



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Master's Thesis Nr. 343

Systems Group, Department of Computer Science, ETH Zurich

High-speed Tracing of Coherence Traffic using FPGAs

by

Manuel Bröchin

Supervised by

Dr. Michael Joseph Giardino

Dr. David Cock

Prof. Timothy Roscoe

March 2021–August 2021

Acknowledgement

I would like to thank my supervisors Dr. Michael Giardino and Dr. David Cock for their useful feedback and encouraging comments. Further, I would like to thank Abishek Ramdas for patiently explaining to me the details of ECI and Adam Turowski for knowing seemingly everything about the Enzian system. Next, I want to thank Prof. Timothy Roscoe for building the Enzian computer and for providing such a great research opportunity. Finally, I thank the whole systems group at ETH for being a stimulating and uplifting working environment.

Abstract

Modern computing platforms are becoming more and more heterogeneous. Reconfigurable hardware additionally lowers the cost of designing custom hardware components. Dedicated components enable higher performance than general purpose CPUs, but this heterogeneity comes at the cost of significantly more data movement. As a consequence, there is an increasing need to inspect and understand the high-speed communication on interconnects between components.

This thesis develops an FPGA module, the tracing engine, that allows inspecting and filtering coherence traffic on the Enzian computer. Being able to directly inspect coherence traffic allows answering a multitude of interesting questions about the Enzian system. Characterizing cache misses of workloads and finding bottlenecks limiting the inter-socket bandwidth are only two of them.

Inspecting raw interconnect data is not satisfactory. Due to the high-speed nature of the interconnect the resulting traces are huge. The tracing engine thus allows the user to provide a filter to limit the output trace to the essential messages. A simple filter language targeted to the coherence protocol allows the user to formulate filters as general NFAs over a set of basic predicates. Filters can be loaded without re-programming the FPGA, thus allowing an interactive use.

Intended for system instrumentation, the tracing engine is designed to be as compact as possible while still providing enough flexibility to enable many use cases. This trade-off is achieved by providing a mixture of compile-time and runtime configurability: The set of basic predicates that can be used to formulate filters, as well as the maximum size of the filter NFA are fixed at compile time, while the concrete filter can be configured at runtime.

We show the versatility and usefulness of the proposed approach by inspecting a set of varied performance and correctness properties of Enzian. Moreover, we thoroughly evaluate the tracing engine: from hardware cost scaling to expressivity of the filter language. We find that the tracing engine allows us to filter high-speed interconnect data at full line-rate of 30GB/s. The tracing engine is developed for one specific interconnect protocol, however, we show that the approach is much more general and can easily be re-targeted to different protocols. It has a small hardware cost and is able to run alongside major applications on the FPGA while providing flexibility both at compile time and at runtime.

Contents

1	Introduction	4
1.1	Problem formulation	4
1.2	Enzian	5
1.3	ECI	6
1.4	Enzian coherency	8
1.5	NFAs	12
1.6	FPGA, Xilinx and Verilog	15
1.7	Use cases of tracing VC layer messages	16
1.8	Overview	18
2	Related Work	19
3	Design	23
3.1	Design overview	23
3.2	Integration into Enzian	25
3.3	HDL design	29
3.3.1	Input Reduction and Decoding	31
3.3.2	Runtime reconfiguration of NFAs	34
3.3.3	Windowing	37
3.3.4	Output reduction	39
3.3.5	Pipelining and Flow control	39
3.4	Frontend	42
3.4.1	Making the frontend generic	46
3.5	Control and output streaming over PCIe	47
4	Experiments	48
4.1	Simple trace filters and modularity of the design	48
4.2	Characterizing cache misses	54
4.3	Validating Enzian coherency	59
5	Evaluation	62
5.1	Hardware resource scaling	64
5.2	Expressivity of the filter language	71
5.3	Mapping algorithm	74
5.4	Design decisions	77
5.4.1	Homogeneous NFAs	77
5.4.2	Choice of overlay graph	79
5.4.3	Runtime reconfiguration with shift registers	81
6	Conclusions	82
6.1	Summary	82
6.2	Limitations and future work	84

1 Introduction

1.1 Problem formulation

The modern computing landscape is becoming more and more heterogeneous. Many types of dedicated hardware accelerators solve a multitude of workloads at much higher performance than general purpose CPUs. While this trend is the logical consequence of the end of Moore scaling, it makes both system construction and user level development more difficult. Modern heterogeneous systems introduce a lot of complexity compared to more classical, homogeneous systems and obtaining near-optimal performance is often a challenge. One big part of the added complexity is due to communication: First, the various hardware components need to communicate at very high speeds to keep up with the increased computational throughput. Second, there is more communication to begin with because computation, previously performed centrally at the CPU, now gets distributed across multiple chips that require synchronization.

A second, related trend is the popularization of reconfigurable hardware platforms such as Field Programmable Gate Arrays (FPGAs). Despite having been around for a long time, FPGAs get a lot of attention as parts of heterogeneous systems recently thanks to huge improvements of their capabilities. Reconfigurable hardware significantly lowers the threshold for developing a custom hardware accelerator. Opening up this space to research and users means that a lot more innovation is to be expected, including the domain of communication protocols.

All these trends call for ways of inspecting and understanding the communication between hardware components.

The systems group at ETH is building the Enzian computer which integrates a 48-core Cavium Marvell ThunderX processor with an FPGA. The two chips communicate over the CPU's native coherence interface. In this effort, all the above problems are faced repeatedly:

- The ThunderX CPU expects its peer to communicate by using its native cache coherence protocol, thus this protocol needs to be understood and then correctly implemented by the developers.
- The FPGA-side implementation of the cache coherence protocol needs to be very high-performance, otherwise it will limit the performance of any workload that runs on Enzian.
- Even with a high-performance communication channel between CPU and FPGA, offloading workloads to the FPGA to improve performance is a difficult problem. By only looking at CPU performance counters it is very hard find flaws in an offloading strategy.
- One goal of Enzian is to experiment with different implementations of the cache coherence, thus these problems will be faced again and again.

In order to help overcome these challenges, we need a convenient way of inspecting the communication on the coherence link between the CPU and the FPGA. In this thesis we develop a toolchain to perform interconnect tracing on the Enzian research computer for this purpose. The main component of this toolchain is a synthesizable hardware description of a tracing module (a tracing engine) that can be implemented on the FPGA alongside other applications. The module unintrusively taps all coherence link traffic and allows the user to inspect this traffic on a remote host. In order to deal with the very high bandwidth of the coherence link (30GB/s), the module provides a way of filtering the stream of coherence communication before transferring it to the user over PCIe. Concretely, the hardware implementation of the tracing engine should satisfy the following requirements:

- The tracing engine should be able to process the coherence data at full line-rate.

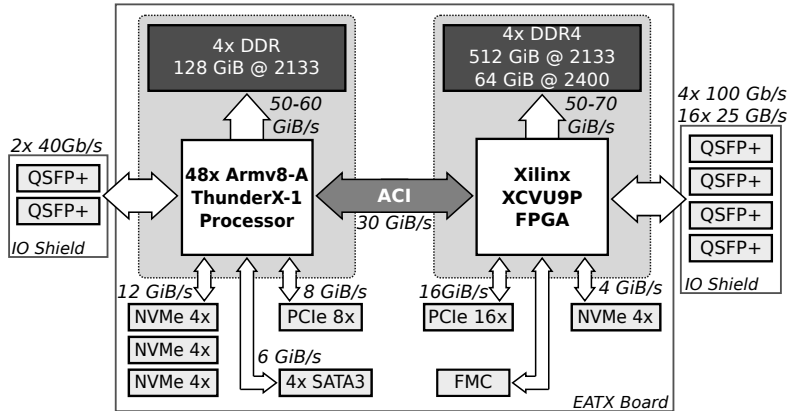


Figure 1: Enzian block diagram

- The filter applied to the data stream should be configurable at runtime to improve usability.
- Enzian is not a finished system but is expected to change with time. The tracing engine should thus be independent of the rest of the system and not rely on a specific implementation of the communication stack or the coherence protocol on the FPGA. This makes it easily portable to future versions of Enzian.
- The engine should have a small hardware cost. It should be able to run alongside major applications that require most hardware resources of the FPGA.
- The design of the tracing engine should be flexible, both at compile time and at runtime. In particular, it should be possible to re-target the tracing engine with relative ease to a different interface at compile time. Moreover, given an implemented tracing engine on a running system, the engine should allow to express many different patterns without requiring re-synthesis.

Of course, there is a trade-off between expressiveness/flexibility and hardware cost. Moreover, flexibility is not a clearly defined metric. We approach this design challenge by deciding on a varied set of use cases and then target the tracing engine towards those use cases. We believe that this will lead to a design that strikes a good compromise between flexibility and cost.

In the remainder of the introduction, we first introduce some specifics of the Enzian system in general and the coherence link in particular. Then we will go over some background on Nondeterministic Finite Automata (NFAs), which are the central formalism we use in this work. Next we introduce some basics of FPGA programming. We then will sketch out possible applications of the tracing engine in order to get some understanding of what a flexible design should allow us to do. Finally, we provide an overview of the remainder of the thesis and describe our contributions.

1.2 Enzian

Enzian [1] is a research computer built at ETH Zurich to explore the full stack of hardware/software co-design from systems software to hardware accelerators. Enzian is designed with the goal of maximizing capacity and configuration flexibility. Enzian is a heterogeneous architecture integrating a 48-core Marvell

Thunder-X (THX) CPU and a Xilinx Virtex Ultrascale+ FPGA as depicted in [Figure 1](#) (taken from [\[2\]](#)). Both nodes have access to local DRAM and coherent access to the other node’s DRAM by communicating over ECI, the cache-coherency protocol of Enzian. ECI is a separately-developed protocol which interoperates with CCPI, the native coherence protocol of the THX.

Enzian provides ample hardware capabilities. We summarise the relevant hardware configuration used in this work: The CPU is a Marvell Cavium ThunderX-1 CN8890-NT with 48 ARM v8.1 cores, each running at 2GHz. The CPU has access to 128GiB of DDR4 DRAM, composed of four 32GiB DIMMs, each providing 2133 MT/s. Extension cards can be connected using a PCIe Gen3 x8 slot, providing 8GB/s throughput.

The FPGA is a Xilinx Virtex Ultrascale+ XCVU9P-FLGB2104-E. It has access to at most 512GiB DDR4 DRAM, composed of 4 128GiB DIMMs, each providing 2133MT/s. The FPGA also has PCIe connectivity via a Gen3 x16 slot, providing a maximum throughput of 16GB/s.

The ECI cache coherence link consists of 24 serial lanes, each providing 10Gb/s throughput. At the time of writing, only 12 lanes are used, limiting the total coherence bandwidth to 15GB/s between CPU and FPGA.

Specifically in this work, the CPU and the FPGA are further connected by DMA over PCIe. In this case, bandwidth is limited to 8GB/s by the CPU’s PCIe slot.

1.3 ECI

ECI connects the L2 cache on the THX with the FPGA. ECI is a 3-layer protocol where the individual layers are called Interlaken, block and Virtual Channel (VC) layer. In this work we are almost exclusively interested in tracing and filtering VC layer messages. In the following we provide a short overview of the entire ECI protocol stack. We also explicitly discuss the lower layers for two reasons: First, many characteristics of the VC layer are results of lower layer protocols, thus it is necessary to understand the full stack in order to understand the VC layer. Second, in [subsection 4.1](#) we will perform an experiment tracing block layer data in order to show the versatility of the toolchain.

At the lowest layer, a version of the Interlaken protocol [\[3\]](#) provides physical connectivity between the two endpoints. Interlaken sends data in blocks of 64 bytes that are striped across the physical lanes. Interlaken uses per-lane meta-frames to align the striped data at the receiver. Each meta-frame consists of a total of 2048 words of 8B each. Four of these words are control words containing meta-information, the remaining 2044 words are payload. Interlaken uses 64B/67B encoding of all transmitted data to ensure enough state changes for clock recovery. Together these two measures introduce some overhead and reduce the efficiency at the physical layer to $\frac{64}{67} \cdot \frac{2044}{2048} \approx 0.95$. The protocol employed as part of ECI differs from the Interlaken spec in that it doesn’t allow variable length of data bursts. Because all transmitted packets have the same size, this additional flexibility is not needed and the additional overhead of packet delimiters at the physical layer can be avoided.

One layer up, the block layer uses the 64B blocks to send data in a fixed format: Each block carries an 8B header and seven 8B data words. There are four different types of blocks: sync, HI-data, LO-data, and idle blocks. The type of a block is indicated by the first 3 bits of its header. The structure of each type of block header is depicted in [Figure 2](#). Sync blocks carry meta information in the header and are used to synchronize the two nodes after an error condition or at startup. They carry no payload in the seven data words. High- and low-data blocks carry higher-layer messages in the seven data words. Idle blocks are sent when neither data nor sync blocks are sent because the physical layer provides a constant stream of blocks. Using meta-information in the headers, the block layer provides reliable transmission of data and flow control. In particular, every header contains a 24 bit CRC checksum field to provide error detection. Moreover, both endpoints acknowledge the reception of data blocks using the dedicated *Ack* field in all

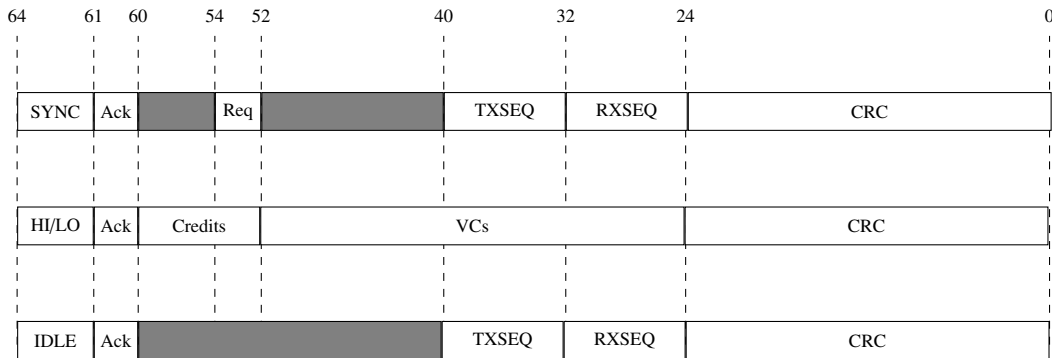


Figure 2: The three types of block layer headers: sync, hi/lo data, and idle blocks. The first three bits indicate the type of the block: SYNC = 110, HI = 100, LO = 101 and IDLE = 111.

headers. Block layer endpoints are controlled by the block-layer state machine that detects transmission errors, executes synchronization and retransmission sequences and controls sending of data blocks during normal execution. The block layer provides 14 channels of messages, called VCs, to the upper layer. The VCs field of data blocks indicates the recipient VC for each of the seven data words. Both high- and low-data blocks can carry data words for any of the 14 VCs. The only difference between high- and low-data blocks is for which VCs they can return credits (to be discussed shortly). Each VC is dedicated to messages with a specific purpose. The handling and interpretation of these messages happens at the third layer. Each VC is implemented at the block layer as a send- and a receive side First-In-First-Outs (FIFOs). The block layer consumes VC-layer messages from the sending FIFOs and puts them into data blocks. Because VC-layer messages have variable size, messages may be split across multiple blocks. Thus, the block layer is also responsible for re-assembly of VC messages at the receive side before putting them into the receive FIFOs. The block layer also provides per-VC flow control to ensure that no receive-FIFO overflows. This is done using a credit system: on link setup, the block layer endpoints exchange credits for each VC corresponding to the size of the respective FIFO. Moreover, upon sending some data for a VC the corresponding “amount” is subtracted from the VC’s credit. Similarly, when VC messages are consumed from the FIFO at the receiving node, this node will notify its peer about the newly freed space by using the *Credits* field in the data header. Using these credits, both block layer endpoints keep track of the fill status of the FIFOs at their peer and by ensuring that no VC’s credit ever drops below 0, overflow is avoided.

The third layer, the VC layer, is responsible for interpreting the messages of each VC and performing the corresponding action. VC messages all have an 8B header but the different types vary in their payload size: Some messages are pure notifications without any payload, other messages deliver a full Cacheline (CL) of data. The payload size of VC messages thus varies between 0 and 128 bytes, and full VC messages including header data vary between 8 and 136 bytes. Table 1 summarizes the message types of each VC. A detailed account of all message types and their meaning can be found in the previous work [4]. The current VC implementation on the FPGA provides four different VC backends to handle these 14 VCs.

- The I/O backend
- The Directory Controller (DirC)
- The multiplexed co-processor module

VC id	Message types	Acronym
0	I/O request	IREQ
1	I/O response	IRSP
2,3	Memory request w data	MREQ
4,5	Memory response w data	MRSP
6,7	Memory request w/o data	MREQ
8,9	Memory forward request	MFWD
10,11	Memory response w/o data	MRSP
12	Multiplexed co-processor data	MXC
13	Multicore debugging & link data	MDLD

Table 1: Virtual channels as provided by the block layer.

- The multicore debugging and link data module

Physical addresses on Enzian are 48-bits wide. The physical address space is partitioned in equal parts into I/O and coherent memory space. Both, I/O and coherent memory addresses, are further divided over the two nodes, CPU and FPGA. I/O memory space allows read and write accesses to I/O registers. A simple backend handles cross-node I/O requests and responses over the two VCs 0 and 1. The DirC module handles the more complicated coherent memory traffic using VCs 2 to 11. We will discuss the coherence messages in some more detail in [subsection 1.4](#). The multiplexed co-processor module allows the FPGA side co-processor to exchange messages with the CPU-side co-processor over VC 12. The multicore debugging and link data module operates on VC 13.

Of particular importance in this work is the throughput at each layer of ECI. Raw bitrate of the physical interconnect is 30GB/s in both directions as depicted in [Figure 1](#). The Interlaken protocol introduces an overhead of about 5%, as discussed above. Thus we get about $\frac{0.95 \cdot 30 \cdot 10^9}{64} \approx 0.45 \cdot 10^9$ blocks per second at the block layer. Because the tracing engine developed in this work runs at the speed of the system clock we are most interested in the rate of arrival with respect to this clock. On the current Enzian system, the FPGA system clock runs at 300MHz, thus the block layer at the FPGA sends and receives just under 1.5 blocks every cycle. Only $\frac{7}{8}$ of each block is VC layer data. As described above the minimum size of a VC layer message is 8 bytes, thus one block can carry at most 7 VC messages which gives an upper bound of $7 \cdot 1.5 = 10.5$ ECI messages per system clock cycle in both directions. At the time of writing, only 12 out of the 24 physical lanes are used. Concretely, the raw bitrate, the number of blocks per cycle, and the number of ECI messages per cycle are all halved with respect to the above numbers. The second set of 12 lanes is currently being integrated as a second link: The whole ECI stack from Interlaken to VC layer is duplicated, providing a second instance of each VC. In this work we explicitly assume that there is only one instance of each VC, however, the proposed solution can be extended in a straight forward way to having two links.

1.4 Enzian coherency

Most of this work is concerned with coherence traffic on VCs 2 to 11 because it has the richest protocol and the largest set of different message types. Thus we now describe the implementation and protocol providing memory coherence in some detail. Despite this focus on coherency VCs, the methodology and the tools developed here work equally well for all other VCs.

The VC backends are responsible for making the FPGA on Enzian look like an ordinary THX NUMA node to the other nodes in the system. In that regard, Enzian can be considered a 2-node NUMA THX

system. One aspect of making the FPGA look like an ordinary THX node is to provide coherent access to its DRAM. The DirC is the VC backend that achieves this.

Coherency between multiple THX nodes is implemented based on directories. Directory-based cache coherence contrasts with snoopy bus protocols. The latter assumes that the caches of all nodes in the system are connected by a single bus. The caches communicate by broadcasting requests, responses and notifications over this bus. The need for a shared bus and broadcasting every transaction to all other caches significantly limit the scalability of this approach. Directory-based caches use point-to-point messages between caches and directories to overcome this limitation. Directories keep track of the contents of the caches in the system to be able to forward requests and notifications to the relevant caches. The number of directories used, their location in the system (central or distributed at every node) and the way in which they store information varies between implementations.

In a THX system, every node has a unified L2 cache and an associated DirC. The DirC of a node maintains an Remote Tag Store (RTG), which is a table of all CLs that are stored in other node's caches but which originate from this node's DRAM. We call the node from whose DRAM a CL originates, the *home* node of that CL. All other nodes are *remote* nodes of that CL. Thus we call the DirC on the home node of a CL the *home DirC* and the DirCs on other nodes the *remote DirCs* of this CL. Similarly, we say a CL is *home* on its home node and *remote* on any remote node. Associated with each CL entry in the RTG is the state of this CL in every node's L2 cache. In Enzian there are only 2 nodes, thus the RTG on the FPGA stores all CLs that are home on the FPGA and that are cached on the CPU. For each such CL it stores the Home State (HS) and the Remote State (RS) of this CL, i.e. the state of this CL in the FPGA's L2 cache and in the CPU's L2 cache. In order to access a remote CL, the remote cache sends a request to the home DirC of the CL (either directly or via its DirC). The home DirC then answers this request by sending back the requested data.

The THX implements a MOESI-based cache coherence protocol, where a cacheline can be in any of the five states modified (M), owned (O), exclusive (E), shared (S), and invalid (I). We will assume the reader is familiar with the meaning of these states and the protocol. In theory there are 12 allowed combinations of HS and RS of which the RTG needs to keep track. However, there are some simplifications that reduce the amount of information the DirC needs to maintain. Because the DirC is responsible for all coherency related communication over ECI, these simplifications also limit the complexity of this communication. The resulting simplified model of coherency on Enzian is a product of previous work in the group¹:

First, we observe that the DirC does not need to distinguish for the HS between M, O, E, and S. It suffices if the DirC knows whether the CL is stored in the local cache or not. The exact state of the CL is kept in the cache. Upon an incoming request for a CL that is cached, the DirC forwards the request to the cache and the cache updates its state accordingly. If the requested CL is not cached, i.e. its HS in the DirC is I, the DirC instead requests the CL from local DRAM. Second, RS can never be O. According to the protocol definition, a remote node becomes the owner of a CL only if it has the CL in exclusive or modified state and then another remote node requests shared access to this cacheline. By definition, this only happens in a system with at least three nodes. Third, the DirC does not need to distinguish for the RS between E and M. One of the motivations of the exclusive state in the MOESI protocol is to allow nodes to silently modify CLs which they have in exclusive state. That is, no ECI message is sent when the remote node transitions from E to M. Thus the DirC can't distinguish between E and M for RS. In conclusion, the DirC only needs to distinguish between I, and {M, O, E, S} for the HS and between I, S, and {M, E} for the RS. The communication between the DirCs of the two endpoints is limited to managing the transitions between these states. All sent messages concern the RS while the HS is updated locally on the home node. The RS states are ordered by freedom of access of the remote node: $I < S < \{M, E\}$. We call a transition from x to y

¹enzian/documentation/eci_working_notes/ECI CC Protocol Model.pdf

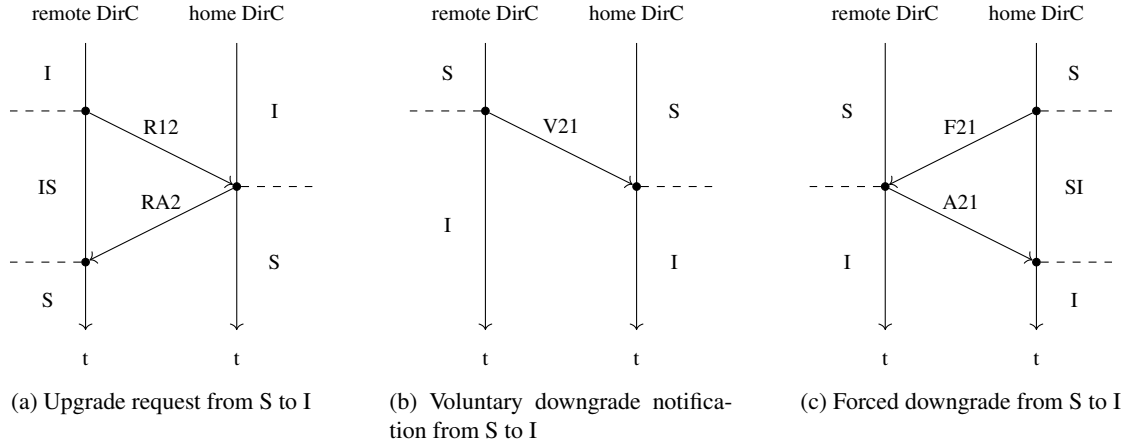


Figure 3: Illustration of request-response pairs and notifications. Downwards arrows symbolize the flow of time at home DirC and the remote cache. Sideways arrows are messages exchanged between home and remote DirC. State annotations on the side indicate the current RS state in the home DirC and in the remote cache in a given time range.

an upgrade if $x < y$ and a downgrade if $x > y$.

The actions a DirC can perform for a given CL depend on whether it is the home or the remote DirC. Concretely, the “action space” of a DirC is the following (with the corresponding message in brackets):

- The remote DirC can request a state upgrade from state x to y (R_{xy})
- The home DirC must acknowledge an upgrade request to state x if asked by the remote DirC (RA_x).
- The home DirC can ask the remote DirC to downgrade its access from state x to y (F_{xy})
- The remote DirC must acknowledge a downgrade request from state x to y if asked by the home DirC (A_{xy}).
- The remote DirC can voluntarily downgrade its access from state x to y (V_{xy})

Here x and y correspond to the remote states I , S , and $\{M, E\}$ discussed before, but we often write 1 for I , 2 for S and 3 for $\{M, E\}$. Note that R_{xy} and RA_y are a request-response-pair: An upgrade request R_{xy} from the remote DirC must be acknowledged by the home DirC with an upgrade acknowledge RA_y . Similarly, F_{xy} and A_{xy} are a request-response-pair: A downgrade request F_{xy} from the home DirC must be acknowledged by the remote DirC with a downgrade acknowledge A_{xy} . A voluntary downgrade, on the other hand, is a standalone notification from the remote DirC to the home DirC that doesn’t require an acknowledge.

The behaviour of the two nodes for each of these request-response pairs and for the voluntary downgrade notification is illustrated in [Figure 3](#).

In [Figure 3a](#), the remote cache wants to upgrade its access to a CL from I to S . The remote DirC thus sends an $R12$ request to the home DirC and waits for a response. While waiting for a response, the remote cache is in transient state IS and can’t perform another action on this CL. Upon receiving the request, the home DirC immediately updates its state record of the CL in the RTG to S and sends the response $RA2$. This may require an update of the HS and a request to the local cache. These local actions are not shown

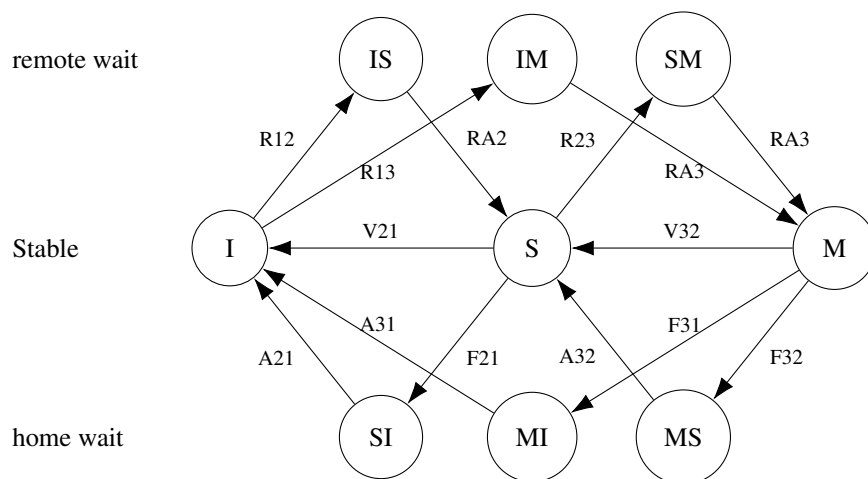


Figure 4: State transitions and message exchange between the two DirCs

in Figure 3 because they are not visible on ECI and thus irrelevant to this work. Upon receiving the RA2 response, the remote cache stores the received data and updates to state S. A very similar sequence happens when the home DirC forces the remote cache to downgrade its access, as illustrated in Figure 3c. On voluntary downgrades, which are simply notifications, neither remote cache nor home DirC transition to a transient state. Instead the remote cache transitions into the goal state immediately and the home DirC does so as well upon receiving the notification as illustrated in Figure 3b. All basic request-response transaction and notifications are visualized more compactly in Figure 4. In reality, the protocol also needs to handle some conflict scenarios that happen due to message re-orderings over ECI or because both nodes send a request simultaneously. We will talk a bit more about these conflict scenarios in subsection 4.3 but for now it suffices to consider the protocol without these conflict scenarios.

The correspondence between the above request, response, and notification messages and the actual ECI messages is given in Table 2. There are several things to note: The coherency VCs are partitioned in two ways: First, *even-numbered* VCs (2, 4, 6, 8, 10) handle messages concerning *even* CL addresses, *odd-numbered* VCs (3, 5, 7, 9, 11) handle messages concerning odd CL addresses. Second, VCs 2 to 5 handle messages that contain CL-data as payload while VCs 6 to 11 handle header-only messages. Some messages optionally have a payload, for instance V31. Whether it has a payload or not depends on whether the CL was dirty and needs to be flushed. Further note that there are two more downgrade responses than illustrated in Figure 4, namely A22 and A11. These messages exist because messages from home and remote DirC may cross on the way and because messages sent from one node may be received out of order at the other node. A11, for instance, is sent by remote when a voluntary downgrade V21 from the remote and a downgrade request F21 from the home have crossed on the way. The remote now sends A11 to clarify that it already is in state I and no longer in state S.

There are more ECI messages that can be sent over the coherency VCs 2 to 11 than listed in Table 2. A full list can be found in the previous work [4]. In this work we focus on the subset discussed above because it largely corresponds to the subset that is currently implemented in the FPGA DirC.

ECI message	VCs	Interpretation
MREQ_RLDI	6,7	R12
MREQ_RLDD, MREQ_RLDX	6,7	R13
MREQ_RC2D_S	6,7	R23
MRSP_VICS	10,11	V21
MRSP_VICC	4,5,10,11	V32
MRSP_VICD	4,5,10,11	V31
MRSP_VICDHI	4,5,10,11	A31
MRSP_HAKD	10,11	A21
MRSP_HAKN_S	4,5,10,11	A32
MRSP_HAKS	10,11	A22
MRSP_HAKV, MRSP_HAKI	10,11	A11
MRSP_PSHA	4,5	RA2
MRSP_PEMD	4,5,10,11	RA3
MFWD_SINV_H	8,9	F21
MFWD_FEVX_EH	8,9	F31
MFWD_FLDRS_2H.E	8,9	F32

Table 2: Interpretation of ECI messages in terms of DirC requests, responses, and notifications

1.5 NFAs

Deterministic Finite Automata (DFAs) are a fundamental mathematical formalism in the area of formal languages. A formal language over an alphabet Σ , is some set of words (finite length strings) consisting of the symbols in Σ . A DFA is defined by a set of states S , an input alphabet Σ , a transition function $\delta : S \times \Sigma \mapsto S$, an initial state $s_0 \in S$ and a set of accepting states $F \subseteq S$. Given some word $\sigma_1\sigma_2 \dots \sigma_n$ over Σ , a DFA either accepts or rejects this word. A DFA accepts or rejects a word as follows: Initially, the state of the DFA is s_0 . Upon arrival of input σ_i , the new active state $s_i = \delta(s_{i-1}, \sigma_i)$ is computed. If s_n is one of the accepting states in F , the word $\sigma_1\sigma_2 \dots \sigma_n$ is accepted, else it is rejected. A DFA thus defines a language over the alphabet Σ , consisting of all words over Σ that it accepts.

A concept that is closely related to DFAs are NFAs. An NFA differs from a DFA in two ways: First, an NFA allows a single transition to activate multiple states. Second, an NFA allows ε -transitions between states. An ε -transition from state s_1 to s_2 will cause s_2 to be activated whenever s_1 is active. The activation of s_2 happens simultaneously with the activation of s_1 . The effect of these two generalizations is that in NFAs multiple states can be active at a time. It can be shown that NFAs are equally expressive as DFAs, despite the above generalizations, i.e. any language that can be expressed with DFA or NFA can also be expressed with the other.

An example of an NFA (that is also a DFA in this case), is illustrated on the left in [Figure 5](#). The graphical depiction in [Figure 5](#) corresponds to the NFA with

$$\Sigma = \{a, b\} \tag{1}$$

$$S = \{s_0, s_1, s_2\} \tag{2}$$

$$\delta = \{(s_0, a) \rightarrow \{s_1\}, (s_0, b) \rightarrow \{s_0\}, (s_1, a) \rightarrow \{s_0\}, (s_1, b) \rightarrow \{s_1\}, (s_2, a) \rightarrow \{s_2\}, (s_2, b) \rightarrow \{s_1\}\} \tag{3}$$

$$s_0 = s_0 \tag{4}$$

$$F = \{s_2\} \tag{5}$$

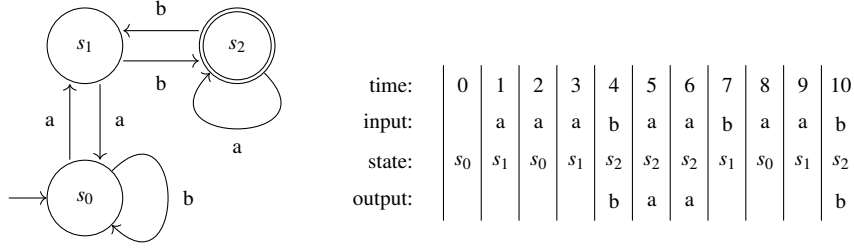


Figure 5: Filter semantics for an example NFA. The NFA is graphically depicted on the left. Input, active state and filtered output are listed ordered by time on the right.

In the graphical representation, we indicate each state with a circle. The transition function δ is depicted as labeled arrows between states: An arrow from s_i to s_j labeled by x means $s_j \in \delta(s_i, x)$. Note that, because this is an NFA, the transition function maps a single state-input pair to a set of successor states. We indicate the starting state s_0 by an incoming arrow that originates from no other state and the accepting states by a second smaller circle nested within the state circle. The alphabet Σ is implicit in the set of all edge labels in graphical representation.

In this work we will often think of NFAs as labelled digraphs. Let $A := (S, \Sigma, \delta, S_0, F)$ be an NFA. The corresponding graph $G_A = (V, E)$ has vertices $V = S$ and labeled arcs E with

$$(s_i, s_j, a) \in E \quad \text{if } s_i, s_j \in S, a \in \Sigma, s_j \in \delta(s_i, a) \quad (6)$$

We additionally require two vertex label functions $\text{is_starting}(s) = s \in S_0$ and $\text{is_accepting}(s) = s \in F$. We often call the edge label of a transition the *trigger* of the transition.

We choose NFAs both as the formalism and as the way of implementing our trace filters. We choose NFAs over DFAs because they are more amenable to implementation in hardware due to the following trade-offs [5]:

- An NFA has at most as many (and usually many less) states as an equivalent DFA, thus implementing NFAs in hardware requires less resources.
- A DFA state update can be done with constant *work* because there is only one active state at any time. In an NFA, all states can be active at the same time, thus updating all states requires *work* linear in the number of states. However, since these updates are processed on an FPGA in parallel hardware, the *time* required to perform a state update is equal in both DFA and NFA.

In our use case we choose Σ as the set of valid VC-layer messages. A difference to the above definition is that we are not observing finite-length words over Σ . Instead, we observe a continuous stream of VC-layer messages. The exact semantics of our filter thus diverts a bit from the standard definition given above. We filter a continuous *stream* (m_1, m_2, \dots) of VC-layer messages $m_i \in \Sigma$ using an NFA as follows: When activating the tracing engine, the NFA is reset the initial state set S_0 to $\{s_0\}$, that is the set containing only the initial state. Upon receiving an ECI message m_i , we update

$$S_i = \bigcup_{s \in S_{i-1}} \delta(s, m_i) \quad (7)$$

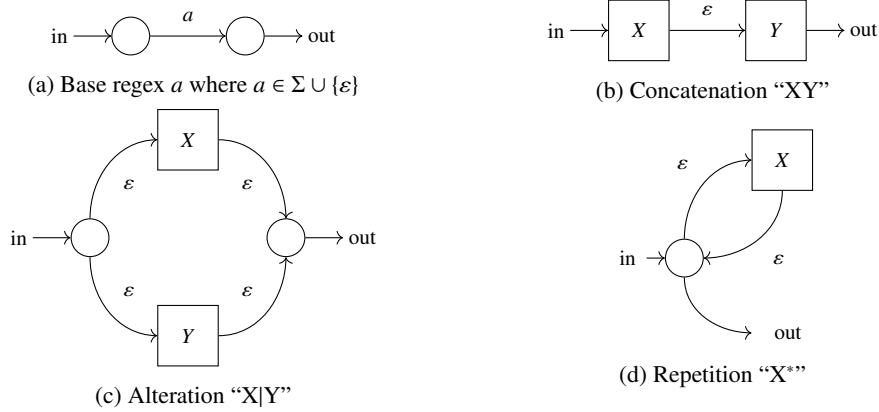


Figure 6: Recursive construction of an NFA to match a regex

If S_i contains an accepting state, i.e. if $S_i \cap F \neq \emptyset$, we output message m_i . In this case, we say message m_i is *accepted* by the NFA. Formally, a message m_i in the output thus indicates that the stream prefix $(m_0 m_1 \dots m_i)$ is a word in the language defined by the NFA. Every accepted message is thus a witness of a word in the language defined by the NFA. This is an important difference to outputting accepted words: by outputting accepted words we would not achieve a filtering of the message stream, instead we would produce a stream of prefixes of the original stream.

The process of filtering an input stream with the NFA given by Equation 1 to Equation 5 is illustrated with an example on the right in Figure 5: The initial state of the NFA at time 0 is s_0 . The input stream prefix “aaabaabaab” is consumed in consecutive timesteps 1 to 10. The NFA state is updated according to the NFA state transition function. Because there is only ever one state active, we omit the set notation for the active states. The filtered output contains exactly the input symbols “baab”, i.e. the substream consisting of the symbols that lead to the accepting state s_2 of the NFA.

Regular Expressions (RegExs) are a convenient way of defining regular languages over some alphabet Σ . Because every regular language corresponds to an NFA, RegExs also allow us to define NFAs. In this work we will use the formalism of RegExs to define filters. There are some other formalisms that could be used, such as LTL-formulas [6], but we find RegExs more intuitive for what we are trying to achieve. The set of valid RegExs over alphabet Σ is defined recursively as follows: (0) The special symbol $\varepsilon \notin \Sigma$ is a valid RegEx over any alphabet Σ . If Σ also contains the symbol ε we can rename it without loss of generality. (1) Every symbol $s \in \Sigma$ is a valid RegEx. (2) If p and q are valid RegExs over Σ , then pq and $p|q$ are also valid RegExs over Σ (3) If p is a valid RegEx over Σ , then p^* is also a valid regex over Σ . The basic RegEx ε corresponds to the language consisting only of the empty word. The basic RegEx s for $s \in S$ corresponds to the language $\{s\}$ consisting only of one word, namely the single symbol s . The concatenation pq of p and q is the set of words $w_1 w_2$ where w_1 is in the language defined by p and w_2 is in the language defined by q . The alternation $p|q$ of p and q is the set of words w that are either in the language defined by p or in the language defined by q . The repetition p^* of p is the set of words $w = w_1 \dots w_n$ where w_i is in the language defined by p for all $i \in \{1, \dots, n\}$. This definition of p^* includes the empty word, in that case $n = 0$.

Because RegExs and NFAs are equivalent, there is for each NFA a corresponding RegEx that defines the same language and vice versa. Figure 6 shows how we can construct an NFA corresponding to any RegEx. The construction recurses over all the constructors for RegExs. For each construction, *in* and *out* edges indicate how to combine the NFAs for sub-RegExs to a bigger NFA for the combined RegEx.

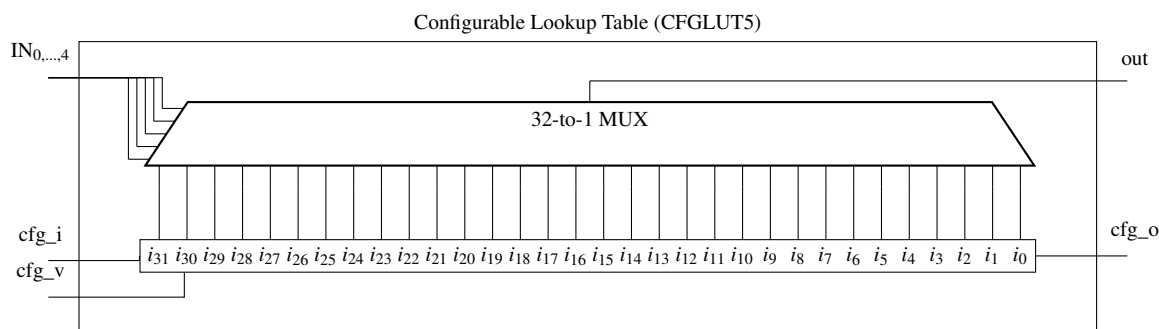


Figure 7: Conceptual implementation of a CFGLUT5 primitive

1.6 FPGA, Xilinx and Verilog

We implement the tracing engine on a Xilinx Virtex Ultrascale+ FPGA. FPGA programming is inherently different from programming a CPU. The goal of this section is to provide some background on programming an Ultrascale+ architecture FPGA and to motivate some of the design goals we set in [subsection 1.1](#).

An FPGA provides configurable hardware elements arranged in a two-dimensional grid with configurable interconnections between them. There are several types of basic hardware building blocks, ranging from configurable logic functions, over several types of memory cells to specialized I/O blocks. In this work we mostly deal with configurable logic blocks (CLBs) as they are the main building block for implementing general-purpose combinatorial and sequential circuits [7]. Every CLB contains eight 6-input Look-Up Tables (LUTs) and sixteen Flip-Flops (FFs). Each LUT can implement an arbitrary 6-to-1 logic function and every FF can store one bit of data. The LUTs within a CLB are connected such as to allow building up more complicated logic functions. For instance, internal multiplexers allow choosing the output of one of the eight LUTs, thereby allowing to define an arbitrary 9-to-1 logic function. The FFs allow implementing sequential logic by synchronizing LUT-outputs on a clock edge. The XCVU9P FPGA we are using has about 1.1M LUTs and about 2.3M FFs. The configurable interconnection network is layed out as a hierarchical grid. Signals that need to be transmitted between hardware building blocks need to be routed across this hierarchical grid. One crucial signal is the clock that drives synchronized elements. If many hardware blocks in distant locations require the same clock for synchronization, this can lead to clock skew and timing issues.

These standard LUTs are configured at compile time by loading a bitstream onto the FPGA. However, the Ultrascale+ architecture also provides a CFGLUT5 primitive that can be re-configured while the FPGA is running. [Figure 7](#) shows the logic behaviour of a CFGLUT5 primitive: A 32-bit config string can be pushed into a CFGLUT5 in 32 clock cycles by pulling the valid bit (`cfg_v`) high and by setting the config input (`cfg_i`) as desired. Pushing more than 32 bits into the config will cause the “oldest” config bits to be shifted out at the `cfg_o` port. Multiple CFGLUT5 elements can be configured in sequence by connecting the config output of one to the config input of the next. The 5-bit input value given by inputs `IN0` to `IN4` set the output (`out`) to the corresponding bit of the currently loaded config string. The loading of a configstring is synchronized by a clock (not shown in graphic), but the input-output behaviour of the CFGLUT5 is combinatorial. A big part of the runtime configurability of the tracing engine will be achieved using CFGLUT5s.

The Vivado design suite [8] takes a high-level description of a hardware design in the form of Hardware Description Language (HDL) code which is then synthesized into a configuration bitstream for the FPGA. To make the FPGA implement the functionality described in the code, combinatorial logic is translated into configurations of specific LUTs and registers are mapped to FFs on the FPGA. In a second step, the routing

between all basic hardware elements is computed. The problem of mapping sequential logic to the basic building blocks on the FPGA and computing the correct routing between them is called Place&Route. One of the core constraints on the solution found by the Place&Route algorithm is that no timing constraints are violated. Roughly, a timing violation exists if the propagation of the signal between two connected registers takes longer than one clock cycle. The time it takes to propagate a signal between two registers (propagation delay) depends on the routing distance between the two registers and the amount of combinatorial logic on the path. The critical path of a design is the path between any two registers that has the longest propagation delay. The critical path thus limits the clock rate of the design. In other words, since the clock in our system is fixed at 300MHz, we are limited in the amount of combinatorial logic we can perform between any two registers. Furthermore, because the amount of available hardware resources (LUTs and FFs) is limited, there is only a restricted amount of logic that can be implemented simultaneously on an FPGA. We set the goal that our tracing design can run alongside major applications on the FPGA. Thus we need to aim for a compact design that doesn't use many resources.

Place&Route constitutes a very hard computational problem that can take a very long time to solve (around one hour for the current Enzian project). This long synthesis time is the major reason why we set the goal of runtime configurability of the tracing engine. We want to be able to apply multiple filters to the ECI message stream without re-synthesizing the FPGA or even powering it down.

There are several properties of a design that make it difficult to Place&Route: First, if there is a lot of combinatorial logic between two registers, these registers need to be placed very close to each other. The combinatorial logic between any two registers can be limited by introducing intermediate registers (pipelining). Second, if the output of some register is used in many different places it will be hard to place all the consumers of the signal close to the producer. If a lot of data needs to be distributed to many different places, this may lead to timing problems. Both of these are problems that we will need to deal with in our design.

1.7 Use cases of tracing VC layer messages

In this section we want to consider some possible use cases of the tracing engine in order to motivate the design choices in [section 3](#).

First we need to consider some basic limitations of the tracing engine:

- The tracing engine only observes ECI messages. In particular, it doesn't have complete knowledge of the actions of the DirC. For instance, it can't observe local events such as DRAM reads/writes.
- The tracing engine sits before the arbiter on both sending and receiving paths. The arbiter schedules the order in which the DirC consumes ECI messages on the receiving path and it schedules the order in which ECI messages are sent over the block layer on the sending path. Thus the tracing engine has imperfect information about the order in which messages are received by either FPGA-side or CPU-side DirC. (The integration of the tracing engine into the ECI stack is visualized later in [Figure 10](#).)

Keeping these limitations in mind, we will now discuss three scenarios in which the tracing engine could provide us with valuable insights. We will concretize each of these use cases in [section 4](#).

Simple message filters Many interesting properties of the system can be measured by looking only at traces. For instance, we could analyze the number of in-flight read requests that are sent by the CPU when observing a performance bug. Or we could determine the average amount of time between two consecutive

requests or responses. Both measurements are quite simple to derive from VC layer traces but would be hard to get without. Research along these lines has already been done in previous work [4].

A major problem in [4] is the size of the produced traces. As discussed in [subsection 1.3](#), the block layer maintains a constant stream of blocks. If there is no data to send, idle blocks are sent. This results in huge amounts of redundant data in unfiltered traces. This problem is less severe on the VC layer because idle blocks are discarded at the block layer. However, there are many VCs and message types and we might not be interested in observing messages on all of them. For instance, when observing coherence traffic, we might not be interested in messages on the I/O VCs. Similarly, we might not be interested in observing the frequent TLB shutdown broadcast messages and their responses. Simple filters, implemented as NFAs, allow us to ignore specific message types and entire VCs. We will consider a concrete use case of simple message filters in [subsection 4.1](#).

Safety properties Building Enzian entails the task of implementing cache coherency on the FPGA in a way that is compatible with how it is implemented on the THX CPU. Cache coherency protocols are usually not designed with interoperability in mind: In general, a CPU of a given vendor will only ever communicate with architecturally equal CPUs. This makes the task of implementing a protocol that cooperates with the THX coherence protocol both novel and challenging. Moreover, a goal of Enzian is to experiment with different ways of implementing this cache coherency on the FPGA. Again, all implementations must adhere to the protocol expected by the THX. In general, the THX implements the MOESI protocol as discussed in [subsection 1.4](#). Using a VC layer tracing tool can be very helpful, both to verify the correctness of an implementation and to debug a faulty implementation once an error has been detected.

All ECI transactions are either request-response pairs or single messages (notifications). For a request-response pair we can be interested in two types of properties

- Liveness properties: For each request, there must be a valid response
- Safety properties: For each request, there is no invalid response. Or: There is no invalid request.

While liveness properties can't be checked in finite time, we can construct an NFA that finds all violations of a safety property. Moreover, we can formulate safety properties for sequences of request-response pairs and notifications.

The distinction between debugging a faulty implementation and verifying the correctness of an implementation is subtle: In the former case we formulate a safety property that characterizes the bug we are observing. In the latter case we derive the safety property directly from the MOESI state machine. In both we formulate an NFA that filters the trace for instances of violations of the safety property and output a window of messages around this violation. In the validation scenario, an output is a witness of a violation of the safety property. In the debugging scenario the output is used to understand how the faulty behaviour came to be.

Consider the following example of a safety property:

P_1 : A node can't request an upgrade to level 3 for some CL (exclusive access) while such a request is already in-flight.

[Figure 8](#) shows the corresponding NFA. One problem we are facing is that the tracing engine sees messages in a different order than both DirC and CPU. We will revisit the question of what kind of safety properties can be expressed in [subsection 4.3](#).

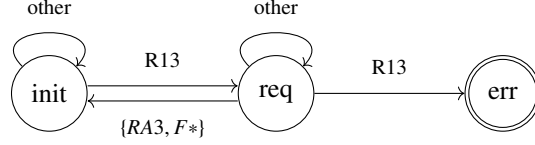


Figure 8: NFA accepting when P_1 is violated

Performance and inspection properties The tracing engine provides us with the unique opportunity to observe cache coherence messages. Information about memory traffic can be very useful when trying to understand performance characteristics of some application. This is even more the case in a heterogeneous platform such as Enzian. On such platforms, one of the hardest questions is to design and evaluate offloading strategies, i.e. which bits of a computation should be performed on which node. An great limiter for accelerator offloading is that it’s hard to find good chunks of work to send to the accelerator.

We can use the tracing engine in conjunction with some simple NFAs to characterize cache traffic and analyze the resulting effect on performance. We will discuss this use case in detail in [subsection 4.2](#).

1.8 Overview

In the remainder of this thesis we will describe and evaluate the design of a tracing engine for the VC layer ECI communication on the Enzian system. The tracing engine allows the user to configure generic trace filters at runtime using a simple filter language. It filters all coherence traffic of the Enzian at line rate according to the provided filter and writes the filtered data to a remote host for post-processing. The tracing engine has a small hardware cost and is mostly independent of the rest of the system. This allows running the tracing engine alongside major applications on the FPGA and to use it in future iterations of the Enzian system.

Our design is inspired by the previous work [9]. The original contributions of this work with respect to [9] are the following

- Our design scales to several hundred states despite the fast 300MHz system clock and at a small hardware cost. These larger sizes provide more expressivity to accomodate filters for the richer VC layer protocols.
- We explore several possible ways of achieving input data reduction that faithfully represents the large set of VC layer messages and lets us express a wide variety of filters.
- Messages on the coherency VCs are naturally partitioned into logical substreams by CL: the messages for each CL are independent of the messages to other CLs. We provide support for filtering these logical substreams individually.
- In a debugging scenario, one is often interested in inspecting a window of messages around one message that indicates the bug. We thus implement support for windowing.
- We provide a modular design with a strict separation of concerns: data reduction, input decoding and NFA operation are strictly separated in modules which allows easy adaptation to different VCs, different interfaces, and different filters.
- We demonstrate the usefulness of the resulting design by inspecting several interesting properties of the Enzian system and by applying it to both the VC and the block layer.

The remainder of this thesis is structured as follows. In [section 2](#) we first review some previous work in the broader area of Complex Event Processing (CEP). Despite the large variety of contexts in which CEP is applied, many of these works use similar techniques to us and provided inspiration for this work.

The design in this work consists of two fundamental components, the tracing engine hardware module running on the FPGA, and the frontend that allows controlling and configuring the tracing engine. In [section 3](#) we will describe in detail both components of this design. Next, in [section 4](#), we will perform a series of experiments to demonstrate the versatility of the design.

In [section 5](#) we evaluate various properties of the design. We will evaluate its hardware cost, the expressiveness it provides and trade-off some of the design decisions we have made. Finally, in [section 6](#), we will discuss some limitations and possible extensions of the design.

2 Related Work

Expressed in full generality, the tools developed in this work solve the following problem

Consume a continuous stream of data, perform an online search for a “pattern” and report when this pattern is detected.

Many previous works have investigated a similar question in a plethora of different settings. Often these works are collectively referred to as Complex Event Processing (CEP). We will now consider three different instances of CEP to give the reader an idea of the breadth of the topic.

A common application of CEP is network intrusion detection. The well known Snort tool [10] defines a large set of rules that allow detecting suspicious network traffic and allows users to extend this set of rules with their own rules. Rules can refer to and specify the values of various header fields of network packets at all layers. Moreover, Snort can inspect the payload of network packets and perform string search to detect suspicious content. Each rule also specifies an action that is taken if some network packet matches the rule.

A very similar approach is discussed in [11] in the context of high-frequency trading. The authors apply CEP to process stock-market events. Abstractly, a market event is a tuple containing a stock name, the current ask/bid price and potentially other information. Similar to Snort rules, users of the CEP engine can post subscriptions to certain events by matching on these tuple fields and partially specifying their values (e.g. specify a lower bound for the bid price of a fixed stock). The CEP engine checks all incoming events for matching subscriptions and performs a specified action, such as buying the stock, whenever a match is found.

A conceptual limitation of both these applications is that they focus on events that are atomic in nature: In both cases, the rules only refer to a single “message”, i.e. a network packet in [10] and a market event in [11].

Other works consider the detection of complex, or compound events that are composed of multiple simpler events. TESLA [12] is a formally specified language to express such complex events. In particular, TESLA is not a system built for CEP in a fixed domain, instead it provides a language of operators with exact semantics and shows how complex events can be built from simple events using these operators. One important class of operators allows specifying a complex event as a sequence of simpler events. The temporal relation between sequences of events can be specified using time units like seconds. A core idea of TESLA is that events are a first class object, that is, events can be used in the definition of other events, thus forming a hierarchy of events.

A concrete architecture for detecting composed events is described by Teubner et al. [13]. The authors use XML projection as an example application to demonstrate their approach. In XML projection, a path

specifier is provided by the user. A constant stream of XML is processed and sub-trees that match the path are output. The XML is consumed byte-by-byte, matching a certain XML tag constitutes a basic event and matching a full sub-tree according to a sequence of tags constitutes a composed event. The patterns are thus composed of multiple smaller events and the CEP engine needs to keep state in order to match a full event.

While the exact meaning of “event” varies a lot between different applications of CEP, many systems use NFAs at their core to detect events. For instance, Snort uses NFAs to perform string search in the payload of network packets, while the XML filter in [13] uses NFAs to check for the right sequence of tags. Early CEP systems, such as Snort [10], often used software implementations of NFAs and ran on general purpose CPUs. Inherent in most CEP systems is the need to keep up with the data source. Thus the evolution of network bandwidth poses a significant problem to these software approaches. A second property of many CEP systems, is that the ruleset changes over time. For instance, the Snort ruleset grows continuously, moreover, users are allowed to add custom rules. This combined need for reconfigurability and high throughput prompted many innovative designs for CEP and thus for NFA processing in general. Next, we will review some designs that influenced this work.

Woods et al. present an FPGA-based architecture that significantly speeds up Snort rule checking [14]. Each rule is compiled to an NFA and then implemented in the configurable hardware of an FPGA. The data, arriving over the FPGA’s network interface, is fed through each of these NFAs in parallel before being sent to the CPU. A matching NFA indicates a rule match, which is reported to the CPU. Offloading the rule checking from the CPU to an FPGA significantly reduces the load on the CPU. The reconfigurable nature of the FPGA allows updating the architecture whenever the Snort ruleset is updated.

Relying of FPGA reconfigurability to update the ruleset has two major downsides: First, re-synthesis of the FPGA often takes significant amounts of time. During this time, network intrusion detection can’t be performed which limits the usability of the architecture. Second, the approach can not be used on Application-Specific Integrated Circuits (ASICs) because updating the ruleset relies on the reconfigurable nature of FPGAs. ASICs, however, offer better performance than FPGAs, thus it may be desirable to provide an approach that can be used on both types of hardware.

Tan et al. thus suggest a different architecture for Snort rule checking with a different way of reconfiguring the ruleset that overcomes these downsides in [15]. Concretely, they design a hardware architecture for DFA processing on an integrated circuit using SDRAM technology: The current state of the DFA corresponds to the row-address, the input symbol to the column address into the DRAM array. The DRAM entry at the position indicated by state and input contains the next state. The implementation reads the input symbol-by-symbol while continuously updating the current state of the DFA. The automata used in this work are not NFAs but Aho-Corasick automata [16], which are specialized DFAs for exact string matching. This is important because the above implementation only allows one state to be active at any time. Moreover, this design is not suitable for DFAs with high fanout (i.e. with a large input alphabet) because every out-transition of a state consumes DRAM resources and increases the next-state lookup time. The authors thus introduce the concept of a bit-split automaton that reduces the size of the input alphabet by considering each bit of a symbol separately. This effectively reduces the fan-out of each state to 2. The original DFA is then implemented by a series of eight smaller bit-split automata (one per bit in the original 8-bit input) that work in lock-step. The architecture allows reconfiguration by overwriting the memory holding the state-transition information. Non-interruptive updates are supported by writing a new state-transition table into a fresh location first, then “atomically” swapping from the original to the new state-transition table.

A disadvantage of this design is the limitation to DFAs. As discussed in [subsection 1.5](#), DFAs can have exponentially more states than an equivalent NFA.

Micron [17] takes a similar approach to Tan et al. that achieves reconfigurable NFA processing in hardware. Micron’s automata processor (AP) is a processing-in-memory architecture that allows processing

arbitrary NFAs. In particular, the AP wasn't built with a particular use case in mind but was designed to accommodate all kinds of NFAs. The AP is based on an adaptation of the DRAM array architecture: A DRAM column corresponds to one State Transition Element (STE) that represents one state of the NFA. Each STE stores one bit corresponding to each input that indicates if this input can activate the state or not. Additionally, each STE is statically connected to several other STEs. Both the input activation and this routing can be configured at runtime. In particular, the routing to some connected STE can be turned off if this STE is not a neighbor of the state represented in this STE. The input symbol of the NFA is interpreted as row-index into the DRAM array. Similar to DRAM operation, the arrival of an NFA input (a row-index), causes the propagation of high voltage along the *word line* to every STE. Similar to DRAM, voltage on the word line causes the propagation of the 1-bit trigger symbol stored in the corresponding cell of each STE. The STE then computes its state based on this trigger bit and the input from other STEs. Note how this approach enables processing both NFAs and DFAs: Multiple states can be active simultaneously because each STE stores its activation locally. Moreover, one STE can have multiple active out-transitions at once, one for each connected STE. A potential limitation of this architecture is that all transitions entering a state must occur on the same input symbols. However, this does not impair expressivity of the representable NFAs: States that activate on multiple different inputs simply need to be spread over multiple STEs. Another difference to Tan et al. is that the predecessor relationship is not encoded in the DRAM memory cells themselves, but an explicit routing between STEs connects neighboring states. This enables each STE to activate independently which is key to implementing NFAs updates in constant time (c.f. [subsection 1.5](#)). A downside of this explicit wiring is that the connectivity and fan-out of each STE is fundamentally limited by the routing hardware provided. The AP provides a hierarchical routing network that aims to provide as much flexibility as possible while still being economical. In general, all states can have fan-out of at most 16. But a limited set of STEs can have a much higher fan-out of up to 2304. The first implementation discussed in [17] is based on DDR3 SDRAM technology and can process 8-bit input symbols at 1Gb/s per chip. Similar to normal DRAM, AP chips can be organized in a rank of multiple chips and provide higher bandwidth. Thereby, each chip processes 1 byte of the input and the chips operate in lock-step similar to Tan et al.

Despite the generality of the Micron AP, it is not suitable for our problem because we require an FPGA implementation that can be hosted on Enzian itself and that doesn't require a separate chip.

Napoly [18] is an FPGA-based implementation of the AP architecture. While the implemented design is conceptually very similar to the AP, the authors have to make various compromises due to their use of a different hardware platform. Napoly succeeds in packing 24k STEs into one FPGA, which is about 50% of Micron's AP. Instead of using the DRAM array structure to store the state transitions of each STE, Napoly uses LUT-based memory to store this information. This LUT usage competes with the resources needed to implement the actual logic. More importantly, the hierarchical routing network of the AP is custom designed and optimized to provide flexibility in placing states to the STEs. The FPGA on the other hand has fixed routing resources and packing a large portion of it with STEs fundamentally limits the connectivity of the individual STE. The authors thus use an interconnect that only connects STEs that are close to each other on the FPGA substrate. Fan-out of any STE is limited to 32. The processing model of Napoly proceeds in rounds: At most 128kB of input data is first loaded into on-chip RAM of the FPGA. Next, the configuration of each STE is loaded into LUT-based memory and registers. This configuration takes place in every round, even if the same NFA is loaded. After configuration, the input is streamed symbol-by-symbol through the NFA. Any accepting states will report their status, this is aggregated and accepting state and current cycle are written into RAM. Finally, the acceptance reports of each cycle are streamed out to the remote host. This round-based processing limits the bandwidth of Napoly because configuration and running of the NFA cannot happen concurrently. Moreover, all streaming on and off the FPGA uses the JTAG interface which is slow. As a consequence, the bandwidth of Napoly is only 90MB/s.

As discussed in [subsection 1.3](#), ECI bandwidth is about 25GiB/s, thus we require significantly more bandwidth than provided by Napoly. Moreover, we wish to provide more flexibility with respect to fan-out of STEs. Many of the restrictions in the design of Napoly come from the fact that as many STEs as possible are packed onto one FPGA. Recall that we don't wish to pack the whole FPGA with the tracing engine. On the contrary, we aim to use as little resources as possible.

A work that has similar priorities to us, and that precedes both Micron's AP and Napoly, is Teubner et al. [13]. The authors introduce the notion of skeleton automaton in the context of XML filtering: The generic structure (skeleton) of an NFA is compiled and programmed onto an FPGA while the configuration of the NFA is configurable at runtime. The generic structure consists of an unconfigured STEs that are connected to some graph structure. (In this work, we often refer to the skeleton as the overlay graph of the NFA.) Conceptually, the skeleton automaton is similar to both Napoly and the Micron AP. A major difference is in goals: Micron and Napoly both try to provide as much connectivity as possible between STEs in order to accommodate arbitrary, user-provided NFAs. On the other hand, the authors of [13] have a concrete query language (XPath) in mind and provision the skeleton of an NFA that can accommodate any valid XPath query up to a given size. The design is optimized for full line-rate streaming where XML data arrives over ethernet. Due to the linear/hierarchical nature of the query language, the skeleton automaton has the static structure of a path. The design uses a simple predicate decoder that is included in each STE to detect tag names. Every STE only needs to recognize one tag. To do this, a predicate decoder performs a string comparison between arriving tag and its goal tag stored in memory. Each STE receives the full input at 1 byte per cycle. Additionally it receives the output state of its sole predecessor STE. Pipelining of the segments is used to increase clock rate. Because processing is strictly linear, both input and accept signal can be registered between consecutive STEs which limits the length of the critical path. The design achieves a maximum bandwidth of 180MB/s. The bandwidth is limited by the 180MHz clock rate of the FPGA and the fact that the design processes one byte per cycle. A serializer receives the accept state of the last STE and outputs the projected XML data, i.e. it outputs the incoming XML stream data whenever the final accepting state is 1.

Kuchler [9] has applied a similar approach as [13] to filtering ECI block layer traffic. The author starts from properties of the block layer protocol to derive the general shape of NFAs that express useful filters of this protocol. However, the block layer protocol doesn't lend itself as well as the XPath query language to deriving the general shape of an NFA. The author thus makes the compromise of providing quite a general skeleton that is still sparse. We will discuss this graph later in this work because we use the same skeleton. Another difference to the works discussed above is the use of centralized data reduction before sending data to the NFA: Interesting information from the block layer headers is extracted and aggregated in a fixed format. This input is smaller than the original block layer message and thus requires less resources to be routed to the STEs. The STEs then individually check for some predicate on this reduced input. There are two major reasons why this approach is desirable: First, the payload of a block is not of interest for the NFA and can thus be discarded. Second, the header of a block contains fixed header fields that can easily be extracted.

In this work we build on the previous work [9] to provide trace filtering at the VC layer of the ECI stack. However, there are several reasons why the previous work cannot be used directly to filter VC layer traffic: First, there are many more types of messages on the VC layer than on the block layer which makes it more challenging to perform useful input reduction. Second, there exists a multitude of protocols on the VC layer, as discussed in [subsection 1.3](#). We will focus on the coherence protocol, but in order to make the design useful in a wider context, we aim for modularity in the design that makes it easy to adapt the tracing engine to a variety of protocols. Third, block layer data arrives at a regular rate of two blocks per cycle. That is, to filter block layer data, a constant rate of two headers per cycle needs to be processed. The seven data words of each block, however, can carry up to seven VC headers. Potentially all of these headers could belong to

the same VC, or all could belong to different VCs. This variability makes it much harder to provide a design that doesn't waste hardware resources.

3 Design

In this section we will discuss the design of the tracing engine in detail. We will first discuss the basic ideas of implementing NFAs on an FPGA. Next, we consider once more the VC layer of ECI to get an idea of the data that needs to be processed by the tracing engine. Given the characteristics of this data source, we discuss three possible ways of pre-processing the input before sending it through the NFA and choose the most suitable option. Next, in [subsection 3.3](#), we go in detail over the HDL design of the tracing engine. In this section we will also decide on a language for filters, that is, we will define the basic predicates that can be used to filter the VC layer messages. The presented design is both a finished tracing engine that can be used to filter VC layer traffic, and a template from which tracing engines for different use cases and with different filter languages can be built. After that, we will discuss the frontend of the tracing engine that compiles a high-level description of a trace filter into a runtime configuration of the tracing engine. Finally, we discuss how the tracing engine is connected to and controlled from a remote host over PCIe.

3.1 Design overview

We first discuss the fundamental ideas behind implementing NFAs in hardware. We start with the definition of homogeneous transitions:

The transitions into some state s are homogeneous if they all trigger on the same input symbol.

Thinking of the labeled digraph interpretation of NFAs, homogeneous transitions mean that all incoming edges of a vertex have the same label. We call an NFA homogeneous if every state has homogeneous transitions. Homogeneous NFAs allow us to associate all transitions with nodes instead of edges. This simplifies their implementation in hardware which is why they are often preferred over general NFAs [19, 17]. Importantly, homogeneous NFAs are equally expressive as normal NFAs as we will show in [subsection 3.4](#). [Figure 9a](#) shows a general NFA and [Figure 9c](#) shows an equivalent homogeneous NFA. Note that the accepting state s_1 in the general NFA triggers on two different inputs, on “a or b” and on “b”. In order to make the NFA homogeneous we need to duplicate s_1 . In the homogeneous NFA, every node has homogeneous transitions and we can associate the transition with the node itself instead of with the edge.

The basic building block for implementing homogeneous NFAs in hardware are STEs. To avoid confusion we will denote states of the NFA as “states” and the current content stored in the hardware register of an STE as its “activation”. An STE represents one state of an NFA. An abstract block diagram of an STE is given in [Figure 9b](#). The STE updates its activation in every clock cycle. To do so, the STE consumes the current symbol of the stream as input and checks whether this input satisfies the predicate associated with the transition into the STE. For instance, a node that activates on input “a” will check $(input = a)$ in its input decoding. The STE also consumes the current activation of all its neighboring STEs, that is, of all NFA states that transition into it. An NFA state will activate in the next step if its transition predicate is true and some predecessor state is currently active. Similarly, the activation of an STE will contain a 1 in the next cycle if its input decoding accepts the current input and the OR over its neighbors' activations is true (c.f. [Figure 9b](#)). A register stores the newly computed activation at the rising edge of the clock. An STE can be configured to be a starting state by storing a 1 in the *start* register. On reset, i.e. at time 0, the activation of the STE is set to the value in *start*.

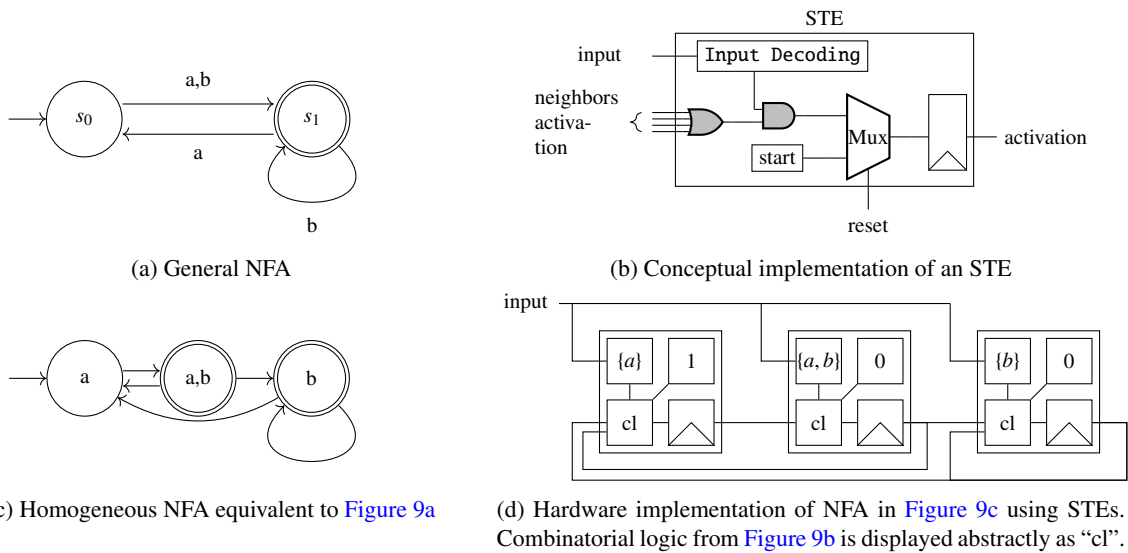


Figure 9: Correspondences between general NFA, homogeneous NFA and the hardware implementation of a homogeneous NFA using STEs.

Figure 9d shows abstractly how the homogeneous NFA from Figure 9c can be implemented using three STEs. Within each abstract representation of an STE, the top left box is the **Input Decoding** with the predicate of the transition. We represent the predicate abstractly as the set of accepted inputs. The top right box is the *start* register. The bottom left box is the combinatorial logic that computes the activation update of the STE in each cycle. The bottom right box is the register that stores the current activation of the STE. This depiction doesn't show the reset input of every STE. Moreover, the implementation doesn't report whether there is an accepting state active or not. Note that the middle and the right STE correspond to accepting NFA states. To report whether any accepting state is active, we would thus compute the OR of the activations of these two STEs.

The filter we aim to implement follows the basic structure seen in Figure 9d. The input will be a stream of ECI messages. Whenever an accepting state is active, we will output the corresponding ECI message. However, in order to make the NFA runtime configurable, we need to be able to configure

- the input decoding
- the value of the *start* register
- whether or not a state is accepting
- the routing between STEs.

We will talk about how we achieve this in detail in subsection 3.3. However, in order to understand the next section, we need to introduce one aspect of this reconfiguration now: the *overlay graph*. An overlay graph is essentially a skeleton NFA, consisting of unconfigured STEs with pre-connected transitions. The idea is to overprovision both STEs and routes for transitions between STEs such that a concrete NFA can be implemented at runtime by (1) configuring some of the STEs and (2) disabling the unused transitions. If

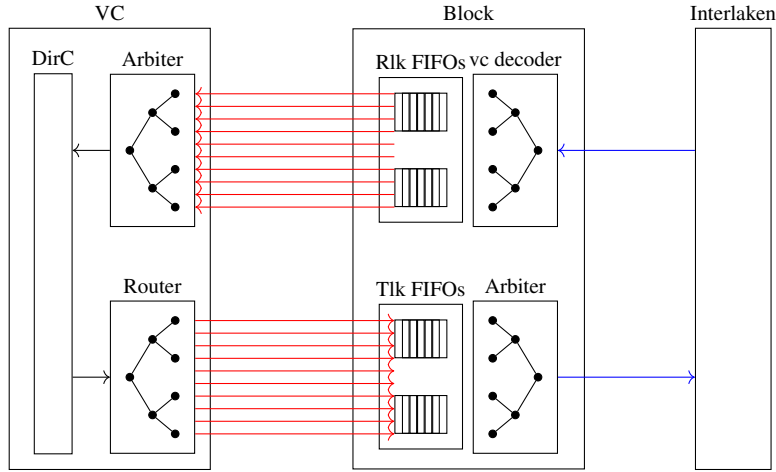


Figure 10: Major elements of the data path between interlaken, block and VC layers of the ECI stack. VC Decoder assembles data words from blocks to ECI messages and assigns them to VCs. Arbiters perform round robin scheduling between their inputs to provide one output. The Router assigns the ECI message created by the DirC to the correct VC. We tap VC layer messages at the Interface between block and VC layer (Red). The interface between block and Interlaken layer is depicted in Blue.

the NFA graph is a subgraph of the overlay, then it can be mapped in this way to the overlay. This idea of a skeleton NFAs was introduced in [13].

3.2 Integration into Enzian

A major goal of this work is to filter the stream of VC layer messages. The first question we need to answer is how to access this stream of ECI messages, i.e. how we can tap the data flow without intruding into the ECI stack. Ideally, we want to tap in a way that makes as few assumptions as possible about the concrete implementation of both the VC layer and the block layer.

We can do this by tapping at the interface between the two layers. As long as this interface stays the same, the tapping will work even if the implementations of VC and block layer change. The interface between VC and the block layer is depicted in Figure 10. It consists of a set of FIFOs: there is one FIFO per vc and per direction (i.e. from CPU to FPGA and from FPGA to CPU). In the receiving direction (from CPU to FPGA), ECI messages are assembled by the block layer in the VC Decoder module and put on the FIFO of the correct VC. The backend of this VC then eventually pops this message off the FIFO to process it. Similarly, in the sending direction (i.e. from FPGA to CPU), ECI messages are created by the VC backend and put on the correct FIFO for sending them over the link. Pushing data onto and popping data from these FIFOs uses a valid-ready handshake: The producer of the data (for instance the Router module) puts the data on the bus and simultaneously asserts a *valid* signal to indicate that the data can be read. As soon as the consumer (in this case the Tlk FIFO) is ready to consume the data, it asserts a *ready* signal. The data transfer happens when both valid and ready are asserted. By observing the valid-ready signals at the interface of all Tlk and Rlk FIFOs and reading data whenever both valid and ready are high, we read all sent and received ECI messages.

We now formulate some assumptions about the implementation of the VC backends in order to (1) make

dependencies of the current implementation clear and, (2) get concrete input data characteristics that help the design process:

1. We observe at most one ECI message per cycle, vc and direction. This assumption is warranted by current implementation of the VC-block layer interface using FIFOs. Note that integrating the second ECI link will double this to two ECI messages per cycle, vc and direction.
2. There is no causal relationship between ECI messages that get pushed/popped in the same cycle. This is a necessary assumption because the tracing unit has no knowledge of the actual order in which ECI messages are processed by the VC layer.
3. For expressing message filters, we are only interested in message headers, not their payloads. Since we are working at the VC layer it makes sense to ignore user-level content. In theory, inspecting the payload and looking for particular content is possible but it is generally harder than analyzing header information because we can't assume anything about the structure of the payload.

One consequence of the assumption 1 is that the tracing engine needs to process at most $2 * 14 = 28$ ECI messages per cycle. The data comes naturally in the format

$$\left(m_i^{\text{in}}\right)_{i=0}^{13} \left(m_i^{\text{out}}\right)_{i=0}^{13} \quad (8)$$

where $m_i^{\text{in/out}}$ is a valid ECI message for VC i or all ones if there is no data on this VC in this cycle. This is the input data for the tracing unit, we call this a *message batch*. The size of a message batch limits the bandwidth of the tracing unit: we need to be able to process $2 * 14 * 64 = 1792$ bits per cycle. This assumption is weaker than what the current VC backend implementation guarantees: currently each of the four VC backends processes at most one ECI message per cycle and direction. So the tracing unit can handle more bandwidth than is currently necessary. There are two reasons for still making this weaker assumption: (1) The design will still be relevant even if all VC backends get parallelized at some point. (2) To profit from the lower bandwidth guarantee would complicate the design. In particular, to reduce the bus width we would need some arbitration that only routes valid inputs onto the smaller bus. This arbitration would need to handle all possible subsets of valid messages which would require non-trivial combinatorial logic.

Let's elaborate a bit on assumption 2 and the timing relationship between ECI messages: The ordering of ECI messages is only preserved for one VC and one direction. For instance, if the tracing unit observes messages x and y in subsequent cycles on the same VC in the same direction, it knows that x is processed before y at the recipient. But if x and y are instead on different VCs or in different directions, no such conclusion is possible. This influences the patterns we can express, in particular, we should not allow patterns that rely on ordering between messages that can be re-ordered.

There are two levels at which we need to be aware of this limitation: (1) In formulating predicates on a message batch: Because two messages in the same batch are not really simultaneous, it makes no sense to allow predicates that ask for two messages to appear in the same batch. What does make sense is asking for *any of a set* of messages or *none of a set* of messages to appear in a batch. (2) When formulating the actual NFA: A pattern asking for two messages *of the same sender* to appear in some *fixed order* cannot be reliably detected because of re-ordering. In most cases we will need to rely on request-response pairs (c.f. [subsection 1.4](#)). For these we know the ordering: Because the response can only be sent once the request has been processed, the tracing unit is guaranteed to observe the request before the response.

Incorporating the order in which ECI messages are processed may be possible on the FPGA side because we could, in theory, observe the individual VC backends. However, the design would then be more tightly integrated with the concrete implementation of the VC layer. In particular, to know the precise order in

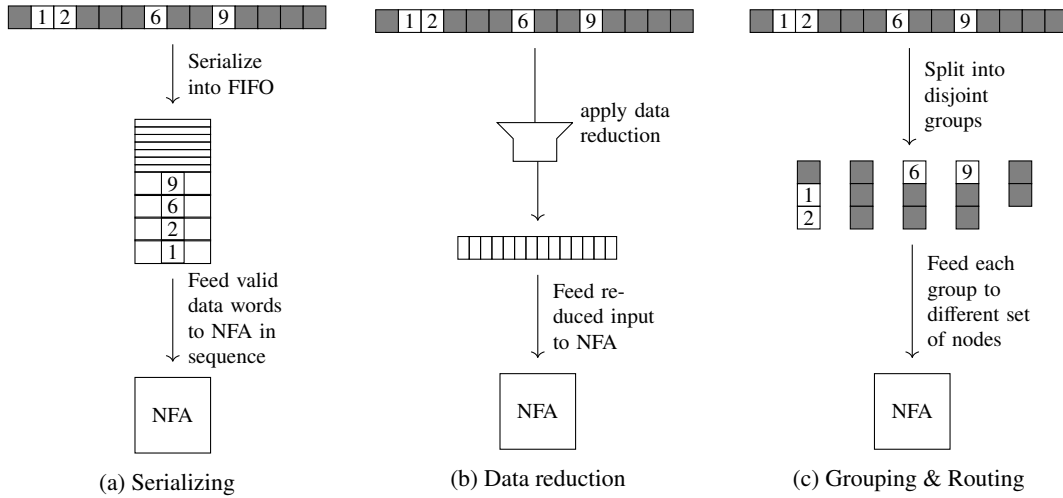


Figure 11: Three different ways of handling wide VC layer input. ECI messages at VCs $i = 1, 2, 6, 9$ have valid input, their headers are denoted by i .

which ECI messages are processed we would need to tap the data where it is consumed by the VC backend. Also, the same is not possible on the CPU side because the tracing engine is implemented on the FPGA.

Independently of the concrete hardware implementation of the NFA inside the tracing engine, we will need to communicate the VC layer input to each STE of the overlay at each cycle. The simplest approach would be to route every header of every VC in both directions at every cycle to each state of the NFA. If there is no valid data for some VC at some cycle, we just send dummy data. As dummy data we insert *IDLE* ECI headers that have opcode 11111. In other words, each NFA state receives at every cycle data in the format of Equation 8. Depending on the size of the NFA this can require considerable routing resources on the FPGA. As discussed in subsection 1.6, these routing resources are limited and using too much of them may prohibit the design from being mapped successfully onto the FPGA. In general, the more routing resources we need, the harder it is for the Vivado toolchain to implement our design on the FPGA and the less routing resources are available for other modules on the FPGA.

As discussed above, the current VC implementation can process at most four ECI messages per cycle and direction (one per VC backend). That is, at any clock cycle there are at most four incoming and four outgoing ECI message. Thus the naive approach outlined above uses way more routing resources than necessary (it provides routing resources to process $2 \cdot 14 \cdot 64 = 1792$ bits and at most $8 \cdot 64 = 512$ bits are actually ever used at once). However, it is very hard to provide a better, implementation-independent upper bound on the number of ECI words that are processed per cycle. Fundamentally, the data rate on the VC layer is limited by the data rate on the block layer. As discussed in subsection 1.3, the block layer can deliver at most 10.5 ECI messages per cycle and direction. Thus the number of ECI messages processed at the VC layer can't be higher than that *on average*. However, there is nothing that stops a hypothetical VC layer implementation from processing ECI messages in bursts. Since we are tapping at the interface between VC and block layer (c.f. Figure 10), we need to be able to handle such a burst at any time.

Given the above discussion, we see three possible ways of handling the VC input data in our tracing engine that limit the routing resource usage and allow processing worst-case data rates. The three strategies are visualized in Figure 11 and discussed below.

Serializing: The first option is to serialize valid ECI message headers and consuming them in multiple, consecutive cycles, as depicted in [Figure 11a](#). The serialized headers are buffered in a FIFO before getting processed. ECI message headers are ambiguous without the index of the VC on which they were transmitted. Because the information of which header arrived on which VC is implicit in the position of a message in the message batch [Equation 8](#), we would need to explicitly add this information to the headers to make this approach work. With an additional 4 bits to accommodate the VC index, this approach reduces the data path bus width to 68 bits. This is close to optimal given that we don't want to apply some data reduction on the input data before sending it to the NFA. The big disadvantage of this approach is that it requires us to have a hard upper bound on the average data rate of the VC layer. If this bound is $d > 1$ headers per cycle, the tracing engine would need to run at clock that is d times faster than the system clock. Otherwise the FIFO will eventually overflow. Running parts of an FPGA design at a higher clock rate is feasible in principle. The system clock on Enzian runs at 300MHz and the maximum clock frequency of the Virtex Ultrascale+ architecture is at 891MHz, however, no practical design would run at this frequency. Moreover, we are not confident that a slightly complex design could meet timing constraints even at only 600MHz. Thus this approach is fundamentally limited in its practicality if we assume an average data rate $d > 1$. One way to reason that practical workloads won't achieve data rates higher than 1 could be to argue that most ECI message transactions involve responses CL data. Since we are discarding payloads of messages, this would reduce the fraction of headers per data word considerably. A second possibility would be to only partially serialize the inputs: instead of processing one header per cycle, we could process n headers per cycle. This hybrid approach would limit the bus width while increasing the necessary bound on the data rate to n . However, we choose to go neither of these routes.

Grouping and routing: The second strategy attempts to avoid the downside of applying a fixed data reduction before sending the inputs to the NFA. Instead, different parts of the input are routed to different parts of the overlay graph, as depicted in [Figure 11c](#). Concretely, we split the set of STEs V of the overlay into ℓ groups V_i for $i \in [\ell]$ and send only some of the ECI headers m_i to each group V_i . Each STE then processes its input to decide whether it should activate or not. In particular, each STE needs to receive all the necessary ECI headers to compute its trigger.

Different filters will require different parts of the message batch, thus making the routing itself runtime reconfigurable would be beneficial to increase the number of NFAs we can implement at runtime. Runtime configurable routing can be implemented using multiplexers where the selection bits are set at runtime. The basic question of this design is how much data is routed to every STE of the overlay. This question has two degrees of freedom

- How many groups of STEs ℓ do we form
- How many ECI headers L are routed to each group

[Figure 12](#) displays an example design of the reconfigurable routing. For each group of STEs of the overlay, an 28-to- L mux would allow choosing L out of the 28 ECI headers. Each STE in a group then performs an input decoding on these L headers. In general, this design will require ℓ muxes with 28 inputs and L outputs of width 64. The major buses to the overlay nodes will have width $L \cdot 64$ instead of $28 \cdot 64$. Moreover, the input decoding at every STE will process $L \cdot 64$ bits of data each cycle. In order to profit from this design despite the overhead of the muxes, we will want L to be quite small. Choosing L to be small also allows discarding unused chunks of data at an early point: if no STE requires data from VC i we can avoid routing this data to any STE at all. However, choosing L too small will limit the expressiveness of the design: First, because every STE now only gets to see a small portion of the input, certain triggers cannot be represented.

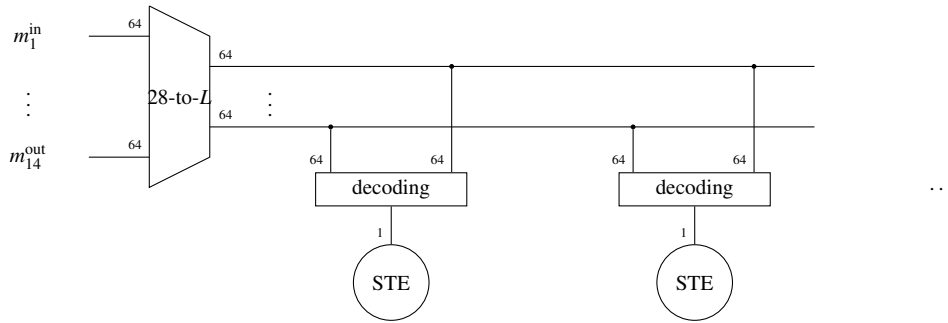


Figure 12: Group&Route architecture for one group: a configurable routing sends a subset of m headers to each STE in the group.

This can be overcome somewhat by replicating NFA state in multiple STEs that are distributed over multiple groups of the overlay, but if too many states need to be duplicated or if we need to form too many groups ℓ , the necessary increase in overlay size may outweigh the benefit of reducing L . Second, each STE is adjacent to a limited set of different groups. Because each NFA state needs to be adjacent to all its neighbor states, a small L will make it impossible to represent certain NFAs.

The fundamental trade-off in this design is thus between the amount of data we route to every STE, the complexity of the routing and input decoding logic. We found it very hard to think of configurations that strike a good balance in this trade-off. In every case there seem to be many NFAs that cannot be efficiently mapped. This is why we did not pursue this idea further.

Data reduction: The third option is to apply data reduction to inputs prior to processing them in the NFA, as depicted in [Figure 11b](#). We extract interesting information from each header and concatenate this information to form the reduced input which is passed to each state of the NFA. In particular, the reduced input still has a fixed format and VC index information is still implicit in the position of the individual bits of information. If the largest part of ECI message headers is uninteresting for the NFA filters we want to formulate, this approach achieves a large reduction in data bus width. For instance, extracting only ECI opcodes reduces the data width from 64 bits to 5 bits per header. The main shortcoming of this approach is flexibility at runtime: This form of data reduction is not easily configurable at runtime. Thus the expressible filters are limited by the information included in the reduced data. Changing the data reduction requires re-synthesis of the design. Since runtime configuration of the filter is a core goal of this work, this is a clear disadvantage. Nevertheless, we will follow this approach. The same approach was also followed in the previous work [\[9\]](#).

3.3 HDL design

Our goal is to produce a generic design of the tracing engine that can easily be integrated in both block and VC layer, as well as potentially in entirely different interconnects. In the following we describe the resulting design.

The major modules making up the tracing unit are depicted in [Figure 13](#).

- All input format related logic is in **Input Reduction** and **Input Decoding**. **Input Reduction** consumes the input as it is tapped from the ECI stack and brings it into the form expected by **Input**

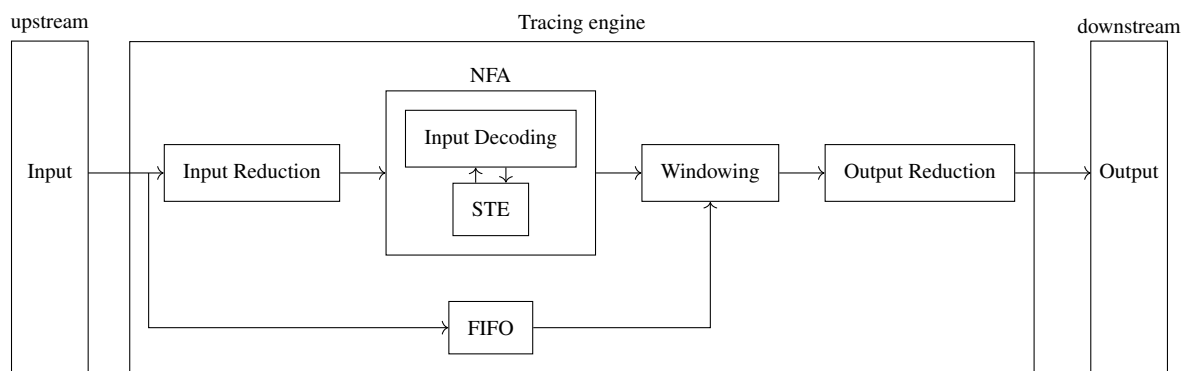


Figure 13: High-level block diagram of the tracing unit design

Decoding. Each STE of the NFA has an `Input Decoding` module that takes reduced input and triggers the activation of the STE. Encapsulating all input-format-related logic in these two modules allows re-using the rest of the tracing unit, in particular the NFA, to process different data than VC headers without any changes.

- The `FIFO` buffers the input while it is being processed and modified by `Input Reduction`, `Input Decoding` and `NFA`.
- The `NFA` contains the overlay network of STEs onto which arbitrary finite state machines are mapped. In particular, the `NFA` specifies the wiring of signals in the compile-time configurable overlay graph of STEs and it allows runtime configuration of these STEs to map different finite state machines without needing to re-synthesize the FPGA bitstream. `NFA` consumes a reduced input each clock cycle and propagates it to each STE. The STEs execute the `NFA` and produce for each input an *accept* and a *log* output. The `NFA` aggregates these outputs and propagates them further.
- `Windowing` consumes the *accept* and *log* outputs of the `NFA` as well as the corresponding input from the `FIFO` and produces the output stream. The output stream is a substream of all logged inputs that are within a window of n around an accepted input.
- The `Output Reduction` module has two purposes: First, it should bring the output stream into the format that can be written out over PCIe. This may involve some compression of the output stream to contain only relevant information. Second, it may add additional information to the output stream, such as a timestamp, that may enable further postprocessing.

The modularity of this design allows easy compile-time configuration and re-customization of the tracing engine to other use-cases. This is achieved mostly by the following separation of concerns: First, only `Input Reduction` and `Output Reduction` know about structure of the input data. Second, only `Input Decoding` and `Input Reduction` know about structure of the reduced data. Third, the `NFA` and the `Windowing` know nothing about the structure of the input data. However, all modules need to know the width of their data path but this does not impair modularity much because all widths can be globally configured in a central location.

3.3.1 Input Reduction and Decoding

We can implement any n -to-1 logic function $f(i_0, \dots, i_{n-1})$ by appropriately connecting enough CFGLUT5s. A basic inductive construction is the following

- for $n \leq 5$ simply use a CFGLUT5
- for $n > 5$ use two $(n - 1)$ -to-1 building blocks to implement $f(i_0, \dots, i_{n-2}, 0)$ and $f(i_0, \dots, i_{n-2}, 1)$, respectively. Further, use a 2-to-1 mux to choose between the outputs of the two $(n - 1)$ -to-1 building blocks based on i_{n-1} . Note that the 2-to-1 mux can be implemented with a CFGLUT5.

So, in theory we can have arbitrary, runtime configurable input decoding logic. However, the hardware cost of this construction scales exponentially in n . We thus face a trade-off between expressiveness of the input decoding and hardware cost. In the following we describe the concrete data reduction and input decoding that we found strikes a good balance between expressiveness and resource cost.

We find that the only ECI header information necessary to implement the use cases introduced in [subsection 1.7](#), are the following:

- The *type* of the ECI message (as listed in [Table 2](#))
- The *direction* of the ECI message (either FPGA-CPU or CPU-FPGA)
- The memory *address* affected by the ECI message.

We next describe how we can extract this information from the various ECI message headers. Different ECI messages have a different header structure. However, all ECI messages contain an opcode in bits [63 : 59]. Moreover, the VC index of a Message is implicit in the position of a message in the message batch. The opcode in conjunction with the VC index uniquely identifies the message type. The direction of the message is also implied by the position of the message in the message batch. Dealing with addresses is less straight forward. In particular, there are two questions we need to tackle.

First, not all ECI headers have an address field. Memory response messages MRSP_PSHA and MRSP_PEMD, for instance, only have a *partial_address* field. Memory forward response messages such as MRSP_PACK don't even have a *partial_address*. Instead, they have an *RReqId* field that identifies the forward request to which they respond. In general, all ECI *requests* and *notifications* contain an address field. All ECI *responses* contain a request id *RReqId* to reference the request. By maintaining a mapping from *RReqId* to address that is updated when observing requests, we could thus always associate a response with an address. This is what the DirC does. However, there is currently a simpler way: since the current implementation of the DirC only implements a subset of all ECI messages we only have to consider those messages. It turns out that for all currently supported messages, we can get the CL index from header bits [39 : 7].

Second, we see essentially two possible ways of how address information could be used to formulate NFA predicates:

1. *inter-cacheline patterns*: detect patterns in sequences of messages that span multiple CLs
2. *intra-cacheline patterns*: detect patterns in sequences of messages that reference the same CL multiple times.

Inter-cacheline patterns subsume intra-cacheline patterns in that any scheme that allows expressing inter-cacheline patterns will also allow expressing intra-cacheline patterns. Basic inter-cacheline patterns can be

expressed by simply matching the address field of a message. For instance, we could formulate the pattern that matches a sequence of three RLDD message for addresses 12, 14 and 13 in that order:

$$\text{MREQ_RLDD}(A : 12) \text{MREQ_RLDD}(A : 14) \text{MREQ_RLDD}(A : 13)$$

However, this isn't very interesting because it requires the user to know in advance exactly which addresses will be accessed. More interesting inter-cacheline patterns could be formulated using wildcards:

$$\text{MREQ_RLDD}(A : X) \text{MREQ_RLDD}(A : X + 2) \text{MREQ_RLDD}(A : X + 1)$$

Expressing a pattern like this requires the NFA to remember a concrete value of X to then check if subsequent address fields equal $X + 2$ and $X + 1$, respectively. In the language of RegExs, such a pattern is said to *capture* the value X and the construct is called a *capture group*. Capture groups cannot be implemented by NFAs per se. However, there are some works that extend hardware implementations of NFAs with some additional storage to provide limited capture group functionality [20]. In this work we follow a different strategy. Since caches maintain separate state for each CL, most interesting patterns require intra-cacheline patterns and not inter-cacheline patterns. This is certainly the case for the correctness and performance properties envisioned in subsection 1.7. However, some use cases, such as simple message filters, will likely want to ignore CL information entirely. We thus support the following two ways of dealing with CL index information

- Ignore CL information.
- Partition the ECI message stream into n disjoint substreams, based on CL address. Filter each substream independently.

Conceptually, the first mode of operations feeds all input messages to the same NFA while the second approach instantiates n NFAs and feeds each substream to a different NFA. Concretely, in the second mode of operation we specify a range of n CLs and one NFA. The ECI messages for each CL form a substream that is fed to a copy of the NFA. While conceptually each CL has its own instance of the NFA, we can implement this behaviour with less hardware duplication as discussed in subsection 3.3.2.

To provide all the necessary information to the Input Decoding module, we perform the data reduction depicted in Figure 14. The ECI opcode op is extracted directly, a valid bit v indicates whether the ECI header should be tracked or not and idx is the CL index that is addressed by this ECI header. In other words, idx indexes the conceptual NFA which should be updated with this information and the update should only happen if v is set.

We allow the user to choose at runtime whether they want to ignore CL information or not. If the user wants to observe $2^m \leq n$ CLs starting from address X , they can set a control bit $CL_mode = 0$, and base address $A = X$. Note that if X is not aligned to 2^m , the implementation will instead observe CLs starting from the closest address to X that is smaller than X and aligned to 2^m . Also note that the maximum number n of CLs we can observe is fixed at compile time because it is necessary to provision wide enough buses to accommodate n CLs. If the user wants to ignore CL information, the comparison with base address A is not performed. Instead, v is set to constant 1 and idx is set to constant 0^{n+1} .

The input comes in message batches as depicted in Equation 8. Given the timing relationship between messages within a batch (c.f. subsection 3.2), predicates that assert some combination of messages within one batch cannot be used reliably. Instead, we provide the following predicates on message batches M :

1. Basic predicate P: checks type, direction and VC of all messages in M and accepts if there is a message $m \in M$ that matches:

$$P_{X,Y,Z}(M) = \exists m \in M. (\text{Type}(m) = X \wedge \text{Dir}(m) = Y \wedge \text{VC}(m) = Z)$$

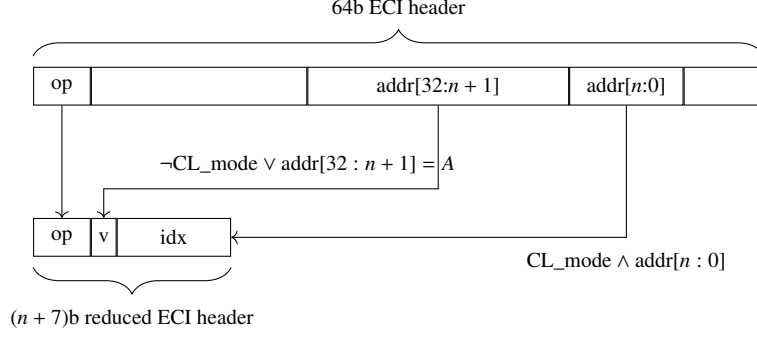


Figure 14: Data reduction performed in Input Reduction module. Direction and VC index are implicit in the position of the (reduced) message in the message batch. The opcode op is taken directly from the header. If the address should be ignored, then CL_mode is set to 1. Else, A is the prefix of the CL addresses to observe and the 2^{n+1} CLs $[A, 0^{n+1}], \dots, [A, 1^{n+1}]$ are observed. v indicates whether the ECI message concerns a relevant CL and idx is the index of the observed CL. Note that v is always 1 and idx is always 0 if CL information is ignored.

2. **Any**(P^1, \dots, P^k): is true if any of the basic predicates P^1, \dots, P^k match on M .
3. **None**(P^1, \dots, P^k): is true if none of the basic predicates P^1, \dots, P^k matches on M .

We propose the architecture depicted in Figure 15 to achieve the decoding of the above predicates on the reduced input. There is one CFGLUT5, one AND-gate, and one demux corresponding to each reduced ECI header, i.e. one for each VC and direction. The opcode op_i of reduced header i is matched by the configurable logic in the CFGLUT5. The result of the match is AND-ed with the valid bit v_i and then fed into the demux. The demux has m outputs, one for each CL that we're tracking. The demux is responsible for propagating its input to the output that corresponds to the correct CL. Which output of the demux is set is controlled by the idx part of the reduced input which indexes the substream of the CL. The Input Decoding further has m AND/OR gates. The AND/OR gates can be configured at runtime to AND-mode or OR-mode using one configuration register. Each AND/OR gate aggregates all partial results for one CL by either AND-ing or OR-ing them. The output of the Input Reduction is then the vector of binary outputs with one bit per CL. The bit for CL i is 1 if the substream of ECI messages corresponding to CL i satisfies the predicate implemented in the input decoding logic.

A basic predicate $P_{X,Y,Z}$ is implemented on this architecture as follows: The CFGLUT5 corresponding to VC Z and direction Y is configured to accept exactly opcode of message type X . Recall that ECI header opcodes are 5 bits wide and that CFGLUT5s can implement any 5-to-1 logic function. Additionally, the AND/OR gate is configured to OR mode.

To implement the composed predicate **Any**(P^1, \dots, P^k) we first group the predicates P^1, \dots, P^k by VC and direction. Then, for each VC v and direction d , we consider the corresponding group of predicates $P_{X_1,d,v}, \dots, P_{X_m,d,v}$ and configure the CFGLUT5 of VC v and direction d to *accept all opcodes* X_1, \dots, X_m . Additionally, we configure the AND/OR gate to OR mode.

To implement the composed predicate **None**(P^1, \dots, P^k) we again group the predicates by VC and direction. Then, for each VC v and direction d , we consider the corresponding group of predicates $P_{X_1,d,v}, \dots, P_{X_m,d,v}$ and configure the CFGLUT5 of VC v and direction d to *accept everything but opcodes* X_1, \dots, X_m . Additionally, we configure the AND/OR gate to AND mode.

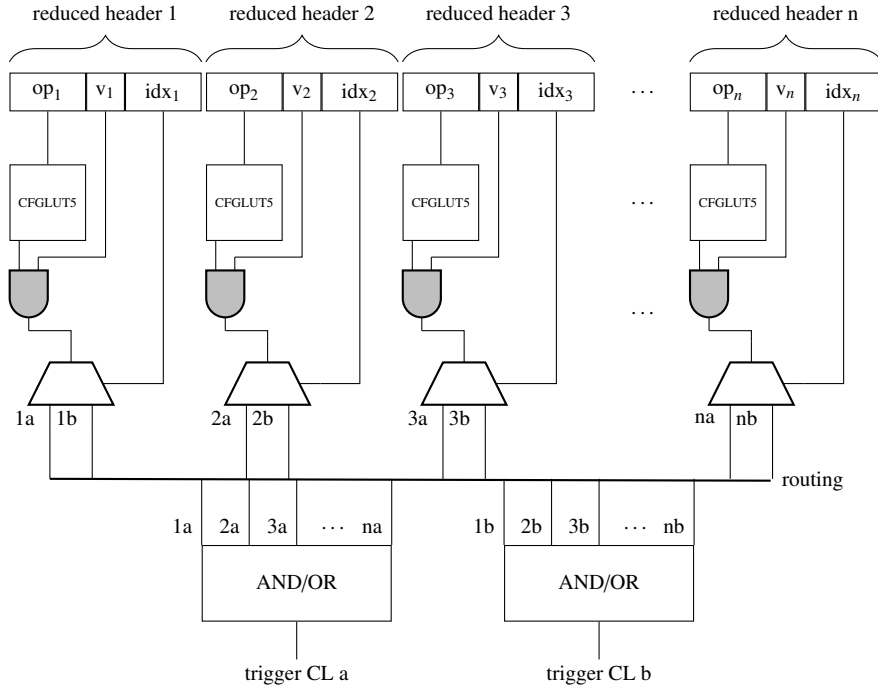


Figure 15: Input Decoding block diagram for n ECI words and distinguishing $m = 2$ different CLs.

Up to the substream demuxing, this input decoding is quite compact: a logic slice on the ultrascale+ architecture contains 8 CFGLUT5s. Moreover, we consider at most 28 ECI headers per cycle (one incoming and one outgoing message for each of the 14 VCs). Thus we require less than 4 full logic slices to accommodate the CFGLUT5s of one input decoding. Some additional resources are spent for the AND/OR aggregation. However, if we want to inspect many substreams, we expect the demuxing logic to consume significant hardware resources. We will discuss resource usage for this design in [subsection 5.1](#).

The Input Decoding module additionally aggregates the valid flags v_i into one *valid* bit for each CL that indicates whether there is any valid input header for this CL (not shown in [Figure 15](#)). This is necessary as we will see in [subsection 3.3.2](#).

3.3.2 Runtime reconfiguration of NFAs

The idea of runtime reconfiguration is that an NFA can be loaded onto the FPGA while the FPGA is running and without needing to re-synthesize the HDL code running on the FPGA. In order to enable runtime reconfiguration, the STEs and other hardware building blocks making up the NFA need to be overprovisioned at compile time. Runtime reconfiguration then loads a configuration bitstring from the outside world onto the FPGA over some interface (in our case PCIe). The configuration bitstring is stored in predefined registers on the FPGA and thereby controls the NFA's operation.

In the following we distinguish between the NFA, which is the mathematical description of a finite state machine, and the overlay, which is the generic hardware implementation that runs on an FPGA and can be configured to implement an NFA. The overlay is a graph of STEs that were introduced in [subsection 3.1](#).

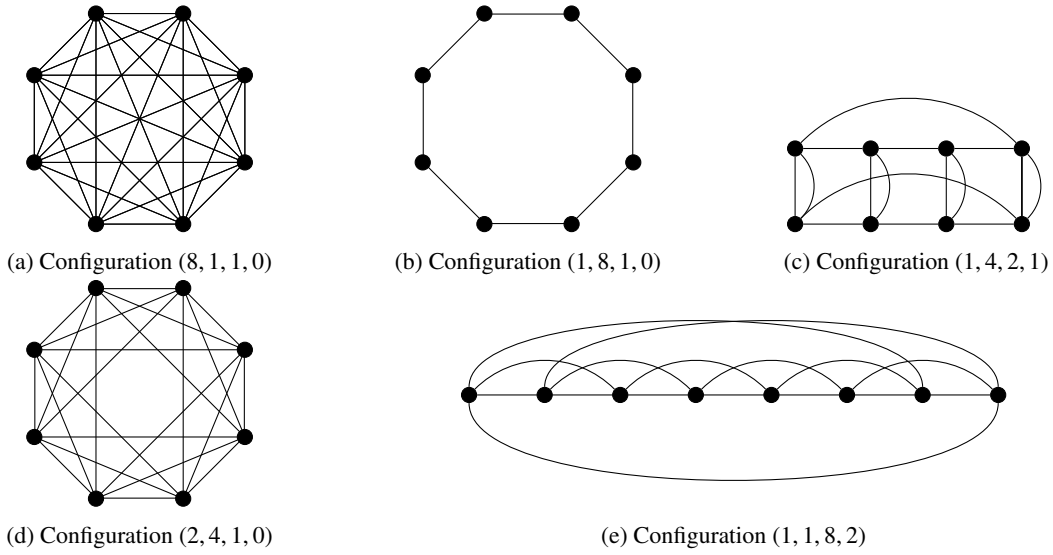


Figure 16: Various configurations of the rings-of-cliques overlay graph.

We denote the process of implementing an NFA on the overlay as *mapping* the NFA onto the overlay. When mapping an NFA onto the overlay, each state of the NFA will be mapped to exactly one STE. When mapping an NFA onto the overlay we need to ensure that adjacent states are mapped to adjacent STEs. We will consider the problem of mapping NFAs in [subsection 3.4](#).

The overlay graph used in this work is the *rings-of-cliques* graph already used in [9]. This graph is parametrized by 4 parameters: clique size C , ring length L , number of rings R and number of neighbouring rings N . We write (C, L, R, N) to denote a particular configuration. Each vertex $v_{c,l,r}$ in the structure is uniquely identified by indices $c \in [C]$, $l \in [L]$, $r \in [R]$ denoting the index of the node in the clique, the index of the clique in the ring and the index of the ring. Two vertices $v_{c,l,r}$ and $v_{c',l',r'}$ are connected if any of the following is true

- Their rings are not more than N apart and their cliques are in the same position of their respective ring. This includes the case where the two vertices are in the same clique. More formally, if $l = l'$ and $\min(r - r' \bmod R, r' - r \bmod R) \leq N$
- They are in neighboring cliques on the same ring. More formally, if $r = r'$ and $\min(l - l' \bmod L, l' - l \bmod L) = 1$

Figure [Figure 16](#) illustrates several configurations of the rings-of-cliques overlay graph. The rings-of-cliques provides us with significant flexibility to control the density of the STE network on the FPGA and thus an easy way to trade off number of states with connectivity between states. As displayed in [Figure 16](#), we can represent the densest graphs (cliques), the sparsest graphs (cycles) as well as many things in between. We'll discuss this choice of overlay graph in more detail in [subsubsection 5.4.2](#).

Given a mapping of NFA states to STEs, we need to configure each STE such that it implements the behaviour of its state. The functional behaviour of a state in a homogeneous NFA is the following

A state activates if either it is an initial state and time $t = 0$, or one of its predecessors was

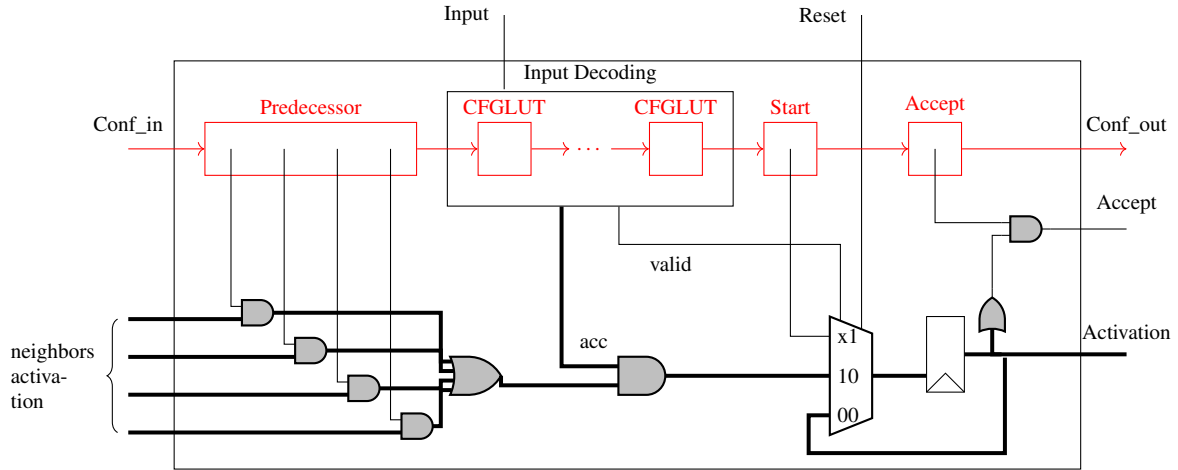


Figure 17: Logic diagram of a single STE with 4 predecessors. Red elements implement runtime reconfigurability. The red signal is the config chain that connects all STEs while black signals visualize the data flow. Thick black lines are buses of width n where n is the maximum number of CLs the NFA can track. Logic gates with one n -bit input bus reduce their input to a single bit. Logic gates with multiple n -bit bus inputs perform the bitwise operation and output an n -bit bus. Logic gates that have both an n -bit bus and a 1-bit signal input first broadcast the 1-bit signal to n -bit bus and then perform the bitwise operation.

active at $t-1$ and it accepts the input of time t . A state de-activates at time t if it doesn't activate at time t and it receives valid input.

The restriction on de-activating is necessary in our setup because on the VC layer there is not necessarily valid input data on every clock cycle.

This is implemented in a runtime reconfigurable way by STEs as depicted in Figure 17. Red are all elements that are used for runtime reconfigurations, black are data path elements. For legibility, pipeline stages are ignored. At time t , we choose one out of three possible values as the next activation: If reset is asserted, we choose the value in the *start* register. Else, if no valid input was received in this cycle, as indicated by the *valid* output of *Input Decoding*, we keep the previous activation. Else, we check if any transition into this state is active. A transition is active at time t if (1) the input of time t is accepted by *Input Decoding*, i.e. if *acc* is asserted, and (2) if a predecessor state was active at time $t-1$.

At time t the STE receives the activation of all connected STEs from time $t-1$, the input of time t and a reset signal. The *Predecessor* register contains one bit for each neighbor STE. The bit indicates whether the two states mapped to the two STEs are neighbors in the NFA. By AND-ing the activation from each neighboring STE with its *Predecessor* config bit, we can detect if an NFA neighbor of this state was active at time $t-1$. By OR-aggregation of this partial result, we can detect if any NFA neighbor of this state was active at time $t-1$. By AND-ing this partial result with the *acc* output of the *Input Decoding*, we can thus compute if any transition into this state is currently active.

We output the activation of this state at time t to all connected STEs, which will use it to compute their activation at time $t+1$. Further, if this STE corresponds to an accepting state of the NFA, as indicated by the *Accept* configuration register, we assert the *Accept* output that is aggregated centrally by the NFA engine.

Note that this description assumes that we are only tracking one CL. In the case of multiple CLs we

need to conceptually execute multiple NFAs concurrently. Most of the work to do this is already performed by the `Input Reduction` and `Input Decoding` modules: As described in [subsection 3.3.1](#), the `Input Decoding` module of an STE outputs an n -bit vector where entry i is 1 if the substream for CL i satisfies the transition predicate of the STE at this point in time. Given this *acc* vector we can compute and maintain a corresponding activation vector where index i indicates that the NFA for substream i is currently active. The only thing that changes in the STE implementation is that some of the signals are now n -bit buses as already indicated in [Figure 17](#). Additionally, also the wiring between connected STEs now consists of n -bit buses. A design choice is how to report the accepting state of the NFA: We could either report n different accepting values, one per substream, or we could aggregate the accepting values of all substreams and globally accept whenever one substream’s NFA is accepting. We choose the latter meaning (note the OR-aggregation in the STE before the *Accept* output in [Figure 17](#)). The reason for this is that there is only one output stream of filtered ECI messages, i.e. the additional information of which substream is accepted is lost in the output anyway. In cases where this information is desired, it would be necessary to write out multiple filtered output streams. This could be achieved by annotating the messages in the output stream with the index of the accepting NFA. However, in our use cases, the substream information can always be extracted quite easily from the captured trace with some post-processing, so we didn’t implement this feature.

A further generalization that we implemented but that is omitted in [Figure 17](#) is the possibility of having multiple, semantically different *Accept* outputs: In an ordinary NFA, every state is either accepting or not. In consequence, the global state of an NFA is binary: at any time t the NFA is either *accepting* or *not accepting*. In our case, we use this global state to decide if we should output an ECI message or not. However, we could imagine an NFA that has non-binary output. In general, we can define c different labels l_1, \dots, l_c and label each state of the NFA with an arbitrary subset of these labels. The global state of the NFA at some time t is then the union of the labels of all active states at time t . This results in at most 2^c different global NFA states, one for every subset of the c labels. Having multiple labels in the tracing NFA would allow us to distinguish between many different actions to perform with messages. To implement c different labels, each STE requires c configuration registers, one per label, and c AND-gates. Similar to how we compute the *Accept* output in [Figure 17](#), we could compute each of the c label outputs by AND-ing the correct config register with the current activation. We will see the `Windowing` as one application of this idea in [subsection 3.3.3](#)

3.3.3 Windowing

Consider the case where we want to use the tracing engine to debug the VC backend by formulating an NFA that detects sequences of ECI messages that are illegal. Given such an NFA definition, all message sequences that lead to an accepting state are considered disallowed. The tracing engine will output all messages that caused the transition into an accepting state. These messages can then be observed in the output stream as witnesses of violations of the safety property defined by the NFA. However, this output will likely be only partially helpful because it only provides a snapshot of ECI at the exact time of the violation. Often it will be more useful to see a window of messages around the violation, so that the steps that lead up to the violation can be understood. This is what the `Windowing` module displayed in [Figure 13](#) achieves. In the following we will discuss the implementation and the semantics of the implemented windowing functionality.

To implement the windowing functionality we use two different labels for NFA states, there are *logging* states and *accepting* states. The concept of using multiple labels in an NFA was introduced at the end of [subsection 3.3.2](#). We say the input at time t is *logged* if some logging state is active at time t . Similarly, the input at time t is said to be *accepted* if some accepting state is active at time t . The output stream consists of the union of all accepted inputs and a substream of all logged inputs. From the logged inputs we include

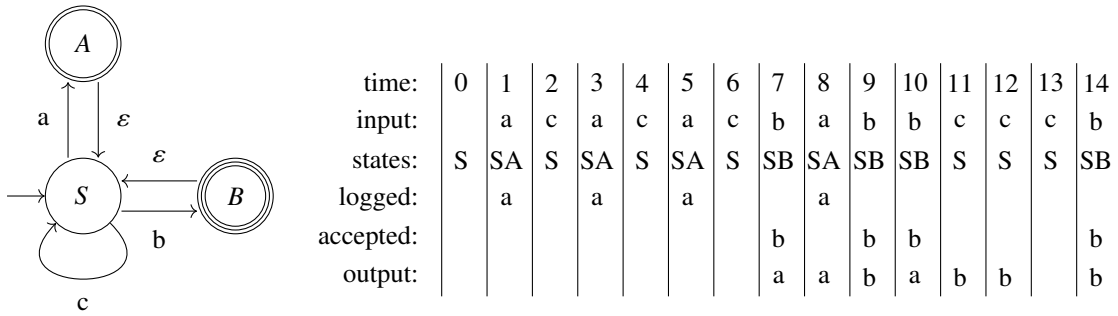


Figure 18: Example of windowing semantics for window size $n = 2$. State A is logging, state B is accepting.

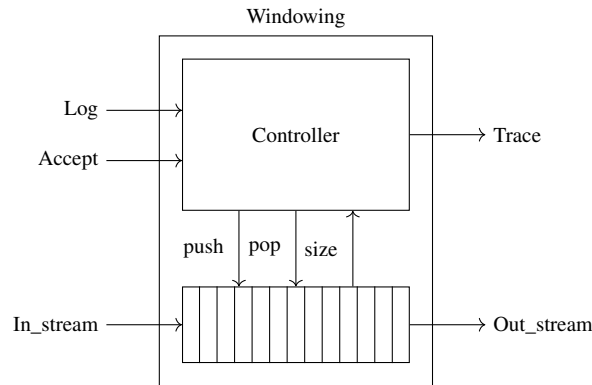


Figure 19: Input and output signals of the controller achieving windowing semantics.

at most n before and after any accepted input. Here n is the parameter that controls the size of the window. Concretely, for any logged input we consider the closest accepted inputs that come before and after it in the input sequence. If there are at most $n - 1$ other logged inputs between this input and any of these two closest accepted inputs, then we include it in the output stream, otherwise we discard it. Note that the substream of logged inputs will always need to buffer at least n inputs, otherwise there is no way of deciding whether some logged input should be put into the output stream or not.

These semantics are visualized for an example NFA and window size $n = 2$ in Figure 18. Note that we didn't convert the NFA in Figure 18 to a homogeneous form because it is more legible this way. State A is logging as indicated by the double ring, state B is accepting as indicated by the tripple ring. The input alphabet is $\Sigma = \{a, b, c\}$. Note that the initial state S gets activated on any input, A gets activated exactly on input a and B gets activated exactly on input b. Thus the logged substream consists exactly of all a symbols in the input stream and the accepted substream consists exactly of all b symbols. The first a symbol of the input is not included in the output stream because it has distance $3 > n$ from the next b, however, all other a symbols are included in the output because they are at most n logged inputs away from the next accepted input. Further, every symbol b is accepted and is thus part of the output stream. Note how the a input at time 3 needs to be buffered until time 7 before it can be added to the output stream. Similarly, the a input at time 1 needs to be buffered until time 5 until it can be discarded, because at that time it becomes clear that it is too far from any accepted input to be put in the output stream.

We implement the windowing functionality using a standard FIFO buffer of size at least n . As visualized in [Figure 19](#), the FIFO takes the ECI message stream In_stream as input and produces Out_stream , the union of accepting and logging substream, as output. A controller consumes the $accept$ and log output of the NFA as well as $size$, the current number of elements buffered in the FIFO and controls the $push$ and pop signals of the FIFO. The controller also produces a $Trace$ signal as output that indicates if some message in the output should be added to the result trace. To compute the control signals, the controller also maintains a counter $write$ that indicates how many of the next elements are included in the result trace. The control logic is straight forward:

$$\begin{aligned}
 push &= Log \vee Accept \\
 pop &= push \wedge size \geq n \\
 Trace &= write > 0 \wedge pop \\
 write &\leftarrow (Accept) ? size + 1 + n : \\
 &\quad (pop) ? write - 1 : write
 \end{aligned}$$

That is, we push all accepted and logged ECI messages onto the FIFO. We pop whenever there are at least n elements on the FIFO and a new element gets pushed. This ensures that we buffer all logged messages as long as necessary to decide whether they should be added to the trace or not. Whenever a new logged message arrives, we “open” a new window, i.e. we output at most n messages that are currently in the FIFO, the accepted message itself, and at least n logged/accepted messages that follow. We decrement $write$ on every message that we pop and we trace all messages that are popped while $write$ is positive, i.e. while a window is “open”.

3.3.4 Output reduction

The `Output Reduction` module consumes the filtered stream of ECI message batches and optionally performs a simple transformation on the message batch before they are sent over PCIe to the remote host. In the current implementation, the `Output Reduction` simply maintains a 64-bit counter that is incremented every clock cycle. This counter is appended to every message batch that is output. This timing information can be very useful for visualizing and understanding the data in a captured trace.

A possible extension would remove IDLE headers from the output stream. Recall that in our definition of a message batch in [Equation 8](#), we fill in an IDLE ECI header where no valid message is available. This padding ensures that the input data is always a full message batch and has always exactly the same format, i.e. that the VC index of each message is implicit in its position in the batch. The filtered output stream of the tracing engine consists of the message batches that were accepted by the NFA. Thus, IDLE headers are still part of the output stream. This facilitates parsing the filtered output stream in post-processing because parsing requires knowing the VC index of each ECI message. However, the downside of outputting these IDLE headers is that it increases the size of the output stream above what is strictly necessary. An alternative would be to remove the IDLE headers and instead annotate the ECI headers with their VC index in the output stream. This would lead to a more compact output stream and save PCIe bandwidth.

3.3.5 Pipelining and Flow control

As discussed in [subsection 1.6](#), a big difficulty in FPGA design is ensuring that the design can be routed in a way that meets all timing constraints. In particular, we need to ensure that the propagation delay of any path between two registers is at most one clock cycle. A common technique to reduce critical path length in

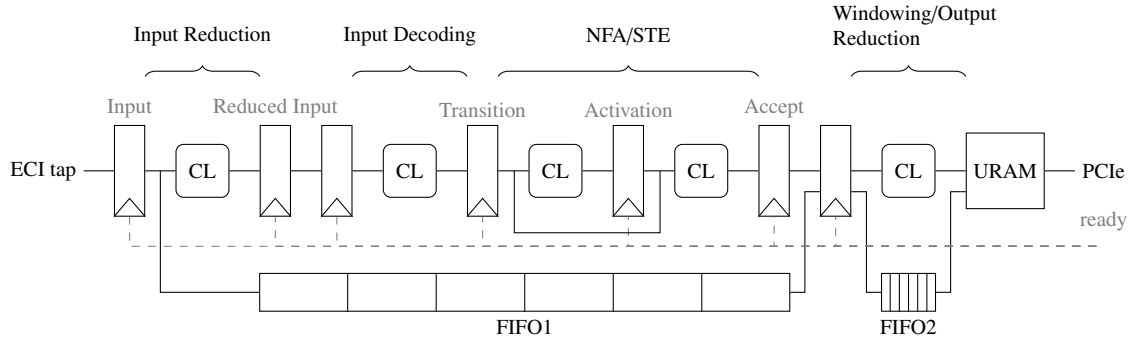


Figure 20: Pipeline stages of the tracing engine. Registers have the name of the value they produce in gray above them. Combinatorial logic blocks are indicated by CL. The name of the modules in which some combinatorial logic happens is indicated over the curly braces. FIFOs that buffer ECI messages while reduced or auxiliary data is processed are displayed below the registers and combinatorial logic. The ready signal controlling advancement of the pipeline is symbolized by the dashed gray line.

large designs is pipelining: The computation is performed in multiple steps and there are registers between consecutive steps.

The tracing engine has many such pipeline stages as displayed in Figure 20. Most registers are placed between two blocks of combinatorial logic. This limits the length of the critical path and helps the Place&Route algorithm to meet timing. However, three registers are placed back-to-back with other registers, namely there are two registers holding the *Reduced Input*, two registers holding the *Accept* output of the NFA and one register holding the tapped ECI message data which comes directly out of a FIFO. The reason for these back-to-back registers is that these signals need to travel a long distance and are propagated to or sent from many different locations. The reduced input is computed centrally in the Input Reduction module and then needs to be distributed to all STEs. Similarly, the *accept* output of every STE needs to be collected centrally at the NFA to decide if any state is accepting. On the other hand, the reason for registering the tapped ECI data is that the ECI stack implementation is quite big and complex. Registering the data allows the tracing engine to be placed farther away from the VC layer, which makes it easier for Place&Route. In all cases, adding these registers helps to meet timing constraints by reducing the distance a signal needs to travel between two registers.

The Figure 20 also illustrates the data flow in the tracing engine. The data is produced at the interface between block and VC layer where we tap the ECI messages. This generates a message batch at every clock cycle. This data is then sent along two different paths: The upper path reduces the message batch in the Input Reduction module, sends this reduced data to the NFA where it is distributed to all STEs. The STEs each perform their input decoding to determine if their transition is activated by this input. Next they perform their activation update and finally they compute their *accept* output. The activation of each STE is sent to all its neighbors for use in the next cycle, while the *accept* output is sent to the NFA for OR-aggregation. The aggregated *accept* output is then sent to the Windowing. The lower path stores the unreduced ECI words in a FIFO while the reduced data is being processed. When the NFA updates its output state, the corresponding ECI message batch is dequeued from the FIFO and also sent to the Windowing. The Windowing forwards the reduced data stream to the Output Reduction, then it is temporarily put into a URAM buffer and finally transferred to the remote host over PCIe.

Pipelined designs like the tracing engine often require some form of flow control. Flow control essen-

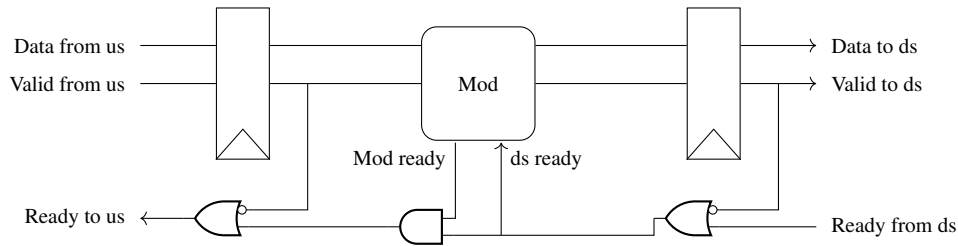


Figure 21: Standard valid-ready flow control. Module Mod gets data from upstream (us) and sends data to downstream (ds).

tially ensures two things: First, that only valid data is propagated through the pipeline, and second, that no data is lost inside the pipeline. Some form of flow control is necessary either if some pipeline stage doesn't consume or produce data at a constant rate, or if some pipeline stage is faster than some other stage. In the tracing engine, every pipeline stage is designed to produce and consume one input every clock cycle. However, there are two places where our pipeline might stall: First, the PCIe bandwidth is limited to 8GiB/s while the ECI tap produces 67GiB/s of data (counting the IDLE headers we insert if no valid ECI header is available). Of course, a considerable portion of this data is discarded by the NFA filter, but in theory there is a considerable speed mismatch between data source and sink. Second, the Xilinx FIFO primitives have some stall cycles after reset. That is, after resetting the FPGA or the tracing engine, there are about three cycles where valid ECI data is tapped, but we cannot process this data because the FIFO is not yet ready. Thus we need some form of flow control.

A standard approach to flow control is valid-ready flow control as visualized in [Figure 21](#). A module Mod is part of a pipeline. It receives input data from an upstream module, computes something on this data and sends its output data to a downstream module. The communication between Mod, upstream, and downstream is synchronized using two registers. The register between Mod and upstream stores the input data of Mod and a valid flag that indicates that the data is valid. Similarly, the register between Mod and downstream stores the output data of Mod and a valid flag that indicates that the data is valid. A ready signal travels in the opposite direction, from downstream through Mod to upstream. Valid-ready flow control synchronizes a producer-consumer relationship. The producer (e.g. upstream) asserts the valid signal whenever it has valid data on its data port. The consumer (e.g. Mod) de-asserts the ready signal whenever it is not ready to consume data. The basic contract between producer and consumer is the following

Data is transferred from producer to consumer if both valid and ready are high.

This contract entails that the consumer actually consumes the data in the cycle after both valid and ready went high. Similarly, in the cycle after both valid and ready went high, the producer either has to register new data or it has to de-assert the valid flag. One benefit of valid-ready flow control is that it decouples producer and consumer: The producer can push new data whenever ready is high. Similarly, the consumer can read data whenever valid is high.

The initial design of the tracing unit used valid-ready flow control to synchronize its pipeline stages. One disadvantage of valid-ready flow control is that the ready signal must travel through all pipeline stages in every clock cycle. In our design, the distance traveled by the ready signal (indicated by the dashed line in [Figure 20](#)) is quite long. In consequence, we often faced timing issues on the ready path. We implemented two ways around this problem.

First, we split the length of the ready path in half by registering the ready signal roughly in the middle

(before the Windowing module). Note that this introduces a problem for the valid ready flow control: An upstream module will only learn after one cycle delay that a downstream module wasn't ready in the previous cycle. To avoid data loss in this scenario, a duplicate set of registers can be installed at the first downstream module after the ready register. These duplicate registers will accommodate any data that is sent by the upstream module while the downstream module isn't ready. Note that per register stage of the ready signal, exactly one such set of overflow registers needs to be installed, because the upstream module sees the ready signal with exactly one cycle delay. This approach works but it is not very clean: Allocating overflow registers uses up additional FPGA resources. In the case of very wide data paths such as the un-reduced ECI message batch, this can be costly. Moreover, it complicates the design because we need additional logic to decide which register (the normal or the overflow register) should be read.

The second solution abandons full valid-ready flow control for a simpler approach that only uses a valid signal and no ready signal. Conceptually, we simply fix the ready signal to 1 at every location. In this setup, a producer will assert the valid signal as soon as it has produced some data and because the ready signal is always high, the producer will either push new data or de-assert the valid signal in the next cycle. That is, this approach ensures that only valid data is read and that all data is read at most once but it doesn't ensure that no valid data is dropped when the consumer is stalling. A closer inspection of our pipeline reveals the following: If we drop data in any pipeline stage before the Activation stage, the NFA will be in an inconsistent state. If we drop data in any later stage, the NFA will remain in a consistent state but the output stream will potentially be incomplete. A pragmatic policy is thus the following: We avoid the first case and we notify the user of the second case. The only module that can stall before the Activation stage is FIFO1. In order to avoid data loss, we use ordinary valid-ready flow control up to this point. The two modules that can stall after the Activation stage are FIFO2 and the URAM. The URAM buffers the outstream before the PCIe transfer. It only stalls when it is full. In that case, the data rate of the VC backend was simply faster than the bandwidth of PCIe and all our buffer space is used. In other words, overflow is the only option and we notify the user of this. Like FIFO1, FIFO2 can also stall on resets. However, every time FIFO2 is reset, we also reset FIFO1 and the whole pipeline. Because the two FIFOs are the same Xilinx primitive, we expect FIFO2 only to stall when FIFO1 also stalls. Because we use valid-ready flow control up until FIFO1, this means that there can never be data loss due to FIFO2. We choose this second approach over the first one because it is much simpler and works equally well.

3.4 Frontend

The frontend is responsible for compiling a textual description of a filter into a configuration bitstream for the tracing unit. The major steps of this compilation process are

1. Parsing the textual representation of the NFA
2. Mapping states of the input NFA to STEs of the overlay graph
3. Translating the triggers of each state, that are given in a plain-text description, into a form that can be implemented in the configurable input decoding
4. Generating a binary string that contains the configuration of each hardware element in the right order

In the rest of this section we will consider these steps more closely.

The textual representation is given as a YAML file, thus we can offload much of the parsing to a YAML library. [Listing 1](#) shows the definition of a simple NFA using structure and syntax as expected by the frontend.

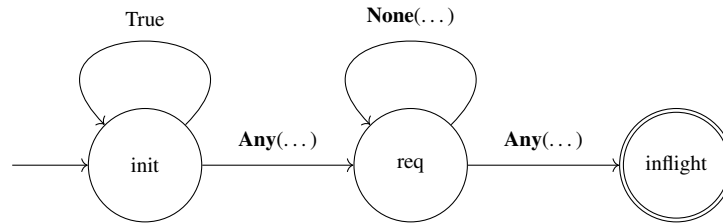


Figure 22: NFA corresponding to the textual description in Listing 1. Predicate `Any(cpu.MREQ_RLDD, cpu.MREQ_RLDI, cpu.MREQ_RLDX)` is abbreviated with “`Any(...)`”. Predicate `None(fpga.MRSP_PEMD, fpga.MRSP_PSHA)` is abbreviated with “`None(...)`”. All states are logging, so we don’t depict this in the image.

Listing 1: Example NFA definition

NFA:

```

init:
  accepting: false
  starting: true
  logging: true
  transitions:
  - pred: init
    trigger: true
req:
  accepting: false
  starting: false
  logging: true
  transitions:
  - pred: init
    trigger: Any(cpu.MREQ_RLDD, cpu.MREQ_RLDI, cpu.MREQ_RLDX)
  - pred: req
    trigger: None(fpga.MRSP_PEMD, fpga.MRSP_PSHA)
inflight:
  accepting: true
  starting: false
  logging: true
  transitions:
  - pred: req
    trigger: Any(cpu.MREQ_RLDD, cpu.MREQ_RLDI, cpu.MREQ_RLDX)
  
```

This NFA filters in-flight requests, i.e. memory requests that arrive at the DirC while it is already processing another memory request. A visual representation of this NFA is given in Figure 22. We define three states, *init*, *req* and *inflight* and for each state we specify if it is a starting state, an accepting state and a logging state. For each state, we also provide a list of input transitions. Each input transition specifies the predecessor state *pred* and the trigger that activates the transition. Note that we allow non-homogeneous NFA in the description of filters. The expressible triggers correspond to the predicates defined in subsection 3.3.1. The syntax `cpu.MREQ_RLDD` specifies a basic trigger that matches ECI messages of type `MREQ_RLDD` that

are sent by the CPU. The YAML parser will treat these triggers as plain-text, thus we implemented a simple parser based on parser combinators that only parses these expressions.

Once the parsing is done, the frontend will move on to step 2 and map the NFA onto the overlay graph. As described in [subsection 3.1](#), we require NFAs to be homogeneous. When given an arbitrary NFA we thus need to transform it to homogeneous form first. Further, unlike the previous work [9], we implement epsilon transitions as ordinary transitions. The major reason for this is that it makes the internal state update logic of STEs simpler: An ordinary transition between two STEs s_1 and s_2 triggers only in the cycle after s_1 has activated. An epsilon transition between s_1 and s_2 , on the other hand, triggers immediately when s_1 activates. Implementing real epsilon thus requires a second round of state updates right after the state update from ordinary transitions has been computed. There is an easy transformation from epsilon transitions to ordinary transitions that we can perform to remove all epsilon transitions.

Thus, in order to map an arbitrary NFA A onto the overlay we need to perform two tasks. First, we need to transform the original NFA A to an equivalent nfa A' that uses only homogeneous transitions and that doesn't use epsilon transitions. Second, we need to find a mapping from states of A' to STEs of the overlay. We can formulate these steps as operations on the labelled digraph G_A associated with A as defined in [subsection 1.5](#).

Epsilon transitions As described in [subsection 1.5](#), the semantics of an epsilon transition from state s_1 to s_2 is that s_2 activates whenever s_1 activates. We can thus replace the explicit epsilon transition (s_1, s_2, ε) by adding all triggers of s_1 to s_2 . More formally, we first add an edge $s_{i_0} \xrightarrow{a} s_{i_n}$ for each path

$$s_{i_0} \xrightarrow{a} s_{i_1} \xrightarrow{\varepsilon} \dots \xrightarrow{\varepsilon} s_{i_n} \quad (9)$$

consisting of a normal transition followed by a positive number of epsilon transitions. Then, we remove all epsilon edges. It can easily be seen that the resulting NFA is equivalent to the original NFA. This transformation is illustrated in [Figure 23a](#): Both s_3 and s_4 activate whenever s_2 activates. s_2 activates at time $t + 1$ if s_1 is active at time t and input a is received. We can thus replace the two epsilon transitions (s_2, s_3, ε) and (s_2, s_4, ε) by (s_1, s_3, a) and (s_1, s_4, a) , respectively.

Homogeneous transitions As defined in [subsection 3.1](#), the hardware implementation using STEs dictates that each node should have homogeneous transitions. An NFA A has only homogeneous transitions if all triggers for one particular node are the same, i.e. if the graph $G_A = (V, E)$ associated with A satisfies

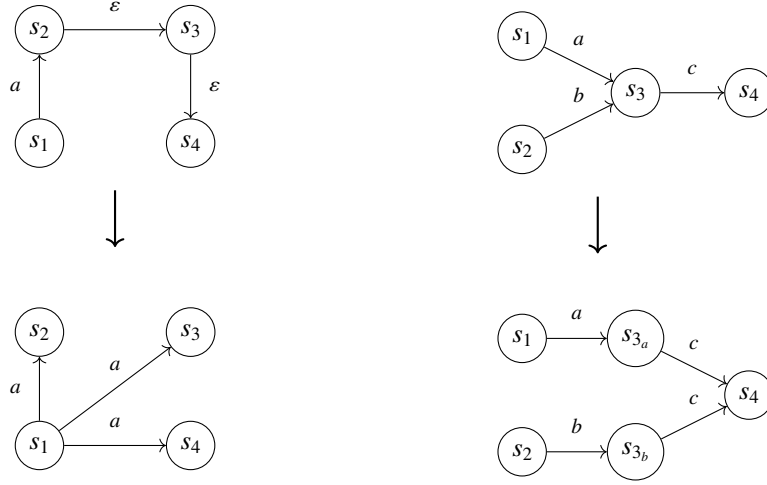
$$\forall s \in V. (s_1, s, a), (s_2, s, b) \in E \rightarrow a = b \quad (10)$$

Requiring homogeneous transitions doesn't restrict the expressiveness of NFAs as for each NFA A with non-homogeneous transitions there exists an equivalent NFA A' with homogeneous transitions. Intuitively, we replicate each state once for each of its different triggers. More formally,

$$A' = (V' = \{s_x \mid s \in V, e = (\cdot, s, x) \in E\}, \quad (11)$$

$$E' = \{(s_x, t_y, y) \mid (s, t, y) \in E, s_x \in V'\}) \quad (12)$$

where we write (\cdot, s, x) to mean an arc with arbitrary source to target s with label x . This transformation is illustrated in [Figure 23b](#): State s_3 has non-homogeneous transitions (s_1, s_3, a) and (s_2, s_3, b) . We thus create two copies s_{3_a} and s_{3_b} of it, one for each of the triggers a and b . Then we create homogeneous transitions by associating trigger a with s_{3_a} and trigger b with s_{3_b} . Finally, we replace transition (s_3, s_4, c) by the two transitions (s_{3_a}, s_4, c) and (s_{3_b}, s_4, c) .



(a) Replacing epsilon transitions by ordinary transitions

(b) Making transitions homogeneous

Figure 23: Examples of the two necessary NFA transformations

Mapping algorithm Given overlay graph $G_O = (V_O, E_O)$ and the NFA graph $G_A = (V_A, E_A)$, the mapping algorithm must solve the subgraph mapping problem: find a mapping $f : V_A \rightarrow V_O$ subject to the constraint that

$$(u, v) \in E_A \rightarrow (f(u), f(v)) \in E_O \quad (13)$$

That is, every edge in the NFA must be maintained by the mapping. Here we can ignore the edge labels because they are not relevant to the mapping problem. The corresponding decision problem, called the subgraph isomorphism problem, is a well-known NP-complete problem. There exist dedicated solvers for subgraph mapping [21]. We chose, however, to formulate the problem as an Integer Program (IP) and solving it using standard IP solvers such as Gurobi [22] or Coin-OR [23]. The main reason for this decision is that we initially implemented the Group&Route design discussed in subsection 3.2. In this design the mapping problem becomes more complicated because we simultaneously choose a mapping of NFA states to the overlay and a routing of the input to different parts of the overlay. These two choices can be optimized together using an IP formulation but this would be impossible using a subgraph mapping solver. We use the following IP formulation of the subgraph mapping problem:

$$\max \sum_{\substack{v \in V_A \\ c \in \text{Cliques}}} x_{v,c} \quad (14)$$

$$\text{s.t.} \sum_{c \in \text{Cliques}} x_{v,c} \leq 1 \quad \forall v \in V_A \quad (15)$$

$$\sum_{v \in V_A} x_{v,c} \leq \text{Clique_Size} \quad \forall c \in \text{Cliques} \quad (16)$$

$$x_{v,c_1} + x_{u,c_2} \leq 1 \quad \forall (u, v) \in E_A. \forall c_1, c_2 \in \text{Non-adjacent-Cliques}. \quad (17)$$

$$x_{v,c} \in \{0, 1\} \quad (18)$$

The binary assignment variable $x_{v,c}$ is 1 if and only if NFA node v is mapped to clique c of the overlay. Note that all nodes of a given clique are equivalent in the rings-of-cliques graph: Any node in a given clique has the same set of neighbors as all other nodes in the same clique (because a node is also connected to itself). Thus we can ignore the individual nodes of the overlay graph and reduce the problem size by only considering its cliques. Each of the $(\text{Rings} \cdot \text{Length})$ cliques of the overlay graph is assumed to have an ID and *Cliques* is the set of all these IDs. Moreover, we denote *Non-adjacent-Cliques* the clique-ID pairs of distinct cliques that are not adjacent. Note that two vertices in the overlay graph are *not* connected if and only if they are in two cliques that are non-adjacent. The overlay graph is static, thus these sets are known in advance. The objective Equation 14 is to maximize the number of vertices that are mapped to some clique. Constraints Equation 15 and Equation 16 ensure that each NFA node is mapped at most once and that each clique of the overlay gets mapped to at most `Clique_Size` many times, i.e. every STE in a clique is used at most once. Constraint Equation 17 ensures that all edges of the NFA graph will be preserved in the overlay. This is the fundamental constraint that ensures that all transitions of the NFA can be executed in the hardware. If the optimal objective value equals to the number of vertices in the NFA graph, then we can successfully map the NFA onto the overlay graph, else there is no such mapping. We will discuss this formulation in subsection 5.3.

We turn now to step 3. In subsection 3.3.1 we discussed how the basic trigger predicates that match on ECI message type, direction and VC and composed predicates **Any** and **None** are implemented in hardware and how they can be configured at runtime. Generating the actual configuration for each hardware element is a matter of implementing these simple steps in code.

A major difficulty of step 4 is making sure that frontend and hardware modules agree on the format of the config string. Because the configuration bitstring is shifted sequentially into all configurable hardware modules, the order of the individual substrings of the produced config must match exactly the order in which the hardware modules are chained together (c.f. the explanation of CFGLUT5s in subsection 1.6). Things like endianness of the serial protocol (PCIe in our case) and the order in which the configuration file is written to the tracing engine are additional moving parts that make it hard to have the frontend and the backend correspond. The core problem is that the frontend is independently written python code that is not directly linked to the HDL description of the tracing engine. A consequence of this missing link is that seemingly simple changes in either the frontend or in the HDL code, such as adding some runtime configurable register to control a further aspect of the tracing engine or changing the order of the config chain, usually force us to re-think big parts of the configuration generation. Right now this is handled by careful hardcoding but we plan to make it less error prone at a later stage.

3.4.1 Making the frontend generic

One design goal of the tracing engine is to provide flexibility, both at runtime and at compile time. In subsection 3.3, we discussed how we achieve compile time flexibility by the modular structure of the HDL design: The tracing engine can quite easily be adapted to interfaces that are different from the VC layer by providing suitable implementations of the `Input_Reduction` and `Input_Decoding` modules. All other modules can stay the same. We extended this modularity to the frontend. Note that major tasks of the frontend are completely independent of the actual properties of the input data: In step 1, most of the textual NFA description is generic, only the specification of trigger predicates depends on the input decoding. In step 2, all NFA transformations and the mapping problem are independent of the input data. However, there is one subtlety: in order to transform the NFA into a homogeneous NFA, we need to be able to decide whether two transitions are equal or not. This depends on the transition triggers that the input decoding implements and on the way these triggers are specified in the YAML definition. Moreover, we need to

be able to distinguish epsilon transitions from ordinary transitions. We do this by using “eps” as a fixed keyword in place of a trigger predicate to denote epsilon transitions. All of step 3 depends on the concrete input decoding. Step 4 only depends on the order in which the config is pushed into the different hardware modules. If only the `Input Reduction` and `Input Decoding` modules are swapped out, the config length will vary, but the order of the individual config strings will stay the same. Thus step 4 will work without any changes, given that the configuration for the individual `Input Decoding` units are correctly generated in step 3.

All this input-specific functionality can be abstracted into a function that translates a trigger predicate, given as a string of text, into a configuration for the `Input Decoding` module. In step 1, when parsing the textual description of the NFA, the trigger can be kept as a plain-text string. In step 2, we can decide whether two transitions are equal by generating the config for both of them and checking if they are equal. In step 3, we simply use this function to generate the input decoding bitstring from the plain-text.

Our implementation allows the user to provide such a function and can thus compile NFA descriptions for arbitrary input formats and trigger predicates. Similar to how the modularity of the tracing engine allows us to specialize it for various data sources by only changing its `Input Reduction` and `Input Decoding` modules, our frontend implementation allows us to specify and compile tracing engines for all these different data sources by only providing a function mapping filter predicates to a configstring. We will re-visit this claim of generality and modularity in [subsection 4.1](#).

3.5 Control and output streaming over PCIe

In the following we will briefly discuss how the tracing engine is controlled and interacted with by a user.

The tracing engine is controlled from a remote host over PCIe and the filtered output stream is written to the same remote host, also over PCIe. In our concrete setup, the remote host is the CPU on the same Enzian as the FPGA that hosts the tracing engine, however, we call it a remote host to distinguish between the two roles of the Enzian CPU as ECI endpoint and as controller of the tracing engine.

The XDMA/bridge Xilinx IP core makes the FPGA show up as a PCIe device on the CPU. A PCIe device has three memory regions: the config, I/O and memory space. The config space has predefined size and structure and contains relevant information about the PCIe device that enables the CPU to properly initialize it using a suitable driver. When the CPU probes and initializes the PCIe device, it will access its configuration space and get the information of what physical addresses contain the I/O and memory space of the PCIe device. On the FPGA side, we can specify a set of I/O registers that are then accessible to the CPU. The CPU can then mmap these memory regions to access I/O registers and memory on the PCIe device. We specify control and status registers for communication in both directions. Some important control registers are: *config* to transmit the configuration bitstream to the tracing engine, *packet limit* to limit the length of the output stream, *enable* to start tracing, *reset* to reset all internal state except the configuration of the tracing engine, *CL mode*, *start CL*, and *ncls* that collectively control the substream mode of the tracing engine. Using these registers, the remote host has full control over the tracing engine. Additionally, the following status registers allow the user to inspect the internal status of the tracing engine: the *done* register is asserted as soon as *packet count* many packets have been emitted on the output stream, *overflow* indicates that the buffer of the PCIe has overflowed and the output stream is incomplete, and *packet count* keeps track of the length of the output stream that was emitted to the remote host.

Apart from communicating via the config space, the XDMA/bridge core also establishes a DMA channel between the FPGA and the CPU. The FPGA-side interface to the DMA channel provided by the Xilinx core uses a simple valid-ready handshake where 32 bits per PCIe lane (i.e. 256 bits in our case of x8 PCIe) can be transferred per cycle. The tracing engine uses this interface to write out the filtered trace. On the CPU, a

DMA engine handles the incoming data and dumps it into some location chosen by the user.

4 Experiments

In the following we will discuss three experiments, one for each use case discussed in [subsection 1.7](#): The first experiment will use the tracing engine as a simple message filter to reduce the size of the produced trace and facilitate its post-processing. The second experiment will use NFAs to characterize the cache miss behaviour of an application by looking at ECI coherence traffic. In the third experiment we will formalize how much the tracing engine can know about the current MOESI state of a CL. From this, we will derive an NFA that can detect certain error cases of the coherence implementation on Enzian.

Apart from discussing these concrete use cases, the goal of this section is to demonstrate the versatility of the tracing engine.

4.1 Simple trace filters and modularity of the design

In this experiment to demonstrate two things

- Simple NFAs can be used to filter out unimportant messages and thus facilitate post-processing by reducing the size of the trace.
- The design we described in [subsection 3.3](#) is modular and can be used not only to filter VC layer traffic, but it can easily be adapted to process different data streams. Here we adapt it to trace block layer data.

As example use case we will perform a bottleneck analysis of the throughput from FPGA DRAM to the CPU. To understand the bottlenecks in the ECI stack, it turns out to be useful to consider both VC layer and block layer messages. Before tracing block layer data, we discuss how our design from [subsection 3.3](#) can be adapted to the block layer. Concretely, we re-implement the functionality provided in the previous work [9]. Thinking back to [Figure 13](#), adapting the tracing engine to a new input source requires us to modify the `Input Reduction` and `Input Decoding` modules. All other modules can be left as they are. (Additionally, it is necessary to modify the widths of the data path signals in the whole tracing engine.)

There are several differences between block and VC layer traffic:

- Because all VC layer traffic is encapsulated in block layer messages, the data rate of the block layer is higher than the data rate of the VC layer. As discussed in [subsection 1.3](#), at the block layer we receive up to 1.5 blocks per direction and per cycle, each block consists of a 64 bit header and seven 64 bit data words, giving a total of 1536 bits per cycle. However, the input data width per se is of limited importance because we perform input reduction prior to sending the data to the NFA. The block layer is more amenable to data reduction than the VC layer because there are much fewer types of blocks.
- Block layer messages have a totally different structure than VC layer messages. This requires us to perform a different data reduction and consequently also different input decoding. Our design facilitates these changes by encapsulating all necessary functionality in the `Input Decoding` and `Input Reduction` modules.
- The different VCs at the VC layer accommodate a whole range of protocols. We focused on the coherence protocol so far. The protocol controlling the block layer, however, is a very different protocol potentially requiring different filters to be formulated. Here, the generality of NFAs allows us to express interesting patterns of these totally different protocols.

```

// incoming block
input[0] = is SYNC block;
input[1] = is HI/LO data block;
input[15:2] = { $\text{\textbackslash gls\{vc\}}_i$  gets data for  $i = 0..13$ };
input[18:16] = Estimated state of input link state machine;
input[20:19] = Request type (for sync block headers);
input[21] = Ack;
// outgoing block
input[32] = is SYNC block;
input[33] = is HI/LO dataa block;
input[47:34] = { $\text{\textbackslash gls\{vc\}}_i$  sends data for  $i = 0..13$ };
input[50:48] = Estimated state of output link state machine;
input[52:51] = Request type (for sync block headers);
input[53] = Ack;

```

Figure 24: Content of the reduced block layer input that is fed to the tracing engine.

The concrete implementation of Input Reduction and Input Decoding modules reuses much of the code from Kuchler [9]. We quickly summarize some important points of the implemented functionality.

The previous work [9] focuses on header data in block layer messages and ignores their payload. This leaves the tracing engine with two headers of 64 bits each per cycle. This data is further reduced to a 64 bit input containing important fields of the two block headers as well as data derived from them. The structure and content of the reduced input is depicted in Figure 24. In particular, the type of each header, the indices of VCs that receive data in data blocks, and control bits are included.

The estimated state of input and output link is derived from the observed headers within the data reduction module. The module re-implements the actual link control state machine. The input decoding logic then allows to formulate an arbitrary logic function on 8 selectable bits from the input, 4 from input[31 : 0] and 4 from input[63 : 32].

We choose the interface between interlaken layer and block layer to tap block layer messages. This is depicted in Blue in Figure 10.

We now analyze the bottlenecks that limit streaming throughput from FPGA DRAM to the CPU. In particular, we run a multi-threaded bandwidth benchmark on Enzian while tracing both VC and block layer messages. To trace both VC and block layer, we require two different instances of the tracing module. While it might be possible to instantiate both instances in the same design, we choose to synthesize two separate designs, one tracing only VC layer data and one tracing only block layer data. We then run the benchmark twice, in one case tracing VC layer data and in the other tracing block layer data. Everything apart from the tracing engine is equal in both designs. The main difficulty of having two separate instances of the tracing engine in the same design would be to synchronize their access to the remote host over PCI. The benchmark starts 16 threads on 16 cores that each stream CL-by-CL through 4 kB of consecutive memory. The memory region of thread i is $[0xi000, 0x(i + 1)000]$. Each thread streams 20 times through its memory and in between threads get synchronized with a barrier. Executing a separate latency benchmark tells us that the latency of requesting a single CL on the FPGA is around $1.3\mu s$. (run `mb_enzian.c -t`) Each core has a single load/store unit, i.e. it can process only one memory request at any given time. As each read request reads a full CL of 128 B, running the streaming benchmark on 16 cores in parallel should achieve around $(16 \cdot 128B)/(1.3 \cdot 10^{-6}s) \approx 1.57GB/s$ However, running the throughput benchmark we only measure

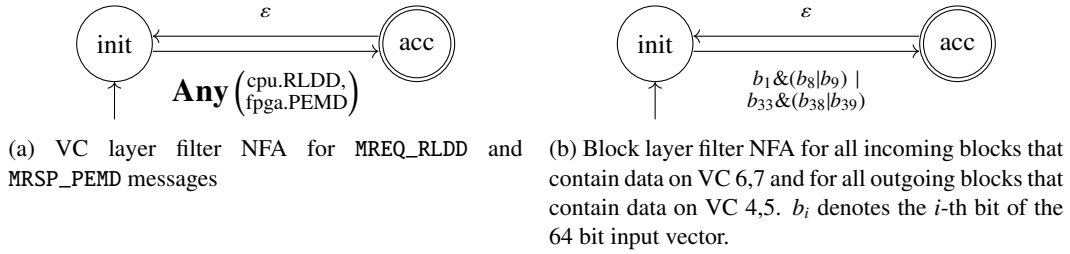


Figure 25: NFAs to filter for memory requests and responses.

0.25GB/s

To understand what limits performance, we are interested in observing requests and responses for different memory locations over time. In particular, we want to produce a trace containing all MREQ_RLDD and MRSP_PEMD messages, both at the block and at the VC layer, together with a timestamp of their arrival and the address they access. On the VC layer, the address is included in the message header. The timestamp of each message is appended to every message by the Output Reduction module as discussed in [subsection 3.3.4](#). To trace only the messages of the desired type, we use the NFA in [Figure 25a](#). Note that our input decoding logic from [subsection 3.3.1](#) conveniently allows us to recognize both types of messages in one instance of the Input Decoding, so we only need one accepting and one initial state to filter out exactly the messages we are interested in. We parse the stream VC layer messages in post-processing using the previous work [4]. On the block layer, we cannot filter by ECI message type because the input reduction and decoding does not inspect the payload of blocks. However, we know that MREQ_RLDD will arrive on VC 6 or 7 on an incoming data block. Similarly, MRSP_PEMD will be sent on VC 4 or 5 on an outgoing data block. We can thus trace all blocks that are either incoming and contain data for VCs 6/7 or are outgoing and contain data for VCs 4/5 as depicted in [Figure 25b](#). The resulting trace can be parsed with code from previous work [4] and a simple application of *grep* will reduce the trace to the desired blocks. From the payload of these blocks we can extract the VC headers of the MRSP_PEMD and MRSP_RLDD messages.

From both traces we have obtained a list of timestamps and corresponding VC headers of type MREQ_RLDD and MRSP_PEMD containing the CL address. We first look at the data from the VC layer tracing (not shown) to find that the average latency of a memory request at the VC layer is $0.135\mu s$. This is the time it takes the DirC to process the request and generate a reply. In particular, this involves popping the ECI message from the receive side VC FIFOs (Rlk FIFO), waiting for round-robin arbitration to schedule the message, the processing time at the DirC, routing the response back to the correct VC and pushing the message onto the send side VC FIFO (Tlk FIFO) (c.f. [Figure 10](#)). Looking at the VC layer data, we also find that there are never more than 2 in-flight requests. Ignoring the CPU-side latency and the transport across ECI, this suggests an upper bound for bandwidth of $(2 \cdot 128B) / (1.35 \cdot 10^{-7}s) \approx 1.9GB/s$. This bound is not satisfactory as it is not tight at all. Moreover, we assumed there to be up to 16 in-flight requests at any given time, one per core.

Next we consider the block layer trace data as displayed in [Figure 26](#). The Figure plots MREQ_RLDD and MRSP_PEMD messages with the timestamp of when they were processed in the Output Reduction on the x-axis and the CL address they concern on the y-axis. Request-response pairs are connected with a blue horizontal line. The blue line indicates the latency of this request at the FPGA block layer. Many interesting properties of our workload and the Enzian system can be derived just from looking at this plot:

1. A single thread never has more than one request outstanding as can be seen by the red MREQ_RLDD

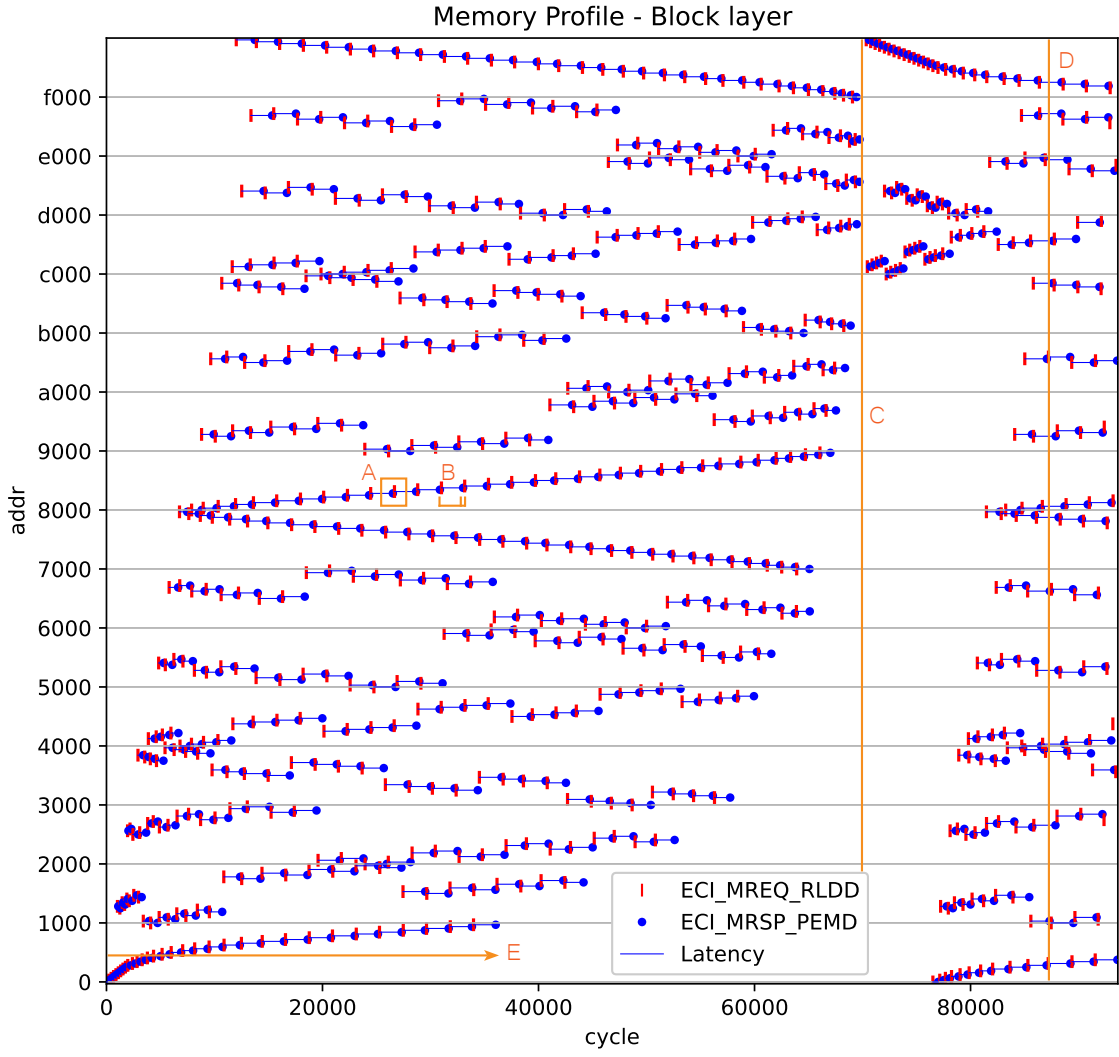


Figure 26: Memory requests (red line) and responses (blue dot) as traced on the block layer when running the streaming bandwidth benchmark with 16 cores. Core i streams through addresses $[0xi000, 0x(i + 1)000]$. Requests and responses are plotted against the memory address they reference and the clock cycle at which they were captured. Blue lines indicate the block layer latency of each memory request. Plot annotations in orange highlight several observations.

markers never crossing a horizontal blue line as indicated for instance by the orange box “A”.

2. In total there are often 16 requests outstanding at the FPGA, one per thread. This can be seen for instance as the vertical orange line “D” crosses 16 blue latency lines.
3. Performance is mostly determined by the latency of the ECI stack on the FPGA side (blue horizontal

lines). Most of the time is spent waiting for the VC layer on the FPGA while the processing between requests is only minimal. This is indicated at annotation “B” where the response message PEMD arriving after a long latency is followed almost instantly by a new request message RLDD.

4. Some cores spend a large fraction of their time waiting for the other cores to finish reading their memory slice. This is due to the barrier synchronization between threads. Thread 0 for instance waits about 1/3 of the whole time we plotted. The barrier is indicated by vertical orange line “C”.
5. The threads don’t start simultaneously after synchronization at line “C”. The threads that finished last start first. A possible explanation of this is that the threads are executed on separate cores and that communication latency between cores is not uniform.
6. Request latency (blue horizontal lines) are largely determined by congestion: the more requests are outstanding at the FPGA, the longer it takes for a request to be answered. This can be seen from latency being short while few threads are running and long while many threads are running. This is indicated for thread 0 by orange arrow “E”.
7. Despite all threads accessing memory linearly from lowest to highest address, we observe a variety of different access patterns (e.g. thread 7 accesses memory from highest to lowest address). This is due to a mapping from physical address to CL index. The CL index c is determined from the physical address a by

$$\begin{array}{rcccccc}
 & a[39 : 20] & a[19 : 15] & a[14 : 12] & a[11 : 10] & a[9 : 7] \\
 \oplus & & a[24 : 20] & a[27 : 25] & a[24 : 23] & a[22 : 20] \\
 \oplus & & & & a[13 : 12] & a[14 : 12] \\
 \hline
 & c[32 : 13] & c[12 : 8] & c[7 : 5] & c[4 : 3] & c[2 : 0]
 \end{array}$$

For our analysis, observations 1 to 6 are most relevant. These observations suggest that our interpretation of the VC layer results were wrong. Observations 1 and 2 match our expectation that every core only has one load/store unit but that the requests of different cores are processed in parallel at the FPGA. Measuring the average latency of a request at the block layer gives $5.2\mu s$, i.e. this is the average time between arrival of an MREQ_RLDD request at the block layer and the arrival of the corresponding MRSP_PEMD message at the block layer in the opposite direction. This is the time it takes between processing the request at the block layer, pushing it onto the VC FIFO, for the DirC to process it and generate a reply, and for the block layer to send the reply (c.f. Figure 10). Using the block layer latency of a request and the fact that we have 16 in-flight requests, we get an upper bound on the bandwidth of $(16 \cdot 128B)/(5.2 \cdot 10^{-6}s) \approx 0.4GB/s$. This is still about twice what we’re observing. Observation 4 suggests that many cores are idle for a large portion of the time. This is due to a memory barrier that synchronizes the blocks after having read all their memory. Moreover, observation 5 suggests that waking up a sleeping core also introduces some overhead. Note, that CPU 0 only recommences shortly after CPU 15 has read all its data. It seems plausible that the synchronization overhead of the barrier accounts for the largest fraction of the discrepancy between computed bandwidth bound and observed bandwidth. Moreover, CPU side latency and the Interlaken layer will add some additional latency to this.

The block layer latency of $5.2\mu s$ and the idle time of cores seems to explain most of the low throughput of 0.25GB/s that we are measuring.. The open question is why the latency of one request is so high at the block layer if it is only $0.135\mu s$ at the VC layer. Clearly, most of the parallel in-flight requests that we observe at the block layer get serialized at the VC layer: At the block layer we observe 16 in-flight requests that get serialized to only 2 in-flight requests on the VC layer. The effect of this serialization at the VC layer can also be observed in the block layer trace: Observation 6 suggests that a request at the block layer must

#Data words	Content	VC
80543	Empty data words	-
29713	PEMD response data	4,5
1830	RLDD headers	6,7
1830	PEMD headers	4,5
422	GINV/GSYNC	6,7

Table 3: Data words of captured blocks grouped by content.

wait longer if there are more other requests waiting to be processed, i.e. it must wait until all other requests have been processed. However, assuming the parallel processing of 2 requests at the VC layer explains only $8 \cdot 0.135\mu\text{s} \approx 1\mu\text{s}$ latency at the block layer, but we observe $5.2\mu\text{s}$ latency.

Thus there is some other bottleneck at the VC layer. To understand this bottleneck we need to understand a bit about the implementation of the DirC. As discussed in [subsection 1.4](#), the DirC uses an RTG to store information about all CLs that are home on the FPGA and that are currently cached on the CPU. The RTG also has an entry for all requests that are currently being processed in the DirC. If the RTG of the DirC is full, it needs to evict an entry from it before continuing to process the new request. It will force the CPU to evict one of its cached CLs that are home on the FPGA. To do so, it sends a forward request to the CPU and stalls until it receives the forward acknowledgement from the CPU. This is in line with the coherence protocol discussed in [subsection 1.4](#). After it receives the acknowledgement from the CPU, it will process the new request by fetching the requested CL from DRAM or cache. Stalling is necessary because the DirC cannot send off this request to DRAM/cache until the point where it can write its state into the RTG. After writing the current request into the RTG and sending the request to DRAM/cache, the DirC will start processing the next request. The current implementation of the DirC uses a dummy RTG with only 16 entries. Thus all but the first 16 CPU L2 misses for CLs that are home on the FPGA will require evicting an entry from the RTG. If the size of the RTG were chosen to match the size of the CPU L2 cache, it would never be a bottleneck. Moreover, the resulting stall of each memory request is linearized in the DirC because the DirC cannot process further requests until there is space in the RTG. On the other hand, DRAM/cache lookup latency can be partially hidden as it can be overlapped with the stalling of the next request. We would thus expect the request latency of $0.135\mu\text{s}$ at the FPGA VC layer to correspond approximately to the stall time of a request, i.e. to the round-trip-time between FPGA side DirC and CPU side L2 cache.

Because this stall time is linearized for all memory requests, we expect to handle the 16 in-flight requests that are outstanding at the block layer in about $16 \cdot 0.135\mu\text{s} \approx 2.1\mu\text{s}$. Thus the expected latency of a memory request at the FPGA block layer is $2.1\mu\text{s}$, which is still about a factor 2.5 away from the observed latency of $5.2\mu\text{s}$.

Even though we cannot explain all of the latency at the block layer, this analysis has provided us with many insights into the Enzian system. All analyses in the subsection could have been done by post-processing raw traces. In this scenario, the major benefit of filtering the traces on board is to reduce their size. At the block layer, we receive about 1.5 blocks of 512 bits each cycle. The full trace considered in [Figure 26](#) spans about $1.6 \cdot 10^9$ cycles of 300MHz clock of the FPGA. This corresponds to the approximately 5 seconds it took to start tracing and run the benchmark. Capturing a raw block layer trace would thus have resulted in approximately $2.4 \cdot 10^9$ blocks with a total size of 1.2TiB. Each block contains 7 data words of size 64 bits, i.e. a total of $1.6 \cdot 10^{10}$ data words would have been captured.

[Table 3](#) summarizes the major contents of the data words in the filtered trace. Most data words are empty because every data block contains 7 data words but often only 1 data word is used. The second largest group

of data word-types contain CL data. For every MREQ_RLDD request, a MRSP_PEMD response will send back 128B of data which corresponds to 16 data words. We are only interested in the 3660 data words containing request and response headers, that is in about 3% of the data words we actually capture. However, had we processed the unfiltered trace, the fraction of interesting data words would shrink to 0.000022%. It is thus very beneficial to be able trim down traces to the relevant bits using the tracing engine. In contrast to the block layer, there is no constant stream of idle messages at the VC layer, thus the amount of useless traffic is considerably lower. On the VC layer, the amount of traffic we want to ignore depends on what other workloads are running on Enzian at the time of tracing. Even if the benefit of this simple tracing is smaller at the VC layer, depending on what we want to analyze and what other workloads are running, it is still very useful to be able to trim down the trace to the essential data.

4.2 Characterizing cache misses

In this experiment we want to demonstrate the following

- The tracing engine allows us to inspect and understand cache miss behaviour of the Enzian system.
- Tracking multiple substreams of ECI messages separately is useful because it allows observing the state of different CLs with separate instances of the NFA.

Being able to directly observe coherence traffic allows us to characterize the cache misses of a given workload. Consider the common characterization of cache misses into

- Compulsory c_0 : the first access to a CL always incurs a miss
- Capacity c_1 : a CL has been evicted because the working set is larger than the cache and now incurs another miss
- Conflict c_2 : a CL has been evicted despite the working set being smaller than the cache because too many other cachelines mapped to the same set and now incurs another miss
- Coherency c_3 : a CL has been evicted due to some other core accessing it and now incurs another miss

Note that we can't distinguish capacity from conflict misses by observing ECI messages alone because we have no information about how many node-local CLs are resident in a node's cache. The NFA depicted in [Figure 27](#) allows us to distinguish compulsory from capacity/conflict and from coherency misses on some fixed CL: Initially, we are in state s_0 waiting for the first request of a fixed CL x . The first request for x is a compulsory miss, thus we transition to C_0 . Instantly, we transition also to state s_1 using an ε transition. In particular, C_0 stays active only one cycle. In s_1 we wait for further messages concerning x . Witnessing either V21 or V31 tells us that the remote node has voluntarily evicted x . Consequently, a subsequent request R12 or R13 constitutes either a capacity or a conflict miss. Similarly, witnessing either F21 or F31 tells us that the home node has forced the remote node to invalidate x . Consequently, a subsequent request constitutes a coherence miss. Having observed a coherence or capacity/conflict miss we again transition immediately to the waiting state s_1 using an ε transition. Note that the number of times state C_i is active thus corresponds to the number of misses of type i .

We run the following workload on the CPU: A perfectly balanced binary search tree with entries 0 to 510 as depicted in [Figure 28a](#), is stored in 3072B of FPGA DRAM. The nodes are laid out in memory from smallest entry to largest entry. We now perform a sequence of n lookups of random even numbers $0, 2, \dots, 510$. Since all even numbers are at the leaves of the tree, every lookup visits exactly 8 nodes in the

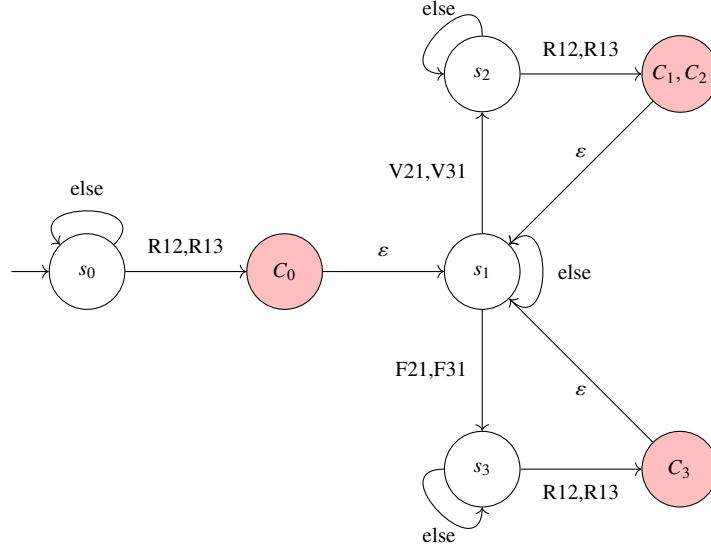
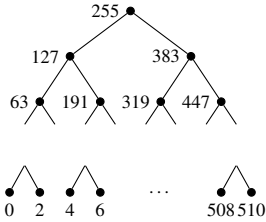


Figure 27: Life-cycle of a CL with different types of misses. States corresponding to cache misses C_0 , (C_1, C_2) and C_3 are red. Transition labels use the notation introduced in [subsection 1.4](#).

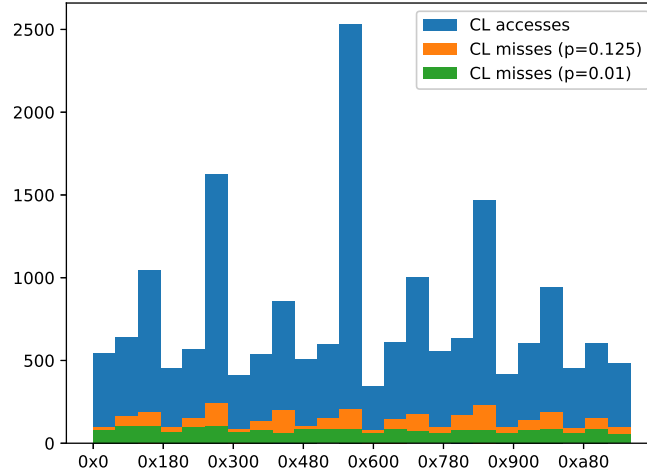
tree. Note that this setup guarantees that a node at depth $i < 8$ in the tree is accessed with probability 2^{-i} at every lookup. In particular, the root gets accessed at every lookup, each of its children get accessed with probability 0.5 at every lookup, and so on. The blue bars in [Figure 28b](#) show the number of accesses for each CL of one particular run of the benchmark. As expected, the number of accesses follows approximately the distribution described before. In order to get consistent results between different runs we generate all randomness in the benchmark from a pseudo-random sequence with a fixed seed, thus the accesses are always exactly as depicted by the blue bars in [Figure 28b](#). Additionally, at every lookup, we perform a CPU-memory intensive computation with probability p . By CPU-memory intensive we mean a computation on the CPU that accesses a big chunk of the physical address space of the CPU. This computation flushes the whole cache.

We are now interested in characterizing the cache misses that this workload generates. To determine the misses of each of the three types, we run the benchmark three times, each time we trace the ECI messages with the above NFA, only changing the set S of accepting states of the NFA: the first NFA filters compulsory misses and has $S = \{C_0\}$, the second NFA filters capacity/conflict misses and has $S = \{(C_1, C_2)\}$, and the third NFA filters coherence misses and has $S = \{C_3\}$. This results in three different traces, each trace consisting of memory request messages corresponding to the given type of cache misses. Capturing a separate trace for each type of cache miss is necessary because the type of cache miss is not apparent from the captured message itself. As a sanity check, we run the benchmark a fourth time with $S = \{C_0, (C_1, C_2), C_3\}$ to count the total number of misses. Moreover, note that we need to track the state of every CL independently from all other CLs. We thus need to process $\frac{3072}{128} = 24$ substreams, one per CL. To do this we use the mechanism described in [subsection 3.3.1](#).

We perform the benchmark twice, each time with $n = 2048$ lookups. The first time we set probability $p = 0.125$, the second time we set $p = 0.01$. [Figure 29](#) shows heat plots of the number of cache misses



(a) Sketch of perfect binary search tree with 511 entries and height 8



(b) Total number of accesses to each CL (blue). Total number of cache misses per CL for two different benchmarks (orange and green).

for all the CLs that hold the binary search tree. We don't show heatmaps for the compulsory cache misses because all CLs have exactly one compulsory miss in both cases.

To understand the results of this experiment we need to know a bit about the FPGA-side application: There is no user-space application running on the FPGA that would access memory and interfere with data cached on the CPU. All ECI traffic coming from the FPGA is thus produced by the FPGA-side directory controller. The DirC mostly reacts to the CPU's requests. The only case where the DirC acts without an external trigger is when its RTG is full. As discussed before, the RTG of the DirC keeps track of CLs that are home on the FPGA but that are currently stored on the CPU. The current DirC implements a 4-way associative RTG with 4 sets, i.e. it can store 16 tags in total. When the DirC receives a memory request while its tag store is full, it needs to evict a tag. The tag to evict is chosen randomly from all tags in the same set as the new request. Which tag is evicted is chosen depending on the current clock cycle, i.e. it looks truly random to a CPU-side application.

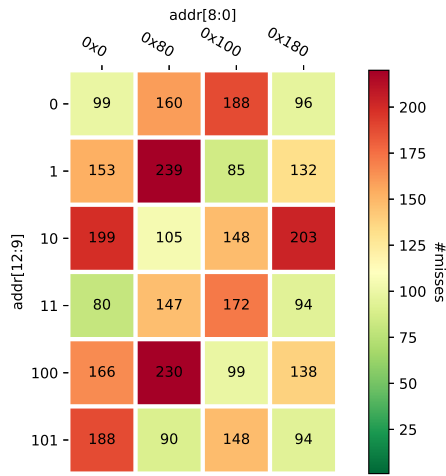
The numbers in Figure 29 are not completely conclusive: we would expect that the total number of cache misses for each CL equals the sum of the cache misses of the three types

$$c = c_0 + c_{1,2} + c_3$$

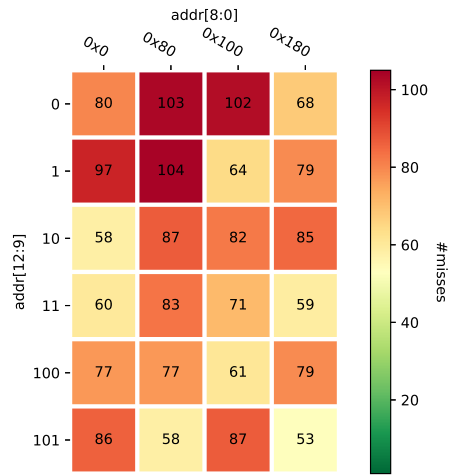
In particular, the numbers in subplot Figure 29a should be 1 minus the sum of Figure 29c and Figure 29e and similarly for Figure 29b. In reality, we see that sometimes $c > c_0 + c_{1,2} + c_3$ (e.g. CL 0x0 for $p = 0.01$) and sometimes $c < c_0 + c_{1,2} + c_3$ (e.g. CL 0x180 for $p = 0.01$). This is an artifact of the experiment setup: We need to run the benchmark four times to count the four different classes of cache misses. The randomness from the directory controller choosing victim tags to evict will thus result in slightly different caching behaviour each time.

Despite this caveat, there are some interesting conclusions we can draw from Figure 29 about our benchmark:

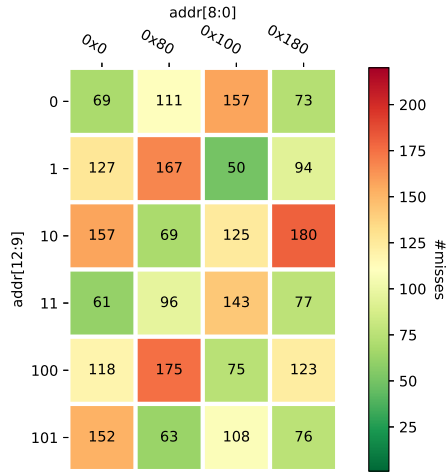
First, for $p = 0.125$, there are many more conflict/capacity misses than coherence misses, while for $p = 0.01$ it is the other way around. Again, this makes sense because for $p = 0.125$ we perform many more



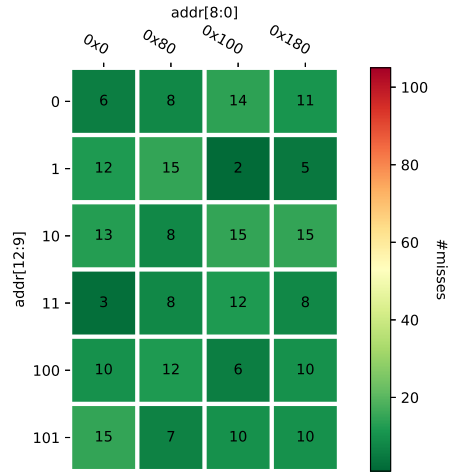
(a) All misses with $p = 0.125$



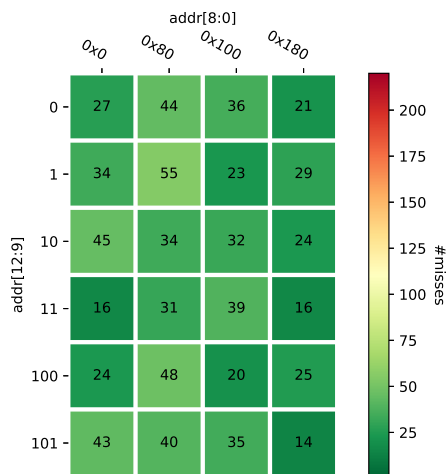
(b) All misses with $p = 0.01$



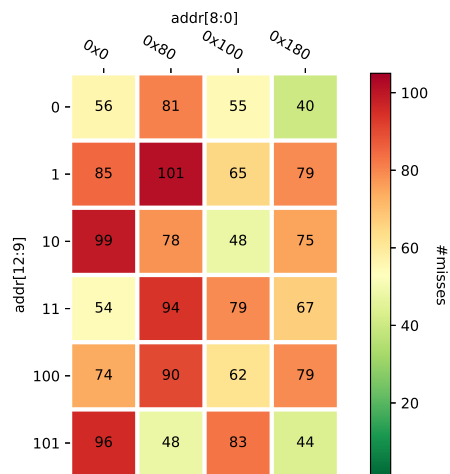
(c) Conflict and capacity misses $p = 0.125$



(d) Conflict and capacity misses with $p = 0.01$



(e) Coherence misses $p = 0.125$



(f) Coherence misses with $p = 0.01$

Figure 29: Classification of cache misses for the binary search tree benchmark with $p = 0.1$ and $p = 0.125$. Heat-plots show the number of misses on each CL holding parts of the binary search tree.

cache flushes. Knowing whether conflict/capacity misses or coherence misses are dominant is very useful when trying to improve the performance of a program. When most misses are capacity/conflict misses we can improve performance by rewriting the CPU-side program alone. In our artificial example the periodic cache flushes are supposed to simulate a memory-intensive computation that has to be performed regularly. Note that reducing the frequency at which this computation needs to be performed (i.e. decreasing p), not only leads to a shift from capacity/conflict misses to more coherence misses, but it also decreases the total number of misses. On the other hand, when many coherence misses are happening we likely won't see much improvement by changing the CPU-side program alone. The problem in this scenario is that CPU and FPGA share too much data, i.e. that the work is not so well partitioned.

Second, the hotness pattern of CL accesses is quite clearly reflected in the number of cache misses when $p = 0.125$. However, for $p = 0.01$, this is much less the case (c.f. also [Figure 28b](#)). That is, a CL that gets accessed often will also incur many conflict/capacity misses but not many coherence misses. This is expected because the memory intensive computation will flush the whole cache every few requests. Thus CLs that get accessed more often will also be flushed more often. On the other hand, all CLs incur about the same number of coherence misses. This may seem unintuitive at first but consider the case where all cache misses are coherence misses. Consider two CLs x and y and assume x gets accessed twice as often as y . That is, on average, the time between two consecutive accesses to x will be half the time between two accesses of y . Since our lookups in the binary tree are random, we can thus expect that there are about twice as many CL misses happening between two accesses of y than there are between two accesses of x . Recall that the directory controller chooses a tag to evict at random every time it receives a request. Thus the probability that the directory controller evicts the tag of y between two accesses of y is about twice the probability that it evicts the tag of x between two accesses of x . Thus, the more often we access a CL, the less often will it happen that it gets evicted between two accesses. These two effects cancel each other out and the expected number of times we miss a given CL is the same for all CLs. More formally, we can compute the expected number of misses of each CL x in this scenario: Denote $A(x) = t_0 < t_1 < \dots < t_k$ the sequence of times where we access x

$$\begin{aligned} E[\#\text{misses } x] &= \sum_{t_i \in A(x)} \Pr[\text{miss } x \text{ at } t_i] \\ &= \sum_{t_i \in A(x)} \Pr[x \text{ was evicted in } (t_{i-1}, t_i)] \\ &\stackrel{*}{=} \sum_{t_i \in A(x)} z \cdot (t_i - t_{i-1}) \end{aligned}$$

In (*), z is the probability of x being evicted at any given time. Because the number of cachemisses per time is expected to be approximately constant, z is approximately equal for all CLs and over the whole run of the program. Thus, we can rewrite the last sum

$$\approx |A(x)| \cdot p \cdot \frac{T}{|A(x)|}$$

where T is the duration of the whole program. Note that the number of accesses $|A(x)|$ of x cancels out. Again, knowing whether the hotness of certain memory addresses is actually reflected in the cache misses is very important when trying to optimize this program.

We have seen that the tracing engine allows us to track interesting coherence data of a workload. We have chosen an artificial workload example where the memory access behaviour can be explained analytically. However, for real workloads it is often impossible to derive a suitable model for the memory accesses. In

these cases, the tracing engine can help the user to understand the behaviour of its workload and to optimize it accordingly. Tracing coherence data requires us to keep track of a separate state for each CL. This is only possible because we allow tracking multiple independent substreams in the tracing engine.

4.3 Validating Enzian coherency

One motivation for implementing the tracing engine is to validate and debug the FPGA side implementation of the ECI stack. The ECI stack entails various protocols for the different VCs and a separate protocol for the block layer. The central protocol at the VC layer is the cache coherence protocol. Apart from being the most complex protocol, coherent access between CPU and FPGA is also a central feature of Enzian. Application specific implementations of the DirC can act as custom memory controllers for the FPGA DRAM to best support the workload.

However, all implementations of the DirC need to follow the basic rules imposed by the MOESI protocol, otherwise memory coherency between CPU and FPGA is no longer ensured.

In this experiment

- We provide an answer to the question to what degree the tracing engine can be used to validate a DirC implementation.
- We approximate the MOESI state of individual CLs and detect if the DirC behaves in an illegal way.
- We demonstrate the suitability of the input reduction and decoding we implemented in [subsubsection 3.3.1](#) for representing complicated predicates on ECI messages.

Recall from [subsection 1.4](#) that the state of any CL is kept in up to four different locations. In particular, the state of a CL that is home on the FPGA and cached on the CPU is kept in the CPU cache and twice in the DirC of the FPGA (HS and RS). Recall that HS and RS are an approximation of the state of the CL in the FPGA cache and the CPU cache, respectively. If the CL is also cached on the FPGA, the state is additionally kept in the FPGA cache. In order to validate the behaviour of an implementation of the DirC, we are interested in approximating the true HS and RS state in the DirC. That is, at any point in time and for any CL, we would like to know what HS and RS a *correct* implementation of the DirC would store. Given this knowledge we can judge if the *actual* DirC implementation behaves correctly or not.

A major difficulty of this is that the tracing engine has a different perception of the world than both the cache at the CPU and the DirC on the FPGA. In particular

- It observes messages in a different order than they are sent and received
- It cannot observe node-internal events. In particular, home can spontaneously up- and downgrade its access to a CL.

We can simplify the model by exploiting the fact that all messages except voluntary downgrades require an acknowledgement. Moreover, a node is not allowed to send another message for some CL if it has already a message outstanding and this message requires an acknowledgement. Re-ordering of messages is thus only possible if voluntary downgrades are involved. We can ignore voluntary downgrade messages and instead consider these downgrades to be invisible to the tracing engine. A state transition of both HS and RS can happen either *prompted* (i.e. upon reception of a message from the peer) or *spontaneous*. Ignoring voluntary downgrades means that all spontaneous transitions are invisible to the tracing engine.

Recall from [subsection 1.4](#) that the DirC only needs to distinguish between some of the MOESI states in order to do its work: For the HS it suffices if it distinguishes the invalid state I from all other states

HS\RS	I	S	{E, M}
I			
S			
{E, M, O}			

Table 4: Legal (green) and illegal (red) HS:RS pairs

$\{M, O, E, S\}$. For the RS it suffices if it distinguishes between I, S and $\{M, E\}$. However, in the tracing engine it is advantageous to distinguish between S and $\{M, O, E\}$ for the HS because it lets us represent the legal behaviour of the DirC more precisely and thus allows us to detect more error cases. Under these assumptions, the valid HS:RS pairs are summarised in Table 4. The interpretation of this categorization into valid and invalid state pairs is the following:

If a faithful representation of HS:RS ever transitions into one of the illegal state pairs, either the DirC or the CPU cache didn't follow the cache coherence protocol.

Because we are assuming that the CPU implements the cache coherence protocol correctly, we can infer that the DirC implementation has a bug. Note that the requirement of a *faithful* representation of HS:RS is necessary, because the *actual* HS:RS values are maintained by the DirC whose correct behaviour we can't assume.

To test this we must implement an NFA that approximates the *faithful* HS:RS. Because the tracing engine can't always know the exact HS:RS, we maintain the set of possible HS:RS pairs. More formally, we model the knowledge of the tracing engine about HS:RS at time i (i.e. after having received i messages) as a set S_i of HS:RS pairs. S_i is the set of *possible* state pairs at time i . In particular, the *true* state pair of a *correct* DirC implementation at time i must be in S_i for every i . Initially, the tracing engine has no knowledge about the system, thus S_0 is the set of all valid HS:RS configurations. We update S_i for every observed message to incorporate the newly gained information and to approximate the *correct* HS:RS as closely as possible.

To derive an NFA that achieves this, we formalize the effect of each message m on S by providing a *precondition* and a *postcondition* for it. The precondition $\mathbf{Pre}(m)$ describes the set of HS:RS in which m can be sent. The postcondition $\mathbf{Post}(m, s)$ describes for each possible current state s a set of new possible states after s has been sent. Thus, looking at the sequence of messages m_i received by the tracing engine, we can compute a sequence of possible state sets S_i as follows:

$$S_0 = \{\text{HS:RS} \mid \text{HS:RS is a valid state pair}\} \quad (19)$$

$$S_{i+1} = \bigcup_{s \in S_i \cap \mathbf{Pre}(m_i)} \mathbf{Post}(m_i, s) \quad (20)$$

Note that for a given CL the roles of home and remote are fixed. Thus each message can be sent in exactly one direction, either from home to remote node or from remote to home node. Pre- and postconditions need to satisfy the following properties:

- $\mathbf{Pre}(m)$ is the set of state pairs HS:RS in which the sender is allowed to send m .
- $\mathbf{Post}(m, \text{HS:RS})$ is the set of state pairs HS':RS' that can be reached from HS:RS by any combination of
 - the transition of receiver that is prompted by the reception of m
 - any spontaneous transitions of receiver after reception of m

m	Pre(m)	Post(m, HS:RS)
RA_y	$RS < y$	$RS \leq y \wedge \mathbf{compat}(HS, RS)$
A_{xy}	$RS = x$	$RS \leq y \wedge \mathbf{compat}(HS, RS)$
R_{xy}	$RS = x$	$RS = x \wedge \mathbf{compat}(HS, x)$
F_{xy}	$RS = s \leq x$	$RS \leq s \wedge \mathbf{compat}(HS, RS)$

Table 5: Pre- and postconditions for all messages (ignoring voluntary downgrades)

- any spontaneous transitions of sender

Table 5 lists the pre- and postconditions that we derived according to these rules. The syntax is similar to subsection 1.4, but we parametrize each message with x and/or y . For instance RA_y stands collectively for RA_2 and RA_3 . To formulate pre- and postconditions, we use comparison operators between states. In particular, $s_1 < s_2$ means that s_1 has weaker access rights than s_2 . Moreover, we write $\mathbf{compat}(s_1, s_2)$ to mean that state s_1 is constrained to values that are compatible with s_2 according to Table 4.

The precondition for RA_y states that the remote node must be in a state $< y$. That is, for RA_2 , the remote node must be in state I, for RA_3 , the remote node must be in state I or S. The reason for this is as follows: The RA_y message is sent by the home node. The home node is only allowed to send this acknowledgement if the remote node has previously requested an upgrade to state y . The remote node is only allowed to request an upgrade to state y if it is in a state $< y$. Because the tracing engine observes request-response pairs in the correct order, it must have observed the request to upgrade to state y . At this point, the tracing engine knew that $RS < y$. Moreover, since we’re currently observing the response RA_y , we know that the remote node has not yet received this response. Because the remote node is not allowed to send any more messages concerning the same CL before receiving RA_y , we know that the remote node has not sent any more messages. Moreover, it did not receive any other RA messages from the home node for this CL since sending the last request, because of the property that no new request can be sent while there is still one outstanding. Thus the only messages the remote node may have received since having sent its request are downgrade requests from home. From this we know that $RS < y$.

The postcondition for RA_y states that the remote node must be in a state $\leq y$ and that the home node must be in a state compatible with the RS. The remote state cannot be in a state $> y$ because in order to upgrade, it first needs to send another request which the tracing engine will observe before it is processed by the home node. However, the remote node can at any time downgrade its state. As discussed above, voluntary downgrades are invisible to the tracing engine, the only postcondition we can set for the home state, is that it must be compatible with the remote state. Recall that we want to approximate the behaviour of a *correct* DirC implementation, thus we can assume that HS and RS are always compatible.

Similar reasoning leads to the pre- and postconditions for the other message types. A major difficulty of this approach is that this kind of dynamic reasoning about HS and RS is very prone to error. In lack of a more formal approach, we chose to at least make all assumptions of the model explicit and accessible in one place in Table 5. The advantage of having these formal pre- and postconditions is that we can then derive the corresponding NFA completely automatically. We can specify an NFA $(\Sigma, T, S_0, \delta, \{X\})$ that tracks S_i and accepts sequences that result in an error state X . Σ is simply the set of valid ECI headers. The set of T is specified inductively from the initial state S_0 and the pre- and postconditions from table Table 5 as follows

$s \backslash m$	A11	A21	A22	A31	A32	F21	F31	F32	R12	R13	R23	RA2	RA3
X	X	X	X	X	X	X	X	X	X	X	X	X	X
010010	X	101001	111011	X	X	111011	111011	111011	X	X	010010	X	111111
101001	101001	X	X	X	X	101001	101001	101001	101001	101001	X	111011	111111
111011	101001	101001	111011	X	X	111011	111011	111011	101001	101001	010010	111011	111111
111111	101001	101001	111011	101001	111011	111011	111111	111111	101001	101001	010010	111011	111111

Table 6: Resulting NFA: if the tracing engine is currently in state s and observes message m , it should transition to the state indicated at entry $[s, m]$. The initial state is 111111, X denotes the error state.

$$S_0 \in T \quad (21)$$

$$A \in T \wedge m \in \Sigma \longrightarrow \left(\bigcup_{s \in A \cap \text{Pre}(m)} \mathbf{Post}(m, HS : RS) \right) \in T \quad (22)$$

Similarly, the transition function δ is defined by

$$\forall A \in T, m \in \Sigma. \delta(A, m) = \bigcup_{s \in A \cap \text{Pre}(m)} \mathbf{Post}(m, s) \quad (23)$$

Computing this NFA for the above pre- and postconditions results in the state transition table given in Table 6. We write states $s \in T$ as a bitvector where each entry corresponde to a valid HS:RS pair

$$s = [(\{E, M, O\}, I), (S, S), (S, I), (I, \{E, M\}), (I, S), (I, I)] \quad (24)$$

Entry s_i is 1 if the i -th HS:RS pair is currently possible. We write X to denote the NFA state where no HS:RS pair is possible. This constitutes the error case: We have excluded all valid HS:RS as possible states of a *correct* DirC implementation.

We can easily implement the NFA in Table 6 in the tracing engine. Note that our input decoding allows for convenient encoding of the necessary predicates: For instance, the transition from state 010010 to error state X triggers on predicate:

$$A11 \vee A31 \vee A32 \vee R12 \vee R13 \vee RA2 \quad (25)$$

These are all messages with a precondition that contradicts $RS = 2$, which we know from state 010010.

Table 7 lists the set of ECI messages that correspond to this trigger. Note that a basic coherence event like A11 can correspond to multiple ECI messages. Further, any ECI message can be sent over several VCs. That is, in order to correctly recognize this predicate, the Input Decoding module must be able to detect a complicated combination of various Opcodes on various VCs. Note that our Input Decoding module described in subsection 3.3.1 can express this complicated predicate without any problems and at minimal hardware cost.

We ran this filter alongside the current DirC implementation of Enzian while running a memory benchmark. However, the NFA never entered an accepting state because no error was detected. Thus the resulting trace was empty and did not reveal any further insights.

5 Evaluation

A central goal of this work is to provide a design with a high degree of runtime configurability for expressing interesting ECI message filters. Another key goal is that the design be as compact as possible such that it

Event	ECI message	Opcode	VC
A11	MRSP_HAKI	6	10
			11
	MRSP_HAKV	8	10
			11
A31	MRSP_VICDHI	3	4
			5
			10
			11
A32	MRSP_HAKD	4	4
			5
	MRSP_HAKN_S	5	10
			11
R12	MREQ_RLDI	1	6
			7
R13	MREQ_RLDD	0	6
			7
	MREQ_RLDX	5	6
			7
RA2	MRSP_PSHA	9	4
			5

Table 7: All coherence events that trigger transition from 010010 to error state X as given in [Equation 25](#), together with corresponding ECI messages, Opcode in the ECI header and the VC on which this message can be sent.

can be synthesized alongside major applications. These two goals are clearly contradicting each other: We can make the design highly flexible but this will come at an increased hardware cost. For instance, an Input Decoding module that can recognize many different predicates will be much larger than one that can only check a single predicate. Similarly, we can make the design very economical and small, but such a design will only have limited runtime configurability. As it is often the case in engineering, we need to compromise between conflicting goals. In this section we will evaluate to what degree we struck a good balance between hardware cost and expressiveness of the tracing engine. The experiments we performed in [section 4](#) show the flexibility of the tracing engine on concrete examples. In this section, on the other hand, we want to evaluate the main determinants of the expressiveness of the framework more generally.

First, the size and the density of the overlay graph limit the complexity of the NFA that can be mapped to it. On the other hand, making the overlay graph larger and denser will increase its hardware cost. In [subsection 5.1](#) we will evaluate this trade-off by measuring how the hardware cost of the tracing engine scales with the size of the overlay graph. We also provide an analytic hardware cost model of the overlay graph as a function of the overlay parameters. The goal is to provide the reader with an idea of what configurations will be synthesizable and how much hardware will be needed to implement a given configuration.

Second, the suitability of the basic predicates that are supported by the tracing engine limit how well the task at hand can be expressed as an NFA of these basic predicates. It is hard to evaluate the suitability of the basic predicates detached from a specific use case. In [subsection 5.2](#) we provide the user with a tool to estimate the size and density of the overlay graph necessary to implement a concrete filter and formalize

the notion of “suitable basic predicates”. We then consider the experiments in [section 4](#) again and conclude that the basic predicates we provide and the **Any** and **None** combinators are very suitable for the general use cases we are interested in.

Third, and maybe surprisingly, the frontend mapping the abstract NFA onto the overlay graph also limits the size of the NFAs that can be mapped. The mapping problem solved by the frontend is an NP-hard problem and for NFAs and overlays that are too large, finding an optimal mapping can take a long time. In [subsection 5.3](#) we will evaluate the mapping algorithm we implemented by comparing it to several other formulations. We show that our mapping algorithm works well for reasonable overlay sizes.

After this evaluation, we will consider the key design decisions that underlie the tracing engine: The choice of the rings-of-cliques configuration as overlay graph, the choice of using homogeneous NFAs, and the choice of using shift registers to load the runtime configuration.

5.1 Hardware resource scaling

In the following we want to evaluate the hardware cost of the tracing engine. In particular, we want to evaluate the share of each submodule on the total hardware cost of the tracing engine, how the hardware cost scales with each of the overlay parameters, and how big we can make the overlay without encountering timing problems. There are essentially three constraints that limit the complexity of a design that can be placed on an FPGA: First, the number of flip-flops and LUTs on the FPGA limit the number of registers and combinatorial logic blocks that can be placed. Second, the routing resources on the FPGA limit the number of connections between flip-flops and LUTs. Third, the constant goal clock rate of a design limit how far apart two connected registers can be placed and how direct the routing between them needs to be.

Hardware cost per submodule: Recall that the tracing engine design is parametrized by 5 constants:

- C is the size of each clique in the overlay graph
- L is the number of cliques that are connected to a ring
- R is the number of rings
- N is the connectivity between rings
- NCLS bounds the number of independently tracked cachelines.

The first four parameters increase size and density of the overlay graph (c.f. [Figure 16](#)), the fifth limits the maximum number of independent instances of the NFA that can be tracked.

[Table 8](#) shows the hardware cost of three different configurations of the tracing engine. Recall that the Virtex Ultrascale+ FPGA we are using has a total of 1.1M LUTs and 2.3M FFs. The hardware cost is listed by module. Modules that contain other modules are listed twice, once with the cost of all submodules included (indicated by *) and once only counting their own LUTs and FFs. The modules correspond largely to [Figure 13](#). Additional modules not shown in [Figure 13](#) are outside the tracing engine and are used to control it. In particular, the Control module is the outermost wrapper module of the tracing engine and additionally contains the PCIe buffer where the output stream is buffered before it is transmitted over PCIe, and the AXI XDMA module, a wrapper module for the PCIe connection to the remote host. The `Windowing` module is not included in these measurements because at the time of performing the evaluation it was still in development. The size associated with module `Control*` corresponds to the total size of the tracing engine. [Table 8](#) shows a small configuration with 4 states, a second configuration with 4 states but

Module	(2,2,1,0,1)		(2,2,1,0,128)		(5,20,3,1,1)	
	LUTs	FFs	LUTs	FFs	LUTs	FFs
Control*	21446	32404	38666	33980	37539	41991
Control	44	486	44	493	44	491
Tracing Engine*	1844	5112	19074	6682	17938	14699
Tracing Engine	1	2141	118	2141	1	2141
FIFO	772	2639	774	2634	773	2640
Input Reduction	256	110	583	236	256	110
NFA*	174	158	16958	1607	16266	9744
NFA	1	110	1	289	1	133
STE*	41	11	3425	265	54	35
STE	4	10	717	264	17	34
Input Decoding	37	1	2708	1	37	1
Output Reduction	642	64	641	64	642	64
PCIe Buffer	45	28	42	27	42	32
AXI XDMA (PCIe)	18748	23388	18741	23388	18745	23388
Routing Time	28:57		36:05		30:19	

Table 8: Hardware cost of the major modules of the tracing engine for three configurations of different sizes. A configuration is indicated by specifying the five parameters ($C, L, R, N, NCLS$). Modules with nested modules have two entries, once with * after the name, where the cost of all children modules is counted and once without, where only this module’s cost is counted. Nesting is indicated by indentation.

with 128 cachelines, and a bigger configuration with 300 states. The table shows that the size of some modules depends on the overlay configuration while others stay more or less constant, independently of the configuration.

First, we note that the size of modules `Control`, `FIFO`, `Output Reduction`, `PCIe Buffer`, and `AXI XDMA` doesn’t seem to be affected by the configuration. This is expected as none of these modules is aware of the configuration parameters. The hardware cost of the `AXI XDMA` module stands out from the others: Even for the very big overlays, the `AXI XDMA` module is responsible for about half the hardware cost in both LUTs and FFs. This is a huge overhead simply for transferring filtered traces to the remote host.

Second, the size of `Input Reduction` and `Input Decoding` depends on the number of CLs we track, but not on the other four parameters of the configuration. These modules need to perform additional work if multiple substreams are distinguished based on CL address as discussed in [subsubsection 3.3.1](#). These modules are unaware of the overlay size and are thus unaffected by it. Because there is only a single instance of the `Input Reduction` module, its increased hardware cost does not significantly affect the total cost. On the other hand, the `Input Decoding` module is included in every STE, thus its increased cost is very relevant. We will elaborate more on `Input Decoding` later.

A third observation is that a big fraction of the increased total hardware cost is explained by the increased number of STEs and the size of an individual STE: The total size of the NFA module `NFA*` is approximately $C \cdot L \cdot R \cdot STE^*$, i.e. the number of STEs times their size. The next observation we make, is that the size of an individual STE increases dramatically when increasing NCLS: For $NCLS = 128$, i.e. when tracking 128 separate CLs, the size of an STE is $\frac{3425}{41} \approx 85$ times higher than when $NCLS = 1$. Most of this increased cost comes from the `Input Decoding` module. As a consequence, the total size of the `NFA*` module in configuration (2, 2, 1, 0, 128) is almost 100 times higher than the total size of `NFA*` in configuration

Module	Function	FFs
STE	Pipe data registers	$4 \cdot \text{NCLS} + 2$
STE	Pipe valid registers	3
STE	Config registers	$\text{NPRED} + 3$

(a) Hardware cost as measured by flip-flop registers

Module	Function	Ops
STE	zero-out active-vectors of non-predecessors	$\text{NCLS} \cdot \text{NPREDS}$
STE	-reduce active-vectors of all predecessors	$\text{NCLS} \cdot \text{NPREDS}$
STE	vector-& predecessor active with activation	NCLS
STE	-reduce active vector of this STE	NCLS
Input Decoding	CFGLUT5	NHEADERS
Input Decoding	Demux CFGLUT5 output with cacheline index	$\text{NHEADERS} \cdot \frac{\text{NCLS}}{2} \cdot \frac{\log(\text{NCLS})}{5}$
Input Decoding	&/ -reduce trigger for each cacheline	$\text{NHEADERS} \cdot \text{NCLS}$

(b) Hardware cost as measured by basic operations (and/or/CFGLUT5)

Table 9: Simplified hardware cost of one state transition element in basic operations (Ops) and registers (FFs). Costs are listed by Function and by HDL Module in which they appear.

(2, 2, 1, 0, 1) This is just a little better than actually replicating the full NFA 128 times, which we wanted to avoid by a smart multiplexing-demultiplexing scheme described in [subsection 3.3.1](#). Similarly, in the last row of [Table 8](#) we see that increasing the number of NCLS affects the routing time of Place&Route much more than increasing the number of states from 4 to 300. We take the increased routing time as an indication that Place&Route struggles to find a good allocation of routing resources. That is, large numbers of CLs seems to create considerable routing congestion.

The central conclusion we draw from these observations is that it is crucial to design the STE and Input Decoding modules as cheaply as possible. These two modules determine most of the cost of the NFA engine and thus, for large overlay sizes, a large part of the cost of the whole tracing engine. We have carefully traded-off hardware cost against expressivity when designing STEs and Input Decoding: Unlike the previous work [9], we decided to require homogeneous transitions in order to reduce the size of an STE by limiting the number of Input Decoding modules per STE. Moreover, we used the Xilinx primitive CFGLUT5 to achieve runtime configurable input decoding at low hardware cost. We tried to find an economic design for accomodating multiple CLs but failed to do so. However, particularly for small NCLS, we think the resulting design is relatively cheap and even an overlay with 300 states consumes only a small fraction of about 3% of the total available hardware resources (LUTs and FFs) of the Virtex Ultrascale+ architecture.

Hardware cost scaling: The above observations indicate that in order to understand the scaling behaviour of the tracing engine, we need to understand how the NFA module scales with each of the parameters, C, L, R, N , and NCLS.

Every STE contains an Input Decoding module. Considering the implementation of a single STE as depicted in [Figure 17](#) and the implementation of the input decoding logic as depicted in [Figure 15](#), we can formulate a model for the hardware cost of one STE, and consequently also of the full NFA engine. Let's first consider the hardware cost of a single STE. We measure the hardware cost using the number of registers (FFs) and the number of basic logic operations (Ops) used. We consider everything that can be implemented with a LUT6 a basic operation. A LUT6 can implement an arbitrary 6-to-1 logic function or two 5-to-1 logic functions on the same inputs. In particular, we consider OR, AND and CFGLUT5 to be basic logical operations. We model the cost of a single STE as a function of the number of CLs (NCLS), the number of static predecessors of each STE (NPREDS), and the number of ECI message headers (NHEADERS) in a

batch.

The number of registers used in one STE is summarized in Table 9. The registers used by a single STE are the configuration registers (shown in Figure 17) and registers for the pipeline stages (shown in Figure 20). The update of each STE is split into three pipeline stages: Transition, Activation and Accept. The Transition stage computes the input decoding and stores for each CL whether it has received valid input and whether this input triggers this state. The Activation stage updates the activation vector for each CL and also stores which CLs have received valid input in this cycle. The Accept stage computes whether the STE is accepting and/or logging in this cycle, i.e. it stores 2 bits. Storing which CLs have received valid input is necessary because we only want to update a CL's state if it has received valid input as discussed in subsection 3.3.2. The pipeline stages thus collectively store $4 \cdot \text{NCLS} + 2$ bits. Each pipeline stage also does valid-ready flow control, which requires one *valid* register per pipeline stage. The configuration registers of an STE store one bit for each of the NPRED neighbors of the STE in the overlay to activate or deactivate transitions between STEs. Recall that only some of the neighboring STEs in the overlay actually correspond to neighboring states of the NFA. Additionally, there are 3 configuration registers that store if an STE is a starting state, an accepting state and a logging state. Each CFGLUT5 stores 32 bits internally to configure the 5-to-1 logic function it implements. However, we ignore these CFGLUT5 configuration registers because they are not general purpose registers available for user logic. They amount to a total number of FFs per STE of

$$4 \cdot \text{NCLS} + \text{NPRED} + 8 \text{ FFs.} \quad (26)$$

The major basic Ops expended in a STE and the Input Decoding are listed in Table 9. Figure 17 shows that we perform four major computation steps inside the STE module: The STE receives one NCLS wide vector from each neighbor STE. First, we zero-out all vectors that come from STE neighbors that aren't our neighbors in the NFA. This corresponds to $\text{NPRED} \cdot \text{NCLS}$ AND operations. Second, we OR all of these NPRED vectors together to find for each cacheline whether there is some neighbor that has this cacheline in an active state. This corresponds to $\text{NPRED} \cdot \text{NCLS}$ OR operations. Third, we AND the result vector of the previous step with the activation vector computed by the Input Decoding to get the active state vector of the next timestep. This corresponds to NCLS AND operations. Forth, we OR the active state vector to find out if any cacheline is currently active. This corresponds to NCLS OR operations. Thus we expend

$$(2 \cdot \text{NPRED} + 2) \cdot \text{NCLS} \quad (27)$$

basic Ops for this part of the computation. Note that some of these basic Ops can actually be combined into one LUT6 element, but we provide a conservative over-approximation.

Considering the computation performed inside the Input Decoding module Figure 15, we essentially perform three major computation steps: First, we feed each of the NHEADERS inputs through a CFGLUT5 to decide whether we should accept this opcode. This corresponds to NHEADERS basic Ops. Second, we demux the output of each CFGLUT5 with the CL index of the corresponding header. The CL index has $\log(\text{NCLS})$ digits and we demux the CFGLUT5 output to NCLS possible outputs. Using $\frac{\log(\text{NCLS})}{5}$ LUT6 elements, we can partially demux the CL index to two out of the NCLS outputs by using the fact that a LUT6 element can implement two 5-to-1 logic functions. Thus we need about

$$\frac{\text{NCLS}}{2} \cdot \frac{\log(\text{NCLS})}{5} \quad (28)$$

basic Ops to completely demux the output of one CFGLUT5. We need to do this for each of the NHEADERS CFGLUT5s of the Input Decoding. Third, we reduce the accept outputs of every header for every

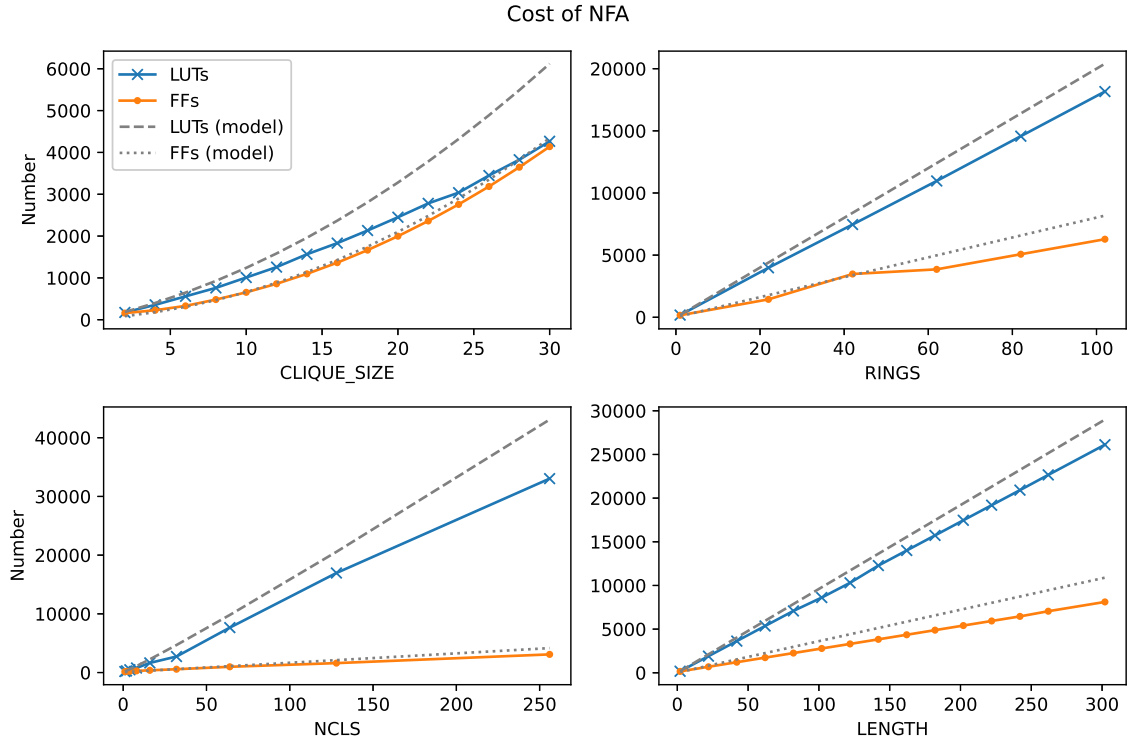


Figure 30: Hardware cost scaling of the NFA* module when increasing the different parameters. Base configuration is (2, 2, 1, 0, 1) in all cases. The cost prediction for LUTs and FFs of our model is indicated in gray.

cacheline with an AND or an OR. This requires about $NCLS \cdot NHEADERS$ many basic Ops. In total we execute about

$$(2 \cdot NPREDs + 2) \cdot NCLS + NHEADERS \cdot \left(1 + \frac{NCLS}{2} \cdot \frac{\log(NCLS)}{5} + NCLS\right) \text{ Ops.} \quad (29)$$

To predict the hardware cost of a full NFA, we need to scale the number of basic Ops and FFs both with the total number of STEs of the overlay.

$$FFs \approx STEs \cdot (4 \cdot NCLS + NPRED + 8) \quad (30)$$

$$LUTs \approx STEs \cdot \left((2 \cdot NPREDs + 2) \cdot NCLS + NHEADERS \cdot \left(1 + \frac{NCLS}{2} \cdot \frac{\log(NCLS)}{5} + NCLS\right) \right) \quad (31)$$

Figure 30 shows how the hardware cost of the full NFA scales when varying each of the parameters CLIQUE_SIZE C , LENGTH L , RINGS R , and NCLS. The predictions of our cost model are compared with the real measurements. We see that the model always over-approximates the measured data. Our model thus provides a conservative estimate of the cost of a given configuration. We attribute this over-approximation to the various optimizations that are performed by the Vivado toolchain in the synthesis process. It is interesting

to see that when varying C, L , and $NCLS$ the measured number of LUTs increases at a slower rate after a certain size. For instance, for $C \geq 23$, the increase in hardware cost is slower than for values smaller than 23. This may indicate that the toolchain can combine some of the logic operations from different places when there is enough logic to choose from.

Recall that the total number of STEs is $C \cdot L \cdot R$. Moreover, the number of predecessors of each STE is $NPREDS = (3 + 2N) \cdot C$. Thus, our model predicts both FFs and LUTs to scale quadratically in C , linearly in L, R , and N , and with $NCLS \cdot \log(NCLS)$. For C, L and R , we can observe this scaling behaviour in [Figure 30](#). However, the predicted scaling of the cost with $NCLS \cdot \log(NCLS)$ is not visible in the plots. This is because $\log(NCLS) \leq 8$ in the above plot and because it is scaled with a constant factor of $\frac{1}{5}$, thus the effect of it is only minimal.

Let's evaluate this scaling behaviour. The linear scaling in L and R is expected: Increasing these parameters by some factor leads to an increase in the number of STEs by the same factor but the individual STEs don't get bigger. The quadratic increase with C is also expected because increasing C increases both the number of STEs and the number of predecessors of each STE. It is interesting that increasing the connectivity between rings using parameter N leads only to a linear increase in hardware cost. This is because increasing N only increases the density of the overlay graph without increasing the number of STEs. The parameter N thus gives the user a convenient way of increasing the density of the overlay graph at a small cost. The scaling of $NCLS \cdot \log(NCLS)$ is both surprising and bad: Replicating the NFA $NCLS$ times in order to track $NCLS$ CLs would result only in a linear increase in cost. Our way of accomodating multiple CLs is thus asymptotically worse than this naive approach. However, there are two factors alleviating this bad result: First, this scaling is only asymptotic and considering the constant factors of $\frac{NCLS}{2} \cdot \frac{\log(NCLS)}{5}$ makes it look better. Considering these constants, the predicted scaling of our approach is better than the naive approach for $NCLS < 1024$. Second, our model over-approximates the actual hardware cost as can be seen in [Figure 30](#).

The above analysis gives us a good idea of the number of LUTs and FFs required to implement a particular instance of the tracing engine. Moreover, the cost model we developed allows the user to estimate the size of an overlay graph before implementing it on the FPGA. This can be a useful tool when judging whether some filter can reasonably be implemented alongside some other applications running on the FPGA.

The hardware cost as measured by LUTs and FFs is not the only factor that determines if a design can be implemented successfully. A second important limiter are routing resources on the FPGA. A major motivation in [9] for choosing the rings-of-cliques overlay graph is that it can accomodate more STEs at a much lower density than a clique. By density we mean the fraction of possible edges that is actually present in a graph. In general, the density of the rings-of-clique overlay graph is

$$\frac{\#Edges}{\text{Max}\#Edges} = \frac{C \cdot (3 + 2N)C}{2} / \frac{CLR}{2} = \frac{(3 + 2N)C}{LR} \quad (32)$$

Note that by increasing the parameter N , we can control the density of the overlay graph without changing the number of STEs. By measuring the time it takes to perform the Route step of the Place&Route algorithm, we can estimate the total routing congestion and the difficulty of meeting timing constraints for a design. We evaluate the effect of overlay density on the routing congestion by synthesizing a set of configurations with increasing density and measuring the runtime of the Route step in each synthesis. Concretely, we consider configurations $(1, 1, 200, N, 1)$ where $N \in \{0, 10, \dots, 90\}$. The lower plot in [Figure 31](#) shows the routing time required to implement each configuration, plotted against its density. Configurations that couldn't be implemented without violating timing constraints are marked red. The routing time seems to increase with increasing overlay density, however, there is considerable variance which makes it hard to determine the correlation. The variance in runtime is probably explained by the non-deterministic nature of the Place&Route

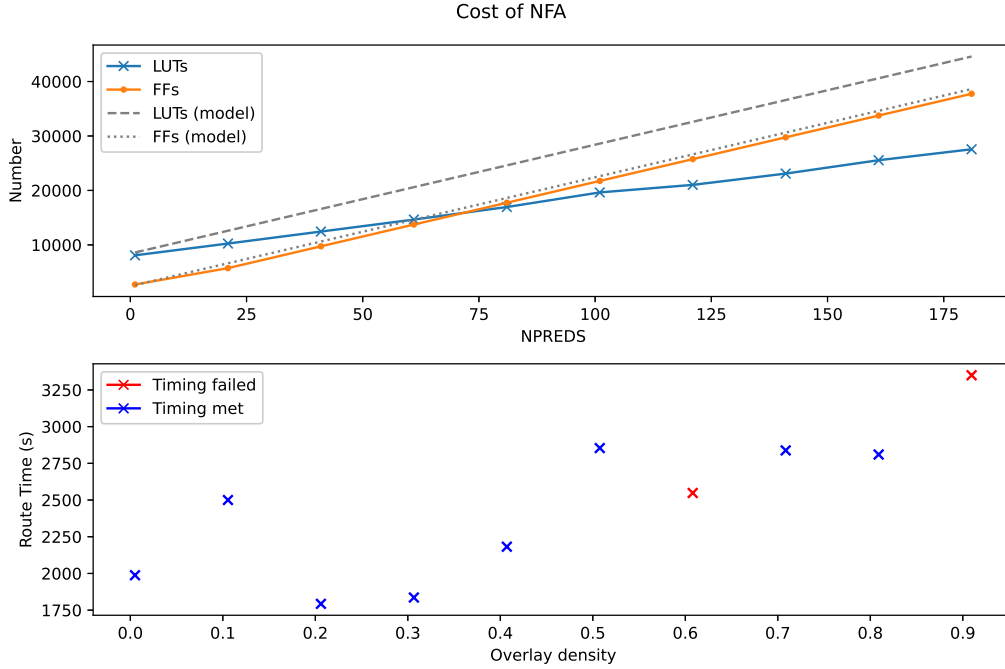


Figure 31

algorithm. This non-determinism also explains that an overlay of density 0.6 fails while an overlay with density 0.8 succeeds. A second problem with inferring that the increased density is responsible for the increased placement time is that the hardware cost increases together with the density. This is shown in the upper plot in Figure 31 where the hardware cost in LUTs and FFs is plotted against the prediction from our model.

In conclusion, we believe that the density of the overlay graph is a limiter for what overlays can be implemented without timing violations but the experimental data is not strong enough to quantify the precise impact.

Practical configurations: Above we have evaluated the impact of the overlay parameters on the scalability of the overlay graph in isolation. To give the user a more concrete idea of practical configurations that can be implemented successfully alongside the full ECI stack, we next discuss several example overlay graphs that we implemented. Figure 32 shows some maximal configurations that finished without timing issues. We observe that we can implement various interesting configurations with many states and high connectivity. For instance the violet configuration accommodates 160 STEs, each with fan-out and fan-in of 24 (recall that every STE is connected both to all STEs in the same clique as well as to all STEs in the previous and the next clique on the same ring). The yellow configuration accommodates 150 STEs on three rings, each STE having fan-out/in of 50. Additionally, this configuration allows to track 4 independent CLs. The red configuration is a 100-clique, i.e. accommodates 100 STEs that are all pairwise connected. Any homogeneous NFA with at most 100 states can be mapped to this overlay configuration. The largest number of states is 604 and is achieved by the green configuration. However, this configuration provides only very little connectivity

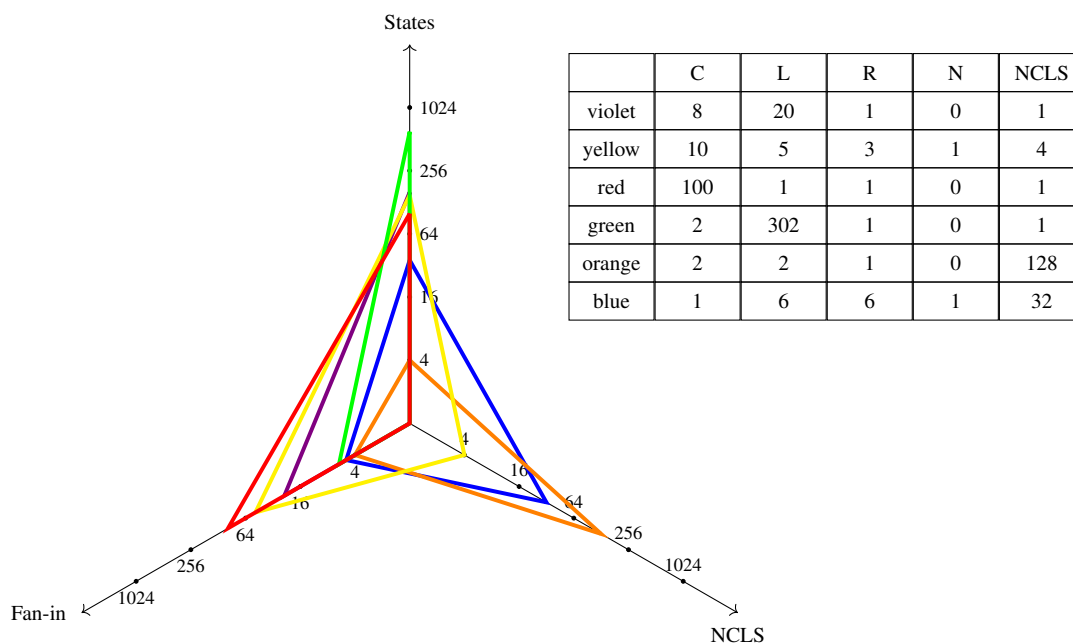


Figure 32: Spider diagram showing some interesting, maximal configurations that can be implemented without timing violations.

as each STE has fan-in only 6. The largest number of independent CLs we can track is 128 in the orange configuration, but this configuration only has 4 states. A compromise is provided by the blue configuration, which allows tracking 32 CLs with a total of 36 states. Each state has a low fan-in of 5.

In conclusion, we find that we can implement a wide variety of useful configurations that allow the user to target the tracing engine for the kind of NFAs they need.

This work improves over the scaling behaviour reported in the previous work [9]. The author of [9] reports timing issues for configuration (3, 7, 2, 1). Note that configuration (5, 20, 4, 1) reported in Figure 32 is a proper supergraph of configuration (3, 7, 2, 1) and has almost 10 times more states. We attribute this improved scaling behaviour mostly to using homogeneous transitions: Homogeneous transitions considerably reduce the cost of high-density overlays. As discussed above, increasing the density of an overlay leads to a linear increase in hardware cost. This is the same in [9], but the linear factor is much larger because adding one predecessor to an STE requires adding one input decoding logic. [9] reports a usage of 9 CLBs per Input Decoding. This corresponds to 72 LUTs and 144 FFs on the Ultrascale+ architecture. In our design, one Input Decoding module uses around 40 LUTs and 1 FF as can be seen in Table 8. Thus the design in [9] induces a considerable hardware cost for increased density. Moreover, all each instance of the Input Decoding requires accessing the NFA input in each cycle. This additional congestion most likely further limits scaling of the design in [9].

5.2 Expressivity of the filter language

In subsection 5.1 we have evaluated how well the tracing engine scales when we vary the different parameters of the overlay graph. The overlay graph limits the size and density of the NFAs we can represent. However,

the second important factor that determines expressiveness of the mappable filters is the choice of base predicates of the NFA. The concrete set of base predicates is determined in collaboration of the `Input Reduction` and `Input Decoding` modules and thus compile-time configurable. Next we will present a way of thinking about the resulting filter language that allows judging its expressiveness and suitability for a given use case.

As discussed in [subsection 1.5](#), there is a one-to-one correspondence between NFAs and RegExs. A convenient way to think about our filter language is thus as RegExs of the base predicates provided by the `Input Reduction` and `Input Decoding` modules. Recall from [Figure 6](#) the construction of an NFA accepting some RegEx. From this construction we can easily derive a recursive formula for the number of states and the number of edges required to build an NFA for a given RegEx.

$$\text{state}(r) = \begin{cases} 2 & , \text{ if } r \text{ is a basic RegEx} \\ \text{state}(X) + \text{state}(Y) & , \text{ if } r = XY \\ \text{state}(X) + \text{state}(Y) + 2 & , \text{ if } r = X|Y \\ \text{state}(X) + 1 & , \text{ if } r = X^* \end{cases} \quad (33)$$

$$\text{edges}(r) = \begin{cases} 1 & , \text{ if } r \text{ is a basic RegEx} \\ \text{edges}(X) + \text{edges}(Y) + 1 & , \text{ if } r = XY \\ \text{edges}(X) + \text{edges}(Y) + 4 & , \text{ if } r = X|Y \\ \text{edges}(X) + 2 & , \text{ if } r = X^* \end{cases} \quad (34)$$

Note that the *in* and *out* edges for each constructor in [Figure 6](#) are never really instantiated, they are just placeholders for the “real” edges of each constructor. Thus we don’t need to count them. The construction of [Figure 6](#), and thus also [Equation 33](#) and [Equation 34](#), uses more states and edges than necessary in some cases. For instance, the RegEx “ $X_1|X_2|\dots|X_n$ ” can be implemented with only 2 auxiliary states instead of $2n$ as suggested by

$$\text{state}(X_1|X_2|\dots|X_n) = \text{state}(X_1) + \text{state}(X_2) + \dots + \text{state}(X_n) + 2n$$

To get a more compact construction, nodes with only one outgoing epsilon transition can be collapsed with their sole successor node. However, note that the construction in [Figure 6](#) naturally yields a homogeneous NFA. When collapsing epsilon transitions, this may no longer be the case and we might need to duplicate some states in order to get back a homogeneous NFA. In general, [Equation 33](#) and [Equation 34](#) provide us with an upper bound on the required size of the NFA.

The base RegExs supported in the tracing engine match *message type* (i.e. opcode, direction and VC) of an ECI header. Further, the occurrence of **Any** or **None** of a set of message types can be matched. This is discussed in [subsection 3.3.1](#). A methodology to upper-bound the necessary size of the overlay graph given some pattern, is thus the following:

1. Express the pattern as a RegEx where the base RegExs match ECI message types in a message batch against one, **Any** or **None** of a set.
2. Compute the number of states and edges necessary to match the RegEx using [Equation 33](#) and [Equation 34](#).
3. Conclude what overlay parameters C, L, R, N are sufficient to accommodate the desired pattern.

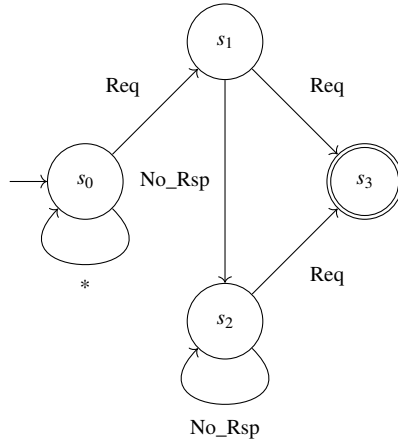


Figure 33: NFA accepting in-flight requests.

We will now consider an example of this estimation. To match in-flight requests, i.e. ECI request messages that arrive at the DirC while it is still processing some other request, we can formulate the following RegEx.

$$\text{Req No_Rsp}^* \text{Req} \quad (35)$$

Where the base predicates are given by

$$\text{Req} = \mathbf{Any}(\text{cpu.MREQ_RLDD}, \text{cpu.MREQ_RLDI}, \text{cpu.MREQ_RLDX}) \quad (36)$$

$$\text{No_Rsp} = \mathbf{None}(\text{fpga.MRSP_PEMD}, \text{fpga.MRSP_PSHA}) \quad (37)$$

Using [Equation 33](#) and [Equation 34](#) we thus get an upper bound for

$$\text{state}(\text{Req No_Rsp}^* \text{Req}) = \text{state}(\text{Req}) + \text{state}(\text{No_Rsp}^*) + \text{state}(\text{Req}) \quad (38)$$

$$= 2 + \text{state}(\text{No_Rsp}) + 1 + 2 \quad (39)$$

$$= 2 + 2 + 1 + 2 = 7 \quad (40)$$

Similarly, we can compute

$$\text{edges}(\text{Req No_Rsp}^* \text{Req}) = \text{edges}(\text{Req}) + \text{edges}(\text{No_Rsp}^* \text{Req}) + 1 \quad (41)$$

$$= 1 + \text{edges}(\text{No_Rsp}^*) + \text{edges}(\text{Req}) + 1 + 1 \quad (42)$$

$$= 1 + \text{edges}(\text{No_Rsp}) + 2 + 1 + 1 \quad (43)$$

$$= 1 + 1 + 2 + 1 + 1 = 6 \quad (44)$$

The actual implementation that runs on the tracing engine is depicted in [Figure 33](#) and requires only 4 states and 4 edges. Note that self loops and the starting edge are not implemented as overlay edges and we thus don't need to count them.

Powerful base predicates thus allow us to implement complicated triggers using fewer states. We find that our base predicates are suitable and powerful for the use cases we have in mind, because in all use cases we are interested in occurrences of certain message types. We find that the basic predicates supported by the

Input Decoding module allow us to express various coherence events very succinctly. This can be seen, for instance, in the experiment in [subsection 4.3](#) where a single **Input Decoding** module can recognize a predicate consisting of various coherence events:

$$A11 \vee A31 \vee A32 \vee R12 \vee R13 \vee RA2 \quad (45)$$

The core feature of the predicates that allows us such succinct representations are the **Any** and **None** combinators. Note that if we were lacking the **Any** combinator, the above predicate would require an alteration of 6 alternatives, i.e. a total of 8 NFA states instead of 1. Also note the power of the **None** combinator: Without the **None** combinator, a simple predicate like `No_Rsp` from [Equation 37](#) needs to be implemented as an alteration of all non-illegal alternatives, i.e. all coherence messages that are not memory responses. Considering only the crucial ECI coherence events listed in [Table 2](#), there are 16 different messages. Each such message can be sent in both directions, thus implementing `No_Rsp` would require an alteration of 30 options, i.e. 32 NFA states. However, with the **None** combinator present, a single NFA state suffices to implement predicate `No_Rsp`.

Given that we are interested in message type, direction and VC of an ECI message, the **Input Decoding** module thus allows us to express very interesting predicates quite compactly. However, depending on the use case, other information might be necessary. In those cases the predicates provided will not be of much help. In this work we have focused on the coherency VCs and ignored the remaining four VCs for I/O, multiplexed co-processor data and multicore debugging data. The protocol of the I/O VCs consists exclusively of request-response pairs and notifications. The crucial information in these messages seem to be opcode and address, similar to messages on the coherency VCs. Thus we believe that the chosen filter language is also well-suited for formulating interesting filters on the two I/O VCs. The multicore debugging VC only knows two message types, thus a simple message filter will probably be sufficient for inspecting this VC. Because almost nothing is known about the co-processor VC, we can't make a statement about the suitability of our pattern language for it.

5.3 Mapping algorithm

The algorithm discussed in [subsection 3.4](#) formulates the problem of mapping NFA states to overlay nodes as an integer program. Two reasons motivated this choice:

- In the beginning we were experimenting with a different design as discussed in [subsection 3.2](#) that distributes different parts of the input to different parts of the overlay graph. This is similar to the bit-split automaton discussed in [\[15\]](#). In this scenario the mapping problem becomes more complicated because it also involves deciding which parts of the input to route to which parts of the overlay, and all NFA states need to be mapped s.t. they get the correct part of the input. Integer programming is very versatile and allows expressing these complicated constraints.
- The subgraph mapping problem is an NP-hard problem. Kuchler [\[9\]](#) and Karakchi et al. [\[18\]](#) both use a heuristic mapping algorithm that is very fast but that does not find an optimal solution in all cases. Kuchler duplicates certain states of the NFA and allows epsilon transitions between arbitrary nodes of the overlay graph to remedy cases where the heuristic fails. Karakchi et al. try their heuristic with bigger and bigger overlay graphs until they find one for which the heuristic works. Neither approach is suitable for this work: we implement epsilon transitions as ordinary transitions between neighboring STEs, thus the approach from Kuchler doesn't work. The re-synthesis required in the approach of Karakchi defeats the purpose of runtime reconfigurability. We want our mapping algo to fail only if the mapping problem is impossible.

The problem of the integer programming formulation is that it solves an NP-hard problem. It is thus to be expected to get long running times trying to map an NFA to the overlay graph. Indeed, we find that the obvious formulation does only work for small sizes of the overlay graph. To facilitate the subsequent discussion we provide next the baseline formulation for the subgraph mapping problem. We want to map the graph $G_A = (V_A, E_A)$ associated with some NFA A onto some instance of the rings-of-cliques overlay graph $G_O = (V_O, E_O)$.

$$\max \sum_{\substack{v \in V_A \\ u \in V_O}} x_{v,u} \quad (46)$$

$$\text{s.t.} \sum_{u \in V_O} x_{v,u} \leq 1 \quad \forall v \in V_A \quad (47)$$

$$\sum_{v \in V_A} x_{v,u} \leq 1 \quad \forall u \in V_O \quad (48)$$

$$x_{v_1,u_1} + x_{v_2,u_2} \leq 1 \quad \forall (v_1, v_2) \in E_A. \forall (u_1, u_2) \notin E_O \quad (49)$$

where the binary assignment variable $x_{u,v}$ is 1 if and only if the NFA state u is mapped to the overlay node v . The objective Equation 46 maximizes the number of NFA states that are mapped to the overlay. Constraint Equation 47 ensures that each NFA state is mapped to at most one overlay node. Similarly, constraint Equation 48 ensures that each overlay node is used at most once. Constraint Equation 49 enforces that neighboring states v_1, v_2 in the NFA are mapped to neighboring overlay nodes.

Two slight variations of this baseline formulation lead to much improved runtimes.

- Symmetry breaking in the IP formulation: Since the overlay graph is highly symmetric, there exist large sets of equivalent solutions. For instance, swapping the indices of all vertices of two cliques in the overlay graph will give an equivalent solution. IP solvers often have problems with highly symmetric problems. We can break many of these symmetries by introducing some additional constraints. For example, fixing the mapping of one NFA state to some arbitrary overlay node will remove many possible solutions while guaranteeing that if there was any feasible mapping to begin with, there will still be one after adding the constraint. Note that this symmetry breaking only works because all STEs are exactly equal up to their label.
- Reducing the size of the problem by exploiting the special clique structure of the overlay graph: Note that any two vertices v, u of the same clique C of the overlay graph have the exact same set of neighbors. Thus, mapping some NFA state to v is equivalent to mapping it to u . We can thus directly map NFA states to cliques of the overlay and ignore the individual vertices of the overlay. This is done in the formulation Equation 14. Note that this reduces both the number of assignment variables and the number of constraints considerably.

To compare the effect of these two simplifications, we perform the following experiment. We fix some overlay configuration $(10, \frac{n}{10}, 1, 0)$ parametrized by the number of STEs n in the overlay graph. The configuration consists of $\frac{n}{10}$ cliques of 10 nodes each that are arranged on one ring. We choose this particular overlay because it corresponds structurally to what we used for our experiments in section 4 and because we think it is a typical example. For each configuration, we choose a random subgraph as NFA graph as follows: We first create the graph corresponding to the overlay configuration $(7, \frac{n}{10}, 1, 0)$ and then we remove each edge with probability 0.3. For each size parameter n , we compute the mapping from NFA to overlay graph using each of the four IP formulations *Baseline*, as given in Equation 46, *Symmetry*, which adds the

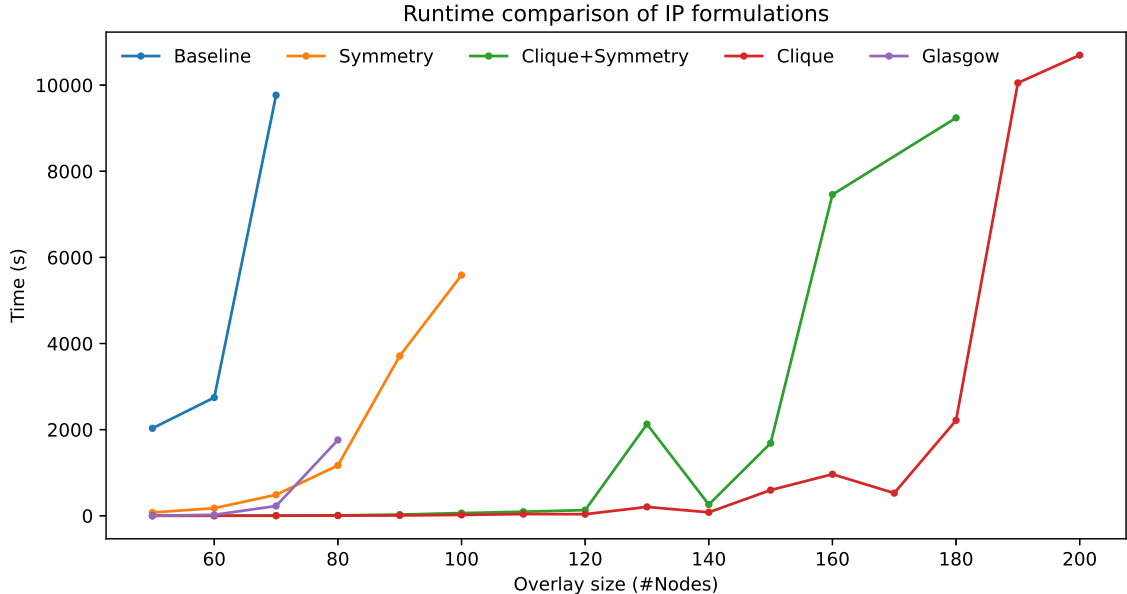


Figure 34: Runtime of four IP formulations and of the Glasgow subgraph solver *Glasgow* for the subgraph mapping problem.

symmetry breaking constraint, *Clique*, which maps NFA states directly to overlay cliques and ignores individual overlay nodes, and *Clique+Symmetry*, which combines the two simplifications in one formulation by fixing the mapping of one NFA node to some overlay clique. Additionally, we compute the mapping using the dedicated Glasgow subgraph solver tool [21] to see how our IP approach compares to a state-of-the-art solver.

Figure 34 plots the runtime of each of the four IP formulations and of the Glasgow subgraph solver on the experiment instances for $n = 50, 60, \dots, 200$. The experiment was performed using the Gurobi optimizer version 9.0.0 on the ETH Euler cluster. All experiments were run on a single core with a maximum of four of threads and a time limit of 4 hours. We see that both simplifications considerably speed up the computation. While the *Baseline* formulation only solves instances with overlay size up to 70, *Symmetry* solves instances with up to 100 nodes in the overlay and *Clique* even solves instances with up to 200 nodes in the overlay. Interestingly, the combined strategy *Clique+Symmetry* performs worse than *Clique* alone. It is not clear to us why this is the case. Further, we note that the Glasgow subgraph solver is considerably faster than the naive *Baseline* formulation. However, it performs clearly worse than all our improved IP formulations. We attribute this to the additional knowledge about the mapping problem that is encoded in the modified IP formulations. The Glasgow subgraph solver has no information about the symmetries in the overlay graph and can thus profit from neither symmetry breaking nor can it reduce the problem size as we do in the *Clique* formulation.

Despite the huge improvement achieved by the *Clique* formulation, all formulations are limited in what they achieve: for sizes $n > 120$ even the fastest formulation, *Clique*, requires more than one minute to solve the problem. This still seems to be a viable solution because we don't expect to run with configurations with much more than 100 nodes in the overlay graph for two reasons: (1) We want to run the tracing engine

alongside major applications of the FPGA, thus the number of states of the overlay graph is limited a priori. (2) The current implementation of the tracing engine itself only scales to a few hundred nodes, except in configurations where every clique is tiny and we only track one CL.

5.4 Design decisions

We will now discuss three major design decisions of the tracing engine. First, we discuss the restriction to homogeneous NFAs and the effect it has on representable NFA size per hardware cost. Second, we discuss our choice of the rings-of-cliques overlay graph by comparing it to overlay graphs used in other works. Finally, we discuss advantages and shortcomings of the way we configure the tracing engine by sequentially shifting in a config string.

5.4.1 Homogeneous NFAs

Homogeneous NFAs trade off the number of required STEs with the complexity of input decoding for one STE. With all transitions into some state triggering at the same inputs, the input decoding for this state can be made significantly simpler. On the other hand, a state which does not have homogeneous transitions needs to be replicated across multiple STEs and its transitions need to be partitioned over the copies such that each copy has homogeneous transitions.

The concrete reasons to prefer homogeneous transitions over general transitions in this work are the following: First, a significant difficulty of the tracing engine design is that it needs to propagate the input data to every input decoding logic every cycle. If we reduce the number of input decoding instances, then there are fewer places that access the input data simultaneously. This alleviates the contention for the input data and thus helps scaling up the overlay to more nodes. Second, in correspondence with the various different ECI layer messages, we need quite a flexible input decoding logic in order to express meaningful predicates as detailed in [subsection 3.3.1](#). Minimizing the number of times this logic is replicated is thus important to reduce the hardware cost.

Both Micron's AP [17] and Napoly [18] require homogeneous transitions. In both cases, this allows to keep the input decoding small and simple: Inputs are 8-bit ASCII characters and the input decoding is conceptually an array with 2^8 entries where entry at position i indicates if the i -th ASCII character can activate this state. On the other hand, the previous work on filtering block layer traffic [9], didn't use homogeneous transitions. This requires replicating the input decoding logic for every connected STE. In general, this overprovisions more input decoding logic than necessary: every STE allocates input decoding logic for every neighboring STE. When mapping a particular NFA, many of these neighboring STEs may not host any NFA states at all, or they might host NFA states that are not connected to the NFA state mapped to this STE, or there might be neighboring NFA states that transition into the current NFA with the same trigger. All these cases make some of the allocated input decoding logic unnecessary. In general, a NFA state with a total of x homogeneous transitions will require x input decoding instances. Overprovisioning this by some constant amount will likely waste hardware resources while replicating states across multiple STEs only uses the input decoding really needed.

On the other hand, the transformation of a non-homogeneous graph to homogeneous form can cause significant node and edge replication. [Figure 35a](#) and [Figure 35a](#) show two very similar NFA graphs, their only difference is the direction of the transitions. [Figure 35c](#) shows the effect of transforming the graph in [Figure 35a](#) to homogeneous form. Every node needs to be replicated three times, once for each input a, b, c it receives. In consequence, also every edge needs to be replicated three times. Thus both, the number of nodes and the number of edges increases by linear factor in the number of inputs. The graph in [Figure 35b](#),

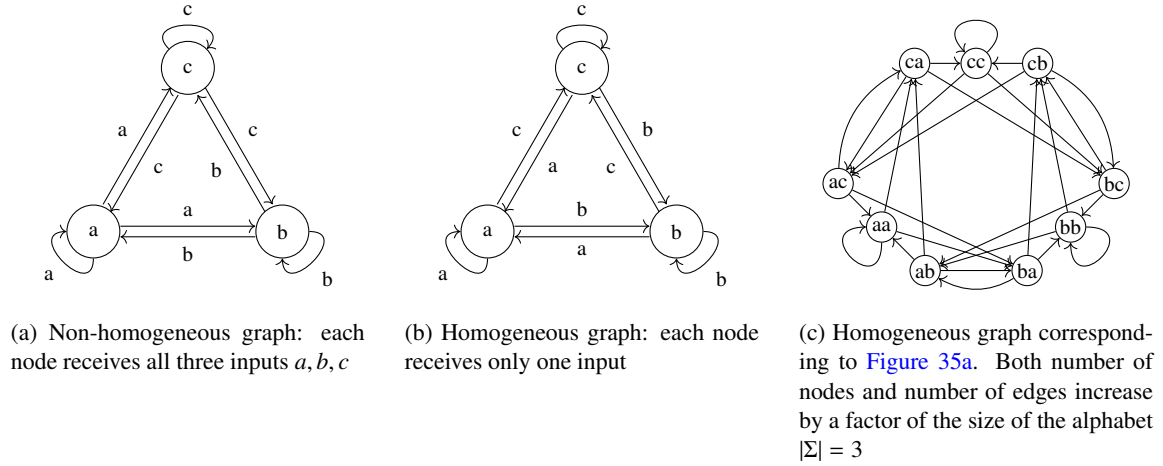


Figure 35: The effect of transforming an NFA to homogeneous form.

NFA	General		Homogeneous	
	States	Edges	States	Edges
Block request-response Figure 25b	2	2	2	2
Cache miss Figure 27	7	8	10	21
MOESI correctness Table 6	5	19	15	53

Table 10: Comparison of the number of states and edges in the general NFA description we give to the frontend and the corresponding number states and edges in the transformed NFA with homogeneous transitions. The edge count ignores start edge and self loops. It counts epsilon transitions in the general NFA as one edge. All epsilon edges are resolved to normal edge in homogeneous NFA.

however, already is homogeneous. Despite looking very similar, the two graphs are vastly different when in homogeneous form. This example generalizes to any number of nodes if given an input alphabet of the same size.

In the worst case scenario of Figure 35c, the negative effect of state duplication most likely outweighs the savings made by reducing the number of Input Decoding units. Of course, this depends on the relative hardware cost of the input transition unit and the rest of an STE.

In Table 10 we show the overhead incurred by transforming a general NFA into homogeneous form for each of the NFAs from experiments in section 4. We find that the MOESI correctness NFA developed in subsection 4.3 requires a state and edge replication of a factor of about 3. This is considerably higher than for the other use cases. Even so, note that the hardware cost of the overlay graph necessary to represent the MOESI correctness NFA is likely lower in the homogeneous case than in the general case: Depending on the concrete configuration, the Input Decoding module makes up the largest part of the hardware cost of an STE (c.f. Table 8). The largest fan-in of any state in the MOESI correctness NFA is 4, thus an overlay implementing the non-homogeneous form of this NFA will require at least 4 Input Decoding modules per STE. The additional hardware cost induced by the threefold state and edge replication is thus most likely offset by the savings of having only one Input Decoding module per STE in the homogeneous case. In all other filters we implemented so far, we find that not much state replication is necessary at all. This is in line

with the findings by Micron [17]. The authors report that the number of additional nodes stays below 5% for the NFAs they considered.

We thus conclude that implementing homogeneous NFAs in hardware is cheaper in most practical cases than implementing general NFAs. This is especially true if the total hardware cost of an STE is mostly determined by the cost of the input decoding, as it is in our design. In these cases, the necessary state replication will be offset by the savings in the cost of an individual STE.

5.4.2 Choice of overlay graph

The overlay graph structure used in this work was directly re-used from Kuchler [9]. In this section we want to reflect on this choice by considering the general problem of choosing an overlay graph and the fundamental limitations of fixing an overlay.

There are essentially two ways of choosing an overlay graph structure: First, if the exact filter language is fixed a priori, we can create an overlay graph that accomodates all NFAs corresponding to valid filters up to a certain size. This is done by Teubner et al. [13] where the query language XPath results in very linear NFAs as discussed in section 2. Second, if the filters are general NFAs, the only way of ensuring that every filter of n NFA states will fit is making the overlay an n -clique. Because allocating so much routing resources is not economical for large numbers of STEs, works like the Micron AP [17] and Napoly [18] make a compromise between expressivity and routing. In either case, filters of a certain size won't fit the overlay. The essential difference between the case where the query language limits the shape of the legal NFAs and the case where there is no such limitation is the *worst case overhead* of the design. By worst case overhead we mean the difference between the size of the smallest NFA that doesn't fit onto the overlay and the overlay size. "Size" can be measured in several ways: nodes, edges, and maximum degree may all make sense. In the case of this work, where we allow general RegExs, we can always create an NFA where one node has an arbitrarily high fan-out: Let $\Sigma = \{\sigma_1, \dots, \sigma_n\}$ be our input alphabet, then an alteration with n choices

$$\sigma_1\sigma_1 \mid \sigma_2\sigma_2 \mid \dots \mid \sigma_n\sigma_n \quad (50)$$

leads to an NFA where one state has fan-out $n + 1$ and one state has fan-in $n + 1$ (c.f. the construction of an NFA corresponding to a RegEx in subsection 1.5)². An overlay configuration $(\frac{n}{3}, L, 1, 0)$ has in this case a factor of $\frac{L}{3}$ more STEs than necessary to accomodate the RegEx, but because any STE has only $n - 1$ neighbors, the RegEx still can't be mapped. In this case, the worst case overhead is $\frac{L}{3}$ and can become arbitrarily large if we increase the length L of each ring.

István et al. suggest a mechanism to deal with this fundamental limitation [24]: A long RegEx r that does not fit onto the overlay structure is split into multiple shorter RegExs r_1, \dots, r_n . Initially, the overlay is configured to match r_1 . If r_1 matches, it is automatically reconfigured to match the next partial RegEx r_2 . A match of the final partial RegEx r_n is reported as a match of the long RegEx r . This approach can effectively lift the inherent limitation of a fixed overlay graph but it has several downsides: First, RegExs can't necessarily be made sufficiently small by splitting them up. Consider for instance the big alteration from Equation 50: Any way of splitting it would need to maintain the initial node with $n + 1$ outgoing transitions. If this fan-out is too large for our overlay, we won't be able to accomodate it even when splitting the RegEx. Second, this mechanism requires fast reconfiguration of the overlay between matching the individual sub-RegExs r_1 to r_n . Due to the sequential way in which we load configurations, our overlay takes many cycles to be reconfigured. Since we need to process a continous stream of VC layer traffic, such a mechanism would require buffering all traffic that arrives whilst reconfiguring. In [24], the computation is

²This assumes that we are representing epsilon transtions as normal transitions as described in subsection 3.4

v	0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	0	0	0	0	0	0	0
2	1	1	1	1	1	1	0	0	0	0	0	0
3	1	1	1	1	1	1	1	0	0	0	0	0
4	0	1	1	1	1	1	1	1	0	0	0	0
5	0	0	1	1	1	1	1	1	1	0	0	0
6	0	0	0	1	1	1	1	1	1	1	0	0
7	0	0	0	0	1	1	1	1	1	1	1	0
8	0	0	0	0	0	1	1	1	1	1	1	1
9	0	0	0	0	0	0	1	1	1	1	1	1
10	0	0	0	0	0	0	0	1	1	1	1	1
11	0	0	0	0	0	0	0	0	1	1	1	1

(a) Adjacency matrix for the Napoly overlay graph for $n = 12$ and $f = 7$

v	0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	1	1	0	0	0	0	0	1	1	1
1	1	1	1	1	1	0	0	0	0	0	1	1
2	1	1	1	1	1	1	0	0	0	0	0	1
3	1	1	1	1	1	1	1	0	0	0	0	0
4	0	1	1	1	1	1	1	1	0	0	0	0
5	0	0	1	1	1	1	1	1	1	0	0	0
6	0	0	0	1	1	1	1	1	1	1	0	0
7	0	0	0	0	1	1	1	1	1	1	1	0
8	0	0	0	0	0	1	1	1	1	1	1	1
9	1	0	0	0	0	0	1	1	1	1	1	1
10	1	1	0	0	0	0	0	1	1	1	1	1
11	1	1	1	0	0	0	0	0	1	1	1	1

(b) Adjacency matrix for the corresponding rings-of-cliques configuration (1, 1, 12, 3)

Figure 36: Comparison of Napoly and rings-of-cliques overlays

an offline computation which can more easily be halted. Moreover, the authors report that their overlay graph can be configured in two cycles. However, this requires storing multiple configurations already on the FPGA. The approach is thus still fundamentally limited by how many configurations can be stored simultaneously.

The benefit of the rings-of-cliques overlay graph chosen in this work is that it allows the user to parametrize at compile time the exact graph layout they want. As seen in [subsubsection 3.3.2](#), a wide range of graphs is representable by the rings-of-cliques overlay. Intuitively, to increase the longest path we can represent, we can increase the length parameter L . In order to increase the highest fan-out of a NFA state, we can increase the clique size parameter C . The R and N parameters provide us with more fine-grained control of the graph density because N controls how many of the rings are interconnected.

The resulting overlay always exhibits a high degree of symmetry. This is advantageous for several reasons: First, it makes the HDL description of the routing between states easier. Second, it also makes the subgraph mapping problem easier as we have discussed in [subsection 5.3](#): Thanks to symmetry, we can ignore individual STEs of a clique and directly map NFA states to cliques.

Interestingly, the rings-of-cliques overlay graph can be configured to be almost identical to the overlay graph used in Napoly [18]. Napoly uses generalization of a path parametrized by fan-out f of each state: $f = 2$ is a path with self-loops, $f = 3$ additionally sends output to the predecessor state. In general, the i -th node in the sequence sends its output to nodes $i - \lfloor \frac{f-1}{2} \rfloor, \dots, i + \lfloor \frac{f}{2} \rfloor$. Considering a particular Napoly graph with n nodes and a fixed f and the rings-of-cliques configuration $(1, 1, n, \lfloor \frac{f}{2} \rfloor)$, we see that they are very similar. Figures [Figure 36a](#) and [Figure 36b](#) show the two incidence matrices for Napoly with $n = 12$ and $f = 7$ and for rings-of-cliques with configuration $(1, 1, 12, 3)$, respectively.

There is one major difference between a Napoly overlay graph with n nodes and fan-out f and the corresponding rings-of-cliques configuration: The rings-of-cliques is a closed ring while the Napoly overlay isn't. In other words, the diameter of the rings-of-cliques is half the diameter of the Napoly overlay graph.

The Napoly overlay is chosen to best exploit the hardware resources of an fpga. It is designed to scale to tens of thousands of states [18] and the idea is that a full FPGA could be packed with overlay nodes. The connectivity of the overlay graph is thus chosen to align well with the wiring resources of an FPGA. Using manual floorplanning, the STEs are laid out on the FPGA horizontally in sequence and across rows in a zig-zag pattern, thus mostly using the horizontal wiring resources of the FPGA. In particular, connecting the

two ends of the Napoly overlay graph is impossible because these ends are in very distant locations on the FPGA.

Packing a full FPGA is not the goal of this work. On the contrary, the goal is to use as little FPGA resources as possible such that the tracing engine can be run alongside other applications. Closing the ring in the rings-of-cliques overlay constrains the Place&Route in placing the individual STEs and necessitates a more compact layout of the overlay on the FPGA. However, the same constraining effect is already a consequence of the way we broadcast input data to STEs: Recall that we send the reduced input data to each STE from the `Input_Reduction` module. This requires that all STEs are close to this central location. For us, the benefit of symmetry thus outweighs the potential negative effect it has on Place&Route.

We conclude that the rings-of-cliques overlay graph provides a large degree of flexibility to the user. Being able to trade-off size and density of the overlay graph with its hardware cost at compile time allows tailoring the tracing engine to various use cases.

5.4.3 Runtime reconfiguration with shift registers

Recall from [subsection 3.3.2](#) that we configure the NFA implemented by the overlay graph at runtime by shifting a configuration string into a set of registers. These registers are conceptually connected in a long chain and loading a configuration consists of shifting it bit-by-bit through this register chain. There are several downsides to this way of loading a configuration:

First, tracing cannot proceed while we are reconfiguring the tracing engine: As long as the new configuration isn't shifted in completely, parts of the old configuration will still be in some of the registers. Continuing processing in such a state will thus lead to garbage output.

Second, loading a full configuration can take many cycles. The total number of configuration bits needed is

$$\text{STEs} \cdot (\text{NPRED} + 3 + \text{NHEADERS} \cdot 32) \quad (51)$$

as discussed in [subsection 5.1](#). While the $\text{NPRED} + 3$ configuration bits used for configuring the input transitions, start, accepting and logging state of an STE are relatively cheap, the number of configuration bits used for the `Input_Decoding` is very high. This is certainly a downside of the chosen `Input_Decoding`. For 14 VCs we are processing 28 headers, so for a large configuration with around 100 states, a full configuration can consist of several hundreds of thousands of bits. As we are loading one bit per cycle, loading a full configuration can take that many cycles.

Third, the PCIe module is responsible for a big portion of the whole hardware cost of the tracing engine as discussed in [subsection 5.1](#). One reason why the PCIe module is needed is to configure the tracing engine from a remote host. Currently it also streams out the filtered ECI messages to the remote host, however, these messages could also be stored to FPGA-local DRAM instead. Thus a different way of configuring the tracing engine could significantly reduce its hardware cost.

The fact that streaming needs to halt for many cycles between two configurations may hinder certain uses of the tracing engine. Let's consider again the mechanism suggested by István et al. and discussed in [subsection 5.4.2](#), where a complicated filter is matched in multiple passes. This approach cannot possibly work if tracing needs to halt for 100000 cycles between two consecutive partial RegExs. One way of achieving immediate switching between configurations would be to duplicate each configuration register and allow to select either copy of a register for both updating and for controlling the NFA. A new configuration could be streamed into copy *B* while copy *A* still holds the original configuration and the NFA is still controlled by *A*. Then, as soon as *B* is completely loaded, we could switch atomically from *A* to *B*. However, this would not necessarily suffice to implement the mechanism from above because the latency of configuring is still the same.

An advantage of the current way of configuring the tracing engine is that it limits the congestion for the configuration signal. A different mechanism could write to multiple configuration registers simultaneously in order to speed up the process, but this would automatically increase the congestion in the design. For instance, writing all configuration registers at once from a central location would most likely fail.

An interesting mechanism for configuration that would not require PCIe is to embed the configuration in the data we are tracing. The XML filter developed by Teubner et al. [13] is configured by processing a special XML string. A special XML tag is recognized by each STE and the configuration is extracted from the content nested inside the tag. However, the XML filter setup differs from ours considerably: While the XML stream can be entirely controlled by the users, we only have limited control over the ECI stack. In particular, we can only control the payload of VC layer messages but not their headers. A similar mechanism might still work, where STEs receive message payloads and have special logic for searching the content for a configuration. A different mechanism would use ECI in a similar way to how we use PCIe now: Configuration data is transferred to the FPGA and from there it is fed into the tracing engine. Both approaches would allow us to get rid of PCIe. A significant disadvantage of both these approaches is that they depend on the correct functioning of ECI. As one of the motivations of this work is understanding and fixing implementation problems of ECI, it is better to be independent of ECI.

6 Conclusions

6.1 Summary

In this thesis we have developed a tracing engine that provides a convenient way of inspecting the communication on the VC and block layer of the high-speed ECI stack of the Enzian computer. To reduce the size of the produced traces, the message stream can be filtered directly on the FPGA before writing out the data to a remote host. We have developed a pattern language that allows specifying complicated filters as NFAs over basic predicates on the ECI messages. We have described the HDL design in detail and performed a varied set of experiments that demonstrate the functionality and versatility of the design. We thoroughly evaluated various aspects of the design such as hardware cost and expressiveness, both experimentally and theoretically.

In the following, we will quickly consider each of the design goals we set in [subsection 1.1](#) and discuss whether it was achieved or not.

Line rate The throughput of the tracing engine is only inherently limited to processing at most 1 message batch per cycle. This is an inherent limitation in the current interface between block and VC layer consisting of one FIFO per VC and direction. As discussed in [subsection 3.3.5](#), the whole tracing engine is pipelined and can process a full message batch per cycle. Tracking VCs 2 to 11 in both directions and inserting IDLE headers when no ECI word is ready, the design evaluated in this work actually processes $10 \cdot 2 \cdot 8 \cdot 3 \cdot 10^9 = 64\text{GB/s}$, way more than full line rate of 30GB/s. However, the width of the message batch that can be consumed per cycle is not inherently limited by the design: As discussed [subsection 1.2](#), increasing the number of physical lanes from 12 to 24 would require us to increase the size of a message batch from 28 to 56 ECI message headers. This increase will double the throughput of the tracing engine without further changes. However, the increased input width will increase the hardware cost of the tracing engine and induce more routing congestion on the FPGA. In consequence, the overlay graph configuration would scale to a smaller number of states. In [subsection 5.1](#) we have seen that the hardware cost scales linearly in the number of ECI message headers, thus a doubling of the message batch size would lead approximately to a

doubling of the hardware cost. However, we estimate that the tracing engine will still scale to big overlay sizes of hundreds of states and reasonable density and thus still be viable with 2 links.

The only inherent bandwidth limit is imposed by the PCIe backend that transmits the data to the remote host. As discussed, this limit is at 8GiB/s which is lower than the theoretical VC layer bandwidth. We never encountered an overflow of the output stream, but in theory this is possible. Moreover, this is not an inherent limitation of the design and a different output interface would be easy to attach to the existing tracing engine.

Runtime configurability The filter applied to the data stream is configurable at runtime. We have discussed how runtime configurability is achieved in [subsubsection 3.3.2](#). Given an implemented tracing engine on a running system, the engine can execute inherently different filters by loading a binary configuration string onto it and without requiring re-synthesis. We have demonstrated and made use of this runtime configuration in [subsection 4.2](#), where we loaded four different filters right after each other to track different properties of one benchmark.

Independence of ECI stack The tracing engine is independent of the rest of the system and does not rely on a specific implementation of the communication stack or the coherence protocol on the FPGA. The concrete design presented in [section 3](#) achieves a large degree of independence by tapping at the interface between VC and block layer and only making very weak assumptions about this interface. The interface is assumed to provide a fixed width input together with a valid signal that goes high whenever the input is valid. On the VC layer, this interface is provided by the set of VC FIFOs where from each FIFO at most one item per message is dequeued and the conjunction of the valid-ready signals of producer and consumer is used as valid signal. However, this is an assumption that will apply to many other interfaces. Thus using the tracing engine as-is when the VC backends are updated should not pose any problems.

Small hardware cost We analyzed hardware cost in [subsection 5.1](#). We have seen that we can create big overlays with hundreds of states and with relatively high density to accommodate big NFAs while consuming only moderate hardware resources. The parametrizable overlay graph provides the user with significant freedom to trade-off overlay density and size. We have also demonstrated that the wiring between these states scales well, i.e. we can accommodate even very dense overlays like cliques up to relatively large sizes ($C = 100$ as seen in [subsection 5.1](#)). The processing of multiple independent substreams, for instance multiple CLs, is very costly. Our attempt of limiting the hardware cost of accommodating multiple NFAs provides only marginal savings as compared to replicating the whole overlay graph. We don't know if there is a cheaper mechanism to track multiple VC substreams simultaneously. Still, for simple NFAs we can track the state of up to 2^7 independent CLs. A major factor of hardware cost is the PCIe interface to write out the filtered stream, even in very big configurations, PCIe makes up about 50% of the total hardware cost of the tracing engines, as measured by both LUTs and FFs. Getting rid of PCIe would be a good way of further improving the hardware cost of the tracing engine.

The small hardware cost can be attributed to the sparse overlay layout, the homogeneous transition requirement and the abolishment of epsilon transitions between arbitrary states. It should be no problem to run even big instances of the tracing engine alongside major applications that require most hardware resources of the FPGA.

Compile time and runtime flexibility The design of the tracing engine is flexible, both at compile time and at runtime. It can be re-targeted with relative ease to a different interface (the block layer) at compile time. We have achieved this compile time flexibility by a modular design of both the HDL code and of the

frontend, that compiles NFA descriptions into filter configurations. We have demonstrated this flexibility in [subsection 4.1](#), where we have re-implemented the previous work [9] with only very small changes to our design.

Moreover, the design provides large flexibility at runtime. The experiments in [section 4](#) demonstrate that we can express various interesting patterns by configuring a generic overlay graph with different NFAs at runtime. Of course not everything in the tracing engine is runtime configurable. In particular, the expressible filters are limited both by the choice of the input reduction and by the size of the overlay graph, both of which are fixed at compile time. We have discussed the possibility of avoiding input reduction to increase the amount of patterns that can be specified at runtime in [subsection 3.2](#). The fundamental trade-off is that a design that doesn't perform input reduction will scale to fewer states than a design that reduces bus widths with input reduction. It is impossible to say which approach is better because some filters may require large NFAs while other filters may require the unreduced input data. We believe the design presented in [section 3](#) strikes a good balance between scalability and expressiveness: The input reduction we apply extracts all the information we considered to be useful but no more.

6.2 Limitations and future work

In the following, we list several shortcomings of the current design and provide ideas as to how these could be made up for.

- This work uses PCIe to both control the tracing engine and to write out the captured traces. As discussed, this approach has several downsides. Most importantly, the XDMA IP core providing PCIe connectivity to the FPGA uses considerable of hardware resources. Moreover, Enzian provides ample alternatives for a more natural setup. First, ECI provides a very high-speed connection between the CPU and the FPGA. However, since the tracing engine filters ECI traffic, it is probably desirable not to stream out traces over ECI while tracing is active as this would induce additional ECI traffic and potentially obfuscate the filtered results. An interesting configuration could be to use the I/O VCs to control the tracing engine and write the traced data to FPGA DRAM. After the trace has been captured, post-processing of the trace could be done on the Enzian CPU by accessing the traces using memory coherence between FPGA and CPU. That is, trace data is still transferred over ECI, but only after the tracing engine has stopped tracing. This approach only makes use of the standard hardware of the Enzian and in particular doesn't require a special PCIe connection to some host.
- The size of the windowing is currently fixed at compile time. It wouldn't be difficult to make it runtime configurable as well. Note that, similar to processing multiple CLs, we will need to provision enough hardware to buffer a window of size n at compile time. In particular, the size of the FIFO buffering the filtered ECI stream needs to have size at least n and the counter keeping track of the number of messages to write out needs to have size at least $\log(n)$. However, given an implementation that provisions hardware for a window of size n , the user can be allowed to set any window size $\leq n$. To do this, we would simply add a configuration register to the tracing engine that sets the windowing size.
- The `Output Reduction` module should remove redundant data from the output stream before writing it out to PCIe. A simple approach to achieving this would be to re-arrange the messages in a message batch such that all valid messages come before all IDLE messages. Then provide a *size* output to the control module to indicate the width of the data that should be written out over PCIe. Currently, the output stream contains the IDLE ECI headers we use for padding to ensure that the message batch format is always the same. The advantage of keeping this IDLE data is that the VC index of each

message is still implicit in its position in the message batch. Removing this IDLE data would require us to explicitly annotate each message with its VC index. Currently, we can use the previous work [4] almost as-is to parse the output trace produced by the tracing engine. Each additional annotation of the output stream will make the parsing of it more complicated. However, removing this IDLE data can save bandwidth and reduce the size of the trace stored on the remote host. Saving bandwidth in this way is relevant for two reasons: First, the output interface PCIe has limited bandwidth and streaming out too much data could lead to data loss. Second, the user currently needs to process data that contains some IDLE data. This IDLE data can easily be removed using *grep*, but for convenience and storage reasons it would be nicer to avoid tracing it all together.

- The tracing engine is able to process data from multiple sources at the same time. In this work we have not explored this option but we believe that it enables interesting use cases. For example, we could allow external triggers that cause immediate capture of a window of messages. This trigger could be controlled by the DirC or some other module, or even user input. In this case, the NFA would be trivial and only distinguish between the two states where this trigger is either high or low. External triggers would allow us to use the tracing engine similar to ILAs in a debugging scenario: The tracing engine takes two kinds of inputs, trigger inputs and data inputs. The input decoding allows the user to specify conditions on the trigger inputs, for instance, an error signal goes high. Whenever the trigger inputs satisfy this condition, the tracing engine outputs a window of n messages that the user can inspect to understand the faulty behaviour.
- At the time of writing, the VC layer is still in development. In particular, the interfaces to different VC backends are currently in different modules. The current implementation of the tracing engine thus only taps VCs 2 to 11, i.e. the coherence VCs. It ignores I/O VCs 0,1 and VCs 12,13. There is no inherent limitation in the design that would make it hard to process these VCs as well, the only reason for this is that not all VCs are accessible from one module. This deficiency can easily be fixed as soon as all VC backends are consolidated in one module.
- The tracing engine is restricted to processing one NFA at a time. As we have seen in [subsection 4.2](#), this can make experiments more cumbersome or even impossible. In the particular case of [subsection 4.2](#), we needed to run a benchmark four times, each time with a different filter, in order to get all the data we needed. The fundamental reason for this restriction is that there is only one output stream: In theory, the tracing engine can process multiple NFAs in parallel, it can also emit multiple different *accept* outputs as discussed in [subsection 3.3.2](#). However, there is currently no mechanism to multiplex multiple output streams over the PCIe interface to the remote host. One way to achieve this would be for the `Output_Reduction` module to annotate messages with the exact *accept* output of the NFA. If each NFA outputs a different *accept* output, a message batch would be labelled with the indices of all NFAs that were in an accepting state after processing it.
- Counting occurrences of events is cumbersome in the given framework. The inherent limitation is that standard NFAs are not good at counting. One way we can achieve counting in the current framework, is to make the event we want to count an accepting state in the NFA, then count the number of messages in the output stream. This is what we have done in the experiment in [subsection 4.2](#). Every message in the output stream is a witness of one occurrence of the event. However, this mechanism doesn't work in cases where we want to count an event as part of a more complex filter pattern. In these cases, we need to do explicit counting in the NFA, i.e. in order to count n instances of an event, we need n separate NFA states. The Micron AP [17] provides counter elements that behave similar to normal STEs but that count the number of times a predecessor state is active and become active themselves

when this counter is $<, =, >$ some value. A similar mechanism could be implemented quite easily in our tracing engine. A difficult follow up question is how many counter elements to allocate and in which locations on the overlay graph. Such a diversification of processing elements always comes at the cost of making overprovisioning enough units of every element more difficult.

- Tracking n CLs independently by implementing multiple substreams in the NFA turned out to be very costly. Our approach doesn't scale to more than 128 CLs because the demultiplexing of the CL index requires significant amounts of logic and routing resources. The inherent difficulty is that every ECI message in a message batch can address a different CL, thus the demultiplexing must be performed separately for every message in a batch. Moreover, performing the multiplexing at every STE results in much duplicated hardware. Because all STEs receive the same messages, it would be possible to perform the demultiplexing centrally and send the demultiplexed CL index to every STE. However, this would dramatically increase the routing resources between a central location (e.g. the `Input Reduction`) and each STE: Tracing NCLS CLs with NHEADERS ECI messages in a message batch requires a bus of width $NCLS \cdot NHEADERS$. It is not clear to us if this would be better and if there even exists a way of implementing this functionality that is much cheaper than the current design.
- We explicitly discard payloads in the tracing engine. The current design could be extended in two ways to consider payloads: First, we could keep the payloads in the output stream of messages for user inspection but restrict input decoding to header information as before. Second, we could additionally allow the user to specify trigger patterns that use payload data. The first option is achievable with minimal changes to the current design. In [Figure 20](#), only the lower data-path needs to be changed, in particular, the two FIFOs that buffer the message batch need to be wider. The only reason not to include ECI payload in the output stream is the limited PCIe bandwidth and the additional cost incurred by making the FIFOs wider. Second, to achieve filter specifications that use payload data, we would need to allow what is commonly called “deep network inspection”. This is for instance done with network packets in Snort to detect malicious traffic. These patterns are usually restricted to string matching inside the payload of a *single* packet. When processing payloads we won't be able to rely on a fixed structure but we will need to perform full text search, i.e. match a RegEx in the payload of packages instead of matching header field bits. It is not clear to us, if this is useful in the context of coherence data.
- A major limitation of runtime configurability in this work is the input reduction: The user needs to decide at compile time what ECI header information will be relevant to them. One way of achieving more flexibility at runtime would be the following: The user provides a bitmask that selects all the ECI header bits that should be included in the reduced input. Same as now, the `Input Decoding` module provides a set of CFGLUT5s that consume these header fields and that can be configured at runtime to check some predicate on them. This requires us to overprovision both the number of extracted header bits in the form of buses between `Input Reduction` and the STEs, and the number of CFGLUT5s in the `Input Decoding`. The additional flexibility is limited by the fact that different ECI headers, even on the same VC, can contain different fields and have a different structure. Maximizing runtime flexibility while maintaining small hardware cost remains a difficult problem.
- The frontend and the tracing engine are currently not linked in any way. The frontend, however, relies on several details of the hardware implementation: The parameters of the overlay graph and the order in which the runtime configuration bitstring is shifted into the hardware elements are required for the frontend to generate a correct configuration. Currently, it is the responsibility of the user to provide these details by hand. While the overlay parameters can easily be provided by hand, the order of the

configuration poses more problems. Mismatches between what the frontend and the tracing engine consider the correct configuration format were a major source of bugs in the development of this work. One approach of reducing the potential for bugs is to provide a formal description of the order in which the config chain is shifted into the hardware elements. For instance:

```

NFA
  STE[0][0][0]
    predecessor (NPREDS)
    logging     (1)
    accepting   (1)
    starting    (1)
    input_decoding (NHEADERS * 32 + 1)
  STE[0][0][1]
  ...
  ...

```

The hardware elements requiring configuration are the leaves in a tree of hardware elements. Each leaf specifies its hardware element and the number of configuration bits it needs. The frontend could then be linked to this description by modularizing the code into separate functions that provide the configuration for the different types of hardware elements. That is, different functions would compute *predecessor*, *logging*, *accepting*, *starting* and *input_decoding* of each STE. Such a modularization is already partly done by a separate function that computes the configuration for the Input Decoding modules (as discussed in [subsection 3.4.1](#)). Such a setup would make the link between HDL description and frontend code explicit, further modularize the frontend code and allow for simple error checking in the frontend.

References

- [1] G. Alonso, T. Roscoe, D. Cock, M. Owaida, K. Kara, D. Korolija, Z. Wang, *et al.*, “Tackling hardware/software co-design from a database perspective,” in *Proceedings of the 6th biennial Conference on Innovative Data Systems Research (CIDR), Amsterdam, Netherlands, January 2020.*, 2020.
- [2] Various, “An open, general, cpu/fpga platform for os research.” under submission, 2021.
- [3] Various, “Interlaken protocol definition: A joint specification of cortina systems and cisco systems, rev. 1.2, 52 pp.(oct. 7, 2008).” http://interlakenalliance.com/wp-content/uploads/2019/12/Interlaken_Protocol_Definition_v1.2.pdf, 2008. Accessed: 2021-09-02.
- [4] J. Meier, “Tools for cache coherence protocol interoperability,” Master’s thesis, ETH Zürich, mar 2020. Masters thesis, ETH Zürich.
- [5] R. Sidhu and V. K. Prasanna, “Fast regular expression matching using fpgas,” in *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’01)*, pp. 227–238, IEEE, 2001.
- [6] M. Y. Vardi, “An automata-theoretic approach to linear temporal logic,” in *Logics for concurrency*, pp. 238–266, Springer, 1996.
- [7] Various, *UltraScale Architecture Configurable Logic Block*. Xilinx, Xilinx, 2017.

- [8] Various, *Vivado Design Suite User Guide*. Xilinx, Xilinx, 2021.
- [9] T. Kuchler, “Fpga-based real-time filtering and analysis of enzian interconnects,” Master’s thesis, ETH Zürich, sep 2020. Masters thesis, ETH Zürich.
- [10] M. Roesch *et al.*, “Snort: Lightweight intrusion detection for networks.,” in *Lisa*, vol. 99, pp. 229–238, 1999.
- [11] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen, “Efficient event processing through reconfigurable hardware for algorithmic trading,” *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1525–1528, 2010.
- [12] G. Cugola and A. Margara, “Tesla: a formally defined event specification language,” in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, pp. 50–61, 2010.
- [13] J. Teubner, L. Woods, and C. Nie, “Skeleton automata for fpgas: reconfiguring without reconstructing,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 229–240, 2012.
- [14] L. Woods, J. Teubner, and G. Alonso, “Complex event detection at wire speed with fpgas,” *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 660–669, 2010.
- [15] Lin Tan and T. Sherwood, “A high throughput string matching architecture for intrusion detection and prevention,” in *32nd International Symposium on Computer Architecture (ISCA’05)*, pp. 112–122, 2005.
- [16] A. V. Aho and M. J. Corasick, “Efficient string matching: an aid to bibliographic search,” *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.
- [17] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, “An efficient and scalable semiconductor architecture for parallel automata processing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014.
- [18] R. Karakchi, C. Daniels, and J. Bakos, “An overlay architecture for pattern matching,” in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160, pp. 165–172, IEEE, 2019.
- [19] R. Karakchi, L. O. Richards, and J. D. Bakos, “A dynamically reconfigurable automata processor overlay,” in *2017 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pp. 1–8, IEEE, 2017.
- [20] M. Becchi and P. Crowley, “Extending finite automata to efficiently match perl-compatible regular expressions,” in *Proceedings of the 2008 ACM CoNEXT Conference*, pp. 1–12, 2008.
- [21] C. McCreesh, P. Prosser, and J. Trimble, “The glasgow subgraph solver: Using constraint programming to tackle hard subgraph isomorphism problem variants,” in *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings* (F. Gadducci and T. Kehrer, eds.), vol. 12150 of *Lecture Notes in Computer Science*, pp. 316–324, Springer, 2020.
- [22] L. Gurobi Optimization, “Gurobi optimizer reference manual,” 2021.

- [23] R. Lougee-Heimer, “The common optimization interface for operations research: Promoting open-source software in the operations research community,” *IBM Journal of Research and Development*, vol. 47, no. 1, pp. 57–66, 2003.
- [24] Z. István, D. Sidler, and G. Alonso, “Runtime parameterizable regular expression operators for databases,” in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 204–211, IEEE, 2016.

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

High-speed Tracing of Coherence Traffic using FPGAs

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

Bräschin

First name(s):

Mannel

With my signature I confirm that


- I have committed none of the forms of plagiarism described in the '[Citation etiquette](#)' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zürich, 3.9.2021

Signature(s)



For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.