BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Thesis

AXI OVER ETHERNET: REMOTE MEMORY TRANSACTION ANALYSIS VIA NETWORKED BUS TRAFFIC FORWARDING

by

PATRICK CARPANEDO

B.A., College of the Holy Cross, 2020

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2025

Approved by

First Reader

Renato Mancuso, PhD
Associate Professor of Computer Science

Second Reader

John Liagouris
Assistant Professor of Computer Science

Third Reader

Sabrina Neuman
Assistant Professor of Computer Science

We absolutely must leave room for doubt or there is no progress and there is no learning. There is no learning without having to pose a question. And a question requires doubt. People search for certainty. But there is no certainty. People are terrified — how can you live and not know? It is not odd at all. You only think you know, as a matter of fact. And most of your actions are based on incomplete knowledge and you really don't know what it is all about, or what the purpose of the world is, or know a great deal of other things. It is possible to live and not know.
-Richard P. Feynman

Acknowledgments

I want to thank my advisor, Prof. Renato Mancuso, for his patience and guidance. I want to thank the people in the systems group for the helpful discussions and feedback. Special thanks to my labmates in the Cyber-Physical Systems lab for the great conversations and for putting up with my constant questions. Especially to Mattia Nicolella, Bassel El Mabsout, Francesco Ciraolo, and Dennis Hoornaert for the great conversations, help, and support in making this thesis. Also, I want to extend a special thanks to my family and friends for their support and love. Of course, I want to thank my partner, Sisary, for her love and support in this journey.

AXI OVER ETHERNET: REMOTE MEMORY TRANSACTION ANALYSIS VIA NETWORKED BUS TRAFFIC FORWARDING

PATRICK CARPANEDO

ABSTRACT

Modern systems are approaching exceedingly complex designs as manufacturers are incorporating heterogeneous CPU architectures, hardware accelerators, and specialized components to address the growing amount of raw data input and the need for diverse computing resources. However, the cost of complexity has exacerbated issues with security, power efficiency, and temporal predictability. The common denominators are a lack of understanding and limited monitoring of the complex interplay between software and hardware components. Efforts have been made to address these pitfalls by introducing methods such as software/hardware containerization, optimized schedulers, and software standards/certifications. These methods have shown considerable improvements in strengthening the security properties of complex systems, achieving power trade-offs, and mitigating key sources of temporal non-determinism. Recent trends in computing models have given ground for new techniques that increase the amount of introspection in complex systems. In particular, given the rise in popularity of programmable logic, vendors are manufacturing tightly coupled FPGAs co-located with traditional compute clusters expanding the hardware/software co-design opportunities with incredible flexibility. We postulate that this model enables remote access to metadata and data traces that are historically confined on-chip.

The proposal of this thesis is to design, implement, and evaluate proof-of-concept

mechanisms with a twofold goal. First, we aim to devise a low-latency non-intrusive mechanism to monitor and/or manipulate information (e.g., about memory transactions, code execution, etc.) flowing through on-chip buses. Second, we tackle the challenge of forwarding data obtained through the first mechanism to a remote node via a dedicated high-bandwidth, low-latency interface. Appropriate packetization techniques are explored accordingly with compression given consideration. Said mechanisms enable novel paradigms for local and remote security threat identification and mitigation of edge systems. Furthermore, it could support complex remote live workload analysis with the objective of (1) achieving better power efficiency and (2) to drive local resource management and scheduling policies to achieve temporal determinism, to name a few.

Contents

List of Tables

List of Figures

List of Abbreviations

ACE	.....	AXI Coherency Extensions
AoE	.....	AXI over Ethernet
APU	.....	Application processor unit
AXI	.....	Advanced eXtensible Interface
BRAM	.....	Block Random Access Memory
CCI	.....	Cache Coherent Interconnect
CLB	.....	Configurable Logic Block
COTS	.....	Commercially-off-the-shelf
CPU	.....	Central Processing Unit
DRAM	.....	Dynamic Random Access Memory
DTB	.....	Device Tree Blob
FF	.....	FrameFormer
FFM	.....	FrameFormer Manager
FFS	.....	FrameFormer Subordinate
FPGA	.....	Field Programmable Gate Array
FPD	.....	Full Power Domain
GTH	.....	Gigabit Transceiver type H
ILA	.....	Integrated Logic Analyzer
LC	.....	Lucent Connector
LPD	.....	Low Power Domain
LUT	.....	Look-Up Table
MAIR	.....	Memory Attribute Indirection Register
NIC	.....	Network Interface Card
PE	.....	Processing Element
PL	.....	Programmable Logic
SFP+	.....	Small Form-factor Pluggable Plus
SoC	.....	System on Chip
VIP	.....	Verification Intellectual Property
XES	.....	Xilinx Ethernet Subsystem

Chapter 1

Introduction

The age of information has brought about a significant transformation in the way we interact with technology. Furthermore, the increase in computational needs has led to the development of more complex systems across sectors of the industry. Commonplace are interconnected heterogeneous systems that combine multiple processing elements, such as CPUs, GPUs, and FPGAs, to achieve high performance and efficiency. These systems are often used in applications ranging from data centers to edge computing, where the need for real-time processing and analysis is paramount. Despite the advancements in hardware and software, the challenge of profiling and monitoring these systems remains a critical issue. Traditional profiling techniques often fall short in providing the necessary insights into system performance, especially in heterogeneous embedded and realtime environments where multiple processing elements interact and communicate in complex ways. Consequently, the increase in computation and accelerators leads to an induced demand as developers and researchers seek to increase the workload of these systems, pushing them to their limits. Moreover the complexity of modern systems can exacerbate the challenges of profiling, as the interactions between different components can lead to unexpected performance bottlenecks and inefficiencies.

This emphasizes the need for more effective profiling techniques that can provide real-time insights into system performance, enabling developers to identify and address performance issues before they become critical. However, there remains a

limit on the amount of information that can be extracted from a single self contained systems. Therefore, it would be appropriate to design a system that can extend the capabilities of traditional profiling techniques by leveraging the interconnected nature of these embedded systems and the same efficient machanisms that enable multiple processing elements to work together

1.1 Motivation

Many hardware and software methods associated with hardware-level profiling involve a high cost of entry or significant overhead to operate. This high (either financial or compute) cost also applies to any method requiring higher information granularity. This work is geared toward addressing the gap between inflexible low-overhead hardware logic analyzers and malleable high-overhead software methods. The goal is to provide a means of analysis that can be handled remotely to alleviate the burdens on embedded systems in an effort to maintain real-time performance. This is achievable by utilizing a growing trend in COTS development boards that integrate a field-programmable gate array (FPGA) that can communicate with SoC components through high-speed buses. Furthermore, the exponential growth of networking speeds and latencies is minimizing the disparity between system buses and external communication. Coupling these two trends might allow us to expand the capabilities of a single system beyond the board by creating a link between system level mechanisms and mature networking infrastructure.

The goal is to provide an easily extendable, accessible, and low-cost hardware profiling solution, EthHelper, that can be used in embedded systems. This paper is structured to provide abstract design considerations, ZCU102 implementation specific details, and evaluation of the prototype. We provide realistic performance metrics using synthetic and pragmatic benchmarks for the system and demonstrate the practicality of our solution. Further, we want to motivate our application of EthHelper to works of control flow integrity (CFI) checks, security threat detection, and workload analysis.

1.2 Related Works

Methods of system observability for developers and researchers are constantly under development and refinement, and have given us a myriad of SW and HW implementations.

The software space has provided multiple frameworks and solutions to address the observability of applications at runtime over the decades. (Scales et al., 1996) is an example of the earliest attempts at providing fine-grain memory access with low overhead. (Ashraf et al., 2015) provides a general overview of common modern memory profiling toolkits such as (Luk et al., 2005) (Bruening et al., 2012) (Nethercote and Seward, 2007). These methods employ Dynamic Binary Instrumentation (DBI) to translate and instrument on the execution of a binary on the fly. This flexibility and low manual instrumentation are coupled with the immense effort of platform-specific porting and high runtime overheads resulting from context switches for each instrumented instruction. Furthermore, memory profiling with this class of profilers requires all memory space references to be instrumented.

Other efforts in the software space aim to leverage baked-in hardware debug infrastructures to offload statistic tracking or trace capturing. Works such as (Nicolella et al., 2022) (Bellec et al., 2020) rely on Performance Monitoring Unit (PMU) (Intel Corporation, 2022) (ARM, 2013), which keep track of hardware events such as retired instructions, cache and memory accesses, to profile an application at runtime with marginal overhead. (Chen et al., 2023) relies on a combination of PMU and trace data by utilizing the ARM Coresight debug infrastructure (ARM Ltd., 2004), which exposes components such as the Trace Memory Controller (TMC)(ARM Ltd., 2010) and Embedded Trace Macrocell (ETM)(ARM Ltd., 2012) for user configuration. This again allows one to achieve acceptable progress of an application despite contention through an added scope of observable events and statistics. The few limitations pre-

sented by this software and hardware combination are the predetermined events that can be monitored, the number of events that can be monitored concurrently, and the fetching blackout window needed to use the hardware.

This overall trend of delegating tasks to hardware for monitoring or accelerating tasks has seen efforts flourish in the programmable logic space. Similar to (Chen et al., 2023), (Hoppe et al., 2021) uses the Coresight infrastructure to monitor applications; however, it uses an FPGA to decrease the decoding time of the debug packets, since the dataflow can be understood and optimized within configurable hardware. In a similar effort to (Bellec et al., 2020), (Feng et al., 2021) attempts to lower the latencies of detecting attacks on control flow integrity by using the same Coresight+FPGA combination to monitor an application CFI exclusively through hardware.

The advancements in monitoring and profiling space have been improving the efficiency of singular embedded platforms by incorporating more hardware-level infrastructure to minimize latencies and overhead. Meanwhile, other spaces in the research community are seeing the applicability of remote resources (Aguilera et al., 2017), and there have been recent efforts to adapt programmable logic to lower the overhead of these understood dataflows. (Mizutani et al., 2021) proposed a fully connected network of tightly coupled FPGAs to provide significant cost reductions (e.g. packet processing time) for ¿100Gbps networks. (Calciu et al., 2019) and (Sidler et al., 2020) are works that try to increase the efficiency of remote memory access through programmable logic, but in two distinct manners. (Sidler et al., 2020) is intended to expand Remote over Converged Ethernet (RoCEv2) semantics and introduce data-shuffling at the Network Interface Card (NIC) level to provide consistent and performant remote data traversal and retrieval. (Calciu et al., 2019) minimizes dirty data amplification and page faulting associated with remote memory by allowing an FPGA to track and monitor cache-coherent traffic for statistics that the host

operating system can use.

The overlap of a permutation of these efforts have resulted in similar works proposes remote solutions to traditionally local operations. (Abera et al., 2016) and (Ammar et al., 2024) propose remote checking of Control Flow Attestations in order to provide a scalable and secure platform. (Basile et al., 2012) explores the space of remote-code integrity verification through the incorporation of an FPGA to generate CFA and harden distributed embedded systems. Further uses of FPGA in remote verification is explored in (Aysu et al., 2016), which proposes a remote integrity verification of the physcial system in which the FPGA is embedded.

Our work builds on the existing body of knowledge by proposing a novel approach to profiling and monitoring embedded systems through a combination of programmable logic and networking.

Chapter 2

Design

The following section provides an overview of the constraints and the requirements to achieve remote monitoring of hardware-level transactions with minimal impact. This section will provided the abstraction of 3 and provide the design choices and backhground that influence the implementation of the system.

2.1 Network Design Overview

In this initial work two systems are connected through a high-speed, low latency link in a producer/consumer architecture and communicate ontop a protocol that abstracts the physical layer. The producer has the capability of sending frames of bus-level data (or copies) without kernel involvement, while the consumer receives, unpacks, and parses the frames to visualize the producer's memory transaction history for observability purposes. This system is designed to work in a many-to-one or many-to-many model, as the consumers would be magnitudes more powerful than the producers to avoid possible bottlenecks and dropped packets.

2.1.1 Network Requirements

The network infrastructure must adhere to stringent requirements to meet the demanding needs of tapping into producer's on-chip high-speed system bus subsystem. A minimum bandwidth of 10Gbps is acceptable, for practical purposes 5·3, to sustain the rapid data exchange demanded by such subsystems. Additionally, latency

must be kept below 1 μs to ensure swift communication and minimize delays in data transmission. By meeting these criteria, the network can seamlessly accommodate the large data amount generated and maintain optimal producer performance.

2.1.2 Consumer Parsing

The consumer possesses the capability to parse incoming packets efficiently, enabling the extraction of transaction details. Within these packets, the consumer can discern the nature of transactions, identify the addresses impacted by each transaction, and determine the originating system component responsible for initiating the transaction. This parsing ability equips the consumer with comprehensive insights into data flow and transactional dynamics, facilitating precise analysis

2.2 Publisher Design Overview

The publisher system needs to provide a few basic components and properties. The system must have a processing element (PE), programmable logic (PL), external interface for network communication, main memory, an Observer within the PL, and a system bus subsystem to tie everything together. All components can communicate with any other components independently through the interconnect or other dedicated internal links. This communication must be standardized on all components and should use address space for directing access to the appropriate components, memory, or I/O devices. Communication with the bus and components is generally assumed to be parallel in nature (e.g. reads and writes can occur at the same time given no conflict). Considering these requirements, we will be focusing on specific applications of bus and external communication going forward. Therefore, we will implement AXI over Ethernet(AoE) in the following manners seen in 2·4.

First, transactions on the system bus are explicitly sent to the PL for duplication 2·1. This provides the shortest critical path for explicit routing given that

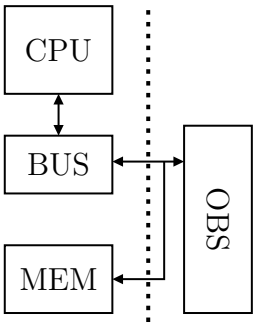


Figure 2·1: Transactions explicitly routed and duplicated within the PL

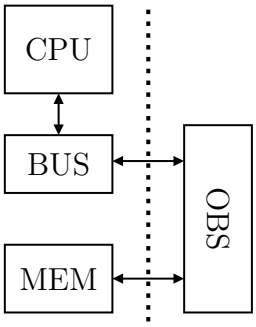


Figure 2·2: Transactions are explicitly routed through the Observer within the PL

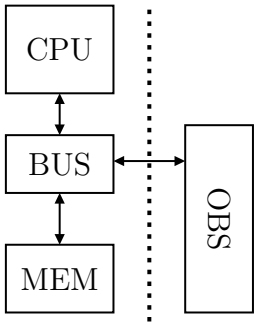


Figure 2·3: Transactions are broadcast on the System Bus with a passive Observer

Figure 2·4: Fundamental Routing Schemas

duplication does not imply blocking caused by the Observer's mechanisms. However, this a fundamentally lossy tracking architecture which may unpredictably drop data at the convenience of not slowing PE.

Second, transactions on the system bus are explicitly routed through the Observer in the PL 2·2. This allows for lossless tracking of transactions albeit at the cost of having all of the mechanisms involving the external interface on the critical path.

Lastly, cache-coherent system buses broadcast transactions which the Observer can passively tie into 2·3. A schema that would allow the Observer mechanisms to function without explicit routing to the PL address space. Furthermore, it comes at minimal cost since the transaction is not explicitly crossing the PE/PL boundary.

For ease of implementation, we will be implementing any modules to follow the 2·2 schema. This will provide a lossless tracking mechanism with minimal engineering overhead.

2.3 PL Design Overview

2.3.1 Memory-mapped Programmable Logic Block

We assume that the PL block has multiple address spaces that can be used to account for possible different power/time domains on a system. These physical address spaces can be allocated to the same or separate physical ports on the Bus. This allows for the PL to contain multiple mechanisms to generate traffic on separate ports to the Bus.

2.3.2 Dedicated hardware

The basic building blocks of FPGAs are Look Up Tables (LUTs) which can be composed into fundamental transistor logic (e.g. AND, OR, etc). Throughout the development of the FPGA dedicated hardware blocks have been embedded into the FPGA to accelerate computation or introduce new functionality. For this paper, we will focus on the use of three dedicated hardware blocks found in modern FPGAs. Digital Signal Processors (DSPs) are computational accelerators for multiplication and division. Block RAM was invented to bring memory inside the FPGAs. Finally, GTH/GTY transceivers were introduced to handle external communication.

2.3.3 Observer Mechanism

The *Observer* has to fulfill several tasks: (1) communicate with the system bus, (2) form packets of transaction data, (3) instantiate a controller to use the external interface, and (4) send the data through the physical port. We show a generic module and layout to achieve the route through schema 2·2 with 2·5.

Serializer/Communicator is the initial point of the *Observer* mechanisms which allows communication to and from the System Bus. Furthermore, This module is responsible for serializing data and metadata of R/W requests and distributing the

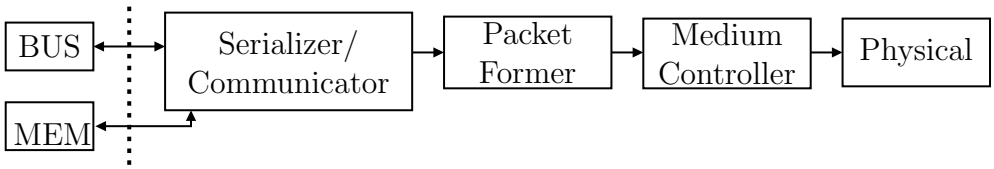


Figure 2·5: General module implementation in PL. For now, Communicator has Bidirectional communication only with non-PL components

data stream among available external interfaces.

The *Packet Former Module (PFM)* is responsible for assembling serialized data and interfacing with the external interface controller. The PFM fulfills its responsibility by accumulating and packaging bus data to comply with the external interface protocols. In addition, this module abstracts control signals between the Serializer/Communicator and the *Medium Controller* to allow for the accumulation of data without blocking the PE and correct communication.

A *Medium Controller* needs to be instantiated to ingest packets from the Packet Former and manage external interface transceivers to transmit data. Moreover, this module offers debug signals to verify protocol compliance and transceiver operation. The medium and protocol will dictate what component in the PL will contribute to the overall additional latency of using AoE. In respect to the minimum requirements, the latency and throughput of the system will be limited by the medium as DDR4 (the defacto memory in use) has a sustained raw throughput of 12.8 GB/s with 15ns CAS latency A·4.

Chapter 3

Implementation Overview

3.1 System Overview

Our test environment contains two systems that serve the purpose of publisher and subscriber. Our subscriber will be a Dell Precision 7950 equipped with Dual Intel Xeon Gold 6130 and a Mellanox ConnectX-2 Dual SFP+ NIC running Ubuntu 22.04. The publisher is a Xilinx Ultrascale+ ZCU102 with ZU9EG SOC running a generic Petalinux with SFP+ transceivers enabled. The FPGA bitstream was compiled with Vivado v2019.2. These two systems communicate through a single point-to-point connection via 10GBase-SR SFP+ LC Transceiver and OM3 fiber cable. We used tcpdump, a cli tool, to save the exported ethernet packets

3.2 ZCU102 Overview

The ZCU102 is an embedded development platform commercially-of-the-shelf (COTS) developed by AMD (previously Xilinx) equipped with ZU9EG SOC. This SOC is a heterogeneous architecture with an application processor unit (APU) composed of quad ARM Cortex-A53, dual Cortex-R5, and an UltraScale+ FPGA. These three main components interact through three communication modules: the Cache Coherent Interconnect (CCI), the Full Power Domain (FPD) main switch, and the Low Power Domain (LPD) main switch 3·1. The scope of this paper will encompass the APU, FPGA with dedicated SFP+ transceivers, and FPD main switch. All memory

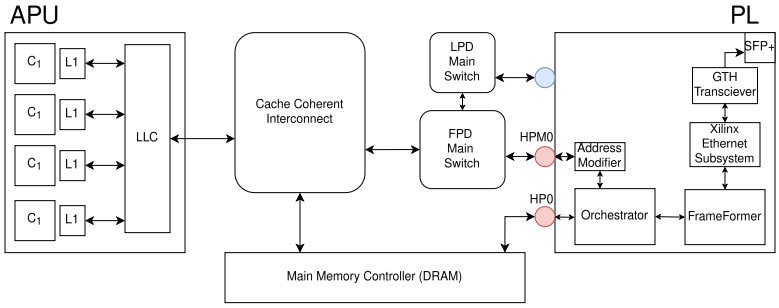


Figure 3·1: Internal Layout of the ZCU102 and EthHelper components

allocated for applications will be routed through the FPGA aperture and ultimately be in system memory. This initial concept is possible through reserving memory in the Device Tree Blob (DTB), RT-Bench, and the Advanced eXtensible Interface (AXI) protocol.

3.2.1 Advanced eXtensible Interface

Heterogeneous systems necessitate a common form of communication to facilitate computation and transactions between components of differing underlying mechanisms. Our choice of platform is constructed around the AXI protocol for most inter-chip communication. AXI Coherency Extensions (ACE) are used exclusively for high-speed components connected directly to the CCI.However, ACE is a bidirectional protocol for maintaing coherence with APU caches. ACE is also limited as only read type transactions can be seen excluding any write backs without explicit routing. This will be out of scope for the paper but is not a limitation of the current work or platform. AXI (and ACE by extension) is a memory-mapped communication protocol that relies on (1) manager and subordinate scheme and (2) a handshake mechanism to execute memory transactions. AXI comes in three variations FULL, LITE, and STREAM. This paper will touch directly on AXI4Full and AXI4Stream as these are the necessary components for Transparent Snooping. AXI4Full is a bidirectional

protocol that employs 5 channels (2 for reading memory and 3 for writing) that operate independently and in parallel. In contrast, AXI4Stream is a single-channel unidirectional protocol intended for data streams.

3.3 FPGA Overview

The FPGA has 4 major tasks: redirecting memory transactions from APU to DDR memory, serializing of AXI4Full to AXI4Stream , forming an ethernet frame, and correctly using the ethernet module to send packets. We created the Orchestrator module and the FrameFormer module to utilize the Xilinx-provided 10g/25g ethernet subsystem (XES) module. We shall refer to all modules collectively as the EthHelper.

The Orchestrator imposes a simple fair scheduler to schedule up to 5 submodules to serialize the AXI4Full channels (e.g. AR, AW, R, W, B) to a single AXI4Stream channel. Furthermore, the Orchestrator was architected to be modular, allowing the creation of submodules that address other protocols that implement handshaking. For a proof of concept, we constructed the AR and AW submodules to give us metadata consisting of transaction type, AXI ID, Burst Length, and a clock-based timestamp. The other channels are connected to passthrough dummy modules that allow uninterrupted execution, yet the scheduler is implemented for all 5 submodules to work. The current iteration of the Orchestrator requires a source and target that communicates through AXI4Full to generate understandable AXI4Stream outputs.

The FrameFormer (FF) is a custom FIFO module that allows configurable ethernet packet arguments (except for 802.1Q tag and CRC) to frame and send incoming AXI4Stream data. The FF is internally split into two parts: the FrameFormerSubordinate (FFS) and the FrameFormerManager (FFM). The FFS provides a shifting register to buffer and send all incoming data for the FFM. the FFM provides the control outputs for the XES and will create and size the ethernet frame with FFS

data and inputs given. The mechanism is as follows: FFS will wait for a single AXI4Stream burst to write into the shifting register, upon receiving the FFS will initiate the FFM to output all parts except payload to the XES, after the FFM will receive all data within the FFS shifting register until packet size is reached, once the packet size is reached it will either idle or redo the cycle depending if there is still more data.

Chapter 4

Implementation Details

4.1 Subscriber In-depth view

4.1.1 Memory and Buffers

Network data transmission relies heavily on buffering mechanisms to reconcile the asynchronous nature of network interfaces and host system processing capabilities. Two primary buffer types are employed: hardware-managed buffers within the Network Interface Card (NIC) and kernel-space buffers. These buffers serve a critical role in mitigating the performance disparity between network link speeds and the processing rate of the host system, preventing data loss and optimizing overall RX/TX efficiency. The hardware NIC buffer, typically one for transmit (TX) and one for receive (RX) operations, provides a temporary storage area for data awaiting processing. Increasing the size of the NIC RX buffer can significantly reduce the interrupt load on the host CPU, as fewer interrupts are required to signal per given packet amount. Furthermore, larger RX buffers enable the transmission and reception of jumbo frames (packets exceeding the standard IEEE 802.3 Ethernet frame size of 1500 bytes), which can reduce protocol overhead by consolidating multiple smaller packets into a single, larger frame. While increasing buffer sizes can improve throughput, it is important to acknowledge the potential impact on network latency, a factor not directly addressed within the scope of this research.

The kernel buffer acts as an intermediary, facilitating the transfer of data from the NIC hardware buffer to the user application memory. Insufficient kernel buffer

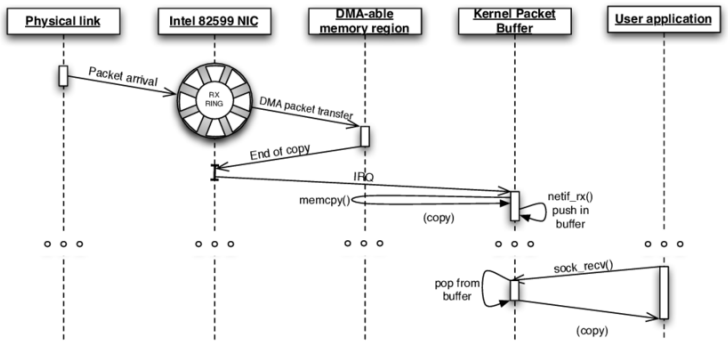


Figure 4·1: Buffer usage in linux pre 2.6.

capacity can lead to two primary issues: NIC buffer overflow (occurs in Linux 2.6+ with New API (NAPI) implementation), where the NIC hardware buffer overwrites incoming data, or excessive kernel buffer thrashing, manifested as frequent calls to netif_rx() for pre-NAPI implementations 4·1, which both are indicative of a bottleneck. The kernel buffer's size must be carefully considered in relation to the NIC's RX buffer size to ensure efficient data flow and prevent these detrimental effects.

A significant performance consideration is the often default configuration of NIC buffer sizes. Many network interfaces are configured with RX buffer sizes ranging from 256 to 1024 bytes, significantly smaller than the NIC's maximum buffer size of 4096-8192 bytes found on modern Ethernet NICs. Consequently, the kernel buffer must be proportionally sized to accommodate the increased data volume handled by the larger NIC RX buffer, ensuring a balanced and efficient data transfer pipeline.

4.1.2 Parsing and visualization

The data acquisition and analysis pipeline constructed for this research utilizes a combination of command-line tools and custom software programs. Packet capture is performed using tcpdump, a well-known command line packet analyzer, during the runtime of the publisher system application under observation. The subsequent processing and visualization of the data are executed offline. Furthermore, the process

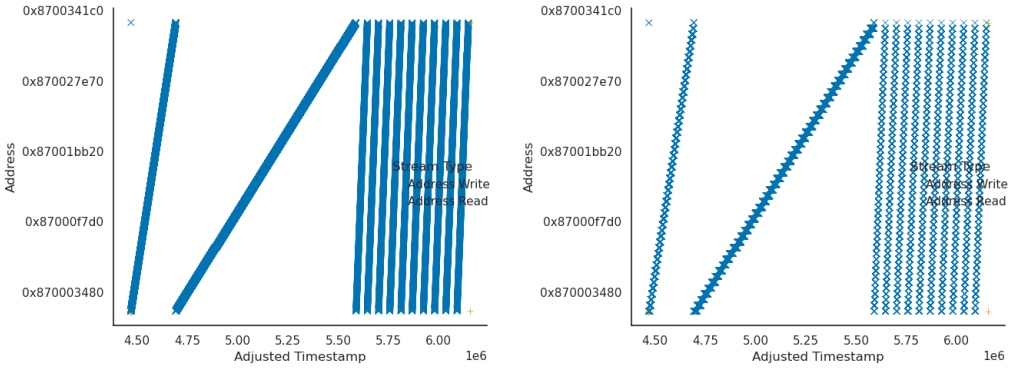


Figure 4·2: Example of a FPGA trace visualized with data obtained by the subscriber

is not limited to this operation scheme.

The proccessing of packets of the pipeline is handle by two C++ parsers: packetStripper.cpp and packet_processor.cpp.The process starts with the preproccessing of raw packets captured by tcpdump with packetStripper.cpp. This step extracts the raw hexadecimal data of the captured packet trace. Then packet_processor.cpp identifies user-defined start and end delimiters within the FPGA data stream. This design choice of start and end delimiters allow identifying a complete and fully formed packet. After, packet_proccessor.cpp performs several transformations: removal of inter-transaction padding, byte order inversion, and reformatting of the data into a human-readable structure. This structured data includes address information, metadata, and a decomposition of the metadata into fields such as AXI ID, AXI burst length, transaction type, and a clock cycle timestamp. To facilitate debugging and modularity, each processing stage is output to a separate file.

The final stage of the pipeline is implemented in Mapper.py, a Python script responsible for constructing the visualization of the processed data. This script uses the parsed and formatted data and generates two scatterplots. These scatterplots depict the distribution over time of transaction types across a defined address range, with the granularity of the address at byte-level and page-level.

4.2 ZCU Details

4.2.1 General Configuration

The ZCU102 has been configured to run Petalinux 2023.2 without modifications to the kernel or user applications. There is a modified device tree blob (DTB) that includes a modified Si570 frequency of 156.25MHz (per XES/IEEE specification) and multiple instantiations within the amba_pl block for configuring the necessary clocks of the EthHelper. Additionally, there is a reserved memory region of 256MB in DRAM for testing purposes to avoid data collisions.

4.2.2 Memory Alignment and Caching

The Orchestrator is capable of handling unaligned accesses without issue due to it being more of a passthrough/monitor module on the critical route. Simply, all transactions, given that components upstream and downstream of the Orchestrator work correctly together. However, the Cortex-A53, by default, is not configured to allow unaligned accesses in device memory or non-cacheable memory. Yet, this feature of unaligned access has been supported since ARMv6(ARM Ltd., 2018). This is corrected in two possible ways: modifications to the Memory Attribute Indirection Register (MAIR), or making the memory region cacheable. We chose for the latter as a colleague had a module from (Izhbirdeev et al., 2024) that could be modified with ease with memremap() to allocate a 3MB range at the specific memory apertures of the FPGA module.

4.3 FPGA In-depth view

4.3.1 AXI4Full to AXI4Stream translation

The Xilinx Ethernet Submodule (XES) used for this preliminary study has the datapath for RX/TX built as an AXI4Stream interface. Along with the limitation of

4 physical ports for the XES to utilize, there required a serializer to convert the 5 channels of AXI4Full into a single AXI4Stream channel. This responsibility formed into the Orchestrator module.

The module itself is quite simple and is composed of a simple round-robin scheduler and a finite state machine (make reference and image) internally with output signals to actuate sub-modules, which the end user could create, that handled the interface responsibilities. Each submodule is responsible for a single channel on an interface. The submodules send up information of valid and ready signals, in progress status, transaction length, along with metadata and data to the Orchrestrator. The Orchestrator's passes down resets, clock, and AXI4Stream ready signals to the submodules. The Orchestrator uses a set of encoding masks that allow the proper control for all modules at in a single clock cycle, so that the proper submodule can unblock a pending transaction. The encoding mask design choice was made to retain the lowest channel transaction latency possible. The Orchestrator was also made with the intent of an easily modifiable (offline or online) and understandable wrapper and framework for any serialization to AXI4Stream .

Given a simple assumption that the downstream AXI4Stream is always ready. We can formulate a simple equation for the longest wait time that one module needs to wait to unblock. Given N enabled submodules and a function $Burst\_Length()$ which gives the transaction length for the n_i we can make 4.1:

$$\sum_{n=1}^{N} 2 * Burst\_length(n_i) + 2 * N \tag{4.1}$$

Where the longest latency will be for the last submodule as it will have to wait the orchestrator to send all metadata and data from each burst of each submodule, with the added overhead of enabling and receiving data from each submodule.

4.3.2 Frame Former

This module allows the decoupling of signals and configuring data link layer parameters. The FrameFormer comprises a FrameFormerSubordinate (FFS) and the FrameFormerManager (FFM).

The FFS submodule provides a shifting register array (that is user-defined in length) to buffer incoming data from the Orchestrator in order to minimize the amount of blocking the downstream sending mechanisms present. The blocking may come from frame-forming actions (e.g., sending start of frame and sending the end of frame) to physical module or protocol actions (inter-frame gap wait, resets on error, etc). The initial motivation for using a shift register was to have the latest data on a wire and minimize the logic to refresh the data. In comparison, a traditional approach would have an additional register to store the output, whereas we intended the first register of the shifting register array to uphold this responsibility. This choice allowed all logic to revolve around the single object of the shifting register. It may have only had a slight clock difference.

FFM is a simpler module that executes the actual framing of a packet and sends the correct signal to the XES for proper transmission. Furthermore, this module allows the users (and in the future programs) to select the parameters of the ethernet frame (e.g. source/destination address, ethernet type, and frame length) to other parameters such as sync words (i.e. to show start and end of FPGA stream) via configuration ports.

4.3.3 10g/25g Ethernet Subsystem

To facilitate testing the Xilinx 10g/25g Ethernet module was employed for availability and documentation purposes. It does require an additional license for xxv-ethernet-3.1 which comes free of cost yet is only available for 180 days. The module has

an AXI4Full interface for configuration of the MANY options. However, the default module configuration is ready to transmit (given the correct signaling) data. Furthermore, there are two AXI4Stream interfaces for rx/tx.

Chapter 5

Evaluation

5.0.1 Evaluation Infrastructure

The behavioral analysis methodology incorporated simulations during preliminary stages, followed by implementation of a System Integrated Logic Analyzer (ILA) in conjunction with supplementary debug pins and counters. The software infrastructure provided by (Nicolella et al., 2022) enabled monitoring of benchmark metrics for (Venkata et al., 2009) and programs in (Valsan et al., 2016), while simultaneously providing the requisite mechanisms for heap reallocation to specified memory regions. Furthermore, as stated in Section 4.2, a kernel module was implemented to facilitate unaligned access and caching to the FPGA memory region. Three physical address apertures were established, directing to physical DRAM, DRAM loopback, and EthHelper, as illustrated in Figure 5·1 with a mechanism similar to (Roozkhosh and Mancuso, 2020) using a modified UARTDriver module from (Ciraolo et al., 2025)

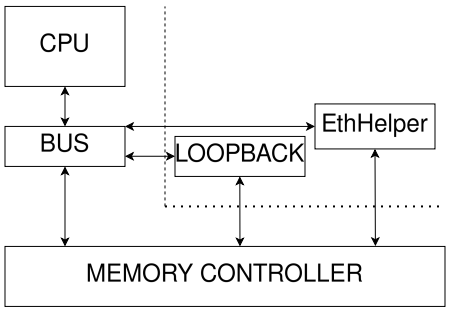


Figure 5·1: General evaluation infrastructure for evaluation experiments

Resource	Utilization	Available	Utilization %
LUT	43878	274080	16.01
LUTRAM	11477	144000	7.97
FF	67303	548160	12.28
BRAM	69	912	7.57
IO	1	328	0.30
GT	1	24	4.17
BUFG	9	404	2.23

Table 5.1: Overall FPGA resource usage in percentage of tested design

Components	LUT %	BRAM %	CLB %
Orchestrator	0.04	0.03	0.11
FrameFormer	1.62	0.78	2.63
AXI Width Converter	0.03	0.05	0.17
XES	3.11	3.88	10.30

Table 5.2: FPGA resource usage of EthHelper components in percentage of tested design

that we will refer to as the *Address Modifier*.

5.0.2 FPGA Utilization and Limits

The design was synthesized with Vivado 2019.2 with a clock frequency of 156.25MHz for the dataflow path and 75MHz for the control path. The overall utilization 5.1 for the tested design is quite light despite the additional debug infrastructure. Furthermore, the core components to enable AoE use significantly less resources 5.2.

5.0.3 Verification of Operation Characteristics

Testbenches were developed within the Vivado environment to evaluate custom modules. AXI Verification IP (VIP) modules were employed to generate and validate AXI4Full and AXI4Stream transactions. The primary objectives of simulation testing were to observe and optimize Orchestrator state transitions and transaction handling latency. Figure 5·2 demonstrates the correctness of the Orchestrator given single and multiple concurrent channel transactions. Additionally, the simple bounding equation proposed in Section 3 demonstrates consistency with System ILA capture

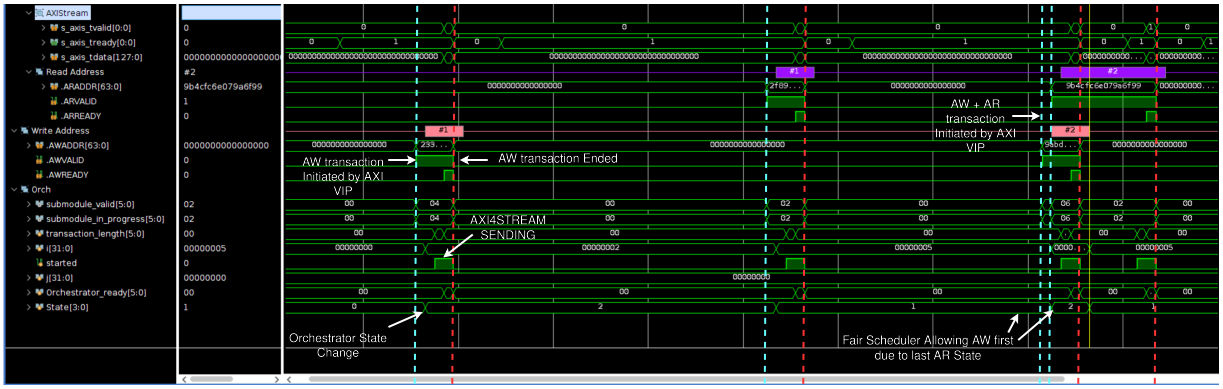


Figure 5·2: Analysis of the Orchestrator functionality

data and simulation.

5.0.4 Preliminary Evaluation

Baseline application and medium throughput metrics were established utilizing Isolbench's bandwidth program. The experimental protocol employed a one-shot configuration on a FIFO scheduler with priority 99 and memory allocation of 12MB, while constraining execution to CPU #1. The experiment was conducted across three non-cached memory regions: direct DRAM access, DRAM loopback in the FPGA, and the EthHelper. This methodological approach facilitated calculation of the sustained throughput of the XES through analysis of counter outputs and benchmark metrics. Empirical data revealed transmission of 290,063 packets, each comprising 1035 bytes, with program execution completing in 0.21111 seconds. Subsequent computational analysis yielded a medium throughput of 10.5953 Gb/s, with application throughput recorded at 277.5679 MB/s.

The results depicted in Figure 5·3 demonstrate that the predominant performance penalty is attributable to traversing the Processing Element/Programmable Logic (PE/PL) boundary, as evidenced by comparable throughput measurements between EthHelper and DRAM loopback configurations. The observed throughput differential can be attributed to disparities in clock frequencies, power domains, and extended

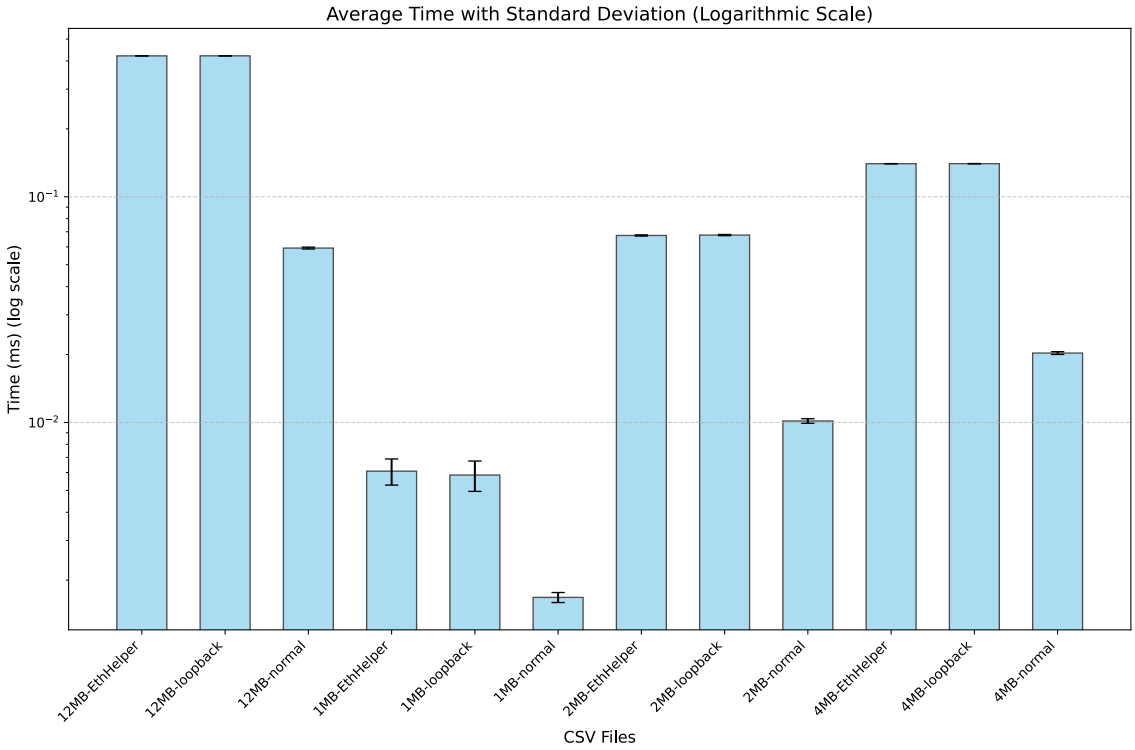


Figure 5·3: Bandwidth.c execution time comparison between different memory routes

critical path length. Moreover, runtime analysis revealed minimal utilization of the FrameFormer buffer, rarely exceeding three buffer allocations, indicating that the performance bottleneck resides primarily within the Orchestrator component.

5.0.5 Pragmatic Benchmarks

The study utilizes (Venkata et al., 2009) as the foundation for pragmatic benchmarks, selected for its provision of realistic, real-time workloads. A kernel module was implemented to perform memremap operations on FPGA memory regions using the CACHE_WB flag. It is imperative to acknowledge the constraints of the module and the associated limitations on memory allocation for pragmatic benchmarks; specifically, mapping capabilities were restricted to a maximum of 3 megabytes, despite a 256 megabyte range reserved and mapped in the Device Tree Blob DTB. Conse-

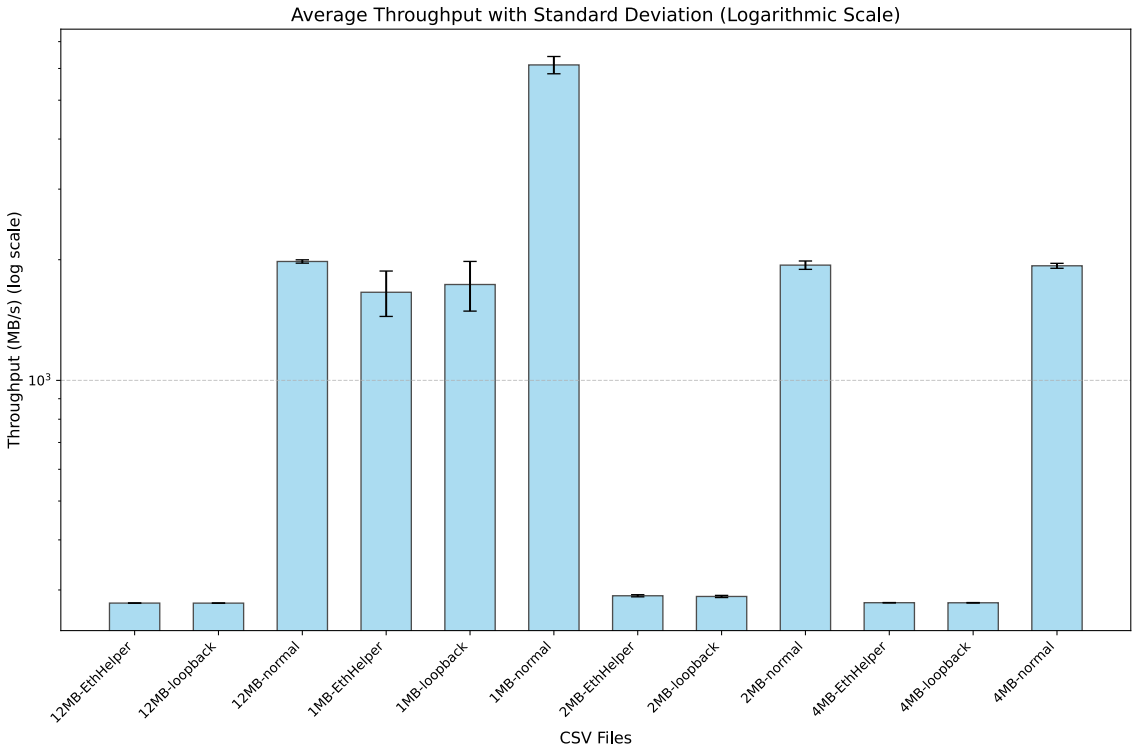


Figure 5·4: bandwidth throughput comparison between different memory routes.

quently, experimental evaluations were constrained to small image sizes categorized as "sim" and "sim_fast." This limitation particularly affects programs characterized by memory-bound operations due to intensive memory transactions, restricting observability of larger image dimensions. The benchmark selection incorporates a balanced distribution between memory-intensive and compute-intensive programs to comprehensively demonstrate the performance impact of the EthHelper during representative workloads.

The module's performance aligns with expectations 5·5 5·6. Memory-intensive applications such as mser, disparity, and tracking exhibit characteristic behavior of memory-bound computations, wherein memory transactions incur substantial penalties associated with power domain transitions. While statistical significance of standard deviation between EthHelper and loopback configurations is not explicitly demonstrated in this experiment, differences are attributable to the Orchestrator's operational requirement to block alternative channels during individual channel transactions.

5.0.6 Tracing and Visualiization

The tracing and visualization infrastructure leverages tcpdump, a conventional command-line interface program utilizing libpcap, in conjunction with custom Python scripts. Hardware and software limitations within the evaluation infrastructure result in occasional data discontinuities and interpretative inaccuracies in the Python scripts responsible for parsing and visualizing data captured from tcpdump packets. It should be emphasized, however, that these limitations do not compromise the core FPGA infrastructure, which maintains data integrity without information loss. Furthermore, the implemented kernel module constrains memory accesses to cache line dimensions (to which the orchestrator is capable of byte alignment and recording). To mitigate Network Interface Controller buffer overruns, memory bomds of extended execution

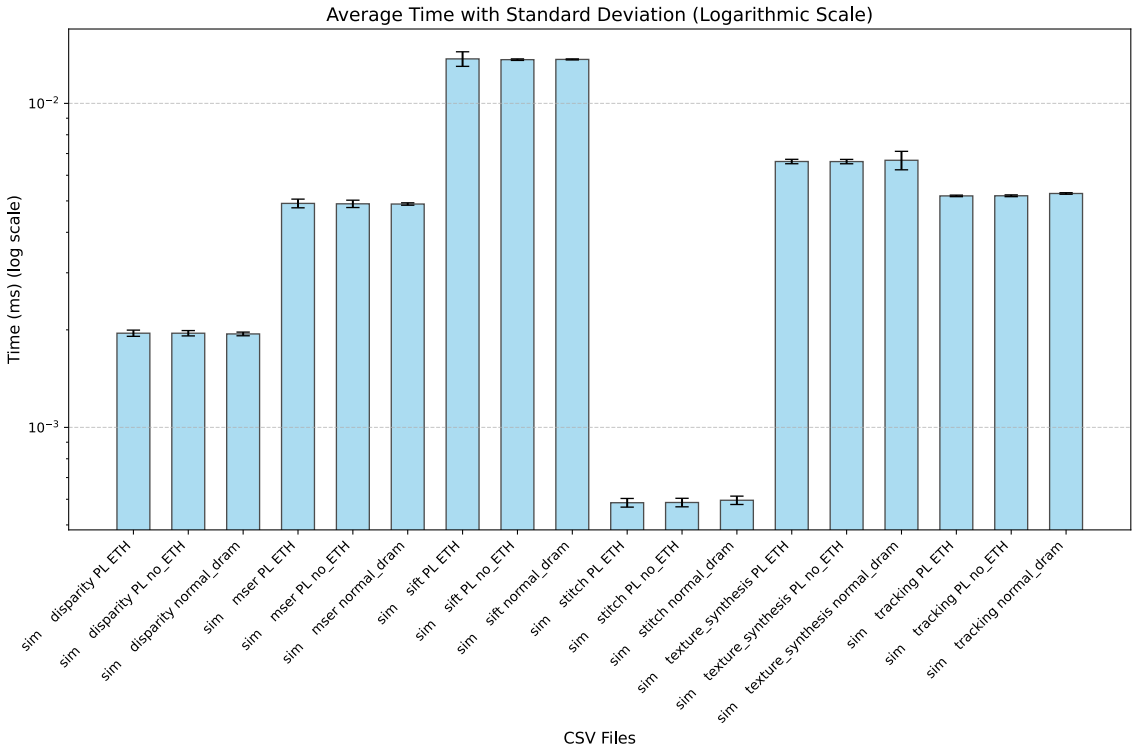


Figure 5·5: SD-VBS suite execution time comparison between different memory routes with sim image size

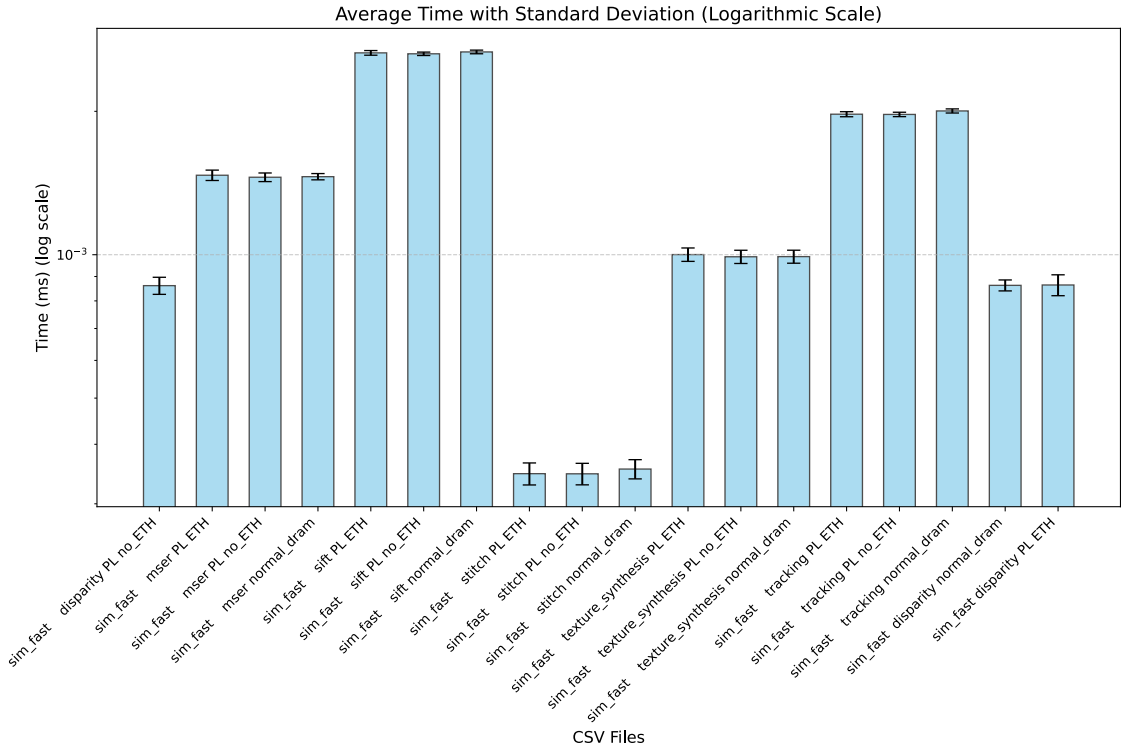
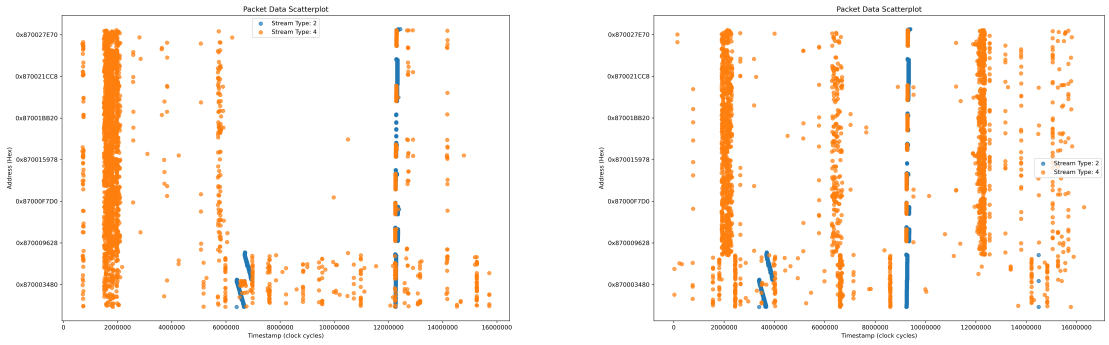


Figure 5·6: SD-VBS suite execution time comparison between different memory routes with sim$_{f}astimagesize$



(a) without interference **(b)** with interference

Figure 5·7: disparity with sim image size

duration are conducted concurrently on all cores except the primary core executing the application under observation. This methodology deliberately induces memory contention to reduce the operational frequency of the memory subsystem, thereby decreasing the frequency of responses to requests from the program under observation. Yet, We did gather a result for the ground truth of the memory access with a small memory size for bandwidth.c 4·2.

This methodological approach is exemplified through analysis of the "disparity" program under observation. Execution with and without interference demonstrates markedly different data capture characteristics; specifically, interference within the memory subsystem reduces packet loss due to diminished NIC buffer overruns resulting from decreased memory transaction response frequency. Comparable effects are observable in alternative applications such as mser A·2 and tracking A·1.

5.0.7 Limitations

While the framework of the Orchestrator is complete, implementation of submodules for the R, W, and B channels remains necessary for comprehensive AXI4Full tracing. The FrameFormer architecture was designed to transmit information immediately, even when padding was required to fulfill minimum packet length requirements.

This design decision results in potential fragmentation of metadata and addressing information across multiple packets, thereby precluding multi-threaded processing as packets lack information completeness (i.e., they are not self-contained). Consequently, packet loss provokes cascading effects within parser logic. Furthermore, the absence of transmission flow control in the FrameFormer potentially triggers Network Interface Controller (NIC) buffer overrun issues with our subscriber. As a result, the simple parser demonstrates deficiencies in identifying or appropriately recovering fragmented data portions, leading to results that inadequately reflect the system's ground truth.

Chapter 6

Future Works

There are several improvements to be made on the AoE implementation. The immediate efforts lie in implementing submodules to handle the remaining AXI channels. Similar efforts wil also create sumbmodules to handle the ACE protocol extension of AXI. We are also interested in the space of extending hardware mechanisms with remote capabilities. Primarily, efforts to extend ACE with remote capabilities to lower the implementation difficulties and overhead related to distributed memory. In addition, we would like to extend the Coresight debug infrastructure in the same manner to potentially adapt methods such as (Chen et al., 2023) and verify feasibility of orchestrating hardware on remote systems.

Chapter 7

Conclusions

This thesis study the feasibility of distributing traditionally on-board profiling techniques by repurposing high-speed external interfaces custom profiling hardware. By utilizing the highly coupled and I/O rich FPGA on the Xilinx Zynq Ultrascale+ platform, a proof of concept networking mechanism for unifying the system bus to the outside world was accomplished and studied.

The proposed solution leverages the SFP+ interface to transmit profiling data from the FPGA to a host computer, enabling real-time monitoring and analysis of system performance. This approach not only enhances the observability of the system but also provides a non-invasive method for profiling, allowing developers to gain insights into system behavior without significantly impacting performance.

The initial applications of this work will be to provide a more powerful and flexible Logic analyzer to complement or replace the traditional In-Circuit Logic Analyzer (ILA) in the Xilinx toolchain. The proposed solution is non-invasive, easy to use, and extendable, allowing for enhanced observability of the system. This mechanism can be adapted to complement other techniques in both hardware and software domains, providing a robust framework for system monitoring and debugging. Furthermore, it offers an opportunity for simpler embedded solutions to feed data into a central, powerful computer, enabling comprehensive Control Flow Integrity (CFI) checking and other advanced methods.

Despite experiencing a performance hit, it is important to note that this is not

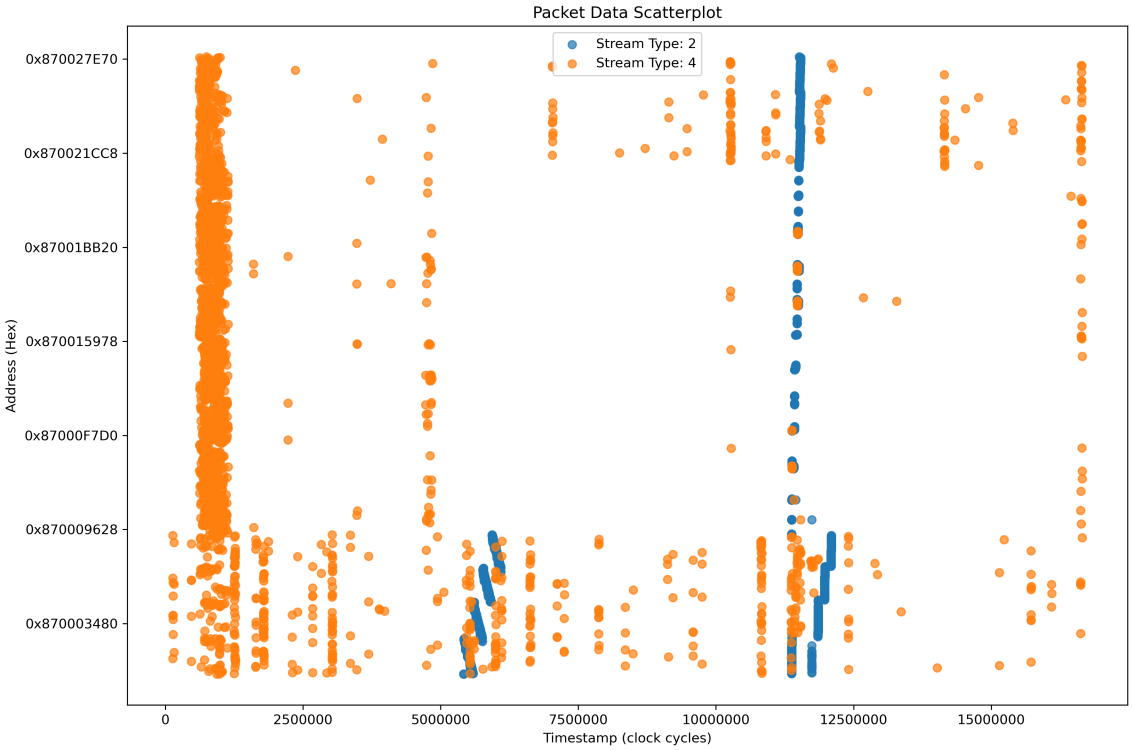
due to our module but rather the inherent penalty of the Processing Element (PE) and Programmable Logic (PL) boundary. Nevertheless, we maintain performance on par with a simple translator, demonstrating the efficiency of our approach. The ZCU platform utilized only one SFP port out of a total of four available, indicating that further exploration of these limitations could yield insights into compatibility with larger swarms of computers performing similar tasks.

Future work will not only focus on sending data but also on receiving data in the same manner, potentially enabling remote memory access. Additionally, direct interfacing with the CoreSight debug infrastructure could provide enhanced tools for system analysis and debugging, further expanding the capabilities of this profiling mechanism.

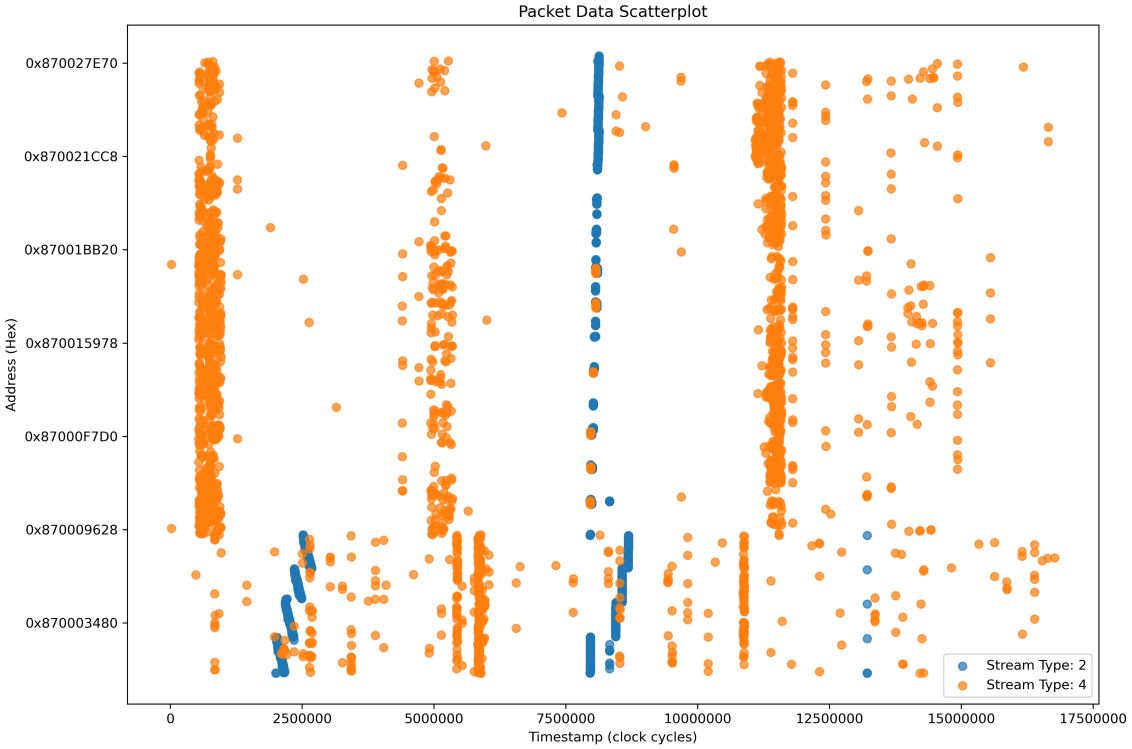
This work proposes working towards AXI over Ethernet (AoE) as a standard for profiling and debugging in heterogeneous systems, leveraging the existing infrastructure and protocols to create a unified approach to system monitoring. Through the use of the EthHelper framework we aim to establish a robust and adaptable profiling solution that can be integrated into various systems, with the initial application for profiling but with the powerful potential for broader applications of remote execution, remote memory, and remote integrity checking.

Appendix A

Additional Results

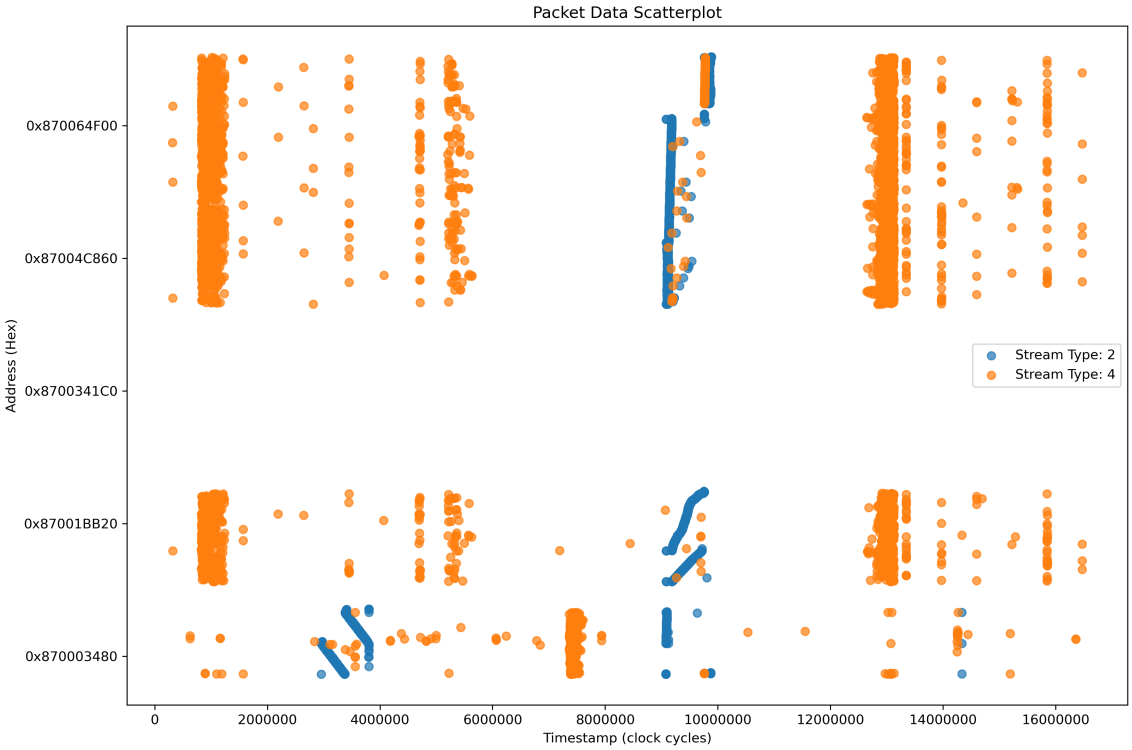


(a) without interference

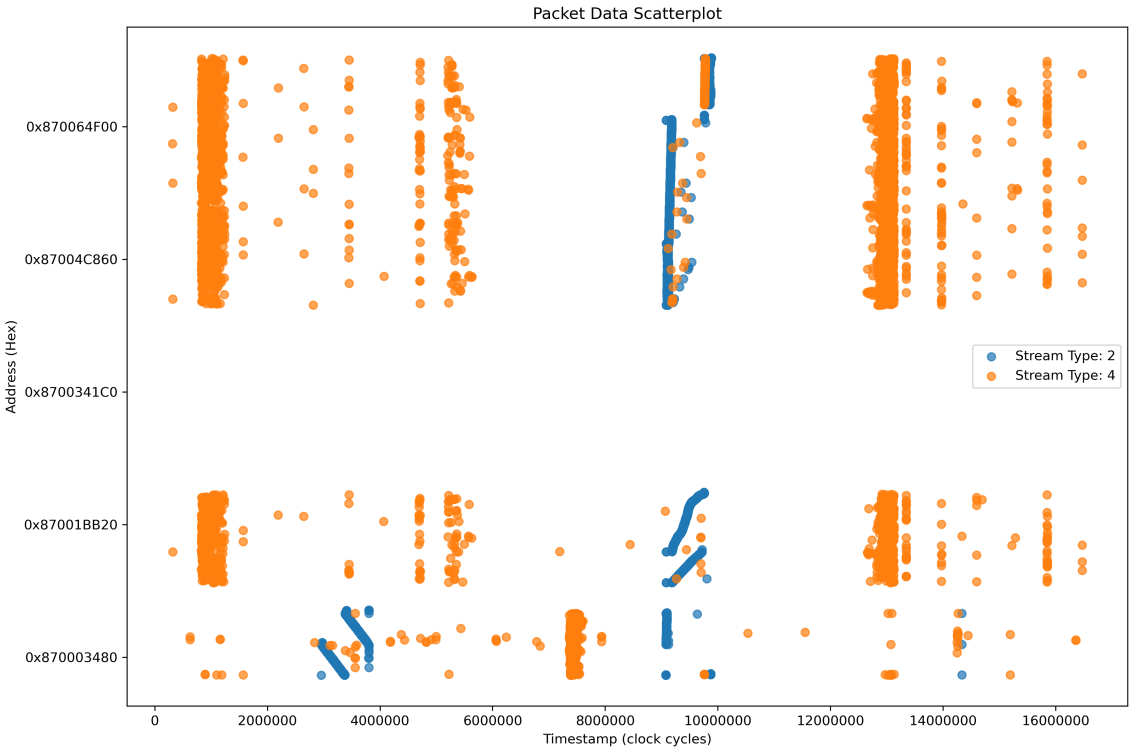


(b) with interference

Figure A·1: tracking with sim image size



(a) without interference



(b) with interference

Figure A·2: mser with sim image size

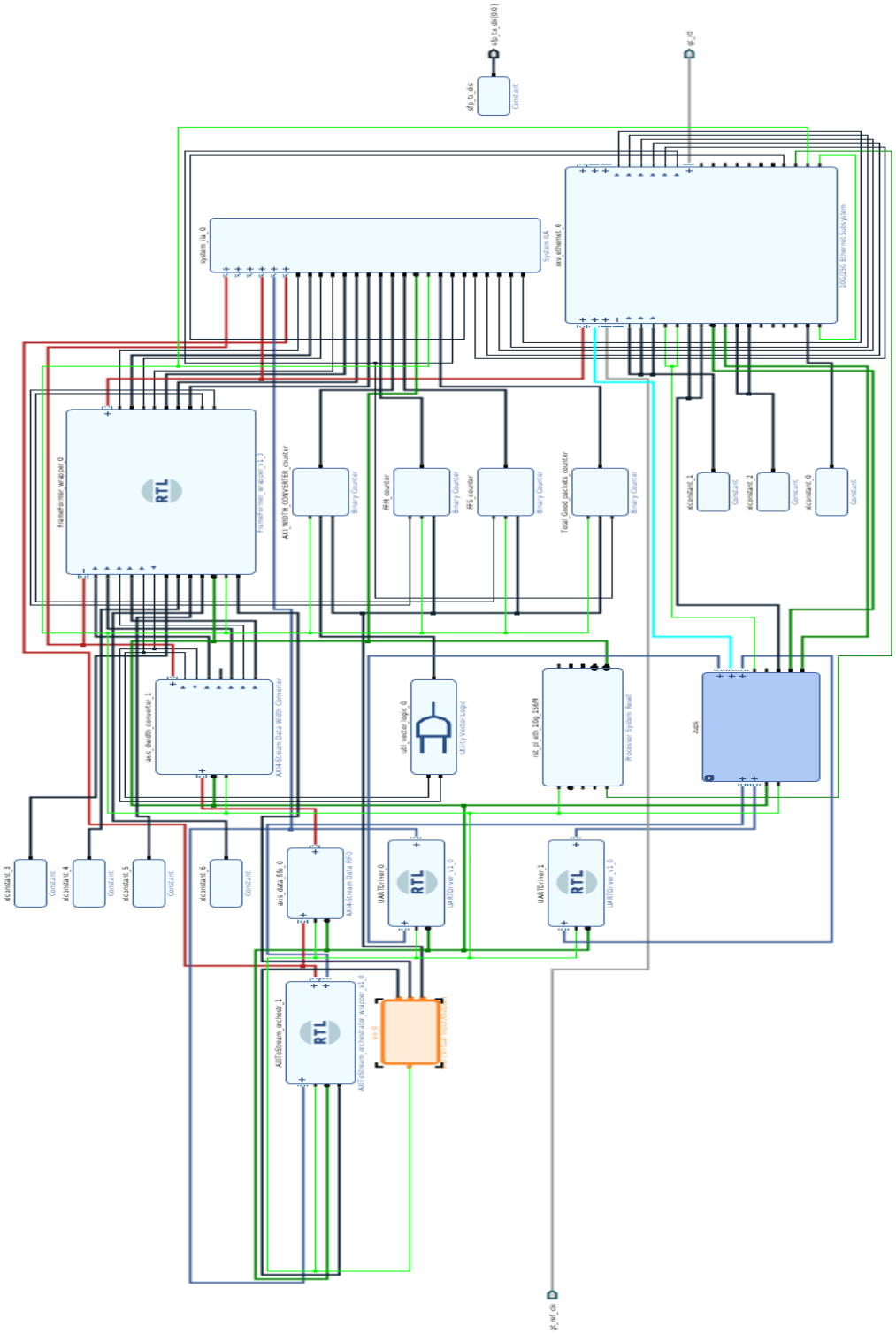


Figure A·3: Entire PL Design used for Debugging and Testing

Standard name	Memory clock (MHz)	I/O bus clock (MHz)	Data rate (MT/s)[b]	Module name	Peak transfer rate (GB/s)[c]	Timings CL-tRCD-tRP	CAS latency (ns)
DDR4-1600J* DDR4-1600K DDR4-1600L	200	800	1600	PC4-12800	12.8	10-10-10 11-11-11 12-12-12	12.5 13.75 15
DDR4-1866L* DDR4-1866M DDR4-1866N	233.33	933.33	1866.67	PC4-14900	14.9333	12-12-12 13-13-13 14-14-14	12.857 13.929 15
DDR4-2133N* DDR4-2133P DDR4-2133R	266.67	1066.67	2133.33	PC4-17000	17.06667	14-14-14 15-15-15 16-16-16	13.125 14.063 15
DDR4-2400P* DDR4-2400R DDR4-2400T DDR4-2400U	300	1200	2400	PC4-19200	19.2	15-15-15 16-16-16 17-17-17 18-18-18	12.5 13.32 14.16 15
DDR4-2666T DDR4-2666U DDR4-2666V DDR4-2666W	333.33	1333.33	2666.67	PC4-21300	21.3333	17-17-17 18-18-18 19-19-19 20-20-20	12.75 13.50 14.25 15
DDR4-2933V DDR4-2933W DDR4-2933Y DDR4-2933AA	366.67	1466.67	2933.33	PC4-23466	23.46667	19-19-19 20-20-20 21-21-21 22-22-22	12.96 13.64 14.32 15
DDR4-3200W DDR4-3200AA DDR4-3200AC	400	1600	3200	PC4-25600	25.6	20-20-20 22-22-22 24-24-24	12.5 13.75 15

Figure A·4: DDR4 Memory speeds and latencies

Program	normal dram	no ETH	ETH
disparity	8.62e-04 ± 2.30e-05	8.61e-04 ± 3.60e-05	8.64e-04 ± 4.40e-05
mser	1.46e-03 ± 2.20e-05	1.45e-03 ± 3.00e-05	1.47e-03 ± 3.70e-05
sift	2.66e-03 ± 2.30e-05	2.64e-03 ± 2.20e-05	2.65e-03 ± 3.00e-05
stitch	3.55e-04 ± 1.60e-05	3.47e-04 ± 1.80e-05	3.47e-04 ± 1.80e-05
texture_synthesis	9.91e-04 ± 3.10e-05	9.90e-04 ± 3.20e-05	1.00e-03 ± 3.20e-05
tracking	2.00e-03 ± 2.00e-05	1.97e-03 ± 2.10e-05	1.97e-03 ± 2.40e-05

Table A.1: Runtime comparison between different memory routings for sim_fast image size.

Program	normal dram	no ETH	ETH
disparity	1.94e-03 ± 2.50e-05	1.95e-03 ± 3.70e-05	1.95e-03 ± 4.30e-05
mser	4.89e-03 ± 4.20e-05	4.90e-03 ± 1.28e-04	4.91e-03 ± 1.52e-04
sift	1.37e-02 ± 4.90e-05	1.36e-02 ± 7.40e-05	1.37e-02 ± 7.10e-04
stitch	5.96e-04 ± 1.80e-05	5.86e-04 ± 1.80e-05	5.85e-04 ± 1.80e-05
texture_synthesis	6.67e-03 ± 4.37e-04	6.61e-03 ± 1.02e-04	6.62e-03 ± 1.04e-04
tracking	5.27e-03 ± 2.90e-05	5.19e-03 ± 3.30e-05	5.18e-03 ± 2.80e-05

Table A.2: Runtime comparison between different memory routings for sim image size.

References

Abera, T., Asokan, N., Davi, L., Ekberg, J.-E., Nyman, T., Paverd, A., Sadeghi, A.-R., and Tsudik, G. (2016). C-flat: Control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 743–754, New York, NY, USA. Association for Computing Machinery.

Aguilera, M. K., Amit, N., Calciu, I., Deguillard, X., Gandhi, J., Subrahmanyam, P., Suresh, L., Tati, K., Venkatasubramanian, R., and Wei, M. (2017). Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 121–127, New York, NY, USA. Association for Computing Machinery.

Ammar, M., Abdelraoof, A., and Vlasceanu, S. (2024). On bridging the gap between control flow integrity and attestation schemes. In *Proceedings of the 33rd USENIX Conference on Security Symposium*, SEC '24, USA. USENIX Association.

ARM (2013). Arm architecture reference manual for a-profile architecture.

ARM Ltd. (2004). Coresight components technical reference manual.

ARM Ltd. (2010). CoreSight trace memory controller technical reference manual.

ARM Ltd. (2012). Embedded trace macrocell architecture specification etmv4.0 to etm4.6.

ARM Ltd. (2018). Armv6-m architecture reference manual.

Ashraf, I., Taouil, M., and Bertels, K. (2015). Memory profiling for intra-application data-communication quantification: A survey. In *2015 10th International Design Test Symposium (IDT)*, pages 32–37.

Aysu, A., Gaddam, S., Mandadi, H., Pinto, C., Wegryn, L., and Schaumont, P. (2016). A design method for remote integrity checking of complex pcbs. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1517–1522.

Basile, C., Di Carlo, S., and Scionti, A. (2012). Fpga-based remote-code integrity verification of programs in distributed embedded systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(2):187–200.

Bellec, N., Rokicki, S., and Puaut, I. (2020). Attack Detection Through Monitoring of Timing Deviations in Embedded Real-Time Systems. In Völp, M., editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:22, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Bruening, D., Zhao, Q., and Amarasinghe, S. (2012). Transparent dynamic instrumentation. *SIGPLAN Not.*, 47(7):133–144.

Calciu, I., Puddu, I., Kolli, A., Nowatzyk, A., Gandhi, J., Mutlu, O., and Subrahmanyam, P. (2019). Project pberry: Fpga acceleration for remote memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 127–135, New York, NY, USA. Association for Computing Machinery.

Chen, W., Izhbirdeev, I., Hoornaert, D., Roozkhosh, S., Carpanedo, P., Sharma, S., and Mancuso, R. (2023). Low-Overhead Online Assessment of Timely Progress as a System Commodity. In Papadopoulos, A. V., editor, *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, volume 262 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:26, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Ciraolo, F., Nicolella, M., Hoornaert, D., Caccamo, M., and Mancuso, R. (2025). Light virtualization: a proof-of-concept for hardware-based virtualization.

Feng, L., Huang, J., Hu, J., and Reddy, A. (2021). Fastcfi: Real-time control-flow integrity using fpga without code instrumentation. *ACM Trans. Des. Autom. Electron. Syst.*, 26(5).

Hoppe, A., Becker, J., and Kastensmidt, F. L. (2021). High-speed hardware accelerator for trace decoding in real-time program monitoring. In *2021 IEEE 12th Latin America Symposium on Circuits and System (LASCAS)*, pages 1–4.

Intel Corporation (2022). Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide.

Izhbirdeev, I., Hoornaert, D., Chen, W., Zuepke, A., Hammad, Y., Caccamo, M., and Mancuso, R. (2024). Coherence-aided memory bandwidth regulation. In *2024 IEEE Real-Time Systems Symposium (RTSS)*, pages 322–335.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York, NY, USA. Association for Computing Machinery.

Mizutani, K., Yamaguchi, H., Urino, Y., and Koibuchi, M. (2021). Optweb: A lightweight fully connected inter-fpga network for efficient collectives. *IEEE Transactions on Computers*, 70(6):849–862.

Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100, New York, NY, USA. Association for Computing Machinery.

Nicolella, M., Roozkhosh, S., Hoornaert, D., Bastoni, A., and Mancuso, R. (2022). Rt-bench: An extensible benchmark framework for the analysis and management of real-time applications. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, RTNS 2022, page 184–195, New York, NY, USA. Association for Computing Machinery.

Roozkhosh, S. and Mancuso, R. (2020). The potential of programmable logic in the middle: Cache bleaching. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 296–309.

Scales, D. J., Gharachorloo, K., and Thekkath, C. A. (1996). Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. *SIGPLAN Not.*, 31(9):174–185.

Sidler, D., Wang, Z., Chiosa, M., Kulkarni, A., and Alonso, G. (2020). Strom: Smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA. Association for Computing Machinery.

Valsan, P. K., Yun, H., and Farshchi, F. (2016). Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12.

Venkata, S. K., Ahn, I., Jeon, D., Gupta, A., Louie, C., Garcia, S., Belongie, S., and Taylor, M. B. (2009). Sd-vbs: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64.