Hardware Accelerated Trace Analysis for Compiler Optimizations

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 441

Systems Group, Department of Computer Science, ETH Zurich

Hardware Accelerated Trace Analysis for Compiler Optimizations

by
Matthew E. Weingarten

Supervised by
Prof. Dr. Timothy Roscoe, Dr. David Cock, Nora Hossle

October 2022–May 2023

D INFK

Abstract

Profiling an application at runtime with hardware support plays an irreplaceable role in monitoring, debugging, runtime verification, and optimizing application performance. Most modern processors are designed with dedicated hardware resources that produce on-the-fly profiling data while incurring negligible overhead. However, the volume of the resulting profiling data can enormous, where a single core can produce upwards of 100 MB/s of data, leading to high storage and offline processing costs. Current approaches focus on reducing profiling data bandwidth with sampling-based techniques at the cost of execution details.

In this work, we introduce a hardware-accelerated profile decoder and analyzer for the ARM Coresight debug and trace architecture that can process execution traces produced by Coresight components in real-time. Profiles collected from a CPU are forwarded to a decoder implementation on a Field Programmable Gate Array (FPGA) that is part of a hybrid FPGA/CPU platform, meaning profiling data is processed on-chip. This technique can produce richer profiling data, potentially enabling more powerful optimizations, while simultaneously eliminating the associated storage and post-processing costs and avoiding the need for throttling the bandwidth of profile collection.

Our novel trace decoding process can handle the compressed instruction trace data stream that follows the Embedded Trace Macrocell (ETM) specification at 1 GB/s per core, increasing the throughput over prior work by 8×. We also greatly increase the reliability of the decoding process by removing the possibility of dropping data and the requirement for buffers, while keeping the area costs comparable.

Contents

List of Figures

List of Tables

Listings

List of Algorithms

List of Terms

AMBA	Avanced Microcontroller Bus Architecture
AOT	Ahead Of Time
APB	Advanced Peripheral Bus
APSR	Application Program Status Register
APU	Application Processing Unit
ATB	AMBA Trace Bus
ATF	Arm Trusted Firmware
AXI	Advanced eXtensible Interface
BC	Bounded Continuous
BD	Block Design
CBC	Cache Block Controller
CCPI	Coherent Processor Interconnect
CFG	Control Flow Graph
CID	Context Identifier
CS	Coresight
CSAL	CoreSight Access Library
CTI	Cross Trigger Interface
CTM	Cross Trigger Matrix
DAB	Debug Access Bus
DAP	Debug Access Port
DDR4	Double Data Rate Synchronous 4
DMA	Direct Memory Access
DMC	Direct Memory Controller
DS	Data Sheet
DTX	Debug Bus Transmitter
ECI	Enzian Coherency Interconnect

ECT . . . . . Embedded Cross Trigger

ELF . . . . . Executable and Linkable Format

EMIO . . . . . Extended Multiplexed I/O

ETB . . . . . Embedded Trace Buffer

ETF . . . . . Embedded Trace FIFO

ETM . . . . . Embedded Trace Macrocell

ETR . . . . . Embedded Trace Router

FDO . . . . . Feedback-Directed Optimization

FPGA . . . . . Field-Programmable Gate Array

FS . . . . . Fixed Size

FSBL . . . . . First Stage Boot Loader

FSM . . . . . Finite State Machine

FTM . . . . . Fabric Trigger Macrocell

GPIO . . . . . General Purpose Input/Output

GWP . . . . . Google Wide Profiling

HO . . . . . Header Only

IoD . . . . . ID or Data

IP . . . . . Intellectual Property

IR . . . . . Intermediate Representation

ISA . . . . . Instruction Set Architecture

ISB . . . . . Instruction Synchronization Barrier

ITM . . . . . Instrumentation Trace Macrocell

JIT . . . . . Just In Time

JTAG . . . . . Joint Test Action Group

LBR . . . . . Last Branch Records

LSB . . . . . Least Significant Bit

LUT . . . . . Lookup Table

MM2S . . . . . Memory Mapped to Stream

MMCM . . . . . Mixed-Mode Clock Manager

MSB . . . . . Most Significant Bit

NLP . . . . . Next Line Prefetcher

OCLA . . . . . On Chip Logic Analyzer

OO . . . . . Object Oriented

OoO	Out of Order
openCSD	open source CoreSight Decoding Library
OS	Operating System
P0	Control flow instructions, like branch instruction
P1	Load and store instructions
PCIe	Peripheral Component Interconnect Express
PE	Processing Element
PEBS	Processor Event Based Sampling
PGO	Profile-Guided Optimization
PL	Programmable Logic
PMU	Performance Monitoring Unit
PoR	Power on Reset
PS	Processing Subsystem
PT	Processor Trace
PTM	Processor Trace Macrocell
QPS	Queries Per Second
QSPI	Quad Serial Peripheral Interface
ROM	Read only Memory
SoC	System on a chip
SRAM	Static Random Access Memory
STM	System Trace Macrocell
tcl	The Tool Command Language
TLB	Translation Lookaside Buffer
TMC	Trace Memory Controller
TPIU	Trace Port Interface Unit
TRM	Technical Reference Manual
UART	Universal Asynchronous Receiver-Transmitter
UC	Unbounded Continuous
US+	Zynq Ultrascale+ MPSoC
VMID	Virtual Context Identifier
WSC	Warehouse Scale Computer
XSA	Xilinx Support Archive
XSDB	Xilinx System Debugger

Chapter 1

Introduction

Collecting the runtime trace of an application is a form of profiling that provides detailed insight into the precise execution behavior of a program. With a runtime trace, the entire control flow path of an application can be reproduced for a given input. In other words, a trace provides information on each control flow-changing instruction that has been executed and how the flow is altered. However, tracing a program is expensive — to make it feasible to collect traces at runtime without taking an unsustainable hit to performance requires hardware assistance. The Coresight architecture introduced by ARM specifies a set of dedicated hardware components that are tightly coupled to the processor and provide zero-overhead debugging and tracing functionality for System on a chips (SoCs) [63, 20, 27]. Coresight has garnered attention mostly in the field of runtime verification [74, 37, 39], debugging [79], and even security [61]. Yet Coresight remains sparsely studied as a tool for performance optimization. In part, this is because Coresight is not available for many processors used commonly in high-performance computing or Warehouse Scale Computer (WSC), which is still dominated by Intel and AMD [62]. Furthermore, different systems that support Coresight may have widely different implementations of the Coresight architecture, requiring system-specific integration with Coresight, which has been used as an argument against profiling with Coresight [47]. Yet Coresight remains a compelling source of profiling data, as it provides a very rich set of profiling features that are not found in other hardware profiling facilities. We predict a growing interest in Coresight as both server-grade ARM processors and heterogeneous systems become more common.

One of the best use cases for profiling data is feeding them to a compiler, which can then use Profile-Guided Optimizations (PGOs) to generate more performant code. Profiles generally provide runtime information to a compiler, enabling optimizations that go beyond more traditional static analysis techniques [68, 86]. PGO traditionally involves a manual three-stage compilation process: compiling an instrumented binary, running the instrumented binary while collecting profiles, and using the profiles to apply optimizations while recompiling. Recently, PGO has evolved to continuously profile live applications and incrementally apply optimizations for new binary releases. This approach is a staple in larger scale datacenters [67, 35]. Recent work has even gone as far as to apply the optimizations *during* the execution of a program [91]. The large amount of effort spent on feeding this always-on and data-hungry machine of constant recompilation with freshly acquired profiling data warrants a re-examination of the current profile collection techniques.

To provide a sense of scale, Google reports collecting around 600GB of compressed profiling data each day, requiring around 8 hours of post-processing on 400 machines [28]. This data is also heavily cherry-picked, as each machine in the data center is only profiled for around 10s per day, and only data on the 1000 hottest binaries is considered to be of interest. This means that only a fraction of the total executed cycles are profiled. Large amounts of profiling data are discarded to reduce the volume of collected data. This begs the question, can we use hardware acceleration to make this more efficient and to cut down on storage costs? Furthermore, hardware acceleration for profile decoding and analysis makes it possible to collect more detailed profiling data. But do more detailed execution profiles result in even more performant code? To the best of our knowledge, the latter question has not been extensively explored, as current approaches focus on applying optimizations with as little profile information as possible to keep profiling data volume to a minimum.

For our work, the source of the trace data is an Embedded Trace Macrocell (ETM), a Coresight component that is responsible for tracing the control flow of an application running on a processor. The data from an ETM follows a trace stream specification, is heavily compressed, and requires multiple expensive post-processing steps to reconstruct the control flow

details so that they are actionable to an optimizing compiler. We focus on applying trace analysis in real-time in hardware on a hybrid CPU/FPGA platform, where the Field-Programmable Gate Array (FPGA) is used as a hardware accelerator to decode and analyze trace data received from the dedicated trace and profiling components.

The bulk of our efforts have gone into the design of a trace decoder for the ETM specification. The trace data is non-trivial to parse in hardware at high data rates as the trace data is hard to process in parallel. This is because the trace protocol is a packet-based protocol that is sent from the processor to the FPGA as a byte stream. Reconstructing packets from the byte stream has to be done ad-hoc, as the structure of a packet (for example how large a packet is) can in some cases only be determined once an entire packet is parsed. Additionally, trace data sent out by an ETM relies on internal state registers to compress the trace stream and the state registers must be mirrored by the trace decoder. Some packets rely on these state register updates from a previous packet.

All Prior trace decoders have used a buffering-based system to reconstruct packets from the byte stream [90, 74, 89]. However, these implementations suffer from low throughputs for specific packet types and do not always support the full specification and all possible packet sizes. The trace decoder we introduce in this work solves these problems by parallelizing the trace data processing with a sophisticated unrolling technique and applying a decoder byte-wise instead of packet-wise.

We additionally gained insights into both the Coresight architecture on the Zynq Ultrascale+ MPSoC and ThunderX. The Zynq Ultrascale+ MPSoC has a complex topology that requires extending the existing CoreSight Access Library (CSAL), an API for interacting with the Coresight subsystem maintained by ARM [2]. Large parts of the ThunderX Coresight system is unique to the ThunderX design and a custom Coresight driver must be written. We elaborate on the internal debug components of the ThunderX and outline how the trace data is propagated through the processor. As of now, we are still unable to access any trace data from the ETMs on ThunderX.

1.1 Contributions

This thesis aims to explore the possibilities of using Coresight as a means of profile collection, which would then be integrated into an optimizing compiler that performs sophisticated PGOs. Concretely our research question is:

— *How can we design a system that can handle the traces produced by the ETM tracing units in real-time on an FPGA?*

We list the following contributions:

— An (almost) end-to-end system for collecting and decoding trace data from Coresight components that include programming the Coresight infrastructure on the Processing Subsystem (PS) (Section 4.3), driving the trace data to the Programmable Logic (PL) (Section 4.3.2), demultiplexing the trace data received on the PL into streams of traces from each core (Section 4.4.1 - Section 4.4.4), and finally decoding the instruction trace stream (Section 4.5).

— A comprehensive guide to the general Coresight architecture (Chapter 3), and device-specific Coresight architectures of the ThunderX (Chapter 5) and Zynq Ultrascale+ MPSoC (Section 4.2.1) devices.

— An extension to the CSAL to enable trace session on complex Coresight topologies and the programming of intermediate buffers in hardware FIFO mode.

— An in-depth explanation of the ETMv4 specification and the challenges of implementing a hardware decoder (Section 4.5.1).

— A performance and resource utilization evaluation of our trace decoder implementation in comparison to prior work (Chapter 6). We show a throughput performance increase of 8× over prior implementations (Chapter 7).

1.2 Thesis Layout

Chapter 2 is a survey on current profile collection techniques and what types of optimizations rely on profiling data and Chapter 3 gives background information on the Coresight architecture and Coresight components. We discuss the device-specific Coresight implementations on the ThunderX in Chapter 5 and on the Zynq Ultrascale+ MPSoC in Section 4.2.1. The implementation is covered in Chapter 4. This section contains a subsection devoted to the ETM trace decoder implementation,

Section 4.5, a section on programming the Coresight components, Section 4.3, and a section handling the raw trace data from Coresight before passing the trace data to the trace decoders in Section 4.4. Our decoding process is evaluated in Chapter 6 and compared to prior implementations in Chapter 7. We conclude the thesis in Chapter 9 and discuss future research directions in Chapter 8.

Chapter 2

PGO & Profiling Landscape

Collecting information on the runtime behavior of an application running on a machine or system is referred to as *profiling*. The collected runtime information can be used to monitor the program execution, verify runtime properties, break down the performance characteristics, and ultimately enable either manual or automatic optimizations. Any optimization that relies on runtime information is called Profile-Guided-Optimization (PGO) or Feedback-Directed Optimization (FDO). This work involves hardware acceleration of zero-overhead profile collection. In order to understand the design implications of a hardware accelerator, we must be aware of the current landscape of profile collection techniques and to what end these profiles are used to perform optimizations.

We categorize and describe different profile collection techniques in Section 2.1, discuss commonly collected metrics Section 2.2 and survey the most prolific types of optimizations that rely on profiling data Section 2.3. We further describe some dedicated profiling hardware Section 2.5 and compare this to Coresight, give a system-wide view of profiling systems applied in datacenters Section 2.4, and conclude with a short discussion on the responsibilities of a compiler vs dedicated hardware components that also perform a variant of PGO.

2.1 Profile Collection

Traditionally, profile collection techniques fall either under the category of *instrumentation-based profiling* or *sampling-based profiling* [85]. We further add the category of *trace-based profiling*.

Instrumentation-based profiling. Instrumentation of an application involves injecting instrumentation code that does bookkeeping, for example, incrementing counters or recording timing metrics. Instrumentation is usually seen as a form of software profiling and is an architecture-neutral approach to profile collection. The power of instrumentation comes from being able to collect arbitrary information during runtime. The designer of the instrumentation code has full control over what metrics to profile. This, however, comes at a cost. Every cycle spent on running the instrumentation code is a cycle that is not spent on doing "useful" work. On top of this, the injection of instrumentation code can break the structure of the code and consecutively changes the resulting binary when run through a compiler (if the instrumentation is not inserted post-compilation). Specifically, instrumentation can cause damage to loop structures, affect inlining decisions made by the compiler and more. In other words, the profiles collected by instrumented code can only provide an approximation of the runtime behavior of the non-instrumented application. As a result, instrumentation-based profiling is usually considered to provide the richest profiling data but has the highest overhead. Of course, this depends on the specific instrumentation implementation and hardware support for instrumentation is available (see Coresight Instrumentation Trace Macrocell (ITM) in Section 3.3).

Sampling-based profiling. On the other hand, sampling-based profiling regularly reads from hardware registers that are designed with profile collection in mind. The most common sampling-based techniques read from the PMU (Section 3.5) to observe hardware events of the underlying processor. Perf tools [7, 11] provide a software abstraction layer onto the hardware for this. A common profile collection approach is to deploy lightweight daemons constantly running on a machine and

collecting data [73, 35, 67]. The main benefit of the sampling-based approaches is the negligible overhead incurred by the profile collection stages, while still giving reasonable amounts of insight into the performance characteristics.

Trace-based profiling. In addition to sampling and instrumentation approaches, we distinguish a third type of profile collection, namely the *trace-based profiling*. A trace records the entire control flow of a running application, meaning the exact execution path through the Control Flow Graph (CFG) of the binary. The collection of such traces without processor hardware support is infeasible as it would result in a major performance hit. Dedicated hardware support is available for this purpose, which includes ARM's Coresight infrastructure [63], the topic of this thesis, and Intel's Processor Trace (PT) [49]. However, if the generation of the traces itself has low or zero overhead, processing or storing the collected profiling data is a daunting task due to the sheer volume of trace data that can be produced by a tracing session, which can be in the range of hundreds of MB per second per core. Modern sampling-based approaches come quite close to trace-based approaches in accuracy, especially when aggregated across a large number of runs. Intel's Last Branch Records (LBR) additionally provides a further hardware component that can be sampled making it easier to reconstruct control flow. However, sampling-based approaches still suffer from sampling only the hottest code regions, so the entire execution history cannot necessarily be reconstructed.

2.2 Profile Types

In section, we discuss exactly what types of profiles a profiler may collect. We break this down into branch frequencies, hardware events and high-level language constructs.

Branch frequencies. Most commonly a profiler will collect *branch frequencies*. Branch frequency profiling data will hold aggregated runtime information on how the program behaves at a given branch. Branch frequency profiles will give an estimate of how likely the branch will be either taken or not taken. More specifically, for any given control flow changing instruction b, the profiles give us frequency $f_e(b)$, where $f_e(b)$ is the frequency a branch is taken and the not-taken frequency $f_n(b)$ is $1 - f_e(b)$.

Runtime trace. Runtime traces are a superset of the branch frequency profiling data. A trace gives global branch relations. Instead of just $f_e(b)$, we can compute the frequency a branch is taken based on the execution path p ending in branch b, so $f_e(b|p)$. Usually, the runtime trace is constrained to a subset of the complete execution path, in a window p_w of the last w executed branch instructions of path p, giving us $f_e(b|p_w)$.

Hardware events. While branch frequencies give information on the execution behavior an application exhibits, profiling hardware counters gives us information on how well an application is using the system resources. For example, a profile may track counters for cache misses, or memory accesses. Based on the counters, optimizations may be focused on specific regions of code. Furthermore, tracking hardware events in addition to overall execution time can be used to evaluate how much an application is benefitting from a given set of optimizations.

High-level language constructs. Often it is beneficial to collect profiling data on high-level language constructs that provide a language-level abstraction to a developer. An example of this would be polymorphism and virtual calls for C++ or Java. Polymorphism provides an abstraction layer for writing high-level code but comes at a performance hit if unoptimized. For one, a jump to the body of a virtual call requires an additional level of indirection through the virtual table lookup, and second, a compiler is often unable to inline virtual calls, missing out on optimization potential. To combat the overhead of virtual calls, profiling data is used to determine the likely runtime type of the receiver. If the likelihood is high that a receiver of a virtual call is always the same type, a compiler may speculatively optimize for this runtime type, a process which has been dubbed devirtualization [51]. Typically, the best way to get this profiling data is to instrument call sites. Most high-level language constructs are profiled with instrumentation-based techniques. However, this is not always the case, another option is to hoist the profiling data to the source code level. However, this is not always possible as some source code information is lost during compilation and the retranslation is not always possible, or very expensive. This is especially true when the structure of the code is changed during compilation, like with loop unrolling or code duplication.

2.3 Typical Optimizations

In this subsection, we survey some additional common compiler optimizations that also rely on profiling data.

The most common motivation to perform PGO has been to target the ever-growing pressure on instruction caches for server workloads. Many cycles in data centers are wasted in front-end stalls [53]. A combination of hot-cold code splitting, code reordering, and software cache prefetching has been used effectively to mitigate this issue [28, 67, 91, 57, 55]. Hot-cold code splitting makes sure that cache space is not wasted on instructions that rarely get executed, code reordering increases the hit rate of instruction caches when jumping from one basic block (or function) to another and software cache prefetching is used to avoid cache misses when the target of long jumps can be predicted with the profiling data. All of these optimizations cannot be performed without accurate and up-to-date branch frequency profiles and directly benefit from more accurate profiling data.

Another source of more complex PGO examples can be found in the GraalVM compiler, which is an optimizing compiler for JavaVM-based languages and performs many optimizations to eliminate the runtime cost of high-level language constructs. Typically, these languages are Just In Time (JIT) compiled and the runtime environment will collect profiles on the fly and recompile sections of the code if they are hot. However, GraalVM native image provides the ability to Ahead Of Time (AOT) compile JavaVM-based languages [87]. Many optimizations typically applied when JIT compiled are lost without profiling data and native image relies on a manual three-stage compilation process to bring the performance of native image close to its JIT compiled counterpart. GraalVM relies heavily on profiling data to make inlining choices that enable further optimizations [70], duplicates and specializes code regions with speculative optimizations based on profile data [76, 64], and uses profiles in static analysis for reducing heap allocations [83] as a few examples.

2.4 Profile Collection in Datacenters

In Chapter 1 we highlighted some metrics on profiling data collection reported by Google. Here we give a general overview of how these profile collection techniques work as a system for continuously applying optimizations for new binary releases. Figure 2.1 shows a simplified profile collection system for large-scale data centers that applies to AutoFDO [35], Google Wide Profiling (GWP) [73] and Bolt [67].

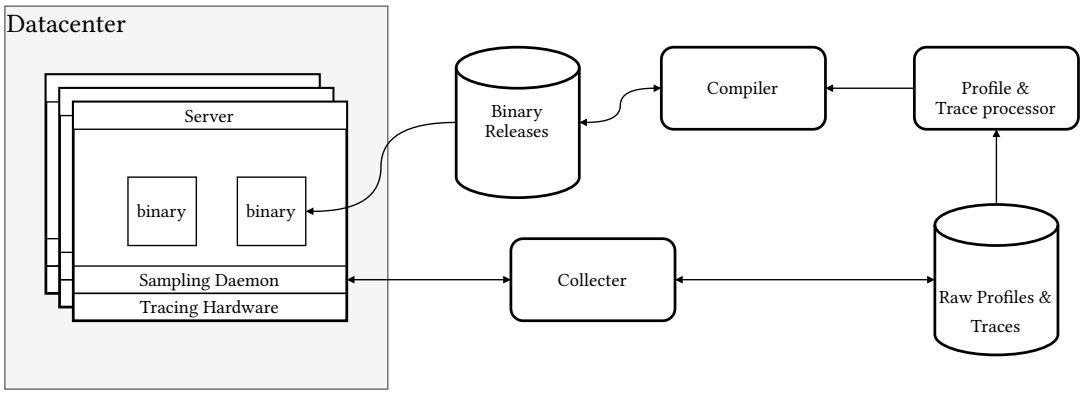


Figure 2.1: Collecting profiles in datacenters.

Every server will have a profile collection daemon that samples the tracing or profiling hardware. How often samples are collected depends on the sampling policies that trade-off profiling data details for profiling data volume. Profiling data is then collected and stored in a database. Raw profiling data goes through an offline post-processing stage. This post-processing stage involves symbolizing the profiling data, attaching debug data, and ultimately hoisting the profiling data to a source code or Intermediate Representation (IR) level. Sometimes more complex post-processing steps are performed, commonly a loop reconstruction algorithms like Havlaks algorithm [46, 82] are applied in the post-processing stage.

Once the profiling data is brought into the form that is actionable by a compiler, the binary is recompiled with the injected profiling data. The binary is now adapted to fresh runtime information and is specialized. Therefore the binary is expected to

need fewer cycles to perform the same task as before. On average, the runtime improvements are between 6%-15% [35, 67, 28], but can sometimes reach up to a 30% [35].

In this thesis, we target hardware acceleration for both the collecting of traces and offline post-processing steps. This could reduce the amount of profiling data that needs to be stored, since raw profiling data is very verbose, or eliminate the need for storage entirely.

2.5 Profiling Hardware

In this section, we discuss what hardware components are most commonly used for profiling and have seen the most attention in recent research.

Intel's LBR has played a profound role in the field of PGO. The LBR can be read from software to collect information on the last taken branches [49]. Typically, the LBR will hold records of the last 16-32 branches in a ring buffer. Each entry will hold the target address a_t and the source address a_s of a jump (Figure 2.2). Updates to the LBR are made in parallel to the execution of the branch instructions and do not affect the execution of a processor. When the buffer overflows, the oldest branch records are overwritten and lost, which makes it ideal for sampling-based profiling techniques. The overhead incurred by collecting profiles from the LBR depends on the sampling frequency.

The LBR profiles more than just branch frequencies, but essentially shows a snapshot of the runtime trace, allowing us to compute branch frequencies $f_e(b|p_w)$, where the window size is between 0 and LBR ring buffer size.

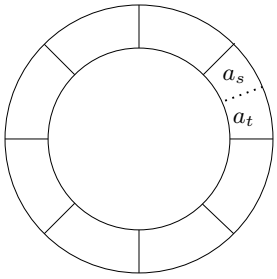


Figure 2.2: Intel's Last Branch Record for profiling branch frequencies.

It is not surprising that most runtime profiling systems rely on LBR, as it is very simple to setup an LBR sampler and branch frequencies can be computed straightforwardly from aggregated LBR values. The authors of Bolt have performed experiments both with and without access to an LBR and have shown that an LBR is responsible for around 30% of the performance improvements, being on average 6% with LBR and only 4% percent without LBR [67]. Recent efforts have gone into emulating the ability of an LBR with instrumentation and architecture-neutral approaches [62].

The LBR cannot be directly compared to a Coresight ETM. Instead, Intel also has similar components, Intel Processor Trace (PT) that can produce profiling data for every instruction and is analogous to an ETM, but with less features [49].

Some Intel processors provide insight into hardware events with Processor Event Based Sampling (PEBS) [50, 49]. This is somewhat comparable to the Coresight Embedded Cross Trigger Interface combined with a PMU, but also does not provide nearly the same level of flexibility and has fewer hardware events to track. PEBS generates interrupts on hardware events, for example, an `INSTRUCTION_RETIRED` . Custom interrupt handlers can be written to collect profiling data. PEBS has seen an interesting use case for optimizing indirect branches with runtime binary patching [19]. The Coresight architecture is well suited for any type of optimizations that rely on PEBS.

2.6 Discussion — Architecture vs Compilers

The name PGO is most commonly used in the context of compiler optimizations. Optimizing compilers such as the LLVM framework [60], GCC [4], and GraalVM [87] all have some form of PGO support. However, the concept of PGO is not unique

to compilers — branch predictors and cache prefetchers may use execution history or knowledge from previous runs to make predictions and avoid branch mispredictions and cache misses respectively [44, 30, 78, 13].

Examining the case study of instruction cache prefetches, both profile-guided compiler optimizations and hardware prefetchers aim to solve the same problem, namely improving cache hit rate, which begs the question: What is the main responsibility of the compiler vs the responsibility of the architecture? Are both required? Can a hardware prefetcher avoid cache misses that a compiler cannot and vice versa?

A full discussion on this topic is outside of the scope of this work and our knowledge, and there is no clear-cut answer for every use case. However, from our perspective, architectural components should be viewed as *general-purpose* optimization techniques. We speculate that is why the most common prefetchers widely believed to be employed in most server-grade processors are Next Line Prefetchers (NLPs) [28], meaning they do not "learn" from previous execution behavior. This is a great example of general-purpose optimizations since NLPs cover the most likely case for any application, as sequential execution of instructions is much more common than long jumps.

PGO done through compilers should be viewed as *application-specfic* optimization techniques that can exploit the common behavior of a single application and disregard the common case. For example, a compiler can insert software prefetches that an NLP would not be able to handle based on previous profiling runs and knowledge of a specific application [55, 28].

In summary, we wish to draw attention to the fact that using profiling data is not owned by compilers and we clearly distinguish the purpose of application-specific compiler optimizations as opposed to general-purpose architectural optimizations.

Chapter 3

Coresight Architecture

The ARM Coresight architecture introduces a family of programmable tracing and debug components (or devices) to a hardware system. Coresight's purpose is to make embedded systems and software engineering for SoC more efficient both in terms of engineering effort and for assisting in optimizations to enable more efficient use of the underlying hardware [63]. Coresight can enable optimizations that rely on instruction & data tracing, code profiling and observation of hardware events. While also Coresight's main selling point, the provided system-wide full execution transparency with a somewhat loose and extensible specification, comes at a cost — a high barrier of entry and the requirement of system-by-system solutions. Two seemingly similar SoCs, that both implement the Coresight subsystem may have widely disparate capabilities and topologies. This chapter is devoted to describing the *general purpose* architecture and configuration options of Coresight, as it pertains to (almost) any hardware. For a deep dive into the platform-specific Coresight architecture of the two systems used throughout this work, refer to Section 4.3 for the Zynq Ultrascale+ MPSoC and Chapter 5 for the Cavium ThunderX. The full instruction trace protocol is characterized in Section 4.5.1.

3.1 Overview

The Coresight system, shown schematically in Figure 3.1, is best interpreted as a set of components coupled by three distinct yet overlapping networks: The *tracing network*, *cross-triggering network*, and *debug network*. Generally, each component will have an Advanced Peripheral Bus (APB) interface to the debug network for accessing registers, a master/slave interface to the AMBA Trace Bus (ATB) for propagating trace data through the components, and an Embedded Cross Trigger (ECT) interface to interact with the cross-triggering-network for sending and receiving trigger events.

The main responsibility of the *tracing network* is to drive trace data from various trace source devices, like an ETM or an ITM among others, to a trace sink. Possible trace sinks include system memory, an off-chip logic analyzer, or directly from the PS to the PL.

The *cross-triggering* network presents an underlying communication interface for devices to communicate when necessary and the debug networks allow an external debugging tool to be hooked up for monitoring and execution steering. We want to gather data for PGO, which means the most value generated for us by the Coresight system originates from the PMU, ETM/Processor Trace Macrocell (PTM) and System Trace Macrocell (STM)/ITM. These components generate traces for us to reconstruct the control flow of an application, support instrumentation, and collect information on hardware events. As such, in this work, we may largely disregard the debug networks and only lightly touch on cross-triggering.

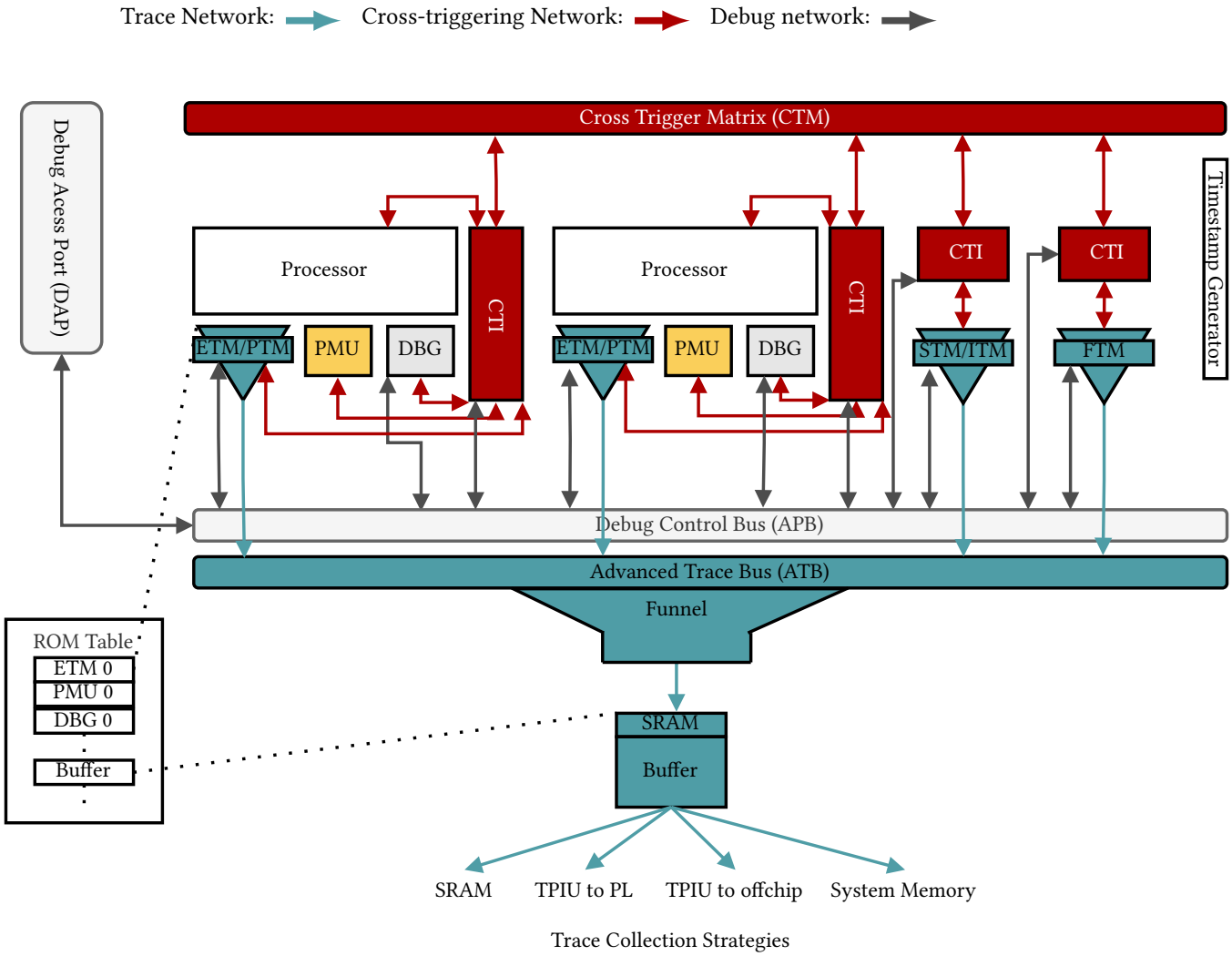


Figure 3.1: Typical Coresight architecture topology.

3.2 Embedded Trace Macrocell

The ETM acts as a trace source and forms the backbone of the Coresight infrastructure. It traces the execution of the processor it is coupled to. The primary function of the ETM is to produce a stream of trace data. The trace data includes the *instruction trace protocol* and the *data trace protocol*. However, the data-trace protocol is an optional feature not supported by most processors.

The ETM is considered the successor to the PTM which is only able to trace instructions and lacks many features supported by an ETM. Both the ThunderX and the Zynq Ultrascale+ MPSoC (US+) implement the ETM specification, so we focus on the instruction trace protocol of the ETM

The goal of the instruction trace protocol is for a trace analyzer to be able to recreate the entire execution flow of a program (we describe the entire protocol in Section 4.5.1). The ETM achieves this by driving data onto the trace stream whenever a branch instruction is executed on the processor. A branch instruction acts as a signpost for control flow, from which the execution of all other instructions can be inferred. Additionally, the ETM generates processing context information, for each instruction we have the following context:

— Virtual address of the executed branch instructions.

— A context identifier, identifying from which process or thread the instruction originates.

— A virtual context identifier, identifying from which guest Operating System (OS) an instruction originates.

— The exception level ∈ {EL0, EL1, EL2, EL3}

— Security State ∈ {Secure, NonSecure}.

For out-of-order processors, a stream of executed branch instructions no longer provide a clear picture of the control flow. The ETM solves this by only streaming information in program order. When it comes to speculative execution, however, an ETM lets us peak under the hood, sending out trace information not only for retired instructions but also for speculatively executed instructions. Whenever a branch instruction is observed in the trace stream, it is assumed to be speculatively executed until explicitly committed.

Though in practice rarely implemented, the ETM specification also supports tracing data streams, including the transfer address of load and store instructions and the data transfer value. The instruction and data trace streams are logically separated and a relative ordering can be rebuilt offline after capture. The relative order is established by synchronization information sent out over both streams. While no devices used in this thesis support data tracing, we believe this provides an interesting future work direction and we discuss this in Chapter 8.

Programming the trace unit is done by writing to a set of control and configuration registers. The three most important registers are the `TRCPRGGCTLR` to enable the trace unit and start the tracing session, the `TRCCONFIGR` register to configure the tracing options, and the `TRCVICTLR` register that controls instruction filtering (refer to the specification [26] for full register summary). The steps for the simplest start to a trace session:

— Unlock the software lock by writing `0xC5ACCE55` to `TRCSLAR`.

— Setup tracing options by writing to `TRCCONFIGR`. As an example, we can enable global timestamping, cycle counting, Virtual Context Identifier (VMID), and Context Identifier (CID) tracing by setting `TRCCONFIGR` to `0x8D0`, assuming all options are supported by the tracing hardware.

— Set `TRCVICTLR` to `0x201` to trace everything and set start/stop logic to start.

— Enable trace unit with `TRCCONFIGR.EN = 1`

One challenge of setting up a trace session for a specific need is due to the wide variety of different implementations for different systems as vendors may pick and choose which parts of the specification they decide to implement. Before any design choices are made for setting up a design that relies on trace one should consult the supported capabilities of an ETM by reading the 14 ID registers `TRCIDR0 ... to TRCIDR13`. We provide an overview of tracing options for the devices used in this thesis, shown in Table 3.1.

The ETM is interconnected with most other Coresight devices, it drives data onto the ATB, which usually propagates data to a funnel or an Embedded Trace Buffer (ETB) for burst absorption. From this point, the data is forwarded to a sink component determined by the desired collection strategy, see Figure 3.1.

We have only scratched the surface of the ETM's capabilities. The ETM enables many customization features to specify exactly what to trace, for example tracing only certain addresses, sending out trigger events of the cross-triggering-network and more.

3.3 Instrumentation Trace Macrocell and System Trace Macrocell

The ITM or the successor component, the STM, are further instances of trace source devices. They support "printf-style" debugging, tracing of OS and application events, and can emit diagnostic information [20, 21]. Both components emit similar trace data to the ETM in the form of packets. In contrast to the ETM the stimulus of the trace is not processor execution, but instead, software writes to internal ITM or STM registers. The STM does not work out of the box like the ETM, but requires code to be instrumented before it can be traced. Any value written into the STM or ITM stimulus registers are packed into

Table 3.1: Overview of tracing features and showing supported features of ThunderX(TX), and US+ and difference to PTM

Feature	US+	TX	PTM	Description
Virtual context identifier	✓	✓	✓	Add a virtual context identifier to trace
Context identifier	✓	✓	✓	Add a context identifier to trace
Cycle counting	✓	✗	✓	Add cycle counting information for fine-grained cycle timing to trace
Global Timestamping	✓	✓	✗	Add system-wide global timestamps to trace
Q Elements	✗	✗	✗	Compress the control flow data to reduce the trace bandwidth.
Return stack	✓	✗	✗	Adds implicit address for indirect branches.
Conditional instruction	✓	✓	✓	Add tracing of conditional non-branch instructions to the trace stream.
Branch broadcasting	✓	✗	✗	Add address information for direct branches.
Data trace	✗	✗	✗	Traces load and store instructions, separate data trace stream
Speculative execution tracing	✗	✗	✗	Add tracing of speculatively executed instructions, with commit information

trace packets and sent out over the ATB.

A typical use case for instrumentation would be to profile higher-level constructs other than instructions. For example, we can trace syscalls by inserting register write before a syscall in our application, or modify the OS to write into these registers. The same goes for a library function call. In the context of PGO being applied to Object Oriented (OO) programming languages, the STM could be used to trace object types by instrumenting constructors. Or profile the type of receiver at a virtual call. The trace data from the ITM or STM can then be correlated to the application code with timestamps and ETM trace data if necessary.

The STM provides features that the ITM does not support, that is the tracing of hardware events. The STM has 64 hardware event input ports that can be emitted onto the STM trace stream. In the case of the US+, the hardware event inputs of the STM are shared by Cross Trigger Interface (CTI) inputs and the remaining 60 hardware event inputs come from the PL. This allows an entire system to be traced, a running application on the processor and simultaneously tracing the state of a block design on the PL.

3.4 Trace Buffers

Coresight supports a variety of buffer types categorized as Trace Memory Controller (TMC)s with dedicated Static Random Access Memory (SRAM) [22, 20]. The buffers avoid losing trace data that is often bursty and reduce the pin requirements to drive trace data from a PS, enable reading trace data from software or Joint Test Action Group (JTAG) and further can provide an Advanced eXtensible Interface (AXI) interface to the system bus for writing directly to system memory. To support the features the TMC allows for three operational modes and comes in three hardware configurations integrated into the chip design. To drive trace data from a trace source to a trace sink, all TMCs in the path must be programmed to correctly forward data.

The operational modes are *circular buffer mode*, *hardware FIFO mode* and *software FIFO mode*. Hardware FIFO mode is the operational mode to forward data along a source-to-sink path. The TMC in this mode acts solely to buffer trace data and absorb bursty data. It also guarantees that no trace data is lost by the TMC itself by exerting backpressure on the source, implying trace data has a single point of overflow potential on the PS side. Both circular buffer and software FIFO mode are mainly used for either sampling-based profiling or debugging purposes and making sure Coresight has been properly configured. In circular buffer mode, the buffer acts as the final trace sink and stores all trace data in its dedicated SRAM. Even for a very short trace session the buffer will wrap and trace data will be lost. However, this allows for reading trace data directly from the associated SRAM after memory mapping. This provides a good starting point for checking ETM configurations or running sampling-based profiling. In software FIFO mode the data can be read from the APB debug interconnect. This is

also not practical for trace sessions, as the throughput requirements will not keep up with generated trace data, but a useful alternative to circular buffer mode for sampling.

A TMC comes in three hardware configurations, Embedded Trace FIFO (ETF), Embedded Trace Buffer (ETB), and Embedded Trace Router (ETR), each supporting a different set of operation modes. To avoid confusion, we clarify that ETB is both a hardware configuration of the TMC *and* the name of the predecessor specification to the TMC, sometimes seen in older devices. If the TMC is configured as an ETB or ETR, it only supports the circular buffer and software FIFO operation, while an ETF additionally provides hardware FIFO mode. However, only an ETR has an AXI slave interface for the system bus and subsequently system memory. Any TMC that is on the path from source to sink must be an ETF and support hardware FIFO mode. We add TMC support to the CSAL, since it previously only supported programming TMCs as circular buffers (Section 4.3.1).

3.5 Performance Monitoring Unit

Almost any processer nowadays has a PMU. In a nutshell, the PMU maintains individual counters for architectural or microarchitectural events, such as cache accesses or retiring instructions, and increments the counters for each event occurrence. When the counter overflows, the PMU triggers an interrupt (nPMUIRQ) which is exported to both the Cross-triggering-network and to external hardware. The counters can be either sampled from software by reading the event counter registers or the PMU may drive the events onto the PMU event bus to other Coresight devices.

This hardware is also what is typically used for profiling software like Perf [10] and most profile-collecting systems like AutoFDO [35] and Bolt [67]. Almost all sampling-based techniques use the PMU in some shape or form.

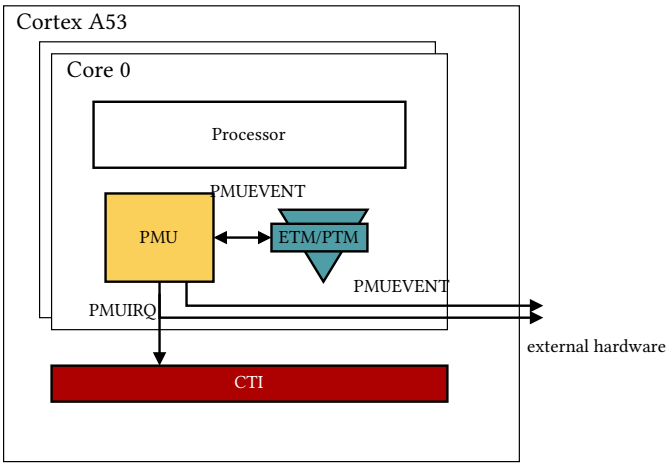


Figure 3.2: PMU event bus on the Cortex-A53

3.5.1 PMU Events

The Coresight PMU is tightly coupled to the rest of the Coresight infrastructure and allows for interaction between the devices that go beyond polling PMU counters. The PMU can export events to the internal debug hardware, typically both the CTI and the ETM, or to external hardware, including the PL on an SoC. The event bus connections of a Cortex-A53 are illustrated in Figure 3.2. In this example, the internal PMU events are only sent to the ETM. The overflow event PMUIRQ is internally only sent to a CTI.

Table 3.2 shows a handful of PMU events that give relevant performance information on a running application. These events can be used to monitor performance, giving an overview of how well an application is performing or how many

Table 3.2: Collection of PMU events of interest for performance profiling available for a Cortex-A53 processor [24].

Event	Description

`L2D Cache Refill`
`L1D Cache Refill`
`L1I Cache Refill`
`L1D TLB Refill`
`L1I TLB Refill`
	A refill event is triggered whenever a miss occurs in a cache or Translation Lookaside Buffer (TLB). This event can be used alongside trace to optimize for better cache performance

`L2D Cache Write-Back`
`L2I Cache Write-Back`
`L1D Cache Write-Back`
	Tracking write-back events can estimate the number of writes to the system memory are made throughout execution of a code region.

`L2D Cache Access`
`L1D Cache Access`
`L1I Cache Access`
	Whenever a cache is access this event triggers. This can be used to evaluate memory intensive code regions for example.

of the system resources it is using. It can also judge how many memory accesses are made throughout execution. It is also applicable for performance profiling, highlighting regions of code with poor cache utilization and subsequently used for PGO.

The ETM can forward the PMU events to the CTI or embed them into the trace data. Embedding events into the trace data gives context to each event in relation to a running binary of the processor. This allows a trace analyzer to more accurately answer questions such as "What is causing the hardware event to occur?" or "How many cycles have passed between two events?". One limitation of embedding events into the trace stream is that only up to four events can be monitored simultaneously during a tracing session.

The ETM interacts with the PMU event bus through its external inputs wires, programmable by two event control registers TRCEVENTCTRL0R and TRCEVENTCTRL1R . Both these registers need to be configured to start a trace session with embedded PMU events.

For optimization, it can be beneficial to understand the variance between the ground-truth event occurrence on the processer vs the observed embedded event in the trace stream — the more accurate a PMU event is the better cause and effect analysis can be made by a trace analyzer. Or in other words, in the case of a cache miss, the most useful feedback a trace analyzer can receive is which *exact* instruction causes this miss.

The variance of PMU event embeddings is implementation-defined and no guarantees are made by the Coresight specification. They do, however, provide a recommendation of when PMU event data should be inserted into the trace data, shown in Figure 3.3 [26].

If a PMU event occurs after a branch b_0, and before or during branch instruction b_1, the event should be observable in the trace stream between branch instructions b_0 and b_n, where n is the number of instructions a Processing Element (PE) can execute concurrently. This gives us a guaranteed window of mapping events to a binary that can be used for optimization. We also propose an experiment in Chapter 8 to empirically narrow down this window.

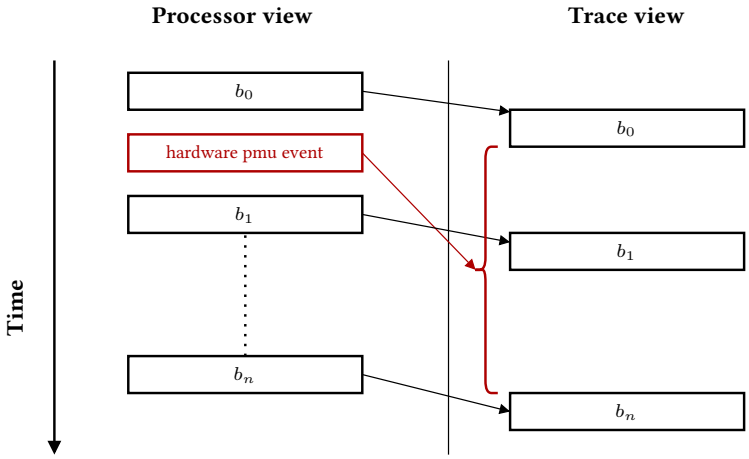


Figure 3.3: PMU event embedding into the trace stream timing guarantees.

3.6 Embedded Cross Trigger subsystem

Complex debugging and tracing features require orchestration among the devices. This functionality is provided by the Embedded Cross Trigger (ECT) [20, 24, 25]. The ECT is propagated through the system with multiple Cross Trigger Interface (CTI) that are distributed across the system that connects to a system-wide Cross Trigger Matrix (CTM). The CTM is an event broadcasting network making an event from one CTI observable from another. Typically, each PE will have an attached CTI that is used by the affine PMU, debug device, and trace source.

In abstract terms, a CTI will interact with a device that listens to events on the CTM. Events follow the ECT specification, shown on the left in Figure 3.4. We again use the ETM as an example of this, which has an EXTIN and EXTOUT wire to the CTI. This could allow starting a trace based on an ECT for example.

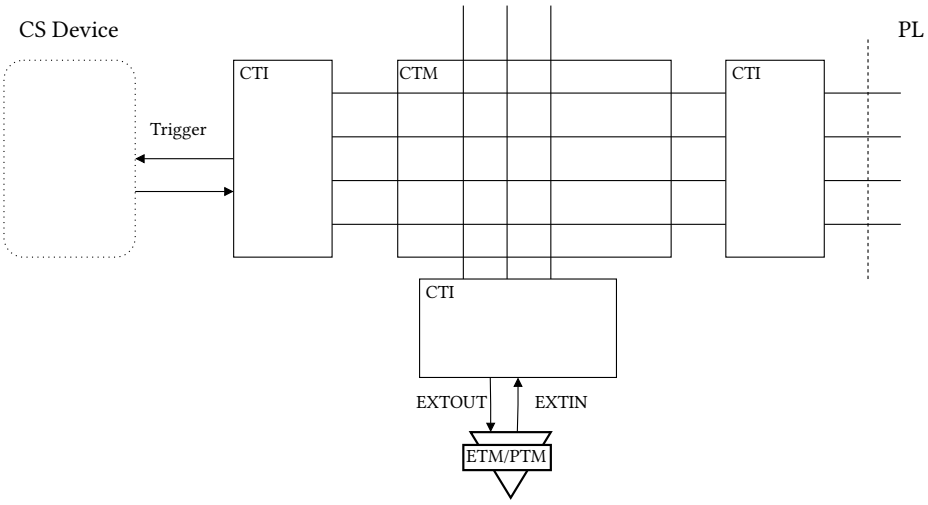


Figure 3.4: CTI and CTM network to enable the ECT.

On SoCs with PL, the CTI may also drive the triggers directly to the PL. To access the triggers from the PL, the block design configuration of the PS will provide an option to expose the triggers to the PL, as is the case with the US+ shown in Figure 4.2. Sometimes the CTM will also be able to observe and send events on a processor interconnect, specifically on the ThunderX the CTM is connected to the Coherent Processor Interconnect (CCPI).

The CTI topology is usually not described in detail for each device in a Technical Reference Manual (TRM). When planning on using the CTI for a project, the CSAL provides a CTI topology detection tool that will generate events on each CTI and observe on which port of which CTI the event is observed [2]. We recommend always starting such a project with the topology detection algorithm.

3.7 Trace Port Interface Unit

The Trace Port Interface Unit (TPIU) is the last device for any trace collection strategy that drive the trace data anywhere that is not in the PS [25, 20]. The only situations when a TPIU is not required is either when trace data is collected in an ETB or in system memory over the ETR AXI interface.

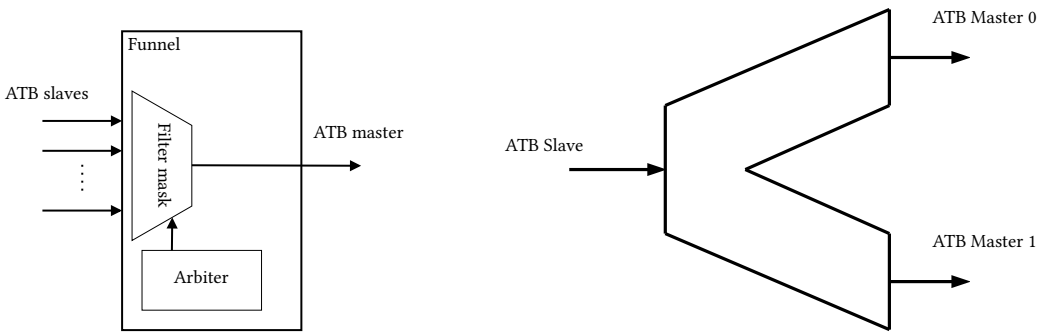
Three output signals are produced by a TPIU, TRC_DATA , TRC_CTL , and a TRC_CLK_OUT signal. TRC_DATA carries the trace data received by the TPIU over the ATB formatted into frames. The frame format allows the TPIU to multiplex the trace port for multiple trace sources. The data signal has configurable width of up to 32 bits. The CTL signal is an optional support signal for legacy trace analyzers to indicate non-valid trace data. The TRC_CLK_OUT is the output clock that a trace analyzer can orient itself around. A trace analyzer reading data from the TPIU should be phase aligned with the output clock and the data signal is valid at both edges of the clock. The output clock will be half the frequency of the input clock. The input clock can also be decoupled from the rest of the system so the PS cannot affect the TPIU frequency and vice versa. Extending this, the TPIU can be optionally fed by an external clock source. The external clock source can be provided by the trace analyzer to throttle the TPIU data rate to avoid losing data.

There are three modes of operation for the TPIU, *continuous*, *bypass*, and *normal*. In continuous mode, the TPIU will send data at every edge of the clock and send "empty" signals when the ATB has no data for the TPIU to send out. Bypass mode omits formatting entirely and will send out raw trace data. This is only possible if we have a single active trace source in the system, or data-to-source reconstruction is no longer possible and the trace data is corrupted. Normal mode is similar to continuous mode but requires the use of the CTL pin.

Internally, the TPIU will always have a pattern generator that can be used to check the pins and test clock speeds and synchronize the trace analyzer with the TPIU.

3.8 Trace Links

Trace links are connection components for Coresight components that share an ATB slave interface ATB, for our purpose, consisting of *funnels* and *replicators*, block designs in Figure 3.5. A funnel is required whenever multiple ATB master interfaces share an ATB slave interface (Figure 3.5a). To forward data the funnel filter mask must be set. An additional arbiter allows the funnel to assign priorities to slave interfaces, assigning a weight to each slave. This can reduce the likelihood more important trace data is lost if some trace sources are less valuable than others.



(a) Trace funnel block design.

(b) Trace replicator block design.

Figure 3.5: Trace Link components, block designs adapted from [20].

A replicator is the functional reverse of a funnel, connecting two slave interfaces of components that share a ATB master interface. In the Coresight topology, a replicator is typically seen towards the end of the ATB path, when the ATB is shared for multiple trace collection strategies. A replicator normally requires no programming.

In some cases, TMC can be classified as a trace link component with a single ATB master and slave interface, but only if it is a TMC in ETF hardware configuration mode. The TMC then acts solely as a buffer to absorb bursts in the trace stream.

3.9 ROM Table & Device Discovery

The Coresight Read only Memory (ROM) table holds a list of the physical addresses for every Coresight component that is available in the system. The only requirement before enabling a tracing or debug session is knowing the base physical address of the ROM table — all further information required can be read from identification registers and gathered with either trial and error or topology detection algorithms. The CSAL library provides tools for device discovery that dump all the information from the Coresight ROM table and identification registers [2]. It further includes both an ATB topology detection algorithm and a CTI detection algorithm that will show what components share a direct connection and both port numbers for each connection interface.

Chapter 4

Implementation

In this section, we describe our complete system implementation on the Zynq Ultrascale+ MPSoC (US+), from enabling the tracing subsystem on the PS (Section 4.3) and the handling of Coresight trace data on the FPGA (Section 4.4) up to the ETM instruction trace decoder (Section 4.5). The implementation can be broken down into two parts, namely the implementation on the Processing Subsystem (PS) and the implementation on the Porgrammable Logic (PL) or Field Programmable Gate Array (FPGA).

4.1 High-level Design Overview

We begin with a high-level overview of our system trace data and describe the lifecycle of the trace data, from when it is produced by a trace source on the PS up until it is decoded and the analysis result are passed back to the PS, as shown in Figure 4.1. Ultimately, the goal of the system is to process the data produced by Coresight hardware components and create profiling data that is consumable by a compiler to apply PGO. To achieve this, the data that is exported to the FPGA by Coresight needs to go through multiple decoding and analysis stages.

First and foremost, the Coresight subsystem must be programmed and all the Coresight components must be configured. Second, all trace data needs to be driven to the FPGA. All trace sources share the same interconnect, requiring the TPIU to multiplex the trace port, which is the interconnect to the FPGA. Once received on the FPGA, the trace data must be demultiplexed based on the trace source. Each ETM produces trace data that must be handled independently, meaning each trace source requires a distinct trace decoding and trace analyzing process. Any results of the analysis are then made accessible to PS via AXI-Direct Memory Access (DMA).

We present a more detailed walkthrough of the system-wide pipeline to be interpreted alongside Figure 3.1:

(1) Processer execution is observed by a trace source and forwarded to the PS ATB.
(2) Trace data is propogated through the ATB, through funnels, TMCs, and replicators until it reaches the TPIU.
(3) The TPIU uses the trace port to direct trace data from the PS to PL in 32 bit TPIU-packets.
(4) The data produced by the TPIU is logically grouped into *frames* (Section 4.4.1) consisting of 4 TPIU-packets. The TPIU produces synchronization metadata to mark the starts of frames. The raw TPIU data enters the frame generator that converts the raw data to physical frames of 128 bits.
(5) The frames contain a mix of trace data bytes and source identifier bytes. The frame decoder handles the conversion from frames into trace data byte streams and attaches the corresponding source IDs to each data byte. A maximum of 4 bytes per cycle can be sent out to a single trace decoder.
(6) Once the data and ID separation is complete, the demuxer filters data by source ID and the data is passed to its matching instruction trace decoder. Each source requires a decoding unit.
(7) All data received by the trace decoder is now stripped of any meta-data and only includes raw trace data in the form of a byte stream with a maximum bandwidth of 4 bytes per cycle. The trace data follows the ETMv4.0 [26] instruction trace protocol which is a packet-based protocol. The trace decoder is responsible for interpreting the byte stream as packets and reconstructing the processor behavior from these packets.

(8) How the data output from the trace data decoder is intentionally kept vague and is application specific. Therefore, we refer to this step as the opaque trace statistics aggregator. Depending on what type of optimizations are intended to be applied. An example would be to compute branch frequencies.
(9) Results of the whole trace data pipeline can be made accessible again to the PS through AXI DMA. In this work, the US+ has PL-specific Double Data Rate Synchronous 4 (DDR4).

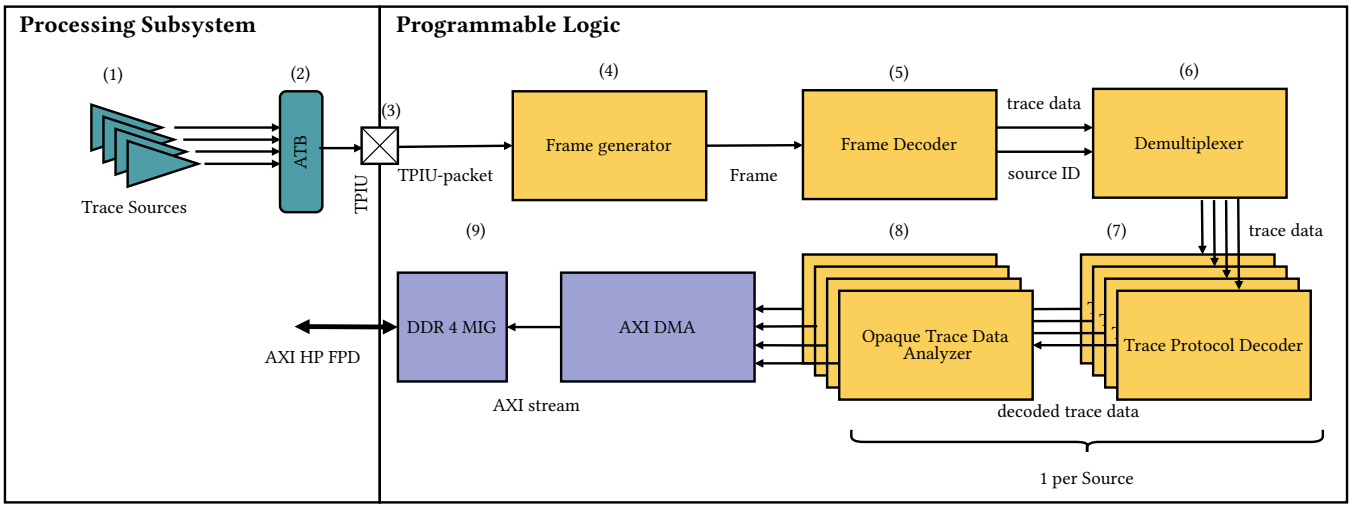


Figure 4.1: High-level design overview.

4.2 Implementation Environment

All designs for this project were implemented on the Zynq Ultrascale+ MPSoC (US+), with TRMs [18, 16, 17, 43]. The Coresight architecture of the US+ is covered in Section 4.2.1. The part identifier is specified in Table 4.1 alongside the Vivado/Vitis tool suite and Linux versions. The device speed grade is -2i, it has 1 GB of DDR44 SDRAM memory dedicated to the PL and 4 GB of DDR44 SDRAM main memory for PS both running at maximum of 1.2 GHz frequency. It has 4 Cortex-A53 processors running at a maximum frequency of 1.3GHz and 2 Cortex-R5 processors with a maximum frequency of 533MHz. On the PL side, the maximum output frequency supported by a Mixed-Mode Clock Manager (MMCM) is 775Mhz.

Table 4.1: Software and device specifications.

Software/Device	Version
Vivado & Vitis	v2020.1
Zynq Ultrascale+ MPSoC	xczu5ev-sfvc784-2-i
Board	Mercury XU5 PE1: ME-XU5-5EV-2I-D12E-R1.2
Reference Design (starting point & constraints)	ME-XU5-5EV-2I-D12E [8]
Linux	Linux buildroot 5.4.0-g751a2e13b

4.2.1 Zynq Ultrascale+ MPSoC Coresight Architecture

In this section, we go over the details of the Coresight topology of the US+. The US+ follows the typical Coresight topology from Chapter 3 quite closely and the complete topology is shown in Figure 4.2.

Each core on the US+ has an ETM, totaling 6 ETMs in the entire system, 2 for Cortex-R5 processors and 4 for the Cortex-A53 processors. This holds for other Coresight components as well, each core has a PMU, CTI and a debug component. In this work, we focus only on tracing the Cortex-A53 cores. Tracing of R-5 cores is analogous to tracing the A-53 cores and our work can be generalized to tracing these cores as well. Additionally, the US+ has one system-wide STM and Fabric Trigger Macrocell (FTM).

In total, there are two trace sinks, an ETR and a TPIU. Together these components cover all possible trace collection strategies, including forwarding trace data to the PL through the trace port and writing to system memory with the ETR AXI interface. It is also possible to do both simultaneously, as the trace data is replicated for the ETR and the TPIU.

The US+ has two interfaces to interact with PL, the Fabric Trigger Macrocell (FTM) and trace port over Extended Multiplexed I/O (EMIO). The trace data can only be forwarded over the trace port and TPIU, but ECT signals can be sent and received over either the General Purpose Input/Output (GPIO) pins or dedicated ECT trigger pins. The FTM is not a standard Coresight component, but an extension to convert PL signals to ECT signals. Our work does not require the use of the FTM and only the trace port is used to interface with FPGA.

The most important differences to the standard Coresight model are the intermediate funnels and buffers that are placed between the trace sources and the sink. In total, there are two funnels and two intermediate TMCs. `Funnel2` is the last funnel in the system and all trace data passes through this funnel. Before being replicated to two trace sinks, trace data is stored in an intermediate TMC `ETF2` . The ETMs of the Cortex-A53 also have an additional funnel and TMC shared by the 4 ETMs cores. This is likely because this section of the Coresight topology has the highest trace bandwidth requirements, as the Cortex-A53 cores run at a higher frequency and we can trace 4 cores instead of 2. All other sources feed directly into `funnel2` .

To start a tracing session on the US+ with a trace collection strategy that requires either the TPIU or the ETR, all the components must be programmed with the component registers. For tracing the Cortex-A53 cores, this means programming the ETMs, `Funnel1` and `Funnel2` , both ETFs `ETF1` and `ETF2` , the replicator, and either the TPIU, the ETR, or both. The ETFs must be configured to hardware FIFO mode, as this is the only mode in which an ETF can operate as a link device instead of a sink device.

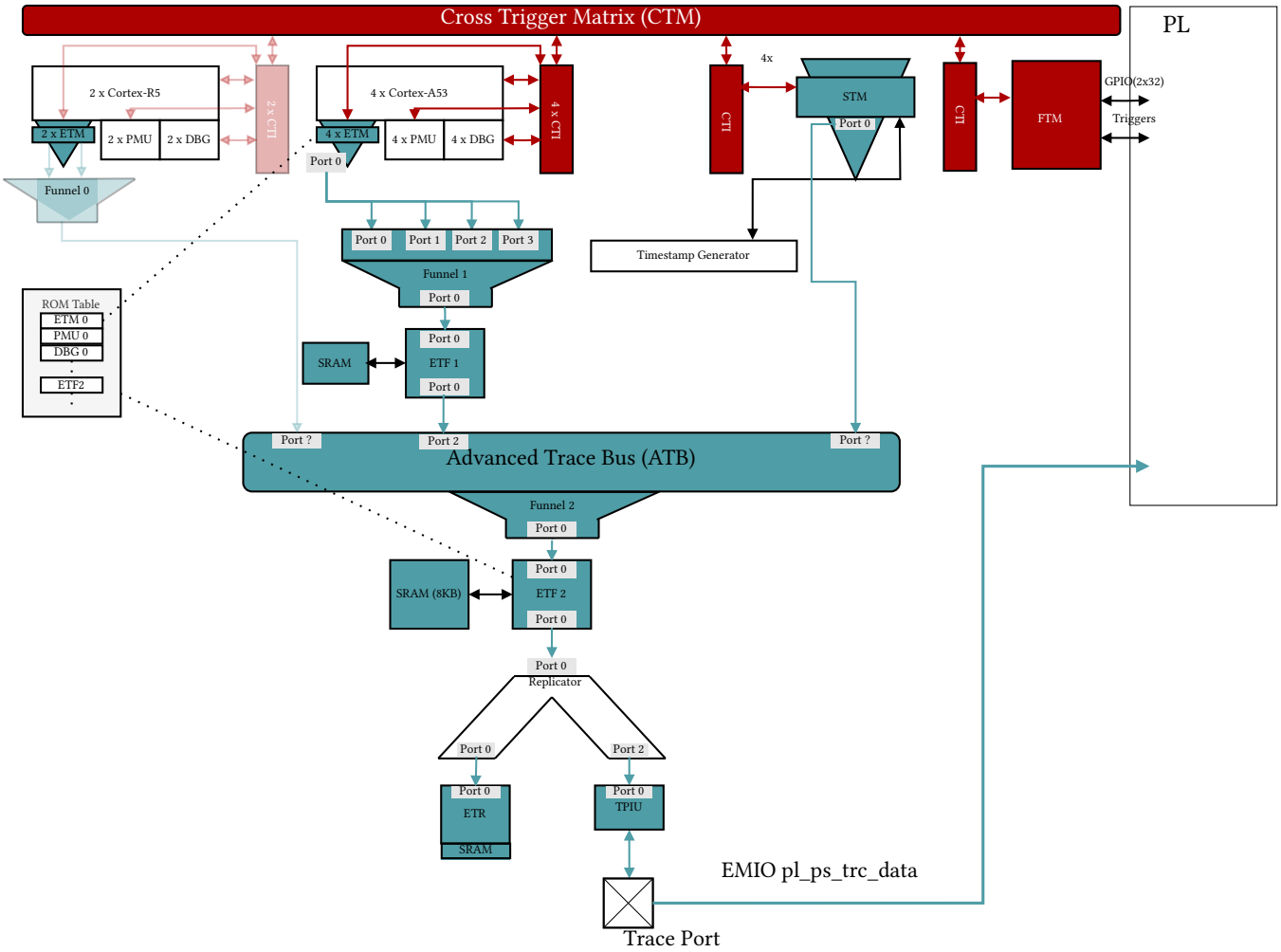


Figure 4.2: Updated Zynq Ultrascale+ MPSoC Coresight Block Diagram with added port numbers from [18], Figure 39-4.

4.3 Programming the Tracing Subsystem

In this section, we describe everything that is required to drive trace data observed by a trace source on a processor up until it is driven to the PL.

4.3.1 Coresight Access Library

The CSAL provides a set of tools for device discovery and topology detection and an interface for programming the Coresight devices exposing the device registers [2]. Everything in regards to enabling the trace unit and driving the trace data from inside the PS up until the trace port is done via the CSAL API. In our case, this includes programming the ETMs, PMUs, CTIs, funnels, TMCs (buffers), and the TPIU. The TPIU requires additional steps to configure the PS- PL interface and the TPIU runs in its own clock domain, see Section 4.3.2.

Before programming the devices, the system-specific topology must be registered into the CSAL. This means each physical address for every component and every connection among components over the ATB, including port numbers, must be registered. Device addresses are set by either consulting the TRM or scanning the Coresight ROM table. The CSAL supports a `csscan` tool to dump information on every Coresight component. Finding the connections between components should also be specified by the TRM. From our experience, this often proves error-prone, as was the case with the US+ TRM does not

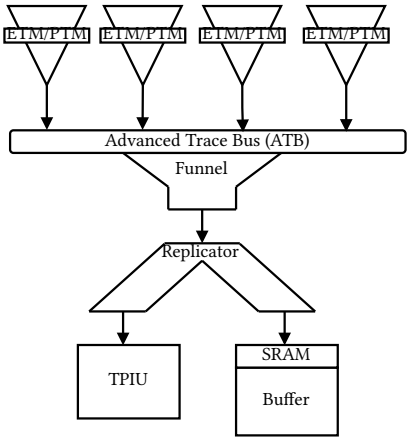


Figure 4.3: Basic Coresight topology.

specify port numbers[1], we provide a more detailed Coresight topology visualized in Figure 4.2 that includes port numbers. Topology discovery is also supported by a CSAL tool. However, from experience, this will not always work out of the gate and some details are uncovered through trial and error.

The functionality of the CSAL includes an interface to start a trace session, reading PMU counters, and collecting results from a buffer. However, the CSAL has quite limited functionality and we need to extend it for our purposes. The CSAL implements everything in software, and only short trace sessions are supported. Furthermore, the CSAL *solely* supports the topology (with any number of trace sources) shown in Figure 4.3. For the Zynq-7000 used in previous work [14, 74], no major revisions to the CSAL are required. The only change is adding support for TPIU programming. We reuse these changes in our implementation from the previous work [74], with slight adaptations and integrated the changes into the CSAL.

Still, this is not enough for our needs on the US+, topology shown in Figure 4.2, or any additional complexity in the topology. For this work, we modified the CSAL to support a topology of any complexity, enable registering multiple intermediate TMCs, like the `ETF1` from US+ Coresight architecture, programmed as Hardware FIFOs. We also provide a few modifications for tooling support on the ThunderX (Chapter 5).

Omitting some details, every device on the board should be registered either by explicitly calling the `cs_device_register` function with the physical address of the Coresight component, or providing the address of the Coresight ROM table. The device is then stored in a struct `cs_device_t` (Listing 4.2), which handles the memory mapping and reads the `idr` registers to identify the device class, i.e sink, link or source and its capabilities. The ATB topology must be explicitly registered with `cs_atb_register` , defining the topology of the trace connections (Listing 4.3).

```
struct cs_devices_t {
  cs_device_t etm[MAX_APUS];
  cs_device_t itm;
  cs_device_t etb;
  // extended:
  cs_device_t hw_fifo_tmcs[MAX_TMCS];
  cs_device_t tpiu;
};
```

Listing 4.1: Board Coresight topology struct [2]

```
struct cs_device_t {
  /* Memory map bookkeeping */
  cs_physaddr_t phys_addr;
  unsigned char volatile *local_addr;

  /* Device class and device affinity */
  unsigned int devclass;
  unsigned int devaff0;

  /* ATB topology graph connections */
  struct cs_device_t *ins[MAX_IN_PORTS];
  struct cs_device_t *outs[MAX_OUT_PORTS];
}
```

Listing 4.2: Coresight device struct [2]

[1] some port numbers are marked with ? since we did not need these ports.

Once the board setup is complete, the CSAL has an overview of all the devices and the ATB connections. The purpose of each device for a tracing session, meaning which device should act as the trace sink, and which device acts as a source, is defined by the fields in a global struct `cs_devices_t` (Listing 4.1) .

```
/*Setting up a board for Coresight.*/
cs_device_t cs_device_register(cs_physaddr_t addr);
int cs_atb_register( cs_device_t from, unsigned int from_port, cs_device_t to, unsigned int to_port);

/*Starting a tracing session.*/
int cs_trace_enable(cs_device_t dev);
int cs_sink_enable(cs_device_t dev);
```

Listing 4.3: CSAL tracing and registration API, adapted from [2].

Beginning a trace session on a board requires the configuration of the source device, in our case an ETM, the sink device, any TMC type for example, and the programming of all the link devices on the path from the trace to the source. This includes the programming of both buffers and replicators.

Programming replicators and filters is straightforward and typically only requires writing a mask value to a register that only forwards data from each port based on the given mask.

As mentioned previously, the CSAL supports enabling tracing sessions for simple topologies, but complex topologies are not supported. More specifically, any topology that has a device on the path from source to sink that is neither a funnel nor a replicator device is currently unsupported. This includes TMCs, of which we have two on the path from an ETM to a TPIU, that is `ETF1` and `ETF` 2, see Figure 4.2. We extend the CSAL to add the enabling and disabling of TMCs with the function definitions shown in Listing 4.4.

The enabling of a TMC involves the following steps:

- Wait until the `TMCReady` bit is deasserted in the Status register, indicating any previous trace sessions have completed.
- The TMC has a `MODE` register, which can be set to `0x2` for hardware FIFO mode if the TMC has the ETF hardware configuration mode.
- The formatter and flush status register (`FFSR`) must be set to enable formatting. This is required if the TMC has multiple input trace sources, otherwise, the data cannot be reassoicated with the trace source. The data is formatted into 16-byte frames, the same as produced by the TPIU shown in Figure 4.6.
- The `BUFWM` register is set to `0x0` . This register marks the number of 32-bit words that must remain vacant in the TMC before it asserts being full. Being full is what causes backpressure on a trace source and we wish to minimize the backpressure generated by the TMC.
- The trace session can begin by setting the enable bit in the `CTL` register.

The disabling of TMC is almost analogous. However, we wish to avoid losing any trace data that is still being held in the TMC. To cleanly shut the tracing session down, we must set the device to “stop on flush” in the `FFCR` and manually trigger a flush event. We must wait until the flush event is complete before we disable the device. The TMCs should always be disabled from closest to the source to closest to sink, so no data remains stuck in the TMCs.

```
/*Programs a TMC as a hardware FIFO */

/* dev must be a TMC in the ETF hardware configuration. */
int cs_tmc_hw_fifo_enable(cs_device_t dev);
int cs_tmc_hw_fifo_disable(cs_device_t dev);
```

Listing 4.4: Extension of CSAL to support TMC as HW FIFO

To achieve a successful run of a tracing session, the only thing left to do is modify the function `cs_source_enable` such that it continues enabling funnels and replicators even if it hits a TMC. This is a small change with a few LOC and consists only of continuing the recursive call through the ATB.

4.3.2 Programming the Trace Port Interface Unit

The TPIU is a unique component as it exposes an interface to drive trace data off the PS, including off-chip or to the PL. On the US+ the TPIU requires an external clock, in addition to programming the component with registers. In our internal CSAL we already have an extension to support TPIU programming provided by Schmid, which we were able to reuse. However, we encountered some challenges when it came to setup the external clock.

4.3.2.1 Demystifing the Trace Port Interface Unit Clock Speed

The TPIU packetizes data in the form of frames (Section 4.4.1) and may output this data over EMIO to the PL. It runs in separate clock domain so that it is not coupled to an on-chip clock. This allows a trace analyzer to configure the TPIU clock based on its supported throughput, see Figure 4.4. Setting up the TPIU clock is surprisingly shrouded in uncertainty, as there is conflicting information between the US+ TRMs and Vivado US+ PS IP (Table 4.2). Note that TPIU is stated to transmit at a double data rate, meaning data is produced on each rising and falling edge, implying a maximum data rate of 250Mhz, and further also suggesting that `ps_pl_trc_clk` with a maximum clock range of 250Mhz sets the *data rate* and not the *clock rate*. This is in fact the case, even though the implemented design in Vivado will throw a pulse width timing error (Table 4.3). A pulse width timing error typically occurs when clock frequency passed to an underlying hardware primitive is not encapsulated by its valid frequency range [15]. In our case, this can be ignored, and most likely stems from a mistranslation of constraints between US+ Data Sheet (DS) and the Vivado IP.

To confirm this we ran a simple micro-benchmark to read the data directly from the `ps_pl_trc_data` pin at varying TPIU and reader clock speeds. Test pattern generators are supported by the TPIU and a possible test the pattern can generate a stream of walking zeros, giving a point of reference for the expected data. When the reader is too fast for the TPIU we should expect to see multiple identical values in succession. Or, we miss values if the the reader is too slow respectively. The experiment confirms that setting both readers and TPIU clocks to 250Mhz is the maximal supported throughput and works correctly. Setting the TPIU clock speed any less will result in reading duplicate values. We safely disregard the failing timing constraints.

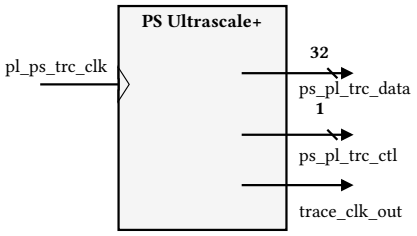


Figure 4.4: TPIU EMIO Pins

Clock	Max	Source
PL_PS_TRACE	125 MHz	US+ TRM [18]
EMIOTRACE	125 MHz	US+ DS [16]
pl_ps_trc	250 MHz	Vivado US+ IP

Table 4.2: Reported clock ranges.

Check Type	Required	Slack
Min Period	8ns	-4ns
Low Pulse Width	4ns	-2ns
Hight Pulse Width	4ns	-2ns

Table 4.3: Pulsewidth error at 250Mhz

4.3.2.2 Applying PS Changes to the Zynq US+

To run our design on the US+ we must program the FPGA with a bitstream and program the PS if any configuration changes are made on the PS block design in Vivado. The former is seamless with both the Vivado or Vitis toolchain, but we faced some challenges with the latter. Programming the PS broke the Universal Asynchronous Receiver-Transmitter (UART) terminal that we relied on. For our design, changes to the PS are necessary to program the TPIU. Failing to apply PS modifications, like configuring the trace port clock, results in an unpowered TPIU and the system crashes whenever an access is made to unpowered TPIU registers. In this section, we outline the necessary steps to apply modifications made to the PS while maintaining an accessible UART terminal to a running Linux OS.

Programming the device from the Vivado hardware manager only writes the bitstream to the PL. Making any changes to the PS is not realized by this step. For applying changes to the PS, we direct our attention to creating a platform and application project with Vitis by exporting the hardware from Vivado with an Xilinx Support Archive (XSA) file. This also generates

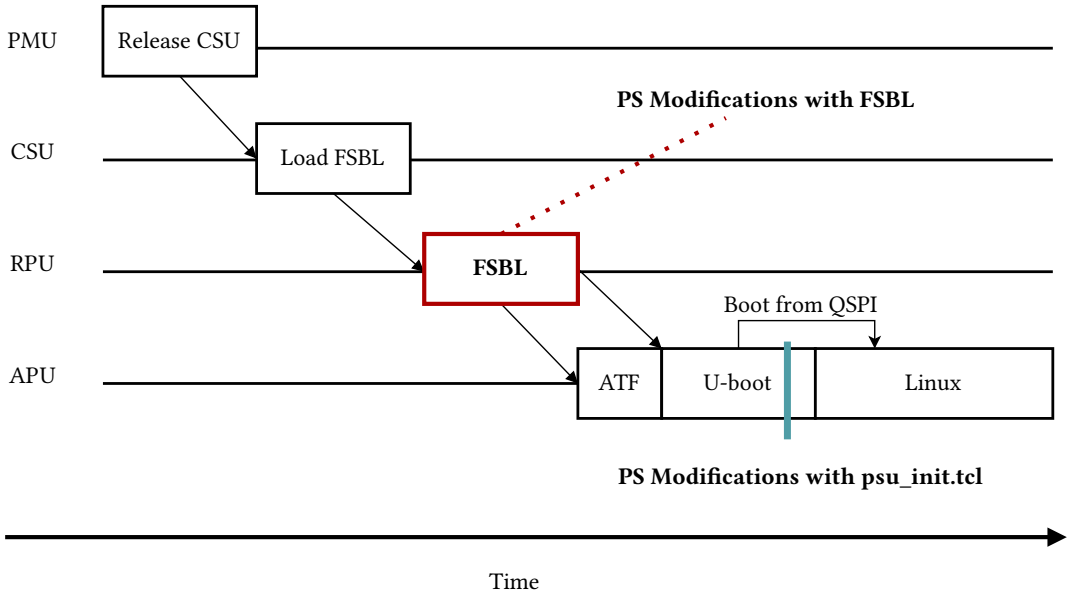


Figure 4.5: Adapted bootflow from [17].

a set of boot components, including a First Stage Boot Loader (FSBL) and `psu_init.tcl` script, which are the responsible components for PS configuration updates. The FSBL is additionally required to load further higher-level boot components like the Arm Trusted Firmware (ATF), U-boot and ultimately the OS, as illustrated by Figure 4.5. The `psu_init.tcl` script directly affects the PS configuration and is invoked by the FSBL (as C code files instead of tcl: `psu_init.c/h`) [5]. Vitis provides the option to use either the FSBL or the The Tool Command Language (tcl) script for PS initialization.

A typical order of operations when booting the with new PS configurations executes the following steps: Resets the entire system, including Application Processing Unit (APU) and PL, programs the PL with a bitstream, initializes the PS with either the FSBL or Tcl script, holds the processors in reset to download the desired executables (for example a standalone executable or a boot image), and finally clears the reset to allow the processor to continue. In our case, we have a bootable Linux image on Quad Serial Peripheral Interface (QSPI) flash from which we wish to boot.

The Vitis toolchain raises some issues for booting into Linux, with both the FSBL and the Tcl script the UART terminal is unresponsive after initialization. It is unclear whether we have properly entered even U-boot at this point or this is a UART issue. When using FSBL, UART terminal prints "entering FSBL", but is unresponsive after. When booting with the Tcl script, no output is shown at all. However, after various attempts, we found a workaround solution by applying the following commands:

(1) Connect to the board with a UART terminal
(2) Connect JTAG and target the PSU.
(3) Use Power on Reset (PoR) to restart the sytem.
(4) Hold the processor in reset after entering U-boot.
(5) Run `rst -system` over JTAG.
(6) Load the Tcl script with `source <path-to-tcl-script>/psu_init.tcl` generated when creating an application or platform project with Vitis.
(7) Run `psu_init` on the JTAG terminal.
(8) Optional check to see if the changes have been applied and in our case TPIU registers are readable with `rrd coresight_soc_tpiu` . If successful, this will display register values, for example, `supported_port_sizes: FFFFFFFF` , if not, will display `supported_port_sizes : N/A` .
(9) Run `con` on the JTAG terminal to allow the processor to resume execution and booting into the Linux kernel.

Once these steps are complete, all PS changes take effect. If no further changes are required, it is sufficient to directly program the FPGA with the bitstream.

4.4 Handling Frames

In this section, we describe all the necessary steps between reading data from the PS through EMIO that is sent out by the TPIU, up until generating the instruction trace stream that can be interpreted by an instruction trace decoder. The TPIU sends data in 32-bit chunks that is read from the EMIO and logically form 128-bit *frames* (Section 4.4.1). This frame data must be converted into multiple individual trace streams. To this end, we require a frame generation stage (Section 4.4.2), a frame decoding stage (Section 4.4.3) and finally a demultiplexing stage (Section 4.4.4), producing multiple raw trace stream that are each driven to their own instruction trace decoder.

4.4.1 Frame Format

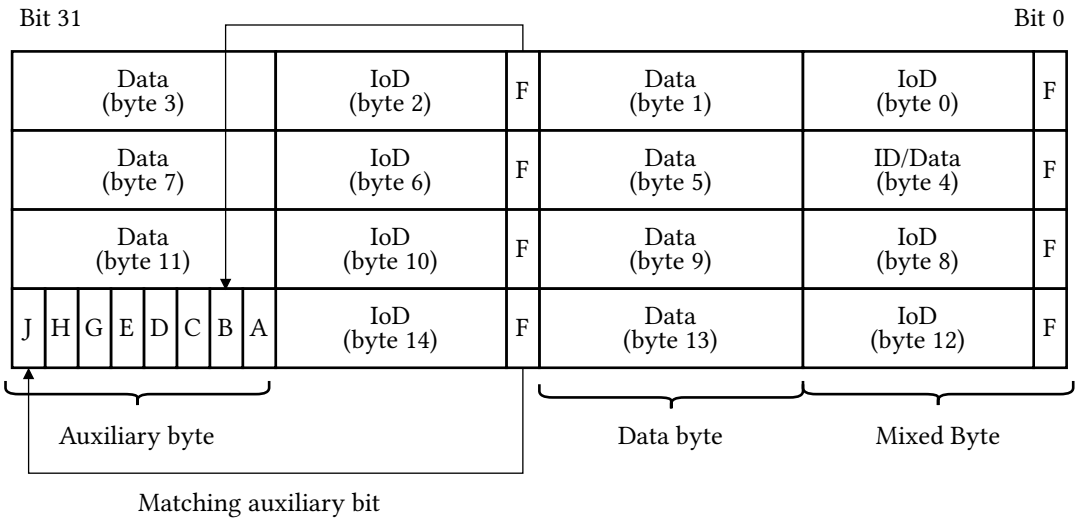


Figure 4.6: Frame format containing seven bytes of trace data, seven bytes of either an ID or Data and one auxiliary byte. The auxiliary bit either completes the data for ID/Data byte, or denotes at which byte the new ID takes effect [27].

The frame format is the logical representation of the data being transmitted by the TPIU. It multiplexes the outgoing trace port for the trace sources and makes it possible for a trace analyzer to reassociate trace data with the appropriate trace source. The frame, visualized in Figure 4.6, is made up of 128 bits and contains both source IDs and trace data. More specifically, it contains eight bytes of *mixed ID or Data bytes (IoD)*, which may be either a new ID or data, and seven *always data bytes* in alternation. The last remaining byte is an *auxiliary byte*. Source IDs are encoded in seven bits, and the Least Significant Bit (LSB) of an ID or Data (IoD) byte is used to denote whether the byte is a data byte or an ID byte. If an IoD byte contains data, we are missing the LSB. To recover the entire set of eight bits, the auxiliary byte completes the data held by an IoD byte. If instead, the IoD byte carries a new ID, the matching bit in the auxiliary byte marks at which point the new ID takes place. The new ID can take place from either the subsequent data byte or mixed IoD byte after that. The auxiliary bits are matched to their mixed bytes based on the position in the frame, meaning the first auxiliary bit `A` belongs to the first mixed IoD byte, the second auxiliary bit `B` belongs to the second mixed byte, and so on.

4.4.2 Frame Generation

The TPIU does not directly output frames, but instead, a stream of configurable width TPIU packets every clock cycle. We set the width to the highest possible width of 32 bits for maximum throughput. The frame generator is the first module in the pipeline and takes the input directly from `ps_pl_trc_data` (Figure 4.4) EMIO pin and produces frames, yielding at most one frame every four cycles.

Depending on which mode the TPIU is running in, the frame generation process may differ slightly. In our case, we use the TPIU in continuous mode(Section 4.3.2). Synchronization packets (of the form `0x7FFFFFFF`) are sent out periodically

indicating the start of a frame and half-synchronization packets (of the form `0x7FFF7FFF`) are sent out when the TPIU has not received any data through the ATB from the trace sources at the current cycle.

The Finite State Machine (FSM) is shown in Figure 4.7. We begin frame generation by scanning for a synchronization packet. Once synchronized, we begin filtering for valid data, i.e. packets that are neither synchronization nor half-synchronization packets. Whenever a valid TPIU packet is received, the FSM increments the state and records the valid data. Once four valid data packets are received, a full frame is created and can be sent out alongside a valid signal. When a half-synchronization packet is received we must discard it and continue with the current state.

Exactly when the fourth valid data packet has been received, a valid signal is sent alongside accumulated 128 bits of frame data. In any other scenario, the valid signal is de-asserted. The frame generator gives its consumer a window of a single cycle to read the data and no ready signal is used. This presents no issue as long as the rest of the pipeline can handle a throughput of at least 32 bits/cycle.

Receiving a synchronization packet when having only partially processed a frame means something must be misconfigured and usually suggests the design and TPIU configuration should re-examined. From experience, this almost always comes from a mismatch in TPIU and frame generation clock frequencies. For now, we attempt to resynchronize, but we note that the Error State in Figure 4.7 has never been observed in practice.

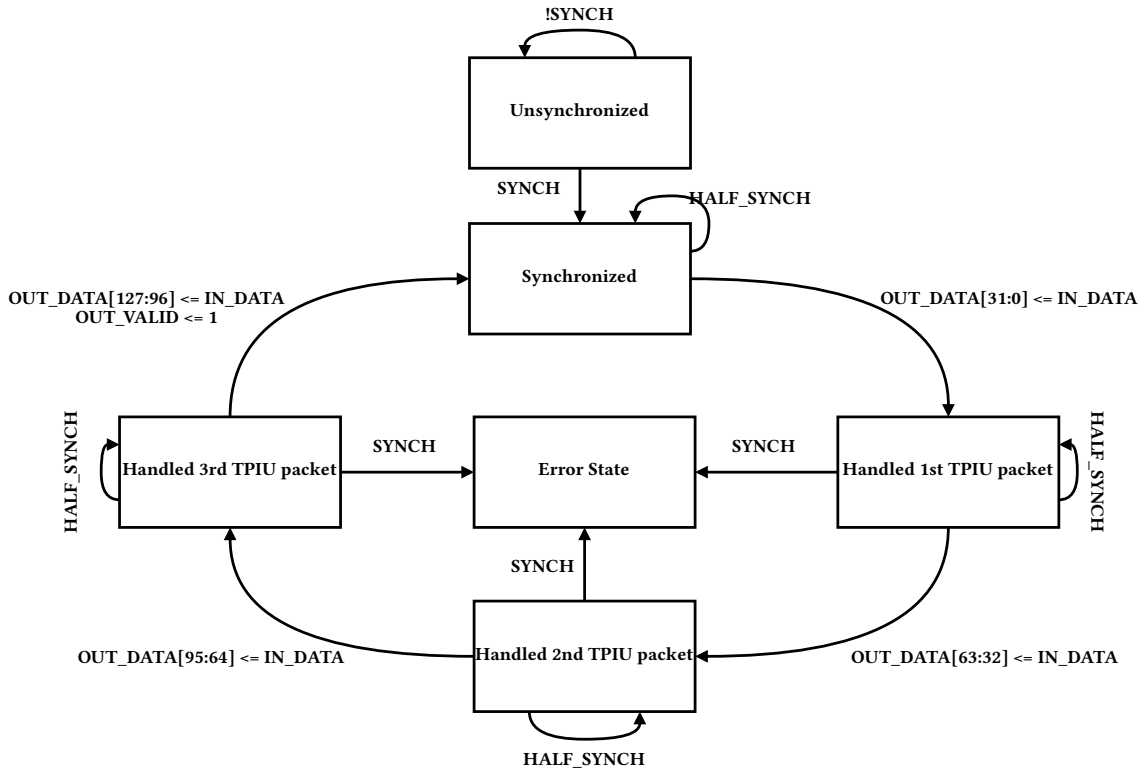


Figure 4.7: Frame generation FSM shows how to handle the TPIU packets. `OUT_VALID` is only set to one when receiving valid data at stage "Handled 3rd TPIU packet", and is cleared at any other state.

4.4.3 Frame Decoding

Once frames are reconstructed, the frame decoder extracts raw trace data and the source IDs for each trace data byte. We describe the frame decoding implementation in this section.

Registering a frame arrival with an accompanying valid signal from the frame generator kickstarts the frame decoding process that separates the trace data from the trace source ID encodings. The complete frame is held as a signal while being processed row-by-row (We refer to the chunks of 32 bits in a frame as a "row", but this is solely logical and not physical). Processing an entire frame takes a total of four cycles, complying with the throughput requirements of the frame generator.

The frame format interleaves ID and trace data, where the ID sets the current trace source ID of all subsequent data until a new ID is seen. In addition, the auxiliary byte is used to either complete the data from a mixed IoD byte, or denote when the new ID takes effect, so the auxiliary byte must be reused across decoding of all frame rows. The decoding process (Figure 4.8) can be thought of as a function $Decode\_Frame$: *row* × *auxByte* × *lastID* → *outData* × *outID* × *valid* × *newLastID*.

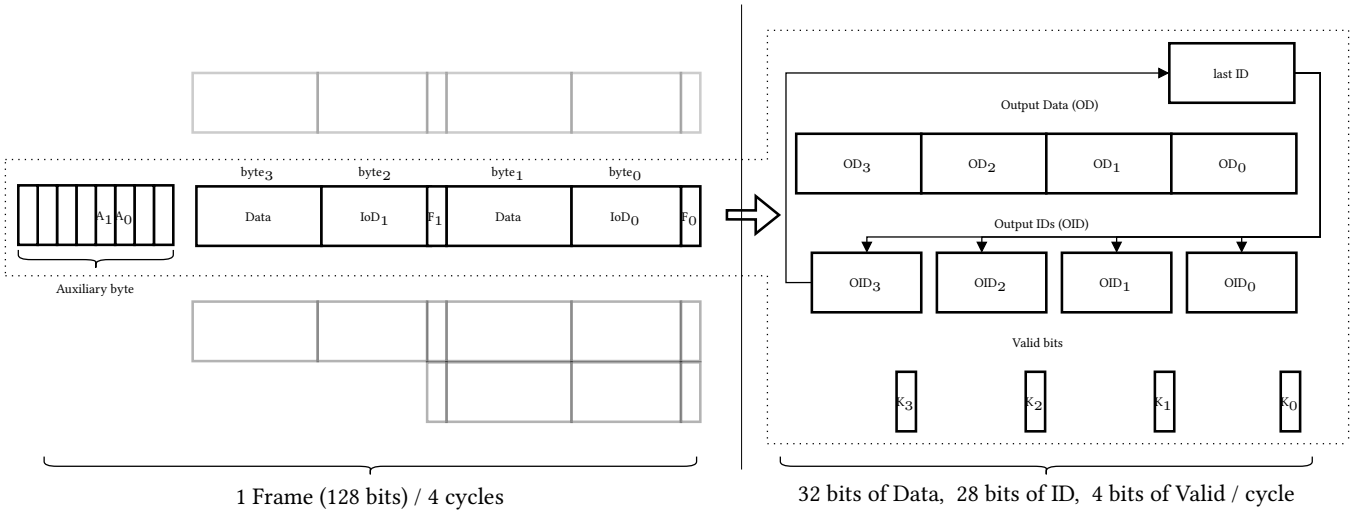


Figure 4.8: Row-wise frame decoding.

For each row we produce four keep signals $K_0 \dots K_3$ informing the consumer which of the four Output Data (OD) signals $OD_0 \dots OD_3$ contain valid trace data. OD is only valid if the corresponding byte is a data byte or a mixed byte and the LSB is zero. The OD is reconstructed with the auxiliary bit if necessary. Formally:

$$K_i = \begin{cases} 1 & \textit{if} \quad (\text{byte}_i \in \text{Data}) \vee \\ & \quad\ (\text{byte}_i \in \text{IoD} \wedge \text{byte}_i[0] = 0) \\ 0 & \textit{otherwise} \end{cases} \qquad OD_i = \begin{cases} \text{byte}_i[7:1] \oplus \text{aux}(\text{byte}_i) & \textit{if} \quad \text{byte}_i \in \text{IoD} \\ \text{byte}_i & \textit{otherwise when} \quad \text{byte}_i \in \text{Data} \end{cases}$$

where the $\oplus$ operation is a bit concatenation operation and $\text{aux}(\text{byte}_i)$ fetches the appropriate auxiliary bit.

Each OD needs an associated trace source ID to pass the trace data to its matching trace decoder, requiring an additional set of four Output IDs (OID) signals $OID_0 \dots OID_3$. The OIDs are computed by combinational logic of lastID state, and all previous mixed IoD bytes in the same row and their respective auxiliary bits. Formally:

$$\mathrm{OID}_0 = \quad \text{last ID}$$

$$\mathrm{OID}_2 = \begin{cases} \mathrm{IoD}_0 & \textit{if} \quad \mathrm{F}_0 = 1 \\ \text{last ID} & \textit{otherwise} \end{cases}$$

$$\mathrm{OID}_1 = \begin{cases} \mathrm{IoD}_0 & \textit{if} \quad \mathrm{F}_0 = 1 \vee (\mathrm{F}_0 = 1 \wedge \mathrm{A}_0 = 0) \\ \text{last ID} & \textit{otherwise} \end{cases}$$

$$\text{lastID} = \mathrm{OID}_3 = \begin{cases} \mathrm{IoD}_1 & \textit{if} \quad \mathrm{F}_1 = 1 \vee (\mathrm{F}_1 = 1 \wedge \mathrm{A}_1 = 0) \\ \mathrm{IoD}_0 & \textit{otherwise when} \quad \mathrm{F}_0 = 1 \\ \text{last ID} & \textit{otherwise} \end{cases}$$

To gain some intuition, refer to figure Figure 4.8. If F_0, the LSB of the byte_0 and first IoD byte, is asserted, $\mathrm{byte}_0[7{:}1]$ sets a new source ID. If the matching auxiliary byte A_0 is 0, the new ID takes effect from the byte byte_1. This lets us set OID_1 to ID_0. If, however, A_0 is asserted, the new ID will only take effect at byte_2, meaning OID_1 is set to lastID and OID_2 is set to IoD_0.

There is one exception to this rule: if the last byte before the auxiliary byte (byte 14) contains a new ID, the matching auxiliary bit must be zero and the new ID takes effect immediately at the beginning of the next frame. The frame format also guarantees that whenever an ID switch is made at least one byte of trace data exists before a future ID switch is made.

Finally, LastID is set to the same value as OID_3 to preserve the ID state across both subsequent rows and frames.

4.4.4 Demultiplexing the Trace Stream

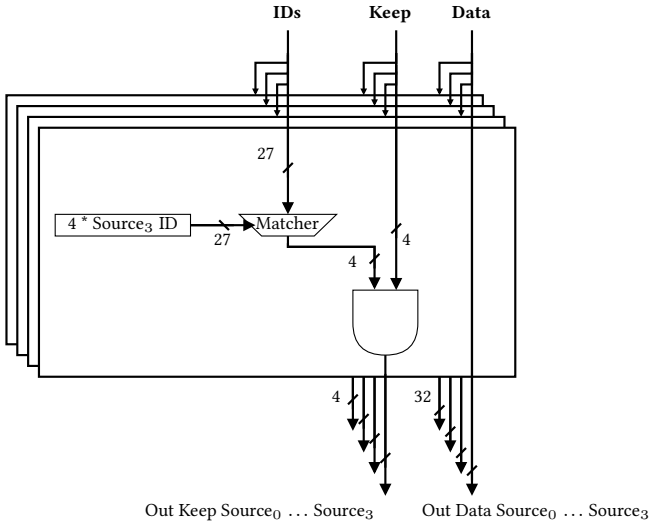


Figure 4.9: Demuxing decompressed frames from four trace sources.

After the frame decoder decodes the frames, the demultiplexer receives a stream of bytes, a source ID for each byte and a keep signal for each byte. Based on these values, each byte of raw trace data must be sent to the trace decoder that is responsible for handling said trace data byte.

Any trace source, like an ETM, ITM, or STM, requires its own decoder and will have an associated ID. For an ETM, the ID is set by the TRCTRACEIDR.TRACEID register, which must be encoded into the demultiplexer a priori.

Demultiplexing, as shown in Figure 4.9 is a straightforward process, as the frame decoding does most of the heavy lifting. The data bytes bypass the demultiplexing process and the demultiplexer sets new keep bits to mark each raw trace data byte as valid for each decoder. To filter only the valid source ID for each decoder, the demultiplexer has a hardcoded value for each trace source ID, concatenated four times, in our case forming a total of 27 bits. The results of the matcher are then ANDed together with the valid data bits received from the frame decoder.

4.5 Instruction Trace Decoder

The trace decoder is by far the most complex unit in the system-wide pipeline. A trace decoder must take in a stream of bytes produced by a single trace source, in our case an ETM, interpret this byte stream as a packet stream, and extract the semantics of the packet stream into meaningful information. To be clear, by semantics, we mean explicit control flow information of an executing binary that is being traced. In simple terms, a trace decoder should be able to say: "the CPU has executed a branch instruction at virtual address a_0, has taken the branch, and jumped to virtual address a_1." After the execution of a binary has been completed, the entire execution trace, i.e every jump in the program counter, should be traced and decoded.

We start this section by giving a detailed overview of the ETMv4.0 trace protocol specification (Section 4.5.1). We elaborate on the design challenges in decoding this protocol given our system-level throughput requirements and motivate why our design was chosen (Section 4.5.2). We conclude this section with a discussion to summarize the key learnings and potential design mistakes that were made along the way (Section 4.5.4).

4.5.1 Emedded Trace Macrocellv4.0 Trace Stream Protocol

Our work focuses on the ETMv4.0 specification [26], which is the specification implemented on both the US+ and the ThunderX. The ETM protocol is considered the successor to the PTM protocol [23] and supports a wider range of additional features, and was the main trace source for previous work done by Schmid [74]. We emphasize at this point that the PTM and ETM protocols are *substantially* different and there is no direct extension opportunity from a PTM decoder to an ETM decoder, but instead must be entirely reimplemented. To avoid confusion, there are three groups of Coresight program trace specifications, the PTM, the ETMv1.0 to ETMv3.5, and the ETMv4.0 to ETMv4.6 specification. Everything in this section may only hold for the latter.

The main purpose of a tracing unit is to provide a compressed stream of trace protocol packets, allowing a trace analyzer to reconstruct the flow of the program. To this end, every single branch instruction encountered during execution is recognized by a trace source and the information is encoded into a packet. ARM refers to instructions that alter the control flow of the execution as *P0* elements. This includes all branch instructions and, additionally, the Instruction Synchronization Barrier (ISB) instruction, which flushes the entire instruction pipeline of the processor. From now on, we will use the terms P0 instructions and branch instructions interchangeably. The tracing of P0 instruction is contained in so-called `Atom` packets, where an `Atom` packet holds information off multiple P0 instructions. In the name of precise terminology, we refer to a data point produced by exactly one P0 instruction as an `Atom` element, that is subsequently sent out as part of an `Atom` packet. An `Atom` packet can contain 1-24 `Atom` elements.

Alongside the generation of `Atom` elements for P0 instructions, the trace source will send out explicit *address packets*. The address packets embedded into the trace stream modify address values in the *trace state*, hence the protocol is a stateful protocol and some packets can only be decoded in the context of the trace state. This holds for the `Atom` packets as well. Alongside three address registers, the trace state, among other things, includes information regarding the processor execution environment and timestamp information (see Section 4.5.1.1 for more details).

The combination of address packets and `Atom` packets, of which there are many types, form the backbone of the protocol and are responsible for tracing the flow of execution. There are many additional packets (Table 4.4), but we provide some intuition first. Observe a simple step-by-step tracing example shown in Figure 4.10.Tracing begins at instruction address `0x0000` in line 0. This triggers the ETM to send a `Trace On` packet to inform a trace analyzer that a trace session is beginning. The initial values of the address registers $\mathcal{A}_0$ and $\mathcal{A}_1$ are undefined. A `Long Address` packet sets the first address register to the current instruction address of the first P0 element. Note that the first instruction is a direct branch instruction, a P0 element, so an `Atom` packet is generated. Each `Atom` element in an `Atom` packet is marked as either executed (E) or not executed (N). We believe the wording here is somewhat misleading, and we will refer to `Atom E` elements as *taken* and `Atom N` elements as *not taken* elements.

Line 1 has an unconditional branch, therefore the branch is taken and produces an `Atom E` element. The following two instructions, lines 1 and 2, are *not* P0 elements, no packets are needed. On line 3 however, we hit another P0 element, namely a conditional direct branch. In our scenario, the equal flag is not set, the branch is not executed, and an `Atom N` element is generated. The following P0 element is more interesting; an indirect branch. An indirect branch must not only produce

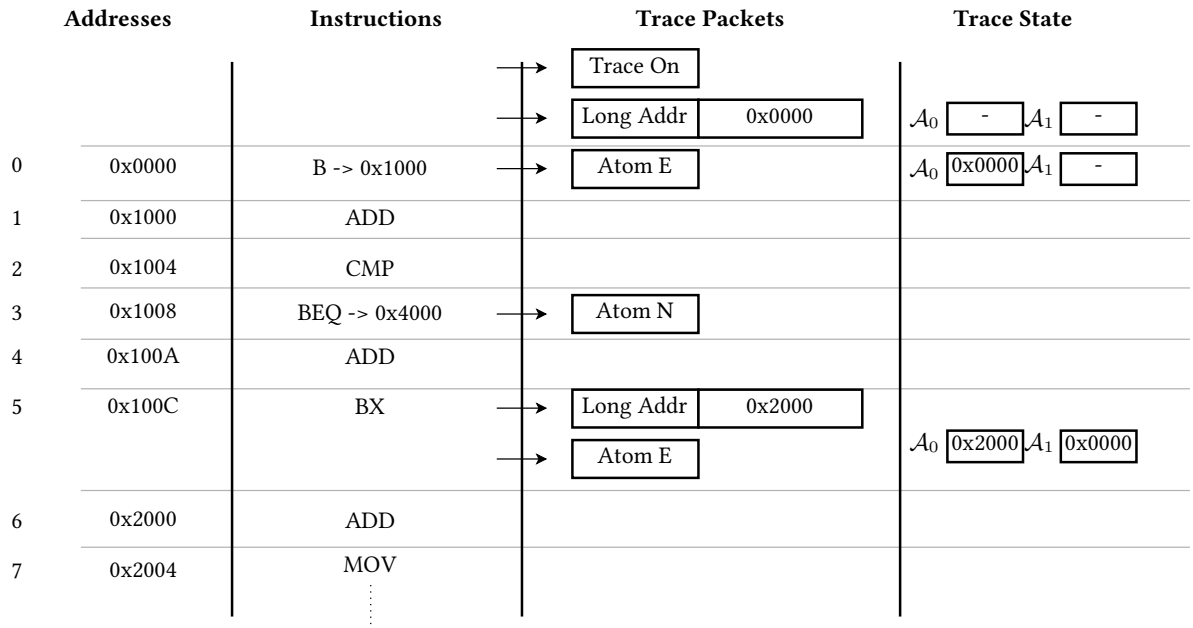


Figure 4.10: Illustration of trace protocol packets generated per executed instruction

an `Atom` element, but an address packet as well. Without updated instruction address values, a trace analyzer would not be able to infer the branch target. In this specific example, the ETM sends out another `Long_address` packet, causing the address registers to shift such that $\mathcal{A}_0$ now contains the new address, and $\mathcal{A}_1$ the previous address. We omit the third address register for simplicity. The branch target of an `Atom` element is always held in the first address register $\mathcal{A}_0$.

A standard ETM configuration could produce the trace data shown in Figure 4.10. Notice, that direct branches do not generate any address information, instead the address can only be inferred with analysis combining both the trace data *and* the program image. More specifically, the first P0 element in the control flow at the address `0x0000` causes a jump to `0x1000` . But this is not reflected in the trace packet stream or in the trace state. Indeed, the trace protocol is designed under the assumption that the program image is available during the decoding process [26]. This defeats the purpose of decoding online on the PL, or requires the decoding logic to perform the reassociation between the observed P0 elements and their address offsets using the program image on the fly. We briefly discuss this in Chapter 8, but there is a key feature that enables interpreting the trace data without the program image, namely branch broadcasting. This feature is part of a trace source and must be enabled in the configuration register of an ETM and forces the ETM to output address packets for direct branches as well. Importantly, only the US+ supports branch broadcasting, the ThunderX does not!

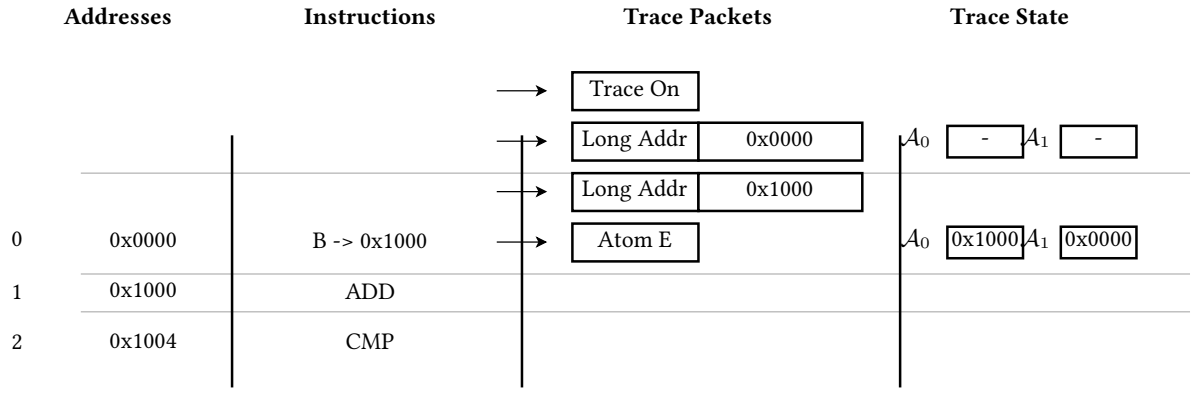


Figure 4.11: Example of first three lines with branch broadcasting enabled.

In Figure 4.11 we show the first three lines of the same program with branch broadcasting enabled, the address registers now reflect the jump location of the first P0 element, even though it is not an indirect branch. Even still, if we wish to extract the semantics of the form: "the CPU has executed a branch instruction at virtual address a_0, has taken the branch and jumped to virtual address a_1." is still not possible without the program image due to lacking jump source information, in the example the address a_0 is unresolvable. Take the `BX` instruction on line 5 with the address of `0x100C`. If we had the last jump target address (with branch broadcasting would be `0x1000`), to infer `0x100C` from the trace data we would need to know how many instructions have occurred between line 1 and line 5 — information that is simply not available to a trace analyzer without the program image. Instead, the best semantics we can produce in terms of control flow is: "the CPU has executed a branch and jumped to target address a_1". In other words, we can reconstruct the exact control flow, but the information is not precise enough to recreate an entire CFG.

In summary, we have illustrated the basic building blocks of the trace protocol, namely the `Atom` and address packets. Together with the branch broadcasting feature, any trace can be decoded without the program image in the PL, giving us the virtual instruction address of every branch target. For now, our implementation relies on branch broadcasting. Moving forward, there are around 400 different (structural) packet types in total for the ETMv4.0 specification (compared to the 11 for the previous PTM specification), as reported by Zeinolabedin et al. [89]. The additional packet types build on top of the flow tracing that is done with `Atom` and address packets by adding additional tracing features. We continue with describing the trace state before moving on to highlighting the most important packets and subsequently providing an overview of all packet types in Table 4.4.

4.5.1.1 Trace State

The trace state holds all the context information between packets in the stream that is required to interpret a packet. It includes the following values:

- Three address registers $\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2 \in$ (64 bit address, 2 bit Instruction set). When branch broadcasting is enabled, an `Atom E` element always indicates a jump to the address held in $\mathcal{A}_0$. If branch broadcasting is not enabled, this only applies to indirect branches. Address registers $\mathcal{A}_1$ and $\mathcal{A}_2$ are only used for compressing the trace data stream. The address registers also hold an additional two bits to identify the Instruction Set Architecture (ISA), which we omit for this work.
- The last seen Timestamp value (TS) (64 bit)
- Context identifier (CID) (32 bit)
- Virtual context Identifier (VMID) (32 bit)
- Security level (SL) $\in$ {Secure, NonSecure}
- Exception level (EL) $\in$ {EL0, EL1, EL2, EL3}

There are additional trace state values that we currently do not support, but are required to support resynchronizing the data trace stream to the instruction trace stream, handling tracing of speculative execution, and matching a conditional instruction to the result flags of the conditional instruction (Application Program Status Register (APSR)). These values are:

- P0 keys: each P0 element has a key associated with it that is used to match the P0 element to the P1 elements on a separate data trace stream.
- C and R keys: each conditional instruction will have an associated C key that can be matched to the result flags of the conditional instruction that each have an R key.
- `curr_spec_depth` : this value holds the number of P0 elements that have been seen by the trace analyzer that are speculatively executed but not yet retired.

A trace analyzer is responsible for keeping these values in the trace state up to date, which includes incrementing the keys and speculation depth when certain packet types are seen. These remain unimplemented for now as they are not required

for the ETMs on the ThunderX or the US+.

For parsing a stream we add additional custom values to this trace state. These are not used for used by the official protocol:

— Cycle count value (CC): this value indicates a lower bound of the number of cycles that have gone by since the last time this was updated. This value gives more precise cycle timing values than the timestamp value. The timestamp value gives a global system-wide timing overview, while this gives a fine granularity basic-block timing information. How often this value is updated depends on the ETM configuration.

From now we refer to the trace state in our decoder as $\mathcal{T}$, which contains all of the above values. Every value in the trace state is updated by packets produced by the ETM and some packets only exist to update the trace state.

4.5.1.2 Trace Packets

We dive into the details of trace packets. Every packet in the ETM the specification is identified by a header byte. The header is always exactly one byte and usually contains more information on the subtype of the packet. For example, a `Timestamp` packet has the header byte `0b00000010` or `0b00000011`. The LSB identifies the subtype of the packet — sometimes the timestamp packet will carry up to three bytes of cycle-counting information on top of the timestamp information, in this case only if the LSB of the header is 1.

A packet may also consist of only a single header byte. We have seen examples of this already in Figures 4.10 and 4.11. Every `Atom` packet is only a header and this applies to all 6 formats of the `Atom` packet.

Most packets, however, contain a payload in addition to the header. A payload is a sequence of bytes, together forming the packet payload. As a rule of thumb, the payload will generally carry values that update the trace state, as is the case with `Long Address` packets we have seen so far.

Each payload will have an integer number of bytes, so the beginning of every packet is always byte-aligned. A payload will end either implicitly, meaning a payload has reached the maximum number of bytes for a specific payload type, or explicitly, meaning the MSB of a payload byte should be interpreted as a *continuation* bit `C`, and the continuation bit is set to zero. Payload sizes are potentially *unbounded*, so $\in [0 - \infty]$. The termination condition for a packet is different for each packet type, but we categorized these into three payload types:

— Header Only (HO): These packet types consist only of a header byte.

— Fixed Size (FS): These packet types have a finite maximum size and every packet will always be the same size. There are no continuation bits in the payload bytes and all 8 bits contain valid data.

— Unbounded Continuous (UC): These packet types have no maximum size and only end if the continuation bit of a payload byte is 0.

— Bounded Continuous (BC): These packet types have a maximum size, but may have fewer bytes in the payload than their maximum size. These payloads have two termination conditions, namely if we reach the maximum size or if the continuation bit is zero. If the payload size is the maximum possible payload size of this packet type, then it has no continuation bit, in other words, we require the byte index to tell if a payload is ending.

To be clear this categorization is not made in the specification but is crucial for efficient decoding. We continue by giving an example of FS packet by showing the `Long Address` packet (Figure 4.12), to give an intuition of the payload types while simultaneously showing the effects of the packet on the trace state. The `Long_address` [2] packet will always have exactly 8 bytes of data in the payload. No checking for continuous bits is required to determine when a payload has finished, only the indices of the bytes.

[2] There are four different Long addresses packets for different ISAs, our implementation supports all of them.

Long Address header		header
Res	Addr[8:2]	byte 0
Res	Addr[15:9]	byte 1
Addr[23:16]		byte 2
Addr[31:24]		byte 3
Addr[39:32]		byte 4
Addr[47:40]		byte 5
Addr[55:48]		byte 6
Addr[63:56]		byte 7

Figure 4.12: `Long_address` packet, adapted from [26]

This packet also modifies the trace state by writing a value to an address register. Whenever any address packet is received the address register values are shifted, dropping the last address register in the process. Then, the value held by the payload of the address packet overwrites the first address value. Notice in Figure 4.12 the first 2 bits of the address are not contained in the payload and are always assumed to be zero, in this case, the ISA has 64-bit instructions.

We formalize the address update in Algorithm 1, the range of address bits contained by the payload, h and l, depend on the ISA, which will always be encoded into the header of the packet. For the `Long Address` packet in Figure 4.12, this will always be from [63:2].

Algorithm 1: Updating the address registers in the trace state.

input : Address registers $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$, new address $\mathcal{A}_n$,
where address $\mathcal{A}_n$ holds bits of new address information in the range $[h{:}l]$
output : Updated address registers $\mathcal{A}'_1$, $\mathcal{A}'_2$, $\mathcal{A}'_3$
Procedure *shift* **is**
  $\mathcal{A}'_0 \leftarrow \mathcal{A}_0$
  $\mathcal{A}'_1 \leftarrow \mathcal{A}_0$
  $\mathcal{A}'_2 \leftarrow \mathcal{A}_1$
Procedure *overwrite* **is**
  $\mathcal{A}'_0[h{:}l] \leftarrow \mathcal{A}_n[h{:}l]$
`/* For a full packet: */`
Procedure *updateAddress* **is**
  $overwrite \circ shift$

Compare the `Long_address` packet to the `Short_address` packet in Figure 4.13, which is an example of BC packet type. This packet is bounded continuous since it may have a maximum of two payload bytes, but also could only have one. This determination cannot be made solely from analyzing the header byte, but the MSB of byte 0, the continuation bit, needs to be checked to see if there is another payload byte incoming.

Short Address header		header
C	Addr[8:2]	byte 0
Addr[16:9]		byte 1

Figure 4.13: `Short_address` packet, adapted from [26]

Packets like the `Short_address` packet are also the reason why $\mathcal{A}_0$ cannot be completely overwritten when seeing a new address packet. The `Short Address` packet is way for Coresight to compress the trace bandwidth and avoids sending out redundant address information. For short jumps the target address will already share many of the more significant bits

with the value already stored in $\mathcal{A}_0$. If we return to our initial trace example in Figure 4.10, an ETM will likely avoid sending out the `Long_address` packet on line 5 in favor of sending out a `Short_address` packet since the most significant bits are shared with the address already in $\mathcal{A}_0$, which are all zero.

At this point we introduce the last address packet type, the `Exact_match_address` packet. This packet is another example of a HO packet type, like the `Atom` packet, that holds the information in the two least significant bits of the header, index $i \in [0,2]$.

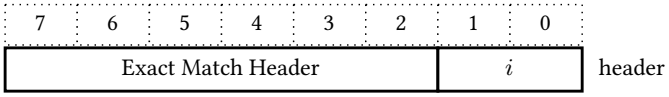


Figure 4.14: `Exact match address` packet, adapted from [26]

The purpose of this packet is similar to the `Short_address` packet and is meant to minimize the trace bandwidth produced by a trace source. If an address is already contained in one of the address registers, the ETM will send out an `Exact address packet`, where the new A_n is set to the address in the address register A_i. The trace state update still follows Algorithm 1 as usual, the A_n depends only on the address registers instead of on any payload data.

In practice, this packet will be observed if we are in a loop with no branches inside the loop and branch broadcasting is enabled. If we assume that for every loop iteration the source sends out an address packet and an `Atom` packet, the trace of each iteration will produce 2 bytes with `Exact match address packets` as opposed to producing 10 bytes of when only using `Long_address` packets, a 5× reduction in trace data.

At this point, after introducing a few different types of packets from the trace protocol, we partially formalize the decoding process. Packets modify the trace state $\mathcal{T}$ and one aspect of decoding the trace stream is applying the trace state updates. For a packet p that belongs to the set of all ETM packets $\mathcal{P}$, we can describe the decoding process as a state update $d : \mathcal{P} \times \mathcal{T} \rightarrow \mathcal{T}'$, which handles an entire packet $\mathcal{P}$ at a given trace state $\mathcal{T}$ a and produces a new trace state $\mathcal{T}'$.

For the address packets we have observed so far, following the address register update algorithm, it holds that the decoding function d is byte-wise decomposable (Equation (4.1)). By this, we mean that modifications do not have to be applied for a whole packet, but can be broken down into smaller sequential updates for each byte p_{b_i} of a packet p. This is important for the decoding process, we elaborate in Section 4.5.3.2.

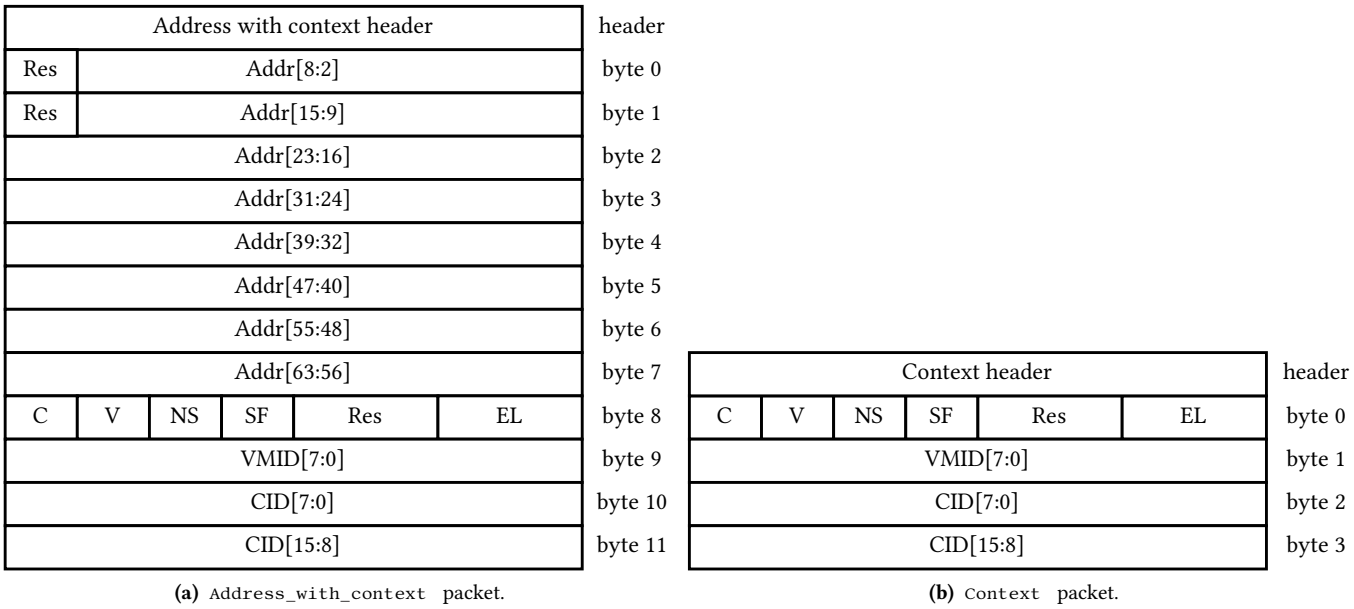
$$d(p) \equiv d(p_{b_n}) \circ d(p_{b_{n-1}}) \ldots \circ d(p_{b_0}) \tag{4.1}$$

However, we require an extra step to make the updates byte-wise decomposable, we break down the $updateAddress$ procedure from Algorithm 1 into $shift$ and $overwrite$. The $overwrite$ procedure can be invoked at each payload byte, while the $shift$ procedure is invoked before any $overwrite$ procedures are performed, meaning once at the beginning of a packet.

Take any address packet we have introduced, for example, the first byte of both the `Long_address` or `Short_address` packet, which holds the bits [8:2] of the new address packet. The $overwrite$ procedure from Algorithm 1 can be applied for only this payload byte with $h = 8$ and $l = 2$ without consideration of any later bytes in the payload, which can be applied ad hoc when these payload bytes are observed. Of course, the $shift$ needs to be performed before we start overwriting the registers.

As a matter of fact, we observe byte-wise decomposability from Equation (4.1) to hold for every single packet type in the ETM protocol and this observation is critical for our trace decoding process.

We proceed with additional important characteristics of ETM packets. The ETM protocol produces some packets that consist internally of two different packet types concatenated with each other, so we have a payload for multiple packets under a single header. We discussed an example of this at the beginning of this section, where a cycle-counting payload is added to the `Timestamp` packet payload. Another example is illustrated in Figure 4.15. Notice the first 8 payload bytes of the `Address_with_context` packet match the payload of the `Long_address` packet. The remaining 4 bytes contain the



(a) `Address_with_context` packet.

(b) `Context` packet.

Figure 4.15: Comparison of composite packet `Address_with_context` packet and `Context` packet, adapted from [26].

payload of a `Context` packet (shown in full in Figure 4.15b), that updates the trace state values CID and VMID in addition to the address registers.

We refer to packets like the `Address_with_context` packet as *composite* packets. The existence of composite packets complicates the design of a trace decoder. From an implementation perspective, this requires large parts of the codebase to be duplicated and overcomplicates the decoding process. As an alternative, we split all the composite packets into smaller chunks, such that multiple bytes that update the same values of the trace state form a single packet. Concretely, for the example of the `Address with context` packet, the entire packet is split into a `Long_address` packet, a `Context info` packet, a `VMID` packet and a `CID` packet. This allows us to reuse the code responsible for decoding a `Long_address` packet when decoding a composite packet. As a result, the number of structural packet types and subtypes is greatly reduced. Some of these packets, like the `VMID` packet, do not exist as true packet types in the specification. We call these *internal* packets and they only exist in our internal decoding process, not at the specification level.

Complicating the decoder even further, the entire structure is not always defined in the header. In most cases, the structure of a payload can be inferred after parsing the header byte, like in both `Short_address` packet and `Long_address` packet, where the header byte contains information on the ISA and which payload byte corresponds to the which bits in the new address $\mathcal{A}_n$. This is not always the case, as sometimes the structural information does not fit entirely into the header. The packet may include an additional information byte that contains structural information on the packet that is required for parsing. See, the `Context` packet in Figure 4.15b. The first payload byte is an informational byte: it contains both information that is relevant to the trace state, *and* to the structure of the packet. The first two bits of information byte, marked EL set the exception level of the current execution, SF denotes that the processor is either in AArch32 or AArch64 and NS defines the security level. The bits C and V, however, let the trace analyzer know whether the payload contains VMID and CID data respectively — this is structural information required to properly decode the `Context` packet.

We have also abstracted away some details of the address packets. Each address packet will have a subtype according to the ISA and address size. So for both `Short_address` , `Long_address` , there are in reality 4 packet types, namely `Short_address_32bit_IS0` , `Short_address_64 IS0` , `Short_address_32_IS1` , and so on. We are required to distinguish these packet types as they determine the length of the packet, and to which address bits each byte of the packet maps to. IS0 applies to any ISA that is halfword aligned and 32-bit vs 64-bit differentiates AAarch64 vs AAarch32. The ETM packets of both ThunderX and Cortex-A53 will both fall under the category of 64-bit IS0.

The ETM protocol is complex and there are a large number of different state components and packet types to keep track

of. We condense the key learnings from this section. The packets we have introduced show all the important concepts and all the requirements to successfully decode the trace data. We repeat that the combination of address and `Atom` packets are the most important and only these packet types are required to recreate the program flow. An equally important feature of the protocol is the fact that the decoding function is decomposable into byte-wise applications of a decoding function. This gives us flexibility in the trace decoding design, especially important for parallelization.

Before moving on to the main challenges faced with designing a trace decoder, we give an overview of all possible packet types in Table 4.4. We refrain from showing the same level of detail as we have done for the packet types introduced so far, and for more details please refer to the protocol specification [26]. We emphasize that the packets we introduce in Table 4.4 do not match exactly with the specification, as we break down all instances of composite packets and represent them as internal packets. We show both internal and composite packets in the table, marked by column Composite Type (CT) in Table 4.4.

For each packet, we show what effects it has on the trace state, the Packet Format (PF) ∈ {FS=Fixed_Size,UC=Unbounded_Continuous, BC=Bounded_Continuous, Ho=Header_Only}, and the Packet Size (PS). We mark whether a packet is composite or internal in the CT column ∈ {C=Composite, I=Internal, N=None}. Composite packets are only part of the specification and not part of our decoder. Each composite packet is broken up into internal packets. These packets are not in the specification. If the CT column is `None`, the packet is neither composite nor internal, and the decoder and the specification share the interpretation of the packet. For each packet, we specify whether the ThunderX (column TX) or the Zynq Ultrascale+ (column US+) can produce these packets. The last column (column IM) shows whether the packet can be decoded by our current implementation.

The packets are categorized into packet types based on their semantics. The packet types are synchronization, timing, address and context, events, atom, speculation, conditional, and miscellaneous. The synchronization packets inform the trace analyzer of trace stream metadata. For example, the `A-Sync` packet is always the first packet seen in the tracing session and it denotes the beginning of the next packet. To start decoding, a trace analyzer must scan for an `A-Sync` packet, after which it can interpret the next byte as the header of the next packet. Another example is the event packet type that contains an event code for the PMU events discussed in Section 3.5.

Some packets have multiple different formats, marked in Table 4.4 with Fx. Each format will have a different packet structure. Having multiple formats allows for different levels of compression relying on the trace state. For example the `Atom` packet has 6 different packet formats that all encode a different number of `Atom` elements and different sequences of `Atom E` and `Atom N` elements.

Table 4.4: Complete list of ETM packets from the perspective of our trace decoder. PF=Packet Format, CT=Composite Type, PS=Payload Size, TX=producable by ThunderX, US+=producable by US+, IM=Implemeted

Type	Packet	State Effect	PF	CT	PS	US+	TX	IM
Synchronization	A-Sync	–	FS	N	11	✓	✓	✓
	Discard	–	FS	N	1	✓	✓	✓
	Overflow	–	FS	N	1	✓	✓	✓
	Trace Info	P0 key, curr_spec_depth	UC	C	1-∞	✓	✓	✓
	Trace Info Plctl	–	UC	I	1-∞	✓	✓	✓
	Trace Info Info	–	UC	I	1-∞	✓	✓	✓
	Trace Info Key	P0 key	UC	I	1-∞	✓	✓	✓
	Trace Info Spec	curr_spec_depth	UC	I	1-∞	✓	✓	✓
	Trace On	–	H	N	0	✓	✓	✓
Timing	Timestamp	TS	BC	N, I	1-11	✓	✓	✓
	Timestamp with Cycle Count	TS, CC	BC	C	1-11	✓	✗	✓
	TS Cycle Count	CC	BC	I	1-3	✓	✗	✓
	Cycle Count F1	CC, curr_spec_depth	BC	C	1-11	✓	✗	✓
	Cycle Count F2	CC, curr_spec_depth	BC	N	1-11	✓	✗	✓
	Cycle Count F3	CC, curr_spec_depth	HO	N	1-11	✓	✗	✓
Addresses& Context	Context	CID, VMID, SL, EL, ISA	FS	C	0-9	✓	✓	✓
	Context Information Byte	SL, EL, ISA	FS	I	1	✓	✓	✓
	Context VMID	VMID	FS	I	0-4	✓	✓	✓
	Context CID	CID	FS	I	0-4	✓	✓	✓
	Address+Context	$\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$,CID, VMID, SL,EL, ISA	FS	C	5-14	✓	✓	✓
	Exact Match Address	$\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$	HO	N	0	✓	✓	✓
	Short Address 32bit IS0	$\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$	BC	I	1-2	✓	✓	✓
	Short Address 64bit IS0	$\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$	BC	I	1-2	✓	✓	✓
	Short Address 32bit IS1	$\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$	BC	I	1-2	✗	✗	✓
	Short Address 64bit IS1	$\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$	BC	I	1-2	✗	✗	✓
	Long Address 32bit IS0	$\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$	FS	I	4	✓	✓	✓
	Long Address 64bit IS0	$\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$	FS	I	8	✓	✓	✓
	Long Address 32bit IS1	$\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$	FS	I	4	✗	✗	✓
	Long Address 64bit IS1	$\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$	FS	I	8	✗	✗	✓
Events	Event	–	HO	N	0	✓	✓	✓

Type	Packet	State Effect	PF	CT	PS	US+	TX	Im
Atom	Atom F1	curr_spec_depth, P0 key	HO	N	0	✓	✓	✓
	Atom F2	curr_spec_depth, P0 key	HO	N	0	✓	✓	✓
	Atom F3	curr_spec_depth, P0 key	HO	N	0	✓	✓	✓
	Atom F4	curr_spec_depth, P0 key	HO	N	0	✓	✓	✓
	Atom F5	curr_spec_depth, P0 key	HO	N	0	✓	✓	✓
	Atom F6	curr_spec_depth, P0 key	HO	N	0	✓	✓	✓
Speculation	Commit	curr_spec_depth	UC	N	1-∞	✗	✗	✗
	Cancel F1	curr_spec_depth	UC	N	1-∞	✗	✗	✗
	Cancel F2	curr_spec_depth	HO	N	0	✗	✗	✗
	Cancel F3	curr_spec_depth	HO	N	0	✗	✗	✗
	Mispredict	curr_spec_depth	HO	N	0	✗	✗	✗
Conditional	Conditional Instr F1	C key	UC	N	1-∞	✓	✓	✗
	Conditional Instr F2	C key	HO	N	0	✓	✓	✗
	Conditional Instr F3	C key	FS	N	1	✓	✓	✗
	Conditional Res F1	C key, R key	UC	N	1-∞	✗	✗	✗
	Conditional Res F2	C key, R key	HO	N	0	✗	✗	✗
	Conditional Res F3	C key, R key	FS	N	1	✗	✗	✗
	Conditional Res F4	C key, R key	HO	N	0	✗	✗	✗
	Conditional Flush	C key	HO	N	0	✗	✗	✗
Miscellaneous	Ignore	—	HO	N	0	✓	✓	✓
	Function Return	curr_spec_depth, P0 key	HO	N	0	✗	✗	✗
	Exception Return	curr_spec_depth, P0 key	HO	N	0	✓	✓	✗
	Exception	$\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$, EL	UC	C	3-12	✓	✓	✗
	Q packet	curr_spec_depth, P0 key	UC	N	1-∞	✗	✗	✗
	Numbered Data Sync Mark	—	HO	N	0	✗	✗	✗
	Unnumbered Data Sync Mark	—	HO	N	0	✗	✗	✗

4.5.2 Design requirements of a hardware trace decoder

In this section, we outline the challenges of designing a trace decoder and conclude with a set of requirements our trace decoder implementation must fulfill. The trace protocol is designed for maximal compression to keep the volume as low as possible. The compression relies heavily on the trace state. This presents a challenge in parallelizing the trace decoding process — directly juxtaposed to worst-case throughput requirements of decoding four bytes a cycle when all trace data is produced by the same source.

First and foremost, the trace data is received in the form of a byte stream and needs to be reconstructed as a packet stream or at least the packet-information needs to be known by the trace decoder. While the state updates can be applied on a byte-by-byte basis, as we have discussed in Section 4.5.1.2, this process requires knowing the context of each byte in relation to

the packet the byte belongs to (payload index for example). The same challenges are shared whether we wish to first entirely reconstruct packets from the byte stream before updating states or applying updates directly after each byte.

This leads us to the first challenge: dealing the context of each byte in the trace data. To process a byte b_i, the previous byte b_{i-1} must be processed. This applies to both a packet representation and byte representation. The reason for this is twofold: First, We require some form of stream state to keep track of which bytes make up a payload of a packet and which bytes are header bytes. Take Figure 4.16, where three of the four bytes received in a single cycle contain a full `Short Address` packet. As a reminder, a `ShortAddress` packet may contain either one or two payload bytes, denoted by the MSB of the first payload byte. It is impossible to know whether the second byte is a header for a new packet or contains the final payload byte before, at least partially, processing the first byte.

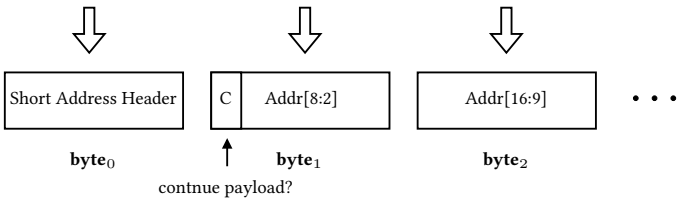


Figure 4.16: A short address bounded continuous packet that requires checking Most Significant Bit (MSB) of byte 1 to decode byte 2.

Our second challenge, the protocol itself has a trace state. The most prominent being the three address registers we covered. When an `Atom` element is sent, it relies on the fact that all previous address packets have been decoded and the registers hold the proper addresses. Combined with the possibility of receiving multiple header-only packets in succession, we are forced into a strong sequential processing requirement. Take the input data as demonstrated by Figure 4.17. The address of the branching instruction represented by the `Atom` element in the second byte is directly determined by the `Exact Match Address` packet, which may point to one of the three address registers in the state. Being able to handle multiple sequential header-only packets is crucial to meet throughput requirements, as Zeinolabedin et al. observe that around 40% of all packets observed throughout their tracing sessions were header-only packets [89].

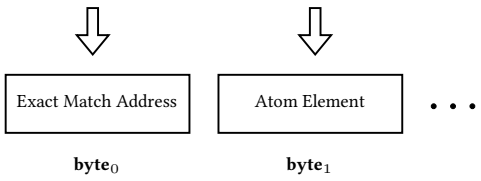


Figure 4.17: Two single byte packets with direct dependency between byte 0 and byte 1.

In the same vein, these properties make the protocol very susceptible to crashing and burning when data is dropped. Even losing a single byte will result in an incorrect stream state, effectively garbling everything up until the next `A-Sync` packet, after which we can restore the stream state.

Making things worse, even if we resync the stream state and once again correctly decipher packet boundaries, if any previous packet has been processed incorrectly, the address registers could be in an incorrect state. This may corrupt the trace state indefinitely for any packet that relies on the address registers until the values in the register are replaced by newer address registers. It remains unclear to us whether the protocol provides a "clean slate" to the trace state after an `A-Sync` packet, as it is not explicitly stated in the documentation. It's plausible that an `A-Sync` packet does not rely on any state modifications that occurred prior to the `A-Sync`, or something akin to this procedure, but this remains speculation and would have to be tested in practice.

Recapping the requirements, the decoder must (1) be able to handle four bytes of data per cycle at a data rate of 250 MHz, (2) each byte must be processed sequentially since there exists a byte-by-byte interdependency, the same applies to each packet, and (3) we should avoid dropping bytes at any cost or the trace state may be indefinitely corrupted.

Running the decoder to process a single byte per cycle at 1000MHz would fulfill requirements (2) and (3), but not (1), as it exceeds the maximum possible clocking frequency of the Zynq Ultrascale+, being slightly below 800 MHz [16]. A possible remedy would be adding a buffer and working under the assumption that the data rate from an individual source only reaches a data rate of four bytes per cycle in rare scenarios, and bursts can be absorbed. This (admittedly weakly) violates requirement (3) and is a fool's errand if we wish to expand the design to the ThunderX running at much higher clock speeds than the Cortex-A53 processors on the Zynq Ultrascale+.

Another design option is to hold trace data bytes in a buffer until a complete packet is observed, decode the entire packet in one cycle, and repeat. This is very similar to all prior ETM and PTM parsers [74, 84, 89], but this design has two problems: It cannot deal with unbounded payload sizes, and the worst-case throughput suffers. The worst-case throughput suffers when we see successive header-only packets. To deal with this, we must implement some form of parallel processing of packets, at which point we have the same problem as we started with.

4.5.3 Trace Decoding

In this section we introduce a trace decoder that fulfills all the requirements we outlined in Section 4.5.2. The key idea behind our trace decoder is to process the trace data sequentially byte-by-byte, but simultaneously in parallel with an *unrolled* byte stream. In other words, if we have a decoding function d that can process a single byte b and we want to handle 4 bytes in one cycle, the decoding function is applied once to b_1, twice to b_2 and so on. This way we can fulfill the strong sequential requirements and state dependencies between each byte while also achieving high enough throughput.

First, to be able to process the trace data byte-by-byte we need a stream state $\mathcal{S}$ that provides context information of the current byte to be handled. Second, the goal of the trace data is not only to reproduce the trace state updates, but also extract semantics from trace data — we want to be able to follow the control flow and retain the processer execution environment. Each byte might encode an action taken by the processor, like executing a branch instruction and jumping to an address a_0. This information is contained in the output produced by the decoder, we refer to this abstractly as $\mathcal{O}$. At a high level, the decoding function of a single byte can be interpreted as a function:

$$d : \mathcal{S} \times \mathcal{T} \times b \to \mathcal{S}' \times \mathcal{T}' \times \mathcal{O} \tag{4.2}$$

and the decoding of an entire byte stream of n bytes $b_0 \dots b_n$ is just the sequential chaining of decoding functions $d(b_n) \circ d(b_{n-1}) \dots \circ d(b_0)$, where only the stream state and trace state are required for processing the next state and the control flow can be reconstructed using the outputs of the decoding function $\mathcal{O}_0 \dots \mathcal{O}_n$.

Using the singe-byte decoding function, we can build a parallelized, or unrolled decoding function (Algorithm 2), that can handle 4 bytes in one cycle. In simple terms, to process 4 bytes simultaneously, we are duplicating the circuit that parses and decodes a byte of raw trace data 10 times onto the FPGA.

Algorithm 2: Unrolling the byte stream.

input : bytes $b_0 \dots b_3$, initial trace state $\mathcal{T}_0$,initial stream state $\mathcal{S}_0$
output : updated trace state $\mathcal{T}_0$, stream state $\mathcal{S}_0$, after processing all input bytes,
four output values for each processed byte $\mathcal{O}_0 \dots \mathcal{O}_3$

```
/* With a throughput of 4 b/cycle:   */
```

Procedure *decode_unrolled* **is**
  $\mathcal{S}_1, \mathcal{T}_1, \mathcal{O}_0 \leftarrow d(b_0)$
  $\mathcal{S}_2, \mathcal{T}_2, \mathcal{O}_1 \leftarrow d(b_1) \circ d(b_0)$
  $\mathcal{S}_3, \mathcal{T}_3, \mathcal{O}_2 \leftarrow d(b_2) \circ d(b_1) \circ d(b_0)$
  $\mathcal{S}_0 = \mathcal{S}_4, \mathcal{T}_0 = \mathcal{T}_4, \mathcal{O}_3 \leftarrow d(b_3) \circ d(b_2) \circ d(b_1) \circ d(b_0)$

The advantage of this decoder design is we no longer require any buffering, it can never drop any data as long as the frequency is the same as the TPIU, and high throughput through parallelization can be achieved – the only the remaining difficulty is for the design to be simple enough such that the longest timing path of 4 successive decoding circuits is less than

4 ns (for 250MHz).

Somewhat surprisingly, the longest path of a naive implementation of 4 chained decoding functions takes around 7 nanoseconds, giving us a maximum operating frequency of around 140MHz and giving us a throughput of 560MB/s, already beating most of the existing trace decoders, but still cannot keep up with the worst-case scenario of handling 4 cycles per second produced by a single trace source.

To achieve high decoding throughput, we additionally pipeline the unrolled trace decoding algorithm. To understand the pipelining, we first introduce the stream state that we require for decoding and subsequently show each stage of the decoding pipeline.

4.5.3.1 Stream State

We introduce the *stream state* as an additional state component separate from the trace state to the trace decoder. The stream state consists of bookkeeping information, allowing the decoder to interpret each byte as part of a packet. To achieve this goal, we track the following information in the stream state:

- Stream mode $\in \{$ `Synchronization` , `Expecting_Header` , `Payload_Unbounded_Continuous` , `Payload_Bounded_Continuous`, `Payload_Fixed_Size` , `Information_Byte` $\}$
- Current header type, where the header type is one of the header types in Table 4.4
- Current payload index (pi), denoting the index of the current payload byte in the packet. Only used when in one of the payload modes.
- Payload size (ps), indicating the end of a packet of in either `Payload_Bounded_Continuous` or `Payload_Fixed_Size` mode.
- Three lookahead states that each include all the above values. We switch to the next lookahead state when a complete package is processed. This is required to handle composite packets as multiple internal packets, that no longer have a header associated with them.

We show the stream state values that the decoding process will carry at every byte when decoding an `Address_with_context` packet in Figure 4.18.

Initially, all previous packets have been completely parsed, indicated by the stream state being in `Expecting_header` mode. This lets the decoder know that the next byte should be interpreted as a header of a new packet.

Decoding the header gives us most of the structural information on the entire packet, the decoder knows at this point everything it should see up until the information byte of the composite `Context` packet. The stream state switches to `Payload_Fixed_Size` mode with a payload size of 8 and sets the header value to `Long_address` . The payload index is initialized to 0 and incremented for each payload byte until a termination condition is reached. Note that the decoder behaves the same way as if it had seen a `Long_address` packet on the trace stream and the decoder has no notion of the composite `Address_with_context` packet, except for the lookahead values in the stream state.

The key to distinguishing the `Long_address` and the `Address_with_context` packet types in the decoding process is the lookahead field. Based on the header byte, the decoder knows to expect an internal `Context` packet (with no header) after the internal address packet is fully decoded. As soon as the termination condition is reached for the `Long_address` packet, the stream state adopts the state held in the lookahead state and consumes it. Every packet type that is not internal or composite will always have the first lookahead stream mode to be `Expecting_Header` .

The `Context` packet is one of the few packet types that require an information byte to fully determine the structure. The processing of information bytes has a separate stream mode — the lookahead stream mode is set to the `Information_byte` and the lookahead header type is `Context` . We cannot set any further lookahead values, since the future structure is unclear until we read the information byte.

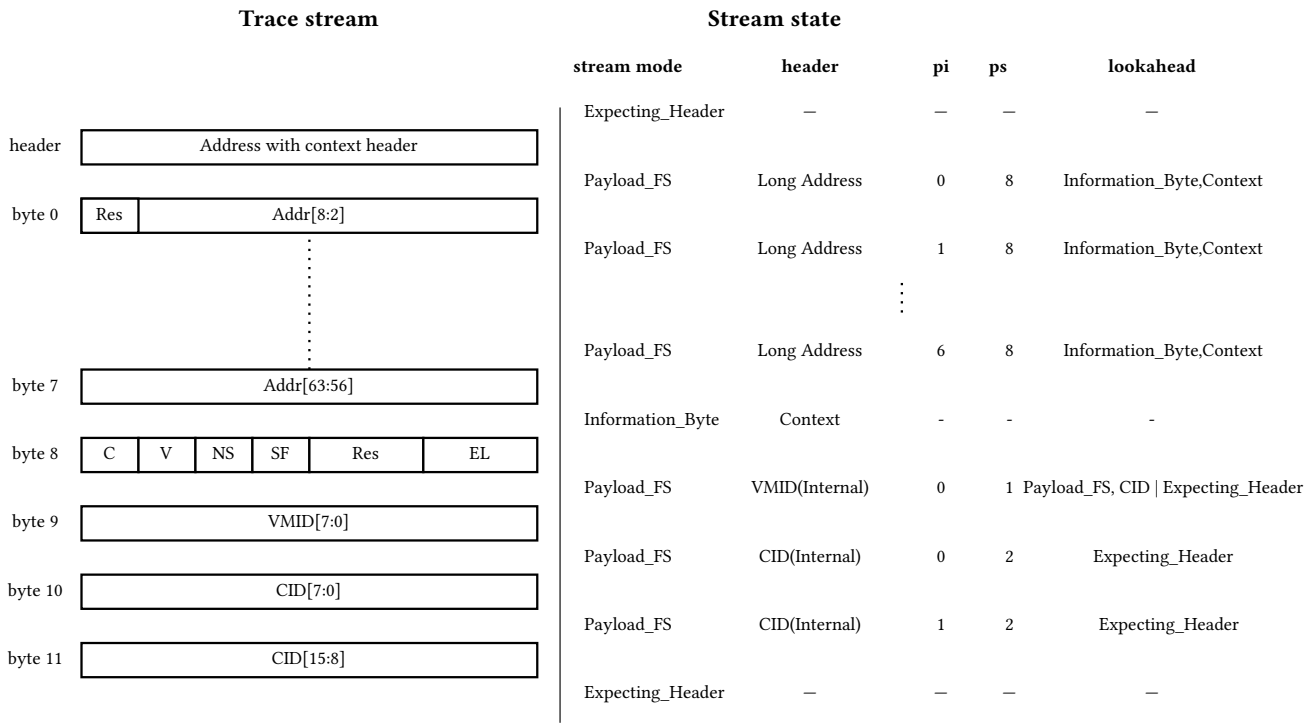


Figure 4.18: Streamstate at before and after processing each byte of an `Address_with_context` packet. We omit everything but the stream mode and header type in the lookahead field, but this will include payload size as well. We show only one of three lookahead fields.

Once the information byte (byte 8) is processed, the packet structure is fully known until the next header. In our example, both bits C and V are set in the information byte and we can expect one byte of VMID and two bytes of CID data. Both of these are internal packet types and the boundary between the packets is not denoted by a header so we rely on the lookahead once more. This time we set the lookahead state to the state required for decoding the internal CID packet. The second lookahead will be `Expecting_Header` , indicating the full packet is processed after the internal CID packet and the decoding process repeats for the next byte available on the stream.

The stream state contains all the information required to handle updates to the trace state. Take the stream state before byte 11. The stream state holds the header type of the packet (CID) and the payload index. Based on the stream state, the trace state bits[15:8] of the field holding the CID can be overwritten with the received byte.

The lookahead fields in the stream state are not strictly necessary for a decoding process but ease the required programming efforts greatly. The packet `Trace_info` is similar to the `Context` packet, but has four different internal packets that may or may not be present indicated by an information byte, whereas the `Context` packet only has two, the CID and VMID packets. Without the lookahead functionality, we would have to split up the trace info into 2^4 packet types, instead of just setting upto 3 lookahead states. This reduction combats the reported requirements by Zeinolabedin et al. from around 400 structural packet types to somewhere between $\sim$30-40.

4.5.3.2 Udating the Trace State

Handling stream state allows us to interpret the place each byte has with respect to the packet it belongs to. In this section, we cover how updates to the trace state are computed from the stream state.

The stream state at b_{i-1} alongside the value of byte b_i uniquely identifies the update that must be performed on the trace state at byte b_i. Each update operation can be encoded into action code, that uniquely identifies the trace state update. We

show the actions that can be applied to a trace state:

- `shift_address` : This action shifts the address registers according to the $shift$ procedure from Algorithm 1. This action can be applied whenever the stream state has stream mode `Expecting_header` and byte b_i has a value of any address packet header.
- `exact_address_x` : This action shifts the address registers, while also putting the address held by the previous address register $\mathcal{A}_x$ into the first address register $\mathcal{A}_0$.
- `update_address_h_l` : This action applies the $overwrite$ procedure from Algorithm 1. The stream state header type, index, and stream mode are all used to generate this action code. There is an action encoding for each possible h and l values from all address packet types.
- `update_xxx_h_l` : This action applies an update to a trace state value similarly to the way the address update occurs. We group these together for brevity, but this includes actions such as `update_vmid_h_l` , `update_cid_h_l` and `update_timestamp_h_l` .

We show the same example from Figure 4.18 in Figure 4.19, omitting the lookahead states, but instead showing the generated action codes. Note that the bytes are still required when generating the action codes. The trace state updates are omitted but self-explanatory from the action codes.

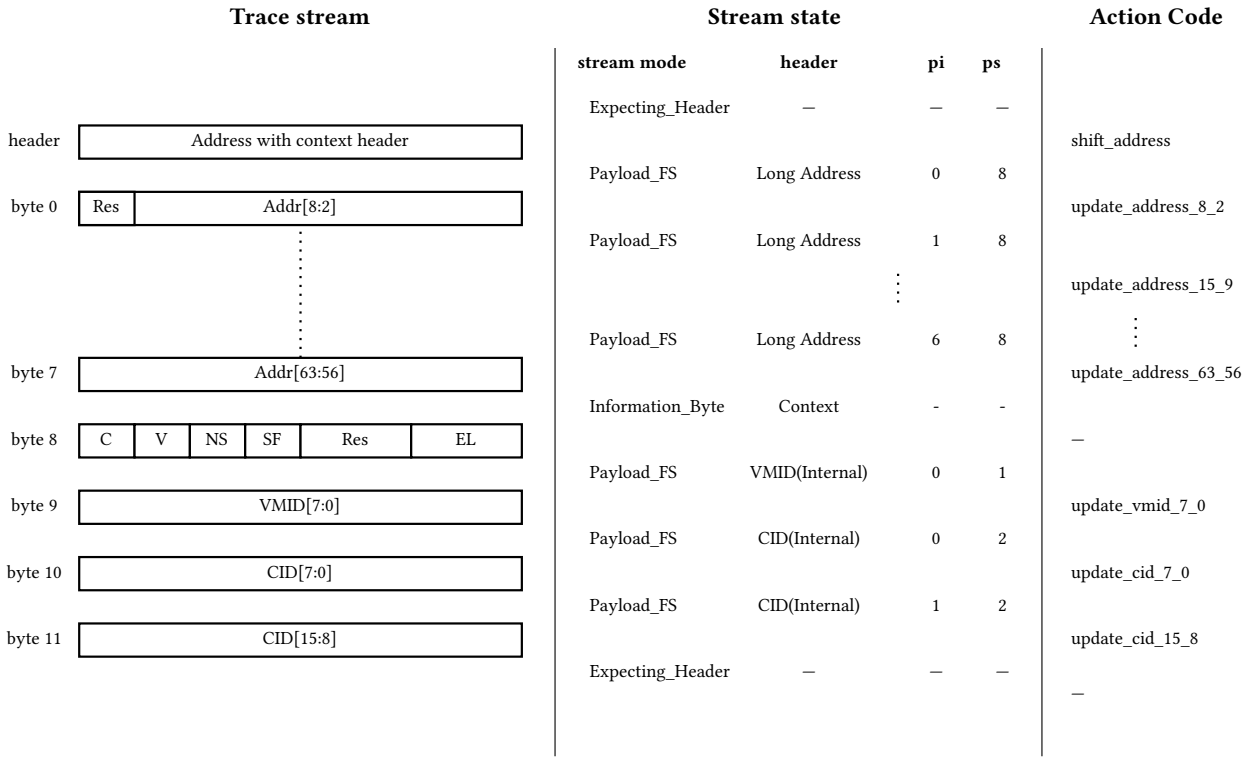


Figure 4.19: Action codes generated based on stream states and trace stream byte.

Observe the first line in the stream state. The stream mode is `Expecting_Header` . In this mode, the action applied to the trace state depends on the resolved header type. In this case, we have an address header. More specifically, an `Address_with_context` header. From the specification, we know that the following payload bytes will contain address values, but the address registers must first be shifted. From the stream state, we generate the action code `shift_address` .

The stream state for the next line is in `Payload_FS` mode, meaning we are inside a payload. The action code can be determined without the header lookup and only depends on the header value in the stream state and the index of the payload.

Based on the specification, the decoder can determine that the first 7 bits of the current byte must overwrite the first address register bits $\mathcal{A}_0[8{:}2]$. Which bits to overwrite depend also on the subtype of the `Long_address` packet (IS0 or IS1). Overall, this is enough information for the decoder to generate the action code `update_addres_8_2` . The action code encodes all parameters required to perform the *overwrite* procedure in Algorithm 1.

Action codes are generated in the same way for the rest of the payload bytes of `Long_address` packet, until we reach the internal `Context` packet. The information byte in our example requires no state updates and a `nop` action code is generated. For a full implementation of the specification, this would generate an action code that updates the exception level and security level, but as of now this is not implemented in our design. Moving on, the payloads of the internal packets `vmid` and `cid` are processed similarly to the payload of the address packet, and generate the respective action codes of `update_vmid_7_0` , `update_cid_7_0` . and `update_cid_15_8` .

Once the action codes are generated, updating the trace state is straightforward and follows naturally from the action codes.

4.5.3.3 Producing Output

Once an action is processed we can produce a valid output $\mathcal{O}$. We keep this intentionally vague for now, as what the output should contain may depend on the use case of the tracing session. To avoid producing any intermediate output, the decoder should only produce output when a complete packet is processed. In other words, each packet should appear atomic to the consumer of the decoded trace stream. Any trace state should only be observed after processing a complete packet, so whenever the stream mode is `Expecting_Header` . The output $\mathcal{O}$ contains a valid signal for this purpose.

An example of a possible output would be to output the target addresses of an `Atom` element. This would involve sending out the first address register $\mathcal{A}_0$ when receiving an `Atom` element HO packet. Additionally, one could include some context information of the trace state at this point, for example including timestamping information, CID and VMID.

A further relevant metric is events for the embedded PMU events in the packet stream. These come in the form `Event` HO packets that have four bits to encode PMU event. Resolving events based on these four bits cannot be done on the fly without prior hard coding of the values, as the encoding depends on the ETM configuration. By resolving the event, we mean being able to map these four bits into a concrete hardware events the PMU can produce. An example event is `L1I_Cache_Refill` . For now, we just output the event code directly into $\mathcal{O}$ and leave the event resolving to future stages.

4.5.3.4 Pipelining the unrolled decoder

In this section, we describe how we can pipeline the unrolled trace decoder from Algorithm 2. We mentioned that without pipelining the decoding process will take around 7ns on the worst timing path, limiting our maximum operating frequency. This follows from 4 applications of the decoding circuit in one cycle, resulting in a path that must traverse somewhere around 30 levels. We avoid nested logic as much as possible and break down the decoding process into minimal self-contained processing chunks.

There is an inherent limitation to the possible pipelining that can be done due to sequential requirements between states of the protocol. Each process that requires information from the previous byte cannot be broken down into stages without breaking the sequential requirement. This is most prevalent when processing the stream state, as the stream state at each byte will always depend on the previous stream state.

We break the decoding down into the following stages:

— Header preprocessing: Assuming the byte is a header, resolve the header type.

— Stream state process: Perform the same steps as shown in Figure 4.18. Each byte is computed from the previous stream state and the header type of the current byte (only required if we are expecting a header).

— Generate action code process: Based on the current stream state and header type of the current byte, resolve how the trace state should be modified and encoded into an action code, for example, `update_address_8_2` or `shift_address` used in Figure 4.19.

— Action process: This will perform the action on the trace state based on the action code and the byte to be processed, for example, perform the $shift$ from Algorithm 1 when we see the code `shift_address` .

The final circuit is visualized in Figure 4.20.

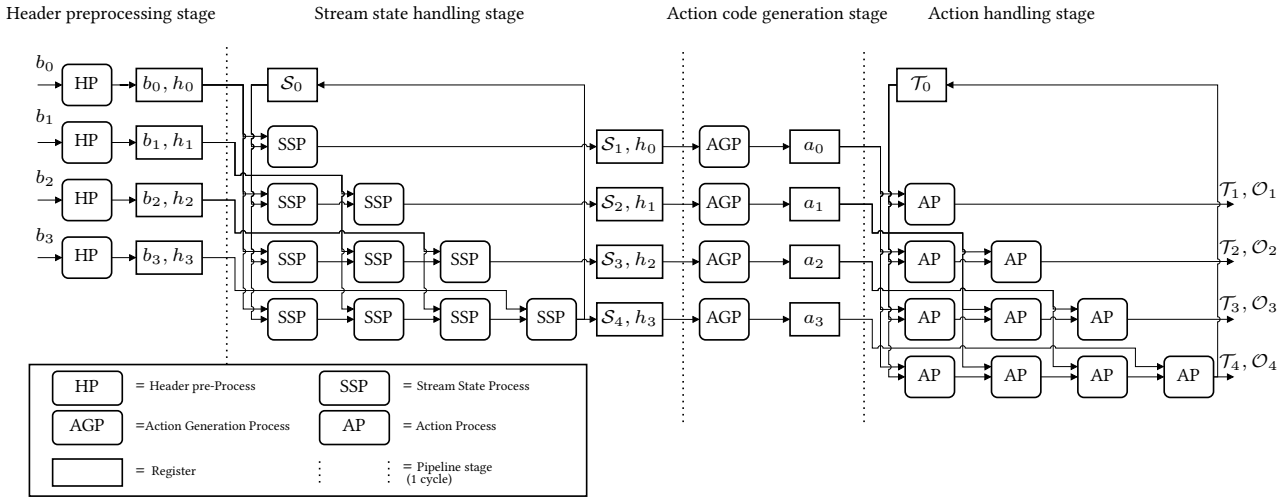


Figure 4.20: Full trace decoding process after unrolling and pipelining

Notice that only the stream state process and the action process are unrolled in the sense that they must be applied multiple times. As we have discussed in Section 4.5.2, both the trace state and the stream state have a strong sequential requirement. When computing a stream state $\mathcal{S}_i$ or a trace state $\mathcal{T}_i$, $\mathcal{S}_{i-1}$ or $\mathcal{T}_{i-1}$ must be fully resolved to make any progress. Both the header preprocessing stage and the action code generation stage precompute any values of use that can be "pulled out" of this strong sequential requirement. In the case of the header preprocessing stage, the header of byte b_i is resolved speculatively. In other words, if the current byte should be interpreted as a header, which can only be determined after the stream state process, the header type lookup is already done. The same concept is applied to the action code generation stage, as each action code a_i can always be uniquely resolved by the stream state and $\mathcal{S}_{i-1}$ header type h_{i-1}, values that are already available when starting the action code generation stage.

For maximal performance, any computation that is not part of the sequential requirement should be computed in parallel outside of the circuit that must be duplicated (SSP and AP). The benefit of pulling out all computations is a reduction in the size of the duplicated circuits and consequently a reduction in resource utilization. More importantly, this also reduces the nesting depth of each process meaning the total delay of critical clock paths is reduced and the maximum operating frequency can be increased. We discuss this more in-depth in Section 6.2.

4.5.4 Trace Decoder — A critical reflection

We take a moment to reflect on the proposed trace decoding design as some lessons were learned while implementing the design that we were unaware of at the beginning.

4.5.4.1 Unbounded payload sizes

Throughout the previous chapters, we emphasized multiple times that an ETM packet may be of unbounded size. However, this is true only if the ETM capabilities are unknown. This is not obvious at first glance based on the header types. From the specification, all packets that may be unbounded are:

— Trace_Info

— Cycle_Count F1

— Commit

— Cancel_F1

— Conditional_Result_F1

— Conditional_Instruction_F1

— Q_packet

Unbounded payload sizes only exist if the ETM supports speculative execution tracing, data tracing (with synchronization algorithm), or Q packets.

For tracing speculative execution, an ETM has a maximum possible speculation depth that the hardware supports, which, in practice limits the size of speculation resolution packets `Commit` , `Cancel_F1` and `Cycle_Count_F1` . The `Cycle_Count_F1` packet is only affected because it has an internal `Commit` packet. In short, the specification itself does not put an upper bound on the payload size, but the hardware most assuredly does.

The packets may contains key values that are of unbounded size. The reason for this is (omitting some details) the resynchronization algorithm for the separate instruction and data trace streams. Resynchronization is also required when the data trace entry needs to be associated with the instruction in the program image that caused the data trace entry in the stream. The resynchronization follows a specific algorithm matching the store and load instructions (P1 elements) to their P0 elements [26]. The algorithm relies on both the synchronization packets (`Numbered Data Sync Mark` for example) and *keys* for both P0 and P1 elements. A P0 is a parent of a P1 element if they have the same key. A similar key-based algorithm is used to associate conditional instructions with their result APSR flags, if tracing of these flags is supported by the tracing hardware. Usually, it is the responsibility of the trace analyzer, or in our case the trace decoder, to increment the key value for each P0 and P1 element. However, the trace `Trace Info` packet will send out the explicit key value for the next P0 element at the beginning of the trace. Once again, this is not bounded by the ETM specification, but it is finite and predetermined by the hardware. The same thing applies to both the condition packets mentioned above.

For both the unbounded key packets and speculation resolution packets, the size of the packet will remain small, as the payload will carry an integer value. So if the max key or max speculation depth is less than 32, the payload size will be 1 byte.

The only true exception to this is the Q element packet. Q elements provide a way to reduce trace bandwidth by compressing `Atom packets` . An arbitrary number of P0 elements can be encoded into a Q element, sacrificing precise flow information in order to reduce bandwidth. The Q packet ends with an unbounded continuous packet indicating the number of P0 instructions that are implied by the Q element. We cannot put a strict upper bound on this value, even in the context of the system. We do expect the size to be reasonably small for the same reasons as with speculation resolution packets, as it holds an integer value. In practice, we do not expect Q elements to be used with a hardware trace decoder, as full control flow cannot be reconstructed if Q elements are enabled. A conservative approach is to disallow generating Q elements to ensure an upper bound of packet sizes or set an upper bound based on empirical evidence.

In summary, the specification allows for unbounded payload size, but realistically a tight upper bound can be set comfortably; it is hard to imagine the max speculation depth or max key size to go far beyond a few bytes. We did not realize this until we gained a deeper understanding of the protocol and the unbounded packet sizes are somewhat misleading.

On top of this, for both the US+ and the ThunderX, data tracing is not supported, the maximum speculation depth of the ETMs are 0, and the tracing of conditional result flags is not supported. This means no form of speculation resolution packets or key value packets will ever be sent out on the trace stream in the first place. There is no requirement to handle unbounded packet sizes for either of the hardware targets we have, only if we wish to decode the trace of an arbitrary trace source, and even then the maximum payload size will be reasonably small. We can no longer in good faith use the unbounded packet sizes as an argument in favor of a byte-wise decoder vs a buffering-based and packet-wise decoder.

Table 4.5: Hardware values that must be known for trace decoding. Column M marks which values are mandatory and must be encoded into the trace decoding implementation to work properly. Optional values affect the trace stream but are not required in our implementation to decode the trace. The C column stands for configurable, meaning these registers can be set before a tracing session. In this case, we need to make sure the decoder and the registers agree on the set the values

Value	Register	Packets effected	M	C
VMIDSIZE	TRCIDR2.VMIDSIZE	All context packets, or our internal VMID packet	✓	✗
CIDSIZE	TRCIDR2.CIDSIZE	All context packets, or our internal CID packet	✓	✗
MAX_SPEC_DEPTH	TRCIDR8.MAXSPEC	Unbounded speculation resolution packets	✗	✗
P0_KEY_MAX	TRCIDR9.NUMP0KEY	Only the Trace info tpacket	✗	✗
COND_KEY_MAX	TRCIDR12.NUMCONDKEY & TRCIDR12.NUMCONDSPEC	Unbounded conditional packets	✗	✗
CC_THRESHHOLD	TRCCTLR.THRESHOLD	No effect on packet size, instead cycle estimations	✗	✓
ETM IDs	TRCTRACEIDR.TRACEID	Trace source IDs	✓	✓
PMU event IDs	TRCEVENTCTRL0R & TRCEVENTCTRL1R	Event packet	✗	✓

4.5.4.2 Device parameters of the trace decoder

Similarly deceptive, there are fixed-size packets that allow for different sizes based on the ETM hardware configuration, meaning the size of a packet cannot be determined a priori without the context of the hardware the trace unit is running on. For example, Context packet we have seen in Figure 4.15b has 1 byte of VMID data in its payload. However, this depends on the ETM register TRCIDR2 that stores the VMIDSIZE value, where the VMIDSIZE can be between 0-32 bits. This makes the portability of the trace decoder more involved than simply copying the Intellectual Property (IP) from one system to another. The trace decoder cannot determine this size on the fly, as there are no continuation bits and no information regarding the packet size is encoded into either the header or the information bytes. For convenience, we list all the values that must be known to the decoder before deploying it to the system as far as we are aware in Table 4.5. These values include the necessary information for defining an upper bound for packet sizes (without Q packets).

4.5.4.3 Practical performance implications

A lot of effort has gone into our design to make sure we are able to handle 4 bytes per cycle. This is arguably overkill, as it is unlikely for a trace source to produce enough data that will reach this worst-case scenario. Ideally, before we began designing the decoder, a rigorous test suite would be established and the decoder would have been designed based on more realistic requirements. Unfortunately, it is non-trivial to design a Coresight trace bandwidth experiment, and it is unclear what type of binary in practice will produce the largest amounts of trace bandwidth and a lack of time led to us designing for the worst-case scenario[3]. The ETM configuration space is also very large and we expect the configurations to have a significant impact on trace bandwidth. We discuss this more in the context of future experiments in Chapter 8.

Nevertheless, we argue that this decoding design has little to no downsides when compared to other implementations. We believe a byte-wise decoder is simpler than a packet-wise decoder, both on a conceptual level and for programming the trace decoder. Our (partial) implementation of the ETM protocol is around ∼2K LOC as opposed to ∼3K LOC of our predecessor packet-based PTM parser, which has far fewer packet requirements and less state[4].

Our design is also adaptable to real-world bandwidth requirements with little to no change to the algorithm. This may be beneficial if PL resources are scarce. Both the unrolling factor and clock frequency are meta-parameters of our implementation. The unrolling factor and clock frequency are inversely correlated and can be chosen based on the use case. As an

[3] A judgment call was made that the implementation for worst-case throughput requirements is not much more difficult than an implementation for less throughput.

[4] We stress that LOC is not the determining factor in how complex a piece of code is, sometimes even the opposite.

example, we could trade off throughput by reducing the unrolling factor and optionally adding a small buffer in front of the trace decoder to absorb trace bursts — this may already be enough for most real-world use cases. On the other hand, if the trace data is used for verification, where dropping trace data can be catastrophic, it could be beneficial to guarantee no trace data is dropped, and our design with unroll factor four provides this.

4.5.4.4 Modularity & alternative design

Our trace decoder consumes the trace stream and extracts semantics received on the received data. We take the perspective from applying PGO-based compiler optimizations. This may not cover all use cases and discard otherwise useful information on the trace stream that was not considered. On top of this, whenever the trace analysis has different requirements, the trace decoder will potentially have to be partially reimplemented. Zeinolabedin et al. use the decoder only to transform the byte stream into a packet stream [89]. We offer an alternative design of how our current trace decoder could be modified to generate only packets without consuming the ETM specification that is built on top of the stream state processor and unrolling in Figure 4.21 and relies on an upper bound of packet sizes.

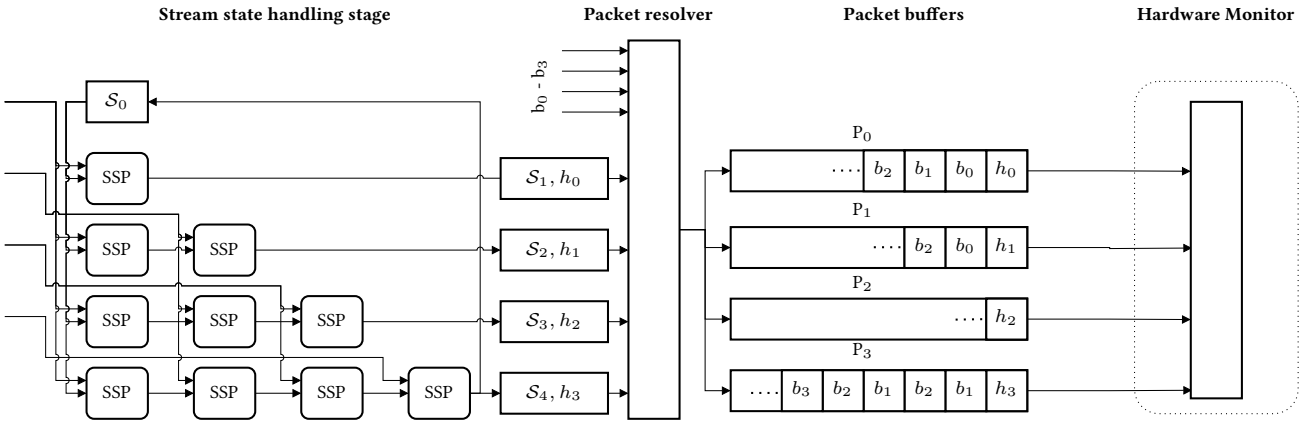


Figure 4.21: Alternative decoder that only reconstructs packets into buffers.

This design only uses the stream state, passing the stream state alongside the trace bytes to a packet resolver, which can push the bytes to packet buffers P_0 to P_3. Packet boundaries can always be resolved based only on our stream state[5]. A monitor that relies on full trace packets can read the buffers and parse each packet individually. This design achieves a level of modularity that is otherwise hard to achieve with the Coresight specification. Of course, the monitor would be now responsible for handling everything regarding the trace state. Also, the packet buffer sizes can be made generic to support a specific device based on the maximum speculation depth and key sizes.

[5]We would also need an extra step to distinguish internal packet types from packet types that are part of the specification.

Chapter 5

Coresight on ThunderX

Originally the goal of this project was to produce a working Coresight system running on a Cavium ThunderX as part of the hybrid CPU/FPGA Enzian platform [38]. The ThunderX-1 machine is a server-class CPU with 48 ARMv8 cores running at a maximum of 2.5 GHz. Enzian couples the ThunderX with a Xilinx Virtex Ultrascale+ XCVU9P-3 FPGA through the high-throughput and low latency Enzian Coherency Interconnect (ECI) [72].

In this section, we start by discussing the unique Coresight architecture of ThunderX. Unfortunately, as of writing this thesis, we were unable to achieve an up-and-running Coresight system. The propagation of the trace data is custom for ThunderX does not follow the standard ATB interface. Our running hypothesis is that trace data is lost in this custom trace bus layer. We are still unsure how to solve this issue, as we have a lack of documentation on the inner working of the custom trace bus. We offer instead a comprehensive guide to our current understanding of the architecture and outline the key learnings that were made throughout this work that might kickstart future attempts.

5.1 ThunderX Coresight Architecture

The ThunderX implements the Coresight infrastructure but does not follow the standard model as we covered in Chapter 3, the full Coresight topology is visualized in Figure 5.1 to the best of our knowledge. The figure shows one cluster of 12 cores, which is present four times on the ThunderX. We also recognize common Coresight components, namely ETMs, CTIs, PMUs, and debug components. However, the topology and link components are very different from the examples we have seen so far in Figures 3.1 and 4.2 and the ATB is replaced by the custom Debug Bus Transmitters (DTXs).

Most prominently, the ThunderX foregoes most link and sink devices one typically expects from a Coresight implementation and replaces these with an On Chip Logic Analyzer (OCLA). The OCLA is not only applicable to Coresight but may be used for other purposes as well. In the context of tracing, however, the idea behind the OCLA is to either forfeit the requirement of an off-chip analyzer entirely and perform analysis on-chip, or at least partially unburden the off-chip analyzer. This can reduce, or even eliminate, the need to ship a high-bandwidth stream of data off-chip and removes strain from the chip's surroundings, that being either network bandwidth, Peripheral Component Interconnect Express (PCIe), or in the case of Enzian the ECI.

To be more specific, there is no longer a TPIU or any TMCs and the OCLA subsumes the functionality of both. OCLA can emulate all TMC configurations as it has an in-block SDRAM storage that can hold trace data. It further provides the ability to overflow into both the L2 cache and system memory, similar to an ETR. The OCLA has two parallel FSM that can be programmed in tandem to support handling of complex expressions. How this may benefit Coresight trace analysis is left for future work. A likely first step would be to use the FSM to format the trace data into the same frames used by the TPIU. This would allow our current system to be integrated quickly on Enzian. However, the OCLA gives us the ability to explore other options since it is fully customizable.

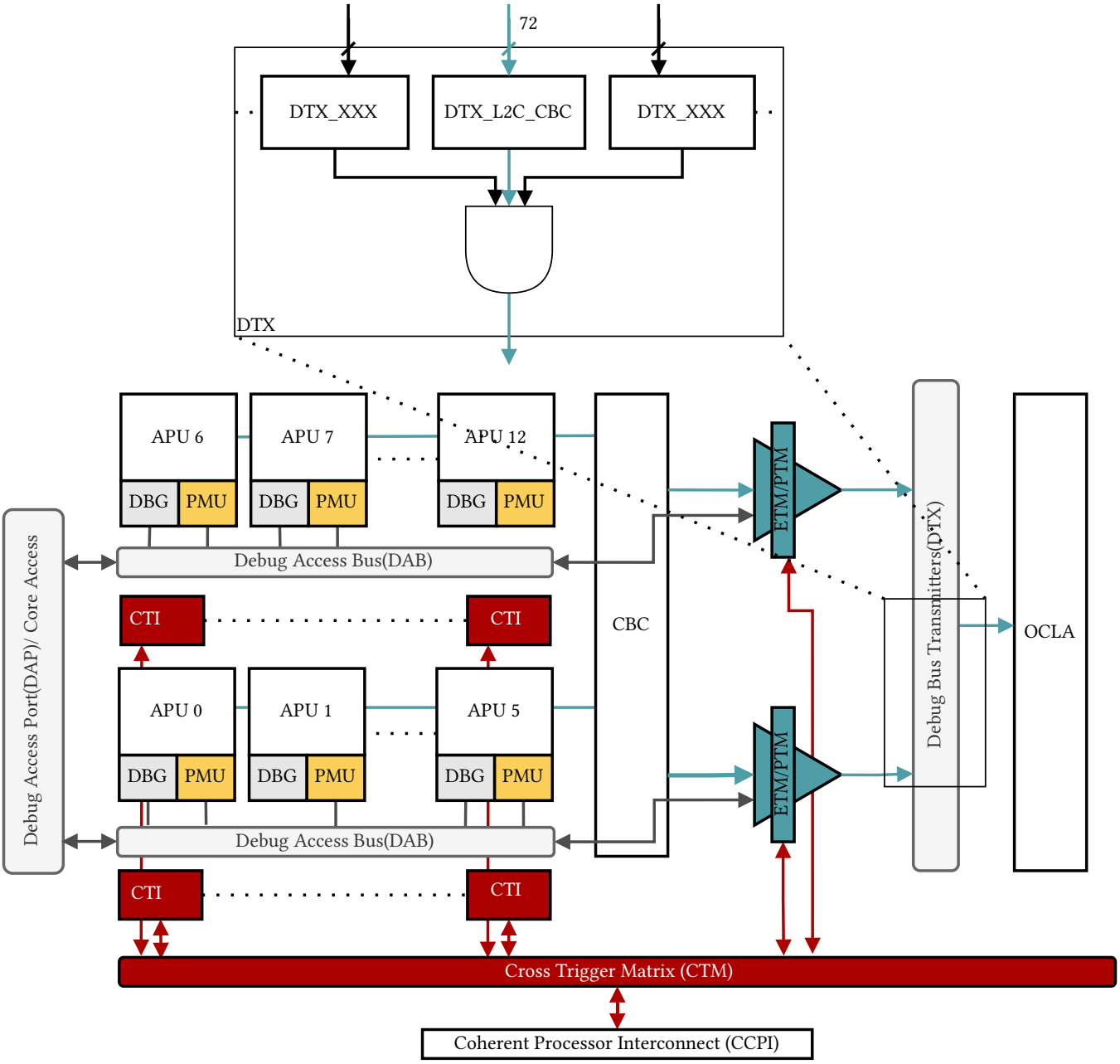


Figure 5.1: Coresight architecture for a single cluster on the ThunderX, adapted from experience, internal TRM and internal documentation from kernel module repository thunderx_trace.

The Coresight devices that remain identical to typical architectures are the CTIs, attached debug cores, PMUs and ETMs, specifically ETMv4.0s. However, while each processor has a PMU, CTI and debug core, the ETMs are shared by six cores, and in total, we have eight ETMs. An OCLA is shared by two ETMs, with four OCLAs in total. There is an additional fifth OCLA on the ThunderX, but this OCLA is not connected to any of the ETMs and is of no use for trace capturing. A group of 12 APUs forms a *cluster*, where each cluster has two ETMs and one OCLA. An ETM can only trace a processor that is in its cluster, restricting the possible tracing configurations — There is no way for a trace session to capture more than two cores per cluster at the same time.

The supported ETM configurations are shown in Section 3.2. A key configuration of both the work done by us and by Schmid [74] rely on, namely branch broadcasting, is not supported on the ThunderX. We show the effects of trace data without

branch broadcasting in Figures 4.10 and 4.11 and show an example of control flow reconstruction without branch broadcasting in Figure 8.1. To check if branch broadcasting is available the ETM register `TRCIDR0` must be read [26].

A good place to start and verify the topology is the device detection script we provide in our internal CSAL repository `csscan.py` . We modified the script (Listing 5.1) to work on the ThunderX as it does not exactly follow the Coresight specification and does not work out the gate.

```
python3 csscan.py -v -thunderx --status --all-status   0x87A000000000
```

Listing 5.1: CSAL Topology scan tool for Thunderx.

The address `0x87A000000000` passed to the script as a parameter is the base address of the Coresight ROM table. This also will dump all the `idr` registers and inform the user of all device capabilities and device affinities.

The way in which the trace data propagates through the system is custom for the ThunderX processor and we suspect this is where the problem currently lies. The ThunderX has a set of DTXs that drive data from different processor regions distributed throughout the system to the OCLA. Each region has its own DTX. For example, there are two NIC DTXs. For trace capturing, the OCLA expects the trace data to be fed over the `L2C_CBC_DTX` , of which there are four in the entire system — one per cluster or one per OCLA. The path the trace data takes in our current understanding is:

(1) The ETMs chooses which processor to trace, confined to APUs in the cluster, and produces trace data.
(2) The data from the ETM is driven to the `L2C_CBC_DTX` .
(3) The `L2C_CBC_DTX` feeds 72-bits of data to the OCLA.

Step (3) is documented in our internal TRM and we have a working implementation in an internal kernel module repository. Step (1) is very well documented by the Coresight specification [26], except for the way in which a trace source chooses which processor to trace. Only step (2) remains unclear, we found no details or documentation regarding these steps.

5.2 Starting a Tracing Session

In this section, we go through the necessary steps to start tracing the execution of an APU. Cavium, now Marvell, has provided a patch to the ATF and a kernel module implementation for trace capture. The ATF patch is required to access the OCLA registers to program the OCLA. Without the patch reading or writing to these registers causes an abort. The kernel module instantiates a character device driver from which trace data can be continuously read. The device driver configures the ETM to capture trace data, programs the DTX to drive the data to the OCLA and programs the OCLA FSMs. The device driver is available in an internal repository. Currently, no trace data is ever observed. We discuss each step individually and highlight the issues in order to narrow down what may cause the issue.

Every DTX has two programmable control registers and one read-only data register. The data register can be used to read the raw data coming into the DTX before any internal masking or selecting is applied to the data. The two control registers are `ENA` and `SEL` . All the registers, including raw registers, are split into low and high registers, each with 36 bits, and together form the complete 72-bit connection. The `ENA` register is used to mask the 36-bit values, the `SEL` register steers which bits should be driven to the low and high bits respectively. Additionally, there are two more registers, `CTL` and `BCST_RSP` that configure the operation mode and enable the DTX. The `CTL` register is used for different DTX modes and diagnostics.

We go through all failed attempts to try and detect the problem. To test if any raw data is observed on the DTX we run a kernel thread that polls the raw registers, which are the only observable points on the trace data path (Figure 5.2). Each test runs a workload on every one of the 48 cores, in case the ETM does not choose the APU we expect it to. All of the following attempts were run with this workload and polling thread.

We further were able to confirm that both the `L2C_CBC_DTX` and the OCLA work. To test the DTX, we can set the `CTL` register to `ECHOEN` . This drives the value of the mask value in `ENA` into the DTX instead of the bus data. The data can then

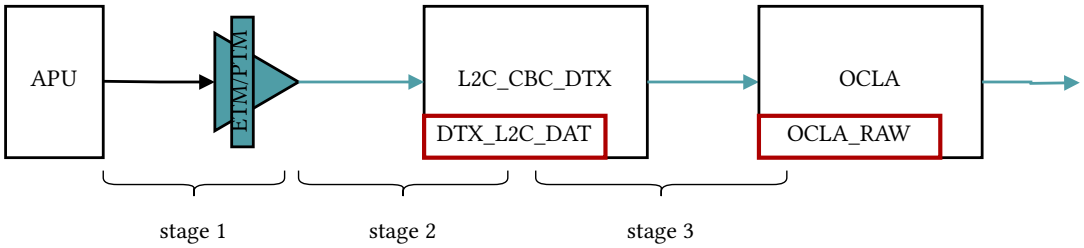


Figure 5.2: Path of trace data from the ETM to the OCLA. The only observable points along this path are the two registers `DTX_L2C_DAT` and `OCLA_RAW`, splitting the path into stages 1-3.

be observed in raw OCLA raw registers, and is propogated through the OCLA as expected.

Another way to test the OCLA is to zero out the "matcher compare value registers" of the OCLA. The OCLA will take in any data it observes in the raw data registers. If no data is passed to the OCLA in the first place, it will read a constant stream of zeros. Essentially, this step removes the check for valid data received from the DTX.

One important note, when the module is inserted into the kernel without modifications as we have received from Cavium/Marvell, the kernel will crash with `Internal error: synchronous external abort: 96000250 [1] SMP`. Decoding the error message did not yield any useful information, it is just a data abort. The cause of the abort is during the DTX setup, namely *the write to any high DTX register*. The low register works fine. We checked the ATF code, but all the DTX registers seem to be properly registered. Keep in mind that if only the low register is configured we will miss up to half the data sent over the DTX, but there should still be observable data in the low registers. This hints towards some misconfiguration for the DTX, but the cause of this is still unclear.

We highlight the approaches and experiments that were undertaken to run a tracing session on the ThunderX:

- We double and triple check the ETM configuration. Everything except for the `TRCPROCSELR` follows the Coresight specification *exactly* and has worked for the ETMs on the US+ device. All registers show the same value if we read them back after writing, indicating the write to the registers is completed successfully and the ETMs are powered. This also includes making sure the OS lock is correctly unlocked in `TRCOSLAR` and making sure `TRCSTATR` shows the ETM as idle when writing to registers. If it is not idle, the specification states the behavior is undefined. We also dumped the entire 4KB trace block that contains all ETM registers and went through them, checked the addresses and if they contain correct values.
- The ETM is confirmed to be powered on, and this value can be checked in the `TRCPDSR` registers. We also checked the powered state of the debug register `DBGEDPRSR`.
- Fuzzing all values of `TRCPROCSELR`. As a reminder, the `TRCPROCSELR` tells the ETM which APU it should begin tracing. The `TRCPROCSELR` follows a custom selection scheme on the ThunderX and there are a few comments in the kernel module that mentions a workaround solution for something that does not work as intended. We tried all possible values of the first three bits and see if any trace data is observed (the rest of the bits are reserved).
- Checking all DTX registers — the ThunderX has a few dozen DTXs that can drive data to the OCLA. Usually, all of these are disabled by the kernel module before we start a tracing session. We extended the polling kernel thread to check the raw registers for every DTX and make sure the trace data is not misplaced.
- We ran all the CSAL topology detection tests. Unfortunately, as the ThunderX does not use a standard topology so this can only give limited information, in our case, it only registers a connection between the CTIs.
- Similarly, we built the kernel with all Coresight configurations turned on. Naturally, this does nothing to help us as this also expects a standard Coresight topology [7].
- We found evidence to suggest that the Coresight tracing system may have trouble when an APU starts idling [12]. To avoid this, we configured the Enzian grub file to turn off idling with `cpuidle.off=1`.

— The patched ATF contains a few differences to the Enzian ATF. We applied the patch (a few LOC) directly on a branch in the Enzian ATF. We scanned the diff between the provided ThunderX ATF and Enzian ATF. There are only very minor differences between the two and this most likely is not what is causing the problem.

During development and experience gained during the work with Coresight infrastructure on the US+ we had additional ideas that are worth testing. The ETM has integration mode control registers. In the TRM they are marked reserved or read-only, so we initially discarded them and assumed that they were not supported. However, it seems running the topology detection algorithm provided by the CSAL [2] does not crash when it enables integration testing mode. The integration mode is implementation-defined, so there is no guarantee it will work, but it might be worth testing the exact code from the CSAL from inside the kernel module while polling registers. If data can be seen in the raw registers, we have narrowed down the issue to an ETM configuration problem.

During the design of the trace decoder, we found there is an entire chapter that gives a good overview of access permissions (chapter 7.2 [26]) for the ETM for different power states in and an overview of power domain models (chapter 3.3 [26]) the ETMv4.0 specification that we were unaware of when attempting to get the Coresight system on the ThunderX to work. The diagram shows that the power domain of the ETM registers are split between core power domains and debug power domains, so ETM registers could be powered and programmable (which we checked), but might still not drive trace data to the rest of the system. These chapters could be of interest if this is a power domain problem. The TRM for the ThunderX states, however, that there is only one debug power domain, and that is the always-on domain.

For future work on the ThunderX, we recommend reproducing all the steps we outlined and seeing if the same behavior is observed. Subsequently, continue debugging from there. For debugging Coresight configurations on the US+ it was very helpful to have access to all device registers over JTAG as well, which we recommend setting up.

Concluding our ThunderX chapter, we have ruled out problems on OCLA, DTX and stages 2 and 3 from Figure 5.2. We believe that there is either something misconfigured or kaputt in the way the ETM chooses the APU, the APU-ETM link, or the connection between the ETM and DTX. It is possible that the ETM itself is not set up correctly, however, this seems to be the least likely scenario and the exact same configuration is shown to work on other devices. We include the ETM misconfiguration as a possible scenario since there is no experiment that we were able to perform to unequivocally rule out the possibility.

Chapter 6

Evaluation

The key evaluation metrics we consider in this section are the worst-case throughput our decoder can handle compared to prior work (Section 6.3), the resource utilization of the decoder on the US+ (Section 6.4) and verifying the correctness of the decoding process (Section 6.4). We further discuss floor planning aspects to meet timing constraints, and what the bottlenecks in the decoding pipeline are that limit the maximum operating frequency, and a breakdown of the critical path (Section 6.2).

6.1 Resource Utilization

In this section, we report on the device utilization of our implementation on the US+ for each component in our system-wide pipeline. We omit parts of the pipeline that require less than 0.1% of the overall FPGA resources, like the frame generator and trace stream demultiplexer. We add the L1 and L2 decoder from related work [89]. Here, the L1 decoder is equivalent to all parts in our system between the TPIU and the trace decoders. The L2 decoder is equivalent to our trace decoder. We emphasize that the L1 and L2 decoder utilization report comes from the implementation on a different device than the US+ and that direct comparison across devices cannot be made. Nevertheless, we include them to highlight a few points and provide a ballpark comparison. Furthermore, our decoder does not implement the entire specification, only around 60% of all packets (Table 4.4). We discuss the L1 and L2 implementations in the related work Chapter 7.

To make sure Vivado would not optimize away any unwired outputs we connected every value in the trace state $\mathcal{T}_1 \dots \mathcal{T}_4$ and each output wire $\mathcal{O}_0 \dots \mathcal{O}_3$ to an external submodule.

Table 6.1: Resource utilization table, with added utilization report of L1 and L2 decoder from [89]. The percentage is calculated based on total FPGA resources. We emphasize that the L1 and L2 decoders are on a different device than our implementation, device id in Table 6.2.

Util	L1 Decoder	Frame Decoder	L2 Decoder	Unrolled Decoder	Single-byte decoder
LUT	225(0.48%)	195(.17%)	3160(1%)	2899(2.48%)	273(0.23%)
Registers	378(0.41%)	203(.09%)	1006(1%)	2605(1.11%)	677(0.29%)
F7 Mux	0	0	0	42(0.07%)	16(0.03%)
F8 Mux	0	0	0	0	2(0.01%)
Block Ram	0	0	8(5%)	0	2(0.01%)

In Table 6.1 unsurprisingly shows the most expensive part of the system is the trace decoder. We compare also the unrolled decoder from Figure 4.20 to an implementation of the decoding function of a single byte. As a reminder, we expect the circuit for the single-byte decoder to be duplicated 10 times in the unrolled version. The resource utilization seems to accurately reflect this when it comes Lookup Tables (LUTs) with around a 10× increase between the single-byte and unrolled decoder. Surprisingly, this does not hold for the rest of the resources and the Vivado tool suite seems successful at optimizing

away the redundancies in the unrolled circuit.

To support a system with a trace decoder for each Cortex-A53 we would need four trace decoders, requiring around ∼10% of the FPGA (using the unrolled decoder) for the most dominant resource, that being the LUTs. Taking this into consideration, the rest of the components are negligible in terms of utilization.

Our unrolled decoder seems to be comparable in terms of resource usage to the L2 decoder. We refrain from commenting on this anymore than this due to the implementations being on different devices.

What we can highlight, however, is that our decoder does not require any block RAM when compared to the L2 decoder. This is a direct consequence of being able to always handle 4 bytes no matter what, instead of buffering data and processing the trace stream packet-wise. This may be important if we have multiple trace decoders on our system (including data trace decoders that require the same amount of block RAM). Putting four instances of both data trace decoders and instruction trace decoders into the bitstream will already use 40% of the block RAM on their Virtex xc6vcx75t-2ff784 FPGA device.

6.2 Operating Frequency & Timing Constraints

For the unrolled decoder we have introduced, the key to performance is passing the timing constraints. This is no easy task, as the unrolled decoder must essentially perform the entire decoding process four times in one cycle. If the the target clock frequency of the decoding process is 250MHz, and the total clock delay of the critical path must be under 4 ns. All discussions on timings assume an unroll factor of 4, which supports any trace data that can be produced on the US+.

To this effect, there are two things that we must minimize, the logic nesting levels and integer sizes of state values. Both of these will have a direct effect on the path levels in the critical path, increasing both net delay and logic delay.

Minimizing the logic nesting is done by pipelining the decoder into four stages we already covered in Section 4.5.3.4. For an initial naive and un-pipelined implementation, the total delay of the worst timing path is ∼7ns and has paths with 35-40 levels. Pipelining the decoder brings the maximum level in all paths down to 12 and the worst total timing delay down to ∼4ns. Both of these have optimized integer sizes already, which we discuss in the following section.

The main bottleneck in terms of timing is the stream state handling stage. Everything else is comfortably below the 4ns mark. More specifically, the critical timing path is the path that leads through the termination condition and checks for the ending of payloads four times.

6.2.1 Constraining Integer Sizes

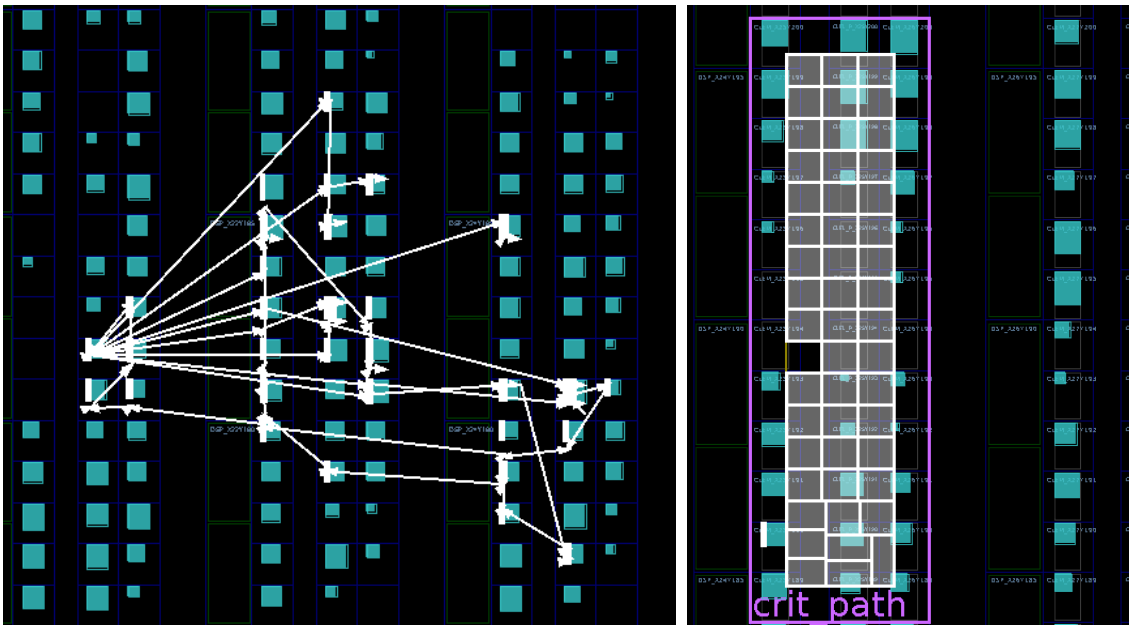
The integer sizes, or how many bits we used to represent these integers, used in the stream state for the indices and for the packet sizes have a significant impact on the total delay of timing paths. In the stream state handling stage, a timing path leads through four stream state processes, meaning there exists a path that traverses the termination condition logic four times. The termination condition of packets that are classified as FS or BC requires checking if the index of the current byte is equal to the max payload size. This equal check quickly becomes expensive the larger the possible integer size is. The larger the integer size, the more carry8 components are required to perform this check, causing critical paths to get longer (have higher levels), increasing both net delay and logic delay, and as a result the total delay. If we do not constrain our integer sizes, the total delay of the worst path in the stream state handling stage is ∼6ns and has 20 levels — far from our 4ns goal.

The key to eliminating this timing failure is constraining our integer sizes, but how can this be done if we have unbounded packet sizes? This is the major influencing point when deciding to distinguish between unbounded and bounded continuous packets. Each BC and FS packet is finite and we can determine an upper bound. For UC the payload index is not required for the termination condition, therefore not needed at all. Additionally, there is no unbounded packet type that requires the indices of the payload to determine the trace state action code, meaning we can completely disregard these packets when choosing an upper bound for the integer size, while still being able to accurately decode the trace stream for an unbounded payload. From all packet types, the max integer required is 11 for the `A-Sync` packet, any larger packet is either unbounded

and we d not require an upper bound, or is a composite packet broken down into smaller internal packets.

6.2.2 Floorplanning

The combined efforts of pipelining and constraining integer sizes bring us very close to our 4ns goal, but we remain off target by around 0.3ns for around 10 paths (depending on the implementation run). In reality, this is likely to only cause problems in very rare scenarios, but we attempt to avoid it nonetheless. All the timing violations still come from the paths that go through termination condition checks. We show in this section that with very minimal floorplanning intervention we can circumvent the timing issues.



(a) Critical paths without floorplanning. **(b)** Critical paths with floorplanning.

Figure 6.1: Solving timing violations with floorplanning

Figure 6.1 shows the critical paths before and after assigning them to a pblock. A Pblock is an area containing cells specifying the placement of the components used by the design on the FPGA. Doing this avoids all the timing violations, it just requires a few extra steps to achieve. The assignment to Pblocks may have to be repeated a few times as the move may affect other paths as well. The Pblock must be small enough to decrease the net delays, but large enough to fit in all the critical paths.

In conclusion, with pipelining, integer constraining and minimal floorplanning the total delay of any path on the decoder is brought under 4ns and our target is achieved.

6.3 Performance

In this section, we analyze and compare the throughputs of our trace decoder in Table 6.2. Trace decoders from prior work are included as comparisons. We specify the devices they were implemented on. More details on the implementation of the other trace decoders are discussed in Chapter 7. We add the device ID on which each decoder was evaluated, as the device could affect the maximum operating frequency and as such a direct comparison should be taken only for bytes/cycle. We add the PTM decoder by Schmid [74] even though it supports a different specification. The different specifications should not be directly compared without caution as they are completely different protocols. However, some of the basic building blocks, i.e. packet structure and even some packet types are similar enough to warrant a comparison.

Table 6.2: Comparing out trace decoder to prior work. The details of the implementations are discussed in Chapter 7. We use the worst-case scenario for bytes/cycle and throughput. We compare our decoder to the decoder proposed by Weiss et al. with one decoding window and our implementation with unroll factor four. B=requires buffering, U=supports unbounded packet sizes.

Decoder	Device	Specification	bytes/cycle	Max frequency	Throughput	B	U
unrolled_decoder	xczu5ev-sfvc784-2-i	ETMv4	4 bytes/cycle	250 MHz	1 GB/s	✗	✓
Zeinabolin et al.	xc6vcx75t-2ff784	ETMv4	1 byte/cycle	125MHz	125 MB/s	✓	✓
Weiss et al.	—	ETMv3.5/PTM	1 byte/cycle	>100MHz	> 100 MB/s	✓	✗
Schmid	xc7z04	PTM	1 byte/cycle	125Mhz	125MB/s	✓	✗

The throughput is evaluated theoretically based on the implementation and not empirically. The bytes per cycle are chosen for the worst-case scenario, meaning what is the sequence of packets that causes the least amount of throughput. All decoders are unable to achieve more than 1 byte per cycle in the worst case, as they have no form of parallelizing the decoding of multiple bytes at the same time. The decoder by Schmid and Weiss et al. struggle when receiving multiple subsequent HO address packets and the L2 decoder by Zeinabolin et al. struggles when receiving successive variable-sized packets. Our decoder can handle both these cases without a hit to throughput due to the combination of the fact that our decoder processes byte-wise and unrolls the byte stream. In summary, our decoder has 4× improvement in bytes per cycle and a 2× improvement in the maximum operating frequency, leading to a total of 8× improvement in trace bandwidth the decoder can support compared to all previous designs.

6.4 Correctness

Every submodule in our system is tested for correctness with a simulation test bench. The trace decoder, however, only has a test bench for the single-byte decoding function, and not the fully unrolled decoding function. For each submodule, we used data observed from the trace stream directly. This is an admittedly weak correctness test suite, and ideally, the entire pipeline from ETM to trace decoding output would be used as the unit under test and compared to existing software Coresight decoders. We leave this as future work due to lack of time and discuss the best way to approach this in Section 8.2.

Chapter 7

Related work

There exists a staggering amount of related work in the field of PGO optimizations, software profile collection strategies and tracing techniques. We covered many aspects of these works in Chapter 2. We focus our efforts in this section mostly on Coresight-specific or ARM-specific projects. We cover prior implementations of trace decoders in hardware for both the ETM and PTM specifications and compare them to our approach (Section 7.1). We continue by comparing our approach to a memory-monitoring system built on Coresight using only PMU events (Section 7.2), and we conclude this section by describing recent efforts to collect profiles on ARM processors for PGO (Section 7.3).

7.1 Hardware Trace Decoders

To the best of our knowledge, there are only three other hardware trace decoders and only two that support the ETM specification. We discuss how these implementations differ from ours and how our implementation has improved on the prior work in this field.

7.1.1 PTM Trace Decoders

Schmid introduces a runtime verification system that feeds Coresight trace stream from PTMs and an ITM to a runtime verifier that is implemented on an FPGA [74]. His work shares many of the same Coresight processing elements, including the TPIU-PL interface, a frame parser, a stream demultiplexer and trace decoder. His work is done on a Zynq-7000 device instead of the US+. The Zynq-7000 has a different Coresight subsystem implementation, namely, it uses PTMs instead of ETMs and has a simpler topology, similar to the one shown in Figure 4.3. His decoder is also not directly portable to the ThunderX, since the ThunderX requires an ETM protocol decoder.

From a design perspective, the main difference between our work and the work done by Schmid is that our implementation supports the newer Coresight specification, namely the ETMv4.0 specification.

The Zynq-7000 exhibits the same TPIU and PL interface through EMIO to the US+. While the implementation is functionally the same, Schmid adds a conversion from TPIU trace data to AXI stream before passing the trace data to his frame decoder. Furthermore, we decided not to reuse his code for frame decoding since it has a different input type, namely an AXI stream instead of raw TPIU data. He also has implemented his pipeline with synchronized buffers between frame decoding and trace decoding stages, and he adds additional timestamping metadata to frames before he decodes the frame that he needs for the later verification steps.

The interesting differences come when discussing the different trace decoding implementations. Unfortunately, his descriptions of the decoding process are very sparse — around 3K lines of VHDL code for trace decoding are described in less than a page in his thesis. The decoding process is also only described in what it does and not how it does the decoding. We try our best to convey the ideas of his work faithfully from the source code.

The PTM parser, as implemented by Schmid, uses a buffer to store all the stream data and processes it packet-by-packet. His implementation relies on the finite size of packets. As an example, a timestamping packet (same in the PTM and ETM specification) is a packet of type BC, meaning it can have a payload size between 1-7. When parsing a timestamping packet, the PTM parser first resolves the header type of the packet to be processed, then, if necessary, scans the entire buffer to resolve the packet size. This means it checks the MSB of every payload byte up to the maximum payload size and checks if the packet is ending. This way, the parser can determine ahead of time the size of the payload. Changes to the trace state can be applied in one go for an entire packet. Our implementation improves this and scales to an unbounded packet size, which is required for an ETM parser.

The throughput requirements for Schmid's PTM parser are the same as ours and require the processing of 4 bytes per cycle. A packet-based parser is not able to achieve this for successive single-byte packets. His solution for this problem is to add an optimization step that can parse multiple sequential `Atom` packets at the same time. This is easily parallelizable, as the `Atom` packets require no state handling, meaning there is no dependency between the packets as long as the ordering of the output is maintained. Also, in the PTM protocol `Atom` packets and address packets[1] are the only possible single-byte packets. And his implementation separates address parsing into an earlier pipeline stage that always occurs before the trace decoding.

However, his solution does not entirely eliminate the problem, even if it handles many practical scenarios. It is hypothetically possible for the PTM parser to overflow the buffer and not be able to keep up with 4 bytes per cycle. To overflow the buffer, the trace source would have to generate many successive two-byte (one payload byte and one header byte) packets in succession. This is possible, if we constantly see a mix of two-byte `Atom` packets[2] and other packet types. This is a realistic scenario, since all packet types may have payload sizes of one [23]. In this situation, the worst-case throughput of this design is 2 bytes per cycle.

Additionally, single-byte address packets are possible in the PTM protocol and we were unable to find any evidence that Schmid's implementation supports handling multiple sequential single-byte address packets the same way as the `Atom` packets. For a PTM parser, this is important, as enabling branch broadcasting will no longer generate *any* `Atom E` elements (only not executed `Atom` packets) and replace them with address packets containing the target address. In a scenario with many sequential short jumps, a source would send out non-stop single-byte address packets. A jump is short only if the first 7 to 5 bits of the currently held last seen address need to be changed to resolve the target address. It subsequently fits into the header byte [23]. The exact number of bits depends on the ISA. Taking this worst case, the address parser stage in his pipeline will become a bottleneck, only handling one byte per cycle.

In conclusion, our trace decoder, which works on the newer and more complex ETM protocol, solves the problem of parsing unbounded packet sizes and does not rely on buffering. We improve the reliability of the decoding process, as it is no longer possible to lose data in the parser. On top of all this, we increase the supported throughput of the decoding process by 2× from the 500MB/s throughput to 1GB/s, when considering the best-case scenario for Schmid's decoder. Being pedantic, the theoretical worst-case throughput is increased by 8× from 125MB/s to also 1Gb/s if we only see single-byte address packets.

7.1.2 ETM Trace Decoders

To the best of our knowledge the only other hardware implementations of Coresight trace decoders are introduced by Weiss et al. [84] that is also used in online trace analysis and verification systems [39, 42, 37] and Zeinolabedin et al. [89, 90], introducing both an instruction trace and data trace decoder. In these implementations, they refer to the frame decoder as an *L1 decoder* and the *L2 decoder* is analogous to our trace decoder. We stress that their L2 decoder only contains a subset of the functionality of our trace decoder, as the purpose of this decoding step is to extract complete Coresight packets and separate any semantic extraction into a different submodule, or a monitoring stage.

The trace decoder introduced by Weiss et al. exhibits a similar issue during the decoding process introduced by Schmid and can only process finite-sized packets. The decoder relies on resolving the size of a packet based on an observation win-

[1] Keep in mind that both the address packets and the atom packets have a different layout as compared to the ETM protocol.
[2] This is only possible in the PTM protocol *not* the ETM protocol.

dow, that would have to be reconfigured to fit the largest possible payload size. From our understanding, a packet can only be decoded if the entire packet is contained in an observation window. In the worst-case scenario of single-byte packets, one byte per cycle can be processed. Their technique to achieve higher throughputs is to have eight parallel windows, but we are not sure how the state is resolved in this scenario or how state dependency is resolved. We believe this parallelization technique only applies to their PTM decoder.

Zeinolabedin et al. improve on the L2 decoding algorithm by Weiss et al. by splitting up the processing of variable-sized packets and fixed-sized packets that can be processed simultaneously. Variable payloads are also processed sequentially, allowing for the decoding of unbounded packet sizes. Internally they use a *Control Core* that is somewhat similar to our stream state to keep track of current payload indices, crucially, without unrolling. This is reflected in the worst-case scenario for this decoder: receiving only variable-sized payload packets. In this case, they can only process one byte per cycle. With their maximum operating frequency of 125Mhz, this leads to a supported data rate of 125 MB/s [89].

Summarizing the related work of trace decoders, all previous work relies on buffering of trace data, increasing area requirements or block RAM requirements on the device, and the only other decoder that can support unbounded packets is the L2 decoder by Zeinolabedin et al. The unrolling of the trace stream allows processing of 4 bytes per cycle *with no exceptions*, which is not possible for any other algorithm and provides safety against dropping packets in worst-case packet streams.

7.2 Monitoring with ECTs and PMU Events

As an alternative approach to sending the entire trace data stream to the PL, Baryshnikov uses the Coresight subsystem, also on an US+, to monitor the number of memory accesses that are made from a processor [31]. For this purpose, he only uses the ETM as a memory event generator on the cross-triggering network. This event is then forwarded to the PL over the CTM and then over the PS to the PL trigger interface (see Figure 4.2). He uses the `L2_CACHE_REFILL` event that we mentioned as well in Section 3.5 to mark a cache miss, and therefore a memory access. The ETM is only used for forwarding triggers to the CTI. This a lightweight approach to memory monitoring that does not use the instruction trace at all (compared to a full trace decoding and analysis process). With this approach, however, it is no longer possible to reassoicate the memory access event with the basic block that triggered this memory event.

7.3 Collecting Profiles on ARM Processors

We mentioned that most work in profile collection for optimization purposes uses LBR that is only available on Intel processors. Some work has gone into emulating capabilities of LBR on ARM processors. For example, AutoFDO added support for the collection of ETM trace data [3] that relies on simpleperf [11] and the Linux Coresight driver module. This data collection only uses the ETB to trace data, meaning it may only be able to trace very short sessions before becoming full. If an ETR is available, it will also support writing to larger regions of system memory, providing the ability for longer tracing sessions or higher sampling rates. This, however, can cause overhead when memory bus contention is high. The trace data is then run through the Open source CoreSight Decoding Library (openCSD) with the program and kernel image before branch frequencies are injected to the IR at higher abstraction levels, creating actionable feedback for the compiler.

On the other hand, some of the authors of AutoFDO further moved on to create an architecture-neutral and lightweight instrumentation technique specifically to support LBR-like profile collection for the growing number of AMD and ARM processors [62]. On ARM, this PMU is part of Coresight, but their system does not directly interact with the Coresight features such as ETMs or ETBs. Instead, they use the debug capabilities of the processer, which is part of Coresight, to set breakpoints at instructions that modify the control flow and the target address cannot be resolved statically. The breakpoint handler is modified to hold a buffer of the last branches, emulating an LBR.

In conclusion, it remains unclear how Coresight fits into the larger context of AutoFDO, where AutoFDO is representative of the most common cases of PGO applications.

Chapter 8

Future Work

In this section, we discuss extending the system to support more features of Coresight (Section 8.1), further we discuss some planned experiments that were not performed due to time constraints (Section 8.2). We conclude with further research opportunities and practical applications of online and on-chip tracing and analysis systems (Section 8.3). Section 8.3 should be considered more speculative and open-ended than the rest of this section.

8.1 Hardware Trace Decoder System Extensions

We begin discussing possible direct extensions to the current system to add more features and support trace analyzers on different hardware configurations.

8.1.1 Coresight on ThunderX

As of writing this thesis, we are unable to receive any trace data in the DTX from the ETMs on the ThunderX. We hope insights gained throughout this work aid a future attempt. A working Coresight subsystem would make the platform more interesting for future projects we discuss here. We also believe that the ThunderX problems are not due to a lack of knowledge of the Coresight system, but missing documentation of the inner workings of the debug system. It may be worth considering tracking someone down who had a working tracing session running on the ThunderX, as the existence of internal documentation we have received suggests that tracing sessions have been successfully performed in the past.

The ThunderX, classified as a server-grade processor, presents a more applicable environment for collecting profiling data for PGO than the processors on mobile SoCs, since PGO is mostly applied to server workloads. Most research efforts have gone into Intel-specific profile collection techniques and the space remains open when it comes to PGO on ARM.

The OCLA on the ThunderX has two programmable FSMs that have access to the trace data before the ThunderX drives trace data away from the PS. An open question is how much of the decoding logic can be pushed to these FSMs to minimize trace volume or aid the decoding process. For an initial trace analyzer design on the ThunderX, the OCLAs should be used to format the trace data into frames so the data can be demultiplexed on the PL. This is because there are no Coresight components on the ThunderX that are capable of wrapping the data into frames, as the ThunderX has no TPIUs or TMCs. From this point, it is worth exploring different use cases for the OCLAs.

8.1.2 Completing the AXI-DMA

We planned to write trace data that has gone through the processing stages on the PL to the PS memory via AXI DMA. The US+ has a PL specific memory that can be accessed by the PS through a high-performance AXI interface. This has been implemented in the block design, and verified to work with the simple test suite including drivers provided Xilinx [1]. However, for now, this only works when running the DMA test as a standalone application and not when booting into the Linux kernel. Here the next step would be to either write a simple AXI DMA driver, or build the kernel with the existing drivers provided

by Xilinx [6]. Once this is configured, the experiments outlined in Section 8.2 can be performed as an end-to-end system. In hindsight, this should have been implemented very early on, providing more practical and concrete requirements for a Coresight decoding implementation. This ties into the discussion in Section 4.5.4.

8.1.3 Tracing with the Program Image

The ETM instruction trace protocol is designed for decoding under the assumption that the trace analyzer *has access to the program image*. This means address packets are sent out explicitly only for indirect branches. Only if branch broadcasting is supported are address packets sent out over the trace stream for direct branches. *The ETMs on the ThunderX do not support branch broadcasting!* It is therefore not possible to reconstruct the execution flow from the trace without the program image on the ThunderX. Also, even if we have branch broadcasting, if we wish to have the source address, address a_0 of a jump in the form: "jump from address a_0 to address a_1", we need the program image.

The only way to properly reconstruct the execution path is to use the program image of the program that generated the trace stream. To allow for trace analysis for arbitrary ETM configurations, we propose an additional processing step in the pipeline that has access to a copy of the program image (or multiple images) currently running and, feeds the images alongside decoded trace data to a trace execution reconstructor that can regenerate the full execution trace, even of direct branches, see Figure 8.1.

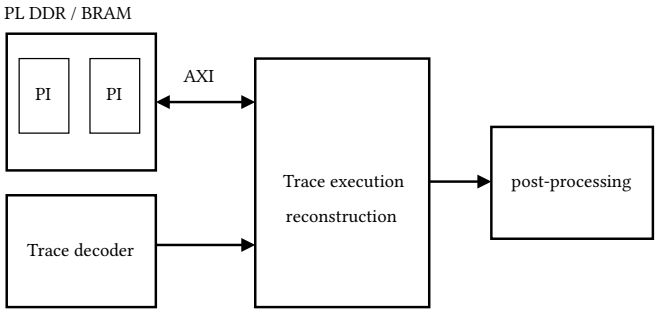


Figure 8.1: Handling profile trace analysis without branch-broadcasting, where PI is the program image.

It may represent quite a challenge to keep up with the trace decoder in terms of throughput. This reconstructor would have to essentially "chase down" addresses on each indirect branch, walk through each basic block in the code based on the received `Atom` packets, and replace `Atom` packets with explicit addresses — while potentially handling multiple packets in each cycle. In practice, we imagine this to look something like this:

- The program image can be compressed in a preprocessing stage, as the only things we need from program images are P0 instructions and the offset address of these instructions in the binary. All other instructions can be discarded. This preprocessing step would also require rewriting the jump addresses to reflect the compressed image.
- The compressed program images are then made available to the PL in the PL memory region connected with some form of AXI interface such that the trace reconstructor is able to request data at a given program counter offset. This requires a form of address translation between the compressed image and uncompressed image.

  Alternatively, the program images can be stored in BRAM if enough space is available.
- The reconstructor only handles two packet types received by the trace decoder: `Atom` packets and `Address` packets. An address packet requires the reconstructor to fetch a chunk of the program image based on this address packet. Each `Atom` packet is then used to walk through the compressed program image and resolve target and source addresses for jumps.

This remains solely an idea, and the requirements may change based on later challenges. It remains unclear whether this can be implemented in a way that keeps up with decoding throughput, or bottlenecks the system.

8.1.4 Supporting Speculation Resolution.

As a reminder, if speculative tracing is enabled and supported by the hardware, the trace protocol sends out `Atom` packets that have been speculatively executed. They remain speculative until either a `Cancel` , `Commit` or `Mispredict` packet is sent out. The number of speculatively executed instructions depends on the max speculation depth that is held in an ETM register.

As of now, this feature is not supported by our decoder, nor is it implemented for the US+ or ThunderX. However, this feature could be built on top of our current decoder. To do so, it makes sense to add a submodule after the trace decoder in the system-wide pipeline (not in the trace decoding pipeline, although this could be done as well). In a nutshell, the speculation resolver should be able to buffer `max_spec_depth` number of `Atom` packets, and hold these packets. This submodule has to be inserted after the trace decoding process, otherwise, the semantics of the byte stream cannot be interpreted.

The proposed speculation resolver keeps a buffer of trace decoding output that extracts meaning from `Atom` packets. Only once a speculation resolution packet is received and decoded, is the output produced by the held `Atom` element released from the speculation resolver. We sketch our idea in Figure 8.2.

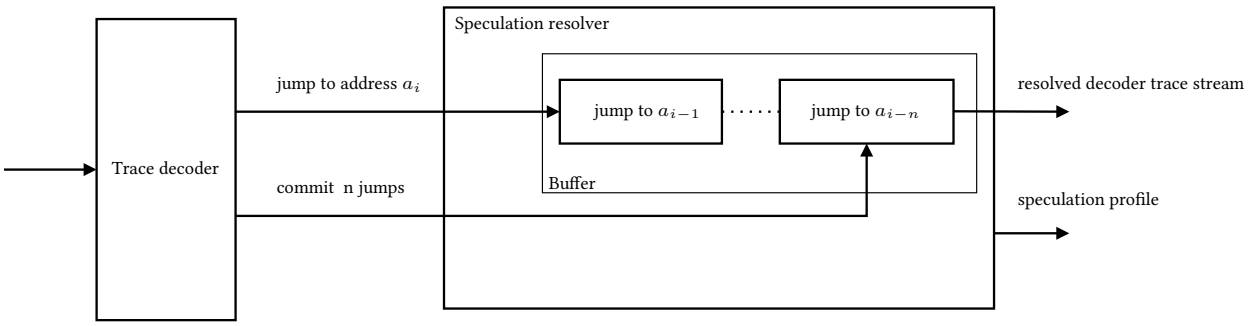


Figure 8.2: Additonal stage for speculation resolution where $n =$ `max_spec_depth` .

This example is somewhat simplified and does not account for the fact that up to `24` `Atom` elements can be sent by a single packet, meaning we could get `96` `Atom` elements per cycle, similarly for commit packets.

8.1.5 Supporting Data Tracing

As discussed, the ETM specification also supports data tracing in addition to instruction tracing. While this is not implemented on the US+ or ThunderX, we believe data tracing can enable even more interesting optimizations than just instruction tracing, discussed some more in Section 8.3. Extending the system to support data trace requires first a data trace decoder similar to the instruction trace decoder. The data trace protocol is much simpler than the instruction trace protocol and we can reuse parts of the design for the data trace decoder, meaning we can keep the same framework as the instruction trace decoder. Most of the stream state handling can be reused without changes, we only have to add the headers and related payload sizes. For the rest of the trace decoding, we merely need to add the new action codes and add the logic associated with each action. Furthermore, the data trace state is very similar to the instruction trace state, containing three address registers, a timestamp value and a few additional state components.

The challenge, in this case, is re-associating the data trace with the instruction trace. The ETM provides an algorithm that must be followed [26]. Without getting lost in the details, it tries to match trace data P0 elements to P1 elements. The algorithm involves scanning for data synchronization markers in both streams simultaneously. At first glance, matching these keys might prove difficult for a hardware implementation since it involves moving both backwards and forwards through the trace streams looking for keys — this could be a very hard problem to solve efficiently. As for as we know, the data trace stream and instruction stream has to be generated as two separate streams by the ETM, so the resynchronization algorithm is unavoidable. We leave this implementation as future work.

Additionally, the data trace stream internally requires a similar resynchronization algorithm to reconstruct the relationships between P1 and P2 elements, where P2 elements are the traced *values* of a load or store instruction. Zeinabolin et al. introduce a hardware implementation for resynchronizing the P1 and P2 elements in [90]. Synchronizing P1 and P2 elements is easier than synchronizing P0 and P1 elements, because P1 elements have a one-to-one relationship to P2 elements, but a many-to-one relationship to P0 elements. To the best of our knowledge, there currently exists no hardware implementation for synchronizing P0 elements to P1 elements.

A further question is how quickly the TPIU could become a bottleneck when collecting both streams. To reduce bandwidth it would make sense to filter the stream and scan for the most interesting data instead of collecting everything always, so we would have to design policies on what to trace. The details of such policies need to be made based on real-world bandwidth requirements and should be tested in practice first.

8.1.6 Supporting Instrumentation

The STM/ITM has its own trace protocol specification [21] and requires a separate trace decoder. The protocol is similar in spirit to the ETM protocol and some elements like the synchronization packets are the same. The number of packets is much smaller, as the packets mostly propagate the values that are written to the STM/ITM registers or hardware events. We suggest using the same technique for the instrumentation trace as with the instruction trace. Here we also have the opportunity to lower the unroll factor and simplify the decoder as we expect a lower bandwidth for the instrumentation trace stream than the instruction trace stream. This entirely depends on how the code is instrumented and should be adapted to each use case.

8.2 Future Experiments

In this section we describe a set of benchmarks we deem interesting and may guide future research directions that were planned on being performed once a full system implementation is complete. Originally, the tracing analysis data was meant to be hooked into the GraalVM compiler and to be evaluated on existing PGO techniques [87]. This presents a problem as the AOT compiled GraalVM native image with PGO is, for one, closed source and may only be applied with a valid enterprise license, and second, would require hoisting of profiling data to high levels of abstraction, as the insertion point for the profiling data is on the IR level, requiring a lot of custom post-processing of analysis results. From our understanding, most of this would have to be implemented from scratch, as GraalVM uses instrumentation-based-profiling and not sampling- or trace-based profiling.

Nevertheless, collecting trace data is compiler agnostic — GraalVM community edition native image has a builtin benchmark suite, which includes ScalaBench [75], Java DaCapo [32] and Renaissance [71] that contain a more representative set of benchmarks than, for example, the solely SPEC CPU benchmarks [41, 69] for server grade workloads. Finagle-http, simulating a server workload used by Twitter, the page-rank algorithm implemented on top of Apache Spark framework, or neo4j-analytics are a few notable examples used by the Renaissance suite are the types of workloads PGO is typically applied to and renaissance benchmarks are widely used across related work [56, 77, 54, 57, 55]. Other representative workloads may include long-running server database processes, like MySQL or MongoDB used by other PGO evaluations [91].

8.2.1 Evaluating Tracing Overhead

First and foremost, one of the most important aspects of hardware tracing is the minimal impact on processing speeds. The first step for a proper evaluation requires quantifying precisely the overhead incurred by a tracing session, ideally, this would be <1% to compete with LBR sampling-based techniques. This may seem straightforward but may depend widely on both the running binary and the tracing configuration. It is also unclear whether backpressure generated by the Coresight devices affects only the trace source, or may also exert backpressure on the CPU in the case of ETMs. Hardware-assisted tracing is dubbed “zero-overhead”, yet still should be empirically reestablished.

8.2.2 ETM Bandwidth Measurement

Similar to evaluating the overhead of tracing, the bandwidth depends heavily on the application, namely the density of direct branch statements that generate `Atom` packets and indirect branches generating both `Atom` packets and some form of address packet. Code with loops that require a small number of cycles per loop iteration step will produce significantly more trace packets on the stream than code with cycle-heavy branch-less computation.

Also affecting bandwidth, maybe even more than the application, is the configuration of the source device configuration. As an example, if a trace session is enabled with cycle counting, the threshold for the number of cycles that occur before a new cycle-count packet must be sent out can be configured. If this threshold is set very low, we expect a significant increase in trace bandwidth. Similarly for branch broadcasting, a requirement for our system at the moment, that incurs an overhead of additional address packets for every direct branch. Currently, to the best of our knowledge, there is no comprehensive study on trace data bandwidth, only general rules of thumb and anecdotal reports.

A trace bandwidth evaluation on the US+ can be used to extrapolate bandwidth requirements for a hardware tracing design on the ThunderX. This can be valuable for design choices, as the ThunderX has no TPIU, so the current system bottleneck on the US+ can be eliminated with higher throguhput CPU to FPGA interconnects. The Cortex-A53 on the US+ runs at around half the speed, running at 1.3GHz compared to ThunderX at 2.5GHz. Measuring the bandwidth of data received at trace decoder on US+ and linearly scaling the results based on the clock frequencies to the ThunderX should give a rough estimate of the required bandwidth requirements, assuming the same ETM configuration and running application.

8.2.3 Evaluating PMU Event Consistency

Profiling hardware events to inform compiler optimizations that rely on precise PMU event data. Accurate timing information of an observed hardware event can inform a compiler how well a binary is performing and where to apply transformations. There is uncertainty present in determining the root cause of an event, say a cache miss, especially for PMU sampling-based techniques and some research has gone into improving the precision of hardware event association with basic blocks and IRs [36, 88].

The ETM specification supports embedding of events into the trace stream. The specification recommends implementing guarantees on the timing of event embeddings (Section 3.5). We propose performing an experiment to test the precision of event embeddings by running many trace sessions on a binary and observing any variability in the ordering of events in the packet stream and the variability in the cycle distances. Once a statistical analysis is made, the trace analysis can provide guarantees of event occurrences to a compiler, which could otherwise not be made, allowing for more sophisticated cause-and-effect inference between negative hardware events, like a cache miss, and the event-causing instructions.

This experiment requires multiple iterations on both warm- and cold-starting states to reproduce the same execution environment across iterations. Building on top of this, understanding how deterministic the general trace from steady-state trace session beginnings can make an interesting experiment when compared to sampling-based profiling.

8.2.4 Testing Correctness of the Trace Decoder

For now, all the testing is done through a set of simulation test benches in Vivado. Each submodule has its own test bench, usually where we feed a set of representative trace packets and check the output that has been precomputed by hand based on the Coresight specification. For end-to-end testing without a AXI-DMA we used a Microblaze connected to the output of trace decoder through the Microblaze AXI interface. This was made more for intermediate “printf-style” debugging and sanity checking. Ideally, to check for correctness, we would compare the trace data results produced by our processing stages in the PL to the results of feeding the same trace data to the OpenCSD [9]. The experiment would work as follows:

— Configure the tracing session as usual, including outputting the trace data over the TPIU to the PL. At the same time, use the ETR on the US+ to write the pure trace data to system memory. This works concurrently.

— Results of the trace decoder are written back to PL memory over AXI-DMA and is accessible from the Cortex-A53s.

— The trace data written to system memory from the ETR is passed to the OpenCSD. This acts as our trace decoding oracle.

— We may have different requirements and output data for our trace decoder, as we attempt to extract semantics. Therefore this step would involve also a piece of translation software to either extract the same meaning from the results of the trace decoder or translate our decoded data to the same format as the OpenCSD.

8.3 Big picture vision

We explore some more practical applications of an online trace analyzer in this section. These project possibilities are larger in scop than direct system extension and we believe these topics are worthy of further discussions.

8.3.1 Runtime Binary Optimizations Performed on the FPGA

The recently published paper Ocolos applies code layout optimizations at runtime. It does so by continuously collecting profiles, pausing the execution of the running binary to perform optimizations. During the pauses, Ocolos uses the collected profiles to reorder basic blocks or functions by duplicating certain code sections, updating relevant pointers in the binary, and subsequently resume the execution of the binary [91]. Why not take this a step further and perform these steps on the PL? While the PL is not suited for more sophisticated static analysis or CFG traversals, code reordering may be feasible to perform on an FPGA. An FPGA could read the memory section holding the program image as a stream of data, and then write back the updated stream back into the PS memory. This could increase the benefits reported by Ocolos, as the system would no longer have to overcome the lost cycles spent pausing execution and performing updates, as the overhead is pushed to the PL and the CPU can continuously perform useful work.

8.3.2 Exploring Trace-based Compiler Optimizations

The case for PGO has been well made, and encouraged numerous solutions, especially in reducing L1-I and I-TLB [58, 59, 33, 66, 65, 77, 56, 91, 67, 35, 28], however all of these apply to branch frequency profiling data. Trace data allows for a greater set of optimizations that are not feasible with sampling-based approaches. The concept of trace-based optimizations is nothing new, especially in the world of JIT compilers [45, 34, 48, 29]. However, the meaning of trace-based optimizations should be interpreted differently from the optimizations we wish to explore in this area AOT compilation. In the context of a JIT, trace-based optimizations try to produce machine code for the hottest paths, in other words, the execution path that is executed most frequently. If a colder path is taken, execution is deoptimized back into the interpreter.

We suggest exploring a similar concept with AOT compilation. For trace-based optimizations, sampling-based profiles are not adequate to make any informed decisions. The reasons are twofold: sampling-based profiling gives us only branch frequencies. We can view branch frequencies as *local* profile information. It is local in the sense that the only information we have at branch b is how likely the branch will be taken/not taken. In contrast, with tracing data, we have a *global* view of the system, meaning given we are at branch b_n, how likely are we to take the branch given that our execution path has been $b_{n-1} \dots b_0$. Of course, the number of paths in the system is theoretically infinite and not all paths have been observed. Yet, this still enables us to observe deep branch correlations. Second, sampling-based techniques will not be able to properly collect data from colder code regions, and some parts of an execution path are likely to be missed due to infrequent execution. This means the execution path can not be properly reconstructed for paths that contain both cold and hot code.

To the best of our knowledge, we are not aware of any research in this exact direction. It remains interesting to examine if deep correlations between branches exist and if this is a pattern we can optimize for.

8.3.3 Informing Cluster-wide Scheduling Policies

An immense body of work has gone into cluster data center-wide schedulers and managers. Without going into too much detail, most schedulers rely on profiling data to make decisions on which jobs to collocate onto the same machine in order to both maximize cluster utilization and minimize interference between jobs while abstracting away the need to specify

resources from the user. Interference comes from sharing memory, caches, and CPU time and an application may cause slowdowns of other applications running on the same machine if they have similar resource requirement characteristics. As an example, Quasar [40] performs an interference classification step, among others, on applications before scheduling them. Here, profiles collected consist mostly only of runtime measurements, i.e how long did a job take to complete, or for example, how many Queries Per Second (QPS) a service can complete. This profiling information could be enhanced with trace-based analysis produced by our trace decoder, as more fine-grain performance information is made available to a scheduler.

Another example is when making scheduling opportunities, cloud providers would want to provision resources in the Goldilocks zone — like provision just the right amount of memory for a serverless application. This can be a challenging objective to solve, even more so considering language runtimes with memory management come into play [52]. This is a possible use case for a combined instruction and data trace, that gives detailed insight into the memory usage of an application. Collecting trace data from a large number of runs for an application may provide very detailed memory requirements that go far beyond merely measuring peak memory usage or other memory use profiling.

Taking things a step further, the trace could be used to throttle code regions that have high memory requirements, taking an approach of fitting the application to the resource it's given as opposed to fitting the resources to the application. Something akin to this has been done before, [80, 81], but instead uses solely (Intel) PMU counters to measure the bandwidth requirements of code regions. A limitation of this work, mentioned by the authors, is the inability to measure cache usage only from PMUs, which is precisely what a data trace could solve. Furthermore, reassociating memory bandwidth measurements with the instructions causing reads and writes to memory is fairly coarse-grained, essentially matching each 1ms interval with the most frequently executed basic block in that 1ms interval. Coresight could offer potentially basic-block-level precision otherwise not achievable.

In summary, an online Coresight tracing technique as we have implemented could aid in estimating resource utilization more precisely than previous techniques can, which may result in better resource assignment policies. We believe this would be most interesting to study on a device that enables data tracing as well as instruction tracing and would require a full Coresight decoding system for both streams.

Unfortunately, as far as we are aware, the only mainstream processor that supports data tracing is the Cortex-M3, so this would have very limited use cases. This begs the question, if we run an application on a Cortex-M3 processor with data and instruction trace and analyze the trace, how applicable are the analysis results of the same application running on another processor? In theory, the data trace produced by an application is not processor specific unless it has been compiled differently for the architecture. This, however, remains an unanswered question. Nevertheless, we believe that Coresight may have interesting applications in the field of resource estimation for cloud workloads, and an instruction and data trace analyzer implemented completely in hardware would make this type of resource analysis feasible in practice.

Chapter 9

Conclusion

In this work we have introduced a novel ETMv4 instruction trace decoder implementation running on a Zynq Ultrascale+ MPSoC xczu5ev-sfvc784-2-i that can handle up to 4 bytes/cycle at 250MHz, giving a total throughput of 1 GB/s per core, an improvement of 8× over any prior work. The three key ideas to achieve this are, first, performing the decoding process byte-wise with the introduction of the stream state, while breaking down the ETM specification into internal packet types such that the specification is more amenable to decoding in hardware. Second, unrolling the byte stream is the only way to parallelize the decoding process fully, regardless of packet types in the trace stream. Lastly, once the decoding process is unrolled, pipelining optimizations need to be applied to support the target operating frequency of 250MHz. The design is also adaptable to specific performance needs and can be tuned by adjusting the unroll factor. With an unroll factor of four and a trace decoder for each of the four cores on the US+, the trace decoders would use up around 10% of the device in the most dominant resource (LUT).

We further designed modular subcomponents for frame generation, frame decoding, and demultiplexing the trace streams that can be reused for any device with a TPIU. We also extended the CSAL to support using TMCs as intermediate buffers in hardware FIFO mode. We cleared up what seems to be a bug in the constraint translation of the Zynq Ultrascale+ MPSoC in regard to the TPIU clock speeds and showed how to apply configuration changes of the PS block design on the US+.

We further provide a comprehensive guide to Coresight in general and specifically for both the ThunderX and Zynq Ultrascale+ MPSoC implementations of Coresight, and the ETM instruction trace stream specification that can be used as a guide for future research in this area. While we have not yet achieved a running Coresight trace session the ThunderX, we have at least narrowed down the problem to a specific stage in the path from the trace source to the OCLA, being the propagation of trace data from the ETM to the DTX.

Working with Coresight proved more challenging than initially expected, especially on the ThunderX. Updates to the CSAL were required and designing the trace decoder took more time than we had foreseen. This is becuase there is no clear path from a PTM, decoder to an ETM decoder, which therefore had to be implemented from scratch. Because of this, many of the experiments and more practical applications of trace data that were planned are still left for future work. In Chapter 8, we outlined in which directions we envisioned future work in this area to commence. Possible directions we described include using Coresight trace data for more sophisticated compiler optimizations that could not be performed with sampling-based techniques, applying code layout optimizations during runtime using the FPGA, or using the traces to get a better idea of resource utilization for cluster-wide scheduling and resource assignment.

Furthermore, this work is foundational to any project that relies on real-time trace data using any system with an ETM, e.g. the ThunderX processor on the hybrid CPU/FPGA Enzian platform. An important takeaway is that previous approaches to tracing an application, proposed both in this work and in the work done by Schmid [74] are not directly applicable to the ThunderX without substantial changes to the trace decoding process, as it does not support the branch-broadcasting mode.

A possible direct next step is completing the system by finishing the AXI-DMA to write back the results to PS memory, and using this for end-to-end testing. Another possibility is adapting the same decoding function to support ITM trace decoder and a data trace stream decoder. Once a data trace decoder is implemented, hardware implementations of the resynchronization

algorithms between the data trace stream and instruction trace stream should be examined. Finally, a possible project could include adding analysis stages of the decoded trace stream in hardware to extract actionable profiles that can be fed directly into a compilation toolchain.

Acknowledgements

First and foremost, I would like to thank my supervisor Nora Hossle for guiding me throughout this thesis and keeping me grounded and focused on the task at hand without losing myself in outlandish ideas. I wish to extend my gratitude to the entire Systems Group at ETH Zurich, the journey has been very insightful and I cannot recall a time in which I have learned so much in such a short period of time. As such, I am thankful to Prof. Dr. Ana Klimovic and Prof. Dr. Timothy Roscoe for allowing me to write my thesis in this group and allowing me to explore a topic I am passionate about. Lastly, I am grateful to Dr. David Cock, Daniel Schwyn, Adam Turowski, and Dr. Michael Giardino for lending a helping hand when I was blocked or for joining the debugging sessions attempting to get a Coresight trace session to work on the ThunderX machine.

Bibliography

[1] Axi dma standalone driver. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842100/AXI+DMA+Standalone+Driver. Accessed: 2023-12-4.

[2] Coresight access library. https://github.com/ARM-software/CSAL. Accessed: 2023-5-4.

[3] Coresight AutoFDO collect etm data for autofdo. https://android.googlesource.com/platform/system/extras/+/master/simpleperf/doc/collect_etm_data_for_autofdo.md. Accessed: 2023-4-4.

[4] Gcc, the gnu compiler collection. https://gcc.gnu.org/. Accessed: 2023-10-4.

[5] Hsi debugging and optimization techniques. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841693/HSI+debugging+and+optimization+techniques. Accessed: 2023-10-4.

[6] Linux dma from user space 2.0. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/1027702787/Linux+DMA+From+User+Space+2.0. Accessed: 2023-13-4.

[7] Linux Kernel Coresight hw assisted tracing on arm. https://docs.kernel.org/trace/coresight/index.html. Accessed: 2023-4-4.

[8] Mercury xu5 pe1 refernce design. https://github.com/enclustra/Mercury_XU5_PE1_Reference_Design. Accessed: 2023-10-4.

[9] Opencsd - an open source coresight™ trace decode library. https://github.com/Linaro/OpenCSD. Accessed: 2023-5-4.

[10] perf - performance analysis tools for linux. https://man7.org/linux/man-pages/man1/perf.1.html. Accessed: 2023-5-4.

[11] Simpleperf. https://android.googlesource.com/platform/system/extras/+/master/simpleperf/doc/README.md. Accessed: 2023-4-4.

[12] The Linux Kernel - CoreSight - ARM Hardware Trace . https://docs.kernel.org/trace/coresight/index.html. Accessed: 2023-5-4.

[13] Adve, S. V., Burger, D., Eigenmann, R., Rawsthorne, A., Smith, M. D., Gebotys, C. H., Kandemir, M. T., Lilja, D. J., Choudbary, A., Fang, J. Z., et al. Changing interaction of compiler and architecture. *Computer 30*, 12 (1997), 51–58.

[14] AMD Xilinx Inc. Zynq-7000 all programmable soc UG585, 2012.

[15] AMD Xilinx Inc. Vivado design suite user guide design analysis and closure techniques UG906, 2021.

[16] AMD Xilinx Inc. Zynq ultrascale+ mpsoc data sheet DS891, 2022.

[17] AMD Xilinx Inc. Zynq ultrascale+ mpsoc software developer guide UG1137, 2022.

[18] AMD Xilinx Inc. Zynq ultrascale+ device technical reference manual UG1085, 2023.

[19] Amit, N., Jacobs, F., and Wei, M. Jumpswitches: Restoring the performance of indirect branches in the era of spectre. In *USENIX Annual Technical Conference* (2019), vol. 150.

[20] ARM Ltd. Coresight™ components DDI0314H, 2009.

[21] ARM Ltd. Coresight™ system trace macrocell DDI0444B, 2010.

[22] ARM Ltd. Coresight™ trace memory controller DDI0461B, 2010.

[23] ARM Ltd. Coresight™ program flow trace™pftv1.0 and pftv1.1architecture specification IHI0035B , 2011.

[24] ARM Ltd. Arm® cortex®-a53 mpcore processor DDI0500D, 2014.

[25] ARM Ltd. Arm® coresight™ soc-400 DDI0480G, 2015.

[26] ARM Ltd. Embedded trace macrocell architecture specification etmv4.0 to etm4.6 ARM IHI0064H, 2020.

[27] ARM Ltd. Arm® coresight™ architecture specification v3.0 ARM IHI0029F, 2022.

[28] Ayers, G., Nagendra, N. P., August, D. I., Cho, H. K., Kanev, S., Kozyrakis, C., Krishnamurthy, T., Litz, H., Moseley, T., and Ranganathan, P. Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture* (2019), pp. 462–473.

[29] Bala, V., Duesterwald, E., and Banerjia, S. Transparent dynamic optimization. Tech. rep., Citeseer, 1999.

[30] Barnes, R., Chaiken, R., and Gillies, D. Feedback-directed data cache optimizations for the x86. In *2nd ACM Workshop on Feedback-Directed Optimization (FDO)* (1999).

[31] Baryshnikov, M. Fpga-based support for predictable execution model in multi-core cpu. Master's thesis, Czech Technical University in Prague, Prague, Czech Republic, 2018.

[32] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., et al. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (2006), pp. 169–190.

[33] Boehm, O., Citron, D., Haber, G., Klausner, M., and Levin, R. Aggressive function inlining with global code reordering. *IBM Technical Paper* (2006).

[34] Bolz, C. F., Cuni, A., Fijalkowski, M., and Rigo, A. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (2009), pp. 18–25.

[35] Chen, D., Li, D. X., and Moseley, T. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (2016), pp. 12–23.

[36] Chen, D., Vachharajani, N., Hundt, R., Liao, S.-w., Ramasamy, V., Yuan, P., Chen, W., and Zheng, W. Taming hardware event samples for fdo compilation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* (2010), pp. 42–52.

[37] Chen, W., Ray, S., Bhadra, J., Abadir, M., and Wang, L.-C. Challenges and trends in modern soc design verification. *IEEE Design & Test 34*, 5 (2017), 7–22.

[38] Cock, D., Ramdas, A., Schwyn, D., Giardino, M., Turowski, A., He, Z., Hossle, N., Korolija, D., Licciardello, M., Martsenko, K., Achermann, R., Alonso, G., and Roscoe, T. Enzian: An open, general, cpu/fpga platform for systems software research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2022), ASPLOS '22, Association for Computing Machinery, p. 434–451.

[39] Decker, N., Dreyer, B., Gottschling, P., Hochberger, C., Lange, A., Leucker, M., Scheffel, T., Wegener, S., and Weiss, A. Online analysis of debug trace data for embedded systems. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2018), IEEE, pp. 851–856.

[40] Delimitrou, C., and Kozyrakis, C. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices 49*, 4 (2014), 127–144.

[41] Dixit, K. M. New cpu benchmark suites from spec. In *Digest of Papers COMPCON Spring 1992* (1992), IEEE, pp. 305–310.

[42] Dreyer, B., Hochberger, C., Lange, A., Wegener, S., and Weiss, A. Continuous non-intrusive hybrid wcet estimation using waypoint graphs. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)* (2016), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[43] Enclustra GmbH. Mercury XU5 SoC Module user manual, 2021.

[44] Fisher, J. A., and Freudenberger, S. M. Predicting conditional branch directions from previous runs of a program. *ACM SIGPLAN Notices 27*, 9 (1992), 85–95.

[45] Gal, A., Probst, C. W., and Franz, M. Hotpathvm: An effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments* (2006), pp. 144–153.

[46] Havlak, P. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems (TOPLAS) 19*, 4 (1997), 557–567.

[47] Iannillo, A. K., Natella, R., Cotroneo, D., and Nita-Rotaru, C. Chizpurfle: A gray-box android fuzzer for vendor service customizations. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)* (2017), IEEE, pp. 1–11.

[48] Inoue, H., Hayashizaki, H., Wu, P., and Nakatani, T. A trace-based java jit compiler retrofitted from a method-based compiler. In *International Symposium on Code Generation and Optimization (CGO 2011)* (2011), IEEE, pp. 246–256.

[49] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual., 2022.

[50] Intel Corporation. Intel. 2020. intel® vtune™ profiler, 2023.

[51] Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H., and Nakatani, T. A study of devirtualization techniques for a java just-in-time compiler. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2000), pp. 294–310.

[52] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[53] Kanev, S., Darago, J. P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y., and Brooks, D. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015), pp. 158–169.

[54] Khan, T. A., Brown, N., Sriraman, A., Soundararajan, N. K., Kumar, R., Devietti, J., Subramoney, S., Pokam, G. A., Litz, H., and Kasikci, B. Twig: Profile-guided btb prefetching for data center applications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (2021), pp. 816–829.

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

[56] Khan, T. A., Ugur, M., Nathella, K., Sunwoo, D., Litz, H., Jiménez, D. A., and Kasikci, B. Whisper: Profile-guided branch misprediction elimination for data center applications. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2022), IEEE, pp. 19–34.

[57] Khan, T. A., Zhang, D., Sriraman, A., Devietti, J., Pokam, G., Litz, H., and Kasikci, B. Ripple: Profile-guided instruction cache replacement for data center applications. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)* (2021), IEEE, pp. 734–747.

[58] Kumar, R., Grot, B., and Nagarajan, V. Blasting through the front-end bottleneck with shotgun. *ACM SIGPLAN Notices 53*, 2 (2018), 30–42.

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

[60] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* (2004), IEEE, pp. 75–86.

[61] Lee, Y., Lee, J., Heo, I., Hwang, D., and Paek, Y. Using coresight ptm to integrate cra monitoring ips in an arm-based soc. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 22*, 3 (2017), 1–25.

[62] Marin, G., Alexandrov, A., and Moseley, T. Break dancing: low overhead, architecture neutral software branch tracing. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (2021), pp. 122–133.

[63] Mijat, R. Better trace for better software: introducing the new arm coresight system trace macrocell and trace memory controller. *ARM, White Paper* (2010).

[64] Mosaner, R., Leopoldseder, D., Stadler, L., and Mössenböck, H. Using machine learning to predict the code size impact of duplication heuristics in a dynamic compiler. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (2021), pp. 127–135.

[65] Newell, A., and Pupyrev, S. Improved basic block reordering. *IEEE Transactions on Computers 69*, 12 (2020), 1784–1794.

[66] Ottoni, G. Hhvm jit: A profile-guided, region-based compiler for php and hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2018), pp. 151–165.

[67] Panchenko, M., Auler, R., Nell, B., and Ottoni, G. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2019), IEEE, pp. 2–14.

[68] Pettis, K., and Hansen, R. C. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation* (1990), pp. 16–27.

[69] Phansalkar, A., Joshi, A., Eeckhout, L., and John, L. K. Measuring program similarity: Experiments with spec cpu benchmark suites. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* (2005), IEEE, pp. 10–20.

[70] Prokopec, A., Duboscq, G., Leopoldseder, D., and Wïrthinger, T. An optimization-driven incremental inline substitution algorithm for just-in-time compilers. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2019), IEEE, pp. 164–179.

[71] Prokopec, A., Rosa, A., Leopoldseder, D., Duboscq, G., Tuma, P., Studener, M., Bulej, L., Zheng, Y., Villazon, A., Simon, D., et al. Renaissance: benchmarking suite for parallel applications on the jvm. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019), pp. 31–47.

[72] Ramdas, A., Cock, D., Roscoe, T., and Alonso, G. The enzian coherent interconnect (eci): opening a coherence protocol to research and applications. *LATTE '21* (2021).

[73] Ren, G., Tune, E., Moseley, T., Shi, Y., Rus, S., and Hundt, R. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro 30*, 4 (2010), 65–79.

[74] Schmid, P. Runtime verification with tessla on enzian. Master's thesis, ETH Zurich, 2019.

[75] Sewe, A., Mezini, M., Sarimbekov, A., and Binder, W. Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (2011), pp. 657–676.

[76] Shabalin, D., and Odersky, M. Interflow: interprocedural flow-sensitive type inference and method duplication. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala* (2018), pp. 61–71.

[77] Song, S., Khan, T. A., Shahri, S. M., Sriraman, A., Soundararajan, N. K., Subramoney, S., Jiménez, D. A., Litz, H., and Kasikci, B. Thermometer: profile-guided btb replacement for data center applications. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (2022), pp. 742–756.

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

[79] Su, A. P., Kuo, J., Lee, K.-J., Huang, J., Jian, G.-A., Chien, C.-A., Guo, J.-I., and Chen, C.-H. Multi-core software/hardware co-debug platform with arm coresight™, on-chip test architecture and axi/ahb bus monitor. In *Proceedings of 2011 International Symposium on VLSI Design, Automation and Test* (2011), IEEE, pp. 1–6.

[80] Tang, L., Mars, J., and Soffa, M. L. Compiling for niceness: Mitigating contention for qos in warehouse scale computers. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (2012), pp. 1–12.

[81] Tang, L., Mars, J., Wang, W., Dey, T., and Soffa, M. L. Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers. *ACM SIGPLAN Notices 48*, 4 (2013), 89–100.

[82] Wei, T., Mao, J., Zou, W., and Chen, Y. A new algorithm for identifying loops in decompilation. In *Static Analysis: 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007. Proceedings 14* (2007), Springer, pp. 170–183.

[83] Weingarten, M. E., Theodoridis, T., and Prokopec, A. Inlining-benefit prediction with interprocedural partial escape analysis. In *Proceedings of the 14th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (2022), pp. 13–24.

[84] Weiss, A., and Lange, A. Trace-data processing and profiling device, Mar. 15 2016. US Patent 9,286,186.

[85] Whaley, J. A portable sampling-based profiler for java virtual machines. In *Proceedings of the ACM 2000 conference on Java Grande* (2000), pp. 78–87.

[86] Wicht, B., Vitillo, R. A., Chen, D., and Levinthal, D. Hardware counted profile-guided optimization. *arXiv preprint arXiv:1411.6361* (2014).

[87] Wimmer, C., Stancu, C., Hofer, P., Jovanovic, V., Wögerer, P., Kessler, P. B., Pliss, O., and Würthinger, T. Initialize once, start fast: application initialization at build time. *Proceedings of the ACM on Programming Languages 3*, OOPSLA (2019), 1–29.

[88] Yi, J., Dong, B., Dong, M., and Chen, H. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems* (2020), pp. 98–105.

[89] Zeinolabedin, S. M. A., Partzsch, J., and Mayr, C. Real-time hardware implementation of arm coresight trace decoder. *IEEE Design & Test 38*, 1 (2020), 69–77.

[90] Zeinolabedin, S. M. A., Partzsch, J., and Mayr, C. Analyzing arm coresight etmv4. x data trace stream with a real-time hardware accelerator. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2021), IEEE, pp. 1606–1609.

[91] Zhang, Y., Khan, T. A., Pokam, G., Kasikci, B., Litz, H., and Devietti, J. Ocolos: Online code layout optimizations. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2022), IEEE, pp. 530–545.