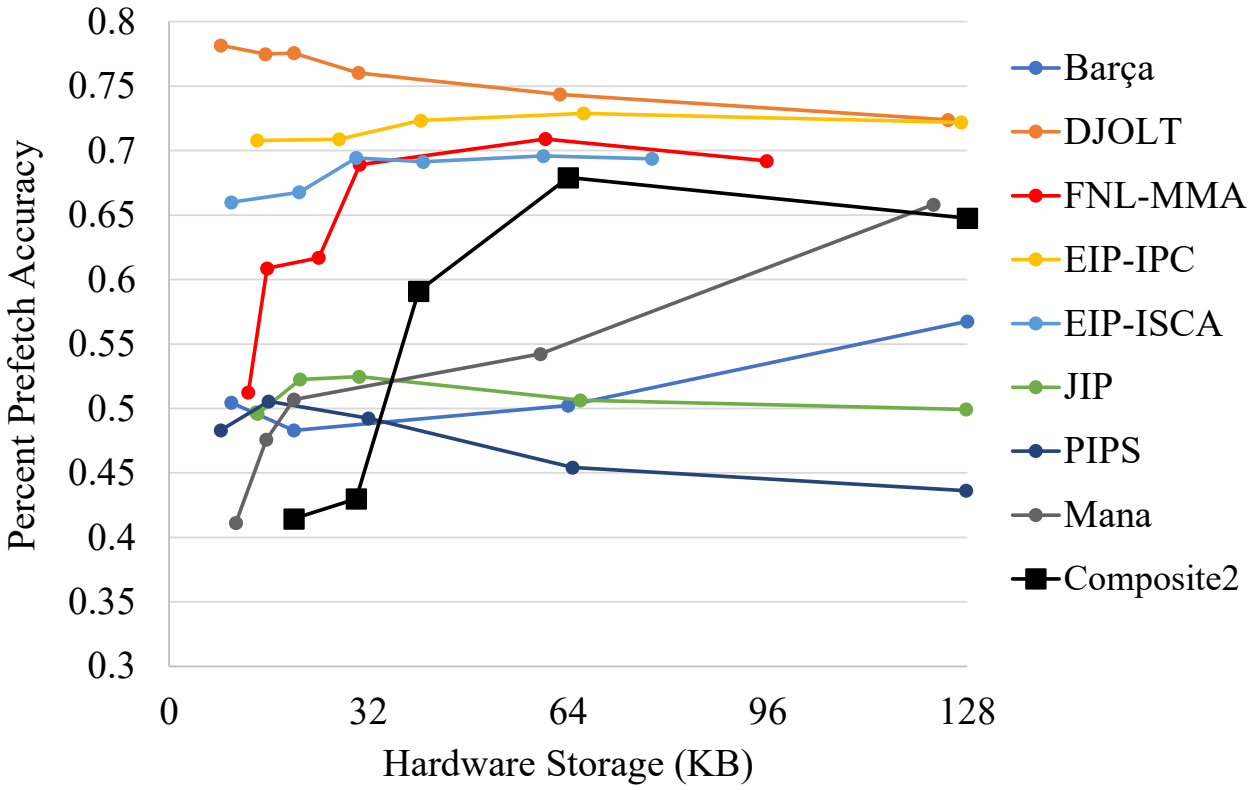


Figure 2.6: Accuracy vs. Hardware Storage (KB) at each metadata storage point for all prefetchers and Composite-2.

prefetcher at even double the size.

2.4.5 Subprefetcher Behavior

2.4.5.1 Accuracy and Issued Prefetches

We evaluate the accuracy of a composite-2 prefetcher compared to the potential subprefetcher components in Figure 2.6. Composite-2's accuracy depends on its component subprefetchers' behavior, with its accuracy increasing as the hardware budget increases. Each prefetcher, except DJOLT, does not train off cache accesses that hit on a prefetched line, resulting in each prefetcher training on misses not covered by other prefetchers. Maintaining a view of only cache misses increases the orthogonality between prefetchers' predictions, as discussed in section 2.4.5.2 while decreasing overall accuracy.

The round-robin selection mechanism considers each subprefetcher's prefetch stream. A more

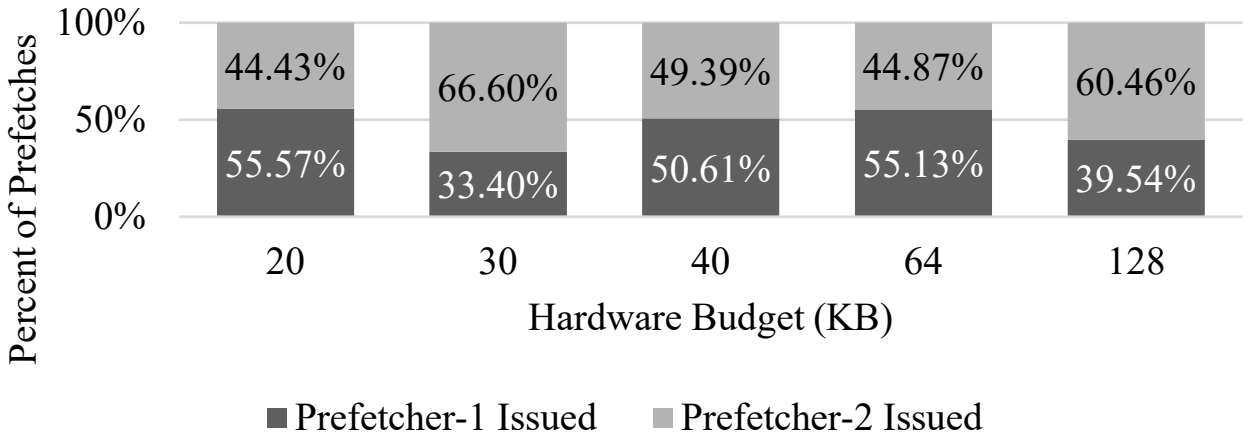


Figure 2.7: Percentage of prefetches issued from each component prefetcher in the best performing Composite-2 prefetcher at each storage overhead. Generally, one subprefetcher does not tend to dominate the prefetches Composite-2 produces.

complex selection mechanism can be implemented to prioritize prefetches from a particular subprefetcher. However, the selection mechanism's impact relies on multiple subprefetchers generating predictions simultaneously. Our evaluation finds that only one subprefetcher generates prefetches for 80% of demand cache accesses at any hardware budget, indicating complex selection mechanisms are unlikely to identify prefetching opportunities that would benefit from prioritizing one subprefetcher over another. Figure 2.7 shows the prefetches issued by each subprefetcher in Composite-2. Overall, no subprefetcher dominates the prefetch contributions, allowing each subprefetcher to provide complementary prefetch candidates.

2.4.5.2 Measuring Subprefetchers' Individual Contributions

Miss: A miss occurs for both prefetchers and in the baseline cache. This is generally a small percentage of accesses (1-2%) for any size of composite-2, indicating that the prefetchers or the baseline cache without prefetching cover most accesses.

PF1 Hit (PF1): This is the number of misses unique prefetches from subprefetcher-1 cover.

PF2 Hit (PF2): This represents the number of misses covered by subprefetcher-2.

Baseline Hit (Base): A hit occurs only for the baseline, without prefetching, indicating that the subprefetchers' behaviors cause a harmful eviction resulting in a cache miss. This scenario occurs

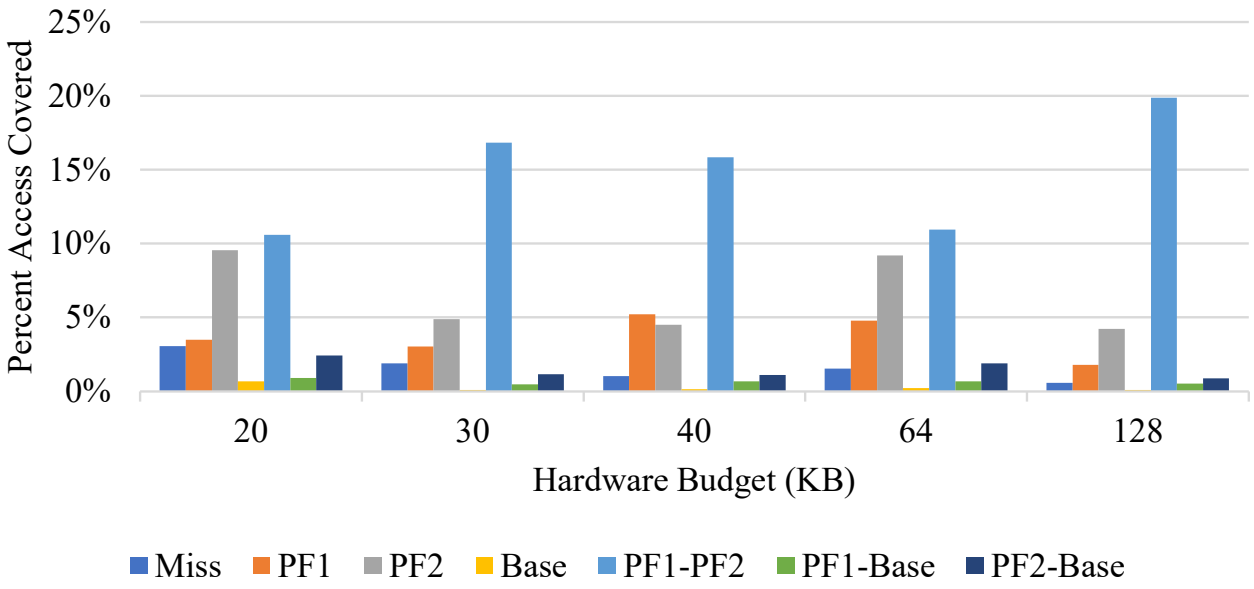


Figure 2.8: Coverage breakdown for Composite-2 prefetchers.

$\leq$1% for all hardware budgets.

PF1 and PF2 Hit (PF1-PF2): A hit occurs for both subprefetchers, showing an overlap between the prefetches selected by the subprefetchers.

PF1 and Baseline Hit (PF1-Base): A hit occurs for both the baseline and subprefetcher-1, meaning that subprefetcher-2 caused a harmful eviction otherwise covered in the baseline.

PF2 and Baseline Hit (PF2-Base): A hit occurs for subprefetcher-2 and the baseline, indicating subprefetcher-1 caused a harmful eviction.

PF1, PF2 and Baseline Hit: A hit occurs for all three indicating the prefetchers are retaining useful cache lines that hit in the baseline system. This is the most common occurrence for more than 70% of accesses. These are not included in Figure 2.8 to improve the visibility of the other scenarios.

We find that the highest occurring scenario is a hit for the baseline and both subprefetchers. We expect this behavior as the prefetchers try to cover misses with high accuracy to avoid thrashing the L1-I cache. This observation is supported by the low number of misses between the subprefetchers and baseline and the low number of unique baseline hits, indicating that the prefetchers

avoid harmful evictions. The number of unique hits for each subprefetcher varies at different sizes because each size contains a different subset of prefetchers. Interestingly, subprefetchers that individually perform better than their partnered subprefetcher (i.e., FNL+MMA vs. D-JOLT) have a higher number of unique hits than the other subprefetcher. As a subprefetcher's size increases to 64KB (128KB of total state), the number of unique hits is lower since each subprefetcher has enough storage to capture an application's behavior resulting in a higher number of overlapped hits between subprefetchers.

2.5 Related Work

Instruction prefetching has been heavily studied in recent years. This section breaks down prior work in instruction prefetching into hardware and software techniques. We then examine prior works in composite prefetching.

2.5.1 Hardware Instruction Prefetching

ConïňĆuence [64] leverages a single stream-based prefetcher, *i.e.,* SHIFT [65], to reduce misses in the L1-I cache and BTB. Confluence mainly overcomes the challenge that block-grain history presented in prior stream-based prefetchers [66, 65] is not suitable for filling the BTB. It synchronizes insertions and evictions from AirBTB—a block-based BTB that reflects instruction-grain information of individual branches with the L1-I. Confluence provides good speedup, but it requires prohibitively large hardware storage of a 10.2KB AirBTB backed by a 240KB SHIFT prefetcher, more than doubling the size of the L1-I and BTB. It also requires complex software support to maintain metadata in the last-level cache tags.

Boomerang [67] identifies BTB misses using a branch-predictor-directed prefetcher that extracts the branch target from prefetched cache blocks. Boomerang can match the performance of Confluence; however, by using the existing BTB and branch direction predictor, Boomerang reduces the prefetcher overhead to nearly zero. Unfortunately, Boomerang is ineffective on workloads with frequent BTB misses. The critical limitation of Boomerang is that a limited-capacity BTB cannot track a sufficiently large control flow working set to ensure efficient instruction

prefetching.

Shotgun [9] is a combined BTB-directed instruction cache and BTB prefetcher. It divides the BTB into dedicated BTBs for capturing global and local control flow, U-BTB and C-BTB, respectively. Shotgun leverages the BTB fill mechanism of Confluence to fill the BTB before the entries are accessed resolves BTB misses using the reactive BTB fill method of Boomerang. This method fetches the associated cache block from the memory hierarchy and extracts the necessary branch metadata. Shotgun improves BTB misses over Boomerang by up to 14%.

2.5.2 Software Instruction Prefetching

Record-and-Replay (RnR) [68] is a software-assisted hardware pre-fetcher that stores sequences of memory offsets based on information provided by programmer-inserted software hints. This information includes which data structures have irregular memory accesses, when to start the recording, and when to start replaying (prefetching) instruction streams. RnR can achieve over 95% prefetching accuracy and miss coverage. However, this scheme requires the programmer to annotate when to start/stop recording instructions to be replayed later and may be difficult to apply in applications with a large code base.

AsmDB [69] is a profile-guided software prefetching technique to identify high-impact misses not covered by a next-two-line pre-fetcher. The analysis selects insertion candidates based on their ranked impact on the system. A high-impact miss then has a instruction prefetch inserted before an instruction based on the likelihood that the instruction is located on an execution path that reaches the targeted miss. The insertion step avoids injecting instructions at areas where control flow heavily varies.

I-SPY [70] also relies on profile-guided analysis and uses AsmDB at link-time to determine frequently missing blocks. The authors propose conditional software prefetching and implement prefetch coalescing. To perform conditional prefetching, I-SPY calculates the conditional probability of each execution path within a control-flow graph leading to a miss in a block. It relies on the *presence* of blocks to identify the context instead of relying on the *order* of blocks. To perform coalescing, I-SPY analyzes all prefetch instructions injected into a basic block, and groups them

by context that they are conditioned on. It then attempts to merge multiple prefetch instructions into a single prefetch instruction. I-SPY injects an AsmDB instruction prefetch if it is unable to provide a conditional or coalesced prefetch

2.5.3 Composite Prefetching

Few composite prefetchers exist in the prior work, mainly in the data prefetching domain. Division of Labor, or DOL [17], attempts to exploit both simple and complex access patterns using a collaboration of specialized subcomponents for each pattern. This composite prefetcher is extendable with additional components as more access patterns are identified. Note that the hardware designer must identify the access patterns missing or necessary, making Division of Labor limited by the designer's knowledge.

Bouquet of Instruction Pointers [56], also creates a composite L1 data prefetcher. Bouquet of Instruction Pointers uses just that, a "bouquet" of pointers to classify instruction pointers and issue data fetch requests based on the classification. This technique covers and identifies a handful of memory access patterns that drive prefetches.

While instruction prefetching and data prefetching are similar in that they attempt to hide latencies induced by the Memory Wall [71], their access patterns and relationships to data diverge. The above works focus on data prefetching that relates certain instructions to data they access. In instruction prefetching, we are prefetching more instructions themselves, causing control flow to become a new important factor.

A final work, Divide and Conquer Frontend Bottleneck [55], warns against BTB-directed instruction prefetches. It presents the "harmful effects" of making instruction prefetchers dependent on BTB content. Instead, it proposes dividing the front-end bottleneck as follows: a sequential prefetcher to cover sequential misses, a discontinuity prefetcher, and pre-decoding prefetch blocks to reduce BTB misses. This divide-and-conquer method has the same area overhead of a BTB-directed prefetcher but outperforms it by 5% on average for their selected workloads.

As seen by the works described above, combining prefetchers, both in data and instruction prefetching, is not novel. However, these component prefetchers are *non-interchangeable* and

tuned for hardware size and prefetch specialty by the programmer, requiring in-depth knowledge of each component and how to make them work together. On the other hand, our proposition requires no knowledge of the prefetcher components and allows for previously unexplored component interchangeability.

2.6 Summary

Instruction prefetchers' performances are limited by hardware overhead constraints but do not gain increased performance with larger hardware budgets. Composite prefetching allows for higher performance at lower hardware budgets by combining the coverage of different complex prefetchers but is challenging to design effectively without making components targeting specific behaviors. We demonstrate a framework for selecting and integrating state-of-the-art complex prefetchers to find the best performing combination at various hardware budgets in a "plug-and-play" fashion that lightens the burden of tailor-making components for specific behaviors.

Our framework provides the basis for future work designing composite hardware prefetchers using heterogeneously sized components. A potential optimization is to share metadata storage between prefetchers and dynamically allocate storage to prefetchers that excel in predicting specific program phases. While not explored in this chapter, our framework provides the basis for future work designing composite hardware prefetching mechanisms using heterogeneously sized components. A potential optimization is to allow metadata storage to be shared between prefetchers and dynamically allocated to prefetchers that excel in predicting specific program phases.

Future work in composite prefetching may also explore selecting subprefetchers on the fly based on an application's specific characteristics. Software analysis of an application can provide hints to the hardware on the most appropriate subprefetchers for specific hardware, such as accelerators.

3. PERFORMANCE EFFECTS OF SOFTWARE INSTRUCTION PREFETCHING IN THE PRESENCE OF AN AGGRESSIVE FRONT-END

This chapter analyzes an industry-standard implementation of a decoupled front-end to identify its potential state. Furthermore, we identify how introducing new instructions by a software instruction prefetcher can degrade performance in these industry-standard decoupled environments as opposed to a more conservative model with which previous software instruction prefetchers have evaluated their techniques. We begin this chapter by introducing decoupled front-ends and modern software instruction prefetching techniques designed to alleviate the front-end bottleneck. We then characterize front-end behavior based on the state of the entries with the structures intended to increase the front-end's behavior and how these states affect performance. Following this characterization, we evaluate a modern software instruction prefetching technique and identify how the additional instructions inserted into the instruction stream adversely affect the front-end's performance.

3.1 Introduction

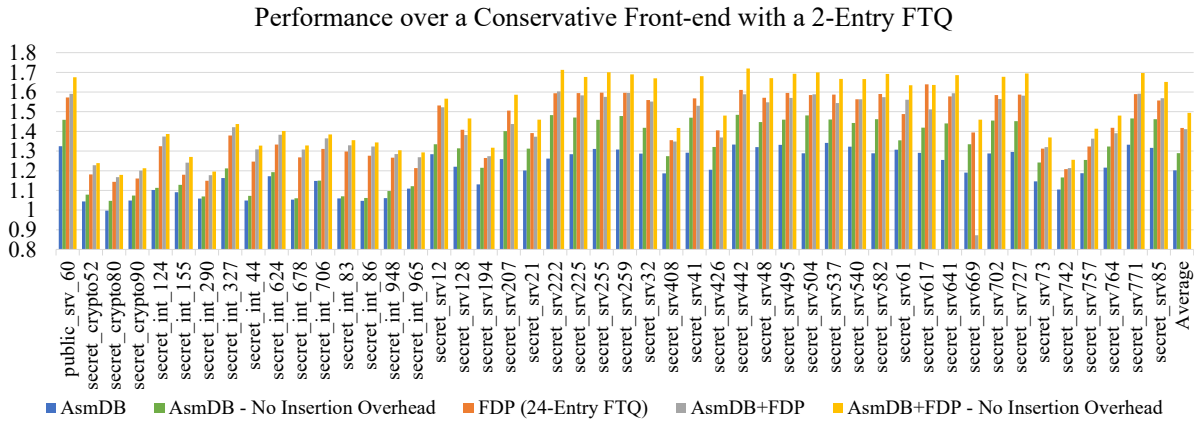


Figure 3.1: Comparison of front-end performance of AsmDB, an industry-standard FDP implementation, and EIP with an industry-standard FDP implementation over a conservative front-end 2-entry FTQ.

Figure 3.1 shows the performance improvement of AsmDB's Instructions-per-Cycle (IPC) in a conservative fetch environment compared to the IPC of a conservative front-end with no prefetching. Here, we implement AsmDB to profile and insert prefetches into a trace for a trace-based simulator and evaluate its performance over a conservative front-end implementing FDP with a 2-entry FTQ. We do not include the additional instructions AsmDB inserts when calculating its IPC. In the figure, we evaluate 48 workloads from the 1st Value Prediction Championship (CVP1) using ChampSim to explore the benefit of implementing AsmDB alongside an industry-standard[2] front-end. In a conservative front-end, similar to that used in the original evaluation of AsmDB, we find that AsmDB improves performance by roughly 20% geomean. The figure also shows the impact of increasing the depth of the FTQ to 24 entries (192, 32-bit instructions) to represent modern front-end designs more accurately. We find that FDP improves $\sim$41% over the conservative 2-entry FTQ implementation and outperforms AsmDB on a conservative front-end by $\sim$20%. Implementing AsmDB with a larger FTQ does not yield significant performance benefits on average and degrades performance in some cases. This result is counterintuitive, as AsmDB should capture performance benefits that FDP cannot cover ahead of its demand fetch. The figure also shows the impact of removing the additional instructions AsmDB inserts into the simulator's trace, allowing it to prefetch instructions at no cost. In this case, we find that AsmDB and an industry-standard FDP improve performance to $\sim$49% over the conservative baseline ($\sim$9% over an aggressive FDP). The conservative FDP combined with AsmDB also benefits from removing software prefetch instructions' overhead, but the industry-standard FDP sees no benefit unless the overhead is removed. This result raises the question: *why does FDP have such a substantial impact on the benefit of software instruction prefetching?*

Here, we deeply examine the front-end's behavior in conservative and industry-standard, aggressive FDP scenarios to understand how to implement better software prefetching mechanisms in future machines. Specifically, we identify three different scenarios the front-end may be experiencing and how introducing software prefetches can negatively impact front-end performance by increasing the occurrence of specific scenarios. We then discuss potential optimizations to soft-

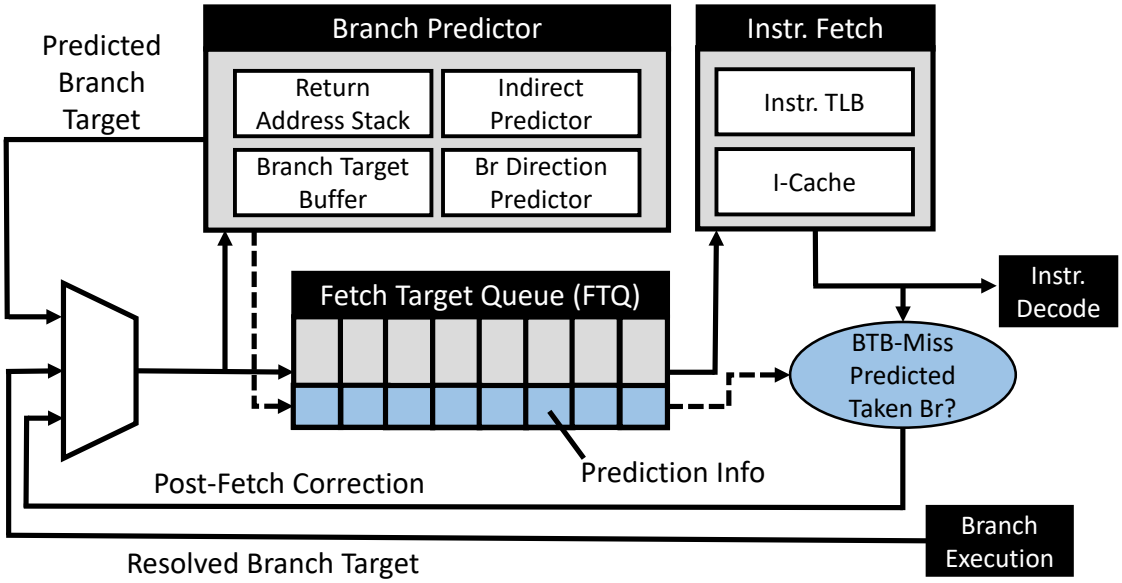


Figure 3.2: Overview of FDP implementation and optimizations from Ishii et al. [1].

ware prefetching that can reduce the overhead of inserting software prefetches and direct future design efforts.

3.1.1 Contributions

In this chapter, we characterize the side-effects of decoupled front-ends in software instruction prefetching. We evaluate state-of-the-art software prefetching techniques in a simulation environment reflecting a realistic decoupled front-end and explore potential solutions.

- We provide an in-depth characterization of front-end stalling behavior in a conservative vs. an industry-standard front-end.
- Evaluation of a state-of-the-art software instruction prefetcher in an industry-standard front-tend.
- Discussion of future areas of research to address the growing front-end bottleneck.

3.2 Background and Motivation

This chapter characterizes decoupled front-ends' side effects in prefetching software instruction. We evaluate state-of-the-art software prefetching techniques in a simulation environment reflecting a realistic decoupled front-end and explore potential solutions.

3.2.1 Decoupled Front-Ends

Reinmann et al. [27] originally proposed FDP to reduce the dependence of the front-end's fetch mechanism on the downstream execution core's branch resolution mechanism. The large size of modern branch target buffers (BTB) and high branch predictor accuracy allow FDP to decouple these branch prediction structures from a core's instruction fetch logic. The decoupling discussed in this design refers to the front-end's isolation from downstream execution elements. The fetch unit is permitted to run independently of execution, though execution will correct the fetch unit on a misprediction. By decoupling these structures, the fetch elements can aggressively run ahead and populate the Fetch Target Queue (FTQ) with speculative instructions, as shown in Figure 3.2. The FTQ is a dedicated buffer containing information about speculative fetches directed by the branch prediction structures on a 32B granularity or eight instructions. Each entry in the FTQ represents a basic block allowing one entry to represent eight instructions to capture varying sizes of basic blocks. FDP fetches the cache line addresses in the FTQ representing the start of a basic block and fills the L1-I before the demand request, regardless of their position in the FTQ. This allows fetches to the L1-I to occur out-of-order, but instructions must move to the decoder in-order to preserve the instructions' ordering. Entries in the FTQ pointing to the same basic block only require a single request to the L1-I, with larger FTQs allowing more aliasing and reducing requests to the cache. As entries become available in the FTQ, the BTB and branch predictors generate new fetch addresses to populate the FTQ.

Figure 3.2 illustrates the branch structures FDP relies on the Return Address Stack (RAS), indirect branch predictor, BTB, and branch direction predictor collectively speculate on future instructions. The branch predictors must continuously feed the FTQ with new instruction addresses,

also called program counters (PCs), to leverage the benefits of a decoupled front-end fully. These predictors rely on previously seen branch behavior to determine the future behavior of a particular branch. In particular, these predictors maintain a Global History Register (GHR) to track the predicted outcomes of each branch. An imprecise GHR can heavily affect the predictors' ability to speculate on future instructions.

We focus our study on a recent FDP-based design that introduces two optimizations to FDP to improve performance and address the challenges of an industry-standards FDP [1, 2]. The first optimization minimizes the noise by running ahead by preventing the GHR from holding the history regarding not-taken branches that miss in the BTB since they do not appear as branches but rather as sequential instruction accesses. Limiting the history of taken branches, the GHR may be updated when the BTB is updated with the new branch information. The GHR can then be flushed and updated to accurately represent the branch history.

The second optimization extends previously proposed *Post-Fetch Correction (PFC)* [72] to allow the information leading to incorrect fetches to be corrected once the branch has resolved. All branch instructions and targets are identified once instructions are pre-decoded after fetch. The branch results are compared to the information stored in the GHR, checking if unconditional and conditional branch outcomes were correctly predicted. If not, the FTQ is flushed, the GHR is corrected, and prefetching continues.

3.2.2 AsmDB: Modern Software Instruction Prefetching

Software prefetching has recently resurfaced as a potential technique to alleviate the front-end bottleneck [32, 73]. In general, software instruction prefetching techniques follow the general steps of **(1)** execute and gather information, **(2)** generate a profile, **(3)** modify the target binary, **(4)** rerun binary with software instruction prefetching.

Collecting statistics about different basic blocks' behaviors allows profiling techniques to generate a control flow graph (CFG) of an application's execution. The software profile recreates the CFG to identify instruction behavior that impacts performance. This chapter focuses on a contemporary state-of-the-art software prefetching technique, Assembly-Database (AsmDB) [32].

AsmDB targets warehouse-scale applications, which prior work has shown to suffer from the front-end bottleneck problem, with front-end stalls accounting for 15-30% of pipeline stalls [8]. AsmDB profiles the target application to examine the program's control-flow behavior and the program's high-impact misses, traverses the CFG and selects insertion sites for the software instruction prefetches based on the likelihood of a particular path leading to the target miss, and reassembles the program with software instruction prefetches at the selected insertion sites. Below we discuss the details of AsmDB's criteria for targeting particular instructions and selecting the most appropriate insertion site for a software prefetch.

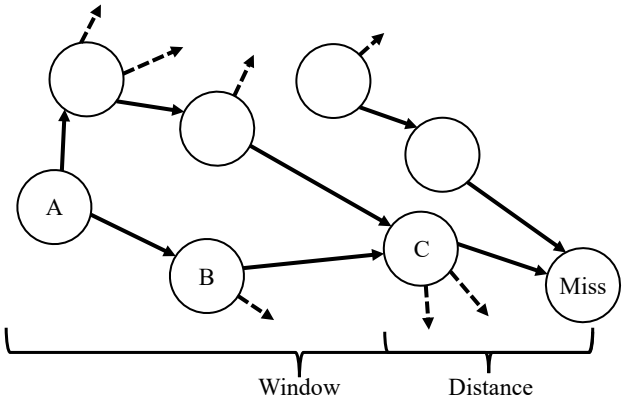


Figure 3.3: An example of the CFG generated by AsmDB's software analysis to select locations to insert software instruction prefetches.

3.2.2.1 Selecting High-Impact Instructions

AsmDB's profiling stage gathers information about an application's instruction stream behavior using the Intel processors' Last-Branch-Record (LBR) hardware. The collected data allows AsmDB to track instructions with high L1-I miss rates and their location within the CFG, representing basic blocks as nodes and branches as edges.

Once AsmDB establishes the CFG, it generates an ordered list of potential prefetch targets by ranking the instructions based on their misses. The highest-ranked instruction candidates are

selected for prefetch insertion. AsmDB prioritizes instructions with high miss rates, assuming these instructions contribute the most to stalling the front-end.

3.2.2.2 Inserting Software Instruction Prefetches

Once the software analysis selects the highest-impact misses to target for software prefetches, AsmDB traverses the CFG backward from the target miss, identifying paths leading to the target. AsmDB requires that the insertion site is beyond a minimum distance away from the miss to ensure that the prefetches are filled in advance of the demand. AsmDB approximates the distance by multiplying an application's Instructions Per Cycle (IPC) by the LLC's access latency. The distance is the worst-case fetch latency of each instruction, giving AsmDB a notion of the minimum instructions ahead of a miss to insert the prefetch to cover its fetch latency successfully. The maximum number of instructions away from a miss a prefetch can be inserted is called the *window*. Figure 3.3 is an example of the CFG analysis. In this example, the node *C* is not the minimum distance away and not a suitable insertion location. Nodes A and B are within the window and the minimum distance, so AsmDB considers them for insertion sites to the miss.

As AsmDB considers an insertion site, it considers what fraction of the succeeding paths include the target instruction within the window, called the *fanout*. Fanout directs the aggressiveness of the prefetch insertion since higher fanout insertion sites are less likely to lead to the target miss. Increasing AsmDB's fanout decreases its accuracy but results in higher miss coverage.

3.2.2.3 AsmDB and Industry-Standard Decoupled Front-Ends

Applying software instruction prefetching requires the application to execute at least once to generate a profile of its instruction stream behavior. Gathering basic block information over multiple runs can improve an application's profile. Regardless of the CFG model's accuracy, software instruction prefetching cannot receive feedback about its predictions during execution. Modern machines, especially in the context of servers, use an out-of-order processor paradigm which results in nondeterministic behavior between different executions of the same application.

Software instruction prefetching and FDP have each been proposed to maximize front-end

bandwidth. FDP has seen widespread deployment in modern processors [2]. Figure 3.1 shows that a deep FTQ gives a strong performance benefit. Critically, however, prior work in software instruction prefetching did not evaluate their proposal in the context of decoupled front-ends due to limitations in the evaluation infrastructure [74]. While AsmDB shows a significant performance improvement when executed on a conservative, 2-entry FDP implementation, combining AsmDB and an industry-standard FDP sees no benefit and causes performance degradation in some workloads. By removing the overhead of inserting instructions into the front-end instruction stream, we find that AsmDB provides further performance benefits over the industry-standard FDP. This observation indicates that software instruction prefetching interacts poorly with an industry-standard FDP and that the overhead of additional prefetch instructions removes any benefit that the prefetches themselves would gain.

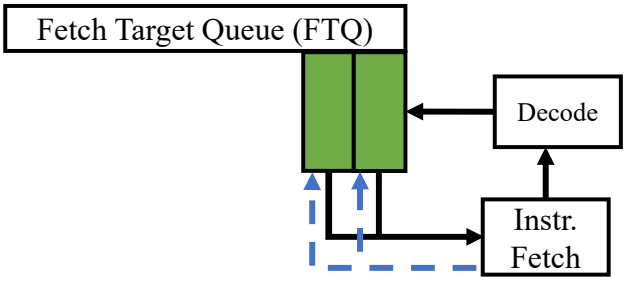
3.3 Characterizing Front-End Behavior

The front-end acts as the forward engine of modern processors attempting to fetch, decode, and issue instructions at a high throughput to drive execution continuously. As described in Sec. 3.2.1, a modern decoupled front-end generally has a form of FDP. We find that interacting with an industry-standard FDP nullifies software instruction prefetching's benefit.

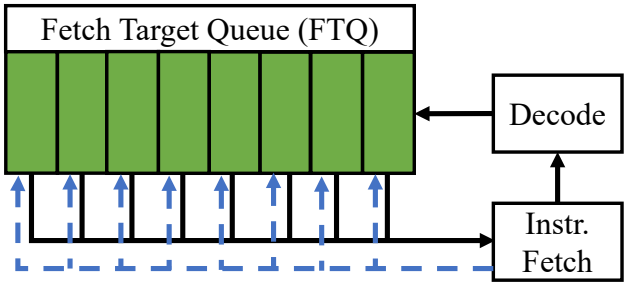
This section investigates the interaction between the FDP state and the overhead of additional prefetch instructions in an application. We provide a taxonomy of three possible front-end states, the causes of each state, and its potential performance penalty. Specifically, we look at the potential outcome if particular FTQ entries stall in a conservative FDP with a 2-entry FTQ and an industry-standard FDP with a 24-entry FTQ. We discuss these scenarios in the context of software instruction prefetching and its influence on the FTQ's state and performance.

3.3.1 Scenario 1: Shoot Through

Scenario 1 is the ideal FTQ state, where every FTQ entry has completed its fetch, and all instructions are available for decoding. The front-end bottleneck is nonexistent here, and the fetch bandwidth depends only on the decode stage's available bandwidth. In the conservative FDP,



(a) Scenario 1 in a conservative pipeline.



(b) Scenario 1 in an industry-standard pipeline.

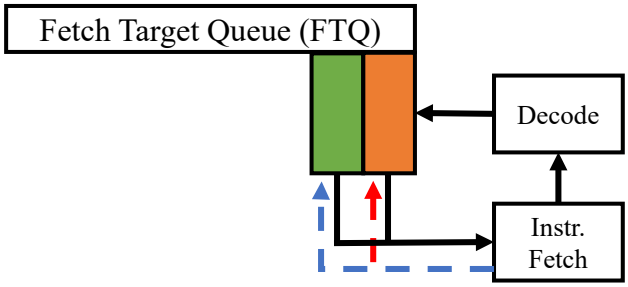
Figure 3.4: Scenario 1 for conservative and industry-standard FDP implementations.

shown in Figure 3.4a, both FTQ entries are ready for decoding. The low number of FTQ entries limits this scenario's benefit. Figure 3.4b illustrates Scenario 1 for the industry-standard FDP implementation in which each FTQ entry has completed its fetch and is available for decoding, limited only by the decode stage's bandwidth.

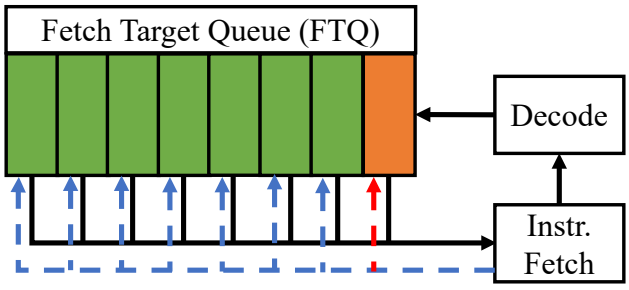
In this scenario, the instruction prefetcher ideally fills the FTQ, prefetching all entries in the FTQ before they can incur stalls. If this scenario is common, it may motivate prefetch designs to have higher coverage without consideration for the cost of redundant prefetches as they attempt to cover as many upcoming instructions as possible.

3.3.2 Scenario 2: Stalling Head Instruction

In the next state, the head FTQ entry is waiting for its cache line to be fetched while the succeeding entries have completed fetch. FDP can issue FTQ entries to the L1-I in any order, but the instructions in the FTQ must be sent to the decoder in program order. For a conservative FDP implementation, shown in Figure 3.5a, a single fetched FTQ entry must stall until the head has



(a) Scenario 2 in a conservative pipeline. A single FTQ entry stalls the head (orange) of the FTQ while the following instruction (green) is ready to be sent to decode.

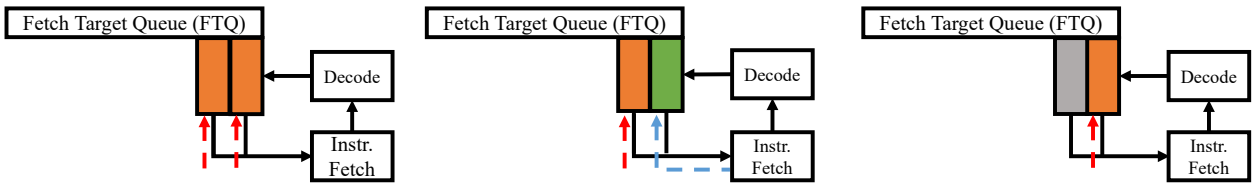


(b) Scenario 2 in an industry-standrard pipeline, where the head instruction is still waiting for fetch to complete, resulting in potentially 23 FTQ entries to stall as a result.

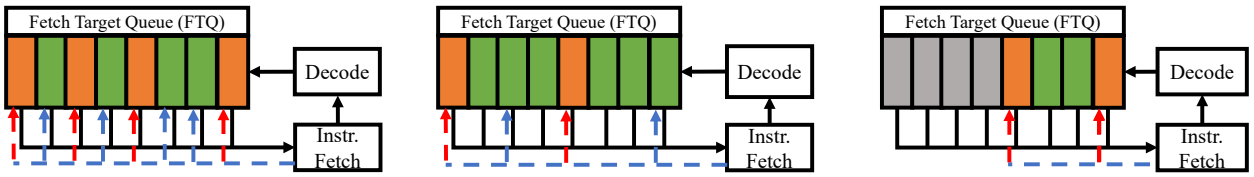
Figure 3.5: Scenario 2 for conservative and industry-standard FDP implementations.

received its cache line. In the industry-standard FDP implementation shown in Figure 3.5b, the head instruction causes up to 23 other FTQ entries to stall after completing their fetch, limiting the achievable fetch throughput. This scenario is the most commonly thought-of manifestation of the front-end bottleneck. The performance penalty depends directly on how long it takes for the head instruction to complete its fetch: a low-latency stall has a lower penalty than a request that misses the last-level cache. Despite this bottleneck, FTQ allows entries with a fetch latency lower than the head entry to be fetched before moving to the head of the FTQ.

As discussed in Sec. 3.2.1, an entry in an FTQ design represents a basic block of up to eight instructions. The decoder's bandwidth may outpace fetch if multiple small basic blocks occupy the queue. Furthermore, FDP may stall to wait for a branch misprediction or BTB miss to resolve, allowing FTQ to issue its contents to the decoder before fetch resumes. As a result, a fetch entry can occupy the head entry for the entirety of its fetch latency, potentially causing a significant loss



(a) Scenario 3 in a conservative pipeline.



(b) Scenario 3 in an industry-standard pipeline.

Figure 3.6: Scenario 3 for conservative and industry-standard FDP implementations.

in potential performance due to halted throughput.

An ideal prefetcher would identify the head entry as the source of the performance bottleneck and prefetch it as early as possible, converting Scenario 2 into Scenario 1. Since previously proposed software instruction prefetchers operate by targeting instructions with high miss rates, they do not target these bottleneck-critical instructions unless they have a high miss rate. Software instruction prefetching introduces new instructions into the instruction stream, possibly incurring additional misses, leading to this scenario more frequently.

3.3.3 Scenario 3: Shadow Stalls

A more complex version of the previous scenario occurs when multiple stalling entries follow a stalling head entry, and the head's fetch latency does not entirely cover the latency of subsequent entries. In the conservative, 2-entry FDP in Figure 3.6a, both entries wait for fetch to complete. The head entry has a shorter fetch latency than the following entry. When it completes its fetch, it promotes to decode. The following entry moves to the head, and the FTQ continues stalling. The performance penalty in the conservative FDP depends on the difference in instruction latencies, possibly stalling the following instruction cache line inserted into the FTQ.

This scenario is more complex in the context of a deeper FTQ, with an example shown in

Figure 3.6b. The head entry hides only part of the subsequent entries' latency. When it completes, multiple completed entries in the FTQ can promote to decode, up to an uncompleted entry whose latency was not covered by the head entry. This scenario's performance penalty varies based on the amount of throughput recovered by entries with fetch latencies covered by the head entry's latency.

A prefetcher can mitigate this scenario by prefetching all instructions in the FTQ ahead of their transition to the head of the FTQ. The impact of the two stalling entries varies based on their fetch latency and can change during execution, making it difficult for a prefetcher to identify this behavior in a steady state. A software instruction prefetcher may identify both instructions due to their high miss rates. However, the inserted instructions may themselves stall, which is not a solution to this scenario, and they may also change the spacing between stalling entries, complicating how the latencies are hidden.

3.3.4 FTQ State and Software Instruction Prefetches

Ideally, the FTQ would always be in Scenario 1, where entries are always available to move to the decode stage. Prefetching attempts to reduce the incidence of Scenarios 2 and 3 by prefetching entries that would stall at the FTQ's head.

In conservative front-ends, the number of entries structurally limits the FTQ's throughput. Mitigating any stall cycles at the head of the FTQ results in increased performance. Introducing new instructions into the instruction stream can, at most, stall both FTQ entries and has a low performance penalty as it covers stalls that will occur later in the program.

In contrast, a deep, industry-standard FDP is much more affected by inserting additional instructions, increasing the chances for Scenarios 2 and 3. Due to the larger number of entries, it is unlikely that a software instruction prefetching scheme can amortize the overhead of the inserted instructions by covering other stalls. Furthermore, by introducing new instructions, the application's miss profile and the impact of particular misses can change drastically.

3.4 Methodology

We evaluate prior hardware and instruction prefetchers using ChampSim [59]. ChampSim is a trace-based simulator commonly used to evaluate prefetching techniques. We use a modified version of ChampSim implementing the decoupled front-end described in [2], modeling FDP. Our system configuration, shown in Table 3.1, is similar to a modern Sunny Cove core. We evaluate the front-end's behavior using a subset of the traces used in the First Value Prediction Championship (CVP1), converted into the ChampSim trace format. We evaluate 47 traces that reflect large instruction working sets and provide high instruction pressure. The selected traces see an average of 25.5 L1-I misses per thousand instructions (MPKI). The traces reviewed here represent a range of instruction footprint sizes resulting in MPKI's ranging from $\sim$2 to $\sim$28 MPKI. We simulate 100 million instructions for each workload for profiling, analysis, and evaluation.

A real-world system would perform profiling and analysis to insert software instruction prefetches into a target binary. However, we require specific hardware and software instruction prefetch formatting to allow the hardware to train and prefetch instructions. We implement previously proposed software instruction prefetchers following the general workflow of prior work. First, we generate instruction traces from ChampSim containing the behavioral information of basic blocks, including how long they stall the front-end and when a miss occurs at the L1-I. We then use the instruction trace to recreate the application's control flow graph. AsmDB traverses the control flow graph and inserts prefetches at the end of basic blocks that lead to the high-impact instructions. AsmDB generates a new ChampSim trace, shifting each instruction's address appropriately to simulate the front-end and L1-I pressure induced by inserting prefetch instructions into an application's binary.

Additionally, we modify ChampSim to recognize software instruction prefetches. The prefetch instructions are treated as any other instruction request, inserted into the decoupled frontend's FTQ, and then fetched from the L1-I cache. Once fetched, we assume a predecoder identifies the prefetch instruction and triggers a prefetch for the target instruction.

We also evaluate AsmDB's idealized performance benefit by ignoring any overhead from the

software instruction prefetches. Each prefetch is issued on a triggering PC, but the prefetch instruction is not inserted into the front-end. The performance of AsmDB with no insertion overhead provides the benefits of prefetching without the cost of interacting with FDP in the front-end.

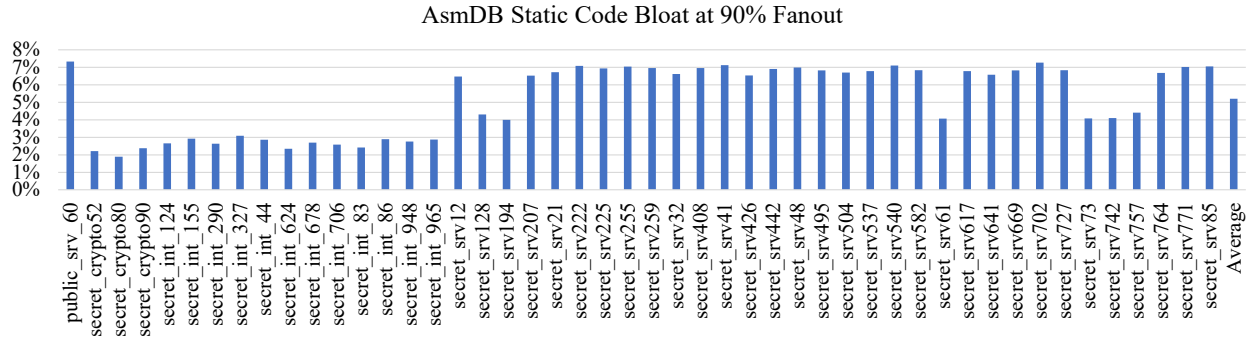
Block	Configuration
Out-of-Order Core	4 GHz 352-entry ROB 128-entry Unified Reservation Station
Fetch Width Decode Width	6-wide Fetch, 24-entry/2-entry Fetch Target Queue 6-wide Decode, 60-entry Decode Queue
L1 BTB L2 BTB Branch Direction Predictor	128-entry, 2-way, 1-cycle latency 8192-entry, 4-way, 2-cycle latency TAGE-SC, 8-table, 1024-entry/table
Indirect Predictor	ITTAGE, 8-table, 512-entry/table
Return Address Stack	32-entry
Branch Predictor Bandwidth	Up to 12-instructions or 1-taken per cycle
L1 DCache	48 KB, 12-way, 5 cycles 16 MSHRs, LRU, Next-Line prefetcher
Private L1 ICache	32 KB, 8-way, 4 cycles 16 MSHRs, LRU
Private L2 Cache	512 KB, 8-way, 10 cycles 32 MSHRs, LRU, Non-inclusive, SPP [75]
Shared LLC	2MB/core, 16-way, 20 cycles 32 MSHRs, LRU, Non-inclusive
DRAM	4 GB 1-Channel (single-core) 8 GB 2-Channels (multi-core) 64-bit channel, 1600MT/s

Table 3.1: Simulation parameters based on previous work [1, 2] evaluating the efficacy of hardware prefetchers in a decoupled front-end environment.

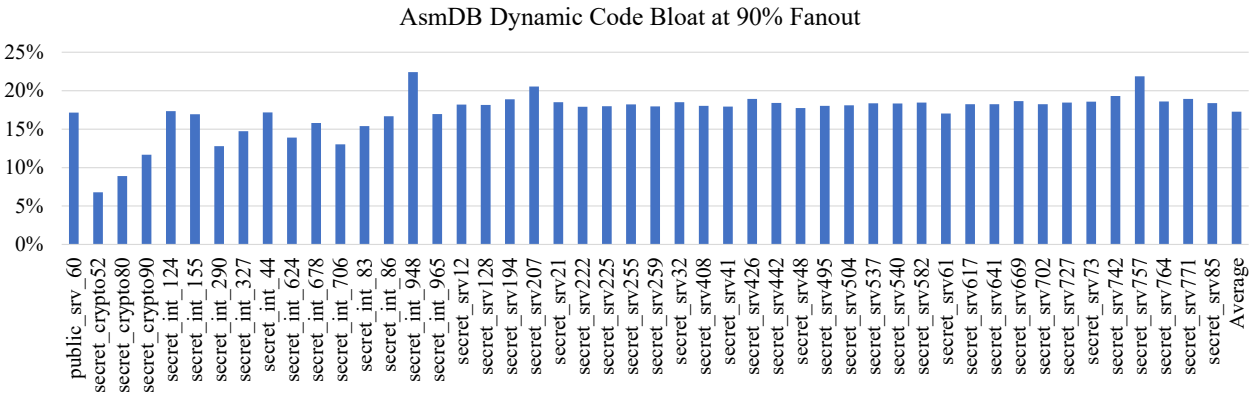
3.5 Front-End Analysis

The performance of any front-end system is proportional to the number of instructions fetched per cycle and inversely proportional to the number of stall cycles. In this section, we evaluate the performance of our design by distilling this figure into the percentage increase in number of instructions, called *code bloat*, and the average number of stall cycles per instruction. Each of these figures should be as low as possible, since they both represent the overhead introduced into

the system by inserting additional instructions into the front-end. We compare AsmDB's software prefetch instructions effect on a conservative (2-Entry FTQ) implementation and a more industry-standard FDP (24-Entry FTQ).



(a) The static code bloat for AsmDB represents the percent increase in the overall size of the binary due to inserting software prefetches.



(b) The dynamic code bloat for AsmDB represents the percent increase in the number of fetched instructions as a result of inserting software prefetches into the application's binary.

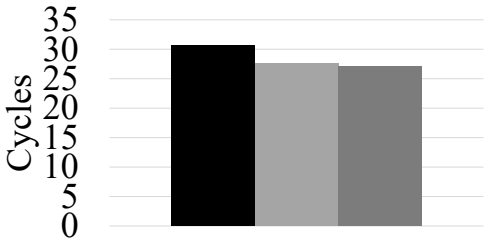
Figure 3.7: Static and dynamic code bloat

3.5.1 Code Bloat

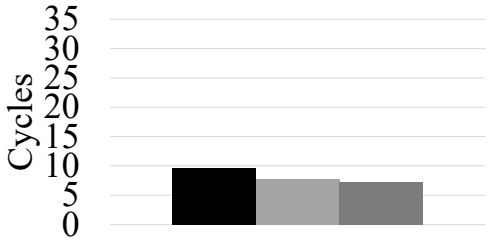
To be performant, AsmDB needs to target a large number of misses and allow for insertion points with high fanout insertion points [73, 32]. Each miss targeted for prefetching requires a software prefetching instruction to be inserted into the binary, increasing its static size. Inserting additional instructions shifts the instruction addresses within the binary, shifting the cache lines'

contents. AsmDB accounts for this shift during prefetch generation, but changing cache lines' contents can potentially change which cache lines may stall the FTQ. Figure 3.7a shows the increase in the program's size, called the *static code bloat*. Figure 3.7b shows the increase in the number of instructions executed due to the inserted prefetches, referred to as *dynamic code bloat*. Each software instruction prefetch can be executed multiple times throughout execution resulting in higher dynamic code bloat than static code bloat. Generally, static and dynamic code bloat should be minimized to reduce prefetch overhead, but applications with large instruction footprints and many high-impact misses require more prefetches to improve performance.

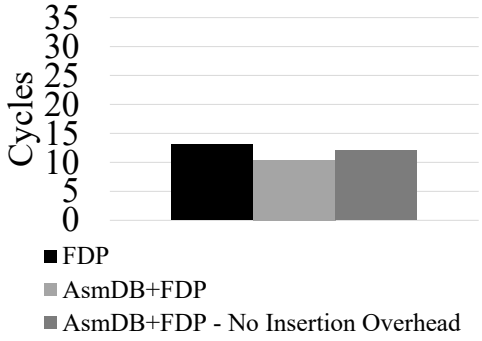
3.5.2 Changes in Stalling Head FTQ Entries



(a) Average cycles to fetch the head of a 24-entry FTQ.



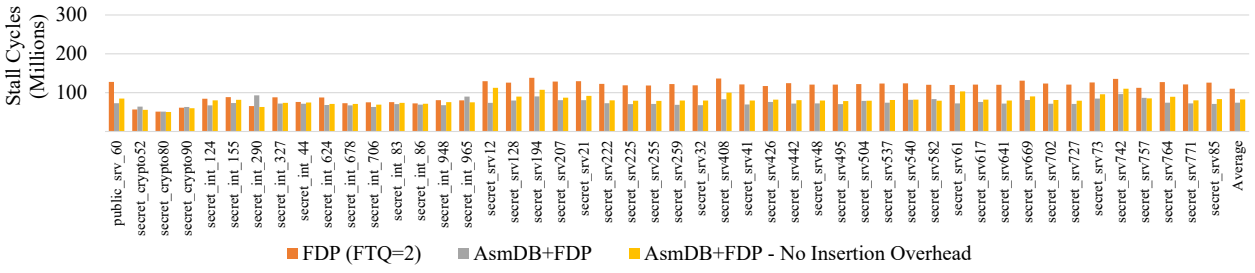
(b) Average cycles to fetch the head of a 2-entry FTQ.



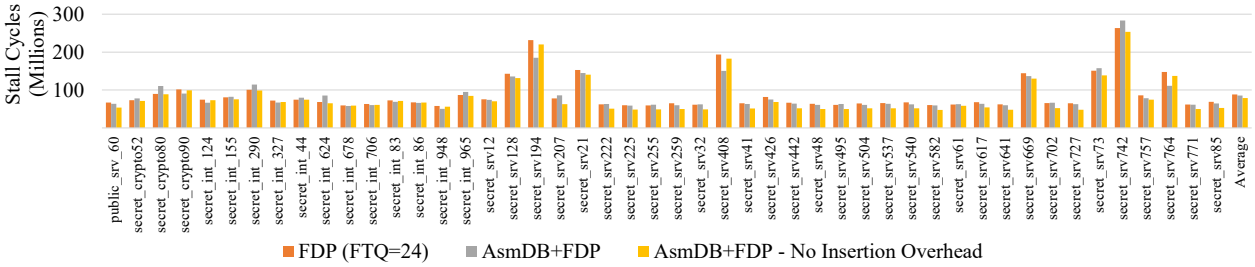
(c) Average cycles to fetch an entry not at the head of a 24-entry FTQ.

(d) Average cycles to fetch an entry not at the head of a 2-entry FTQ.

Figure 3.8: The number of cycles to cover a head instruction tends to be larger versus an FTQ entry not at the head, indicating that the head of the FTQ tends to be a miss in the L1-I.



(a) Stalls Caused by Head Entries in 2-Entry FTQ



(b) Stalls Caused by Head Entries in 24-Entry FTQ

Figure 3.9: Number of stalls incurred by the head entries fro the 24-entry and 2-entry implementations of FDP.

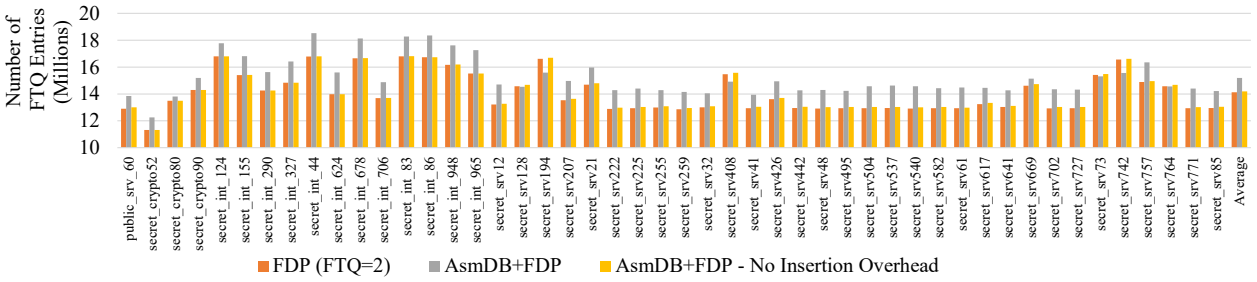
Comparing the average number of cycles to fetch the head FTQ entry to the number of cycles to cover an entry not at the head of the FTQ in Figure 3.8, we find that stalling head entries tend to have higher latencies. Comparing the average fetch times between the 24-entry and 2-entry FDPs in Figures 3.8a and 3.8b, we observe that a deeper FTQ has longer fetch times.

Since the FTQ merges requests to the same cache line (*e.g.* due to loops), a deeper FTQ has more opportunities for positive aliasing between entries. We find that the 24-entry FDP experiences $\sim$14 % less L1-I accesses than the 2-entry FDP on average. The remaining FTQ entries are more likely to be sent to the cache and have longer fetch latencies. The following FTQ entries that hit in the L1-I are filled before moving to the head of the queue, and the remaining entries that can potentially stall the head must take longer than the current head instruction to fill.

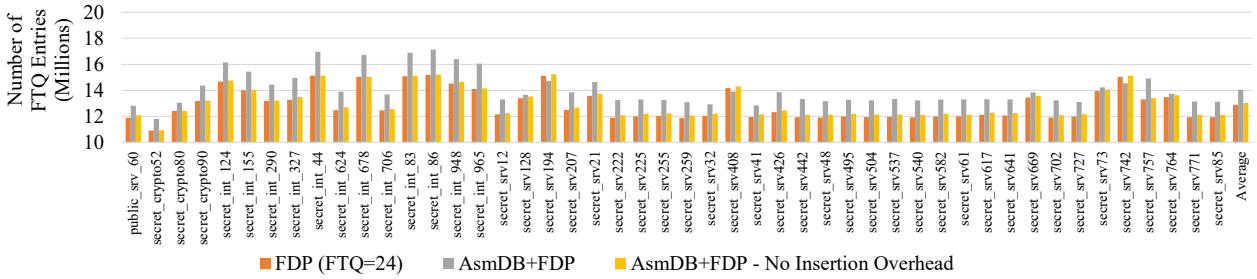
3.5.3 AsmDB's Impact on the Occurrence of Scenario 2

We measure the number of stalls incurred by the FTQ's head instruction in Figure 3.9. The 24-entry FDP generally experiences fewer stalls at the head instruction. This is a side-effect of the

deeper FTQ allowing for instructions to have more opportunities to alias, resulting in the remaining instructions having long fetch latencies or completing fetch before reaching the head of the FTQ. This effect is supported further by Figure 3.10, which shows that the number of entries that have completed their fetch and are waiting for the head instruction is lower. The figure illustrates that, on average, the 24-entry FDP decreases the number of waiting instructions compared to the 2-entry FDP due to its deeper FTQ and experiencing head entries with large fetch latencies.



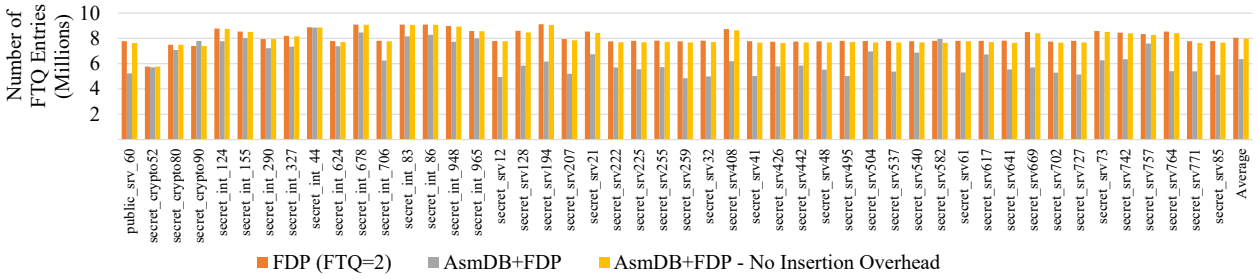
(a) Number of FTQ Entries waiting on a Stalling Head Entry in a 2-Entry FTQ



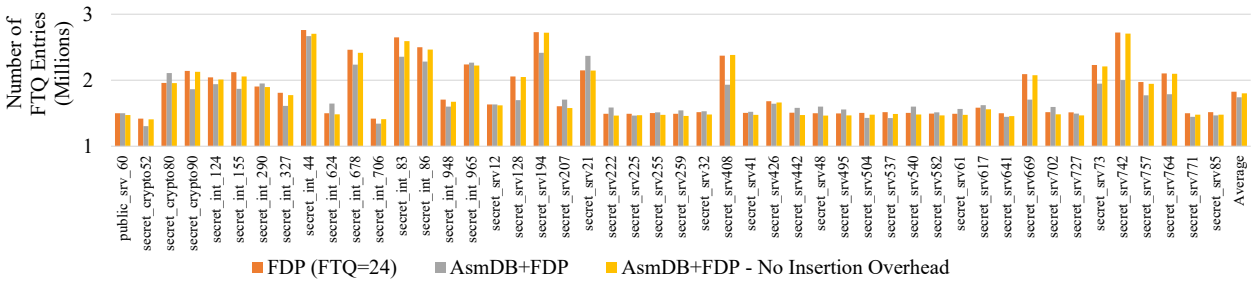
(b) Number of FTQ Entries waiting on a Stalling Head Entry in a 24-Entry FTQ

Figure 3.10: This figure illustrates the number of FTQ entries that are forced to wait on a stalling head instruction before progressing through the FTQ. While the conservative FDP has more waiting instructions overall, the increase in waiting instructions in the 24-entry FDP represents a loss of potential performance.

We measure the number of instructions forced to stall due to waiting on the head instruction to complete its fetch in Figure 3.10. Comparing the results of Figures 3.9 and 3.10 relative to AsmDB, we find that it increases the number of stalling instructions for conservative and industry-standard FDP implementations compared to their respective baselines, indicating an increase in the occurrence of Scenario 2 (Sec. 3.3.2. The average number of cycles to fill a head instruction



(a) Number of FTQ Entries Partially Covered by a Stalling Head Instruction in a 2-Entry FTQ



(b) Number of FTQ Entries Partially Covered by a Stalling Head Instruction in a 24-Entry FTQ

Figure 3.11

is much higher in the industry-standard. The number of overall waiting entries is lower in the industry-standard FDP than the conservative FDP, meaning any delay in the industry-standard FDP will have a higher impact on the throughput of the FTQ. Although AsmDB tends to remove the number of stalls caused by the head entry, any stall at the head of the deeper FTQ that delays sending instructions to decode will result in a loss of potential performance gain.

Increasing the number of stalls caused by an entry at the head of the FTQ has less impact in a conservative front-end since it will delay a single FTQ entry at most, or roughly 16 instructions assuming each instruction is 32 bits. In contrast, additional stalling entries in the FTQ heavily affect an industry-standard FDP since, in the worse case, it can delay up to 23 FTQ entries or 184 instructions. Although any stall causes delayed execution overall, the high throughput of the industry-standard FDP and longer fetch times make the performance impact of stalling instructions much higher than the conservative FDP. The significant number of stalling entries introduced by AsmDB, regardless of their fetch latency, results in more waiting entries that consume its potential performance benefit.

3.5.4 Software Instruction Prefetching Impact on Scenario 3

We measure the number of stalling entries in the FTQ that move into the head entry position before completing their fetch or are partially covered by the previous stalling head entry in Figure 3.11. We find that overall the 24-entry FTQ experiences fewer partial stalls than the 2-entry FTQ, as the stalling head entries have more significant latencies capable of covering the following outstanding requests.

AsmDB with instruction overhead demonstrates a decrease in partially covered instructions, reducing the occurrence of Scenario 3 (3.3.3). A reduction in Scenario 3 indicates that when Scenario 2 occurs, the head entry covers the fetch latency of the following entries. This conversion contributes to the increased number of waiting entries in Figure 3.10b as previous partially covered entries are complete and waiting for decoding.

3.6 Related Works

3.6.1 Hardware Prefetching

Prior work in hardware instruction prefetching has examined using, modifying [29, 31], or replicating [76] branch prediction structures to direct the L1-I prefetch engine's prefetches by building a context and history leading to the current execution behavior [30, 77, 78]. Previously proposed designs also leverage the decoupled nature of modern processors' front-ends to prefetch future instructions, with optimizations to handle mispredicted branches and branch targets [27, 79].

An alternative to predicting an application's control-flow is leveraging the repetitive nature of instruction streams to record recurring instruction behavior. A triggering point is selected to allow the prefetcher to replay misses in the context of recently executed instructions [80, 81, 82, 83, 84, 21]. These prefetchers often attempt to predict control-flow outcomes or disregard them as noise to provide a more concise view of the current execution context.

3.6.2 Software Prefetching

Callahan [35] proposed one of the first software prefetching designs for data and inserts non-blocking software prefetches into a binary at compile time. They assume that the accesses to

elements within an array in nested loops cause many misses and place prefetches before these accesses.

Luk and Mowry propose cooperative prefetching [34], which implements a software prefetcher in tandem with a prefetch filter. Their software analysis recreates the control flow graph of an application and then targets discontinuities (branches) between basic blocks for prefetching. They include additional analysis to combine, remove, compress, and hoist software prefetches to reduce the overhead of inserting prefetches into a binary. Similarly, Mowry et al. propose targeting instructions that frequently cause misses within a loop, inserting prefetches outside of an unrolled loop and scheduling based on an estimated access latency [33].

I-SPY [73] extends AsmDB to build a context of the paths leading to a miss by tracking the branch information leading to the miss. The context is embedded in prefetches with recurring contexts, and prefetching hardware compares it to the execution context to conditionally issue a particular prefetch. They also propose coalescing prefetches with addresses that are within a set distance from one another. If I-SPY cannot issue a conditional or coalesced prefetch, it defaults to AsmDB's base software prefetches.

3.7 Summary

Software instruction prefetching is a promising solution to the front-end bottleneck; however, prior work uses a conservative front-end model, which inflates the performance benefit of software instruction prefetching. Implementing AsmDB with a contemporary FDP model does not provide further performance benefits due to the overhead of the additional instructions the software prefetcher inserts. We identify the possible scenarios the front-end encounters when fetching instructions and how changing the likelihood of these scenarios occurring can impact performance. An industry-standard front-end model is sensitive to changes in the instruction stream as they exhibit high throughput heavily impacted by additional fetch latency.

Future work in software instruction prefetching may alleviate the overhead of inserting instructions by reducing the number of inserted instructions or directing the software prefetcher to adapt to an application's front-end behavior. We hope to excite future research in creating software in-

struction prefetchers aware of the effects of additional instructions and may leverage front-end characteristics to improve performance.

4. HARDWARE TROJANS CAPABLE OF EXPLOITING CACHE COHERENCE IN 2.5D CHIPLET SYSTEMS [1]

This chapter explores the basic threats an attacker can mount against the coherence protocol with a hardware Trojan in a chiplet-based 2.5D environment. Understanding possible threats against chiplet-based designs is vital to developing defenses capable of hardening the coherence system. These basic threats are ineffective as standalone attacks. However, we demonstrate that these attacks can form the fundamental stages within a more complex attack capable of violating the integrity of memory operations in an entirely separate memory space and chiplet than the compromised chiplet containing a hardware Trojan. First, we provide a background in hardware Trojans, 2.5D integrated chiplet environments, and coherence protocols. We then detail the basic attacks a hardware Trojan can mount against a 2.5D system's coherence protocol. Following these basic attacks, we design and demonstrate a complex hardware Trojan attack against a 2.5D system.

4.1 Introduction

In this chapter, we propose Trojan attacks that leverage the coherence system protocol to maliciously manipulate the victim process' memory. We first describe fundamental attacks that a Trojan can mount on coherence systems, based on *passive reading*, *masquerading*, *modifying*, and *diverting* attacks [85]. We examine how to implement these attacks, exploiting the coherence system at a hardware level, thus increasing the scope of their attack surface. While each of these attacks may violate the security of a system individually, we further show that adversaries can orchestrate them to perform complex attacks that modify *any* process' memory. These are purely hardware-centric attacks that contemporary software defense mechanisms *cannot* thwart since all exploited coherence interactions are transparent to software and legal within the coherence protocol. No prior work considers such attacks on coherence systems, neither in the context of 2.5D

[1] Reprinted with permission from G. A. Chacon, C. Williams, J. Knechtel, O. Sinanoglu and P. V. Gratz, "Hardware Trojan Threats to Cache Coherence in Modern 2.5D Chiplet Systems," in IEEE Computer Architecture Letters, vol. 21, no. 2, pp. 133-136, 1 July-Dec. 2022, doi: 10.1109/LCA.2022.3216820.

systems with chiplets nor for traditional 2D systems.

Contributions. This chapter provides new insights into how Trojans can manipulate coherence systems to violate the security of a chiplet system. We present a simulated example of a substantial attack that can directly manipulate memory in an address space other than that of the compromised chiplet. This chapter makes the following contributions:

- We present a classification of potential attack surfaces for Trojans that a malicious actor could exploit.
- We demonstrate how a fundamental hardware Trojan attack can create a significant side-channel attack when conspiring with a spy process.
- We demonstrate how to use these different and fundamental attacks to orchestrate a complex Trojan attack in a chiplet-based system.
- We provide a basis for future work exploring possible threat vectors and hardening modern chiplet designs.

4.2 Design of Hardware Trojans Targeting Coherence Systems

Coherence protocols ensure updates to cached copies of data are visible to all cores and other IP blocks in modern multi-core designs [45, 37, 40]. Coherence schemes can be broadly categorized as broadcast (or snooping) protocols [46, 47, 48] and directory protocols [49, 50, 51]. While simple to implement, broadcast protocols suffer from high traffic due to the amount of messages multi-core systems require to maintain coherence. Directory protocols allow for fine-grained state tracking and unicast messages, making them highly scalable but difficult to implement and have higher access latencies. Coherence protocols are integral to maintaining shared memory and a critical subsystem within modern multicore systems.

4.2.1 Coherence Protocols

Multi-processor systems incorporate cache coherence protocols to ensure the coherency of data stored in the processor's private caches. All communication between cores and main memory

conforms to the coherence protocol, making it an ideal attack target for a Trojan co-located with a processor's private caches. In this location, a Trojan can undetectably snoop on the request or response messages made by other processors, manipulate those messages, or even generate messages without incurring exceptions and, thus, remaining invisible to software running in the system.

Prior work demonstrates how software-based attacks can exploit coherence protocols to leak sensitive information [86, 87, 88]. While such attacks represent a significant threat, a Trojan can enable more powerful attack vectors as it is not dependent on software execution and can remain stealthy or undetectable by software.

Here we target the *MOESI Hammer* [89] coherence protocol, a hybrid broadcast-directory system. MOESI Hammer provides a coarse-grained directory at each memory controller that selectively broadcasts or unicasts messages depending on the requested memory's state. This design allows MOESI Hammer to have a directory protocol's scalability without the complexity of traditional directory protocols. Unlike conventional directory protocols, MOESI Hammer also captures the low-latency response time of broadcast protocols without incurring their typical network traffic overhead. Though our focus is MOESI Hammer, our attack scenarios can easily be ported to other coherence schemes.

4.2.2 Basic Trojan Attacks on Coherence Systems

First, we discuss a set of basic Trojan attacks, wherein we attempt to map prior work's Trojan threat classification of *passive reading*, *masquerading*, *modifying*, and *diverting* attacks [85] to cache coherence. We demonstrate the feasibility of these basic attacks by introducing the GETX*spy* attack, which uses a snooping Trojan to observe messages from a conspiring spy process. These basic attacks can adversely affect the system but are incapable of complex interactions resulting in an attacker gaining full control over the memory system. We then propose a novel, more sophisticated, and powerful attack which integrates several of these basic attacks to maliciously modify data belonging to another core, even on a different chiplet from the Trojan.

Figures 4.4-4.3 illustrates the basic coherence attacks. We assume Trojans are placed at a

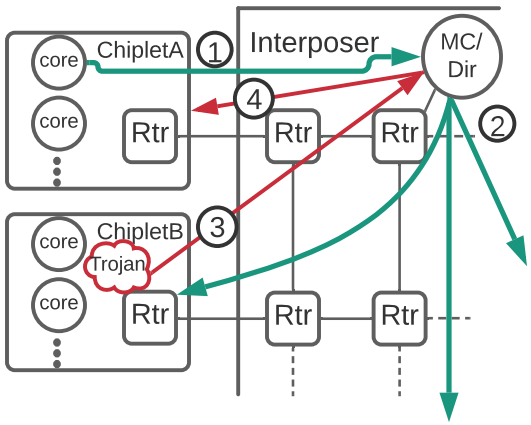


Figure 4.1: **Masquerading:** Trojan acts as another core. **(1)** Miss causes GETX to directory; **(2)** broadcast invalidations to each chiplet; **(3)** Trojan blocks local observation, replies with different core ID; **(4)** requesting core proceeds, leaving local caches incoherent.

core's cache controller and can intercept coherence messages from the network interface ahead of the state directory.

Masquerading(Fig. 4.1): Masquerading, or spoofing, occurs when a Trojan modifies the packet's `sender` field such that the packet appears as if it originates from a different core. If the target packet is a request, such an attack can result in a deadlock since all responses from the directory or other cores are sent to the incorrect core. If the target packet is a response, the Trojan may block it and respond with an acknowledgment that appears to be from a different core, resulting in an incoherent memory state.

Modification(Fig. 4.2): Such attacks occur when the Trojan directly modifies the `message type` of a coherence message. This attack may result in a deadlock since the Trojan may cause the memory controller's directory to assume the data is in one state, due to a modified packet, while the local directory holds the data in a different—incorrect—state.

Diverting(Fig. 4.3): Trojans can launch diverting attacks by blocking the local state directory from observing a request and then resending the request with a different `destination` field. This results in the compromised core and the original requestor becoming incoherent with respect to the rest of the memory system.

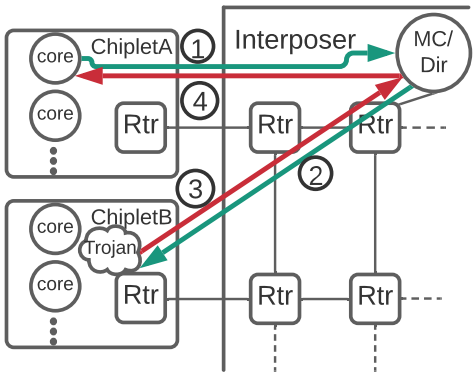


Figure 4.2: **Modifying:** Trojan modifies a message to achieve incoherent state. **(1)** Chiplet A sends GETS to directory; **(2)** directory forwards request to Trojan's core which has line in 'E' state. Trojan blocks GETS and **(3)** replies with GETX to requestor, **(4)** invalidating Chiplet A's cache entry, leaving attacker in control of another cache's contents.

Passive Reading (Fig. 4.4): Trojans passively reading, or snooping, observe incoming coherence messages from the chiplet's network-on-chip (NoC) sub-system as they reach the L2's state directory. The Trojan may buffer messages, identify specific request patterns, and facilitate a covert communication channel. The Trojan does not affect the system's state but may activate/trigger a more complex Trojan.

Each of these attacks may cause coherence or allow an attack to exfiltrate information related to the system's operation. We demonstrate the feasibility of these fundamental attacks, in particular passive reading, in a coherent chiplet system.

4.2.3 The GETX*spy* Attack

Here we introduce and demonstrate a new, fundamental, attack exploiting the coherence mechanism through hardware Trojans in untrusted chiplets. Specifically, our attack a) can transmit any data from one chiplet, via a regular user process acting as *spy* that generates tailored write-ownership coherence messages (GETX), and b) employs a hardware Trojan in a compromised chiplet that passively reads those GETX requests.

We call the attack GETX*spy* as it relies on GETX requests generated by the spy. No prior security scheme we are aware of can prevent this kind of attack. That is because GETX*spy* observes

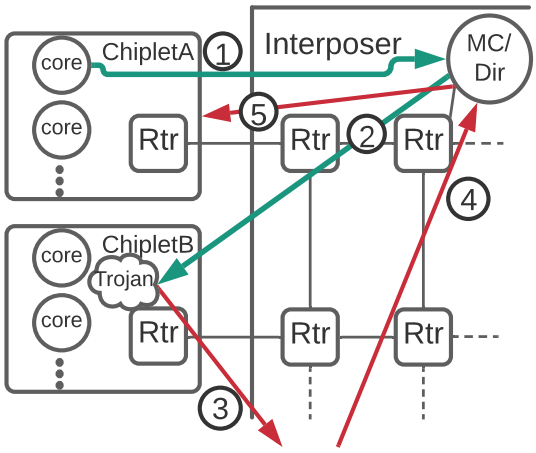


Figure 4.3: **Diverting:** Trojan diverts invalidation requests. **(1)** Chiplet A sends GETX to the directory; **(2)** directory broadcasts invalidations. **(3)** Trojan blocks message and diverts a request to another core, **(4)** which responds with a negative-acknowledge or acknowledgment resulting in **(5)** the directory allowing original requestor to continue.

the addresses of legal invalidation messages; it does not violate the system's coherence protocol, evading the defense mechanisms of prior work.

While the demonstration is specific to MOESI Hammer, the working principle can be applied to various broadcast or directory protocols in interposer-based systems.

4.2.3.1 Working Principle

MOESI Hammer (Sec. 4.2.1) uses a coarse-grained directory distributed between multiple memory controllers (MCs). Each core has its own local directory to maintain coherence. When an MC directory receives a GETX request without an existing entry, a broadcast message is sent to all cores. This expected interaction can be used to create a simple covert-channel between a spy process and a hardware Trojan placed at the cache controller directory in one of a chiplet's cores to receive information via broadcasted GETX messages.

While our attack exploits the coherence state of specific addresses, similar to Yao *et al.* [90], it differs in some important aspects. First and foremost, GETX*spy* does not require the spy and Trojan to operate within the same virtual address space. Second, our covert-channel does not rely on a Trojan process to query the targeted addresses. Third, our attack is not reliant on timing memory

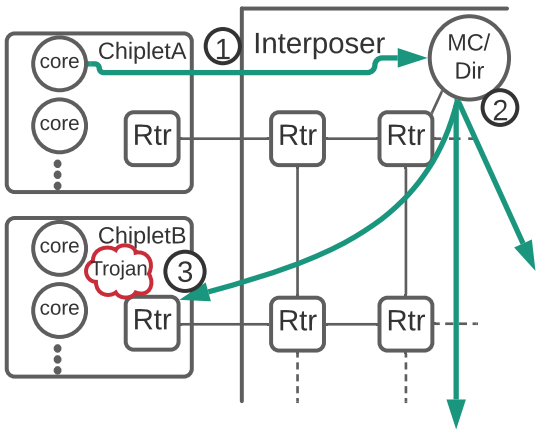


Figure 4.4: **Passive Reading:** Trojan passively observes write traffic for other chiplets. **(1)** Misses from Chiplet A cause **(2)** broadcast invalidations to all chiplets; **(3)** Trojan snoops invalidation addresses.

accesses. Finally, our Trojan is simply a malicious observer of memory requests, representing a realistic and concerning scenario that is hard to mitigate.

Figure 4.5 shows the attack orchestration. **(1)** The spy process allocates a large memory region to continually cause remote requests without pausing to flush the L2. **(2)** The spy writes to targeted sets, causing misses in the L2. **(3)** Each miss generates a new GETX request. **(4)** The GETX is sent to the MC to check for a directory entry or "hit" in the probe filter. **(5)** The GETX misses in the MC, resulting in a broadcast GETX to invalidate any shared copies of the data present in other cores. **(6)** The chiplet containing the hardware Trojan receives the broadcasted GETX, which buffers the request. Using the L2 set index bits, the Trojan checks if a synchronization message has been received. **(7)** After synchronization, the Trojan observes GETX requests from the spy process to receive covert messages.

Critically, the chiplet holding the Trojan (Chiplet 8 here) does not need shared access to the spy process's virtual address range, as the coherence protocol mandates GETX requests be broadcast to all cores, *regardless of physical page ownership*. This attack does not require priming memory region or flushing the caches before a new transmission.

GETX*spy* can be reworked into a side-channel attack: the GETX*spy* Trojan would passively

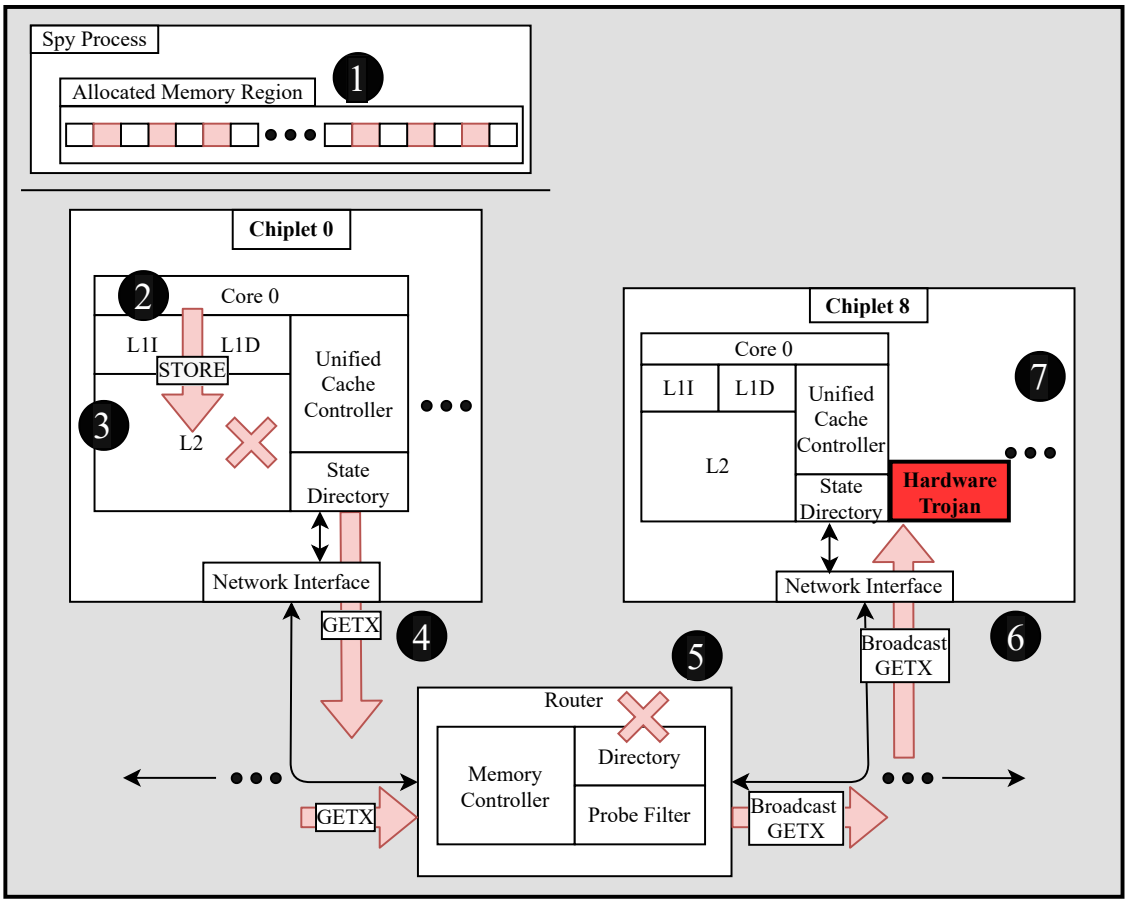


Figure 4.5: The GETX*spy* attack, executed as spy process in Chiplet 0's core 0, sending covert-channel messages to the hardware Trojan located in Chiplet 8's core 0.

watch for GETX-induced, invalidation broadcasts, to spy on the write address patterns of processes in other chiplets. Here again, the chiplet containing the Trojan need not have any access to the virtual address space or physical pages of the processes being spied upon.

4.2.3.2 Target System

We demonstrate the effects of the GETX*spy* on a 64-core processor with eight chiplets (eight cores per chiplet), based on the Rocket-64 architecture proposed by Kim et al. [91]. Each core has a private L1 instruction and data cache, and a unified L2 cache per chiplet. An NoC connects each chiplet and four memory controllers that each maintain a portion of the global state directory. The cache controllers generate coherence messages that the network interface in each chiplet then

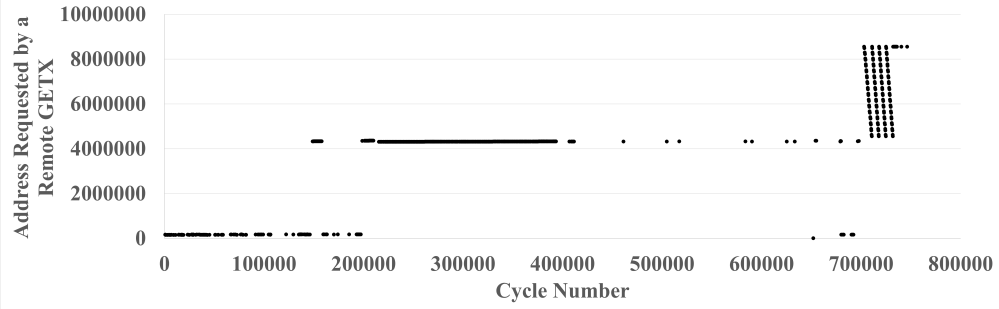


Figure 4.6: Addresses the hardware Trojan sees, as GETX requested from the spy process. The attack occurs later in execution when the spy targets specific addresses to trigger misses in the L2 and the MC's directory.

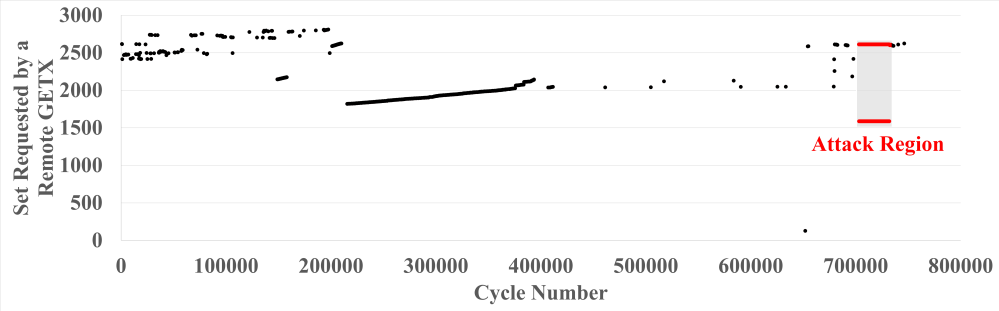


Figure 4.7: As the Trojan observes addresses requested, it awaits a synchronization message pattern, labelled as "Attack Region." This message pattern means the spy begins a transmission.

converts to network packets.

4.2.3.3 GETXspy Case Study

We implement GETX*spy* on the system described in Sec. 4.2.3.2. The system has an 8-way associative L2 cache and 4-way associative MC directory. Thus, targeting 32 addresses, 16 for each set representing '1' or '0' to transmit, allows for continuous flushing of the L2 target sets.

The evaluation setting is described in Sec. 5.6.1. As discussed in Section 5.1.1, this case study is done in simulation, since it requires a known hardware Trojan embedded within a chiplet.

Figure 4.6 shows the addresses requested by the spy process via GETX, as seen by the Trojan within a different core and chiplet. The sets referenced by the addresses are shown in Fig. 4.7.

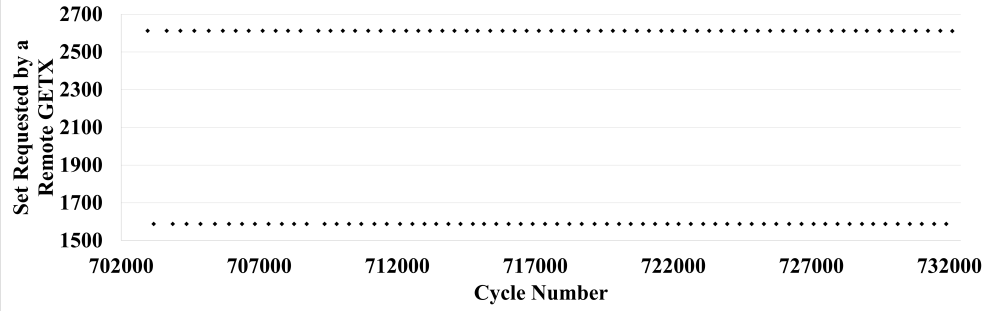


Figure 4.8: The attack region is zoomed-in here, showing the sets the Trojan considers as part of a synchronization message. The higher set represents '1' bits and the lower set represents '0' bits.

Message Size	128 bits
Cycles Taken to Transmit	28924
Clock Frequency	1GHz
Total Time Taken to Transmit	28.92μs
Megabits per Second	4.22
Percentage of NoC Bandwidth	0.013%

Table 4.1: GETX*spy* Covert-Channel Characteristics

At first, the spy process performs various memory-allocation requests, which are viewed by the Trojan as irrelevant. The attack region considered by the Trojan, Fig. 4.8, shows the spy later sending requests between two distinct sets to represent a '1' or '0,' respectively.

Table 4.1 shows the characteristics of the GETX*spy* attack's covert-channel for the unsecured baseline system. With a 4.22 Mbps bandwidth, GETX*spy* has similar capabilities as recent work that targets the coherence system [92] and is substantially higher than other prior cache-based [93, 94, 95, 96] and coherence-oriented [90, 97] side-channel attacks.

4.2.4 Limitations of Basic Attacks

Any of the above attacks can individually result in covert side-channels, side-channels, incoherence, or deadlocks but cannot directly manipulate another core's data. Combining these attacks allows for a more complex set of attack vectors that would enable a Trojan to pose a significant security threat. In the next section, we use the GETX*spy* attack as a stage within a more complex

hardware Trojan to demonstrate the threat these attacks pose when orchestrated.

4.3 Multistage Complex Hardware Trojans

This section describes a novel attack using the basic coherence attack vectors above to maliciously manipulate data in memory that the Trojan does not have permission to access. We refer to this attack as the *Forging Attack*.

4.3.1 Target System

This attack targets the same hardware system described in 4.2.3.2. Again, we assume that the victim process runs in an uncompromised chiplet, separate from the Trojan. The victim allocates an array of 64-bit unsigned integers and then performs intermittent accesses to the array, setting each value to '0' or '1'. The Trojan aims to modify the contents of this array despite the page not being mapped into the memory space of any process on any core in the Trojan's chiplet.

4.3.2 Working Principle

The *Forging Attack* manipulates legal coherence transactions to allow the Trojan to write to a target address in a different process operating in a different chiplet. The compromised chiplet containing the Trojan does not have access to the victim process' address space, but can observe coherence interactions broadcasted by the MOESI Hammer protocol. As the Trojan resides between the network interface and a target core's state directory, the Trojan has a complete view of incoming or outgoing coherence messages, thereby enabling the Trojan to block the core from observing specific interactions. The Trojan contains three registers to track the target data's current state relative to the Trojan. These registers imitate the core's state directory to ensure the Trojan correctly responds to the global directory.

- **Target Address Register:** 64-bit register containing the address the Trojan targets.
- **Response Counter:** A 6-bit register that counts the number of responses received once the Trojan issues its request.

- **State Register:** The target address' state, 2-bits to track when the data moves between expected states.

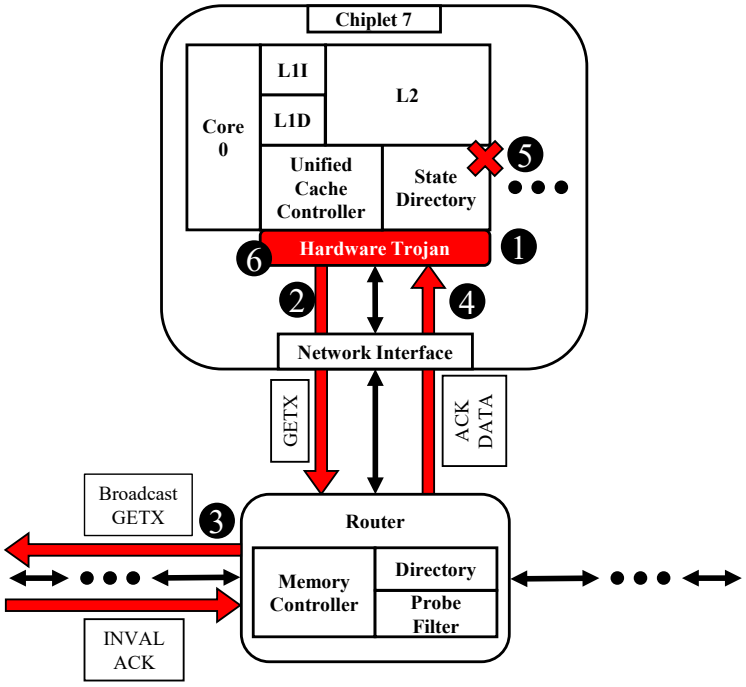


Figure 4.9: Phase 1 of the Forging attack in which the Trojan gains control of the target address.

4.3.3 Operation

Here we assume the Trojan has a predefined target address. In a real-world scenario, the Trojan can observe coherence messages broadcasted to the compromised chiplet of the network to select its target. The coherence protocol requires that the global directory sends invalidation messages each time a core sends a write request, or GETX, to a line that it does not own. The invalidation broadcast removes all copies in other cores before updating the line with new data.

The Trojan operates in two phases. During the first phase, the Trojan deceives the global directory into giving the Trojan access to the data. During the second phase, the Trojan follows the protocol's required transactions to write to the target address, which the victim will later read.

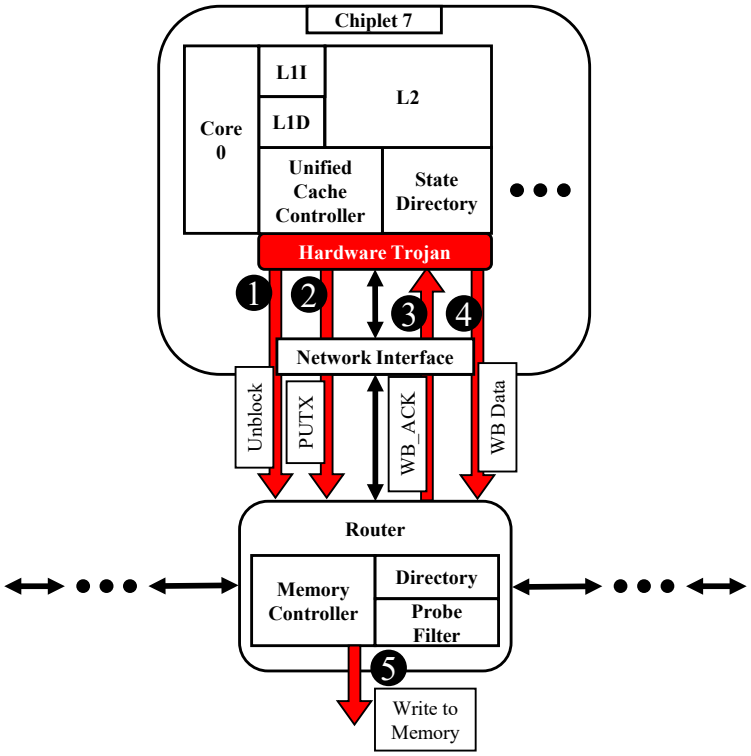


Figure 4.10: Phase 2 of the Forging attack that enables the Trojan to mimic the steps required to write back maliciously formed data to main memory.

The interactions caused by the Trojan in both phases are legal from the perspective of the global directory. Furthermore, they are transparent to the software executing in the victim process and all other security software in the system.

Phase 1: Acquiring Access Permissions to the Target Data: Figure 4.9 illustrates the initial steps the Trojan takes. These steps are required as the Trojan must first gain access permissions to the target address before it can maliciously write to it. The steps to gain access permissions are as follows: (1) The Trojan observes coherence requests, waiting for a specific address to trigger the attack (similar to GETX*spy*). (2) The Trojan generates a malicious GETX packet to the target address. (3) The directory receives the GETX request, broadcasts an invalidation to all cores, and waits for all cores to acknowledge. (4) The directory forwards the data and all acknowledgments to the compromised core. (5) The Trojan blocks the local directory from seeing any response from the directory or cores, waiting to receive all acknowledgments. (6) Once all acknowledgments are

received, the Trojan has access to the data, and the directory views the compromised core as the owner of the data.

Phase 2: Writing the Malicious Data: Once the access permissions are acquired, the global directory assumes that the Trojan's core is the exclusive owner of the data. Figure 4.10 illustrates Phase 2 of the attack. This phase allows the Trojan to mimic the legal operations that enable writing to main memory as if the core was evicting the data after modifying it. The steps of the attack are as follows: (1) Once the Trojan receives the final ACK, the requests to the target address are unblocked. (2) The Trojan immediately sends a PUTX to the directory to indicate that it is "evicting" modified data. (3) The directory responds with a `WRITEBACK_ACKNOWLEDGEMENT`, allowing the Trojan to proceed with "evicting" the maliciously changed dirty data. (4) The Trojan responds to the `WRITEBACK_ACKNOWLEDGEMENT` with a `WRITEBACK_EXCLUSIVE_DIRTY` response containing the malicious data. (5) The data is written to memory.

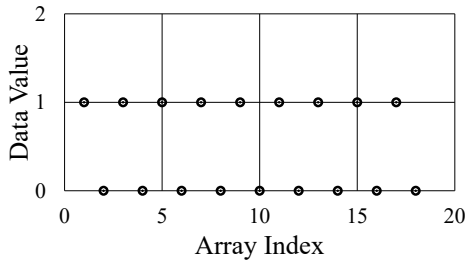


Figure 4.11: Data received by the victim when the Trojan is **not** activated. The application reads an alternating sequence of '1' and '0.'

4.3.4 Results

We evaluate the Trojans in *gem5*, targeting a victim which iterates over an array to set each value to '1' or '0' and then reads the array to compute a sum. Figure 4.11 shows the data the victim process observes without the Trojan enabled. The victim writes '0' or '1' to various locations in its data array and then re-reads these locations, seeing the expected data values. Figure 4.12 shows the data the victim receives when it attempts to read the data array after writing to all indexes.

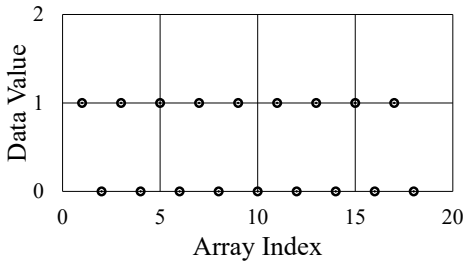


Figure 4.12: Data received after the Trojan has completed its attack. The first entry in the array is now set to '5,' instead of the expected '1.'

The *Forging Attack* successfully modifies the data array's first value, which the victim then reads unknowingly of the manipulation. This demonstrates our Trojan can manipulate the coherence system to modify data that another application is operating on, even without requiring shared memory access. Critically, unlike prior work which focuses on Trojans modifying packets [98, 99, 100] we leverage the coherence mechanism itself to modify data in memory that was never touched and technically not owned by the chiplet containing the Trojan.

4.4 Summary

As industry moves toward chiplet-based designs, hardware Trojans pose a significant threat to security. These systems will rely heavily on coherence to ensure that data remains up-to-date in all components, making the coherence protocol an attractive target. Critically, unlike prior work which focuses only on packet modifications, we show that a coherence-centric Trojan attack can modify memory that is not even owned by the chiplet. We provide an example of a complex Trojan implementation capable of modifying memory without relying on any malicious software components. This chapter highlights the need for mechanisms that can protect the coherence scheme from such attacks.

5. COHERENCE COUNTERMEASURES IN INTERPOSER-BASED SYSTEMS [1]

In this chapter, we propose a security monitor system to thwart potential Trojans targeting the coherence subsystem and operating in untrusted chiplets. Future 2.5D devices follow a design flow in which various IPs are sourced from multiple vendors and delivered as chiplets to be connected via an active interposer. We propose constructing the active interposer and integrating the chiplets at a trusted facility, making the active interposer a secure-by-construction root of trust to build Coherence Message Checkers (CMC) which monitor and prevent malicious coherence-oriented behaviors from connected chiplets. We first provide a background of active interposers and hardware Trojans. Then we discuss the threat model of the system. Following this, we provide the characteristics of the active interposer, which make it a suitable basis for creating a root of trust. Once we establish the active interposer as the root-of-trust (RoT), we present our secure active interposer design and the security features that enable the detection and prevention of coherence-based Trojan attacks.

5.1 Introduction

Interposer-based systems are vulnerable to not only traditional attacks, but also a range of dedicated, new attacks. For example, vulnerabilities may be introduced through various third-party chiplets,[2] e.g., via untrusted fabrication [102] or design of chiplets, malicious or simply buggy third-party IPs [103] within the chiplets, or collusion of multiple malicious actors across chiplets. If not addressed properly, the vulnerability of a single chiplet may undermine the entire system's security.

Coherence is an essential mechanism which ensures all components maintain a consistent view

[1] Reprinted with permission from "Coherence Attacks and Countermeasures in Interposer-Based Systems" by G. Chacon, T. Mandal, J. Knechtel, O. Sinanoglu, P. Gratz, V. Soteriou 2021. arXiv.

[2] As of today, both the chiplets and interposer are typically manufactured by the same company in the same fab. That said, we expect that in the near future there will be a strong market of varied chiplet vendors selling mix-and-match components manufactured in various fabs for third-party 2.5D integration, similar to the existing motherboard manufacturing industry today. This is also the key paradigm for the DARPA CHIPS program[101], among others, on future chip design and manufacturing.

of the system's memory, not only for interposer-based systems but interconnected SoCs in general. The predictability and prevalence of coherence systems makes them an attractive target, yet only few attacks have been proposed [90, 104, 54, 97].

Importantly, a Trojan can obviate existing memory protection hardware and software by directly manipulating the coherence scheme [54] (Ch. 4). Integrating defenses into the coherence system is a difficult task requiring extensive verification and design effort. Defenses that (naively) interact with the coherence system may cause functional bugs and deadlocks. Whereas defenses that ignore coherence and operate only on lower-level packet protocols may miss potential attacks.

While prior work in *secure network-on-chip (NoC) fabrics* consider untrusted IP modules, they do not address the full scope of a coherence-oriented attack. These defenses are generally limited to the detection of attacks, limited to a single class of attack [105, 106], fail to prevent attacks against coherence-system interactions [107, 108, 109], require additional complex hardware [108, 110, 111], or require packet authentication through error-correction codes [112] or key exchanges [113, 114, 115] which increases network bandwidth pressure.

Critically, in an interposer-based system where the interposer's design may be trusted and manufactured in a trusted fab, prior schemes for secure NoCs are tackling the wrong part of the problem. Instead of trying to secure each part of the NoC at a low, link level, we propose a defensive strategy that targets the coherence-level communication from the untrusted chiplets directly at their interface with the trusted system.

Establishing some *root of trust* is critical to ensure the security and integrity of data in modern systems containing various third-party IP components and software applications interacting on the same platform. Commercial solutions such as ARM's TrustZone [116] and Intel's SGX [117], as well as academic proposals [114, 118, 119, 120], typically rely on dedicated microarchitectural support and other measures, e.g., memory encryption. These approaches often incur high performance and storage overheads and are prone to dedicated attacks [121, 122], while being susceptible to hardware Trojans throughout the outsourced supply chains [123, 124], a fact often overlooked in prior art. Further, these schemes do little to address Trojans that interface with the coherence

system directly like the basic attacks, the GETX*spy* attack, and *Forging* attack demonstrated in Chapter 4.

5.1.1 Security Promise, Our Contributions

We leverage the notion of interposer-based system design to establish a secure-by-construction root of trust in modern multi-core, multi-chiplet systems. Importantly, unlike prior art for secure system design, *we do not assume/require trusted manufacturing of the whole system, only of the interposer, to provide system-level security promises.*

The contributions of our work are as follows:

1. We propose an active interposer as the physical backbone for a secure-by-construction root of trust, including a security-centric interconnect fabric, for multi-chiplet systems to protect against coherence-oriented threats targeting system-level communication.

2. We introduce a novel microarchitecture to secure communication passing from untrusted chiplets onto the interposer, and thus into the system, based upon per-packet validation at the interposer ingress links. Our design does not interfere with the system's underlying coherence protocol, but rather prevents sensitive information from being divulged to, or manipulated by, untrusted chiplets. *The key objective of our proposal is to realize a secure large-scale system out of untrusted chiplets.*

3. As existing hardware does not contain known Trojans for experimentation, we implement and evaluate our proposed technique in *gem5* and examine the implications of our security features. We characterize the performance impact as a low, $\sim$4% overhead. Further, we show the overhead decreases as workloads scale.

5.2 Background and Contributions

Here, we review key concepts of interposer technology, hardware security, and cache coherence protocols.

We also motivate the contributions of our work considering the security challenges and promises for the respective state-of-the-art.

5.2.1 Interposer Technology

Interposer technology, also known as 2.5D integration, is the process of manufacturing two or more chips, or chiplets, separately and subsequently integrating and interconnecting them using a carrier made of silicon or other materials [37, 38, 39, 40, 41, 42]. Compared to traditional, monolithic SoC designs, 2.5D integration drastically reduces time to market. A system designer can procure IP as commodity chiplets and integrate them at the physical system level, with effort only required for designing the interposer. 2.5D integration is highly desirable as it allows for design and manufacturing process optimization, increasing yield for chiplets.

While future 2.5D designs may be more heterogeneous, current state-of-the-art systems are largely homogeneous, cache-coherent, multi-core chiplet designs [37, 45, 38, 39, 40].

Active interposers contain active devices (e.g., NoC routers, voltage regulators, sensors, etc.), while passive interposers act solely as an integration carrier and wiring medium. Although passive interposers are cheap to manufacture, their physical design can be quite challenging [125, 38]. In contrast to active interposers with buffered interconnects, passive interposer wires incur significant power and delay overheads.

An active interposer with an embedded NoC fabric serves well for large-scale chiplet integration and system communication. The chiplet interconnect fabric is encapsulated away from the interposer NoC beyond the edge router on the interposer to which it is attached. Such heterogeneous fabric allows for cross-optimization of topologies across chiplets and interposer, opening up considerable opportunities for system design [125, 37, 45, 126, 127, 128]. Further, active interposers improve testability [42, 129, 37] and thereby help to manage the yield of the final system.

Active interposers are typically manufactured in relatively older nodes [37, 130]. Therefore, it is realistic that a trusted facility is available for manufacturing of such active interposers. *Here we propose an active interposer-based root of trust with security features embedded within its NoC routers.*

5.2.2 Hardware Security

5.2.2.1 IC Manufacturing

Industry has widely adopted a work mode where IC design and verification is carried out by a design house and partners, but fabrication and testing is outsourced to off-shore facilities typically providing access to advanced technologies.

While this practice reduces the cost of production and streamlines the time to market [131], it raises concerns regarding the trustworthiness of the outsourced fabrication facilities, which may seek to insert security vulnerabilities in general or hardware Trojans in particular [102].

In other words, the threat vector posed by untrusted fabrication facilities implies the ICs they manufacture are untrustworthy. This causes a security challenge for modern systems in multiple ways. First, any hardware security feature embedded in such outsourced IC may no longer offer the desired protection, presenting a profound challenge. Second, a modern system may be composed of chiplets with various levels of trustworthiness. Any malicious chiplet behavior may compromise the whole system due to its interconnected nature.

The interposer technology can help to avoid such complications. This is because an interposer can be fabricated separately in a trusted facility and may also embed security features. *As we show in this chapter, an interposer designed to constitute a hardware-enforced root of trust can be built upon to ensure the overall system's trustworthiness.*

5.2.2.2 Hardware Trojans

Hardware-centric attacks such as the malicious insertion or modifications of circuitry, also known as hardware Trojans, can lead to catastrophic security failures within a system. For example, Bidmeshki et al. [53] provide an attack scenario wherein a hardware Trojan renders the cryptography subsystem vulnerable, Khan et al. [52] demonstrate Trojans that can leak data from cache memory of processors, and Kim et al. [54] introduce Trojans which inject malicious coherence messages to create a denial-of-service attack.

Our work is orthogonal to and compatible with prior art on Trojan detection and mitigation,

e.g., [132, 133, 134]. Here we do not seek to prevent Trojans, but their attacks from affecting the system-level security. Specifically, we seek to prevent hardware-centric attacks from executing through the memory and coherence system. This notion of system-level security is enforced by a clear physical separation of untrusted commodity chiplets and security features residing in the trusted interposer. *Prior art on Trojan detection and mitigation cannot offer such secure-by-construction organization as ours.*

5.2.2.3 Secure Interconnect Fabrics

Prior art for secure NoCs assumes that malicious activities arise from connected components or the network fabric itself. Fiorin et al. [135] propose security features for policy-based message checking against untrusted components. Selected works focus on securing the system through encryption/decryption of packets exchanged through NoC fabrics [115, 136]. Kinsy et al. [114] propose organizing secure and non-secure software/hardware entities as tenants and configure the NoC routers to securely exchange messages. The amount of key exchanges required to isolate nodes/tenants incurs high latencies and is not easily scaled.

Nabeel et al. [130] propose an interposer-based architecture where security modules monitor the interconnect fabric at the level of bus addressing, to block transactions that violate memory access policies. While their design represents a relevant first work toward secure 2.5D integration, it has several limitations. First, the authors consider an overly simplistic architecture, ignoring the fact that state-of-the-art 2.5D designs are fully memory-mapped and cache-coherent. We find addressing the coherence model is critical to providing system-level security. Second, the authors overlook new security challenges arising for interposer designs. Critically, their design would fail to hinder the GETX*spy* covert/side-channel attack studied in the previous chapter, as GETX*spy* does not violate memory access policies/permissions.

For most prior art, networks are not secure-by-construction, hence high-overhead solutions are required such as key-based security [113, 107, 106], model checking [108, 137], or additional structures to verify traffic patterns [108, 110, 111]. While packet-checking schemes similar to our design have been proposed in the past, e.g., [109], the underlying defense mechanisms of-

ten address only a single attack vector [105, 106] and/or fail to address the coherence system's exploitable nature [107, 108, 109].

While these works check the message's memory operation, they do not differentiate between specific coherence message types and how coherence messages can be exploited beyond simple read or write traffic. Even more concerning, most prior art assumes, often implicitly, trusted manufacturing of the whole system. Outsourced supply chains challenge such an assumption. These concerns are only exacerbated for 2.5 integration using chiplets from various vendors.

In contrast, our work does not make such overarching assumptions. *We enforce system-level security for untrusted commodity chiplets by integrating them on an interposer-based root of trust, the only component requiring trusted fabrication, thereby providing a secure-by-construction NoC.* Without the need to secure the integrity of the NoC, a more simplified approach can be taken to ensure the overall system's security, resulting in lower overheads.

5.2.2.4 Hardware Support for Root of Trust

Intel's SGX provides trusted execution environments (TEEs), called enclaves [117, 138]. Enclaves prevent unprivileged access to secure data during security-sensitive execution. Specifically, SGX maps protected memory pages to reserved memory regions in which the pages are encrypted by a hardware encryption module.

ARM's hardware-enforced TEE isolates secure execution from untrusted software [116]. AMD's TEE leverages a normal OS running in tandem with a secure OS. The latter has access to the full range of a device's peripherals and memory, whereas the normal OS only has access to a subset of peripherals and memory regions, to prevent unauthorized access of sensitive resources. Both schemes have considerable impact on system performance.

Recent works have shown vulnerabilities in SGX, due to programming errors and untrusted software [139, 140], as well as due to speculative execution [141, 142, 143, 144]. Similarly, TEEs are prone to vulnerabilities due to architectural, implementation, and hardware issues [145].

In contrast, *our approach has little impact on system performance and its key components are secure-by-construction.*

5.2.3 Cache Coherence

Coherence protocols ensure updates to cached copies of data are visible to all cores and other IP blocks in modern multi-core designs [45, 37, 40]. Coherence schemes can be broadly categorized as broadcast (or snooping) protocols [46, 47, 48] and directory protocols [49, 50, 51]. While simple to implement, broadcast protocols suffer from high traffic due to the amount of messages multi-core systems require to maintain coherence. Directory protocols allow for fine-grained state tracking and unicast messages, making them highly scalable but difficult to implement and have higher access latencies.

A coherence protocol is generally oblivious to the software and may permit malicious accesses that leak sensitive information [90, 104]. Existing countermeasures address conflict-based and transient-execution side-channel attacks, but do not consider threats from maliciously manipulated/malformed coherence message packets [146, 147, 148, 149].

Given that coherence protocols act only based on rules for how memory is updated across multiple parties, attackers may exploit the protocol's low-level behavior. We demonstrate one such attack in Sec. 4.2.3. It is important to note that coherence is a hardware-managed, micro-architectural feature which is neither influenced by, nor exposed to, the software executing on the system, rendering software-based defenses ineffective.

Our solution does not require modifying the coherence protocol. Rather than risking complex, adversarial system behavior side-effects, *we the ensure coherence messages' integrity and prevent untrustworthy chiplets from exploiting the coherence protocol and system-level memory management.*

5.3 System Architecture Overview

Figure 5.1 outlines the secure, interposer-based, multi-chiplet and multi-core system proposed in this work. The baseline system is loosely based on the architecture of the Rocket-64 design proposed by Kim et al. [38]. In addition to the overview in this section, more details are provided in Sec. 5.5.

5.3.1 Chiplet and Interconnects Architecture

In this system, we employ eight chiplets, each containing eight CPU cores, for 64 cores in total, similar to recent AMD processors [39, 40]. Each core has an L1 instruction and data cache and a unified L2 cache; all cache levels are private to each core. An NoC of 2D mesh topology residing in the active interposer interconnects the chiplets to each other and four memory controllers (MC). The cache controllers generate coherence messages that the network interface (NI) in each chiplet converts to network packets before injection into the interposer NoC (via interface routers). The interface routers, depicted along the east and west edges of the system, serve as ingress links for the chiplets into the interposer NoC.

Our proposed system can be ported to many other architectures [125, 37, 45, 126, 127, 128], given it is interposer-based and has a cache-coherent shared memory system. Notably, the security principles leveraged in our work are extendable to various physical fabrics and communication protocols in homogeneous and heterogeneous systems. For example, interfaces such as PCIe are typically memory-mapped; checking memory-system messages can prevent unauthorized access by any malicious chiplets. Although some heterogeneous systems may not fully enforce coherence, communication between processing elements, I/O, etc., is still done using memory-system messaging.

5.3.2 Principles for System-Level Security

We propose the interposer as the root of trust for integrating untrustworthy chiplets into a secure system by enforcing policy checking of all system-level communication. The key attributes to enable such a security system are: (1) the interposer is manufactured separately from the untrusted chiplets in a trusted facility, and (2) the interposer serves as the integration and communication backbone between chiplets.

Any system-level communication across chiplets must pass through the interposer. For example, if a CPU core wants to read/write data from/to memory, a corresponding coherence message (embedded in a packet) must traverse the interposer NoC. Similarly, if a core wants to communi-

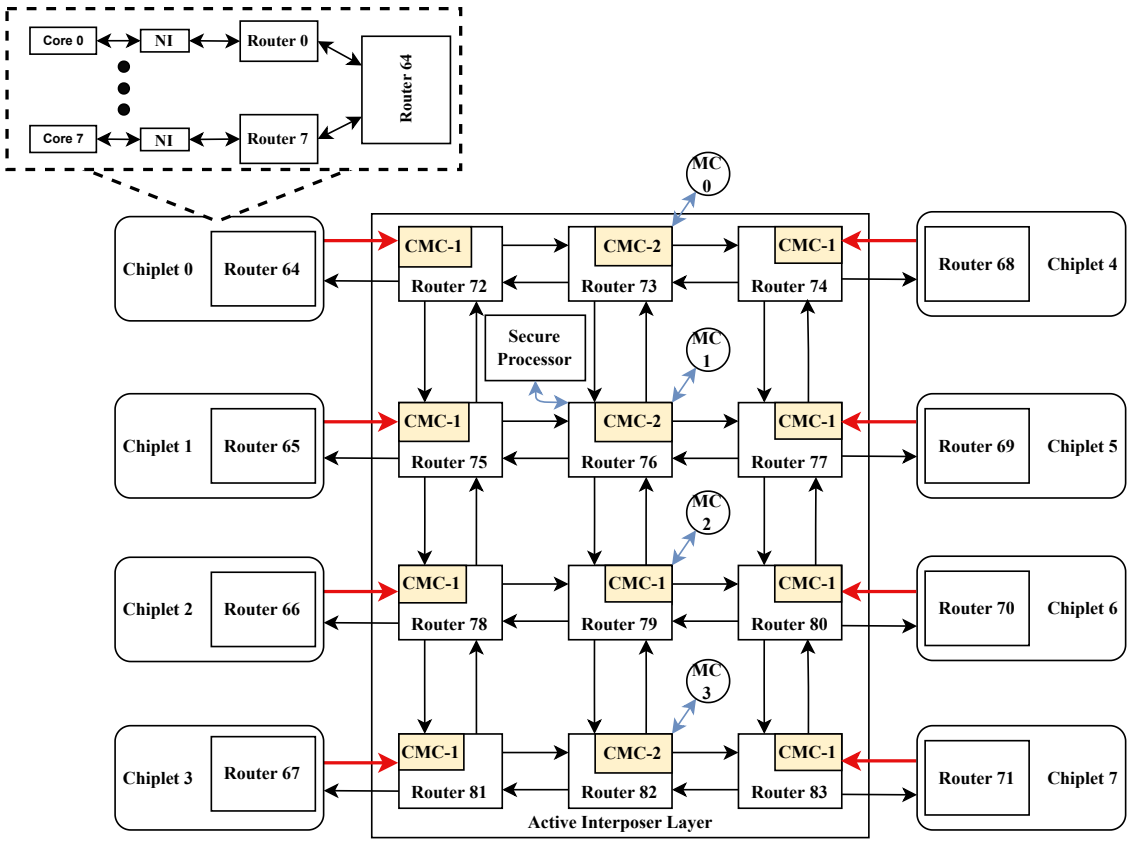


Figure 5.1: Secure system. Routers 64–71 lie within chiplets, connecting them to the interposer NoC. Routers 0–63 connect the CPU cores within their respective chiplet's NoC (see zoom-in). The proposed Coherence Message Checkers (CMCs), marked in yellow, are embedded in the interposer and placed along the ports connected to chiplets (CMC-1, red arrows) and memory controllers (CMC-2, blue arrows). Also note the secure co-processor embedded in the interposer.

cate with another core in another chiplet, such direct messages must traverse the interposer NoC. Significantly, *all direct communication messages are limited to legal coherence messages*, as is typical in most multi-processor systems.

Accordingly, we embed our proposed security features within the interposer NoC such that all messages must inevitably traverse through, and be checked by, the trusted active interposer. For example, we add *Coherence Message Checkers (CMCs)* to the physical ingress links to validate all coherence messages coming from the chiplets into the active interposer. For another example, we add a secure co-processor for critical tasks including system-level memory allocation. *Since CMCs and all other security features are implemented exclusively within the trusted active interposer,*

their hardware is trustworthy and free from Trojans by construction.

5.3.3 Cache Coherence Protocol

We focus on the *MOESI Hammer* cache coherence protocol [89] as basis for our implementation, which is used in many AMD systems as scalable protocol for multi-core systems. Our approach is extendable to other schemes as well.

MOESI Hammer is a hybrid protocol; it encapsulates the scalability of directory-protocols without high implementation complexity while achieving the low-latency of broadcast protocols without overly increasing broadcasted coherence message traffic. To that end, MOESI Hammer maintains a sparse directory between multiple home nodes to track cache lines' states and owners. Coherence requests access a cache line's home-node directory and DRAM in parallel to reduce the cost of a directory miss, cancelling the DRAM response if a directory entry is found. Traffic is reduced by only broadcasting to all cores for specific state transitions.

5.4 Threat Model

5.4.1 Scope and Assumptions

The focus of this chapter is a system wherein multiple chiplets have been fabricated by various untrusted third parties and are then connected using an active, trusted interposer.

Our work is orthogonal to prior art on Trojan detection and mitigation for regular, non-interposer-based systems, e.g., [132, 133, 134]. That is, we do not seek to prevent Trojans but to prevent their attacks from affecting the system-level security.

The critical assumption is that the fabrication and operational behavior of chiplets, either designed in-house or composed of third-party IPs, cannot be trusted. In other words, we assume that some Trojan(s) may exist in some chiplet(s). We also assume that attacks target memory-system traffic, the only type of traffic physically passing through the interposer.

Another key assumption is that all proposed security features are designed, manufactured, and operated in a fully trustworthy manner. Furthermore, we assume a secure boot-up and OS environment for memory management tasks. Both assumptions are physically enforced, by implementing

the related hardware exclusively within the trusted interposer.

Our scheme protects on a chiplet granularity. Thus, attacks across cores but within the same chiplet [150, 151], are out of scope. Similarly, out of scope are attacks wherein code running on one core attempts to violate the security of other processes running on that same core or another core in the same chiplet. Further, attacks wherein one chiplet leverages transactions to its assigned memory region to modify DRAM rows that are not assigned to it, e.g., Rowhammer [152], are out of scope. Importantly, prior art against all these threats is orthogonal to our work and can be applied as needed.

Regarding denial-of-service attacks via memory allocation (which is securely handled by the trusted OS in the interposer), we assume that detecting maliciously excessive memory requests is dealt with otherwise. Furthermore, denial-of-service attacks resulting from some chiplet dropping coherence messages are out of scope.

5.4.2 Threat Vectors

Attacks on interconnected systems can be categorized as outlined by Basak *et al.* [153]. Accordingly, our model considers the related four threat vectors.

1. *Passive reading, aka snooping*: This threat occurs when a malicious chiplet can read data it does not have permission for. The GETX*spy* attack demonstrated in Sec. 4.2.3 is an example of such a threat in that the Trojan monitors broadcasted GETX requests to snoop a tailored message.

2. *Masquerading, aka spoofing*: This threat occurs when a malicious chiplet disguises itself as another chiplet to gain access to sensitive data or control of resources. Malicious chiplets can modify the requester IDs and memory addresses embedded in cache coherence messages, tricking directories or other unsuspecting cores into divulging sensitive data.

3. *Modifying*: This generic threats concerns any malicious modifications of coherence messages.

4. *Diverting*: In shared-memory applications, a malicious chiplet may divert data meant for one chiplet to another untrusted chiplet, bypassing memory permissions. It may also divert cache coherence messages, undermining the protocol.

The above threats apply to any form of coherence protocol with invalidations. While this chapter examines the security challenges of a hybrid broadcast/directory protocol, the system's exact attack surface depends on the protocol used. Directory protocols, while more resilient to passive-reading attacks (due to a lack of broadcasted messaging), may still be deceived by maliciously modified coherence messages. For example, to replicate our proposed GETX*spy* attack (Ch. 4.2.3) in a directory protocol, the Trojan must only be slightly modified, namely to issue regular GETS requests for the cache lines used by the *spy* process in communicating with the Trojan. When the *spy* process modifies those lines, the Trojan would be able to observe invalidation requests to the associated lines, replicating the covert channel. While such changes in settings might reduce the channel's bandwidth, they would otherwise not prevent its general working principle. For another example, both local and global directories may be targeted to interfere with the system's coherence, through masquerading and modified messages. Diverting legitimate packets from a directory to other cores allows an attacker to divulge information about a target core through probing the directory [90, 104]. Furthermore, we assume that any combination of the above basic attacks can be orchestrated into a complex attack such as our proposed *Forging attack* (Ch. 4.3).

5.5 System Design

Our proposed design prevents attacks running on any given chiplet from violating the overall system's security, in the sense that we physically enforce protection against any unauthorized access to shared-memory regions and conduct continuous checking of the integrity and validity of cache coherence messages. Next, we discuss the system design.

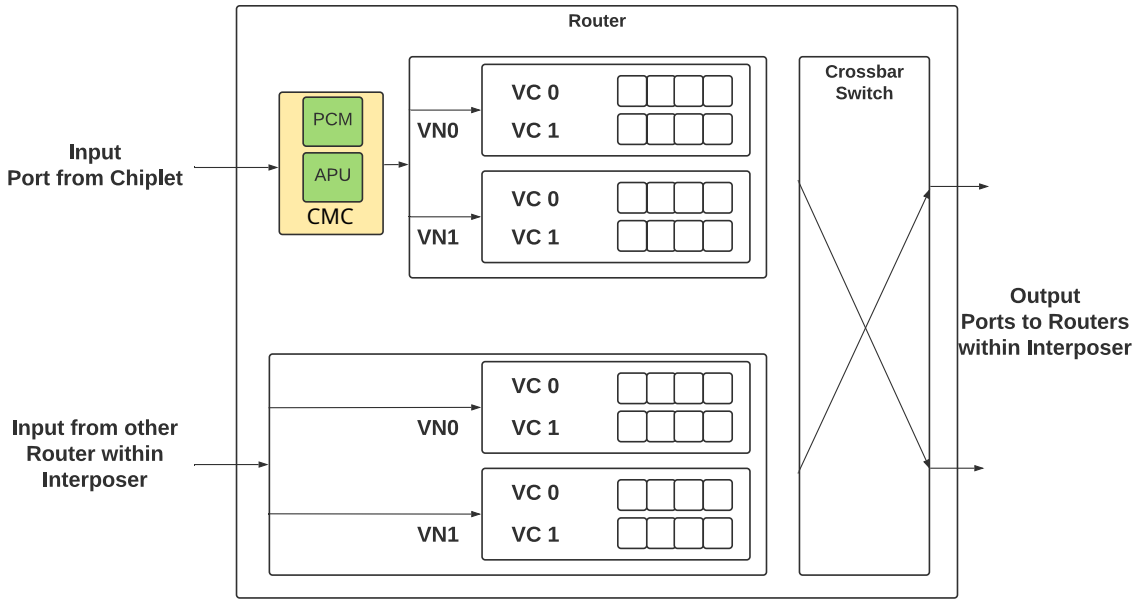


Figure 5.2: A CMC, embedded within an interface router of the interposer NoC, monitoring the incoming packets.

5.5.1 Microarchitecture

5.5.1.1 CMC Overview

With the proposed CMCs, we monitor and validate all incoming packets to the interposer. Figure 5.2 depicts the CMC embedded in a router of the interposer NoC. The CMC monitors messages traversing the physical links before entering the virtual channel buffers within the routers. Each CMC has two components described as follows.

Packet Checker/Modifier (PCM): The PCM monitors and modifies cache coherence messages as needed. Because the proposed system follows standard shared-memory semantics, all legal communication between cores, other IPs, I/O buses, and memory occurs through memory accesses, creating cache coherence messages. Thus, the PCM operates on coherence messages to check addresses and permissions, modifying messages as needed.

Address Protection Unit (APU) Table: This is a direct-mapped, SRAM-based look-up table with entries for each memory region and their associated per-chiplet permissions. As outlined in

Sec. 5.5.2, the main physical memory is partitioned into multiple fixed-size regions; each memory region has a corresponding entry in the APU.

5.5.1.2 CMC Types and Placement

Recall Fig. 5.1, depicting CMCs embedded in the secure interposer-based system. The CMCs connected to the physical links coming from chiplets are denoted as "CMC-1" and those connected to the physical links for MCs are denoted "CMC-2." CMC-1 only monitors and verifies coherence messages entering the interposer, whereas CMC-2 modifies certain coherence messages at the MC directories (to counter passive-reading threats on broadcast messages). Router-to-router connections running exclusively within the trusted interposer do not require CMC monitoring.

CMC-1: Prevents the attached chiplet from injecting malicious coherence messages into the system that violate the provisions of the shared-memory organization, as outlined in Sec. 5.5.2. The PCM within CMC-1 monitors all traffic from the attached chiplet based on the physical address the packet refers to. This physical address is compared against the per-region permissions stored in the APU table (described further below). If a message is of an allowed type to an allowed memory region for the given chiplet (e.g., a GETX to a read-only memory region it owns), the message may proceed into the interposer NoC. Otherwise, if the packet is rejected, a dedicated security signal realized as a machine-check exception is thrown, and system execution stops.[3]

CMC-2: Prevents the broadcast of coherence messages to chiplets that are not permitted to access the related memory region. As described in Sec. 5.3.3, MOESI Hammer does not maintain per-core sharing information, hence certain message requests cause an MC directory to broadcast the request to all cores. The cores then respond based on whether that core shares the cache block. This raises a concern of passive reading/snooping; recall the GETX*spy* attack in Sec. 4.2.3.

To prevent snooping, the PCM uses the APU table to determine whether a given broadcast message is directed towards a chiplet allowed to access the referred memory region. If the chiplet does not have access, the broadcast message is converted into an appropriate direct broadcast

[3]This is a secure and protocol-conform approach. For the sake of system-level throughput, one may want to only isolate the chiplet(s) triggering a security violation. Doing so safely, however, is not trivial, as it would require significant modifications of the coherence protocol itself to prevent deadlocks.

Memory Region	Chiplet 7	Chiplet 6	Chiplet 5	Chiplet 4	Chiplet 3	Chiplet 2	Chiplet 1	Chiplet 0
Entry N	00	00	00	00	00	00	11	11

Figure 5.3: Exemplary entry of the APU table, covering some region of the physical memory. The entry describes access permissions for each chiplet individually; here, the related region is read-write shared between Chiplets 0 and 1.

message directed only to the original requester. (For example, an invalidation request for data owned by Chiplet 0 does not need to be observed by any other chiplet.) Note that this approach is legal within the coherence scheme: if a chiplet is not granted access to a memory region, then its caches cannot contain lines associated with that region. This allows the CMC-2 to safely divert broadcast messages from the directory and prevent snooping.

5.5.1.3 APU Table

The APU table is a lookup table containing entries describing the access permissions for applications running within a respective chiplet.

The access permissions are determined by a secure OS that is running exclusively within the active interposer, independently of the regular OS running on the chiplets. The permissions are programmed into the APU tables during runtime, as outlined in Sec. 5.5.2.

Figure 5.3 show one entry in the APU table. Each entry represents one memory region, with two bits allocated to represent the access permissions of applications running in some chiplet: a chiplet may have no access permissions (‘00’), read-only permissions (‘01’), or read/write permissions (‘11’). The encoding ‘10’ is unused.

When the PCM intercepts a packet, the upper bits of its physical address are extracted and used to index into the APU table. The related entry is read and handed back to the PCM to compare the request type, requester ID, and destination ID against the permission levels in the APU table entry.

5.5.2 OS Support and Memory Organization

Here we extend and build upon prior work in security-enabled OS environments [154, 155, 156, 157], TEEs (Sec. 5.2.2.4), and secure boot-up and execution environments [116, 158, 159]. In our scheme, critical tasks, including updating the APU table, must be delegated to such a secure environment.

Such environment must ensure that malicious OS threads running on an untrusted chiplet are physically incapable of purposefully assigning memory regions that would result in violating the security policies. Toward this end, we use a trustworthy OS located in a co-processor embedded in the interposer, where the active interposer's construction physically prevents attacks on the APU and other security components.

5.5.2.1 Representing Memory Regions

In shared-memory systems, permissions are typically defined per physical page by the OS during memory allocation. For our system, enforcing per-page permissions in a CMC poses several challenges. Specifically, page-level tracking requires a TLB-like structure to cache translations [160]. The support required to maintain the structure in coherence with the full system's page table significantly increases hardware complexity and performance overhead. Thus, we argue that such page-level implementation at the interposer is excessive in a system of relatively few and coarse-grained chiplets, and we partition physical memory into coarse-grained memory regions,[4] similar to prior art [161, 154, 158, 157].

Each memory region is designated as read- or write-able independently to any given chiplet, with permissions updated as needed. Data private to a single chiplet is placed in a region (or set of regions) only accessible by that chiplet. A page shared across multiple chiplets is assigned to a memory region the given chiplets are allowed to access.

[4]We aim for a "sweet spot" between too coarse-grained, where only few memory regions are available and capacity is wasted to fragmentation, versus too fine-grained, where the APU table could not hold the excessive number of regions without incurring high access latency or placing entries in a backstore. We find that a total number of memory regions between 4x–8x the number of chiplets is sufficient, allowing for diverse private and shared memory regions without too much fragmentation. Thus, for our design with eight chiplets and 4GB of physical memory, we implement the APU with 64 entries, representing 64MB each or 16,384 pages each.

5.5.2.2 *Memory Allocation and OS Modifications*

The interposer includes a trusted co-processor, to host a secure and trustworthy OS for system-level memory allocation. Thus, the OS threads running on the chiplets must delegate their page allocation to the OS running on the interposer. The code in the chiplet's OS threads must be extended accordingly, i.e., to call the interposer's secure OS for all page allocation.

The interface API between the OS threads running on the chiplets and the secure OS is composed of two functions, `APU_ALLOCATE` and `APU_DELETE`. The `APU_ALLOCATE` interface function is called by the chiplet's OS when access to a new physical page is required. In that, the secure OS provides the chiplet's OS threads with a physical page based on the chiplet's current memory regions and its access permissions, as described below. Similarly, when the chiplet's OS threads are ready to free a physical page they call the `APU_DELETE` function to return that page to the secure OS.

Initial memory partitioning and permission setting occurs during the initial soft page fault on a virtual page. Region allocations and permissions are updated via an API call from the OS, e.g., similar to Intel's SGX page allocation model [117]. After a page fault, the following occurs:

1. The chiplet's OS requests a physical page for the process from the secure OS operating on the interposer's co-processor via the `APU_ALLOCATE` interface function.

2. The secure OS running in the interposer then searches for an available page with the correct permissions for the given chiplet, differentiating three scenarios:

   (a) If the chiplet already has a region allocated and assigned in the APUs, and this region has unassigned physical pages, a page from this region is selected.

   (b) If the chiplet does not have an entry in the APU or its current region is fully allocated, the interposer updates the APU tables to allocate a new region with appropriate permissions for the chiplet that requested the page. The secure OS then selects a free page from the newly allocated region.

(c) If no space is available in the assigned regions and there are no more unassigned regions the memory allocation fails.[5]

3. The secure OS then provides the allocated physical page to the unsecure OS.

Since the APU table update occurs on the trusted interposer, chiplets not involved in the allocation process are unaware of the memory allocation request. Critically, a malicious chiplet that somehow gains knowledge of the request cannot access the region due to the newly set permissions in the APU tables before any malicious operation may target the memory region.

5.5.3 Implementation Details

5.5.3.1 NoC Configuration

Regardless of the interposer's NoC topology, CMCs are emplaced at the interface between chiplets or MCs and the active interposer. However, the width of the physical link does impact the CMC design and its logic. In our implementation and evaluation, the link width is 128 bits within chiplets and 64/128 bits within the interposer.

In MOESI Hammer (Sec. 5.3.3), every control message fits within a single 128-bit flit. When a flit enters the interposer, it is broken down into two/one flits which are analyzed in the CMC logic over two/one clock cycles, depending on the 64/128 width of the interposer link. In the case of 64-bit links, depicted in Fig. 5.4, we dedicate the first cycle to extract the control parameters from the head and the second cycle to extract the address for the cache block being accessed. The CMC logic is similar for request and response messages, as both cases require the first two flits to be analyzed.

5.5.3.2 Cache Coherence Protocol

The system's cache coherence protocol affects the CMC design and logic as the CMC must analyze the coherence message fields. MOESI Hammer's response messages are either a control or data message; a control response follows the same flit structure as a request, whereas a data

[5]In future work, we may consider swapping in this scenario. However, this would increase complexity of the process and require careful investigation.

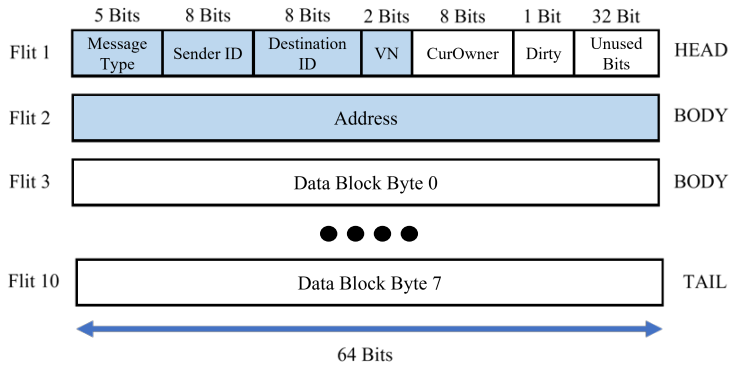


Figure 5.4: Structure of messages. Request messages do not include the 'CurOwner' or 'Dirty' fields. Flits 3-10 are only sent for response messages in response to a request message. Fields highlighted are to be checked by CMCs.

response carries additional flits containing a total of 64 bytes of data.

Based on the message type and identified threats (Sec. 5.4.2), the CMC must analyze certain key parameters; these are highlighted in Fig. 5.4. The parameters are extracted by the PCM and compared with the permissions set in the APU table. Since an attacker may exploit either request or reponse messages, both message types must be analyzed.

5.5.3.3 Protocol Compliance

First, coherence messages are converted into network packets by the chiplets' NIs. However, these packets are not guaranteed to adhere to the rules of the network and coherence protocols. For example, a Trojan may fabricate an invalid message type, yielding undefined, possibly vulnerable behavior. Second, messages corresponding to particular virtual networks (VNs) must follow a specific, limited set of requester/destination IDs and message types.

To address both aspects, the PCM checks the possible field values to verify the legality of messages. Since these checks are orthogonal to memory-region permission checking, they are performed in parallel and incur no extra delay.

5.5.3.4 Design Cost

We design the PCM module with three pipeline stages for lookup, packet checking, and packet modification. The third stage is bypassed in CMC-1 instances as they only monitor packets on

Component	Variable
Chiplet Architecture	
Core	8 RISC-V cores
Private L1 I-Cache	32KB
Private L1 D-Cache	64KB
L2 Cache	2MB
NoC	Eight-port, 128-bit Crossbar
vc_per_vnet	4, 6, 8, or 10
Chiplet Frequency	1GHz
System Architecture	
Chiplets	8
MCs	4
Main Memory	4GB
Memory Regions	64, 64MB each
NoC	3x4 2D-Mesh, 64 or 128 bit
vc_per_vnet	4, 6, 8, or 10
Interposer Frequency	250MHz
Cache Coherence	
Model	AMD MOESI Hammer
Simulation Configuration	
Processor Model	TimingSimpleCPU
Simulation Model	System emulation

Table 5.1: System Architecture Configuration

ingress to the interposer. An APU table requires two bits for identifying each chiplet's permissions, and there are 64 table entries; 1024 bits are required for an APU table. An APU table is in each of the twelve routers (8 for the chiplets, 4 for the MCs) in the interposer, imposing a total memory footprint of only 1.5KB.

5.6 Evaluation

We first discuss our evaluation methodology. Then, we examine the security coverage our design provides. Finally, we examine the performance overheads caused by our scheme.

5.6.1 Methodology

We implement and evaluate our proposed system for system emulation using gem5 [162]. Table 5.1 depicts the configuration details. As indicated, while flexible in general, the particular proof-of-concept system studied here is inspired by the Rocket-64 design [38]. Thus, we simulate

an 8-chiplet, 64-core system as described in Sec. 5.3. The interposer is assumed to be fabricated using an older process node; it operates at 250MHz, a quarter of the chiplets' frequency.

Performance impact is measured as IPC speedup/slowdown for the secure, CMC-enabled configuration over the unsecure baseline configuration. The CMCs latencies are discussed in Sec. 5.5.3 and disabled for the unsecured baseline. Due to long simulation times induced for this large system, we evaluate the IPC using a subset of the SPEC 2006 benchmarks. We perform single-threaded and multi-programmed benchmark simulations to better understand the impact of the CMCs.

5.6.2 Security Analysis

5.6.2.1 Threat Model Coverage

Our scheme addresses the threat model (Sec. 5.4.2) as follows:

Passive reading: This threat is prevented by rerouting broadcast messages as they enter the CMC-2 located at the interposer/MC boundary. Broadcast messages from the directories are converted into negative acknowledgments back to the requester for chiplets that do not have permissions to the message's memory region.

Masquerading: Every CMC-1 is programmed with the range of ID's expected in each coherence message's requester ID field. For example, in Fig. 5.1, the CMC-1 in router 72 can expect requestor IDs to be in the range of 0–7 and will reject any message with an ID outside of this range, as discussed in Sec. 5.5.3.3. In this event, the CMC will throw a security check exception and halt execution.

Modification: This is detected by comparing a message type, such as GETX/GETS, with the access permissions in the APU table. A security check exception is thrown if a message seeks to access memory outside of its allowed address space.

Diversion: This threat is detected by checking the destination ID and the message type. Only specific message types can have other cores as the destination ID. This, along with the memory region permissions in the APU table, allows us to detect any malicious diversion of messages. A security check exception is thrown if a threat is detected.

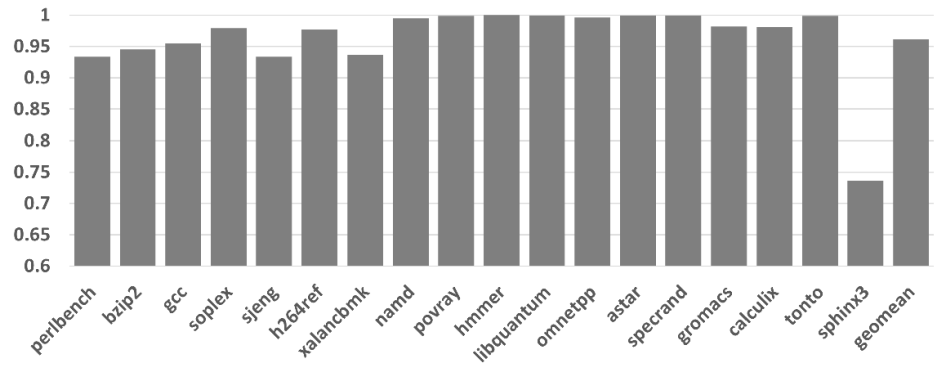


Figure 5.5: Slowdown for the CMC-enabled system for vc_per_vnet of 4 compared to the non-secure baseline.

Complex Attacks: This threat is countered by disrupting the fundamental attack stages, preventing complex attacks from triggering or effecting the system.

Our design prevents unauthorized accesses to memory regions due to privilege escalation or exploitation using mechanics described above for hardware threats, including when orchestrated as a complex attack. Importantly, since coherence messages are generated by hardware, a solely software-driven attack cannot engage in masquerading, modification, or diversion threats through packet manipulation without malicious hardware intervention.

5.6.2.2 Security Testing

To test the system's ability to counter the discussed threats, we inject tailored, malicious coherence messages at the network interface of cores. We verify that, for masquerading, modification, and diversion, a respective check exception is thrown and no malicious packets enter the interposer NoC before the system halts. For passive reading, we are successfully able to prevent GETX*spy* from viewing requests from other cores. Similarly, the *Forging Attack* does not trigger as it does not have the ability to passively read other cores requests and is unable to send requests to memory regions that its chiplet does not have permission to access (Sec. 5.5.2.2).

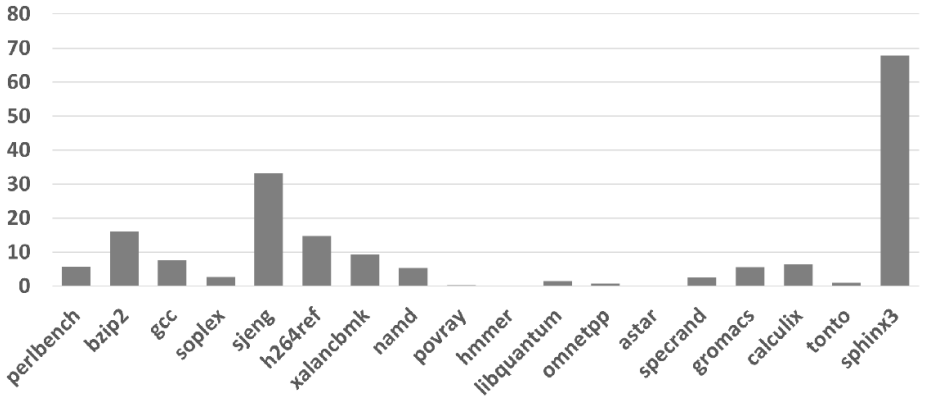


Figure 5.6: L2 cache miss rate [%].

5.6.3 Single-Threaded Performance Impact

Figure 5.5 shows the speedup/slowdown of the system with CMCs enabled compared to the baseline configuration. As expected, all workloads experience a speedup of less than 1 as the CMCs introduce higher latencies to the network. As the figure shows, the CMCs impose an average performance loss of ∼4%, with several benchmarks (*povray*, *hmmer*, *libquantum*) showing little to no impact. *sphinx3*, however, is an outlier, showing a significant ∼27% performance loss.

To analyze further, we examine the L2 miss rates of each benchmark in Fig. 5.6. The figure demonstrates that the variation between each benchmark's result in Fig. 5.5 is highly correlated to a benchmark's cache hit rate. For instance, *sphinx3* shows a much higher L2 cache miss rate than other benchmarks at ∼68%. The CMCs must process each packet resulting in increased memory access latencies. Thus, the CMC-enabled system's performance depends on the number of coherence messages that L2 cache misses inject into the NoC.

The performance degradation in some benchmarks is analyzed in Fig. 5.7, showing the percentage change for pre-injection queuing latency versus in-network latency and the total latency experienced by packets in the network. Interestingly, while the queuing latency increases by ∼80%, the in-network latencies drop by 5–10%. The increase in queuing latency is expected, due to the extra pipeline delays on network insertion that the CMCs cause. The decrease in in-network latency is

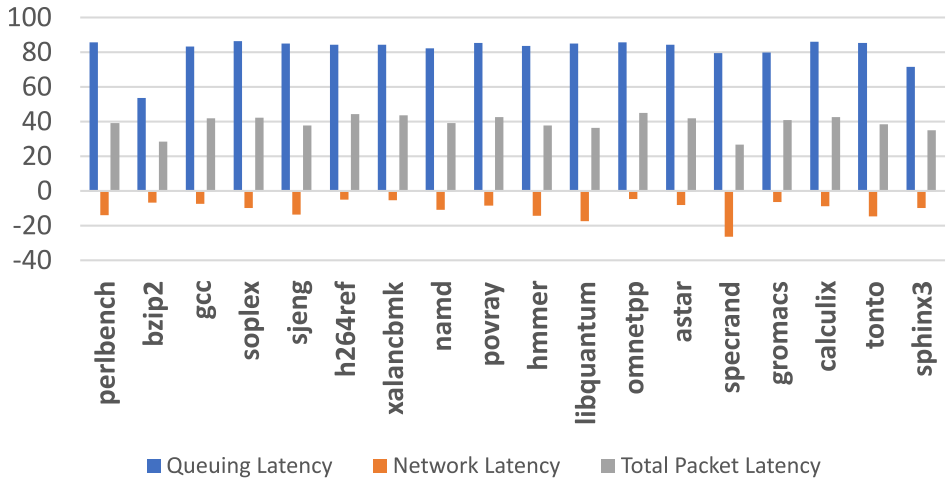


Figure 5.7: Change in packet latency induced by CMCs [%].

due to CMC-2 instances rerouting acknowledgment messages back to only the original requester (as a negative acknowledgment). Thus, the CMC-2 reduces total network load by removing one packet in the transaction.

The total packet latency increases by 39% on average. Interestingly, although *sphinx3* incurs a higher performance impact than the other benchmarks, it does not see a significantly different packet latency. That is, *sphinx3*'s performance loss is due to a higher L2 miss rate and hence higher packet injection, as discussed above, not a higher per-packet latency. Its higher miss rate exposes *sphinx3* more to the increase of network latency than other applications, which have lower miss rates.

Figure 5.8 depicts the speedup of the benchmarks with three different virtual channel configurations (vc_per_vnet). We observe that the geometric mean speedup approaches 0.98 with more virtual channels. We see a significant improvement in speedup for *sphinx3* due to the improvement in queuing latencies at the network interfaces. These significant gains imply that increasing VC count is a good way to improve performance if the application has a high cache miss rate.

In Fig. 5.9, we analyze the impact of increasing the interposer link widths to 128 bits versus the baseline of 64 bits.[6] This larger bandwidth provides slightly better speedup compared to the base-

[6]Due to runtime constraints for such large-scale simulations running on our shared high-performance computing

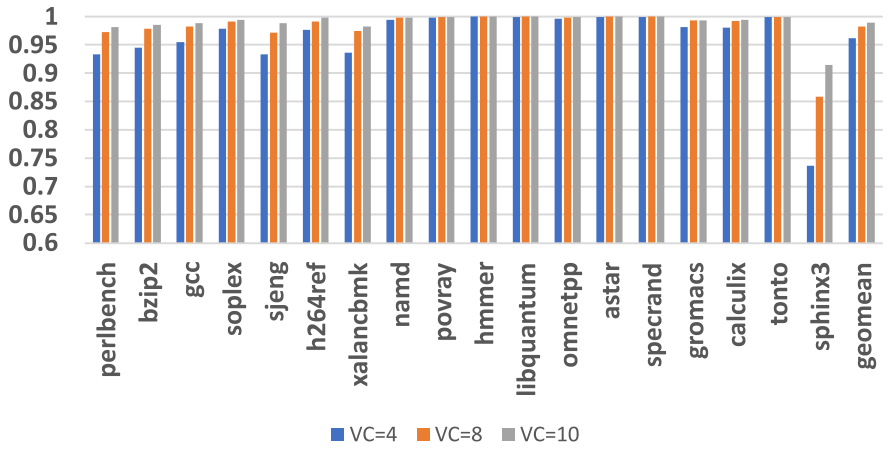


Figure 5.8: Slowdown for different VC configurations.

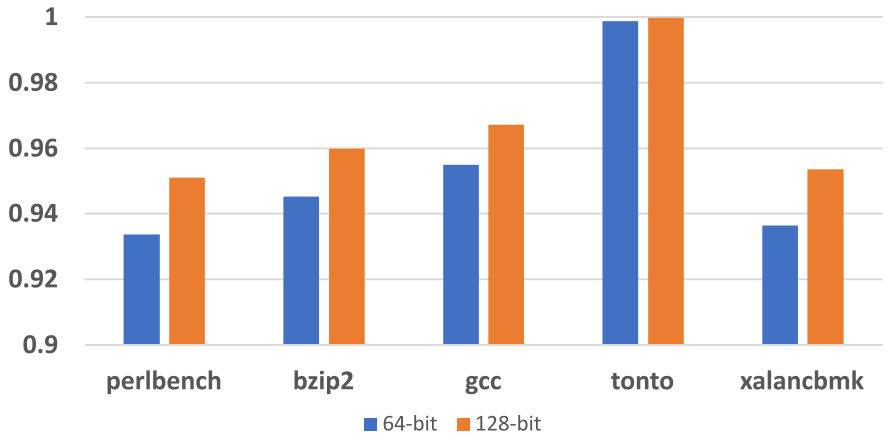


Figure 5.9: Slowdown for 64- versus 128-bit interposer links.

line. These modest gains imply that increasing the bit-width for the physical links in the interposer is likely not worthwhile, although this depends on the designer's trade-off for costs/overheads and scalability of the system.

5.6.4 Multi-Programmed Performance Impact

We evaluate the impact of the CMCs for multi-programmed workloads using random mixes of two benchmarks each, executed in two cores in separate chiplets. Here we simulate until all

cluster, we focused on a representative subset of benchmark runs for that particular experimentation.

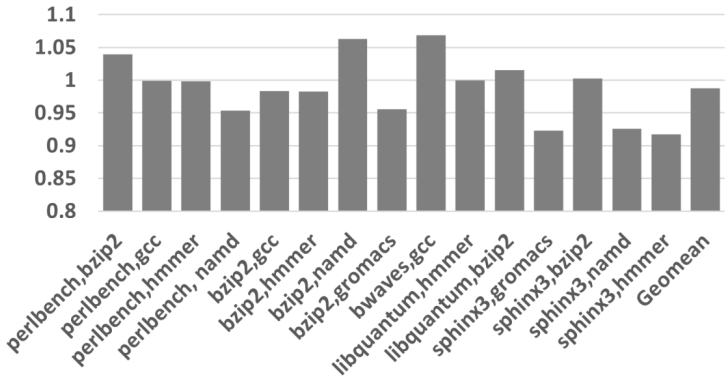


Figure 5.10: Speedup/slowdown for multi-program workloads.

applications complete at least five billion instructions and we report the weighted speedup of the combination using a methodology from Kadjo *et al.* [163].

Figure 5.10 shows the speedup for these multi-programmed workloads. In general, speedups range between 0.95 and 1.06. In some cases, namely *bzip2-namd* and *bwaves-gcc*, the speedup with the CMCs enabled was better than the baseline. Further, the mixes which included *sphinx3* showed reduced performance loss versus the stand-alone *sphinx3*. As before, the improvement is a result of CMC-2 filtering out packets otherwise sent to unauthorized chiplets. This reduces the bandwidth pressure that multiple applications induce on the NoC and appears to reduce the performance overhead as the number of workloads increase.

5.7 Summary

In this chapter, we propose the use of an active interposer as root of trust for modern chiplets-based systems, by implementing hardware security features directly within the interposer. More specifically, we devise a coherence message checker (CMC), which we propose to include at the boundary between the interposer and chiplets, memory controllers. We show how such a scheme addresses various attacks arising from malicious chiplets, with relatively low performance impact, $\sim$4% on average, compared to a non-secure baseline system.

6. CONCLUSION

While the direct benefits of Moore's Law are waning, modern architectures are moving towards more creative and innovative solutions to improve machine performance to meet consumer demand. These new designs present new challenges for both memory performance and security.

In the realm of performance, the Memory Wall [3] has been a challenge due to the disparity between processor and memory speeds. This dissertation explores the Memory Wall effect on the L1-I, presented as the front-end bottleneck. The large instruction footprints of modern workloads apply higher pressure on the L1-I. It is difficult for instruction prefetchers to train and predict due to their long-term temporal locality and lower spatial locality. We propose a new methodology for creating coordinated prefetchers to address this challenge. This methodology performs a directed design space search to combine multiple instruction prefetchers with complementary behaviors. We then explore recently proposed software instruction prefetchers in the context of modern front-end designs. From our evaluation, we characterize the three scenarios representing the front-end's behavior and state and how software instruction interacts with it to cause performance degradation.

Industry is moving towards chiplet-based designs to improve manufacturing efficiency and yield. However, these systems rely on a complex supply chain of various vendors and manufacturing facilities of varying levels of trustworthiness. This poses a security challenge for memory systems. A bad actor could insert a hardware Trojan capable of exploiting the coherence system to gain control of a system's entire memory space. In this dissertation, we highlight the vulnerability of coherence systems, which act as the communication protocol for transferring and updating memory between devices in a system. We demonstrate fundamental attack vectors a hardware Trojan may target and show how these fundamental stages can be orchestrated to enable complex attacks fully deceive the coherence subsystem into giving the Trojan access to any memory address. Following this, we propose using the active interposer in chiplet-based systems as the root of trust to embed security features on. Due to its low-performance demands, the active interposer can be manufactured in a controlled and trusted facility, allowing us to trust our proposed security

features. These security features act as Coherence Message Checkers (CMC), which use a packet checker/modifier (PCM) to verify the origin of messages and ensure they meet the requirements of a standard, non-malicious message. Further, the address protection unit (APU) in the CMC allows the CMCs to verify that a message is to a region allowed by the originating chiplet and filters messages to chiplets without accessing the target region.

Performance and security are ongoing areas of study, and the designs presented in this dissertation address challenges that must be addressed. As server and database computing needs increase and industry moves to more creative designs, the Memory Wall and the defense of the memory system will be further explored to provide realistic solutions. This dissertation seeks to enable these future solutions through the quick design and implementation of instruction prefetchers, provide a basis for the design of software instruction prefetchers, and make future designers aware of the vulnerability of coherence protocols and how to harden them in environments with connected components with varying trustworthiness.

REFERENCES

[1] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Rebasing instruction prefetching: An industry perspective," *IEEE Computer Architecture Letters*, vol. 19, pp. 147–150, July 2020.

[2] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Re-establishing fetch-directed instruction prefetching: An industry perspective," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 172–182, 2021.

[3] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, Mar. 1995.

[4] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker, "Performance characterization of a quad pentium pro smp using oltp workloads," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, (USA), p. 15âĂŞ26, IEEE Computer Society, 1998.

[5] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "Dbmss on a modern processor: Where does time go?," in *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, (San Francisco, CA, USA), p. 266âĂŞ277, Morgan Kaufmann Publishers Inc., 1999.

[6] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-out processors," *SIGARCH Comput. Archit. News*, vol. 40, p. 500âĂŞ511, June 2012.

[7] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," *SIGPLAN Not.*, vol. 47, p. 37âĂŞ48, Mar. 2012.

[8] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual*

International Symposium on Computer Architecture, ISCA '15, (New York, NY, USA), p. 158âĂŞ169, Association for Computing Machinery, 2015.

[9] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the front-end bottleneck with shotgun," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, (New York, NY, USA), pp. 30–42, ACM, 2018.

[10] S. Kanev, J. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a Warehouse-scale Computer," in *ISCA '15 Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 158–169, 2014.

[11] S. Mirbagher-Ajorpaz, E. Garza, S. Jindal, and D. A. Jiménez, "Exploring predictive replacement policies for instruction cache and branch target buffer," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, pp. 519–532, 2018.

[12] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, p. 7âĂŞ21, December 1978.

[13] B. Falsafi and T. F. Wenisch, *A Primer on Hardware Prefetching*. 2014.

[14] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 8–24, 1967.

[15] J. Pierce and T. Mudge, "Wrong-path instruction prefetching," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, (USA), p. 165âĂŞ175, IEEE Computer Society, 1996.

[16] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 63–74, 2007.

[17] S. Kondguli and M. Huang, "Division of labor: A more effective approach to prefetching," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 83–95, 2018.

[18] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, pp. 7–21, Dec. 1978.

[19] A. Ros and A. Jimborean, "The entangling instruction prefetcher." https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/eip_final.pdf.

[20] T. Nakamura, T. Koizumi, Y. Degawa, H. Irie, S. Sakai, and R. Shioya, "D-Jolt: Distant Jolt Prefetcher." https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/D-JOLT.pdf.

[21] D. A. JimÃl'nez, G. Chacon, N. Gober, and P. V. Gratz, "Branch agnostic region searching algorithm." https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/bar%C3%A7a.pdf.

[22] A. Seznec, "The fnl+mma instruction cache prefetcher." https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/FNLMMA-final.pdf.

[23] N. Gober, G. Chacon, D. A. JimÃl'nez, and P. Gratz, "The temporal ancestry prefetcher." https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/tap_final.pdf.

[24] A. Ansari, F. Golshan, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Mana: Microarchitecting an instruction prefetcher." https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/mana.pdf.

[25] V. Gupta, N. S. Kalani, and B. Panda, "Run-jump-run: Bouquet of instruction pointer jumpers for high performance instruction prefetching." https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/JIP.pdf.

[26] P. Michaud, "Pips: Prefetching instructions with probabilistic scouts." https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/pips_final.pdf.

[27] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 16–27, Nov 1999.

[28] G. Reinman, T. Austin, and B. Calder, "A scalable front-end architecture for fast instruction delivery," *SIGARCH Comput. Archit. News*, vol. 27, p. 234âĂŞ245, may 1999.

[29] C. Kaynak, B. Grot, and B. Falsafi, "Confluence: Unified instruction supply for scale-out servers," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 166–177, Dec 2015.

[30] R. Kumar, C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A metadata-free architecture for control flow delivery," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 493–504, 2017.

[31] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the front-end bottleneck with shotgun," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, (New York, NY, USA), p. 30âĂŞ42, Association for Computing Machinery, 2018.

[32] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), p. 462âĂŞ473, Association for Computing Machinery, 2019.

[33] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," *SIGPLAN Not.*, vol. 27, p. 62âĂŞ73, Sept. 1992.

[34] Chi-Keung Luk and T. C. Mowry, "Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors," in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 182–193, Dec 1998.

[35] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," *SIGOPS Oper. Syst. Rev.*, vol. 25, p. 40âĂŞ52, Apr. 1991.

[36] D. Chen, D. X. Li, and T. Moseley, "Autofdo: Automatic feedback-directed optimization for warehouse-scale applications," in *CGO 2016 Proceedings of the 2016 International Symposium on Code Generation and Optimization*, (New York, NY, USA), pp. 12–23, 2016.

[37] P. Vivet, E. Guthmuller, Y. Thonnart, G. Pillonnet, G. Moritz, I. Miro-PanadÃĺs, C. Fuguet, J. Durupt, C. Bernard, D. Varreau, J. Pontes, S. Thuries, D. Coriat, M. Harrand, D. Dutoit, D. Lattard, L. Arnaud, J. Charbonnier, P. Coudrain, A. Garnier, F. Berger, A. Gueugnot, A. Greiner, Q. Meunier, A. Farcy, A. Arriordaz, S. Cheramy, and F. Clermidy, "A 220GOPS 96-core processor with 6 chiplets 3D-stacked on an active interposer offering 0.6ns/mm latency, 3Tb/s/mm2 inter-chiplet interconnects and 156mW/mm2@ 82%-peak-efficiency DC-DC converters," in *Proc. Int. Sol.-St. Circ. Conf.*, pp. 46–48, 2020.

[38] J. Kim, G. Murali, H. Park, E. Qin, H. Kwon, V. Chaitanya, K. Chekuri, N. Dasari, A. Singh, M. Lee, H. M. Torun, K. Roy, 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 *Proc. Des. Autom. Conf.*, 2019.

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

[40] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. H. Loh, M. Subramony, and S. White, "Pioneering chiplet technology and design for the amd epyc and ryzen processor families : Industrial product," in *Proc. Int. Symp. Comp. Archit.*, pp. 57–70, 2021.

[41] 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.

[42] 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 *Proc. Int. Sol.-St. Circ. Conf.*, pp. 434–435, 2013.

[43] R. Mahajan, R. Sankman, N. Patel, D. Kim, K. Aygun, Z. Qian, Y. Mekonnen, I. Salama, S. Sharan, D. Iyengar, and D. Mallik, "Embedded multi-die interconnect bridge (emib) – a high density, high bandwidth packaging interconnect," in *2016 IEEE 66th Electronic Components and Technology Conference (ECTC)*, pp. 557–565, 2016.

[44] D. Cutress, "Bringing geek back: Q&a with intel ceo pat gelsinger," 2021. https://www.anandtech.com/show/17042/bringing-geek-back-qa-with-intel-ceo-pat-gelsinger.

[45] P. Coudrain, J. Charbonnier, A. Garnier, P. Vivet, R. VÃl'lard, 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. RibiÃĺre, 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. ChÃl'ramy, and G. Simon, "Active interposer technology for chiplet-based advanced 3D system architectures," in *Proc. Elec. Compon. Tech. Conf.*, pp. 569–578, 2019.

[46] E. E. Bilir, R. M. Dickson, Ying Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood, "Multicast snooping: a new coherence method using a multicast address network," in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, pp. 294–304, 1999.

[47] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner, "Power5 system microarchitecture," *IBM Journal of Research and Development*, vol. 49, no. 4.5, pp. 505–

521, 2005.

[48] N. Agarwal, L. Peh, and N. K. Jha, "In-network snoop ordering (inso): Snoopy coherence on unordered interconnects," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pp. 67–78, 2009.

[49] J. Archibald and J. L. Baer, "An economical solution to the cache coherence problem," in *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, (New York, NY, USA), p. 355âĂŞ362, Association for Computing Machinery, 1984.

[50] J. Zebchuk, M. K. Qureshi, V. Srinivasan, and A. Moshovos, "A tagless coherence directory," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 423–434, 2009.

[51] J. Laudon and D. Lenoski, "The sgi origin: A ccnuma highly scalable server," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, (New York, NY, USA), p. 241âĂŞ251, Association for Computing Machinery, 1997.

[52] M. N. I. Khan, A. De, and S. Ghosh, "Cache-out: Leaking cache memory using hardware trojan," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 6, pp. 1461–1470, 2020.

[53] M. Bidmeshki, G. R. Reddy, L. Zhou, J. Rajendran, and Y. Makris, "Hardware-based attacks to compromise the cryptographic security of an election system," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pp. 153–156, 2016.

[54] M. Kim, S. Kong, B. Hong, L. Xu, W. Shi, and T. Suh, "Evaluating coherence-exploiting hardware trojan," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 157–162, 2017.

[55] A. Ansari, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Divide and conquer frontend bottleneck," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 65–78, 2020.

[56] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–131, 2020.

[57] M. Bakhshalipour, M. Shakerinava, F. Golshan, A. Ansari, P. Lotfi-Karman, and H. Sarbazi-Azad, "A survey on recent hardware data prefetching approaches with an emphasis on servers," 2020.

[58] A. Ros and A. Jimborean, "A cost-effective entangling prefetcher for instructions," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 99–111, 2021.

[59] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The championship simulator: Architectural simulation for education and competition," *arXiv preprint arXiv:2210.14324*, 2022.

[60] G. Reinman, B. Calder, and T. Austin, "Fetch Directed Instruction Prefetching," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 16–27, 1999.

[61] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Re-establishing fetch-directed instruction prefetching: An industry perspective," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 172–182, 2021.

[62] International Symposium on Computer Architecture, *The 1st Championship Value Prediction Competition (CVP-1), http://www.microarch.org/cvp1*, June 2018.

[63] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path Confidence-based Lookahead Prefetching," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[64] C. Kaynak, B. Grot, and B. Falsafi, "Confluence: unified instruction supply for scale-out servers," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 166–177, ACM, 2015.

[65] C. Kaynak, B. Grot, and B. Falsafi, "Shift: Shared history instruction fetch for lean-core server processors," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 272–283, ACM, 2013.

[66] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive Instruction Fetch," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 152–162, 2011.

[67] R. Kumar, C.-C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A metadata-free architecture for control flow delivery," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pp. 493–504, IEEE, 2017.

[68] C. Zhang, Y. Zeng, J. Shalf, and X. Guo, "Rnr: A software-assisted record-and-replay hardware prefetcher," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 609–621, 2020.

[69] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "AsmDB: Understanding and Mitigating Front-end Stalls in Warehouse-scale Computers," in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), pp. 462–473, ACM, 2019.

[70] T. A. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "I-spy: Context-driven conditional instruction prefetching with coalescing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 146–159, 2020.

[71] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, p. 20âĂŞ24, March 1995.

[72] N. Adiga, J. Bonanno, A. Collura, M. Heizmann, B. R. Prasky, and A. Saporito, "The ibm z15 high frequency mainframe branch predictor industrial product," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 27–39, 2020.

[73] T. A. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "I-spy: Context-driven conditional instruction prefetching with coalescing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 146–159, 2020.

[74] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *SIGARCH Comput. Archit. News*, vol. 41, p. 475âĂŞ486, jun 2013.

[75] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[76] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, (New York, NY, USA), pp. 176–186, ACM, 1991.

[77] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak, "Branch history guided instruction prefetching," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pp. 291–300, Jan 2001.

[78] A. Kolli, A. Saidi, and T. F. Wenisch, "Rdip: Return-address-stack directed instruction prefetching," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 260–271, Dec 2013.

[79] A. Perais, R. Sheikh, L. Yen, M. McIlvaine, and R. D. Clancy, "Elastic instruction fetching," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 478–490, Feb 2019.

[80] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 152–162, Dec 2011.

[81] L. Spracklen, Yuan Chou, and S. G. Abraham, "Effective instruction prefetching in chip multiprocessors for modern commercial applications," in *11th International Symposium on High-Performance Computer Architecture*, pp. 225–236, 2005.

[82] C. Kaynak, B. Grot, and B. Falsafi, "Shift: Shared history instruction fetch for lean-core server processors," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 272–283, Dec 2013.

[83] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal streaming of shared memory," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, (Washington, DC, USA), pp. 222–233, IEEE Computer Society, 2005.

[84] A. Ansari, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Divide and conquer frontend bottleneck," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 65–78, May 2020.

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

[86] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?," in *IEEE HPCA*, pp. 168–179, 2018.

[87] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *IEEE S&P*, pp. 888–904, 2019.

[88] C. Trippel, D. Lustig, and M. Martonosi, "MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols," *CoRR*, vol. abs/1802.03802, 2018.

[89] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache hierarchy and memory subsystem of the amd opteron processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, 2010.

[90] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 168–179, 2018.

[91] J. Kim *et al.*, "Architecture, chip, and package co-design flow for 2.5D IC design enabling heterogeneous IP reuse," in *ACM/IEEE DAC*, pp. 1–6, 2019.

[92] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring(s): Side channel attacks on the CPU On-Chip ring interconnect are practical," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 645–662, USENIX Association, Aug. 2021.

[93] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, pp. 605–622, 2015.

[94] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, (San Diego, CA), pp. 719–732, USENIX Association, Aug. 2014.

[95] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *21st USENIX Security Symposium (USENIX Security 12)*, (Bellevue, WA), pp. 159–173, USENIX Association, Aug. 2012.

[96] C. Percival, "Cache missing for fun and profit," in *In Proc. of BSDCan 2005*, 2005.

[97] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 888–904, 2019.

[98] D. M. Ancajas, K. Chakraborty, and S. Roy, "Fort-NoCs: Mitigating the threat of a compromised NoC," in *ACM/EDAC/IEEE DAC*, pp. 1–6, 2014.

[99] M. H. Khan, R. Gupta, J. Jose, and S. Nandi, "Dead flit attack on NoC by hardware Trojan and its impact analysis," in *ACM NoCArc*, pp. 10–15, 2021.

[100] N. Prasad, R. Karmakar, S. Chattopadhyay, and I. Chakrabarti, "Runtime mitigation of illegal packet request attacks in networks-on-chip," in *IEEE ISCAS*, pp. 1–4, 2017.

[101] P. Garrou, "DARPA envisions CHIPS as new approach to chip design and manufacturing," 2018. https://www.3dincites.com/2018/10/iftle-396-darpa-envisions-chips-as-new-approach-to-chip-design-and-manufacturing/.

[102] 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.

[103] J. J. Rajendran, O. Sinanoglu, and R. Karri, "Building trustworthy systems using untrusted components: A high-level synthesis approach," *Trans. VLSI Syst.*, vol. 24, no. 9, pp. 2946–2959, 2016.

[104] C. Trippel, D. Lustig, and M. Martonosi, "Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols," 2018. arXiv.

[105] T. Boraten, D. DiTomaso, and A. K. Kodi, "Secure model checkers for network-on-chip (noc) architectures," in *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, pp. 45–50, 2016.

[106] V. Y. Raparti and S. Pasricha, "Lightweight mitigation of hardware trojan attacks in noc-based manycore computing," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2019.

[107] L. Fiorin, G. Palermo, S. Lukovic, V. Catalano, and C. Silvano, "Secure memory accesses on networks-on-chip," *IEEE Transactions on Computers*, vol. 57, no. 9, pp. 1216–1229, 2008.

[108] A. Prodromou, A. Panteli, C. Nicopoulos, and Y. Sazeides, "Nocalert: An on-line and real-time fault detection mechanism for network-on-chip architectures," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 60–71, 2012.

[109] A. Saeed, A. Ahmadinia, M. Just, and C. Bobda, "An id and address protection unit for noc based communication architectures," in *Proceedings of the 7th International Conference*

on Security of Information and Networks, SIN '14, (New York, NY, USA), p. 288âĂŞ294, Association for Computing Machinery, 2014.

[110] A. P. D. Nath, S. Boddupalli, S. Bhunia, and S. Ray, "Ark: Architecture for security resiliency in soc designs with network-on-chip (noc) fabrics," 2019.

[111] M. LeMay and C. A. Gunter, "Network-on-chip firewall: Countering defective and malicious system-on-chip hardware," *CoRR*, vol. abs/1404.3465, 2014.

[112] T. Boraten and A. K. Kodi, "Packet security with path sensitization for nocs," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1136–1139, 2016.

[113] S. Charles and P. Mishra, "Securing network-on-chip using incremental cryptography," in *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 168–175, 2020.

[114] M. A. Kinsy, S. Khadka, M. Isakov, and A. Farrukh, "Hermes: Secure heterogeneous multicore architecture design," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 14–20, 2017.

[115] C. H. Gebotys and R. J. Gebotys, "A framework for security on noc technologies," in *IEEE Computer Society Annual Symposium on VLSI, 2003. Proceedings.*, pp. 113–117, 2003.

[116] A. Limited, "âĂIJsecurity technology building a secure system using trustzone technology," tech. rep., 2009.

[117] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "IntelÂő software guard extensions (intelÂő sgx) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, (New York, NY, USA), Association for Computing Machinery, 2016.

[118] P. Maene, J. GÃűtzfried, R. de Clercq, T. MÃijller, F. Freiling, and I. Verbauwhede, "Hardware-based trusted computing architectures for isolation and attestation," *TC*, vol. 67, no. 3, pp. 361–374, 2018.

[119] H. Zhang, S. Ghosh, J. Fix, S. Apostolakis, S. R. Beard, N. P. Nagendra, T. Oh, and D. I. August, "Architectural support for containment-based security," in *Proc. Arch. Supp. Programm. Lang. Op. Sys.*, pp. 361–377, 2019.

[120] I. Lebedev, K. Hogan, J. Drean, D. Kohlbrenner, D. Lee, K. AsanoviÄĞ, D. Song, and S. Devadas, "Sanctorum: A lightweight security monitor for secure enclaves," in *Proc. Des. Autom. Test Europe*, 2019.

[121] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *Proc. USENIX Sec. Symp.*, pp. 523–539, 2017.

[122] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "VoltJockey: Breaching TrustZone by software-controlled voltage manipulation over multi-core frequencies," in *Proc. Comp. Comm. Sec.*, pp. 195–209, 2019.

[123] S. Bhunia and M. M. Tehranipoor, eds., *The Hardware Trojan War: Attacks, Myths, and Defenses*. Springer, 2018.

[124] D. Mehta, H. Lu, O. P. Paradis, M. A. M. S., M. T. Rahman, Y. Iskander, P. Chawla, D. L. Woodard, M. Tehranipoor, and N. Asadizanjani, "The big hack explained: Detection and prevention of pcb supply chain implants," *J. Emerg. Tech. Comp. Sys.*, vol. 16, no. 4, 2020.

[125] N. E. Jerger, A. Kannan, Z. Li, and G. H. Loh, "NoC architectures for silicon interposer systems: Why pay for more wires when you can get them (from your interposer) for free?," in *Proc. Int. Symp. Microarch.*, pp. 458–470, 2014.

[126] J. Yin, Z. Lin, O. Kayiran, M. Poremba, M. Shoaib Bin Altaf, N. Enright Jerger, and G. H. Loh, "Modular Routing Design for Chiplet-Based Systems," in *ACM/IEEE ISCA*, pp. 726–738, 2018.

[127] A. Coskun, F. Eris, A. Joshi, A. B. Kahng, Y. Ma, A. Narayan, and V. Srinivas, "Cross-layer co-optimization of network design and chiplet placement in 2.5D systems," *Trans. Comp.-Aided Des. Integ. Circ. Sys.*, 2020.

[128] S. Bharadwaj, J. Yin, B. Beckmann, and T. Krishna, "Kite: A family of heterogeneous interposer topologies enabled via accurate interconnect modeling," in *Proc. Des. Autom. Conf.*, 2020.

[129] G. Hellings, M. Scholz, M. Detalle, D. Velenis, M. de Potter de ten Broeck, C. Roda Neve, Y. Li, S. Van Huylenbroek, S.-H. Chen, E.-J. Marinissen, A. La Manna, G. Van der Plas, D. Linten, E. Beyne, and A. Thean, "Active-lite interposer for 2.5 & 3d integration," in *Symposium on VLSI Technology (VLSI Technology)*, pp. T222–T223, 2015.

[130] 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. 1611âĂŞ1625, Nov 2020.

[131] 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.

[132] W. Hu, C. H. Chang, A. Sengupta, S. Bhunia, R. Kastner, and H. Li, "An overview of hardware security and trust: Threats, countermeasures and design tools," *Trans. Comp.-Aided Des. Integ. Circ. Sys.*, 2020.

[133] T. Trippel, K. G. Shin, K. B. Bush, and M. Hicks, "ICAS: an extensible framework for estimating the susceptibility of IC layouts to additive trojans," in *Proc. Symp. Sec. Priv.*, pp. 1742–1759, 2020.

[134] X. Guo, R. G. Dutta, J. He, M. M. Tehranipoor, and Y. Jin, "QIF-Verilog: Quantitative information-flow based hardware description languages for pre-silicon security assessment," in *Proc. Int. Symp. Hardw.-Orient. Sec. Trust*, pp. 91–100, 2019.

[135] L. Fiorin, G. Palermo, S. Lukovic, V. Catalano, and C. Silvano, "Secure memory accesses on networks-on-chip," *IEEE Transactions on Computers*, vol. 57, no. 9, pp. 1216–1229, 2008.

[136] S. Evain and J. Diguet, "From noc security analysis to design solutions," in *IEEE Workshop on Signal Processing Systems Design and Implementation, 2005.*, pp. 166–171, 2005.

[137] T. Boraten and A. K. Kodi, "Mitigation of denial of service attack with hardware Trojans in NoC architectures," in *IEEE IPDPS*, pp. 1091–1100, 2016.

[138] D. Costan, V., "S. intel sgx explained.," tech. rep., 2016.

[139] M. R. Khandaker, Y. Cheng, Z. Wang, and T. Wei, "Coin attacks: On insecurity of enclave untrusted interfaces in sgx," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), p. 971âĂŞ985, Association for Computing Machinery, 2020.

[140] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh, "Nested enclave: Supporting fine-grained hierarchical isolation with sgx," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 776–789, 2020.

[141] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), p. 991–1008, USENIX Association, Aug. 2018.

[142] M. Schwarz, S. Weiser, and D. Gruss, "Practical enclave malware with intel SGX," *CoRR*, vol. abs/1902.03256, 2019.

[143] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pp. 142–157, 2019.

[144] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, (Baltimore, MD), USENIX Association, Aug. 2018.

[145] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1416–1432, 2020.

[146] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 347–360, 2017.

[147] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 974–987, 2018.

[148] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," MICRO-51, p. 428âĂŞ441, IEEE Press, 2018.

[149] M. Yan, J. Wen, C. W. Fletcher, and J. Torrellas, "Secdir: A secure directory to defeat directory side-channel attacks," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 332–345, 2019.

[150] 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.

[151] 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.

[152] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[153] 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.

[154] E. Witchel, J. Rhee, and K. Asanović, "Mondrix: Memory isolation for linux using mondriaan memory protection," *SIGOPS Oper. Syst. Rev.*, vol. 39, p. 31âĂŞ44, Oct. 2005.

[155] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, (New York, NY, USA), p. 2âĂŞ13, Association for Computing Machinery, 2008.

[156] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security 16)*, (Austin, TX), pp. 857–874, USENIX Association, Aug. 2016.

[157] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, (New York, NY, USA), p. 437âĂŞ452, Association for Computing Machinery, 2017.

[158] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," *SIGARCH Comput. Archit. News*, vol. 42, p. 457âĂŞ468, June 2014.

[159] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing ARM trustzone," in *26th USENIX Security Symposium (USENIX Security 17)*, (Vancouver, BC), pp. 541–556, USENIX Association, Aug. 2017.

[160] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, (New York, NY, USA), p. 304âĂŞ316, Association for Computing Machinery, 2002.

[161] B. W. Lampson, "Protection," *SIGOPS Oper. Syst. Rev.*, vol. 8, p. 18âĂŞ24, Jan. 1974.

[162] 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. MÃijck, 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 ÃL'der F. Zulian, "The gem5 simulator: Version 20.0+," 2020.

[163] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez, "B-fetch: Branch prediction directed prefetching for chip-multiprocessors," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 623–634, 2014.