

hXDP: Efficient Software Packet Processing on FPGA NICs

Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, Roberto Bifulco





UNIVERSITA' degli STUDI di ROMA
TOR VERGATA



consorzio nazionale
interuniversitario
per le telecomunicazioni



Axbryd
Building a lighter world

NEC

Joint work of

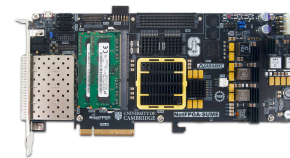
Background

- Network packet processing is ubiquitous



- CPU performance issues
 - Starvation of Moore's and Dennard's scaling laws
 - Need to save CPU cycles for things that cannot be done elsewhere 😊

- Welcome to network accelerators!
 - One size doesn't fit all



Much smaller
Uses 20W



Takes space
Uses >200W

Why FPGAs?

The Microsoft Catapult Project

Derek Chiou

Partner Hardware Architect, Microsoft
Research professor, University of Texas at Austin

All new Microsoft Azure and Bing servers are being deployed with an FPGA that server and the data center network and on the PCIe bus. The FPGA is currently be networking on Azure machines and search on Bing machines, but could very c retargeted to other uses as needed. In this talk, I will describe how we decided on this data center model it introduces, and the benefits it provides.

Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin,

To meet the computational demands required of deep learning, cloud operators are turning toward specialized hardware for improved efficiency and performance. Project Brainwave, Microsoft's principal infrastructure for AI serving in real time, accelerates deep neural network (DNN) inferencing in major services such as Bing's intelligent search features and Azure. Exploiting distributed model parallelism

olution

- Increasing dep
- Machine Learn
- 5G radio acces

FPGA FOR 5G: RE-CONFIGURABLE HARDWARE FOR NEXT GENERATION COMMUNICATION

Vinay Chamola, Sambit Patra, Neeraj Kumar, and Mohsen Guizani

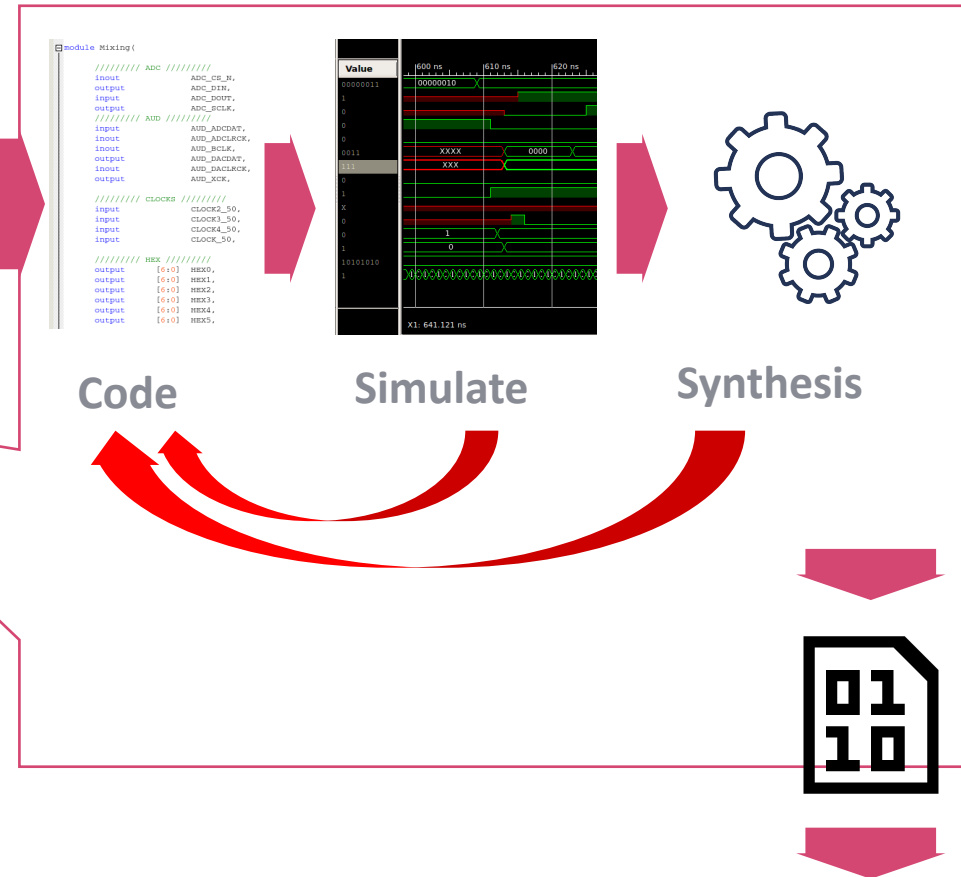
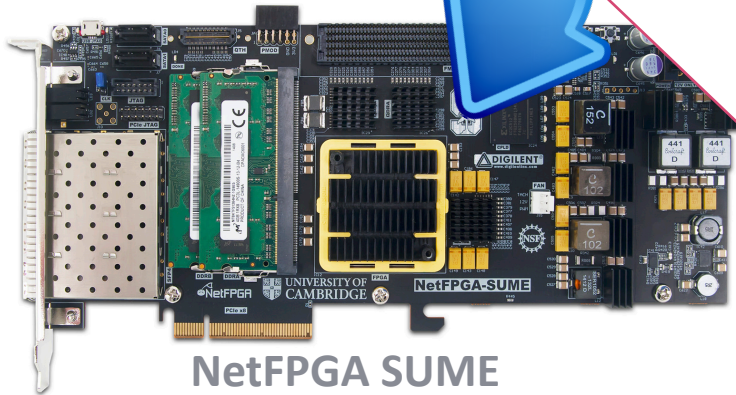
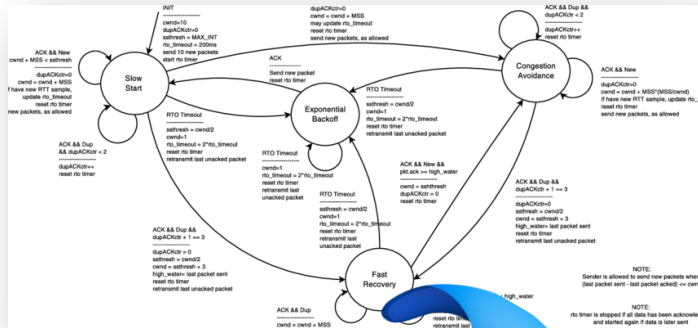
Microsoft Corporation

DNNs continue to deliver major breakthroughs in challenging AI domains such as computer vision and natural language processing, their computational demands have steadily outpaced the performance growth rate of standard CPUs. These trends have spurred a Cambrian explosion of specialized hardware, as large companies, startups, and research efforts shift *en masse* towards energy-efficient accelerators such as GPUs, FPGAs, and neural processing units (NPU)s¹⁻³ for AI workloads that demand performance beyond mainstream proces-

The problem with FPGA-based NICs

Programming them is Hard!

Network Function Logic



Making programming easier

Code

Simulation

Synthesis

All the approaches assume that a significant portion of the FPGA is dedicated to networking tasks, consuming a significant amount of HW resources

Expressive

Hardware expertise

ClickNP [Sigcomm '16],
Emu [ATC '17]

NF Logic focused

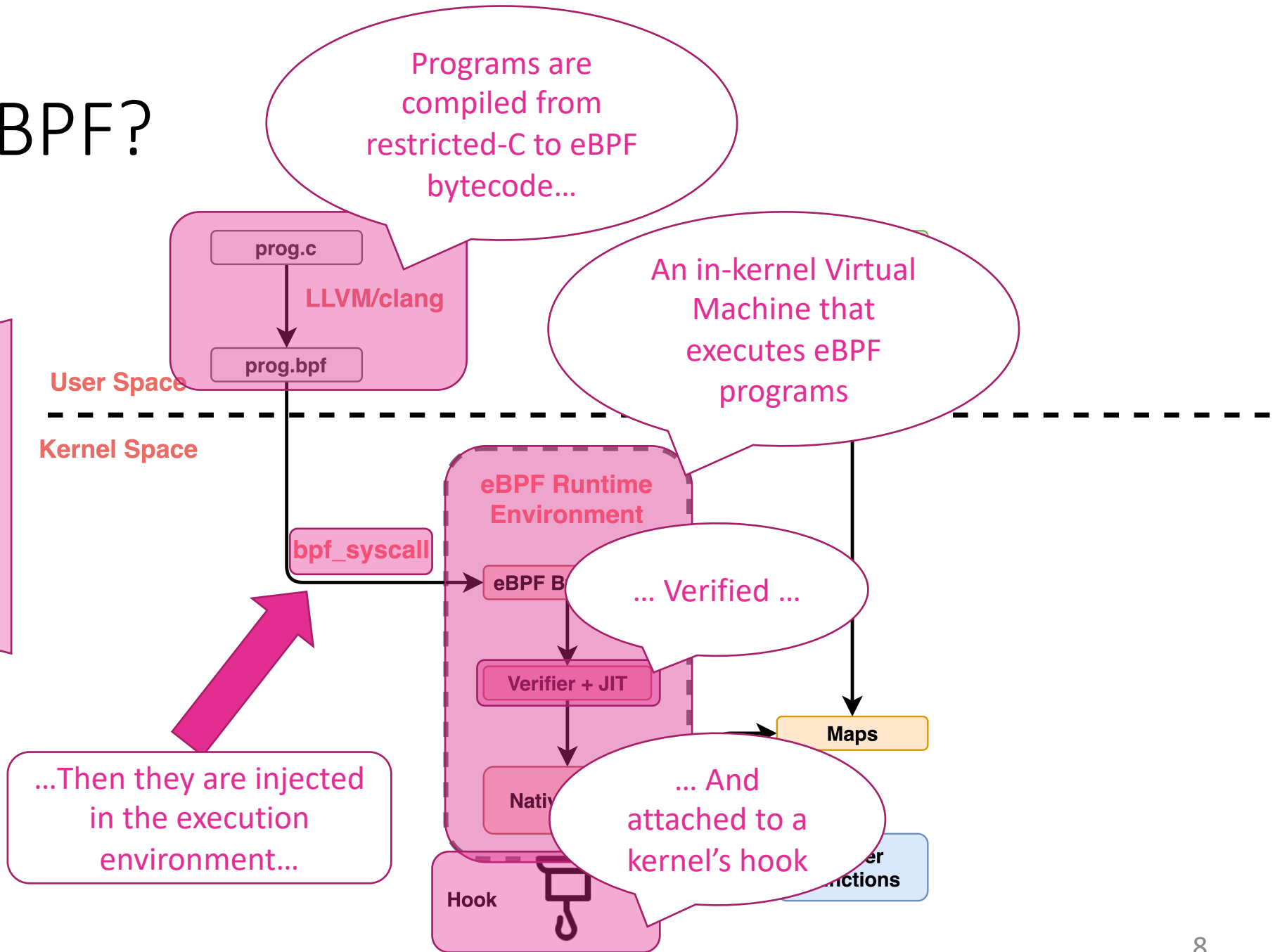
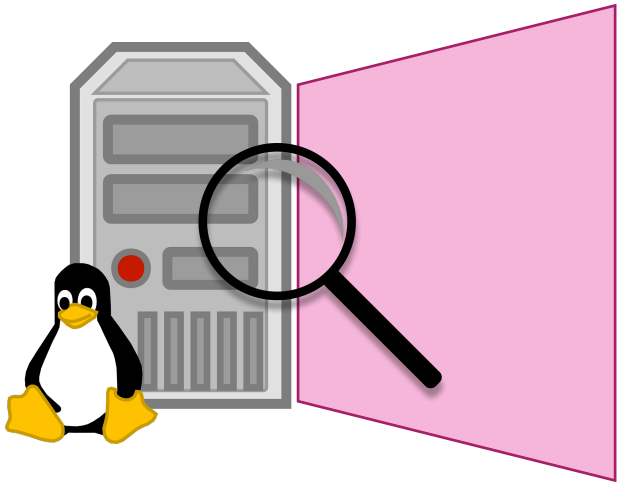
Exotic/Limited prog. model

P4 [CCR '14],
Domino [Sigcomm '16]
FlowBlaze [NSDI '19]

Our approach

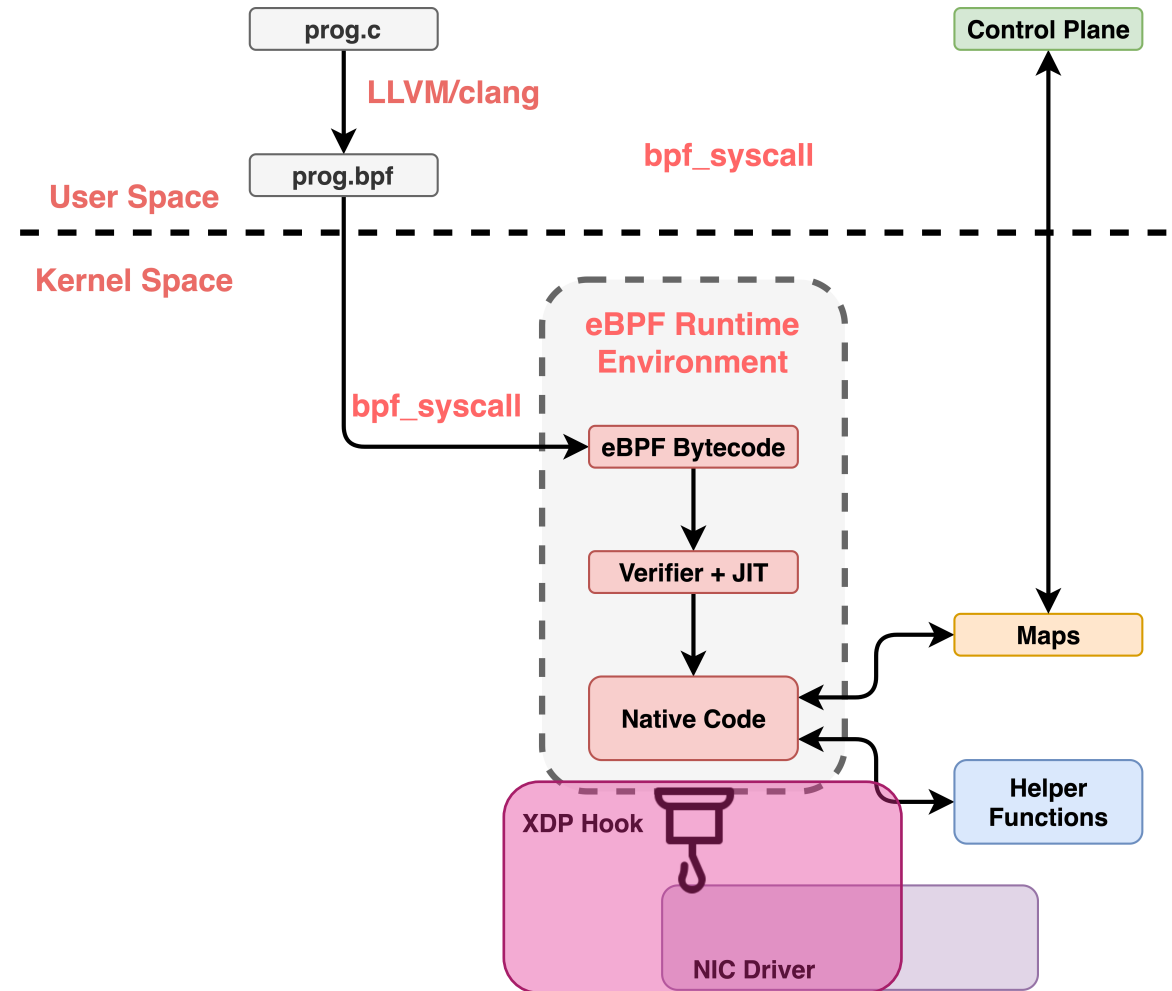
1. Take the eBPF infrastructure
 - Packet filter implemented in Linux Kernel 4.18+
 - RISC-inspired in-kernel virtual machine that executes eBPF bytecode
 - In-kernel “Maps” and “Helper Functions”
2. Re-create the same infrastructure on the FPGA
 - VLIW core to execute optimized eBPF bytecode
 - Hardware-based Maps and Helper Function
3. “Offload” the eBPF execution to the FPGA

What is eBPF?

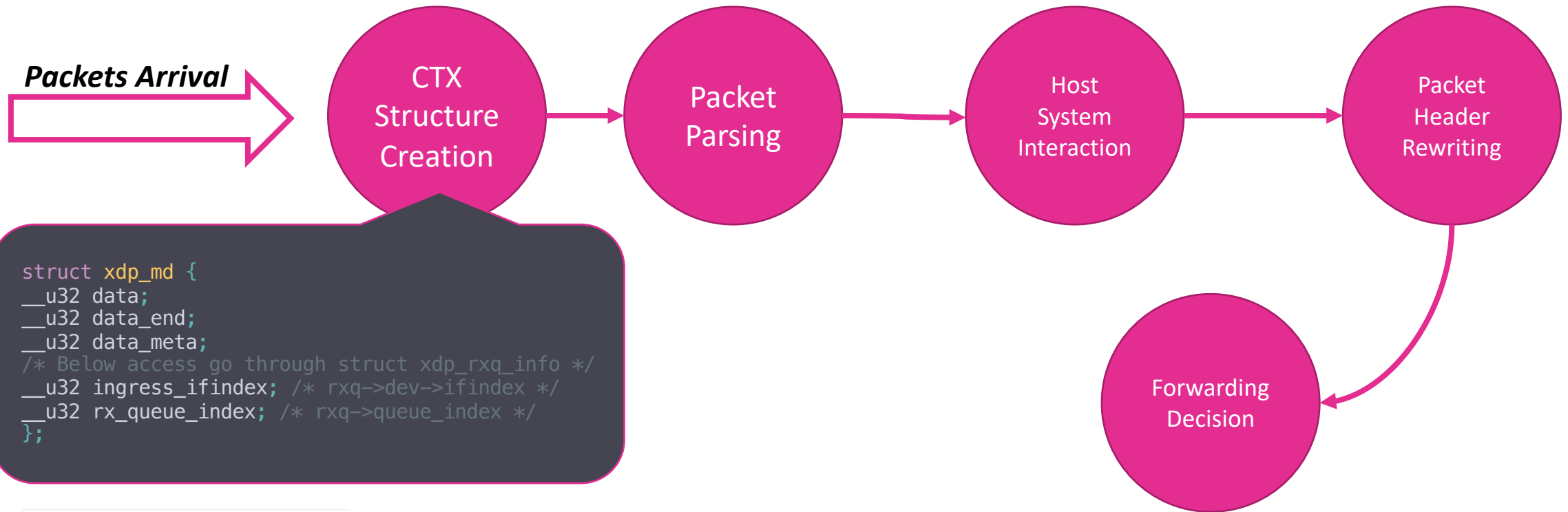


eXpress DataPath

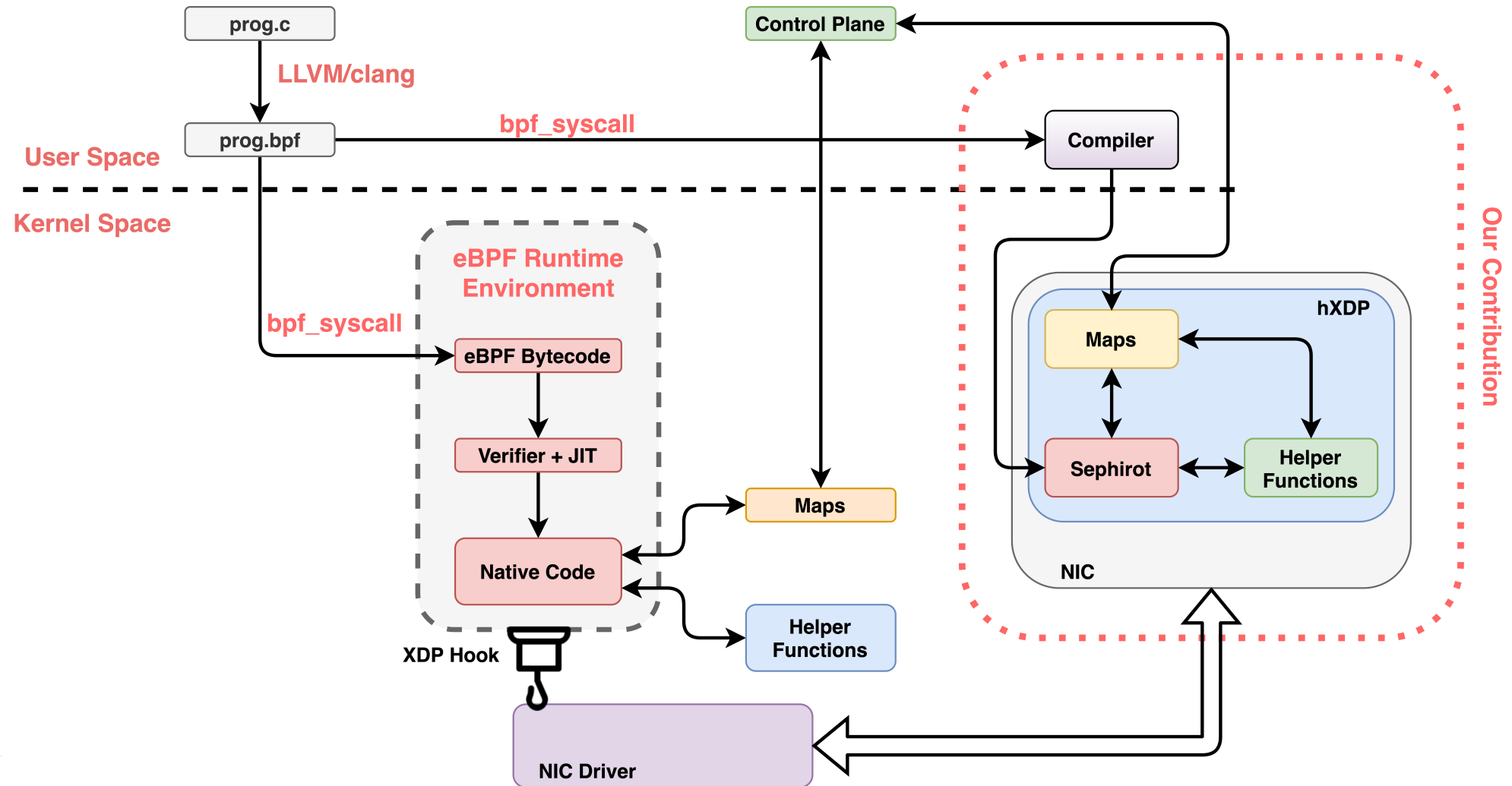
- One of the many eBPF hooks
 - At the earliest point in the stack
- Avoids kernel bypass
- CPU load scales with traffic load
- Transparent to the host



XDP program life-cycle

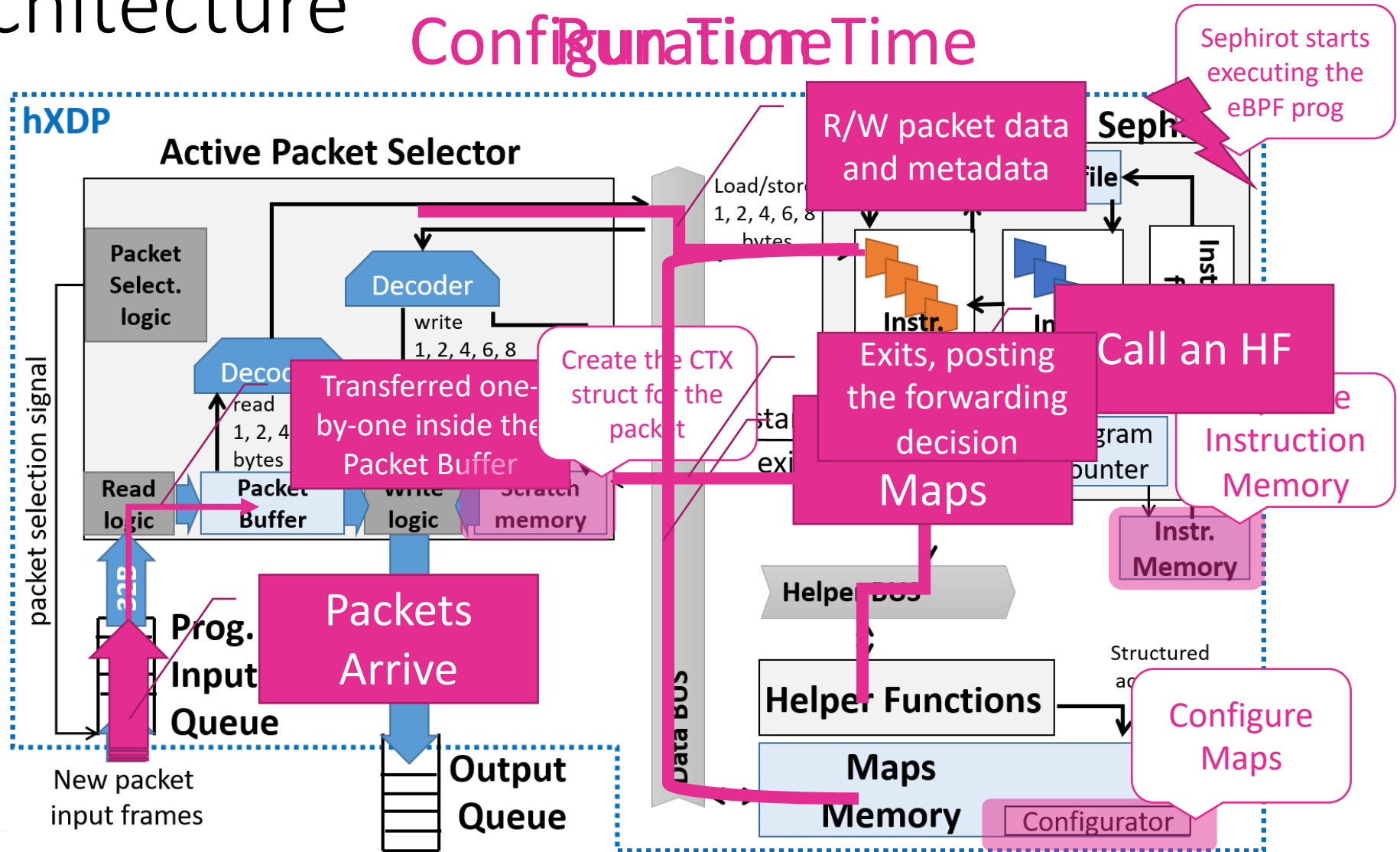
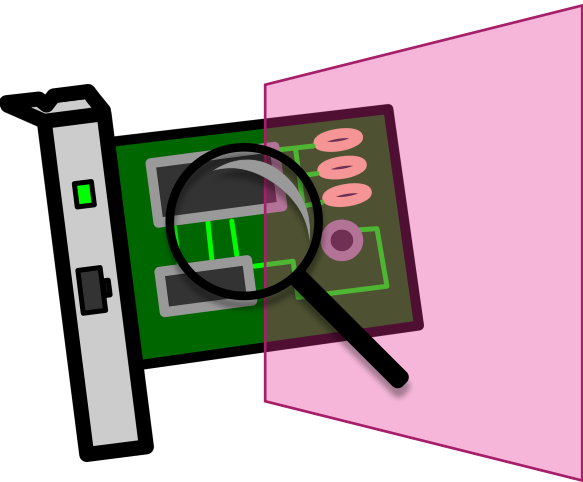


hardware eXpress DataPath



hXDP architecture

Configuration Time



Challenges

- hXDP resource occupancy must be small
 - Minimize HW resources requirements
 - Allow designers to fit different Accelerators on the FPGA

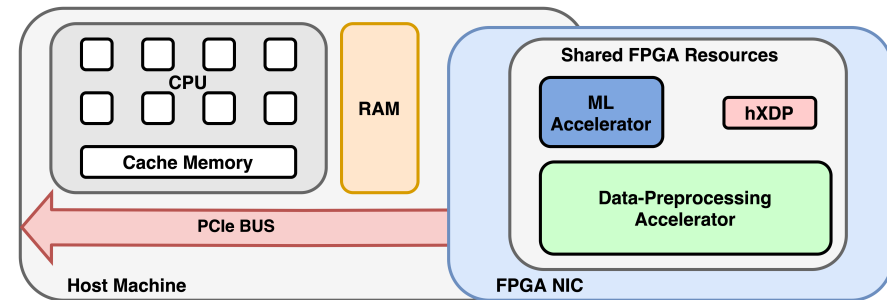
- hXDP performance must be comparable to the ones of an x86 CPU
 - be as fast as a server-grade CPU core
 - FPGAs is clocked at 5x-10x lower frequency than server CPUs

Challenge: make it small!

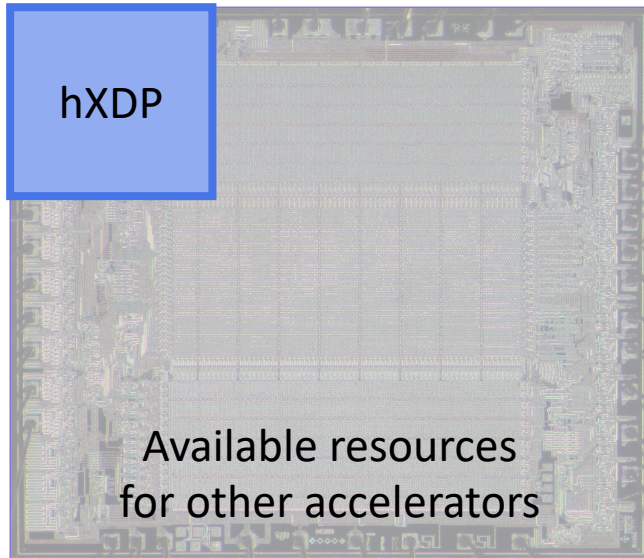
- We assume the FPGA is used for other accelerators

- hXDP Design Principles

- Keep hardware simple
- Adapt ISA to simplify HW design and gain performance
- Move the ILP extraction complexity to the compiler/optimizer



hXDP resources utilization



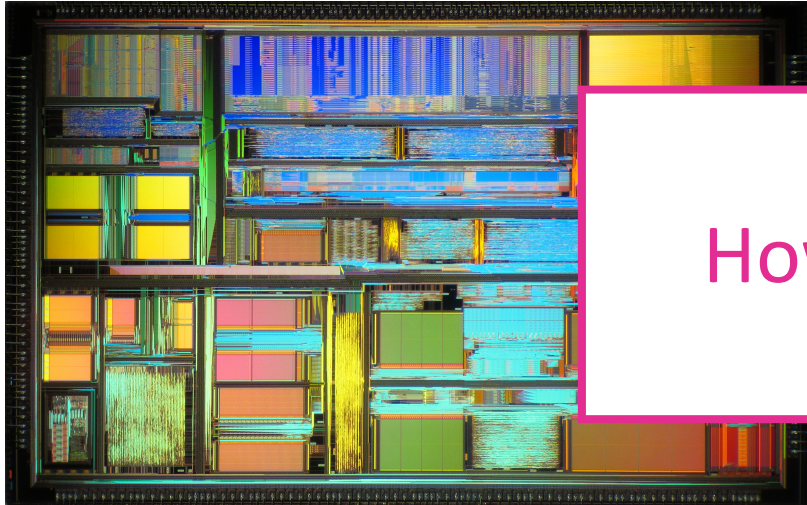
NetFPGA Virtex-7 Die

Table 1

COMPONENT	LOGIC	REGISTERS	BRAM
PIQ	215, 0.05%	58, <0.01%	6.5, 0.44%
APS	9K, 2.09%	10K, 1.24%	4, 0.27%
SEPHIROT	27K, 6.35%	4K, 0.51%	-
INSTR MEM	-	-	7.7, 0.51%
STACK	1K, 0.24%	136, 0.02%	16, 1.09%
HF SUBSYSTEM	339, 0.08%	150, 0.02%	-
MAPS SUBSYSTEM	5.8K, 1.35%	2.5K, 0.3%	16, 1.09%
TOTAL	42K, 9.91%	18K, 2.09%	50, 3.40%
W/ REFERENCE NIC	80K, 18.53%	63K, 7.3%	214, 14.63%

Closed timing @156.25MHz an a NetFPGA-SUME

Challenge: make it fast!



x86

- Clock Frequency: 2-4 GHz
- Hardware-enhanced ILP extraction
- Deep Pipeline stages
- Specialized iterative execution

How to fill the gap?



FPGA

- Clock Frequency: 100-350 MHz
- Not suited for any complex ILP hardware*
- Short pipeline stages
- Killer app: parallel execution

Filling the gap

- Execute eBPF bytecode in a specialized VLIW CPU
 - All the complexity for code parallelization is pushed at “compile” time

- To illustrate code optimizations, we will use a simple eBPF UDP firewall program

warding

- Code Optimization
 - eBPF Instruction Set Architecture extension
 - Pruning of unnecessary instructions

Optmizing eBPF: zeroing

```
SEC("xdp_fw")
int xdp_fw_prog(struct xdp_md *ctx)
{
    void *data_end = (void *) (long) ctx->data_end;
    void *data = (void *) (long) ctx->data;

    struct flow_ctx_table_leaf new_flow = {0};
    struct flow_ctx_table_key flow_key = {0};
    struct flow_ctx_table_leaf *flow_leaf;

    struct ethhdr *ethernet;
    struct iphdr *ip;
    struct udphdr *l4;

    int ingress_ifindex;
    uint64_t nh_off = 0;
    u8 port_redirect = 0;
    int ret = XDP_PASS;
    u8 is_new_flow = 0;
    int vport = 0;
}
```

eBPF Bytecode

0:	61 13 04 00 00 00 00 00	r3 = *(u32 *) (r1 + 4)
1:	61 12 00 00 00 00 00 00	r2 = *(u32 *) (r1 + 0)
2:	b7 04 00 00 00 00 00 00	r4 = 0
3:	63 4a fc ff 00 00 00 00	*(u32 *) (r10 - 4) = r4
4:	7b 4a f0 ff 00 00 00 00	*(u64 *) (r10 - 16) = r4
5:	7b 4a e8 ff 00 00 00 00	*(u64 *) (r10 - 24) = r4
6:	b7 06 00 00 01 00 00 00	r6 = 1

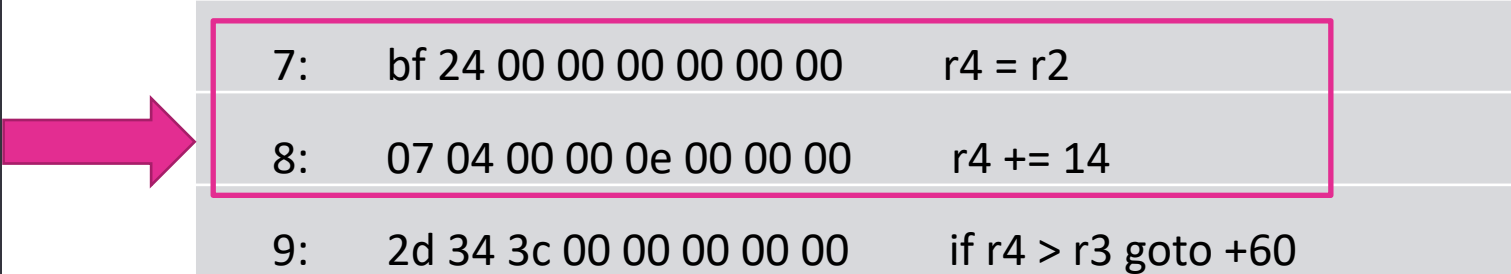
Unnecessary on hardware → we can provide zeroed memory

Optimizing eBPF: 3-operands instructions

```
ethernet:
{
    ethernet = data;
    nh_off = sizeof(*ethernet);


    if (data + nh_off > data_end)
        goto EOP;

    ingress_ifindex = ctx->ingress_ifindex;
    switch (ntohs(ethernet->h_proto))
    {
    case ETH_P_IP:
        goto ip;
    default:
        goto EOP;
    }
}
```



7:	bf 24 00 00 00 00 00 00	r4 = r2
8:	07 04 00 00 0e 00 00 00	r4 += 14
9:	2d 34 3c 00 00 00 00 00	if r4 > r3 goto +60

Merge #7 and #8 into a single instruction



r4 = r2+ 14

Trivial to do in hardware and to recognize at compile time

Optimizing eBPF: Boundary Checks

```
ethernet:
{
    ethernet = data;
    nh_off = sizeof(*ethernet);

    if (data + nh_off > data_end)
        goto EOP;

    ingress_ifindex = ctx->ingress_ifindex;
    switch (ntohs(ethernet->h_proto))
    {
    case ETH_P_IP:
        goto ip;
    default:
        goto EOP;
    }
}
```



7:	bf 24 00 00 00 00 00 00	r4 = r2
8:	07 04 00 00 0e 00 00 00	r4 += 14
9:	2d 34 3c 00 00 00 00 00	if r4 > r3 goto +60

Provide Boundary Check in HW!

Extending eBPF: 6B load/store

```
LDX48_OPC    : std_logic_vector(7 downto 0) := "01011001"; -- 0x59
LDXW_OPC     : std_logic_vector(7 downto 0) := "01100001"; -- 0x61
LDXH_OPC     : std_logic_vector(7 downto 0) := "01101001"; -- 0x69
LDXB_OPC     : std_logic_vector(7 downto 0) := "01110001"; -- 0x71
LDXDW_OPC    : std_logic_vector(7 downto 0) := "01111001"; -- 0x79

STX48_OPC    : std_logic_vector(7 downto 0) := "01010010"; -- 0x52
ST48_OPC     : std_logic_vector(7 downto 0) := "01011010"; -- 0x5a
```



6 Bytes loads & stores

Optimizing eBPF: exit compression

```
EOP:  
return XDP_DROP;
```

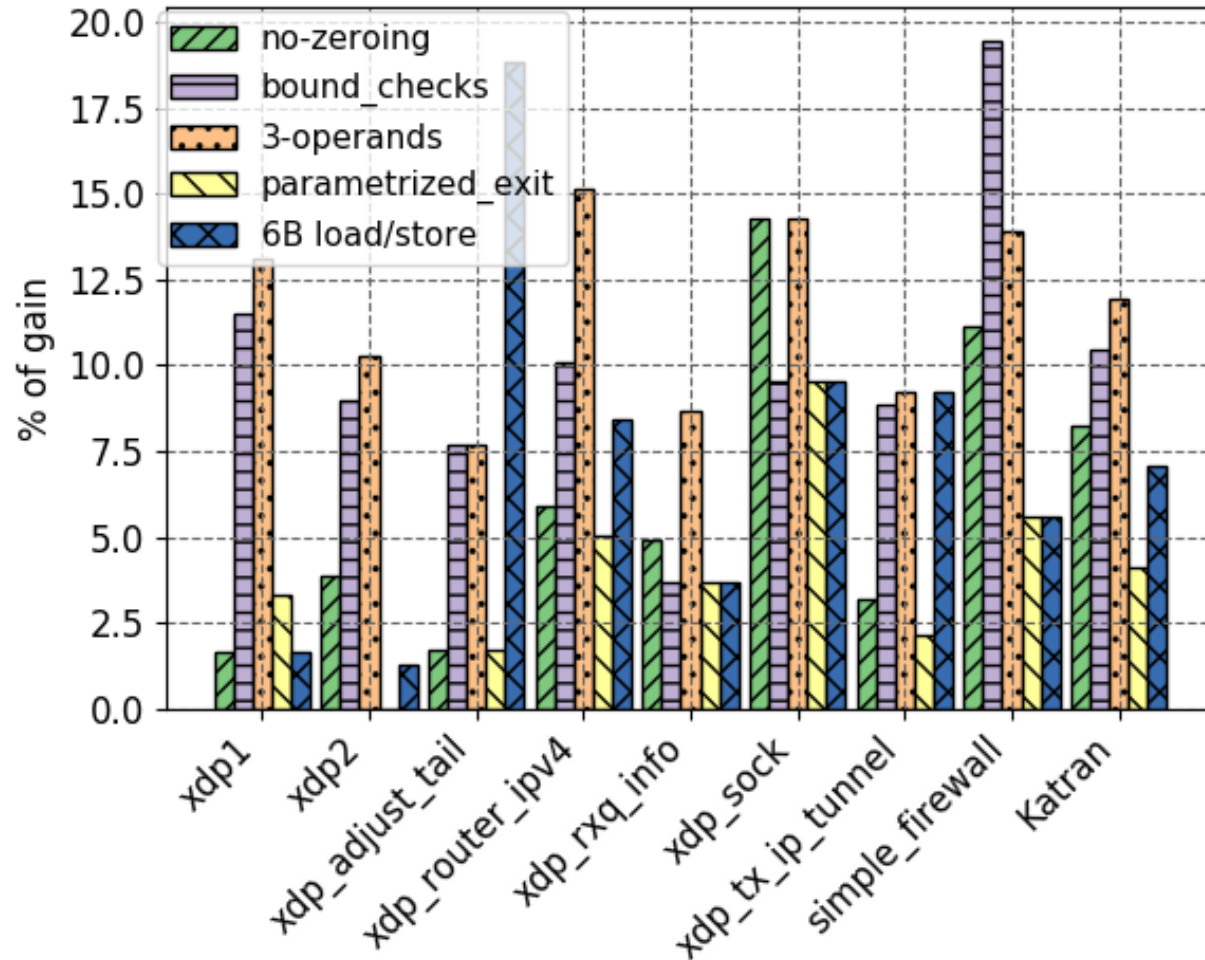
70:	bf 60 00 00 00 00 00 00	r0 = 1
71:	95 00 00 00 00 00 00 00	exit

Define per-action exit



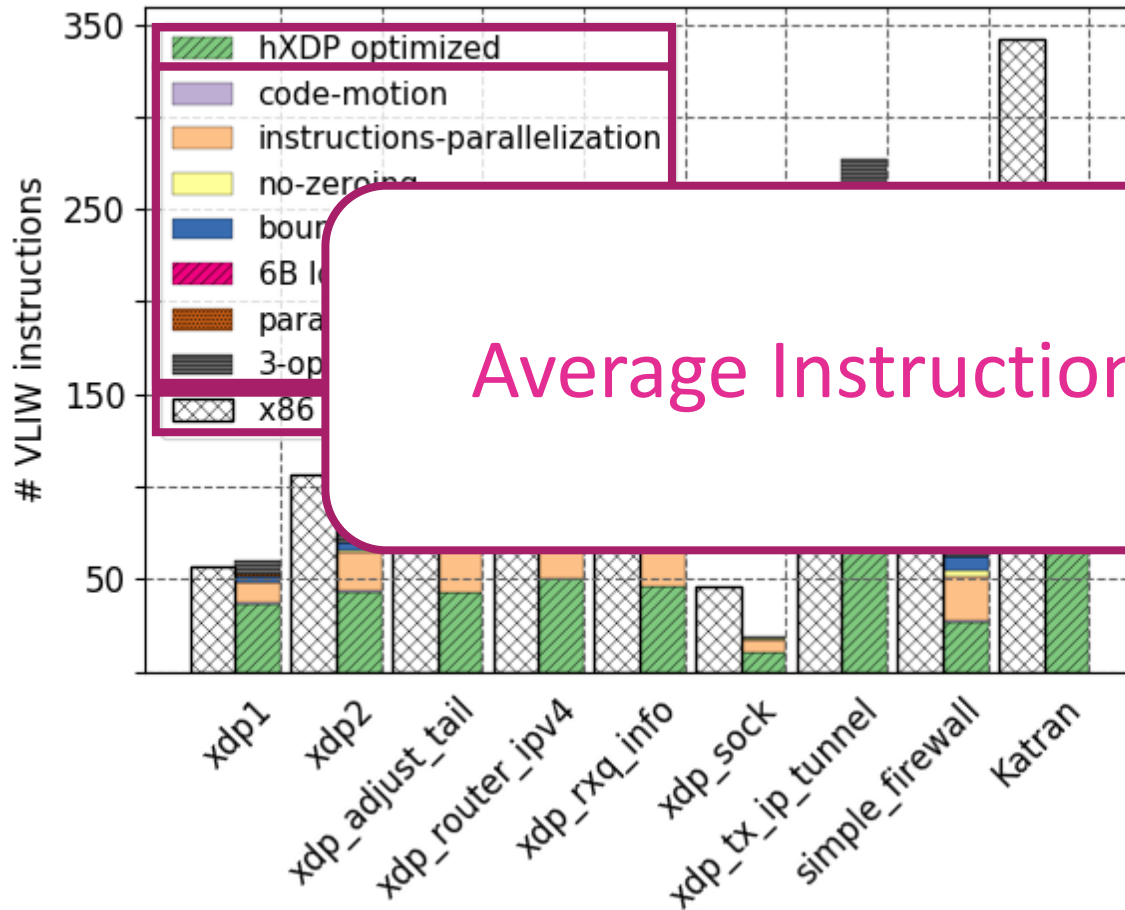
```
exit_drop
```

Impact of Code optimizations on original eBPF bytecode



$$\%gain = 100 \times \frac{\#_{original\ instr} - \#_{optimized\ instr}}{\#_{optimized\ instr}}$$

Overall gain: optimization + ILP



Average Instructions Per Cycle: 2.31

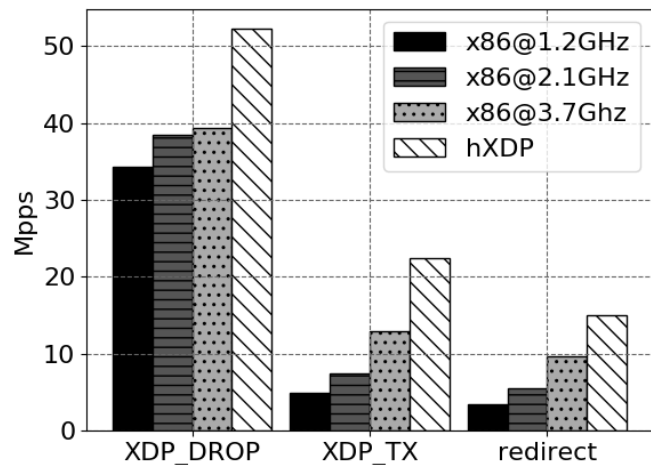
Original eBPF bytecode length

Reduced due to parallelization

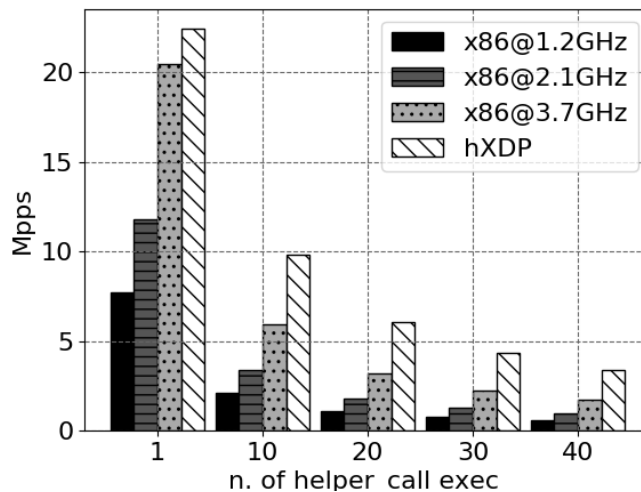
Final VLIW instructions

X86 JIT Compiler mostly expands code

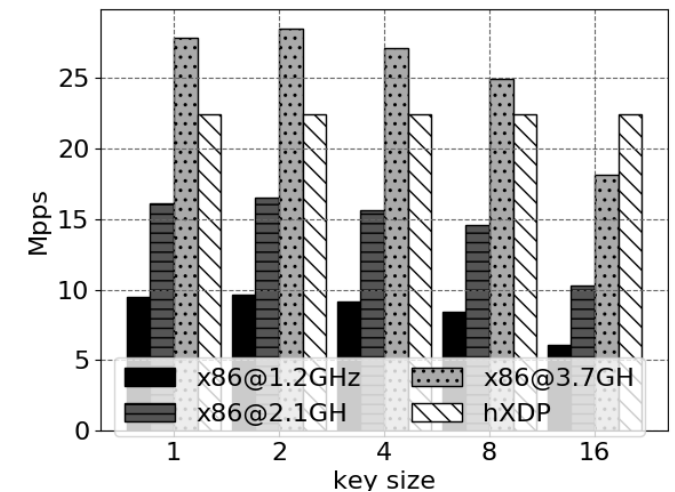
Performance evaluation: Microbenchmarks



Baseline tput measurements
for basic XDP programs

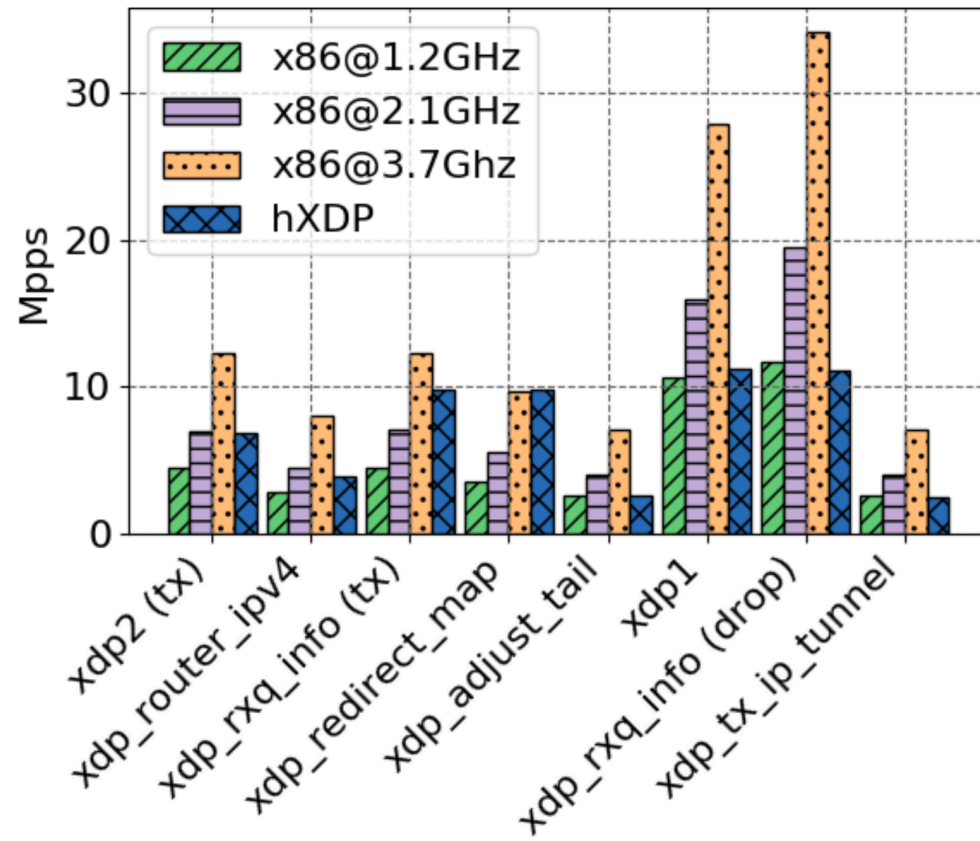


Forwarding tput when
calling a helper function



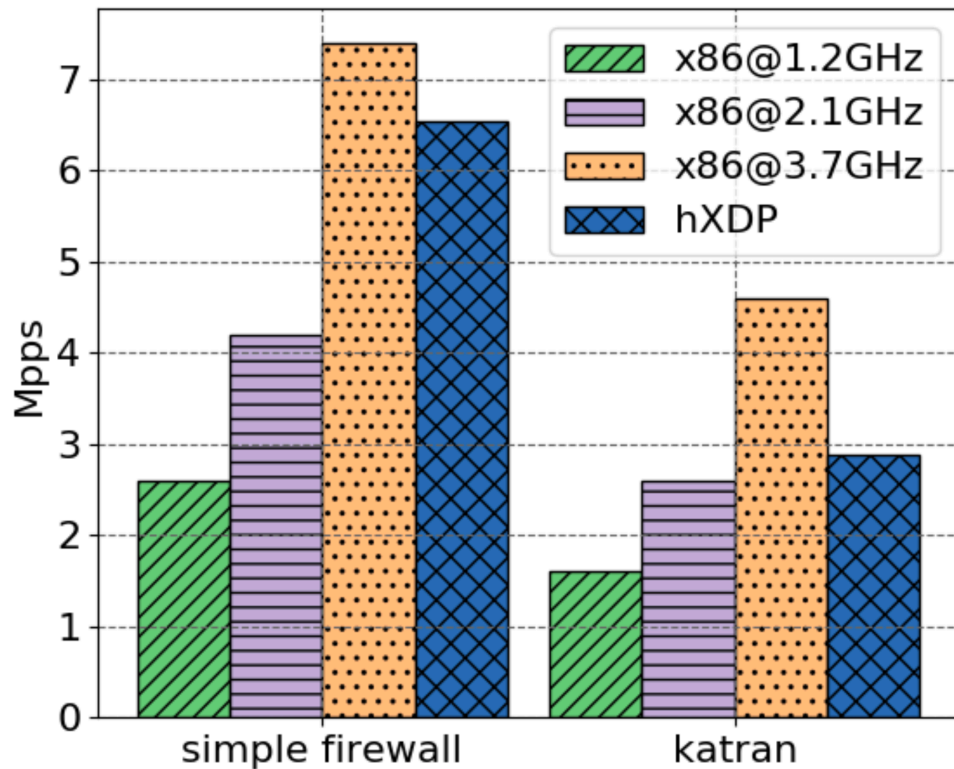
Impact on forwarding tput
on map accesses

Performance evaluation: Linux XDP programs



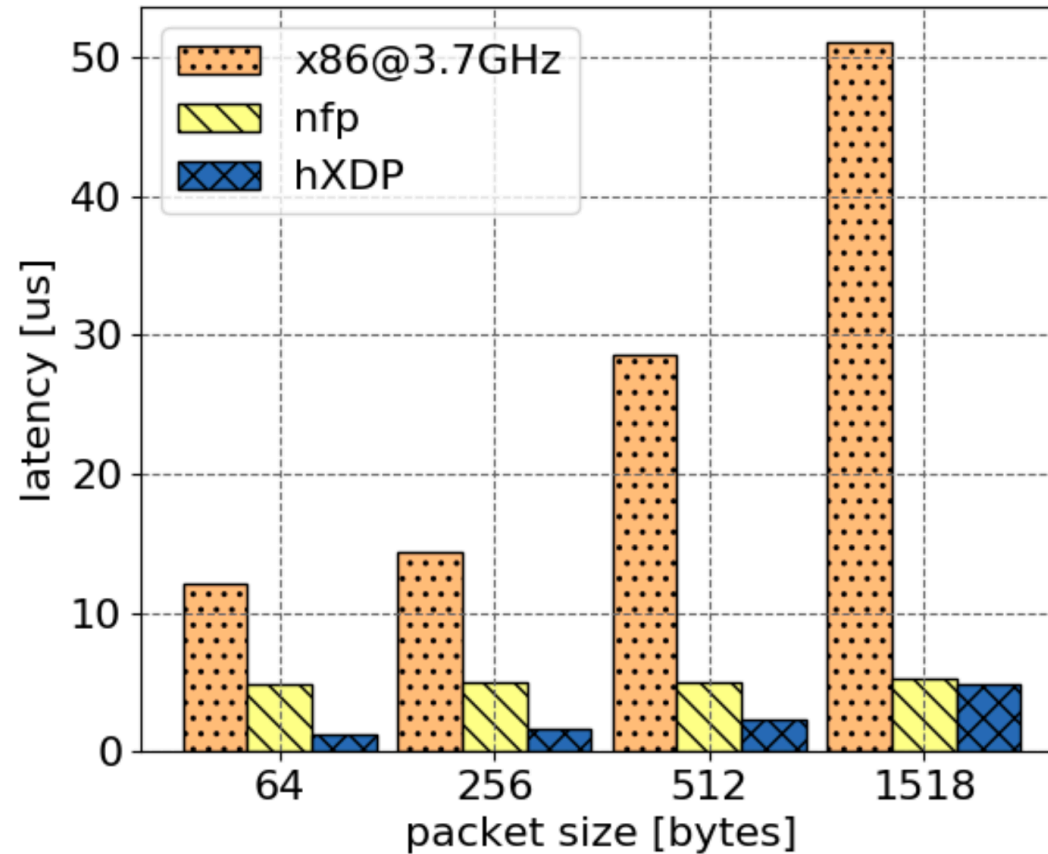
hXDP@156.25MHz has comparable performance to an x86@2.1GHz for programs that live entirely in the NIC

Performance evaluation: Real-world applications



hXDP@156.25MHz
outperforms an x86@2.1GHz

Performance evaluation: Latency Measurements



Conclusion

- **hardware** eXpress DataPath
 - eBPF infrastructure on FPGA NICs
- Benefits
 - Executes unmodified eBPF programs
 - Low Hardware resources
 - Frees up CPU cores with similar performance and 10x better latency

Future Work

- Compiler
 - Re-order memory access instructions to improve ILP
- Hardware Parser
 - Offload large sections of eBPF programs to dedicated HW block
- Multi-core
 - Trade area for forwarding performance
 - Increased memory complexity (need to manage mutex)
 - Use larger memories → DRAMs and HBMs
- ASIC
 - Fixed functionalities (e.g. Sephirot) → put them into custom silicon

hXDP is open and maintained by:



<https://github.com/axbryd/hXDP>





Axbryd

Building a lighter world