UC San Diego
UC San Diego Electronic Theses and Dissertations

Title

Reducing the development cost of customized hardware acceleration for cloud infrastructure

Permalink
https://escholarship.org/uc/item/3z18g6vw

Author
Khazraee, Moein

Publication Date
2020

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Reducing the development cost of customized hardware acceleration for cloud infrastructure

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Moein Khazraee

Committee in charge:

Professor Aaron Schulman, Chair
Professor Ryan Kastner
Professor Farinaz Koushanfar
Professor Alex C. Snoeren
Professor Steven Swanson
Professor Michael B. Taylor
Professor Geoffrey M. Voelker

2020

Copyright
Moein Khazraee, 2020
All rights reserved.

The dissertation of Moein Khazraee is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

____

____

____

Chair

University of California San Diego

2020

DEDICATION

To my parents with everlasting gratitude

EPIGRAPH

"The way you get meaning into your life is to devote yourself to loving others, devote yourself to your community around you, and devote yourself to creating something that gives you purpose and meaning."

—Mitch Albom, Tuesdays with Morrie

TABLE OF CONTENTS

Signature Page . . . iii

Dedication . . . iv

Epigraph . . . v

Table of Contents . . . vi

List of Figures . . . ix

List of Tables . . . xii

Acknowledgements . . . xiii

Vita . . . xv

Abstract of the Dissertation . . . xvi

Chapter 1 Introduction . . . 1
- 1.1 Challenges . . . 5
- 1.2 Insights . . . 9
- 1.3 Thesis statement . . . 10
- 1.4 Contributions and dissertation organization . . . 11

Chapter 2 Background And Related Work . . . 13
- 2.1 Economics of cloud infrastructure . . . 13
- 2.2 Customized hardware accelerators . . . 14
  - 2.2.1 Silicon technology nodes . . . 15
  - 2.2.2 FPGAs . . . 19
  - 2.2.3 Comparing ASICs and FPGAs . . . 21
  - 2.2.4 Custom hardware acceleration development challenges . . . 22
- 2.3 Reducing development cost for in-datacenter compute . . . 23
- 2.4 Reducing development cost for in-network compute . . . 26
- 2.5 Reducing development cost for wireless base stations . . . 29
- 2.6 Summary . . . 32
- 2.7 Acknowledgments . . . 33

Chapter 3 Building A Model For ASIC NRE Cost . . . 34
- 3.1 Principle components of NRE cost . . . 35
- 3.2 Modeling NRE cost . . . 38
- 3.3 Summary . . . 40
- 3.4 Acknowledgments . . . 40

Chapter 4 ASIC Clouds for Datacenter Compute 41
4.1 Bitcoin: An early ASIC cloud 43
4.2 Ramping the technology curve to ASIC cloud 45
4.3 Pareto- and TCO- optimality 48
4.4 Architecture of an ASIC Cloud 50
4.5 Design of an ASIC server 54
4.5.1 ASIC server overview 54
4.5.2 ASIC server model 56
4.5.3 Thermally-aware ASIC server design 57
4.5.4 Linking power density & ASIC server cost 64
4.6 Bitcoin ASIC Cloud design 65
4.6.1 Building a Bitcoin ASIC Cloud 66
4.7 Expanding the methodology to variety of applications 71
4.7.1 Litecoin ASIC Cloud design 71
4.7.2 Video transcode ASIC Cloud design 73
4.7.3 Convolutional Neural Net ASIC Cloud design 76
4.8 When do we go to ASIC Clouds? 79
4.9 Computing Pareto-optimal designs for each tech node 80
4.9.1 Comparing Pareto-optimal points across nodes 80
4.9.2 Comparison of TCO-optimal designs across nodes 82
4.9.3 Server cost change across tech nodes 85
4.10 Finally, NRE+TCO optimal ASIC Clouds 87
4.10.1 TCO per op/s improvement versus NRE increases 87
4.10.2 Optimal nodes given pre-/post- ASIC Cloud TCO 87
4.10.3 Picking the node 90
4.10.4 Advanced nodes like 16 nm not always better 91
4.10.5 Tech parity nodes: getting to *just enough* 91
4.11 Summary 91
4.12 Acknowledgments 92

Chapter 5 FPGA SmartNICs for In-Network Compute 93
5.1 Motivation and design goals 96
5.1.1 Motivating example: IDS acceleration 97
5.1.2 Design goals for Dastgāh 99
5.2 Abstraction 100
5.2.1 Modular: Software and hardware collaboration 101
5.2.2 Flexible: Reprogrammable software and hardware 102
5.2.3 Debuggable: Checking hardware using software 103
5.2.4 Memory subsystem 104
5.3 System design 106
5.3.1 Switching subsystem 106
5.3.2 Packet subsystem 108
5.3.3 Messaging subsystem 110

5.4 Implementation 111
5.5 Evaluation 115
5.5.1 Experiment setup 116
5.5.2 Forward throughput 118
5.5.3 Forward latency 118
5.5.4 Virtual interface to host 120
5.5.5 Broadcast messaging latency 121
5.5.6 Inter-core loopback throughput 121
5.6 Case studies 122
5.6.1 5-tuple hash accelerator 122
5.6.2 Regular Expression accelerator 125
5.7 Discussion 127
5.8 Summary 129
5.9 Acknowledgments 129

Chapter 6 Efficient Backhaul And Compute For Wireless Base Stations 130
6.1 Motivation 134
6.1.1 Popular bands are sparsely occupied 134
6.1.2 SDRs need smarter downsampling 135
6.2 Design 136
6.2.1 Frequency-and-time downsampling 137
6.2.2 Sparse representation of signals with STFT 139
6.2.3 Universal signal detection for STFT 141
6.2.4 Efficient reconstruction of partial STFTs 144
6.2.5 Phase offset compensation 145
6.3 Evaluation 146
6.3.1 STFT window function 146
6.3.2 STFT length 147
6.3.3 Threshold value 149
6.4 Implementation 150
6.4.1 SparSDR's downsampling hardware 150
6.4.2 SparSDR's host-based reconstruction 156
6.5 Case studies 156
6.5.1 IoT gateway 157
6.5.2 Cloud SDR 161
6.6 Summary 162
6.7 Acknowledgments 163

Chapter 7 Conclusions And Open Questions 164
7.1 Longevity 165
7.2 Open questions 167

Bibliography 169

LIST OF FIGURES

Figure 1.1: Year-after-year exponential cost reduction for Internet Transit, storage, and memory [105, 97]. 2
Figure 1.2: End of Dennard scaling, followed by end of Moore's cost scaling [129, 39]. 3

Figure 2.1: Node Technology trade-offs, normalized to 250nm. 16

Figure 3.1: IP Licensing Costs increase with advancing Tech Nodes. 37
Figure 3.2: System-level (non-ASIC) NRE varies based on PCB, Firmware and Cloud Software complexity. 39
Figure 3.3: NRE Cost Breakdown Across Tech Nodes. 40

Figure 4.1: Rising Global Bitcoin ASIC Computation and Corresponding Increase in Bitcoin ASIC Cloud Specialization. 46
Figure 4.2: High-Level Abstract Architecture of an ASIC Cloud. 47
Figure 4.3: Daily revenue, in $ per GH/s paid out by the Bitcoin network. 49
Figure 4.4: The ASIC Cloud server model. 54
Figure 4.5: ASIC Server Evaluation Flow. 55
Figure 4.6: The Delay-Voltage curve for 28-nm logic. 56
Figure 4.7: Thermal verification of an ASIC Cloud server using Computational Fluid Dynamics tools to validate the flow results. 58
Figure 4.8: Heat sink performance versus die area. 60
Figure 4.9: PCB Layout Candidates. 61
Figure 4.10: Power comparison of three PCB layouts. 62
Figure 4.11: Max power per lane for different number of ASICs in a lane. 63
Figure 4.12: Minimizing cost by optimizing the number of RCA's per ASIC Server lane (measured in mm^2), and the number of chips it should be divided into. . . . 65
Figure 4.13: Bitcoin Voltage versus Cost Performance. 67
Figure 4.14: Bitcoin Cost versus Energy Efficiency Pareto. 68
Figure 4.15: Bitcoin servers cost breakdown. 70
Figure 4.16: Litecoin Cost versus Energy Efficiency Pareto. 72
Figure 4.17: Video Transcoding Pareto Curve. 75
Figure 4.18: Video Transcoding Server cost breakdown. 76
Figure 4.19: Convolutional Neural Net Pareto Curve. 76
Figure 4.20: Breakeven point for ASIC Clouds. 80
Figure 4.21: Pareto frontiers improve in both energy and cost efficiency for newer technologies. 81
Figure 4.22: Total server cost and relation of its components change across technology for TCO-optimal servers. 86
Figure 4.23: Selecting the best tech node for Bitcoin. 87
Figure 4.24: For these applications, cutting edge node technologies become TCO-optimal only for extreme-scale ASIC Clouds. 88

Figure 4.25: Comparing marginal NRE and TCO per op/s improvements for each node. 89
Figure 4.26: Optimal node selection based on TCO (X axis) and on *tech parity node*. . . 90

Figure 5.1: Implementing hardware acceleration for a software IDS with Dastgāh only requires porting C code and implementing custom accelerators. 98
Figure 5.2: Gusheh is FPGA packet-processing hardware that contains programmable software and reconfigurable hardware. 101
Figure 5.3: The hybrid shared-memory subsystem in each Gusheh. 104
Figure 5.4: Switching subsystem overview. 107
Figure 5.5: Packet subsystem overview. 109
Figure 5.6: Layout of components in XCVU9P FPGA. 112
Figure 5.7: Layout inside of a Gusheh. 114
Figure 5.8: Experiment setup. 116
Figure 5.9: Packet generator performance on the tester FPGA. 117
Figure 5.10: Full-throttle packet forwarding performance, and maximum spare cycles available on each Gusheh. 119
Figure 5.11: Round trip latency added by Dastgāh. 120

Figure 6.1: SparSDR's goal is to make SDRs capture primary transmissions rather than entire channels. 131
Figure 6.2: A snapshot of the 2.4-GHz band collected by a OneRadio wideband (125-MHz) SDR shows that even popular unlicensed bands are sparsely occupied. 135
Figure 6.3: Distribution of occupied bandwidth over 10 microsecond intervals at 2.4 GHz. 136
Figure 6.4: Comparison of Full-capture SDRs and SparSDR architecture. 138
Figure 6.5: Frequency response of common window functions 140
Figure 6.6: An example of SparSDR's threshold-based energy detector applied to an STFT of a Bluetooth signal. 142
Figure 6.7: Compute-efficient reconstruction with partial IFFT 144
Figure 6.8: Downconverting the frequency-domain bins results in a discontinuity in the time domain. 145
Figure 6.9: Window function affects the number of bins that must be backhauled to accurately reconstruct a sinusoid. 147
Figure 6.10: Longer STFTs result in improved backhaul efficiency without degrading reconstruction fidelity. 148
Figure 6.11: Average fraction of STFT bins that were above the threshold to achieve BER of less than 10^{-5} for different SNR values. 149
Figure 6.12: Overview of SparSDR's FPGA-based downsampling pipeline. 151
Figure 6.13: SparSDR reconstructs 100-MHz captures in near real time on a Raspberry Pi. 157
Figure 6.14: IoT transmissions are sparse in time and frequency. 158
Figure 6.15: Added latency from reconstruction is significantly less than decode latency. 159
Figure 6.16: SparSDR's backhaul and compute scale linearly with the rate of received BLE advertising packets. 160

Figure 6.17: The distribution of SparSDR's backhaul for capturing transmissions in 10-MHz bands from 50 MHz to 1 GHz. 161

LIST OF TABLES

Table 2.1: Wafer and mask costs rise exponentially with process node. 17
Table 2.2: Real nominal supply voltages for each tech node. 18

Table 3.1: Mask costs and Backend cost for different process nodes. 35
Table 3.2: Node-independent NRE parameters in San Diego, CA in late 2016. 36
Table 3.3: IP Licensing Costs increase with advancing Technology Nodes. 37
Table 3.4: Application-dependent NRE parameters. 38

Table 4.1: Inputs of the server model. 57
Table 4.2: Essential parameters of an ASIC heat sink. 59
Table 4.3: Bitcoin ASIC Cloud Optimization Results. 69
Table 4.4: Litecoin ASIC Server Optimization Results. 74
Table 4.5: Video Transcoding ASIC Cloud Optimization Results. 77
Table 4.6: Convolutional Neural Network ASIC Cloud Results. 78
Table 4.7: CPU Cloud vs. GPU Cloud vs. ASIC Cloud Deathmatch. 79
Table 4.8: Bitcoin TCO-optimal ASIC Server properties across tech nodes. 82
Table 4.9: Deep Learning TCO-optimal ASIC Server properties across tech nodes. . . 83
Table 4.10: Litecoin TCO-optimal ASIC Server properties across tech nodes. 84
Table 4.11: Video Transcode TCO-optimal ASIC Server properties across tech nodes. . 85

Table 5.1: Dastgāh resource utilization, without any accelerators. 113
Table 5.2: Gusheh resource utilization 115

Table 6.1: FPGA resource requirements for different maximum FFT window sizes. . . 154
Table 6.2: Resource utilization for different modules in our USRP N210 implementation. 154
Table 6.3: Resource requirements for our SparSDR implementations, and available FPGA resources of various popular SDR models. 155
Table 6.4: Latency of the FFT module for different window sizes. 155

ACKNOWLEDGEMENTS

One does not simply get through getting a Ph.D. in the course of becoming an expert, one must understand that all valuable work stems from efforts of many. The biggest lesson to learn is one of cooperation and orchestration. Colleagues, family, and friends, all helped me throughout my endeavours. This thesis would therefore be incomplete without expressing my gratitude to all of them.

First and foremost, I owe immense gratitude to my advisor Aaron Schulman, who gave me the opportunity to conduct the research in an extremely free environment, and taught me how to better articulate my ideas and insights. He was more than an adviser for me, outside the professional relationship he was more of a supportive friend who helped me through challenging times. I dedicate my deepest gratitude for his support, guidance, and encouragements.

Also I want to thank Michael Taylor who was my advisor for the first half of my Ph.D. I got most of my insights throughout my Ph.D. from the time I was working with him. He taught me how to do reliable research and tackle problems that others are not willing to.

While doing an internship at Google, I had the opportunity to work with Partha Ranganathan and Alex Ramirez. This collaboration gave me a new assessment of the problems in computer science and helped me find more fundamental problems to tackle.

During the course of my Ph.D. studies, I had the privilege of working with other scholars on different projects. I want to thank Alex Snoeren who gave me the courage to continue time-consuming open-ended projects without initial results. I want to thank Dinesh Bharadia whom I learnt a lot from about wireless communication and looking to problems from a different angel. I want to thank Kirill Levchenko who changed the course of projects with his affirmative comments. I want to thank Alberto Dainotti and Gary Cottrell who were truly supportive of my research.

I would also like to thank my committee members, Ryan Kastner, Farinaz Koushanfar, Alex Snoeren, Steven Swanson, Michael Taylor, and Geoffrey Voelker, for their advice and comments on my dissertation.

Finally, I want to thank my true friends whom none of my achievements, if any, would have been possible without their unconditional support and help throughout my Ph.D. journey. I am indebted to many friends for their genuine friendship. I can only name a few here, but they all supported me nonetheless. I want to especially thank Atieh Lotfi, Bahram Kheradmand, Shelby Thomas, Nishant Bhaskar, Alex Forencich, Alireza Khodamoradi, Hamid Tavakoli, Alex Santana, and AmirAli Abdolrashidi. I also want to thank my friends from the BSG group, Lu Zhang, Luis Vega, Michael Barrow, and Xiaochu Liu.

The material in this dissertation is based on the following publications.

Chapter 2 contains reprints of M. Khazraee, L. Zhang, L. V Gutierrez, M. B. Taylor, "Moonwalk: NRE optimization in ASIC Clouds, or, accelerators will use old silicon", *ASPLOS*, 2017. The dissertation author is the primary author of this paper.

Chapter 3 contains reprints of M. Khazraee, L. Zhang, L. V Gutierrez, M. B. Taylor, "Moonwalk: NRE optimization in ASIC Clouds, or, accelerators will use old silicon", *ASPLOS*, 2017. The dissertation author is the primary author of this paper.

Chapter 4 contains reprints of I. Magaki, M. Khazraee, L. V Gutierrez, M. B. Taylor, "ASIC clouds: specializing the datacenter", *ISCA*, 2016, and also M. Khazraee, L. Zhang, L. V Gutierrez, M. B. Taylor, "Moonwalk: NRE optimization in ASIC Clouds, or, accelerators will use old silicon", *ASPLOS*, 2017. The dissertation author is the primary author of these papers.

Chapter 5 contains reprints of M. Khazraee, A. Forencich, G. Papen, A. Snoeren, A. Schulman, "Dastgah: Software-defined FPGA SmartNICs", currently being prepared for submission for publication. The dissertation author is the primary author of this paper.

Chapter 6 contains reprints of M. Khazraee, Y. Guddeti, S.Crow, A. Snoeren, K. Levchenko, D. Bharadia, A. Schulman, "SparSDR: Sparsity-proportional Wideband SDRs", *Mobisys*, 2019. The dissertation author is the primary author of this paper.

VITA

2013	Bachelor in Electrical Engineering, Sharif University of Technology
2016	Masters in Computer Science, University of California San Diego
2017	Intern, Google, Sunnyvale, California
2020	Doctor of Philosophy in Computer Science (Computer Engineering), University of California San Diego

- M. Khazraee, Y. Guddeti, S.Crow, A. Snoeren, K. Levchenko, D. Bharadia, A. Schulman, "SparSDR: Sparsity-proportional Wideband SDRs", *Mobisys*, 2019.
- Y. Guddeti, R. Subbaraman, M. Khazraee, A. Schulman, D. Bharadia, "SweepSense: Sensing 5 GHz in 5 Milliseconds with Low-cost SDRs", *NSDI*, 2019.
- S. Xie, S. Davidson, I. Magaki, M. Khazraee, L. Vega, L. Zhang, M. B. Taylor, "Extreme Datacenter Specialization for Planet-Scale Computing: ASIC Clouds", *ACM SIGOPS Operating Systems Review, Special Topics*, 2018.
- M. Khazraee, L. V Gutierrez, I. Magaki, M. B. Taylor, "Specializing a planet's computation: ASIC Clouds", *IEEE Micro top picks*, 2017.
- M. Khazraee, L. Zhang, L. V Gutierrez, M. B. Taylor, "Moonwalk: NRE optimization in ASIC Clouds, or, accelerators will use old silicon", *ASPLOS*, 2017.
- I. Magaki, M. Khazraee, L. V Gutierrez, M. B. Taylor, "ASIC clouds: specializing the datacenter", *ISCA*, 2016.
- M. Khazraee, "Promises and Perils in 3D Architecture", *UCSD*, 2016.

ABSTRACT OF THE DISSERTATION

Reducing the development cost of customized hardware acceleration for cloud infrastructure

by

Moein Khazraee

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2020

Professor Aaron Schulman, Chair

Customized hardware accelerators have made it possible to meet increasing workload demands in cloud computing by customizing the hardware to a specific application. They are needed because the cost and energy efficiency of general-purpose processors has plateaued. However, creating a custom hardware accelerator for an application takes several months for development and requires upfront development costs in the order of millions of dollars. These constraints have limited their use to applications that have sufficient maturity and scale to justify a large upfront investment. For instance, Google uses customized hardware accelerators to process voice searches for half a billion Google Assistant customers, and Microsoft uses

programmable customized hardware accelerators to answer queries for ~100 million Bing search users. Reducing development costs makes it possible to use hardware accelerators on applications that have moderate scale or change over time.

In this dissertation, I demonstrate that it is feasible to reduce the development costs of custom hardware accelerators in cloud infrastructure. Specifically, the following three frameworks reduce development cost for the three main parts of the cloud infrastructure. For computation inside data centers, I built a bottom-up framework that considers different design parameters of fully customized chips and servers to find the optimal total cost solution. This solution balances operational, fixed and development costs. Counter-intuitively, I demonstrate that older silicon technology nodes can provide better cost efficiency for moderate applications. For in-network computations, I built a framework that reduces development cost by offloading the control portion of an application-specific hardware accelerator to modest processors inside programmable customized hardware. I demonstrate that this framework can achieve throughput of ~200 Gbps for the compute-intensive task of deep packet inspection. For base stations at the cloud edge, I built a flexible framework on top of software-defined radios which significantly reduces their required computation performance and bandwidth. I show that it is possible to backhaul the entire 100 MHz of the 2.4 GHz ISM band over only 224 Mbps instead of 3.2 Gbps; making it possible to decode BLE packets in software with requirement of a wimpy embedded processor.

Chapter 1

Introduction

Economics plays a critical role in cloud computing. In 1997, Professor Ramnath K. Chellappa defined Cloud Computing as "the new computing paradigm, where the boundaries of computing will be determined by economic rationale, rather than technical limits alone [29]". The two main components of cloud computing are a centralized compute at datacenters, alongside the Internet network infrastructure to transfer data between the datacenters and the users. Centralization allows cloud providers to benefit from economies of scale to reduce Total Cost of Ownership (TCO) for a given compute-intensive service. Specifically, centralized services can share land, cooling and power infrastructure, networking, and administration costs. Centralization also makes it possible to transparently upgrade services to more cost efficient infrastructure when it becomes available. Namely, adapting new technology has led to significant reductions in the cost of storage, memory, networking, and compute [105, 97, 129], shown in Figure 1.1 and Figure 1.2. Cost reductions from upgrades have made it possible for cloud providers to add new services [22] and increase the number of users they serve [56] without significantly increasing their TCO.

Although cloud providers are still benefiting from economies of scale, the benefits of upgrading to more cost-efficient infrastructure have diminished. This is because the historical

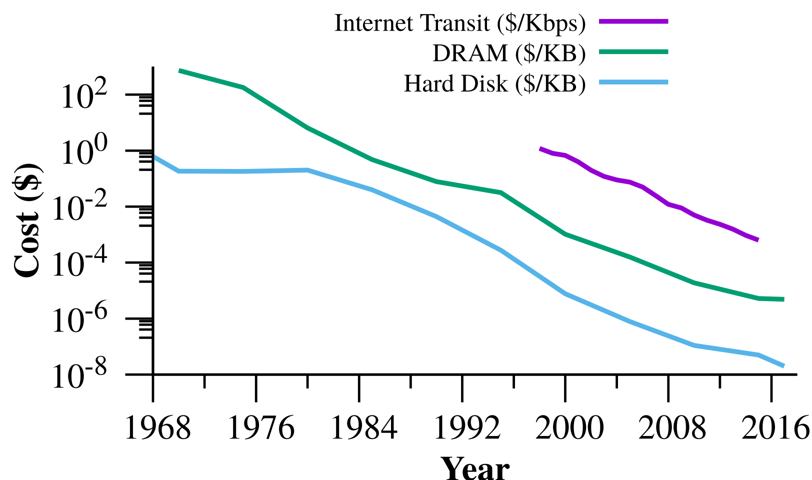


Figure 1.1: Year-after-year exponential cost reduction for Internet Transit, storage, and memory [105, 97].

year-over-year decline in the cost of general-purpose computing has ended. Specifically, in the early 2000‘s Dennard scaling ended, and soon after, Moore's cost scaling ended [39]. Figure 1.2 shows the end of Dennard scaling by change in operating frequency and safe thermal budget after the 2000's. Also it shows while the number of transistors is still following Moore's prediction, the number of transistors that could be bought for the same cost stopped increasing, and eventually started to decrease in mid-2010s. The end of these trends resulted in cloud providers being unable to use general-purpose processors to add new services or scale up existing services. For example, in 2013, Google estimated that they could not use general-purpose processors for a new voice search service because they would have had to double their datacenter infrastructure [72]. The end of these trends also limited how much cloud providers could scale in-network compute (e.g., middle-boxes) to support new faster network infrastructure in charge of improving efficiency and security. Network link speeds of 100 Gbps are already in use, 400 Gbps networks are currently being deployed and in the near future, terabit per second link speeds will become available [11]. Also new technologies such as 5G are being rolled out for cellular networks at the edge of cloud infrastructure. For example, Verizon reported 5G cellular speeds of 713 Mbps to 1.07 Gbps instead of average 53.3 Mbps speeds available for 4G [95]. As a result, cloud providers could not effectively benefit from reductions in network cost and improvements in link bandwidth.

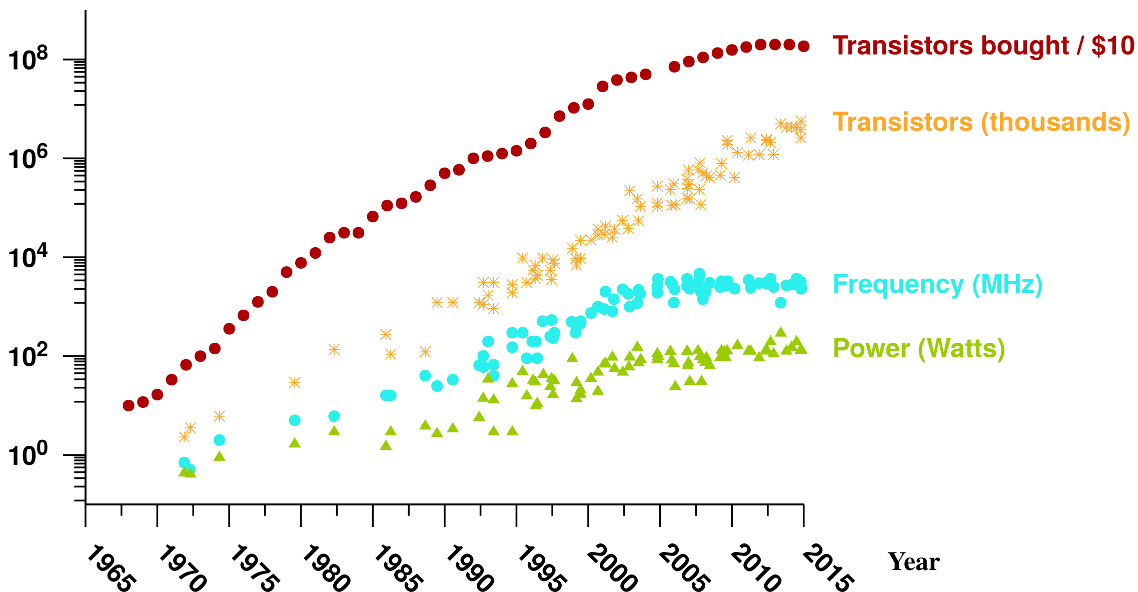


Figure 1.2: End of Dennard scaling, followed by end of Moore's cost scaling [129, 39].

It is well-known that one method to address compute bottleneck is to deploy application specific hardware accelerators. Indeed, by tailoring hardware for an application, customized hardware accelerators can provide an order-of-magnitude improvement in both compute performance and energy efficiency as compared to general-purpose processors. Customized hardware accelerators have been deployed in limited scenarios throughout the cloud infrastructure to satisfy the throughput and latency demands, which could not be met using general-purpose processors. For example, Field Programmable Gate Arrays (FPGAs) are used in wireless base stations to handle compute-intensive digital signal processing tasks. Also, fully customized Application-Specific Integrated Circuits (ASICs) are used for fixed-function in-network compute tasks, such as network firewalls, that can have high network bandwidth (e.g., 100 Gbps). In datacenters, customized hardware accelerators have been deployed to make large-scale compute-intensive services more efficient. For instance, Google has used ASICs for their voice search service in 2015. They reported that ASICs are 15–30× more performant, and 30–80× more energy efficient than running the same service on general-purpose processors [72]. Also, from 2015 Microsoft has deployed FPGAs at scale for compute-intensive applications such as the Bing

search engine [117, 28, 43, 108].

The main reason for the limited adoption of hardware accelerators in cloud infrastructure is that their upfront development costs are on the order of millions of dollars. These development costs are known as Non-Recurring Engineering (NRE) cost. NRE consists of two main components: design and manufacturing. The design phase, also known as front-end development, is where hardware and software engineers collaborate with each other to convert the application from software that runs on a general-purpose processor to a custom hardware design. Hardware development introduces several new constraints that must be met, such as placement, routing and timing. Finding a design that satisfies these constraints significantly increases total development time, compared to implementing the same functionality in software. For example, when Microsoft used FPGAs for the Bing search engine, the design process took 20 PhD holders several years to complete [117, 98]. Furthermore, designing ASIC chips requires additional costs, including back-end design (transistor placement and routing), silicon mask manufacturing, chip packaging, board development, and Intellectual Property (IP). Compared to software, these additional costs add several million dollars to the total development cost, and they can extend overall development time to more than a year.

These high development costs and long turnaround times have limited use of custom hardware accelerators in cloud infrastructure. Even cloud providers that can afford the high upfront cost, limit their use to few services that have enough demand and profit to justify the investment in development cost. These services also must have reached a stable state to accommodate the long turnaround times associated with hardware development. These challenges makes customized hardware accelerators systems potentially undesirable for applications in cloud infrastructure that needs to be updated regularly, such as intrusion detection. They also cannot be used to prototype new ideas, for instance, they cannot be used in base stations to test new wireless protocols that are under active development. Today, both of these tasks are run on inefficient general-purpose processors. In summary, new and dynamic tasks in cloud computing cannot benefit from the

performance and energy efficiency offered by hardware accelerators; unfortunately, companies cannot justify the development cost, until the service has reached massive scale and sufficiently stable state.

In this dissertation I will evaluate how customized hardware development costs can be reduced, or traded-off with operational costs, to lower the total cost of using hardware accelerators in the cloud infrastructure. In the rest of this chapter, I will first explain the challenges involved with reducing the hardware accelerator development costs in each of the three primary components of cloud infrastructure. Then, I will describe my insights which enable us to overcome all of these challenges. Next, I will present my thesis statement, and finally I will summarize my dissertation's organization, and contributions.

1.1 Challenges

Although centralization simplifies upgrading to highly efficient custom hardware for large cloud services, their multifaceted and ill-defined development cost limits their use in practice. There are two components to this cost: (1) Design (or front-end development) costs, i.e., writing code in Hardware Description Language (HDL), which requires many developers and significant time, (2) Manufacturing (or back-end development) costs, to layout the design on silicon and to manufacture fabrication masks. FPGA manufacturing companies have already paid for manufacturing costs, in exchange for more expensive chips and less customization capabilities for the cloud providers. On the other hand, for ASIC development, manufacturing costs are the dominant factor, as the cost of the mask alone is in order of millions of dollars. Moreover, different silicon technologies have different manufacturing costs, as well as different cost and energy efficiency characteristics. Unfortunately, many of the price information is proprietary and only internally available to manufacturing companies, or is in the form of agreements between manufacturing companies and vendors. **Understanding and reducing development costs for**

ASICs without a comprehensive model is not feasible.

Furthermore, even with a model of prices for ASIC development, we need a connection between development costs and TCO. Full customization leads to an excessive number of options and knobs that impact the efficiency of a customized hardware system, and hence its TCO. In addition to several constraints for designing an efficient hardware accelerator, there are several system-level design options that must be selected: such as the number of accelerators per chip, operating voltage, and the number of connected memory units. Customized chips also need a hosting board designed specifically for them. The placement of these chips, the power supply chain, the method of cooling, and heat sink design should also be jointly optimized with the rest of the design. Note that the challenge of board design also applies to FPGAs, but most of the configuration knobs are fixed, such as available resources and cost per chip, and there is no manufacturing sunk cost. **Excessive number of inter-dependent knobs that impact the efficiency of an ASIC solution, and therefore its TCO, demand for a framework to incorporate chip and system level costs, as well as development costs, to accurately calculate TCO of the system.**

For new and changing in-network computations, we cannot use customized ASICs. We must have a platform that has some of the flexibility of general-purpose processors. Programmable FPGAs provide a middle ground, however, FPGAs are much more constrained than fully customized ASICs. They have limited computation and routing resources, and also run at considerably lower clock frequencies. For example, hardware developers have to further parallelize and pipeline their implementation beyond what is needed for fully customized ASICs to meet the high performance and throughput requirements of in-network computations. Orchestration of these units with limited routing and placement options incurs significant development time. **Although FPGAs do not have the chip manufacturing costs of fully customized ASICs, the heavily constrained design process for FPGAs requires high development costs.**

Moreover, maintenance of an application in FPGAs is not as flexible as in software. As

an application is fully converted from software to customized hardware, any changes to the application will either require changes to the functional units, or changes to the flow of the processing. This forces the hardware developer to re-orchestrate the entire flow among the hardware modules to maintain the same performance. The primary solution to bridge the gap between software and hardware developers is using High-Level Synthesis (HLS) to write hardware code in a software-like language. HLS simplifies hardware design to some extent, but it is still a hardware development language and the developer needs to be fully aware of hardware restrictions and dependencies among different hardware modules. Additionally, designs in HLS usually achieve significantly lower performance and throughput than designs in hardware description languages [86], diminishing TCO improvements from hardware acceleration. **Even though FPGAs offer reprogrammability, any change to an application can result in time-consuming and costly development processes, whereas several networking applications require rapid updates.**

Operating at the cloud edge introduces new challenges that make hardware development costly. Cellular wireless communication has latency constraints, and high throughput requirements. Alongside limited general-purpose processing capability per base station, this demands for carefully optimized signal processing performed on customized hardware accelerators. Moreover, Operators want to deploy new radio protocols, such as the new narrow-band Internet of Things (IoT) protocols [133]; however, these deployments require upgrading hundreds of thousands of base stations [1]. Adding these protocols by deploying a new ASIC chip to each of the base stations is not practical. Fortunately, most base stations are already equipped with FPGAs that can be updated. **However, there are many variants of FPGAs, and few FPGA resources per base station, which limits the number of protocols that can be deployed.**

One solution to this problem is to make base stations act as Software Defined Radios (SDRs). For instance, in 5G networks, Cloud Radio Access Networks (CRAN) send raw radio

[1] 300,000 base stations only in the United States [106].

sample data to datacenters to perform the compute-intensive task of decoding in software at a central location [55, 156]. This method enables developers to update protocol decoding by changing the software, eliminating the need to pay high hardware development costs. However, SDRs require high bandwidths to backhual the samples to the datacenter. For instance, typical 4 bands of 25 MHz of radio frequency for cellular networks require 3.2 Gbps of bandwidth for raw data. Not only providing those high-bandwidth links to each individual base station further increases the TCO, it is also impractical for many base stations. Additionally, the computation costs of decoding inside datacenters on general-purpose processors is inefficient. Even though SDRs promise universality for wireless protocols, as well as significant reductions in hardware development costs, they require high backhaul bandwidths, in addition to high compute demands in datacenters, which renders them infeasible for many base stations and can potentially end up increasing TCO rather than decreasing it.

In summary, the development cost for customized hardware accelerators is difficult to reduce because: (1) the large number of inter-dependent knobs that impact the efficiency of a fully customized ASIC design, alongside uncertain development costs, makes it infeasible to accurately determine the TCO of an ASIC solution and share of the development costs, let alone to reduce the development costs for a target TCO; (2) high throughput requirements of networking applications, rapid changes to agile application, and constrained design process for FPGAs incur high and continuous development costs; (3) latency constraints, high throughput requirements, and the limited backhaul bandwidth to each base station require carefully optimized signal processing performed on customized hardware accelerators in the base stations. Additionally, limited FPGA resources per base station limit the number of wireless protocols that can be deployed at each base station.

1.2 Insights

In this dissertation, I demonstrate that ***we can reduce development cost by trading the excess of performance and energy efficiency of accelerators for reductions in NRE, while still reducing TCO***. For instance, if the current estimated cost of running a cloud workload is 10 million dollars, the difference between a 10× and 11× improvement in TCO is only $90k after the initial saving of $9M from the 10× improvement. In this instance, the increase in NRE required to get an 11× TCO improvement rather than a 10× improvement must be less than $91k. Based on this insight I demonstrate opportunities to reduce development cost without increasing TCO. I show that this insight can be applied to all three primary components of cloud infrastructure.

Insight 1: The silicon process technology node can be used to trade-off NRE for TCO efficiency in ASIC-based hardware accelerators in datacenters. Based on the NRE model that I present in this dissertation, I show that using the most advanced silicon technology incurs an exponentially higher NRE cost compared to older technology, while only offering marginal TCO efficiency improvements. I apply this insight to lower the total cost of a compute-intensive service by using older process technology nodes, making it possible for applications with moderate workloads to also see a TCO benefit from hardware acceleration.

Insight 2: Due to the inherent parallelizable packet workloads in networking, we can bring software-like flexibility and modularity to FPGAs for in-network compute. Parallelization can be achieved among functional units by incorporating packet schedulers to distribute and aggregate the packets among functional units. This enables us to distribute the load among a pipeline of low-throughput FPGA-based accelerators. Lowering processing requirements per pipeline opens up the opportunity to offload control and application semantics from hardware to software running on moderate embedded processors. This separation has marginal impact on the efficiency of the accelerator, since the majority of the computation and efficiency improvements

come from the hardware-accelerated functional units and not the control portion of application. This separation allows for hardware accelerator reuse and partial application updates in software. The added flexibility and reusability of this technique can significantly reduce the front-end development cost for FPGAs, and also enables software-like application maintenance.

Insight 3: SDR network bandwidth and compute requirements can be reduced by adding a hardware accelerator at the base station that filters signal by sparsity in time and frequency domains. Radio signals are known to be sparse both in time and frequency domains [96, 67, 112]. Therefore, we can lower the Internet transit bandwidth for SDR backhaul if the SDR hardware only backhauls active signals. This technique, where the signal detection and extraction is offloaded to the radio hardware, maintains the universality of SDR systems, because they still backhaul raw signals, but only a useful and informative fraction of it. Moreover, it will reduce signal processing computation to decode the radio signal on general-purpose processors. Even though this system is not as efficient as using hardware accelerators to decode the radio signals in the base stations, it requires significantly less hardware development cost, without requiring a high backhaul bandwidth over the Internet and high computation demands in datacenters.

1.3 Thesis statement

In this dissertation, I present methods to reduce development costs for hardware accelerators, in different parts of the cloud infrastructure. First, I build a comprehensive model for ASIC NRE costs, which enables a hardware developer to evaluate the total cost of their system. Next, I develop a methodology to determine the TCO, and find the TCO optimal solution for an ASIC based computation inside datacenters. This methodology balances efficiency and development cost, and further reduces the development cost by removing the design burden for new hardware accelerated servers.

Next, I reduce the FPGA development cost for in-network compute. I show that it is possible to separate hardware functional units and software control in FPGAs, instead of fully converting software into hardware.

Finally, I reduce the development cost for base stations in the cloud edge by significantly lowering the operational cost barrier for SDR systems. I develop an accelerator and accompanying software that can significantly reduce the required bandwidth for backhauling sparse transmission, while simultaneously reducing the signal processing compute demand.

In summary, the thesis statement that I defend in this dissertation is as follows:

"It is feasible to reduce the development costs of custom hardware accelerators in the cloud infrastructure by developing the following frameworks: 1) A framework for ASIC-based compute in datacenters which balances TCO and NRE costs and finds the optimal total cost solution, 2) A framework for FPGA-based network middle-boxes where control portion of compute is offloaded to the software running on wimpy general-purpose processors, and 3) A framework for FPGA capable base stations where backhaul and compute of software defined radios are proportional to signal sparsity."

1.4 Contributions and dissertation organization

Based on the insights mentioned earlier, I design and implement frameworks to reduce development cost in different parts of the cloud infrastructure.

Chapter 2: Background And Related Work. I describe the main components of TCO, and how development cost can be linked to TCO. Next I evaluate the trends in both ASIC and FPGA platforms, and constraints that make their development time-consuming and costly. Additionally, I describe how related work in literature addresses the development cost of custom hardware accelerators in cloud infrastructure.

Chapter 3: Building A Model For ASIC NRE Cost. I build a comprehensive model

for NRE to enable researchers to use it as a metric for their proposed accelerators.

Chapter 4: ASIC Clouds For Datacenter Compute. I build a framework to incorporate different knobs and levers for designing a datacenter from ASIC chips. Based on the TCO model from Barraso [17] and NRE model from the previous chapter, I find the optimal ASIC Cloud design based on the workload computation demand. I apply this model to four applications with different characteristics.

Chapter 5: FPGA SmartNICs For In-Network Compute. I build a framework for in-network compute on FPGAs where user can use a scheduler to distribute packets among the reconfigurable pluggable units. These units are comprised of a tiny processor, function accelerators, and a shared local memory. I show that this system can hit line rates of 200 Gbps while benefiting from the added flexibility, modularity and debuggability, which in turn significantly reduces the development cost.

Chapter 6: Efficient Backhaul And Compute For Wireless Base Stations. I build a framework for Software Defined Radios on both FPGA and software side. The FPGA side hardware accelerator performs signal detection and extraction. A lightweight process on the backeend software transforms the received data to become compatible for the conventional SDR software modules. This work significantly reduces the required backhaul and compute of SDR systems for receiving sparse radio signals.

Chapter 2

Background And Related Work

In this chapter, I provide background on why hardware accelerators reduce TCO on cloud infrastructure, and I explain how NRE is related to TCO costs for ASICs. Then I explain the advantage of FPGAs as customized hardware accelerators, and what the trade-offs between fully customized ASICs and FPGAs are. I then outline the development challenges for custom accelerator, specifically those related to their design and manufacturing. Finally I review the related work on reducing development cost of hardware accelerators in different parts of cloud infrastructure.

2.1 Economics of cloud infrastructure

Recall that the ultimate economic goal of cloud providers is to optimize the Total Cost of Ownership (TCO) for cloud services, without violating the performance and latency constraints of each service [17]. TCO consists of a combination of fixed and variable costs. Fixed costs in cloud computing are costs that are paid once to provision a certain amount of compute capability. Examples of the fixed costs include the following: computing and networking equipment, land, and infrastructure for power distribution and cooling. Variable costs in cloud computing are the costs that are proportional to the amount of computation performed to provide a cloud service.

The primary variable costs in cloud computing are personnel, electricity, and Internet Transit bandwidth. A useful term in TCO analysis is marginal cost, which is the increase in TCO to add new computational capabilities to support more users of cloud services within the same physical infrastructure. Marginal costs consist of the fixed costs for adding new equipment, in addition to the variable costs for the new computation, such as electricity and network and bandwidth.

Hardware that has lower fixed and variable costs for doing the same amount of compute is considered to be more efficient, and results in lower TCO, or only marginal cost, for the same amount of computation. For example, before the plateau of general purpose processor efficiency, every couple of years datacenter operators gradually updated their fleet of computers with processors made with newer silicon technology. They only did this when it provided sufficient improvement to cost and energy efficiency to make it financially beneficial. Customized hardware accelerators offer order-of-magnitude improvements in performance, resulting in significant cost and energy efficiency gains, and therefore reduction in marginal costs. However, the development costs for hardware accelerators, called Non-Recurring Engineering (NRE) costs, add a new fixed cost for cloud providers. Cloud providers have to balance the NRE costs and savings in marginal costs, or simply savings in TCO costs, to determine if using hardware accelerators is a financially sound decision.

2.2 Customized hardware accelerators

In this section, I describe the two methods for customized hardware acceleration. First, I discuss fully customized ASICs, and review the performance, energy efficiency and mask cost trends for silicon technology that directly impacts the TCO and NRE of fully customized ASICs. Then, I explain how performance and energy efficiency improvements of FPGAs made them much more capable and a practical platform for high performance acceleration in cloud infrastructure. Also the limitation in silicon improvements of fully customized ASICs has narrowed the gap

between FPGAs and ASICs. Finally I provide an overview of the challenges for developing hardware accelerators for both ASICs and FPGAs.

2.2.1 Silicon technology nodes

As discussed earlier, transistor technology advancement increased the number of transistors per silicon area. CMOS (Complementary metal–oxide–semiconductor) is the transistor fabrication process used for compute devices, and the main indicator of technology generation is the feature size, which shrinks with technology development. Figure 2.1 examines dominant COMS technologies from 250 nm to 16 nm. This graph is based on data we collected from four sources in order of preference: 1) using CAD tools in our lab, 2) via Internet disclosures of technical data 3) by interviewing industry experts, and 4) using CMOS scaling to interpolate missing points. Recall that in CMOS scaling, the factor S refers to the ratio of feature widths of two nodes; for example, given 180nm and 130nm, S=180/130=1.38×. Typical scaling factors between successive nodes are often assumed to be S=1.4×. Typically, transistor count increases with S^2, transistor frequency with S, and transistor capacitance (and energy per op at a fixed voltage) decreases with S.

Because of our use of historical and current data rather than predictive scaling theory, the represented nodes are different than typical scaling theory nodes, reflecting the reality of available process technology. In today's nodes, 40nm has supplanted 45nm, and 28nm has supplanted 32nm. We exclude 20nm because it has been supplanted with 16nm FinFET.

Although most of the tech node feature widths are spaced by S=1.4×, 65nm and 40nm are spaced by S=1.6×, and 28nm and 16nm are spaced by S=1.75×. Accordingly, we have plotted the data on a log-log plot with the X axis plotting feature width. Thus a straight line with slope of 1 indicates feature-width-proportional scaling. For mask costs, we have standardized on 9 metal layers if the process supports it, and otherwise the maximum number of layers for older processes (i.e. 5 layers for 250nm and 6 layers for 180nm). More metal layers entails more masks, incurring

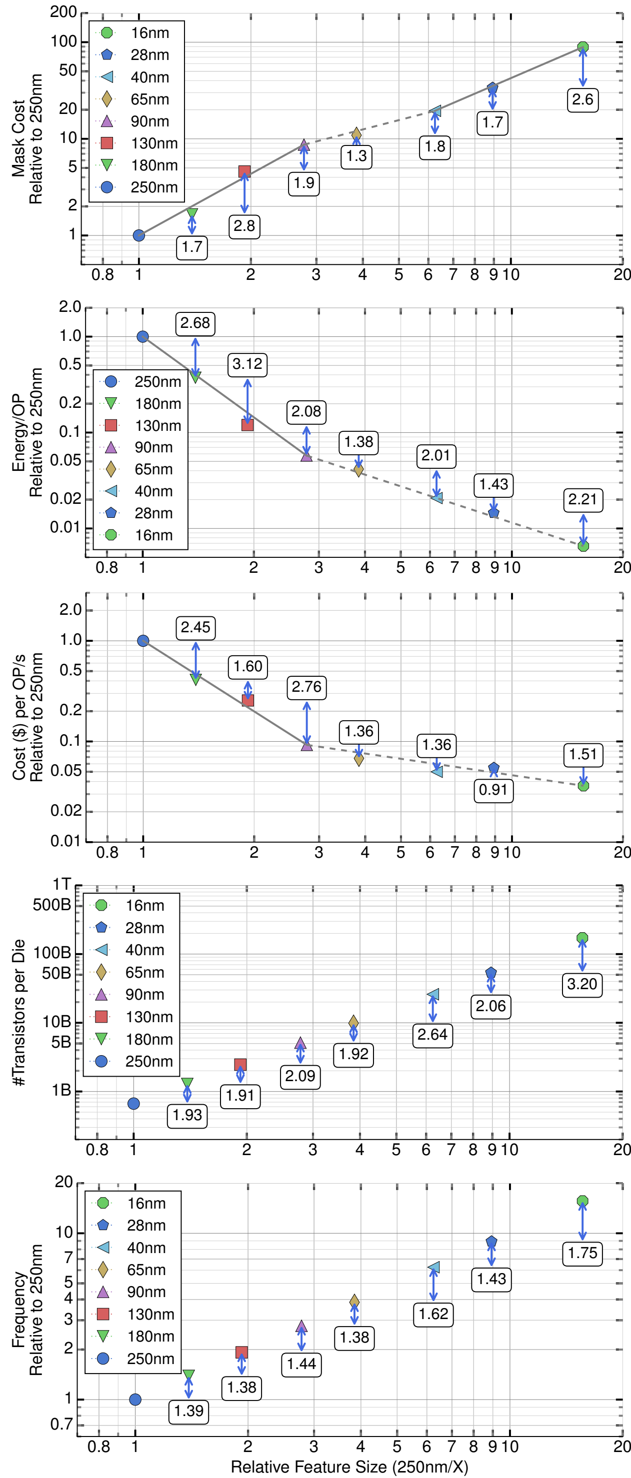


Figure 2.1: **Node Technology trade-offs, normalized to 250nm.** #'s indicate multiplicative benefits as node advance. Lines in mask cost indicate different regimes of mask cost scaling.The dotted lines in energy and cost per op/s graphs indicate the post-Dennard slowdown in voltage scaling.

more NRE.

Due to the nature of CMOS scaling, these metrics improve exponentially with more advanced process nodes. At the same time, mask NRE worsens exponentially as nodes advance. The space from 250nm to 16nm spans a **89× range in mask cost**, a **152× range in energy/op**, a **28× range in cost per op/s (558× for non-power density limited designs)**, a **256× range in maximum accelerator size** in transistors, and a **15.5× range in maximum transistor frequency**. Note that the Y axis typically spans two decades of range, but frequency is only slightly more than one decade, and transistor count spans a full three-decades.

Table 2.1: **Wafer and mask costs rise exponentially with process node.**

Tech	250nm	180nm	130nm	90nm	65nm	40nm	28nm	16nm
Mask cost ($)	65K	105K	290K	560K	700K	1.25M	2.25M	5.70M
Cost per wafer ($)	720	790	2,950	3,200	3,300	4,850	7,600	11,100
Wafer diameter (mm)	200	200	300	300	300	300	300	300

Mask costs. Figure 2.1-A and Table 2.1 show mask costs, which range from ∼65K for 250nm to almost ∼6M for 16nm. Mask cost scaling with feature width actually varies widely, as indicated by the varying slope of the segments. For example, 65nm and 40nm are particularly cheap steps, and 180nm to 130nm is a large step, relative to the previous node. Overall, mask cost multiples are smaller after 90nm than before, possibly because the number of metal layers has stabilized.

Energy per op. As can be seen in Figure 2.1-B, energy per op (e.g. CV^2) improvements are markedly different after 90nm. This coincides with the end of Dennard scaling [147] after 90nm. Prior to 90nm, energy improvements were driven by S voltage scaling and by S capacitance scaling, and in 65nm and later, they are driven by S capacitance scaling and only marginal voltage scaling (about 1.04× per node, post-Dennard scaling, as shown in Table 2.2). *Thus, given that energy per op is a major TCO driver, the benefits of nodes after 90nm are much more limited than before 90.*

Table 2.2: **Real nominal supply voltages for each tech node.**

Tech Node (nm)	250	180	130	90	65	40	28	16
Nom. V_{dd} **(V)**	2.5	1.8	1.2	1.0	1.0	0.9	0.9	0.8

Marginal cost per op/s. Figure 2.1-C graphs $ per op/s, i.e. the marginal silicon cost of adding computing capacity in a throughput-dominated workload. $ per op/s is hurt by exponentially increasing wafer costs, but helped by improvements in wafer size and ops/mm^2 compute density due to transistor frequency and density scaling. Table 2.1 shows that wafer costs scale approximately with S, but are also related to wafer size. During Dennard scaling, compute density improves as S^3, but below 90nm, it is limited to S by power density. 28nm has higher $ per op/s than 40nm because wafer cost rises faster than usable compute density improves. For applications that are not power-limited in 90nm, scaling continues more as a straight-line continuation of the pre-90nm curve, but bends towards the power-limited case at advanced nodes. In practice, some accelerators operate the logic at below-nominal Vdd levels (e.g 0.5–0.8V), in order to improve performance within the thermal budgets.

Maximum design size. Figure 2.1-D graphs the maximum number of logic transistors per die; memories are scaling less well than shown in this graph. Generally speaking, transistors per die mostly places limits on how old a process node can be used before the accelerator does not fit.

Transistor frequency. Transistor frequency improvements are graphed in Figure 2.1-E. For post-Dennard nodes that are power-density limited and do not operate at maximal clock rates, this metric still tracks the frequency of SerDes (Serializer Deserializer) in DRAM controllers and high-speed off-chip interfaces. At older nodes, frequency limits accelerator serial performance, potentially resulting in unsatisfied datacenter latency or Service Level Agreement (SLA) requirements.

Based on these trends, an unexplored opportunity for fully customized ASICs is reducing NRE by going to an older node technology. Our rich menu of available nodes means that we

have an equally rich trade-off space that links *mask cost* NREs, *energy efficiency* (i.e. joules per operation), *cost efficiency* (i.e. $ per op/s, a function of frequency, transistor count and wafer cost), *maximum transistor count* per accelerator, and *frequency* (i.e. serial performance per accelerator). Using a range of technology nodes provides a wide dynamic range of potential accelerator implementations. Nodes newer than 90nm show reduced marginal benefit in terms of the energy efficiency of the accelerator. These nodes also show reduced marginal cost benefits per unit of accelerator computation, because of post-Dennard scaling and rising wafer costs.

2.2.2 FPGAs

FPGAs offer an advantage over ASICs for NRE in that they are reprogrammable. However, this naturally results in less efficient use of silicon area and routing resources, which means, compared to ASICs, a fixed silicon area can host a smaller design and it would also run at lower frequencies. For example, consider a typical FPGA design can operate at 250 MHz, while an ASIC counter part usually can run up to 3 GHz or more. To keep up with performance and latency requirements, FPGAs require parallelizing hardware designs by making them wider or replicate the functional units, which in turn increase area utilization. Due to these limitations, initially the number of applications that could run on an FPGA were very limited and FPGAs had a very small share of market.

However, as silicon technology advanced at the rate of Moore's law, FPGAs have grown in capacity by more than a factor of 10000, as well as a factor of 100 in performance and factor of 1000 in cost and energy efficiency [146]. By mid-2000s large FPGAs were larger than the average ASIC design, and they were increasing their share of market [146]. For example, Xilinx's VU19P features 9 million system logic cells, up to 1.5 Tbps of DDR4 memory bandwidth and up to 4.5 Tbps of transceiver bandwidth, and over 2,000 user I/Os [158]. Another challenge for FPGAs was their limited on-chip memory, but as an example VU19P offers 224 Mb of on-chip memory. There are more recent FPGAs that benefit from High Memory Bandwidth (HBM) technology and

are equipped with 16 GB of low latency DRAM memory.

There were two other trends that made FPGAs desirable for customized hardware accelerators. First, the increased interest in FPGAs gave incentive to Electronic Design Automation (EDA) companies to develop software specifically for FPGA designers, which reduced the amount of engineering costs for developing hardware for FPGAs [146]. The other trend was specialization of FPGAs based on the targeted market. For example, in early 2000's over half of total FPGA business was the sales to the communications industry segment, which gave incentive to FPGA companies to customize the FPGAs for this market [146]. This resulted in adding Digital Signal Processing (DSP) units inside the FPGAs for add and multiply operations. This trend continued by integrating more and more ASIC like blocks inside the FPGA and make it more of an System on Chip (SoC) than just a logic for replicating an ASIC. To benefit from hardware-software co-design, FPGA manufacturers also added embedded general purpose processors such as an ARM inside some FPGAs. The hard blocks for managing the interface to DRAM memories or PCIe interface for communicating with a host computer became the standard in many FPGAs. New hard IP blocks were introduced for different markets, such as Ethernet MAC modules for networking systems [146], or Artificial Intelligence (AI) cores for machine learning applications [159].

Another added capability to FPGAs is Partial Reconfiguration (PR): where a user can update part of the FPGA and leave the rest of the design untouched. This feature gives the user the ability to update only part of the design without halting the system, which is valuable for updating systems that can not be shut down for updates. Another use case for partial reconfiguration is to save area by time-sharing sub-tasks of a computation: as a result a design can fit in a smaller FPGA than if all sub-tasks must be instantiated at the same time. In other words, without changing the interface to other system components, we can benefit from hardware acceleration for different portions of a program by swapping hardware in a serial manner. The main challenge for using this technique is how to save the state before the reconfiguration, and startup the new configuration with potentially a modified state, which further adds to the design and development challenge of

FPGAs.

These improvements made FPGAs sufficient for many cloud services, even for parts that require high performance computing. Initially, this improvements made FPGAs as a middle ground platform before developing an ASIC. FPGAs allowed for the ASIC developers to test their design before tape-out, instead of finding bugs in their design later and repaying part of the manufacturing cost. Also vendors could start with an FPGA, and if the application demand grow sufficiently, go towards the ASIC solution. However, FPGA improvements are narrowing the gap between FPGAs and ASICs. For instance, Microsoft started a custom hardware acceleration project called Catapult in 2010, and they are using FPGAs as the primary hardware acceleration platform in thousands of their machines. Also, Amazon recently introduced their FPGA capable machines in Amazon AWS.

2.2.3 Comparing ASICs and FPGAs

Note that although FPGAs can be reprogrammed and do not incur the manufacturing costs, it is not always the clear choice for flexible hardware accelerators. First of all, the unit cost of FPGAs is much higher than ASICs: a capable FPGA costs at least $1000 while each new ASIC chip costs about a dollar. In addition, programmablity adds overhead to FPGA cells, and there is less flexibility for routing due to predefined programmable connection maps. This gives ASICs much more flexibility in placement and routing, and they can run in much higher frequencies. For instance, a simple microprocessor made from FPGA logic can run at a maximum frequency of a few hundred mega-hertz, e.g. 300 MHz, while a much more powerful processor, such as Intel Xeon, can run at 3 GHz or more. More customization of ASICs results in more efficient use of silicon, which alongside lower unit cost makes them more energy- and cost-efficient, and overall, more TCO-optimal. Service providers need a framework to be able to calculate NRE costs and TCO savings, and based on their demand, decide whether FPGA or ASIC solution is more cost effective for them. Moreover, some applications can cope with long turnaround time of ASICs,

such as video Transcoding that have standard encoding algorithms, or Deep Neural Networks that have specific computation patterns. The main concern is if there is sufficient demand to justify the NRE cost. On the other hand, FPGAs might be the only option for several workloads that need more flexibility and agility.

2.2.4 Custom hardware acceleration development challenges

There are several reasons why hardware development is costly. First, there are many constraints that need to be met for a hardware design. For example, there are logical paths inside hardware which are made of several transistors. If these paths become too long, the time it takes for the logic output to get to a steady state increases, and hence the operating frequency is reduced. Also, even for a single direct connection, the signal going from one side of a chip to another side might take more time than the latency budget, and require the path to be broken in smaller paths. Such timing constraints as well as the limited routing resources make placement and routing of the transistors and connections among them very challenging.

Development time is also long for customized hardware, for instance translating the hardware designer's code to transistors in ASICs or logic blocks in FPGAs is done through synthesizer applications, such as Cadence Synopsys, Xilinx Vivado or Intel Quartus. The high number of constraints and complexity of the problem, a single run can take from a few hours to a few days. Moreover, result of the run might show that the constraints were not met, so the designers have to change one or several modules and run the synthesizer again. Usually it takes a couple of iterations to finally meet these constraints, and long run times per iteration significantly increase the development and debugging times.

Moreover, a minor change to functionality of the accelerator can cause the failure of these constraints and require more iterations. Even the placement of modules next to each other impacts these constraints; therefore, a design that worked for a FPGA implementation (or ASIC chip) cannot be used right away for another FPGA implementation (or ASIC chip) with

different adjacent modules. In summary, the complexity of hardware development makes it time consuming, with limited ability to reuse code.

Flexibility is an important factor for cloud computing. We need flexibility to be able to keep up with the changes in old applications and the increasing number of new applications, or to adjust with new traffic patterns and security attacks for in-network compute. Previously compiling a software code took seconds to minutes and we could deploy it right away on the current general purpose framework. For FPGA systems, assuming there is a part of the datacenter that hosts FPGA equipped servers, the deployment is comparable to general purpose solution, but the development time is in the order of days. For ASICs, we have to build new units which takes about 18 months for development, in addition to deployment time to integrate the new chips in datacenter, while they cannot be used for any other application. These long turn-around times make deployment of hardware accelerators challenging for the cloud providers.

2.3 Reducing development cost for in-datacenter compute

We have already seen signs of customization in data centers. Intel provided customized processors for cloud providers [61]. such as Xeon-D Processors specifically for Facebook which had some accelerators for their workload. Graphics Processing Units (GPU) were also repurposed and used in clouds for applications, such as Baidu and others who are building them in order to develop distributed neural network accelerators.

To provide TCO improvements, clouds have been made by using FPGAs (Field Programmable Gate Arrays). Microsoft validated and deployed FPGAs for Bing [116], JP Morgan Chase uses FPGAs for hedge-fund portfolio evaluation [134] and by almost all Wall Street firms for high-frequency trading [70]. In these cases, companies were able to ascertain that there was sufficient scale for the targeted application so that the upfront development and capital costs would be offset by a lower TCO and better computational properties.

Another example of TCO improvements achieved by hardware acceleration were clouds made from ASICs, which are built for single applications that has the required demand. The first examples of such chips were Crypto-currency miners, such as Bitcoin and Litecoin miner clouds. Google also manufactured a Tensor Processing Unit (TPU) for Deep Neural Networks which offers more TCO savings than GPUs for the inference part of the application. Although ASICs are inflexible, they are a good match for scale-out applications, meaning the target workloads consists of many independent but similar jobs (eg., the same function, but for many users, or many datasets).

However, recall that hardware acceleration comes with high development costs. There are two main components for the development costs. First we have design cost for accelerators, where the application software is transformed into hardware. This part of development is called front-end, where the design is usually ported to FPGAs for verification. Next, for ASICs, we need to map the design to transistors, which is called back-end development. Also we need to design a package to host the silicon die, and a board to host the chip, while considering power distribution and cooling constraints. Details of these costs and a more comprehensive model for hardware development costs is presented in Chapter 3.

Over the years, there have been many studies that examine the design of systems based on accelerators or application specific chips. Some of the oldest work is the GF11 [18] scale-up physics simulator. Anton [126] targeted scale-up molecular dynamics simulation. More recent work that targets accelerators for a broad spectrum of application domains, including neural networks [59, 121, 32, 76, 91, 4, 124, 34, 164, 69, 21], big data [74, 52], 3D Ultrasound [123], graph analytics [109, 57, 2], databases [81, 155], key-value stores [54, 88], natural processing language [138], regular expressions [48], speech recognition [65, 161], irregular integer applications like SpecInt [147, 148, 122], Android Hotspots [50, 51], gzip compression [1], H.264 encode [58] and convolution [118], graph processing [2], database servers [155], Web Search RankBoost [160], Machine Learning [92, 31], gzip/gunzip [1] and Big Data Analytics [74]. Tan-

don et al [138] designed accelerators for similarity measurement in natural language processing. Many efforts [54, 81, 88, 84] have examined hardware acceleration in the context of databases, key value stores and memcached. [140] examined the ramp from FPGA to GPU to ASIC Cloud-hardware for Bitcoin. Baymax [30] examined non-premptive accelerators in Clouds. Google developed Tensor Processing Unitr (TPU) [72] which is designed for Deep Neural Network model inference.

One benefit of using already available accelerators is that part of the final design has been developed. The desired accelerator might be open-sourced and used free of charge, or can be bought as an Intellectual Property (IP) to reduce the development time. Since FPGA based designs do not incur the manufacturing cost for the user, this IP reuse is one of the main methods to reduce the NRE cost. In addition, recent open source hardware efforts [9, 13, 45] propose a path to reduced IP costs. In a similar effort open source hardware libraries are developed to reduce front-end design time [35, 3, 139, 157].

Another set of recent efforts have focused on reducing cost by creating chip generators, languages and estimators [132, 12, 125] and leveraging pre-built systems for ASIC bringup [149]. Also mask NRE reduction techniques have been proposed at different levels covering manufacturing and assembling. Kim et al. proposed to build SoCs out of pre-existing libraries of custom chiplets [78]. Structured ASICs try to reduce NRE [153, 154, 163], but with significant penalties.

Another part of development cost is design of servers to benefit from the hardware accelerator chips. Previously, datacenter compute were mostly done by homogeneous general purpose computers. Therefore, even accelerators such as GPUs where introduced as addition to the general purpose computers, mainly being added over general purpose PCIe bus. Power supply chain and cooling infrastructure was also tailored for general purpose machines. Customizing such components in a server is an opportunity to improve cost and energy efficiency. For example, GPU mining rigs for Bitcoin was an early case of these optimizations. Bitcoin application running on GPUs did not use the bandwidth provided by 16x PCIe lanes and only 1x slot was sufficient.

So they used a cheap cable converter from the 16x GPU connector to the 1x mother-board slot, allowing them to benefit from more GPUs per server. They also had to get rid of the case and create rail set to to suspend the GPUs over the motherboard. This change to case also provided more surface for heat dissipation [140].

There were some similar work for datacenter-level power and thermal optimizations. Skach et al [131] proposed thermal time shifting for cooling in datacenters. Facebook optimized power and thermals in the context of general-purpose datacenters [44]. Pakbaznia et al [111] examined ILP techniques for minimizing datacenter power. Lim et al [89] examined wimpy cores, cooling, thermals and TCO. However, they are for general purpose processors and focus on dynamic processing balancing to improve cooling efficiency. There is still a need for a framework for server design for hardware accelerator based systems.

Finally, there were efforts to reduce back-end, packaging and test costs of ASIC manufacturing by developing frameworks for packaging and external connections to the pins [142, 143]. These works are complimentary to the work presented in this dissertation.

Note that none of these works evaluated use of an older process technology node to trade off NRE and TCO. Moreover, they did not do a comprehensive server level optimization for adjusting the knobs and levers of ASIC-based server development.

2.4 Reducing development cost for in-network compute

The network middle-boxes could not keep up with the increasing network line speeds. This was mostly due to the compute bottleneck, as well as some server design limitations such as PCIe (peripheral component interconnect express) bus in the general purpose computers. When line speeds got to 10 Gbps, designers added some hardware accelerators inside the Network Interface Cards (NIC) for common networking computations, especially for handling common network protocol computes such as TCP (Transmission Control Protocol) and UDP (User Datagram

Protocol).

At line speeds of 25 Gbps, offloading protocol functionalities to hardware was not sufficient and to make the system more cost- and energy-efficient while keeping the flexibility of software, designers have put several less powerful cores inside the NICs and marketed them as SmartNICs. This approach is similar to using less powerful cores inside the datacenter to improve TCO, and furthermore it skips some of the server level bottlenecks, such as the latency overhead of the PCIe bus between NIC and the main processor in Desktop and Server computers. SmartNIC manufacturers such as Mellanox, Cavium, and Broadcom integrated more and more network accelerators into their designs, such as encryption engines, or Regular Expression engines used for Intrusion Detection. However, even these SoC-based SmartNICs with several small processors and network specific accelerators are facing challenges for 40 Gbps and cannot keep up with higher speeds [42].

One of the main limitations of SmartNICs is the fact that they try to keep the flexibility of general purpose processing, and they are facing challenges such as shared memory being the bottleneck in their system. The agility of network application made it infeasible to fixate a full application is ASIC, and at most some common functional hardware accelerators can be found in SmartNICs. However, this means that only applications that use those functional accelerators really benefit from hardware acceleration speeds ups. Other applications mostly see some partial benefit from skipping the PCIe latency, in exchange for running on lower performance cores than the ones available in host processors.

The shortcoming of SmartNICs resulted in switch to FPGA accelerated solutions for networked applications. As mentioned in the subsection2.2.2, FPGAs were equipped with high speed network interfaces as well as some hard IP blocks for basic networking tasks, such as Ethernet MAC. These features made them a suitable platform for networked applications, since the network interfaces and hard IP blocks were very similar to their ASIC counterparts and they had enough resources for many in-network computations. Moreover, since the design was

not burnt into silicon, they did not require to over-provision for potential workloads, making area inefficiency of FPGAs less of a problem [42]. Therefore, custom hardware accelerators implemented on FPGAs have already been developed for wide variety network applications, including the following: web search [117], deep learning [108], compression [43], key-value stores [85], Software Defined Networking (SDN) [42, 150], congestion control [60, 87], intrusion detection [151, 100], and network debugging [6].

The main development challenge for in-networking accelerators is the high development time to convert software to hardware. FPGA hardware developers have to collaborate with software developers to convert functions in software to highly-parallelized functional units in hardware. In AzureNIC paper, Microsoft reported that "[they] have found the most important element to successfully programming FPGAs has been to have the hardware and software teams work together in one group" [42]. These challenges have limited the use of FPGA SmartNICs in production to organizations that can afford and provide specialized FPGA development teams.

One set of efforts to reduce development costs were projects such as NetFPGA [93, 166]. NetFPGA offers some premade hardware blocks for in-network computation. Moreover, there are frameworks such as NICA [40] which provides virtualisation of FPGA accelerators to user applications. These frameworks were a considerable improvement in reducing design time, but they are still difficult to use because converting software to FPGA still requires developing the entire implementation in hardware.

Another effort to make network hardware easier to program was making software programmable hardware accelerators for networking systems. For example, introducing programmable state machines that [115, 101, 19]. These state machines can capture state about a large number of flows. Or Microsoft AzureNIC [42] to enforce Software Defiend Networking (SDN) policies in their datacenter network. However, most of these efforts are bound to a specific category of applications and have very limited configurability. A similar effort was using High-Level Synthesis to convert software to hardware. But as mentioned in the introduction

chapter, this approach is still a hardware language and time consuming, while achieving lower performance and functional for a limited scope of applications, such as the ClickNP for Network Functions (NFs) [86].

Furthermore, Microsoft has already paid a very high development cost for this project but their design is proprietary, meaning other researchers outside Microsoft cannot benefit from this system. Moreover, there is very limited configuration from software for the current FPGA implementations of SmartNICs. One of the main limitations is the high PCIe latency to talk to host, which makes it infeasible for fine grained control [42].

Finally, SoC-based SmartNICs offer much finer software control due to the fact that software is running next to the accelerators. We have already seen use of such soft cores, such as Xilinx Microblaze or Intel Nios, in FPGAs to benefit from hardware software codesign. But rebuilding the SoC based SmartNICs model in FPGA results in much lower frequencies for the cores, and higher latency for caching systems. ASIC based implementation of this model could not keep up with 40 Gbps speeds and we need to rethink this model for FPGAs. In chapter 5, I will explain the new model in FPGAs to benefit from flexibility of SmartNICs as well as hardware accelerator customization of FPGAs.

2.5 Reducing development cost for wireless base stations

Due to the challenge of adding new protocols to base stations, use of SDRs are becoming a popular solution for 4G and 5G networks. Software Defined Radios (SDRs) have long promised extreme flexibility: in principle, they can capture, decode, and analyze a variety of signals across arbitrary frequency ranges without any changes to the hardware. SDRs derive their versatility from the decoupling of the radio front-end—including analog RF components and digitizers—from the signal processing back-end, which is typically deployed on general-purpose computing devices like CPUs [137], GPUs [77], or re-programmable DSPs [14]. When equipped with

wideband analog-to-digital converters (ADCs), SDRs can capture a broad range of frequencies, and support for new protocols can be added by implementing the requisite signal processing in software [145, 99].

The universality of the SDR architecture, however, comes at a price: raw samples need to be backhauled between the front-end and the signal-processing back-end. Most existing SDR implementations are inefficient in that they require backhaul capacity and processing performance in proportion to the SDR's sampling frequency, irrespective of the (typically far more limited) bandwidth of the signal being captured. For instance, a general-purpose IoT gateway needs to support frequency-hopping protocols such as Bluetooth and ZigBee that span 80 MHz of bandwidth (Fig. 6.1 left). Such an application requires 2.5 Gbps of backhaul capacity and server-class processing capability. (Although a single transmitter could be aliased into a narrower bandwidth, the full 80-MHz bandwidth is required to support simultaneous transmissions).

Crowdsourced Cloud SDRs and IoT SDRs Cloud SDR systems such as ElectroSense [120] and RadioHound [80] have emerged from the development of low-cost SDRs such as the RTL-SDR[1] and low-cost hosts such as the Raspberry Pi [79]. Researchers have also made the case for a universal IoT SDR built on such low-cost platforms [38, 102]. Research on crowdsourced SDR infrastructure has focused on giving inexpensive SDRs the capabilities of premium SDRs such as the USRP, namely full-spectrum tuning range [80, 120] and accurate frequency measurement [25]. With crowdsourced SDR infrastructure, researchers have demonstrated that by combining information from many networked SDRs they can estimate the occupancy of the spectrum [68], reveal anomalous transmissions [119], and even jointly decode wideband signals received from many narrowband sensors [24]. However, existing implementations of these applications require making a tradeoff: Either the low-cost SDRs do a significant fraction of the processing locally—only allowing a wideband SDR to be used for one application at a time—or the SDRs must backhaul raw I/Q samples, limiting their deployments to locations with well-provisioned backhaul (i.e.,

[1] https://www.rtl-sdr.com/

universities and businesses).

One approach was benefiting the fact that the RF spectrum is sparsely occupied across both time and frequency [96, 67, 112]. Techniques such as compressed sensing [37, 27] were introduced to explit the available sparsity; however, it requires *a priori* knowledge of how sparse the signal is, so it is not adaptable to dynamic conditions. Furthermore, compressed sensing requires random-sampling-based ADCs which are hard to create and require computationally intensive algorithms to reconstruct the original signal [83]. Compressed sensing has therefore been very costly and difficult to realize in real time. Similarly, sparse FFT [63] solution requires strict sparsity assumptions. In general, there is no way to ensure *a priori* that the captured spectrum will meet the sparsity constraints.

Another more recent approach is benefiting from reprogrammability of FPGAs, specifically their Partial Reconfiguration feature [94]. This approach is another method of balancing the NRE and marginal costs. Using FPGA accelerators or ASIC chips to perform the decoding significantly reduces the backhaul and back-end compute. Also by using FPGAs there is no need to change the hardware and face deployment friction. Using Partial reconfiguration allows us to choose which decoder(s) we want to use, instead of over-provisioning for every possible decode. The main drawback of this solution is increase in NRE, specially making a design capable of partial reconfiguration introduces new constraints and challenges for hardware developer. Moreover, some generality of SDRs are lost, for example we can use SDR data to monitor and detect unauthorized use of a band, but we lose the raw data in this method.

Our solution, SparSDR detailed in Chater 6, significantly reduces the backhaul requirements of wideband SDRs so they can even operate across residential-class access links as well. The main idea behind this work is to benefit from sparsity by only backhauling active signals to reduce the required backhaul and compute. There has been prior work that detected signals to take advantage of sparsity. For example, Narayanan and Kumar detect narrowband signals are over time by correlating the received samples with a pre-defined preamble sequence common

to various protocols in [102]. Their work is complementary to SparSDR in that they can detect all signals in a given narrowband in the time domain, whereas SparSDR detects signals in the frequency domain.

In contrast to SparSDR, BigBand [64] uses a complementary approach to wideband spectrum sensing and decoding. Hassanieh *et al.* alias multiple RF signals into a capture band, and reconstruct the individual signals within the aliased capture. BigBand could use SparSDR to reduce its backhaul and computation requirements at arbitrary capture bandwidths.

2.6 Summary

In this chapter, I briefly reviewed economies of cloud computing. Next, I provided more details about custom hardware accelerators and their technology trends, followed by challenges for their development and the trade-offs between fully customized and programmable hardware accelerators.

Later in this chapter, I described the related work that reduced development cost in different parts of the cloud infrastructure. I showed that the previous methods for lowering ASIC development NRE did not benefit from processing node as a knob. I described how for clouds made from ASICs we lack a framework to evaluate efficiency of different chip and system level knobs in server design to determines the TCO of the system. Next, I described the methods used for high throughput in-network compute platforms, as well as the reasoning for use of FPGAs for 100 Gbps network speeds and above. I explained the limitations of the proposed solutions, and the fact that we still incur high NRE costs in converting application software to hardware accelerators. Finally, I explained the potential of Software Defined Radios to reduce NRE cost and the fact that this solution has been blocked by SDR's high backhaul and compute requirements. I also pointed out other methods in the literature to benefit from sparsity of radio signals, but they impose new constraints to the signal, or add additional costs to back-end or front-end of SDR systems.

2.7 Acknowledgments

Chapter 2 contains reprints of M. Khazraee, L. Zhang, L. V Gutierrez, M. B. Taylor, "Moonwalk: NRE optimization in ASIC Clouds, or, accelerators will use old silicon", *ASPLOS*, 2017. The dissertation author is the primary author of this paper.

Chapter 3

Building A Model For ASIC NRE Cost

Over the years, there have been many studies that examine the design of accelerators or application specific chips. However, there is no comprehensive model of Non-Recurring Engineering (NRE) costs for hardware development. Therefore we do not know the total cost of hardware accelerator from the development to operation.

In this section, first I describe principle cost components for manufacturing, development and deployment of ASIC chips: mask, package design, labor, CAD tool, and IP costs. These costs cover both development for chips, as well as system development cost to utilize these chips. Note that I focused on ASIC development cost as it is more comprehensive than FPGA. For FPGA we only have to pay the frontend development costs and system development to mange them.

Next I describe our NRE model by applying these costs to four applications across technology nodes. Note that not only some of the NRE cost values change across technology nodes, but also some of them are application dependant. Therefore, I compute the NRE costs across different technology nodes for four applications with different characteristics, that are further detailed in Chapter 4.

3.1 Principle components of NRE cost

Mask costs. Table 3.1 show mask costs, which range from ∼65K for 250nm to almost ∼6M for 16nm, which can be a significant component of NRE, totaling as much as 90% of the NRE in advanced-node Bitcoin and Litecoin designs examined in this dissertation. However, other NRE components can be significant. On old tech nodes with many third-party IP blocks, these non-mask NRE's can total up to 95% of the cost.

Table 3.1: **Mask costs and Backend cost for different process nodes.**

Tech	250nm	180nm	130nm	90nm	65nm	40nm	28nm	16nm
Mask cost ($)	65K	105K	290K	560K	700K	1.25M	2.25M	5.70M
Backend labor cost per gate ($) [71]	0.127	0.127	0.127	0.127	0.127	0.129	0.131	0.263

Packaging costs. A common and practical package for chips is Flipchip, further discussed in Chapter 4. As an example, Flipchip package design and tooling costs contribute about $105K to NRE, shown in Table 3.2.

Labor costs. Labor costs include application-to-architecture design time, frontend development (e.g. Verilog) and testing costs, backend design and verification costs (known as Verilog-to-GDS), IP validation costs (the significant cost of adopting somebody else's IP) and non-ASIC costs like PCB design, system-level interface development, and Cloud API coding. ASIC frontend design mainly involves *IP qualification*, *design specification*, *RTL implementation*, *module integration* and *functional testing*. The backend process consumes human time in *floorplanning*, *power and clock networking*, *placement*, *routing*, *timing closure*, *design rule verification*, and a few more marginal tasks before signing-off the chip layout. System NRE includes the system level code development for interfacing the ASICs with outside world, including firmware for the server's FPGA controller, FPGA code for job distribution across ASICs, and software modifications to an existing cloud to use ASIC Cloud servers. Also, ASIC Cloud servers require a custom PCB design.

Based on our analysis, frontend labor costs do not vary much with technology node, and

Table 3.2: **Node-independent NRE parameters in San Diego, CA in late 2016.** Mm=man-month. Backend Tools are more expensive than the people using them. Flip-chip packages add significant NRE.

Frontend Labor Salary [46]	$/yr	115K
Frontend CAD Licenses	$/Mm	4K
Backend Labor Salary [46]	$/yr	95K
Backend CAD Licenses	$/month	20K
Overhead on Salary		65%
Top-level gates		15K
NRE, flip-chip BGA package	$	105K

relate more to design complexity (as measured imperfectly in lines of code, functional blocks, or gates.) For backend labor costs, costs scale with the number of unique design gates being mapped to the die, and by the complexity of the target node. Advanced nodes like 16nm that employ double-patterning suffer an additional multiplier based on greatly escalated back-end design costs. Since the ASIC Clouds employ regular arrays of accelerators on-die connected by a simple NoC (Network on Chip), we assume a hierarchical backend CAD flow that scales with RCA complexity rather than raw instance count on the die. A fixed gate count overhead is considered for I/O and NoC at the top-level of the chip. The model described in [71] gives the total backend labor cost in terms of gates, shown in Table 3.1. Frontend and backend labor salary rates as well as top-level overheads are shown in Table 3.2. 65% overhead is assumed for employee benefits and supplies.

Tool Costs. The tool costs include the frontend tools (e.g. Verilog Simulation and Synthesis), backend tools (e.g. RTL-to-GDS tools like Synopsys IC Compiler or Cadence Innovus), and PCB design tools. Of these tools, the backend tools are by far the most expensive, shown in Table 3.2. To calculate the required man-months for backend CAD tools, we divide the backend cost based on the gate count by the backend labor salary.

IP Costs. Each application's IP licensing cost depends on that application's specific IP requirements. Almost all accelerators will need standard cells (e.g. VLSI layouts for the gates, and basic LVCMOS I/O cells) and generator programs for making SRAMs. Typically, these are

Table 3.3: **IP Licensing Costs increase with advancing Technology Nodes.** Commonly used IP licensing costs across tech nodes, in late 2016, thousands of USD. Costs generally rise with node, but there are some irregularities.

Tech Node (nm)	250	180	130	90	65	40	28	16
DRAM Ctlr	NA	NA	125	125	125	125	125	125
DRAM PHY	NA	NA	150	165	175	280	390	750
PCI-E Ctlr	NA	NA	90	90	125	125	125	125
PCI-E PHY	NA	NA	160	180	325	375	510	775
PLL	15	15	15	20	30	50	35	50
LVDS IO	7.5	7.5	0	150	90	36	40	200
Standard Cells, SRAM	0	0	0	0	0	100	100	100

provided free for nodes at 65nm and older, and cost $100K or so for advanced nodes at 40nm & up. Designs that use fast (> 150 MHz) clocks need an internal PLL. For systems that use DRAM, two IP blocks are required: a DRAM controller, and a DRAM PHY, the mixed-signal block that does high-performance signaling outside the chip. Similarly, for high-speed interfaces like PCI-E or HyperTransport, a controller and PHY IP block are required. Simple applications like Bitcoin may not need any IP beyond the standard cells, while a video transcoder might require a DRAM PHY, and a neural network ASIC Cloud might require a PCI-E or HyperTransport block. These

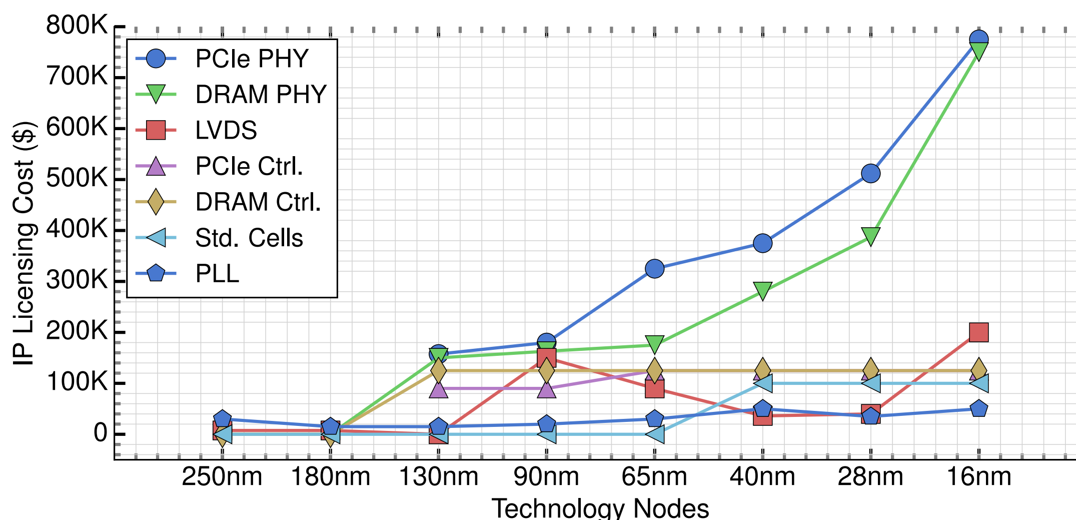


Figure 3.1: **IP Licensing Costs increase with advancing Tech Nodes.** High-speed I/O blocks rise exponentially.

IP costs greatly escalate the NRE of these accelerators. Table 3.3 shows typical IP licensing costs.

Note that IP costs rise rapidly as the technology node increases, shown in Figure 3.1, and that the most expensive IP blocks in general are PHY blocks found in PCI-E and DDRs. For 180nm and 250nm, no DDR DRAM blocks are available, and so a free SDR controller suffices. At advanced nodes like 16nm PCI-E and DDR cost almost $1M.

3.2 Modeling NRE cost

In this section we evaluate the NRE costs for each technology node. We break the NRE into two components: 1) system NRE which are the development cost to benefit from the ASIC chips, and 2) manufacturing and development cost of ASICs themselves.

Table 3.4: Application-dependent NRE parameters. PCB design costs are for late 2016. Mm = man-months, FE = Front End

Application	**Bit-coin**	**Lite-coin**	**Video Tr-anscode**	**Deep L-earning**
RCA gate count	323K	96.7K	3.56M	1.51M
FE CAD-months	8	12	23	26
FE Mm	9.5	15	24	30
FPGA job distr. code, Mm	1	1	3	2
FPGA “BIOS” code, Mm	1	1	1	1
Cloud Software, Mm	2	2	7	6
PCB Design cost ($)	37K	37K	50K	37K

Note that for both system NRE and ASIC NRE costs, the main application dependant factor of NRE is development time and hence the labor cost. To model labor cost, shown in Table 3.4, we measured our development time for Bitcoin and Litecoin frontend to calculate the required man months (Mm). We estimated the DDN front-end labor based on the paper's description and our experience designing neural network accelerators. For Video Transcode, we assumed that the company already had an internally-developed encoder IP, but had to license the

decoder for $200K.

System NRE is comprised of labor cost for system development, as well as PCB design cost. PCB design costs shown in Table 3.4 were based on vendor quotes, and the system-level coding costs came from Bitcoin mining software repository dates. FPGA firmware is estimated based on our implementation work with similar PCI-E and GigE bridges. Figure 3.2 shows the system-level, non-ASIC NRE costs for each application.

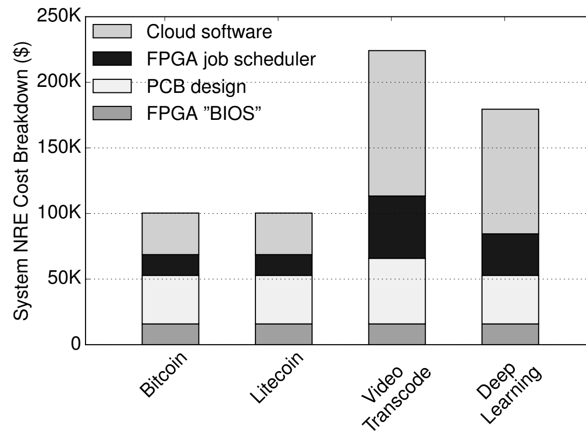


Figure 3.2: **System-level (non-ASIC) NRE varies based on PCB, Firmware and Cloud Software complexity.**

ASIC NRE can be computed based on the cost values described at the beginning of this chapter, while considering the labor time. The ASIC NRE cost breakdown across nodes and applications is shown in Figure 3.3. The trend clearly shows that the overall NRE cost rapidly increases as technology node advances and that mask costs for newer nodes become the dominant part of NRE. Labor, tool costs, IP costs and system NRE vary widely between applications but with the exception of backend labor in 16nm and PHY IP, is relatively constant across nodes. Figures 3.1 and 3.3 show how IP prices scale across tech nodes.

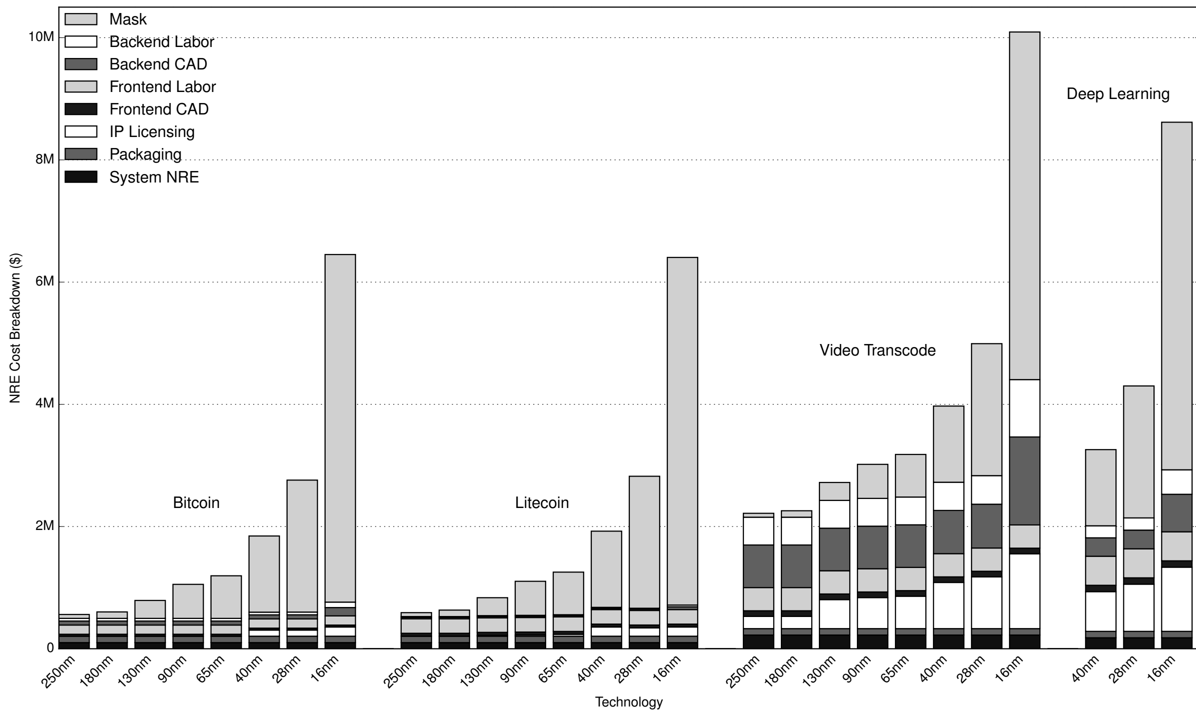


Figure 3.3: **NRE Cost Breakdown Across Tech Nodes.** Mask costs rise rapidly for newer technology nodes and become the dominant part of NRE. IP, CAD Tool and Labor Costs are application-dependent but can dominant mask costs in older nodes. Frontend labor and CAD is constant across nodes. IP costs for DDR and PCI-E/HyperTransport for newer nodes rises quickly.

3.3 Summary

In this section we presented a comprehensive NRE model for developing ASIC chips. We reviewed the principle cost components for manufacturing, development and deployment of ASIC chips, and applied these costs to four different applications across different technology nodes.

3.4 Acknowledgments

Chapter 3 contains reprints of M. Khazraee, L. Zhang, L. V Gutierrez, M. B. Taylor, “Moonwalk: NRE optimization in ASIC Clouds, or, accelerators will use old silicon”, *ASPLOS*, 2017. The dissertation author is the primary author of this paper.

Chapter 4

ASIC Clouds for Datacenter Compute

At a single node level, we know that ASICs can offer order-magnitude improvements in energy-efficiency and cost-performance over CPU, GPU, and FPGA. In this chapter, we extend this trend and study clouds made of ASIC chips. ASIC Clouds are purpose-built datacenters comprised of large arrays of ASIC accelerators, whose purpose is to optimize the TCO of large, high-volume chronic computations that are emerging in datacenters today. ASIC Clouds are not ASIC supercomputers that scale up problem sizes for a single tightly-coupled computation; rather, ASIC Clouds target workloads consisting of many independent but similar jobs (eg., the same function, but for many users, or many datasets), for which standalone accelerators have been shown to attain improvements for individual jobs.

In addition, we can specialize the ASIC Clouds in a multi-level fashion. First, at the heart of every ASIC Cloud is an ASIC design, which typically aggregates a number of accelerators into a single chip. ASIC designers can dial in exactly the optimal voltage and thermal profile for the computation. They can customize the I/O resources. Next, printed circuit board, cooling systems, power delivery system, number and type of DRAMs as well as number of I/O devices and connectors can be tailored to the application needs. Finally, ASIC Clouds can also exploit specialization at the datacenter level, optimizing rack-level and datacenter-level thermals and

power delivery that exploit the uniformity of the system.

The main challenge for building an ASIC Cloud is the high up-front non-recurring engineering (NRE) costs of developing an ASIC system. Ultimately the deployment of accelerators in the clouds must result in a net financial benefit for the underlying company. Actually ASIC Clouds already exist for a very limited set of applications. Bitcoin clouds implement the consensus algorithms in Bitcoin cryptocurrency systems. Although much is secretive in the Bitcoin mining industry, today there are 20 megawatt facilities in existence, and 40 megawatt facilities are under construction [23], and the global power budget dedicated to ASIC Clouds, large and small, is estimated by experts to be in the range of 300-500 megawatts.

In this chapter we develop an end-to-end methodology to build NRE plus TCO optimal ASIC Cloud. We first begin by examining Bitcoin mining ASIC Clouds as the first large-scale real-world case of ASIC Cloud, and distill both their unique characteristics and characteristics that are likely to apply across other ASIC Clouds. We develop the tools for designing and analyzing Pareto- and TCO- optimal ASIC Clouds. By considering ASIC Cloud chip design, server design, and finally data center design in a bottom-up way, we reveal how the designers of these novel systems can optimize the TCO in real-world ASIC Clouds. From there, we examine other ASIC Clouds designs, extending the tools for three exciting emerging cloud workloads: YouTube-style Video Transcoding, Litecoin mining and Convolutional Neural Networks.

Next, we examine when it makes sense to design and deploy an ASIC Cloud, considering NRE. Since inherently ASICs and ASIC Clouds gain their benefits from specialization, each ASIC Cloud will be specialized using its own combination of techniques. Our experience suggests that, as with much of computer architecture, many techniques are reused and re-combined in different ways to create the best solution for each ASIC Cloud. We explain how we can relate the performance and energy efficiency benefits of an accelerator to the NRE and the (pre-accelerated) TCO of the targeted workload. As described in 2, process technology node (e.g. 16nm) is one of the most important knobs for controlling the NRE cost.

Finally, we explain how we can determine the optimal technology node based on the TCO of the computation for an accelerator. We show that advanced nodes like 16nm are optimal only for a limited set of ASIC Clouds, and that nodes like 180nm, 65nm and 40nm can broaden the applicability of ASIC Clouds. We introduce the concept of a *tech parity node* which can, when combined with pre-ASIC datacenter TCO, be used to estimate the ideal target node for an application.

4.1 Bitcoin: An early ASIC cloud

In this section, we overview the underlying concepts in the Bitcoin cryptocurrency system embodied by Bitcoin ASIC Clouds. An overview of the Bitcoin cryptocurrency system and an early history of Bitcoin mining can be found in [140].

Cryptocurrency systems like Bitcoin provide a mechanism by which parties can semi-anonymously and securely transfer money between each other over the Internet. Unlike closed systems like Paypal or the VISA credit card system, these systems are open source and run in a distributed fashion across a network of untrusted machines situated all over the world. The primary mechanism that these machines implement is a global, public ledger of transactions, called the *blockchain*. This blockchain is replicated many times across the world. Periodically, every ten minutes or so, a block of new transactions is aggregated and posted to the ledger. All transactions since the beginning can be inspected[1].

Mining. A distributed consensus technique called *Byzantine Fault Tolerance* determines whose transactions are added to the blockchain, in the following way. Machines on the network request work to do from a third-party *pool server*. This work consists of performing an operation called *mining*, which is a computationally intense operation which involves brute force partial inversion of a cryptographically hard hash function like SHA256 or scrypt. The only known

[1] See `http://blockchain.info` to see real-time ledger updates.

way to perform these operations is to repeatedly try a new inputs, and run the input through the cryptographic function and see if the output has the requisite number of starting zeros. Each such attempt is a called a *hash*, and the number of hashes that a machine or group of machines performs is called its *hashrate*, which is typically quote in terms of billions of hashes per second, or *gigahash per second* (*GH/s*). When a machine succeeds, it will broadcast that it has added a block to the ledger, and the input value is the *proof of work* that it has played by the rules. The other machines on the network will examine the new block, determine if the transaction is legitimate (i.e. did somebody try to create currency, or transfer more money than was available from a particular account, or is the proof-of-work invalid), and if it is, they will use this new updated chain and attempt to post their transactions to the end of the new chain. In the infrequent case where two machines on the network have found a winning hash and broadcasted new blocks in parallel, and the chain has ”forked”, the long version has priority.

The first two ASIC Clouds analyzed in this chapter target mining for the two most dominant distributed cryptocurrencies: Bitcoin and Litecoin. People are incentivized to perform mining for three reasons. First, there is an ideological reason: the more machines that mine, the more secure the cryptocurrency network is from attacks. Second, every time a machine succeeds in posting a transaction to the blockchain, it receives a *blockchain reward* by including a payment transaction to its own account. In the case of Bitcoin, this reward is substantial: 25 bitcoins (or BTC), valued at $10,725 on the BTC-USD exchanges in April 2016. Since approximately 144 blocks are mined per day, the total value per day of mining is around $1.5M USD. Mining is the only way that currency is created in the Bitcoin system. Third, the machine also receives optional tips attached to the transaction; these tips comprise only a few percent of revenue. In order to control the rate at which new Bitcoin are created, approximately every 2016 blocks (or two weeks), the difficulty of mining is adjusted by increasing the number of leading zeros required, according to how fast the last group of 2016 blocks was solved. Thus, with slight hysteresis, the fraction of the 3600 bitcoins distributed daily that a miner receives is approximately proportional

to the ratio of their hashrate to the world-wide *network hashrate.*

Economic value of Bitcoins. Bitcoins have become increasingly valuable over time as demand increases. The value started at around \$0.07, and increased by over 15,000$\times$ to over \$1,000 in late 2013. Since then, the price has stabilized, and as of late April 2016, is around \$429, and over \$6.5 billion USD worth of BTC are in circulation today. Famously, early in the Bitcoin days, a pizza was purchased for 10,000 BTC, worth \$4.3 million USD today. The value of a BTC multiplied by yearly number of BTC mined determines in turn the yearly revenue of the entire Bitcoin mining industry, which is currently at \$563M USD per year.

4.2 Ramping the technology curve to ASIC cloud

As BTC value exponentially increased, the global amount of mining has increased greatly, and the effort and capital expended in optimizing machines to reduce TCO has also increased. This effort in turn increases the capabilities and quantity of machines that are mining today. Bitcoin ASIC Clouds have rapidly evolved through the full spectrum of specialization, from CPU to GPU, from GPU to FPGA, from FPGA to older ASIC nodes, and finally to the latest ASIC nodes. ASIC Clouds in general will follow this same evolution: rising TCO of a particular computation justifies increasingly higher expenditure of NRE's and development costs, leading to greater specialization.

Figure 4.1 shows the corresponding rise in total global network hashrate over time, normalized to the difficulty running on a few CPUs. The difficulty and hashrate have increased by an incredible factor of 50 billion since 2009, reaching approximately 575 million GH/s as of November 2015.

By scavenging data from company press releases, blogs, bitcointalk.org, and by interviewing chip designers at these companies, we have reconstructed the progression of technology in the Bitcoin mining industry, which we annotate on Figure 4.1, and describe in this section.

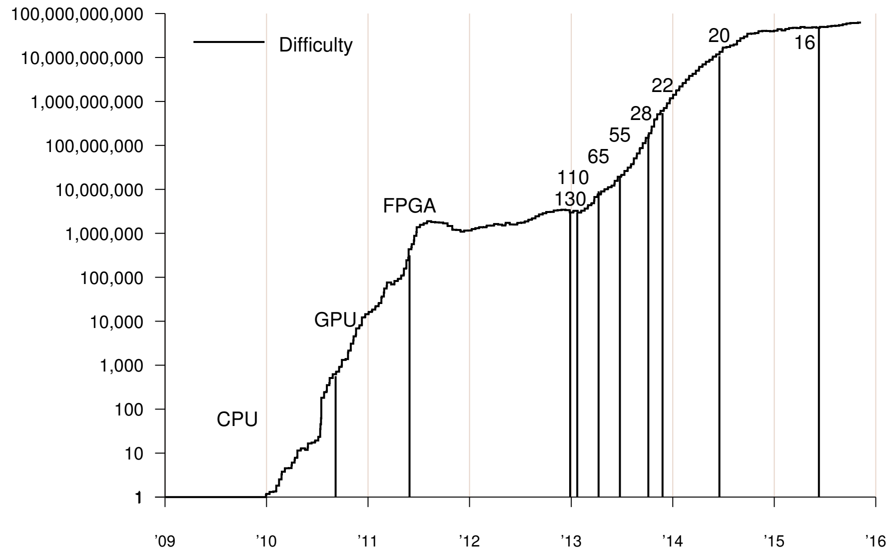


Figure 4.1: **Rising Global Bitcoin ASIC Computation and Corresponding Increase in Bitcoin ASIC Cloud Specialization.** Numbers are ASIC nodes, in nm. Difficulty is the ratio of the current world Bitcoin hash throughput relative to the initial mining network throughput, 7.15 MH/s. In the six-year period preceding Nov 2015, throughput has increased by 50 billion times, corresponding to a world hash rate of approximately 575 million GH/s. The first release date of a miner on each ASIC node is annotated.

Gen 1-3. The first generation of Bitcoin miners were CPU's, the second generation were GPU's and the third generation were FPGAs. See [140] for more details.

Gen 4. The fourth generation of Bitcoin miners started with the first ASIC (ASICMiner, standard cell, 130-nm) that was received from fab in late December 2012. Two other ASICs (Avalon, standard cell, 110-nm and Butterfly Labs, full custom, 65-nm) were concurrently developed by other teams with the first ASIC and released shortly afterwards. These first ASICs, built on older, cheaper technology nodes with low NREs, served to confirm the existence of a market for specialized Bitcoin mining hardware.

These first three ASICs had different mechanisms of deployment. ASICMiner sold shares in their firm on an online bitcoin-denominated stock exchange, and then built their own mining datacenter in China. *Thus, the first ASICs developed for Bitcoin were used to create an ASIC Cloud system.* The bitcoins mined were paid out to the investors as dividends. Because ASICMiner did not have to ship units to customers, they were the first to be able to mine and thus captured a

large fraction of the total network hash rate. Avalon and Butterfly Labs used a Kickstarter-style pre-order sales model, where revenue from the sales funded the NRE of the ASIC development. As the machines become available, they were shipped sequentially by customer order date.

Gen 5. The fifth generation of Bitcoin miners started when, upon seeing the success of the first group of ASICs, a second group of firms with greater capitalization developed and released the second wave of ASICs which used better process technology. Bitfury was the first to reach 55-nm in mid 2013 with a best-of-class full custom implementation, then Hashfast reached 28-nm in Oct. 2013, and there is evidence that 21, Inc hit the Intel 22-nm node around Dec 2013.

Gen 6. The current generation, the sixth generation of mining ASICs, is by companies that survived the second wave, and targets bleeding edge nodes as they came out (e.g. TSMC 20-nm and TSMC 16-nm). So far, these advanced nodes have only been utilized by ASIC manufacturers whose intent is to populate and run their own ASIC Clouds.

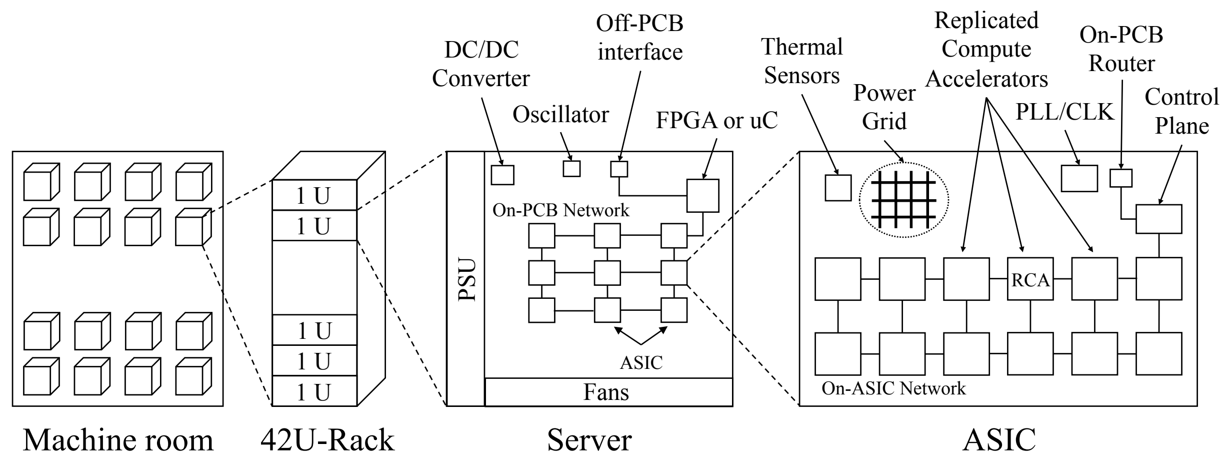


Figure 4.2: **High-Level Abstract Architecture of an ASIC Cloud.**

Moving to cloud model. Most companies that build Bitcoin mining ASICs, such as Swedish firm KnCminer, have moved away from selling hardware to end users, and instead now maintain their own private clouds [104], which are located in areas that have low-cost energy and cooling. For example KnCminer has a facility in Iceland, because there is geothermal and hydroelectric energy available at extremely low cost, and because cool air is readily available.

Bitfury created a 20 MW mining facility in the Republic of Georgia, where electricity is also cheap. Their datacenter was constructed in less than a month, and they have raised funds for a 100 MW data center in the future.

Optimizing TCO. Merged datacenter operation and ASIC development have become the industry norm for several reasons. First, the datacenter, enclosing server and the ASIC can be co-designed with fewer unknowns, eliminating the need to accommodate varying customer environments (energy cost, temperature, customs and certifications, 220V/110V, setup guides, tech support...) and enabling new kinds of optimizations that trade off cost, energy efficiency and performance. Second, ASIC Cloud bringup time is greatly shortened if the product does not have to be packaged, troubleshooted and shipped to the customer, which means that the chips can be put into use earlier. Finally, meeting an exact target for an ASIC chip is a challenging process, and tuning the system until it meets the promised specifications exactly (energy efficiency, performance) before shipping to the customer delays the deployment of the ASICs and the time at which they can start reducing TCO of the computation at hand.

4.3 Pareto- and TCO- optimality

In ASIC Clouds, two key metrics define the design space: hardware cost per performance ($ per op/s, which for Bitcoin is $ per GH/s), and energy per operation (Watts per op/s, equivalent to Joules per op, which for Bitcoin is W per GH/s). Designs can be evaluated according to these metrics, and mapped into a Pareto space that trades cost and energy efficiency. **Joint knowledge and control over datacenter and hardware design allows for the ASIC designers to select the single TCO-optimal point by correctly weighting the importance of cost per performance and energy per op among the set of Pareto-optimal points.**

In the Bitcoin space, these two metrics take on a special meaning, determining how one trades these two metrics off. W per GH/s determines the energy cost of making use of that

capacity. If we multiply this number by the ASIC Cloud's energy-related costs in $ per W day, we get $ per GH/s which is the cost of producing a GH/s for a day. If this is less than what the Bitcoin network is paying per day for a GH/s, then your ASIC Cloud is losing money each day and should be shut down as quickly as possible. If costs per GH/s is far below revenue per GH/s, profit is maximized by increasing the ASIC's voltage to produce more GH/s, even though it increases W per GH/s. Typically, since network revenue per GH/s generally decreases exponentially over time, cloud operators will start out running their machines at a higher voltage, and then gradually lower the voltage as the revenue per GH/s drops.

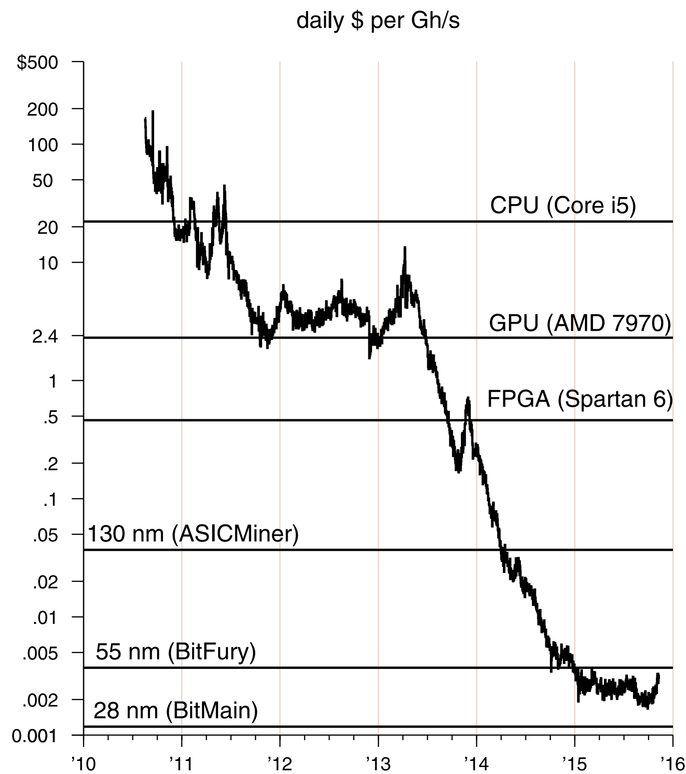


Figure 4.3: **Daily revenue, in $ per GH/s paid out by the Bitcoin network.** The horizontal lines show the point at which the energy costs (set at 20 cents per KW-hr) of each generation of accelerator exceeds the daily revenue, causing the accelerator to no longer be profitable.

Revenue per GH/s is affected by two factors. First, as the network hash rate increases, the revenue per GH/s drops. Second, if the value of a Bitcoin increases, the revenue per GH/s will increase proportionally. Over the last few years, we have seen the revenue per GH/s drop

exponentially as new technology is deployed, even as the value of Bitcoins rise. In Figure 4.3, we graph revenue per GH/s over time. We also annotate the graph with the points at which time each technology starts to lose money, assuming an energy cost of 20 cents per KW hour. Correlating against the release times of each technology in Figure 4.1, we can see that GPUs were profitable for almost 2.75 years, while FPGA was profitable for 2.25 years, 130-nm ASIC was profitable for 1.25 years, the BitFury 55-nm ASIC was profitable for 1.5 years, and 28-nm ASIC has been competitive for 2.25 years and is still going strong. However, the 28-nm ASIC has a very small gap between revenue and cost per GH/s and thus they must be operated at a very low voltage to maintain profitability.

Given these expectations about profitability, the hardware cost in $ per GH/s, determines the breakeven period of the hardware assuming an average $ per GH/s during the positive profit lifetime of the machine.

In more conventional ASIC Clouds, where the primary objective may be to save energy costs versus an non-ASIC Cloud providing the same service, $ per op and W per op can be used with other inputs in a total cost of ownership (TCO) equation to determine the number of days before savings are realized versus an alternative data center implementation based on CPU or GPU technology.

4.4 Architecture of an ASIC Cloud

We starting by examining the design decisions that apply generally across ASIC Clouds. Later, we design four example ASIC Cloud for Bitcoin, Litecoin, Video Transcoding, and Convolutional Neural Networks. We can specialize the ASIC Clouds in a multi-level fashion.

First, at the heart of every ASIC Cloud is an ASIC design, which typically aggregates a number of accelerators into a single chip. ASICs achieve large reductions in silicon area and energy consumption versus CPUs, GPUs, and FPGAs because they are able to exactly

provision the required resources needed for the computation. They can replace area-intensive, energy-wasteful instruction interpreters with area-efficient, energy-efficient parallel circuits. ASIC designers can dial in exactly the optimal voltage and thermal profile for the computation. They can customize the I/O resources, instantiating precisely the right number of DRAM, PCI-e, HyperTransport and Gig-E controllers, and employ optimized packages with optimal pin allocation.

Second, we can specialize the server containing the ASICs. A typical datacenter server is encrusted with a plethora of x86/PC support chips, multi-phase voltage regulators supporting DVFS, connectors, DRAMs, and I/O devices, many of which can be stripped away for a particular application. Moreover, typical Xeon servers embody a CPU-centric design, where computation (and profit!) is concentrated in a very small area of the PCB, creating extreme hotspots. This results in heavy-weight local cooling solutions that obstruct delivery of cool air across the system, resulting in sub-optimal system-level thermal properties. ASIC Servers in Bitcoin ASIC Clouds integrate arrays of ASICs organized evenly across parallel shotgun-style airducts that use wide arrays of low-cost heatsinks to efficiently transfer heat out of the system and provide uniform thermal profiles. ASIC Cloud servers use a customized printed circuit board, specialized cooling systems and specialized power delivery systems, and can customize the DRAM type (e.g., LP-DDR3, DDR4, GDDR5, HBM...) and DRAM count for the application at hand, as well as the minimal necessary I/O devices and connectors required. Further, they employ custom voltages in order to tune TCO.

Finally, we can specialize at the ASIC Datacenter level. ASIC Clouds can also exploit specialization at the datacenter level, optimizing rack-level and datacenter-level thermals and power delivery that exploit the uniformity of the system. More importantly, cloud-level parameters (e.g., energy provisioning cost and availability, depreciation and ... taxes) are pushed down into the server and ASIC design to influence cost- and energy- efficiency of computation, producing the TCO-optimal design.

To realize this goal, we make a model of a basic ASIC Cloud architecture. At the heart of any ASIC Cloud is an energy-efficient, high-performance, specialized *replicated compute accelerator, or RCA,* that is multiplied up by having multiple copies per ASICs, multiple ASICs per server, multiple servers per rack, and multiple racks per datacenter. Work requests from outside the datacenter will be distributed across these RCAs in a scale-out fashion. All system components can be customized for the application to minimize TCO.

Figure 4.2 shows the architecture of a basic ASIC Cloud. Starting from the left, we start with the data center's machine room, which contains a number of 42U-style racks. In this work, we try to minimize our requirements for the machine room because in many cases, after an array of GPU or CPU-based machines has been replaced with a new kind of nascent ASIC Cloud, it may occupy only a tiny part of a datacenter[2], and thus have little flexibility in dictating the machine room's parameters. Accordingly, we employ a modified version of the standard warehouse scale computer model from Barroso et al [16]. We assume that the machine room is capable of providing inlet air to the racks at 30° C.

ASIC Cloud Servers. Each rack contains an array of servers. Each server contains a high-efficiency power supply (PSU), an array of inlet fans, and a customized printed circuit board (PCB) that contains an array of specialized ASICs, and a control processor (typically an FPGA or microcontroller, but also potentially a CPU) that schedules computation across the ASICs via a customized on-PCB multidrop or point-to-point interconnection network. The control processor also routes data from the off-PCB interfaces to the on-PCB network to feed the ASICs. Depending on the required bandwidth, the on-PCB network could be as simple as a 4-pin SPI interface, or it could be high-bandwidth HyperTransport, RapidIO or QPI links. Candidate off-PCB interfaces include PCI-e (like in Convey HC1 and HC2), commodity 1/10/40 GigE interfaces, and high speed point-to-point 10-20 gbps serial links like Microsoft Catapult's inter-system SL3 links. All

[2]In the case of Bitcoin, the scale of computation has been increased so greatly that the machine rooms are filled with only Bitcoin hardware and as a result are heavily customized for Bitcoin to reduce TCO, including the use of immersion cooling [75].

these interfaces enable communication between neighboring 1U modules in a 42U rack, and in many cases, across a rack and even between neighboring racks.

Since the PSU outputs 12V DC, our baseline ASIC server contains a number of DC/DC converters which serve to step current down to the 0.4-1.5 V ASIC core voltage. Finally, flip-chip designs have heat sinks on each chip, and wire-bonded QFNs have heat sinks on the PCB backside.

ASICs. Each customized ASIC contains an array of RCA's connected by an on-ASIC interconnection network, a router for the on-PCB (but off-ASIC) network, a control plane that interprets incoming packets from the on-PCB network and schedules computation and data onto the RCA's, thermal sensors, and one or more PLL or CLK generation circuits. In Figure 4.2, we show the Power Grid explicitly, because for high power density or low-voltage ASICs, it will have to be engineered explicitly for low IR drop and high current. Depending on the application, for example, our Convolutional Neural Network ASIC Cloud, the ASIC may use the on-ASIC network for high-bandwidth interfaces between the replicated compute accelerators, and the on-PCB network between chips at the 1U server level. If the RCA requires DRAM, then the ASIC contains a number of shared DRAM controllers connected to ASIC-local DRAMs. The on-PCB network is used by the PCB control processor to route data from the off-PCB interfaces to the ASICs to the DRAMs.

We examines a spectrum of ASIC Clouds with diverse needs. Bitcoin ASIC Clouds required no inter-chip or inter-RCA bandwidth, but have ultra-high power density, because they have little on-chip SRAM. Litecoin ASIC Clouds are SRAM-intensive, and have lower power density. Video Transcoding ASIC Clouds require DRAMs next to each ASIC, and high off-PCB bandwidth. Finally, our DaDianNao-style [33] Convolutional Neural Network ASIC Clouds make use of on-ASIC eDRAM and HyperTransport links between ASICs to scale to large multichip CNN accelerators.

Voltage. In addition to specialization, voltage optimization is a key factor that determines

ASIC Cloud energy efficiency and performance. We will show how the TCO-optimal voltage can be selected across ASIC Clouds.

4.5 Design of an ASIC server

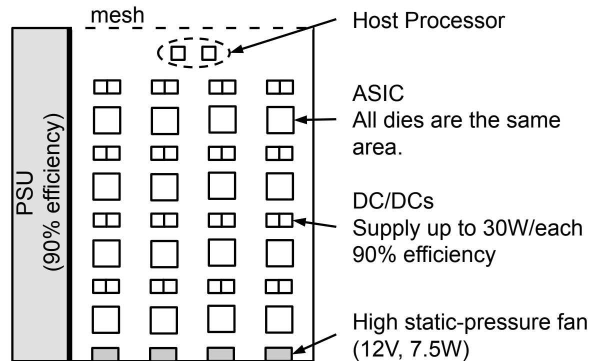


Figure 4.4: **The ASIC Cloud server model.**

In this section, we examine the general principals in ASIC Cloud Server design to find the Pareto frontier across *$ per op/s* and *W per op/s*. Using area, performance and power density metrics of an RCA we show how to optimize the ASIC Server by tuning the number of RCAs placed on each chip; the number of chips placed on the PCB; their organization on the PCB; the way the power is delivered to the ASICs; how the server is cooled; and finally the choice of voltage. Subsequent sections apply these principles to our four prototypical ASIC Clouds.

4.5.1 ASIC server overview

Figure 4.4 shows the overview of our baseline ASIC Cloud server. In our study, we focus on 1U 19-inch rackmount servers. The choice of a standardized server form factor maximizes the compatibility of the design with existing machine room infrastructures, and also allows the design to minimize energy and components cost by making use of standardized high-volume commodity server components. The same analysis described in this chapter could be applied to 2U systems

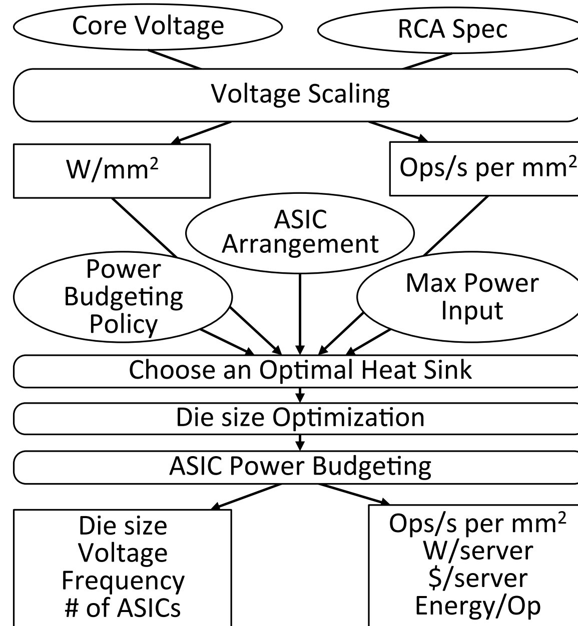


Figure 4.5: **ASIC Server Evaluation Flow.** The server cost, per server hash rate, and energy efficiency are evaluated using RCA properties, and a flow that optimizes server heat sinks, die size, voltage and power density.

as well. Notably, almost all latest-generation Bitcoin mining ASIC Cloud servers have higher maximum power density than can be sustained in a fully populated rack; so racks are generally not fully populated. Having this high density makes it easier to allocate the number of servers to a rack according to the data center's per-rack power and cooling targets without worrying about space constraints.

The servers employ forced-air cooling system for heat removal, taking cold air at 30° C from the front using a number of 1U-high fans, and exhausting the hot air from the rear. The power supply unit (PSU) is located on the leftmost side of the server, and a thin wall separates the PSU from the PCB housing. Because of this separation and its capability of cooling itself, the PSU is ignored in terms of thermal analysis in the remaining part of this section. Figure 4.4 provides the basic parameters of our thermal model.

4.5.2 ASIC server model

In order to explore the design space of ASIC Cloud Servers, we have built a comprehensive evaluation flow, shown in Figure 4.5, that takes in application parameters and a set of specifications and optimizes a system with those specs. We repeatedly run the evaluation flow across the design space in order to determine Pareto optimal points that trade off *$ per op/s* and *W per op/s*.

Given an implementation and architecture for the target RCA, VLSI tools are used to map it to the target process (in our case, fully placed and routed designs in UMC 28-nm using Synopsys IC compiler), and analysis tools (e.g. PrimeTime) provide information on frequency, performance, area and power usage, which comprise the *RCA Spec*. This information and a *core voltage* is then applied to a voltage scaling model that provides a spectrum of Pareto points connecting W per mm^2 and op/s per mm^2. From there, we compute the optimal ASIC die size, ASIC count, and heat sink configuration for the server, while ensuring that the transistors on each die stay within maximum junction temperature limits. Then, the tool outputs the optimized configuration and also the performance, energy, cost, and power metrics. Table 4.1 shows the input parameters.

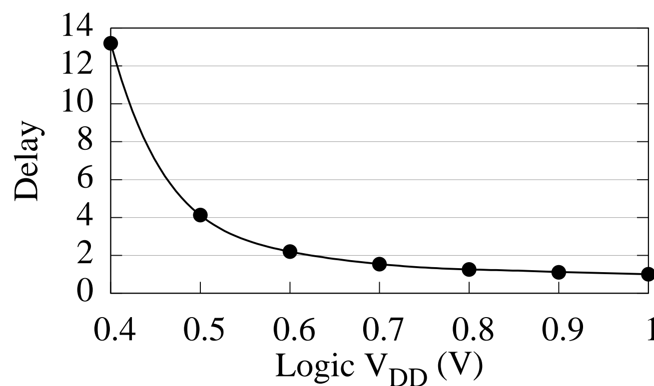


Figure 4.6: **The Delay-Voltage curve for 28-nm logic.**

To assess the effect of voltage scaling throughout the rest of this chapter, we applied a delay-voltage curve shown in Figure 4.6 to the logic part of the critical path, computing the effect on both power density (W/mm^2) and performance (ops/s per mm^2). This curve is inferred from

I-V characteristics taken from [110, 162] and normalized at 1.0 V. For SRAMs, we assume the SRAM is on a separate power rail that supports a higher minimum supply voltage than for logic, because of the challenges in downward scaling SRAM voltages. The dynamic power is evaluated by the new frequency and voltage while leakage is affected only by the voltage.

Iterative trials find the best heat sink configuration, optimizing heat sink dimensions, material and fin topology.

Table 4.1: **Inputs of the server model.**

Parameters	**Description**
Replicated Compute Accelerator	Power density (W/mm^2), Perf. density (Ops/s/mm^2), Critical paths @ nom. voltage, 1 V
Core Voltage	0.4 V–1.5 V
DC/DC Efficiency & Cost PSU 208–12V Efficiency & Cost	90 %, $0.33 per Amp 90 %, $0.13 per Watt

4.5.3 Thermally-aware ASIC server design

In this subsection, we optimize the plumbing for ASIC Cloud Servers that are comprised of arrays of ASICs. We start by describing the power delivery system and packaging. Then we optimize the heat sinks and fans for the servers, and optimally arrange the ASICs on the PCB.

Our results are computed by building physical models of each component in the server and simulating each configuration using the ANSYS Icepak version 16.1 Computational Fluid Dynamic (CFD) package. For example, Figure 4.7 shows the simulation for 4 rows with 4 ASICs in each, where the thermally bottlenecking ASIC is the one farthest to the fan. This ASIC has the highest temperature and is the bottleneck for power per ASIC at a fixed voltage and energy efficiency. Based on these configurations, we built a validated Python model. After completing our Pareto study, we resimulated the Pareto-optimal configurations to confirm calibration with

Icepak.

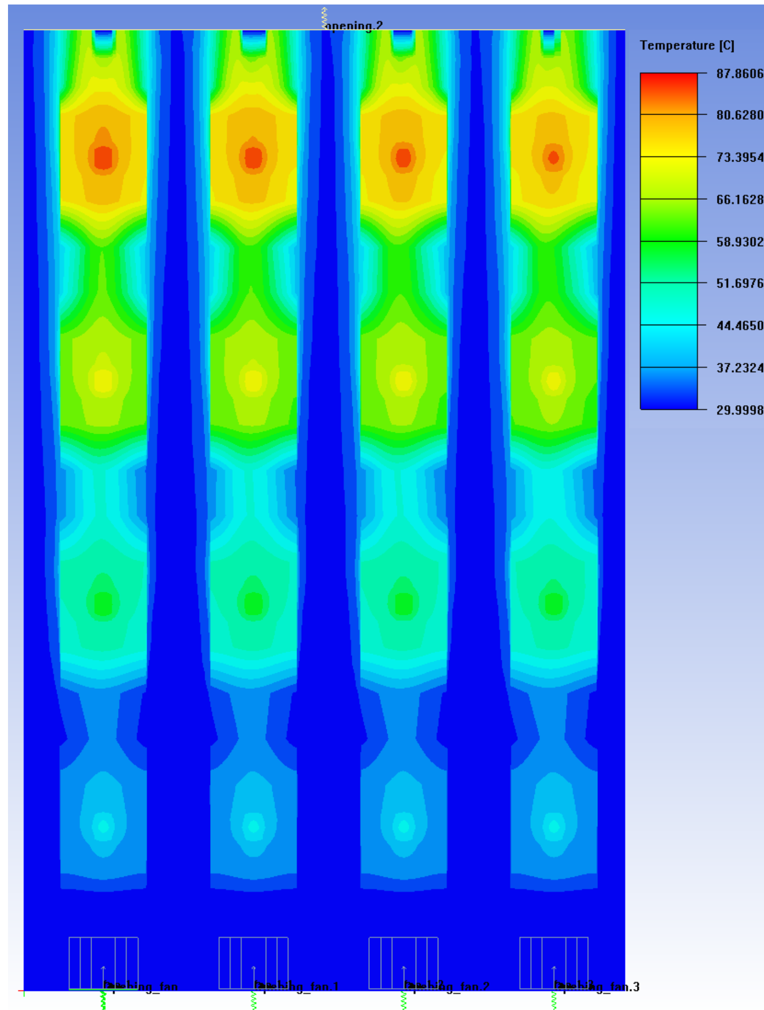


Figure 4.7: **Thermal verification of an ASIC Cloud server using Computational Fluid Dynamics tools to validate the flow results.** The farthest ASIC from the fan has the highest temperature and is the bottleneck for power per ASIC at a fixed voltage and energy efficiency.

Power delivery & ASIC packaging

Power is delivered to the ASICs via a combination of the power supply, which goes from 208V to 12V, and an array of DC/DC converters, which go from 12V to chip voltage. One DC/DC converter is required for every 30A used by the system. The ASICs and onboard DC/DC converters are the major heat sources in our ASIC server model (Figure 4.4). Host processor heat

is negligible in the ASIC Servers we consider.

We employ Flip-Chip Ball Grid Array (FC-BGA) with direct heat sink attach for ASIC packaging because of its superior thermal conductivity and power delivery performance relative to alternative wire-bond based technologies. For example, Wire-Bond BGA or QFN that have high inductance, periphery interconnect and suffer the additional thermal resistance of a FR4 PCB or BT substrate. FC-BGA also tend to be more area efficient because of the lack of a pad ring.

Table 4.2: Essential parameters of an ASIC heat sink.

Parameters	Value
Width	$\leq$ 85 mm (Limited by ASIC density)
Height	35 mm (Limited to 1U height) A heat spreader of 3 mm thick is included
Depth	$\leq$100 mm (Limited by the PCB depth)
Fin thickness	0.5 mm
# of fins	$\geq$ 1 mm between two fins
Materials	Al (200 W/mK) for fins Al or Copper (400 W/mK) for heat spreader
Air Volume	Determined by static pressure and fan curve

Optimizing the heat sink and fan

Thermal considerations have a great impact on a packaged ASIC's power budget and the server's overall performance. The heat sink's cooling performance depends not only on its dimensions and materials but also on how those ASICs and heat sinks are arranged on the PCB because of the airflow. This section looks closely into the cooling capability of a heat sink, and airflow is considered in the next section.

The ASIC's forced-air cooling system comprises a heat spreader glued to a silicon die using a thermal interface material (TIM). Fans blow cool air over fins that are welded to the heat spreader. The fins run parallel to air flow to maximize heat transfer. Table 4.2 shows the key parameters.

Heat conducts from a die to a larger surface through a TIM and a heat spreader. Increasing heat spreader size increases the surface area for better cooling, and provides more area for fins to improve the total heat resistance. Larger silicon dies can dissipate more heat since the thermal resistance induced by TIM is the dominant bottleneck because of its poor thermal conductivity and inverse proportionality to die area (Figure 4.8). However, increasing fin count and depth enlarges the pressure drop induced by the heat sink, resulting in less airflow from the cooling fans.

Commercial fans are characterized by a *fan curve* that describes how much air it can supply under a certain pressure drop due to static pressure. Our model takes as input a fan curve and ASIC count per row and evaluates the heat sink dimensions that achieve maximum power dissipation as a whole for a certain die area. According to the evaluation, the number of ASICs in a row affects the depth of each heat sink. As the number of ASICs increases, the heat sinks becomes less deep to reduce pressure drop and keep the airflow rate up. Generally, the densest packed fins are preferable.

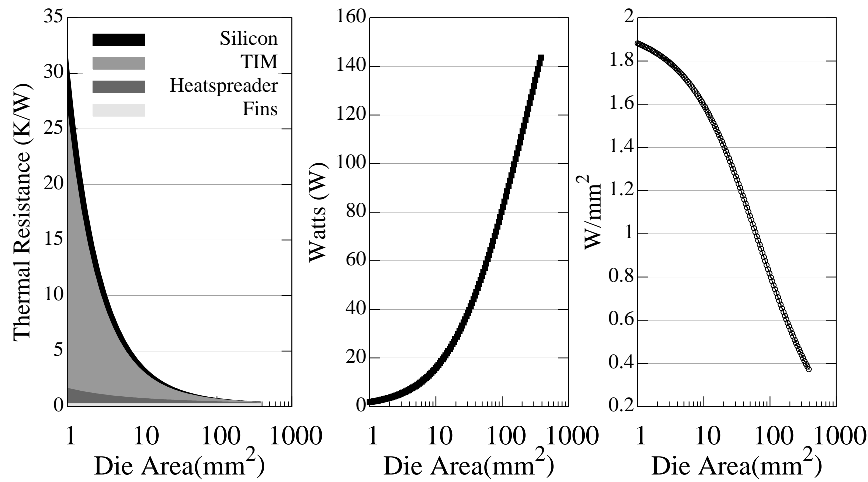


Figure 4.8: **Heat sink performance versus die area.** The thermal resistance for small dies is dominated by TIM. Bigger dies make the most out of the heat sink. However, acceptable power density decreases as the die area increases, resulting in smaller area for a higher power density design.

How should we arrange ASICs on the PCB?

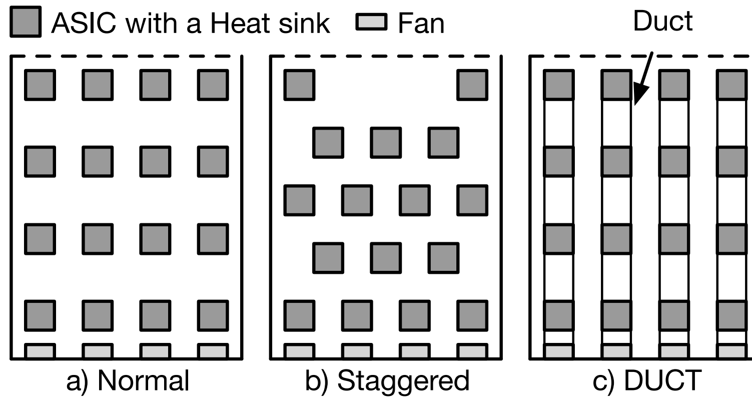


Figure 4.9: **PCB Layout Candidates.** (a) Grid pattern (b) Staggered pattern to reduce bypass airflow (c) Deploying ducts that surrounds ASICs in a column to further reduce bypass airflow (DUCT layout)

In this section, we analyze the impact of ASIC placement on the PCB for our ASIC server. The PCB layout matters when the server houses multiple high-power components with large heat sinks. Those components make the server design more challenging not only due to heat flux but also because they behave as obstacles that disturb smooth airflow coming from the fans. The result is inefficient airflow that induces a lot of bypass airflow venting out of the server without contributing to cooling. In other words, ineffective PCB layout requires more fans and fan power to remove the same amount of heat.

We examine three different PCB layouts for our ASIC server, shown in Figure 4.9. For this experiment, we assume 1.5 kW high-density servers with 16 ASICs on a PCB, employing the same number and type of fans. The ASIC is assumed to be 100 mm^2 with an optimal heat sink. All the ASICs consume the same power.

In *Normal* and *DUCT* layout, ASICs are aligned in each column, while in the *Staggered* layout, ASICs of odd and even rows have been staggered to maximally spread hot air flows. The DUCT layout has four enclosures (”ducts”) that constrain all the airflow supplied by a corresponding fan to ASICs in the same column. The fan is directly abutted to the front of the

enclosure, and hot air is exhausted out the rear.

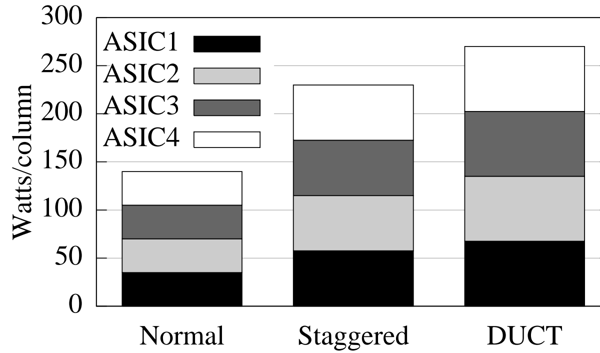


Figure 4.10: **Power comparison of three PCB layouts.** The staggered layout improves by 64 % in removable heat over the normal layout while DUCT gains an additional 15 %. DUCT is superior because it results in less bypass airflow between two ASICs. In each layout, the optimal heat sinks are attached on top of the dies. All layouts use the same fans.

Employing identical cooling fans, iterative simulations gradually increase the ASICs' power until at least some part of one die reaches the maximum junction temperature of the process, 90 °C. Figure 4.10 shows the results.

By moving from Normal layout to Staggered layout, *65 % more heat per chip* can be removed with no additional cost. The DUCT layout is even better, allowing *15 % more heat per chip* than the Staggered layout, because almost all airflow from the fans contributes to cooling. Although Staggered layout is more efficient than the Normal layout, wide temperature variation is observed in the Staggered because of uneven airflow to each ASIC and the ASIC receiving the poorest airflow would constraint the power per chip. In DUCT layout, inexpensive enclosures gain 15 % improvement in consumable power for the same cooling. Accordingly, we employ the DUCT layout in our subsequent analysis.

More chips versus fewer chips

In the design of an ASIC Server, one choice we have is, how many chips should we place on the PCB, and how large, in mm^2 of silicon, should each chip be? The size of each chip

determines how many RCAs will be on each chip.

In a uniformly distributed power layout in a lane, each chip receives almost the same amount of airflow, while the chip in downstream receives hotter air due to the heat removed from upstream ASICs. So, typically the thermally bottlenecking ASIC is the one in the back.

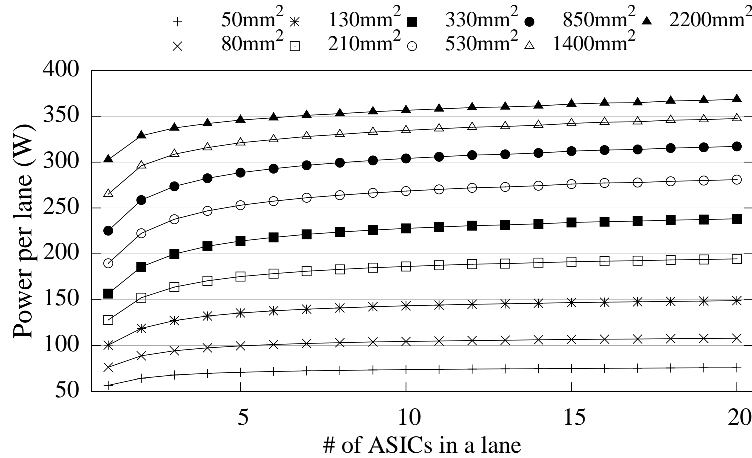


Figure 4.11: **Max power per lane for different number of ASICs in a lane.** Increasing the number of ASIC dies while keeping the total silicon area fixed increases the total allowable power, because heat generation is more evenly spread across the lane. Greater total area also increases the allowable power since there is more TIM.

Intuitively, breaking heat sources into more smaller ones spreads heat sources apart, so having many small ASIC should be easier to cool than a few larger ASICs. This intuition is verified in Figure 4.11. We hold the total silicon area used in each row to be fixed, and evaluate the influence of spreading the silicon by using more chips. In our analysis, we reduce the depths of the heat sinks as chip count increases in order to keep airflow up and maximize cooling. Additionally, as we increase the total amount of silicon area, our capacity to dissipate heat also increases. This is because the thermal interface glue between die and heat spreader is a major path of resistance to the outside air, so increasing die size reduces this resistance. Thus, placing a small total die area in a lane is ineffective, because limited surface area between die and spreader becomes the dominant bottleneck.

From a TCO point of view, increasing the number of chips increases the cost of packaging, but not by much. Using Flip Chip, the packaging cost is a function of die size because of yield

effects. Pin cost is based on the number of pins, which is set by power delivery requirements to the silicon. Our package cost model, based on input from industry veterans, suggests the per-chip assembly cost runs about $1.

4.5.4 Linking power density & ASIC server cost

Using the server thermal optimization techniques described in the previous subsection, we can now make a critical connection between an RCA's properties and the number of RCA's that we should place in an ASIC, and how many of those ASICs we should place in an ASIC Server.

After designing an RCA using VLSI tools, we can compute its power density (W/mm^2) using simulation tools like Primetime. Then, using our server thermal design scripts, we can compute the ASIC Server configuration that minimizes $ per op/s; in particular how many total RCA's should be placed in an ASIC Server lane, and also how many chips the RCAs should be divided into.

Figure 4.12 shows the results, which are representative across typical ASIC Cloud designs. In this graph, Watts (W) is a proxy for performance (ops/s); given the same RCA, maximizing Watts maximizes server performance. If power density is high, then very little silicon can be placed in a lane within the temperature limits, and it must be divided into many smaller dies. As power density decreases, then more silicon can fit per lane, and fewer chips can be used. Sensibly, moving left, silicon area cost dominates total server cost; moving right, PCB, cooling and packaging costs dominate.

In the next section, we will continue by connecting voltage scaling of the RCA's to our model. By adjusting voltage, we can change the power density of the RCA's to traverse the X axis in Figure 4.12, creating a spectrum of tradeoffs between the two key metrics for ASIC Servers: $ per op/s and W per op/s. The next section will also incorporate DC/DC converter costs, which are application dependent.

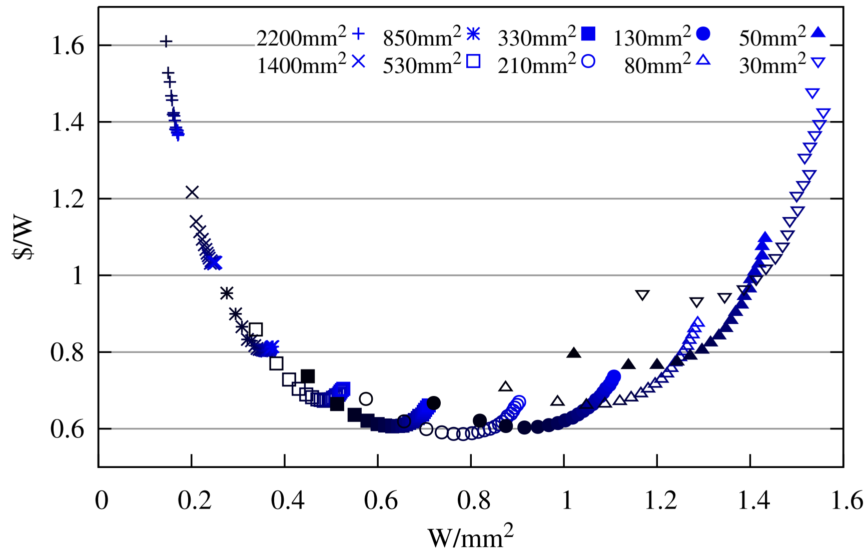


Figure 4.12: **Minimizing cost by optimizing the number of RCA's per ASIC Server lane (measured in mm^2), and the number of chips it should be divided into.** Point series with equal silicon area start from the right with the maximum number of chips, 20, and decrease until individual chip sizes hits the limit of 600 mm^2. Low power density RCA's are dominated by silicon costs, while high power density RCA's are dominate by cooling, packaging and server component overheads.

4.6 Bitcoin ASIC Cloud design

In this section, we specialize our generic ASIC Cloud architecture to the architecture of a Bitcoin ASIC Cloud. Architecturally, the Bitcoin RCA repeatedly executes a Bitcoin hash operation. The hash operation uses an input 512 bit block that is reused across billions of hashes; and then repeatedly does the following: It mutates the block and performs a SHA256 hash on it. The output of this hash is then fed into another SHA256 hash, and then a leading zero count is performed on the result and compared to the target leading zero count to determine if it has succeeded in finding a valid hash. The two SHA256 hashes comprise 99% of the computation. Each SHA256 consists of 64 rounds.

There are two primary styles for implementing the Bitcoin RCA. The most prevalent style is the pipelined implementation, which unrolls all 64 rounds of SHA256 and inserts pipeline registers, allowing a new hash to be performed every cycle by inserting a new mutated block each cycle [15, 140]. The less prevalent style, used by Bitfury, performs the hash in place, and has

been termed a rolled core.

We created an industrial quality pipelined Bitcoin RCA in synthesizeable RTL. We searched the best design parameters such as the number of pipeline stages and cycles per stage to realize the best energy efficiency, Watts per GH/s, by iteratively running the designs through the CAD tools and performing power analysis with different parameter values.

The Bitcoin implementation is fully-pipelined and consists of 128 one-clock stages, one per SHA256 round.

The RTL was synthesized, placed, and routed with the 1.0V UMC 28nm standard cell library and Synopsys Design Compiler and IC Compiler. We performed parasitic extraction and evaluated dynamic power using PrimeTime. The resulting delay and capacitance information are used to evaluate the clock latency and power consumption of the design.

Our final implementation occupies 0.66 mm^2 of silicon in the UMC 28-nm process. At the UMC process's nominal voltage 1.0V, it runs at 830 MHz, and attains a staggering power density of 2W per mm^2. Because SHA256 is comprised of combinational logic and flip-flops, and contains no RAMs, the energy density is extremely high. Moreover, data in cryptographic circuits is essentially random and has extremely high transition activity factors: 50% or higher for combinational logic, and 100% for flip flops.

4.6.1 Building a Bitcoin ASIC Cloud

Though Bitcoin performance scales out easily with more RCA's, it is a near-worst case for dark silicon, and high power density prevents full-frequency implementations.

Using the infrastructure we developed in Section 4.5, we ran simulations exploring the design space of servers for a Bitcoin ASIC Cloud Server. Figure 4.13 shows an inital set of results, comparing $ per GH/s to power density, W/mm^2, as the amount of silicon is varied and the number of chips decreases from 20 going from right to left. The corresponding core voltages that with frequency adjustment tune the RCAs to that particular power density are given. These

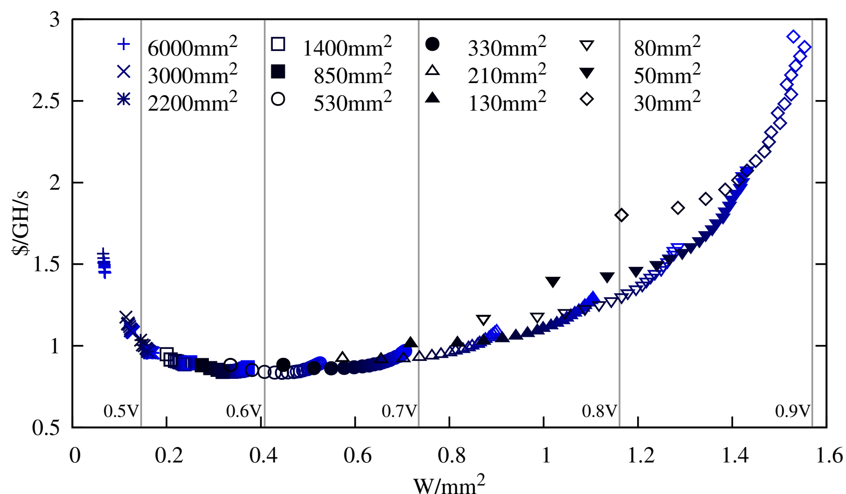


Figure 4.13: **Bitcoin Voltage versus Cost Performance.** Each series represents a fixed total amount of silicon per lane, while going from right to left decreases chip count from 20. Due to Bitcoin's high power density, voltage over 0.6 V is suboptimal for $/GH/s because of high cooling overheads and small silicon per server.

voltages represent the maximum voltage that that server can sustain without exceeding junction temperatures. In non-thermally limited designs, we would expect higher voltage systems to be lower cost per performance; but we can see that Bitcoin ASICs running at higher voltages are dominated designs, because the power density is excessive, and server overheads dominate. All Pareto-optimal designs are below 0.6 V.

Pareto-Optimal Servers. Although power density (W/mm^2) and voltage are important ASIC-level properties, at the ASIC Cloud Server level, what we really want to see are the tradeoffs between energy-efficiency (W per op/s) and cost-perf, ($ per op/s). To attain this, we ran simulations that exhaustively evaluated all server configurations spanning a range of settings for total silicon area per lane, total chips per lane, and all operating voltages from 0.4 up in increments of 0.01V, and pruning those combinations that violate system requirements. In Figure 4.14, we show a subset of this data, which shows for servers with 10 chips per lane, the relationship between silicon area per lane and voltage. Points in a series start on the top at 0.4V and increase down the page. Note that these points represent voltage-customized machines rather than the metrics of a single system under DVFS.

Generally, as more silicon is added to the server, the optimal voltages go lower and lower,

and the server energy efficiency improves. Initially, $ per op/s decreases with increasing silicon as server overheads are amortized, but eventually silicon costs start to dominate and cost starts to rise. Cost-efficiency of the smaller machines declines with lower voltage because performance is decreasing rapidly and many PCB- and server-level cost overheads remain constant.

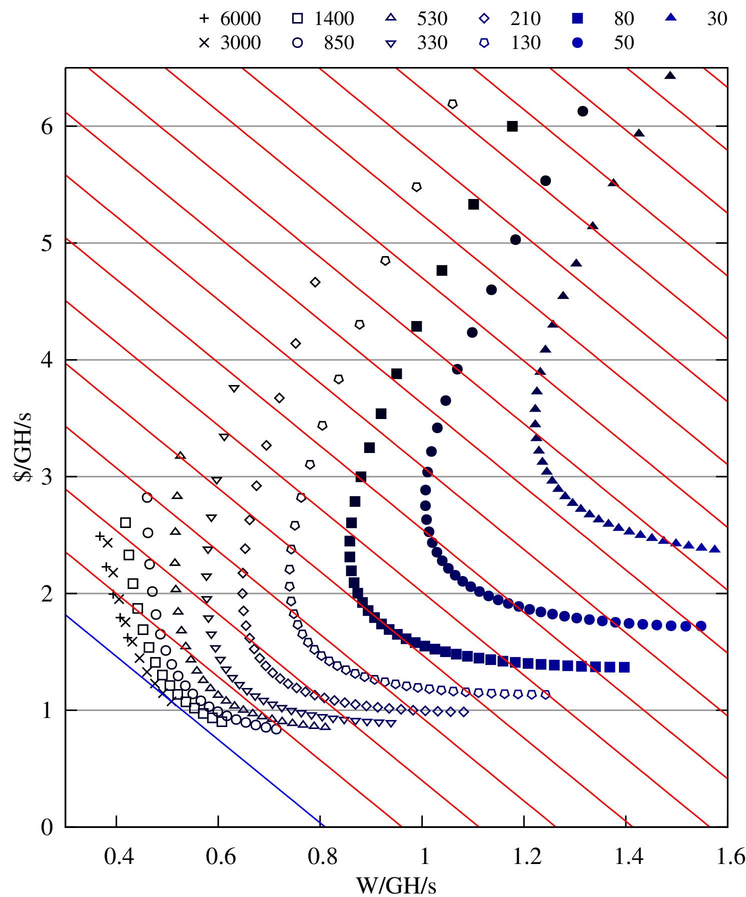


Figure 4.14: **Bitcoin Cost versus Energy Efficiency Pareto.** Servers with 10 chips per lane. Each point series shows total silicon per lane and voltage increases from top to bottom points. Diagonal lines represent equal TCO per GH/s, with min. TCO at lower left.

Table 4.3 shows the Pareto-optimal points that represent energy-optimal and cost-optimal designs across all simulation points. These are the end points of the Pareto frontier.

In the most energy-efficient design, the server runs at ultra-low near-threshold voltages, 0.40V and 70 MHz, and occupies 6,000 mm^2 of silicon per lane spread across 10 chips of maximum die size, 600 mm^2. Since it employs almost a full 12" wafer per server, the cost is

Table 4.3: **Bitcoin ASIC Cloud Optimization Results.**

	W/GH/s Optimal	TCO/GH/s Optimal	$/GH/s Optimal
# of ASICs per lane	10	10	5
# of Lanes	8	8	8
Logic Voltage (V)	0.40	0.49	0.62
Clock Frequency (MHz)	70	202	465
Die Size (mm^2)	600	300	106
Silicon/Lane (mm^2)	6,000	3,000	530
Total Silicon (mm^2)	48,000	24,000	4,240
GH/s/server	5,094	7,341	2,983
W/server	1,872	3,731	2,351
$/server	12,686	7,901	2,484
W/GH/s	0.368	0.508	0.788
$/GH/s	2.490	1.076	0.833
TCO/GH/s	4.235	3.218	4.057
Server Amort./GH/s	2.615	1.130	0.874
Amort. Interest/GH/s	0.161	0.069	0.054
DC CAPEX/GH/s	0.884	1.222	1.895
Electricity/GH/s	0.319	0.441	0.684
DC Interest/GH/s	0.257	0.355	0.550

highest, at $2.49 per GH/s, but energy efficiency is excellent: 0.368 W per GH/s.

In the most cost-efficient design, the server is run at a much higher voltage, 0.62V and 465 MHz, and occupies less silicon per lane, 530 mm^2, spread across 5 chips of 106 mm^2. $ per GH/s is minimized, at $0.833, but energy efficiency is not as good: 0.788 W per GH/s.

Figure 4.15 shows the cost breakdown for these two Pareto-optimal servers. In the energy-optimal server, silicon costs dominate all other costs, since high energy-efficiency minimizes cooling and power delivery overheads. In the cost-optimal server, large savings in silicon cost gained by higher voltage operation are partially offset by much higher DC/DC converter expenses, which are the dominate cost of the system. PSU and fan overheads also increase with voltage.

TCO-Optimal Servers. A classic conundrum since the beginning of energy-efficiency research in computer architecture has been how to weight energy efficiency and performance

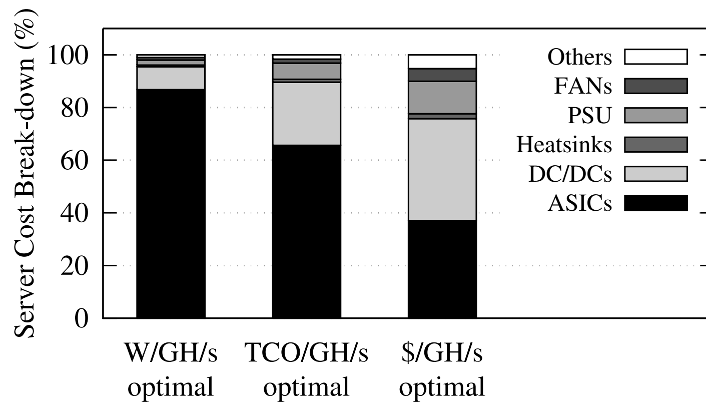


Figure 4.15: **Bitcoin servers cost breakdown.**

against each other. In the absence of whole system analysis, this gave birth to such approximations as Energy-Delay Product and Energy-Delay^2. A more satisfying intermediate solution is the Pareto Frontier analysis shown earlier in this section. But, which of the many optimal points is most optimal? This dilemma is not solely academic. In Bitcoin Server sales, the primary statistics that are quoted for mining products are in fact the exact ones given in this chapter: $ per GH/s and W per GH/s. Many users blindly chose the extremes of these two metrics, either over-optimizing for energy-efficiency or for cost-efficiency.

Fortunately, in the space of ASIC Clouds, we have an easy solution to this problem: TCO analysis. TCO analysis incorporates the datacenter-level constraints including the cost of power delivery inside the datacenter, land, depreciation, interest, and the cost of energy itself. With these tools, we can correctly weight these two metrics and find the over-all optimal point (TCO-optimal) for the ASIC Cloud.

In this chapter, we employ a refined version of the TCO model by Barroso et al [16]. Electricity is $0.06 per KWh. In Figure 4.14, we annotate lines of equal TCO according to this model. The lowest TCO is found on the bottom left. As we can see, TCO is most optimized for large silicon running at relatively low, but not minimal, voltages.

The TCO-optimal design is given in Table 4.3. The server runs at moderate near-threshold voltages, 0.49V and 202 MHz, and occupies 3,000 mm^2 of silicon per lane spread across 10 chips

of moderate die size, 300 mm^2. Cost is between the two Pareto-optimal extremes, at $1.076 per GH/s, and energy efficiency is excellent but not minimal: 0.508 W per GH/s. The TCO per GH/s is $3.218, which is significantly lower than the TCO with the energy-optimal server, $4.235, and the TCO with the cost-optimal server, $4.057. The portion of TCO attributable to ASIC Server cost is 35%; to Data Center capital expense is 38%, to electricity, 13.7%, and to interest, about 13%. Finally, in Figure 4.15, we can see the breakdown of Server components; silicon dominates, but DC/DC is not insignificant.

Voltage Stacking. Because DC/DC power is significant, some Bitcoin mining ASIC Clouds employ voltage stacking, where chips are serially chained so that their supplies sum to 12V, eliminating DC/DC converters. We modified our tools to model voltage stacking. The TCO-optimal voltage-stacked design runs at 0.48V and 183 MHz, employs the same chips, and achieves 0.444 W per GH/s, $ 0.887 per GH/s, and a TCO per GH/s of $2.75, a significant savings.

4.7 Expanding the methodology to variety of applications

In this section we apply the methodology developed for Bitcoin to three other applications with very different characteristics: Litecoin, Video Transcoding and Convolutional Neural Net. We explain how their ASIC Cloud design is different and evaluate what is the optimal TCO for each application.

4.7.1 Litecoin ASIC Cloud design

The Litecoin cryptocurrency is similar to Bitcoin, but employs the Scrypt cryptographic hash instead of the Bitcoin hash, and is intended to be dominated by accesses to large SRAMs. We implemented a Scrypt-based Litecoin replicated compute accelerator in RTL and pushed it through a full synthesis, place and route flow using Synopsys Design Compiler and IC Compiler. We

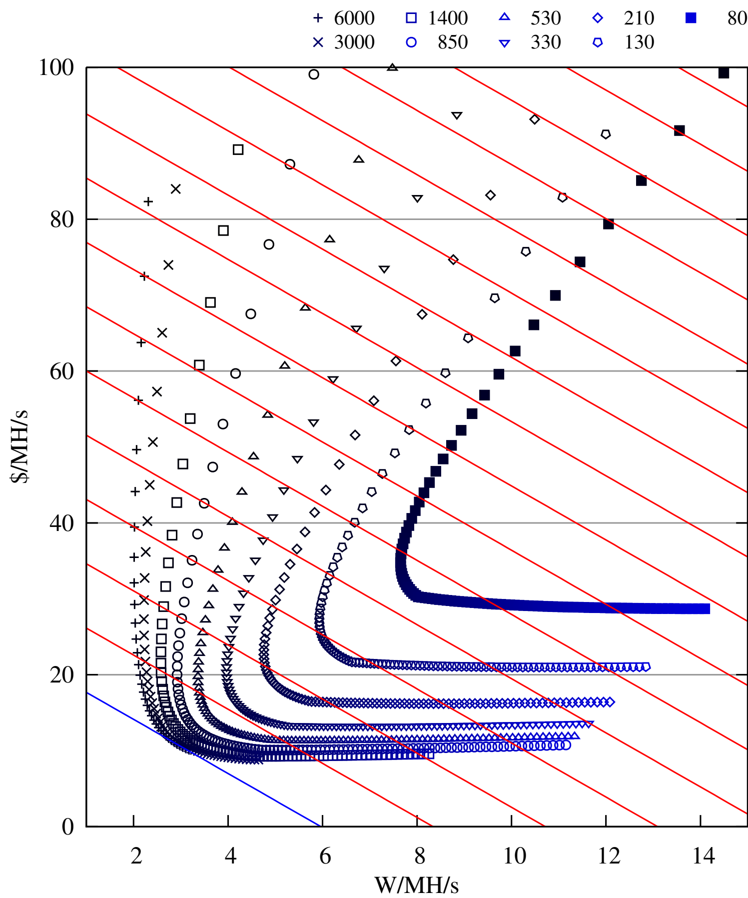


Figure 4.16: **Litecoin Cost versus Energy Efficiency Pareto.** Servers with 12 chips per lane. Each point series shows total silicon per lane and voltage increases from top to bottom points. Diagonal lines represent equal TCO per MH/s, with min. TCO at lower left.

applied our ASIC Cloud design exploration toolkit to explore optimal Litecoin server designs. Our results show that because Litecoin consists of repeated sequential accesses to 128KB memories, the power density per mm^2 is much lower, which leads to larger chips at higher voltages versus Bitcoin. Because Litecoin is so much more memory intensive, performance is typically measured in megahash per second (MH/s). SRAM V_{min} is set to 0.9V.

Pareto and TCO analysis is in Figure 4.16 and stats for the final designs are given in Table 4.4.

Litecoin results are remarkably different from Bitcoin. In the most energy-efficient design, the server runs at moderate near-threshold voltages, 0.47V and 169 MHz, and occupies 6,000

mm^2 of silicon per lane spread across 10 chips of maximum die size, 600 mm^2. Since it employs almost a full 12" wafer per server, the cost is highest, at $36.67 per MH/s, but energy efficiency is excellent: 2.011 W per MH/s.

In the most cost-efficient design, the server is run at a much higher voltage, 0.91V and 849 MHz, and occupies less silicon per lane, 300 mm^2, spread across 10 chips of 300 mm^2. $ per MH/s is minimized, at $8.75, but energy efficiency is not as good: 4.475 W per MH/s.

The TCO-optimal server operates at moderate super-threshold voltage, 0.70V and 615 MHz, and also occupies 6,000 mm^2 of silicon per lane spread across 12 chips of large die size, 500 mm^2. Cost is between the two Pareto-optimal extremes, at $10.842 per MH/s, and energy efficiency is excellent but not minimal: 2.922 W per MH/s. The TCO per MH/s is $23.686, which is significantly lower than the TCO with the energy-optimal server, $48.86, and the TCO with the cost-optimal server, $27.532. The portion of TCO attributable to ASIC Server cost is 48.1%; to Data Center capital expense is 29.7%, to electricity, 10.7%, and to interest, about 11.5%.

Qualitatively, these varying results are a direct consequence of the SRAM-dominated nature of Litecoin, and thus the results are likely to be more representative of accelerators with a high percentage of memory. The current world-wide Litecoin mining capacity is 1,452,000 MH/s, so 1,248 servers would be sufficient to meet world-wide capacity.

4.7.2 Video transcode ASIC Cloud design

Video transcoding is an emerging planet-scale computation, as more users record more of their lives in the cloud, and as more governments surveil their citizens. Typical video services like YouTube receive uploaded videos from the user, and then distribute frames across the cloud in parallel to re-encode in their proprietary format. We model an ASIC Cloud, *XCode*, that transcodes to H.265 (or HEVC), and model the RCA based on parameters in [73]. For this design, the RCAs on an ASIC share a customized memory system: ASIC-local LPDDR3 DRAMs to store the pre- and post- transcoded video frames. *Thus, this RCA is most representative of accelerators*

Table 4.4: **Litecoin ASIC Server Optimization Results.**

	W/MH/s Optimal	TCO/MH/s Optimal	$/MH/s Optimal
# of ASICs per lane	10	12	10
# of Lanes	8	8	8
Logic Voltage (V)	0.47	0.70	0.91
Clock Frequency (MHz)	169	615	849
Die Size (mm^2)	600	500	300
Silicon/Lane (mm^2)	6,000	6,000	3,000
Total Silicon (mm^2)	48,000	48,000	24,000
MH/s/server	319	1,164	803
W/server	641	3,401	3,594
$/server	11,689	12,620	7,027
W/MH/s	2.011	2.922	4.475
$/MH/s	36.674	10.842	8.750
TCO/MH/s	48.860	23.686	27.523
Server Amort./MH/s	38.508	11.384	9.188
Amort. Interest/MH/s	2.366	0.700	0.565
DC CAPEX/MH/s	4.835	7.024	10.759
Electricity/MH/s	1.746	2.537	2.886
DC Interest/MH/s	1.405	2.041	3.126

that require external DRAM.

In our PCB layout, we model the space occupied by these DRAMs, which are placed in rows of 3 on either side of the ASIC they connect to, perpendicular to airflow, and limit the number of ASICs placed in a lane given finite server depth. We also model the more expensive PCBs required by DRAM, with more layers and better signal/power integrity. We employ two 10-GigE ports as the off-PCB interface, and model the area and power of the memory controllers assuming that they do not voltage scale, and model pin constraints.

Our ASIC Cloud simulator explores the design space across number of DRAMs per ASIC, logic voltage, area per ASIC, and number of chips. DRAM cost and power overhead are not insignificant, and so the Pareto-optimal designs ensure DRAM bandwidth is saturated, which means that chip performance is set by DRAM count. As voltage and frequency is lowered, area

increases to meet the performance requirement. One DRAM satisfies 22 RCA's at 0.9V.

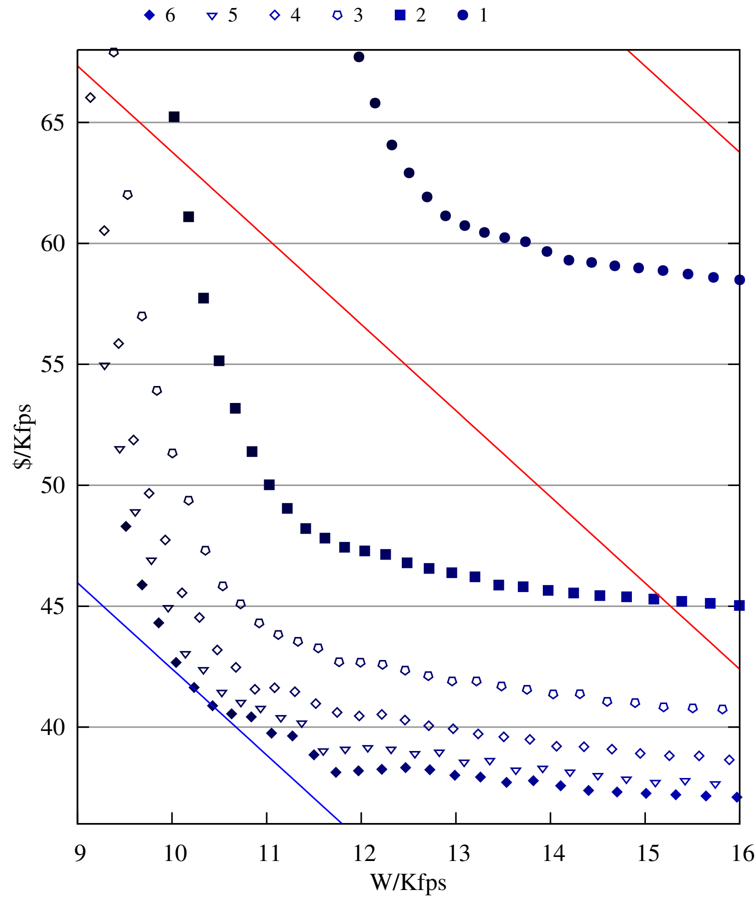


Figure 4.17: **Video Transcoding Pareto Curve.** Each point series corresponds to 5 ASICs per lane and a certain number of DRAMs per ASIC, and varies voltage; highest logic voltage is at lower right. Lower-left most diagonal line indicates lowest TCO per Kfps.

Figure 4.17 shows the Pareto analysis, using point series corresponding to # of DRAMs per ASIC. Points to the upper left have lower voltage and higher area. To an extent, larger DRAM count leads to more Pareto-optimal solutions because they have greater performance per server and minimize overheads. The Pareto points are glitchy because of variations in constants and polynomial order for various server components as they vary with voltage.

Table 4.5 shows stats for the Pareto optimal designs. The cost-optimal server packs the maximum number of DRAMs per lane, 36, maximizing performance. However, increasing the number of DRAMs per ASIC requires higher logic voltage (1.4V!) to stay within the max die

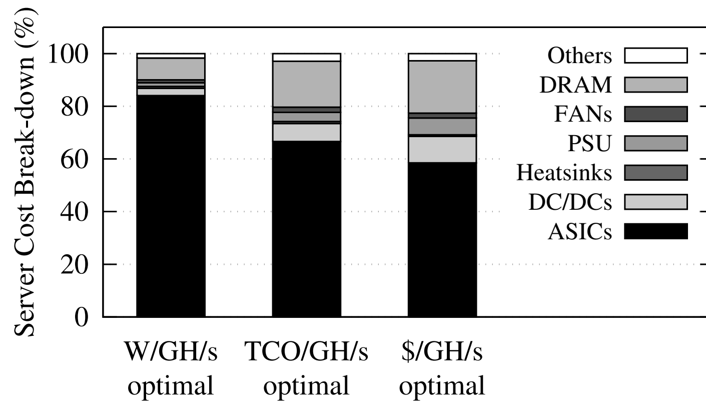


Figure 4.18: **Video Transcoding Server cost breakdown.**

area constraint, resulting in less energy efficient designs. Hence, the energy-optimal design has fewer DRAMs per ASIC and per lane (24), while gaining back some performance by increasing ASICs per lane which is possible due to lower power density at 0.53V. The TCO-optimal design increases DRAMs per lane, 30, to improve performance, but is still close to the optimal energy efficiency at 0.8V, resulting in a die size and frequency between the other two optimal points.

Figure 4.18 shows the cost breakdown. Silicon always dominates, but DRAMs and DC/DC occupy a greater percentage as performance and voltage are scaled up, respectively.

4.7.3 Convolutional Neural Net ASIC Cloud design

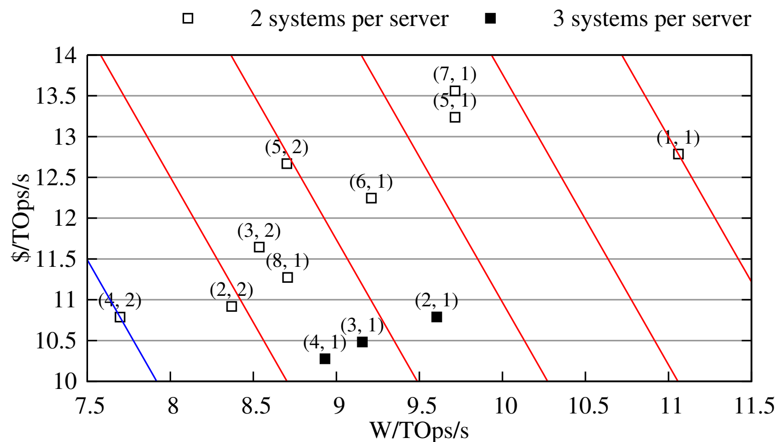


Figure 4.19: **Convolutional Neural Net Pareto Curve.** Twelve possible ASIC Cloud designs for 8x8 DaDianNao. The parenthesized numbers show number of RCAs per chip in each direction. Lower left diagonal line represents min. TCO per TOps/s.

Table 4.5: **Video Transcoding ASIC Cloud Optimization Results.**

	W/Kfps Optimal	TCO/Kfps Optimal	$/Kfps Optimal
# of DRAMs per ASIC	3	6	9
# of ASICs per lane	8	5	4
# of Lanes	8	8	8
Logic Voltage (V)	0.53	0.80	1.40
Clock Frequency (MHz)	163	439	562
Die Size (mm^2)	595	456	542
Silicon/Lane (mm^2)	4,760	2,280	2,168
Total Silicon (mm^2)	38,080	18,240	17,344
Kfps/server	127	159	190
W/server	1,109	1,654	3,216
$/server	10,779	6,482	6,827
W/Kfps	8.741	10.428	16.904
$/Kfps	84.975	40.881	35.880
TCO/Kfps	129.416	86.971	107.111
Server Amort./Kfps	89.224	42.925	37.674
Amort. Interest/Kfps	5.483	2.638	2.315
DC CAPEX/Kfps	21.015	25.07	40.639
Electricity/Kfps	7.590	9.055	14.678
DC Interest/Kfps	6.105	7.283	11.806

We chose our last application to be Convolutional Neural Networks (CNN), a deep learning algorithm widely used in data centers. We based our RCA on DaDianNao [33] (DDN), which describes a 28-nm eDRAM-based accelerator chip for Convolutional and Deep Neural Networks. In their chip design, they have HyperTransport links on each side allowing the system to gluelessly scale to a 64-chip system in an 8-by-8 mesh. We evaluate ASIC Cloud servers that implement full 8-by-8 DDN systems, presuming that the maximum CNN size is also the most useful. Our RCA is identical to one DDN chip, except that we sensibly replace the HyperTransport links between RCAs with on-chip network links, if the RCAs are co-located on the same ASIC. HyperTransport interfaces are still used between chips. The more RCAs that are integrated into a chip, the fewer total HyperTransport links are necessary, saving cost, area and power. In this

scenario, we assume that we do not have control over the DDN micro-architecture, and thus that voltage scaling is not possible. RCAs are arranged in 8x8 arrays that are partitioned across an equal or smaller array of ASICs. ASICs connect over HyperTransport within a lane and also nearest-neighbor through the PCB across lanes. For example, a 4x2 ASIC has 4 nodes in the lane direction and 2 nodes in the across-lane direction. 2 ASICs per lane and 4 lanes would be required to make a complete 8x8 system. We then pack as many 8x8 systems as thermally and spatially possible, and provision the machine with multiple 10-GigE off-PCB links. Up to 3 full 64-node DDN systems fit in a server.

Table 4.6: **Convolutional Neural Network ASIC Cloud Results.**

	W/TOps/s Optimal	TCO/TOps/s Optimal	$/TOps/s Optimal
Chip type	4x2	4x2	4x1
# of ASICs per lane	2	2	6
# of Lanes	8	8	8
Logic Voltage (V)	0.90	0.90	0.90
Clock Frequency (MHz)	606	606	606
Die Size (mm^2)	454	454	245
Silicon/Lane (mm^2)	908	908	1,470
Total Silicon (mm^2)	7,264	7,264	11,760
TOps/s/server	235	235	353
W/server	1,811	1,811	3,152
$/server	2,538	2,538	3,626
W/TOps/s	7.697	7.697	8.932
$/TOps/s	10.788	10.788	10.276
TCO/TOps/s	42.589	42.589	46.92
Server Amort./TOps/s	11.327	11.327	10.790
Amort. Interest/TOps/s	0.696	0.696	0.663
DC CAPEX/TOps/s	18.506	18.506	21.474
Electricity/TOps/s	6.684	6.684	7.756
DC Interest/TOps/s	5.376	5.376	6.238

TCO analysis in Figure 4.19 shows the results for the 12 different configurations, and Table 4.6 shows the Pareto and TCO optimal designs. Similar to the XCode ASIC Cloud,

performance is only dependent on the number of 8x8 DDN systems, making the system more cost efficient by putting the most possible number of systems on a server. We allow partial chip usage, e.g. arrays that have excess RCA's that are turned off, but these points were not Pareto Optimal.

The cost-optimal system used ASICs with fewer RCAs, so 3 systems to be squeezed in. The energy- and TCO- optimal system only fit two 8x8 systems per server, but had a larger, squarish array of RCAs (4x2), removing many HyperTransport links, and thus minimizing energy consumption.

4.8 When do we go to ASIC Clouds?

In Table 4.7, we step back and compare the performance of CPU Clouds versus GPU Clouds versus ASIC Clouds for the four applications that we presented. ASIC Clouds outperform CPU Cloud TCO per op/s by 6,270x; 704x; and 8,695x for Bitcoin, Litecoin, and Video Transcode respectively. ASIC Clouds outperform GPU Cloud TCO per op/s by 1057x, 155x, and 199x, for Bitcoin, Litecoin, and Convolutional Neural Nets, respectively.

Table 4.7: **CPU Cloud vs. GPU Cloud vs. ASIC Cloud Deathmatch.**

Application	Perf. metric	Cloud HW	Perf.	Power (W)	Cost ($)	lifetime (years)	Power/ op/s.	Cost/ op/s.	TCO/ op/s.
Bitcoin	GH/s	C-i7 3930K(2x)	0.13	310	1,272	3	2,385	9,785	20,192
Bitcoin	GH/s	AMD 7970 GPU	0.68	285	400	3	419	588	3,404
Bitcoin	GH/s	28nm ASIC	7,341	3,731	7,901	1.5	0.51	1.08	3.22
Litecoin	MH/s	C-i7 3930K(2x)	0.2	400	1,272	3	2,000	6,360	16,698
Litecoin	MH/s	AMD 7970 GPU	0.63	285	400	3	452	635	3,674
Litecoin	MH/s	28nm ASIC	1,164	3,401	12,620	1.5	2.92	10.8	23.7
Video Transcode	Kfps	Core-i7 4790K	0.0018	155	725	3	88,571	414,286	756,489
Video Transcode	Kfps	28nm ASIC	159	1,654	6,482	1.5	10.4	40.9	87.0
Conv Neural Net	TOps/s	NVIDIA Tesla K20X	0.26	225	3,300	3	865	12,692	8,499
Conv Neural Net	TOps/s	28nm ASIC	235	1,811	2,538	1.5	7.70	10.8	42.6

Given these extraordinary improvements in TCO, what determines when ASIC Clouds should be built? We have shown some clear examples of planet-scale applications that could merit ASIC Clouds. The key barrier is the cost of developing the ASIC Server, which includes both the mask costs (about $ 1.5M for the 28 nm node we consider here), and the ASIC development

costs, which collectively, we term the non recurring engineering expense (NRE).

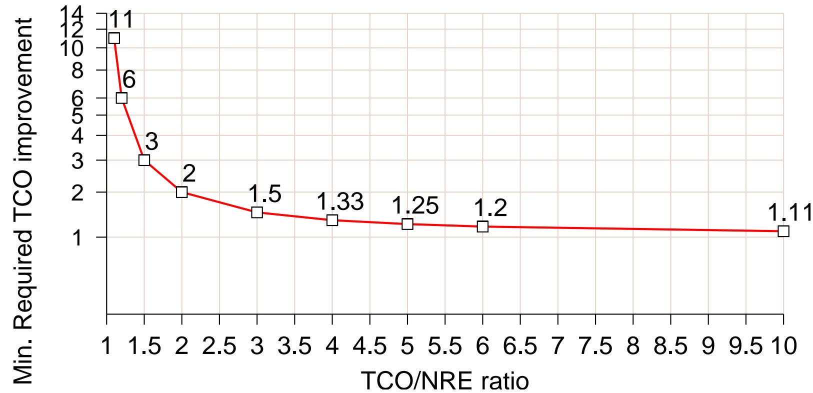


Figure 4.20: **Breakeven point for ASIC Clouds.**

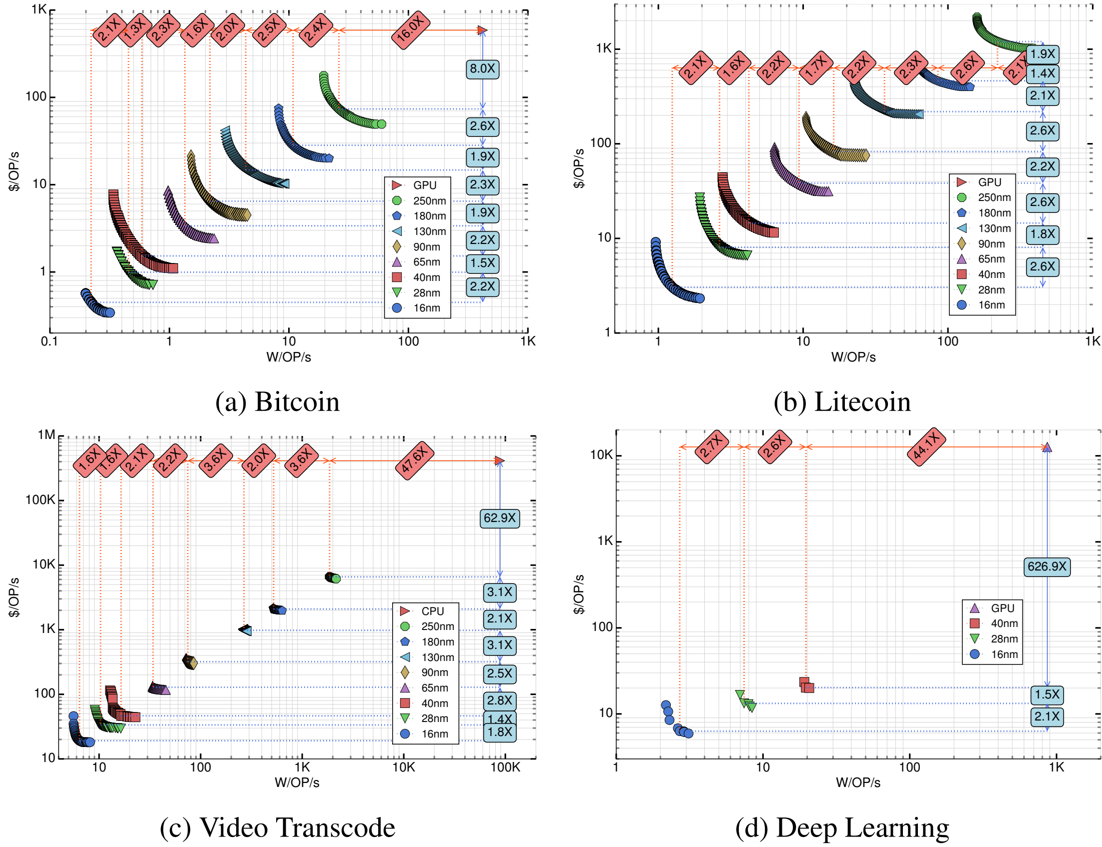
We propose the *two-for-two rule*. If the cost per year (i.e. the TCO) for running the computation on an existing cloud exceeds the NRE by 2X, and you can get at least a 2X TCO per op/s improvement, then going ASIC Cloud is likely to save money. Figure 4.20 shows a wider range of breakeven points. Essentially, as the TCO exceeds the NRE by more and more, the required speedup to breakeven declines. As a result, almost any accelerator proposed in the literature, no matter how modest the speedup, is a candidate for ASIC Cloud, depending on the scale of the computation.

4.9 Computing Pareto-optimal designs for each tech node

Here we find the TCO per op/s optimal design for each technology node. We examine trends among different technologies for these Pareto-optimal designs.

4.9.1 Comparing Pareto-optimal points across nodes

Sections 4.6 and 4.7 explored different designs to find the optimal TCO per op/s design. This evaluation is performed for different technology nodes and the Pareto-optimal points are selected according to $ per op/s and W per op/s, considering different die area values, operating



(a) Bitcoin

(b) Litecoin

(c) Video Transcode

(d) Deep Learning

Figure 4.21: **Pareto frontiers improve in both energy and cost efficiency for newer technologies.** Each set of points represent the Pareto frontiers in each technology; dotted lines indicate the TCO-optimal Pareto point for each technology. Relative improvements in cost per performance and power per performance for the TCO-optimal points in each two consecutive technology nodes are indicated, and between the oldest evaluated node and baseline GPU/CPU.

voltages, and number of ASICs per lane. Figure 4.21 shows these points for all technologies and compares to baseline GPU/CPU server.

Among different technology nodes we have both energy per op and cost per performance benefits from going towards newer nodes. Due to smaller changes in nominal voltages among newer nodes than older ones, and also the dark silicon phenomenon [147, 50, 144, 141, 41, 113], the improvement among two consecutive technology nodes degrades for more advanced nodes. Transistor density, operating frequency range and voltage range among different technologies impact the power density of the dies, while thermal extraction limit is technology-independent. Therefore the spread of Pareto-optimal points for each technology node is different. Also in most

Pareto curves 28nm and 40nm are closer to each other compared to other adjunct technologies. They have the same nominal voltage which reduces the energy efficiency gains.

Table 4.8: **Bitcoin TCO-optimal ASIC Server properties across tech nodes.** Bitcoin is extremely power dense resulting in TCO-optimal servers operating at very low voltages.

Tech	**250nm**	**180nm**	**130nm**	**90nm**	**65nm**	**40nm**	**28nm**	**16nm**
RCAs per Die	10	20	39	83	159	377	769	1,818
Die Area (mm^2)	559	579	588	600	599	540	540	420
Die Cost ($)	16	18	29	32	33	42	66	74
Dies/Server	120	120	120	120	120	120	72	48
Logic Vdd	1.081	0.857	0.654	0.563	0.517	0.433	0.459	0.424
Freq. (MHz)	37	54	77	93	100	121	149	169
GH/s	42	121	347	914	1,888	5,466	8,223	14,687
Power (W)	1,089	1,314	1,509	1,997	2,541	3,217	3,736	3,246
Cost (K$)	3.1	3.4	5.1	5.9	6.4	8.3	8.2	6.6
W/GH/s	26.21	10.83	4.350	2.183	1.346	0.589	0.454	0.221
$/GH/s	73.71	28.29	14.72	6.469	3.383	1.527	0.994	0.449
TCO/GH/s	186.2	74.55	33.68	15.88	9.115	4.039	2.912	1.378
NRE K$	561	602	790	1,054	1,194	1,845	2,760	6,451

For Bitcoin, even 250nm shows a $\sim$12$\times$ TCO improvement over the baseline GPU. Litecoin is memory dominated and as a result the 45nm GPU surpasses 250nm by 1.9$\times$ in $ per op/s, but underperforms by 2.1$\times$ in energy. In the end, the 250nm ASIC has superior TCO. Video Transcode benefits substantially from going to ASIC, since many Transcode units are placed per chip and accelerator performance and energy efficiency far outpaces CPUs, even at 250nm. Due to our assumption about SLA requirements for the Deep Learning application, older technology nodes than 40nm cannot be used. Therefore, the initial jump from GPU to ASIC is substantial, but comes at significant NRE.

4.9.2 Comparison of TCO-optimal designs across nodes

The TCO-optimal point in each curve is selected and detailed results for each of the applications in different technology nodes can be found in Tables 4.8, 4.9, 4.10, and 4.11.

Table 4.9: **Deep Learning TCO-optimal ASIC Server properties across tech nodes.** Frequency is kept constant for Deep Learning servers to satisfy SLA requirements.

Tech	40nm	28nm	16nm
RCA per Die	2x1	2x2	4x2
Die Area (mm^2)	259	298	195
Die Cost ($)	29	61	62
Dies/Server	32	64	80
Logic Vdd	1.285	0.900	0.615
Freq. (MHz)	607	606	617
TOps/s	118	470	1,176
Power (W)	2,312	3,493	3,184
Cost (K$)	2.4	6.2	7.4
W/TOps/s	19.60	7.431	2.708
$/TOps/s	20.25	13.25	6.304
TCO/TOps/s	100.4	44.28	17.78
NRE K$	3,259	4,301	8,616

Looking across technology nodes in each application, we see a general trend of decreasing voltages. The explanation is as follows: the non-power limited performance of the silicon is improving because of transistor frequency increases and transistor count increases, scaling as S^3. At the same time, wafer costs (and thus cost per mm^2) are increasing by S, resulting in a net silicon potential improvement of S^2 in $ per op/s. At the same time, capacitance is decreasing by S, improving energy efficiency by the same amount. TCO-optimal systems will balance improvements in $ per op/s with improvements in W per op/s. Due to this S^2 versus S mismatch in CMOS scaling (especially with dark silicon, where nominal Vdd stops dropping), optimal designs drive the voltage further and further below nominal Vdd to make up for the difference.

How many ticks before a tock? Intel made famous the idea of separating process scaling (ticks) from architectural refactors (tocks). To understand the importance of data-center customization and co-sensitivity of architecture and process, we ported the TCO-optimal ASIC design in each process technology to future nodes, for example, the 250nm Bitcoin design was mapped from 180nm to 16nm. In this porting, RCAs count per ASIC is fixed. Only the operating

Table 4.10: **Litecoin TCO-optimal ASIC Server properties across tech nodes.** Litecoin design is SRAM dominated with low power density, thus TCO optimal servers use a voltage close to technology nominal voltage to benefit from the available cooling opportunity to gain higher performance.

Tech	250nm	180nm	130nm	90nm	65nm	40nm	28nm	16nm
RCAs per Die	12	22	47	98	188	446	910	2,150
Die Area (mm^2)	567	539	599	599	599	540	540	420
Die Cost ($)	16	17	29	32	32	42	66	74
Dies/Server	120	120	120	120	120	120	120	80
Logic Vdd	1.845	1.378	1.138	0.924	0.816	0.697	0.656	0.594
Freq. (MHz)	78	109	173	239	281	417	576	776
MH/s	2	6	21	62	139	491	1,384	2,938
Power (W)	516	525	769	1,000	1,298	2,068	3,662	3,664
Cost (K$)	2.8	2.9	4.6	5.1	5.3	7.1	11.2	9
W/MH/s	219.8	85.19	36.45	16.25	9.360	4.216	2.645	1.247
$/MH/s	1203	463.4	218.7	82.83	38.41	14.53	8.059	3.052
TCO/MH/s	2,214	854.8	388.5	156.8	79.97	32.94	19.49	8.353
NRE K$	591	633	835	1,104	1,254	1,924	2,823	6,404

voltage is changed to fit the thermal budget and then the TCO-optimal voltage is selected. DRAM count per ASIC for Video Transcode and operating frequency for Deep Learning are also fixed, but the PCB is redesigned.

The farther the destination node is from the source node, the less optimal a ported design is compared to the TCO-optimal design for the destination node. Porting the optimal ASIC design in 250 nm to 16nm is worse in TCO by 2.14X, 3.68X, 6.71X for Litecoin, Bitcoin and Video Transcoding respectively. Porting the optimal ASIC design in 65nm to 16nm has worse TCO by 1.34X for Deep Learning.

On the other hand, porting across a single node leads to smaller TCO penalty: 1.05X in Bitcoin, 1.08X in Litecoin, and 1.06X in Deep Learning. For Video Transcode, designs are less stable, with a geo-mean of 1.53X at 65nm and above; and a geo-mean of 1.07X at 40nm and below. The large changes are due to successive DRAM technology improvements, ramping to LPDDR3 in 65nm.

Table 4.11: **Video Transcode TCO-optimal ASIC Server properties across tech nodes.** Video Transcode optimal servers try to saturate DRAM bandwidth and trade-off operating voltage with RCAs per ASIC.

Tech	**250nm**	**180nm**	**130nm**	**90nm**	**65nm**	**40nm**	**28nm**	**16nm**
RCAs per Die	2	5	9	19	37	92	153	140
DRAMs per Die	1	1	1	1	1	3	6	9
Die Area (mm^2)	493	634	627	619	623	594	498	177
Die Cost ($)	14	21	32	35	35	49	65	34
Dies/Server	64	64	64	64	64	64	40	32
Logic Vdd	2.533	1.818	1.501	1.171	1.015	0.957	0.754	0.710
Freq. (MHz)	56	77	115	165	215	358	429	705
Kfps	0.3	1.3	4	12	30	126	158	190
Power (W)	628	674	985	875	1,024	2,077	1,633	1,220
Cost (K$)	2.2	2.7	3.6	3.7	3.9	5.9	5.3	3.6
W/Kfps	1860	523.0	266.9	74.70	34.01	16.47	10.34	6.418
$/Kfps	6582	2094	978.8	319.6	128.7	46.50	33.56	19.13
TCO/Kfps	14,722	4411	2151	652.8	278.4	117.2	78.46	46.80
NRE K$	2,216	2,258	2,721	3,017	3,179	3,971	4,993	10,093

4.9.3 Server cost change across tech nodes

Server cost comprises die cost, power delivery components, cooling components, DRAMs and system components such as PCB. Figure 4.22 shows the server cost of the TCO-optimal servers in different technology nodes. System cost also includes fan costs, PCB cost, FPGA controller cost, and network card cost if required. These system costs stay relatively constant across technology nodes. Bitcoin and Litecoin server costs increase as tech nodes advance except for 16nm. Ultradense transistors in 16nm create thermal limits that reduce usable silicon per lane. However, TCO still improves.

Video Transcode server costs decline after 40nm. These servers are constrained by growing board space required by DRAMs required to feed each ASIC. 40nm and older have 8 ASICs per lane and 1-3 DRAMs, but TCO-optimal 28nm ASICs saturate 6 DRAMs each and only 5 ASICs fit in a lane. 16 nm ASICs rise to 9 DRAMs and only 4 ASICs fit. Still these configurations are more energy and cost efficient, despite the growing ratio of DRAM to silicon cost. In 28nm and 16nm there are 240 and 288 DRAMs per server respectively, compared to

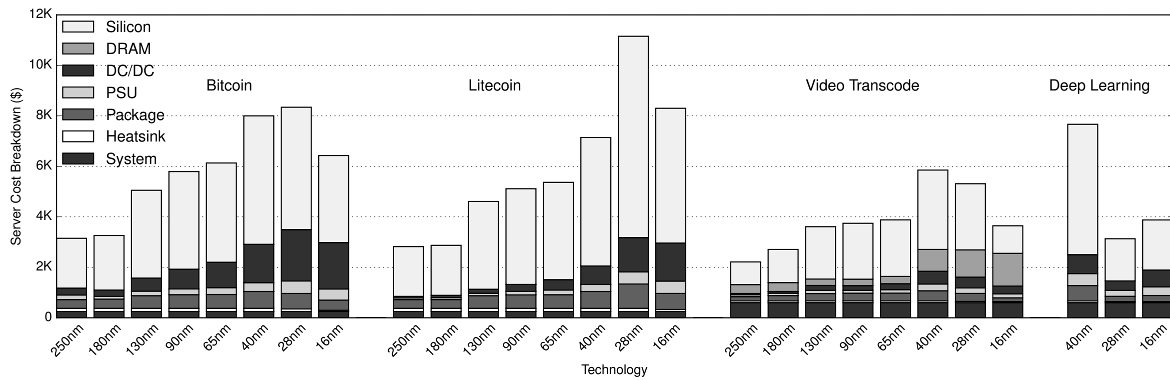


Figure 4.22: **Total server cost and relation of its components change across technology for TCO-optimal servers.** Server cost increases as more advanced technology nodes are used, except 16nm where power density becomes the limiting factor. Silicon is the dominant factor in server cost and newer nodes increase silicon utilization despite rising wafer costs. Package and power delivery components cost increase with tech node and remaining system costs stay relatively constant.

192 DRAMs in 40nm. Designs in 130nm, 90nm and 65nm cannot saturate a single DRAM's bandwidth and the DRAM cost remains constant. Finally for 250nm and 180nm, DRAM cost increases marginally due to use of SDRAM instead of LPDDR, derived by lack of DDR IP availability and NRE cost savings.

Deep Learning has large RCAs and very limited layout options which worsen the power density issue, especially in 28nm. This eliminated some TCO-optimal points due to small violations of thermal hotspots. To address this problem, we added dark silicon to spread hotspots. For example, the TCO-optimal design in 28nm uses $40mm^2$ or 15.5% extra silicon per die to be able to have more ASICs per lane. This marginal increase in silicon cost pays off because fixed parts of system cost are amortized over more RCAs per server. Since the ASIC's operating voltage in 16nm is lowered to match the SLA requirement, power density issues are mitigated and no dark silicon is necessary. This enables more silicon per lane and makes the 16nm servers cost more than 28nm ones.

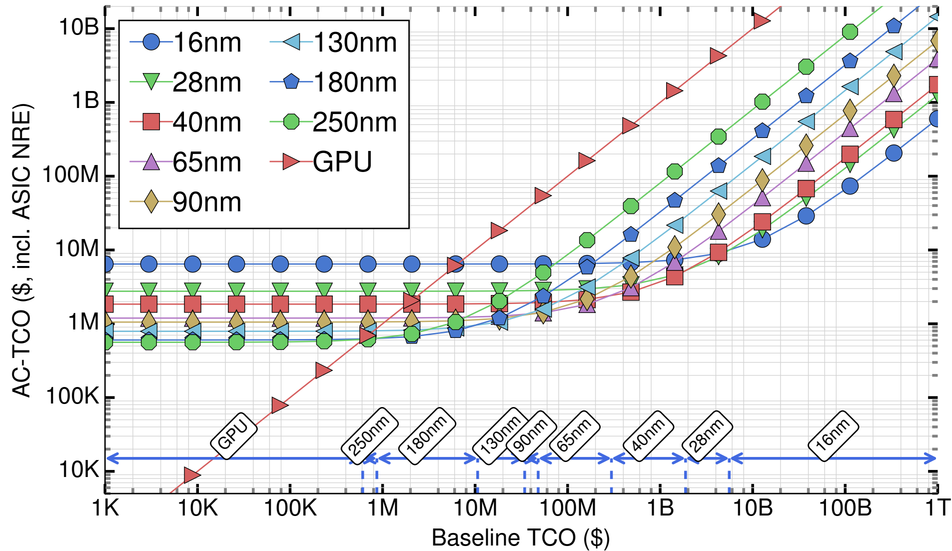


Figure 4.23: **Selecting the best tech node for Bitcoin.** Nodes start at higher total cost but eventually become the lowest TCO for a period, resulting in savings. White boxes represent the best node for a range of TCO for Bitcoin.

4.10 Finally, NRE+TCO optimal ASIC Clouds

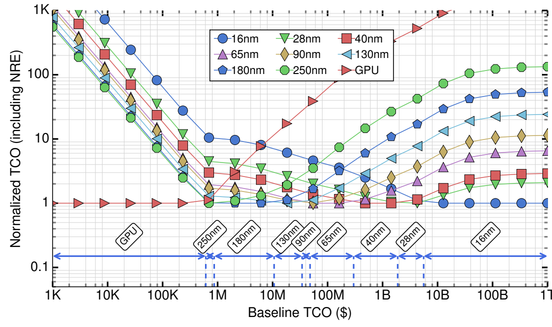
In this section, we finally combine our NRE and TCO analyses from the previous section to derive NRE+TCO optimal ASIC Clouds.

4.10.1 TCO per op/s improvement versus NRE increases

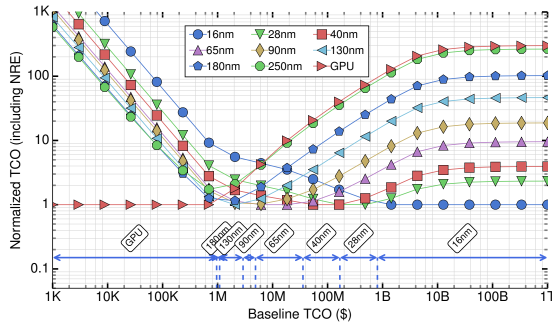
To understand how NRE and TCO per op/s scale together across nodes, Figure 4.25 shows relative NRE and TCO per op/s changes among different technology nodes. From 250nm to 65nm, TCO per op/s increases more rapidly than NRE costs. After 65nm, the slope fundamentally changes, as NRE increases more rapidly, and TCO per op/s performance improvements flatten as described in Section 4.9.1.

4.10.2 Optimal nodes given pre-/post- ASIC Cloud TCO

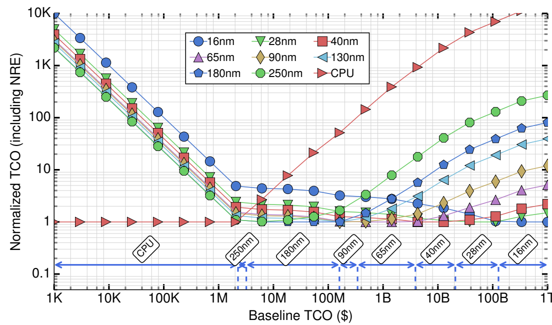
As Cloud application demand increases, the baseline TCO spent on non-ASIC servers increase and creates the opportunity of going towards ASIC servers, based on the Two-for-Two



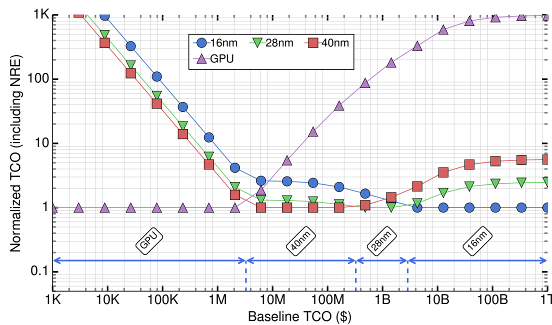
(a) Bitcoin



(b) Litecoin



(c) Video Transcode



(d) Deep Learning

Figure 4.24: **For these applications, cutting edge node technologies become TCO-optimal only for extreme-scale ASIC Clouds.** For each total cost value, the best server scenario is set to 1 and values for other scenarios represent the ratio of total money spent. After the break-even point with baseline system, there is an enormous money saving opportunity.

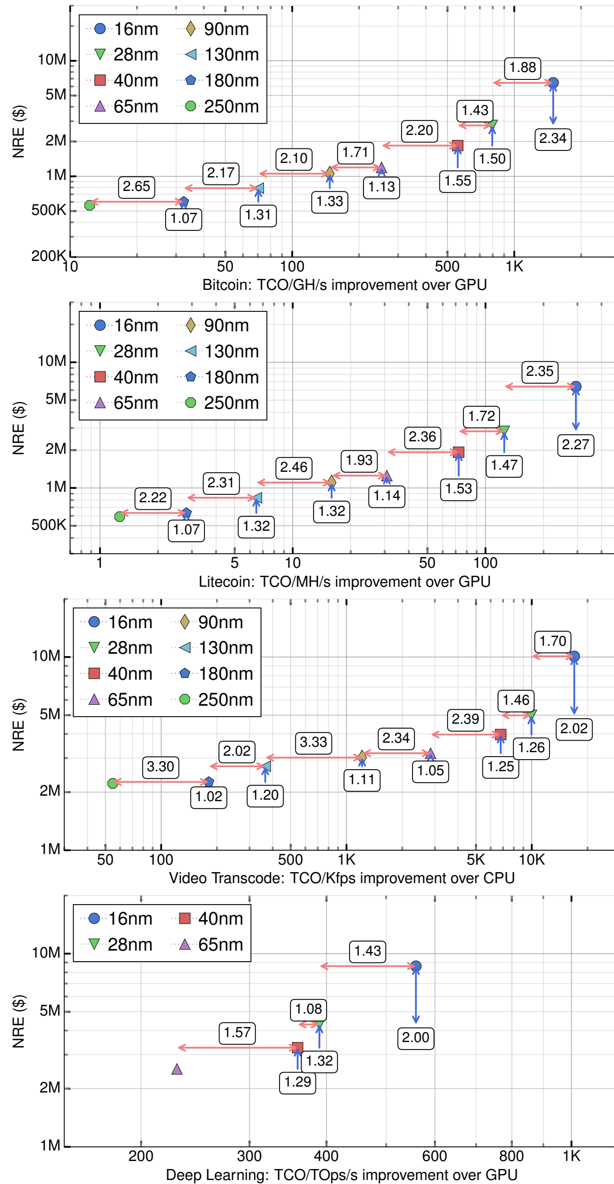


Figure 4.25: **Comparing marginal NRE and TCO per op/s improvements for each node.** Summary of NRE for different technology nodes versus TCO improvement over baseline.

rule. Figure 4.23 presents total cost for a variety of ASIC Cloud implementations of Bitcoin across tech nodes. The arrows indicate ranges of optimality for an ASIC Cloud implemented in a given node. For example, when TCO of using GPU servers exceeds $610K, 250nm becomes the least expensive option. Similarly, when the TCO reaches $867K, then 180nm becomes the least expensive option, and so on to $1.9B for 28nm and $5.6B for 16nm.

To compare nodes, Figure 4.24 shows the normalized version of Figure 4.23 where for

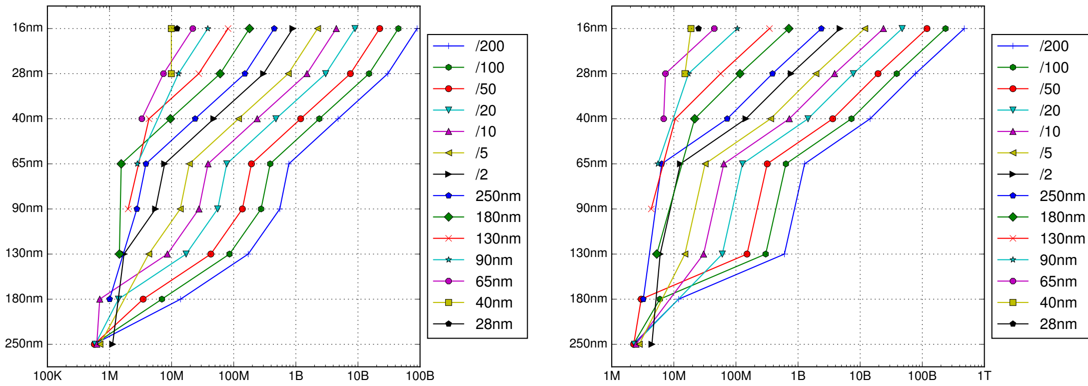


Figure 4.26: **Optimal node selection based on TCO (X axis) and on *tech parity node* (shown in key): the tech node closest to the pre-accelerated version's TCO per op/s.** Applications with small IP NRE (like Bitcoin) are represented on left and medium amounts of IP (like Video Transcode) on right. Parity nodes of /*N* indicate $N\times$ the TCO per op/s of 250nm.

each pre-ASIC Baseline TCO, we divide by TCO of the best technology node. The arrows in these graphs indicate, given an input TCO, what node should be targeted to minimize TCO including NRE.

For example, for Bitcoin, Litecoin, and Video Transcode, 180nm is optimal for small TCO's from $860K-$10.6M, $960K-$1.1M, and $3.2M-$160M respectively. Deep Learning's SLA requires >= 40nm, which is optimal from $3M to $326M. 130nm and 90nm have narrow applicability.

4.10.3 Picking the node

A company can simply plug in their forecasts for the demand and baseline TCO of the application to determine what node to use to minimize NRE+TCO. Accelerator researchers have less clarity on what TCO to use for their proposed ASIC Cloud. They could estimate the application's demand in a datacenter, just like the company, which would allow them to select a node. However, in some cases, these parameters are difficult to estimate because the application is emerging and there is not yet demand.

4.10.4 Advanced nodes like 16 nm not always better

Our data suggests that using the latest node (e.g. 16nm) for an emerging datacenter accelerator can be a mistake. For example, our results show that 16nm was optimal only for TCOs starting at $805M for Litecoin and reaching a geomean of $6.36B across all four applications. Effectively, by choosing an advanced node to do a study, a researcher is setting too high of an NRE on the technology, preventing a prospective company or investor from adopting the technology. Rather, the optimal node must provide *just enough* TCO improvement over the baseline. Moreover, reduced NREs allow an ASIC Cloud to be more agile, updating ASICs more frequently to track evolving software.

4.10.5 Tech parity nodes: getting to *just enough*

To address this issue, we introduce the *tech parity node*, the technology node at which an ASIC would have similar TCO per op/s to the best alternative. Using this formalism, and knowing the estimated TCO of the workload, Figure 4.26 gives the target node that best reduces TCO+NRE. For example, for a low-IP-NRE Bitcoin-like app, if the parity node is 250nm (key), and the emerging computation has a $25M TCO (x-axis), then 40nm would be a reasonable target node (y-axis).

4.11 Summary

We proposed ASIC Clouds, a new class of cloud for planet-scale computation architected around pervasive specialization from the ASIC to the ASIC Cloud Server to the ASIC datacenter. We examined both the architectural tradeoffs and Non-Recurring Engineering costs for ASIC Clouds, and applied the results to four types of ASIC Clouds. We believe that the analysis and optimizations in this section will speed the development of new classes of ASIC Clouds.

4.12 Acknowledgments

Chapter 4 contains reprints of I. Magaki, M. Khazraee, L. V Gutierrez, M. B. Taylor, "ASIC clouds: specializing the datacenter", *ISCA*, 2016, and also M. Khazraee, L. Zhang, L. V Gutierrez, M. B. Taylor, "Moonwalk: NRE optimization in ASIC Clouds, or, accelerators will use old silicon", *ASPLOS*, 2017. The dissertation author is the primary author of these papers.

Chapter 5

FPGA SmartNICs for In-Network Compute

Recall that network bandwidth is rapidly increasing, while host CPU performance has mostly plateaued. The result is, hitting line rate at and above 40 Gbps requires networked application developers to offload computation from host CPUs to hardware accelerators in SmartNICs. For common networking application functions (e.g., crypto and packet matching), developers have been able to use hardware accelerators in System-on-Chip (SoC) SmartNICs to hit line rate. These platforms allow software developers to directly port their code to run on multi-core processors on the SoCs [114, 90].

However, as link speeds continue to increase, the generic accelerators on SoC SmartNICs will no longer be sufficient to hit line rate. Application developers need to develop custom high-performance hardware accelerators that can be deployed on FPGA-based SmartNICs. This is why FPGA SmartNICs have become a standard component in many production cloud datacenters: Microsoft has deployed FPGAs-based SmartNICs on thousands of hosts [28, 42], and Amazon offers hosts connected to FPGA SmartNICs in their cloud.

In this chapter we present *Dastgāh*[1], a software-defined framework for developing networked applications on FPGA SmartNICs that can operate at *200 Gbps* and beyond. From a software developer's perspective, Dastgāh is similar to programming a networked application on a host CPU (e.g., with DPDK), or on a multi-core SmartNIC. Packets are processed by *Gusheh*, reconfigurable units in the FPGA that contain a CPU and one or more hardware accelerators. The CPUs are built with the 32-bit RISC-V instruction set, so they are programmable with existing C toolchains. Software running on these CPUs can call hardware accelerators to process a packet through memory-mapped IO; whichever is the most clear and efficient interface for the hardware developer to implement their functional unit.

By using Dastgāh, software developers can reconfigure the hardware in their Gusheh at runtime to select hardware accelerators specific to their workload, or one that matches updated requirements (e.g., IDS rulesets). Hardware developers can also focus on designing a highly-parallel hardware accelerator; they do not need to implement the application semantics. This allows them to reuse accelerator designs across different applications. Fortunately, it also makes this framework complementary to prior work that provides tools to automatically generate functional units for packet processing from software into hardware (e.g., ClickNP [86] and P4FPGA [150]). In addition, there were several recent work in academic avenues on designing more flexible and performant packet schedulers [87, 7, 135, 136, 127]. Dastgāh provides a platform to test new scheduler designs in hardware without requiring to implement a full system from scratch. Dastgāh allocates a region in FPGA for scheduler module and provides necessary abstractions for each Gusheh, and test can be performed within the connected host and RISCV processors in Gushehs.

There are two primary insights that makes Dastgāh achieve high data rates and be further scalable. The first insight is the concept of packet in networking. One challenge for designing a system of parallel functional units is the distribution among them. But packets are self-contained

[1]A system of traditional Persian music that develops around melodies, called "Gusheh".

and can be distributed among parallel compute units based on well-known scheduling and traffic engineering techniques. Another feature of packet is the distinction between header and payload, which enabled us to develop a new hybrid memory system to benefit from larger memory blocks on more recent FPGAs. The second insight is the fact that most of the processing time is spend in the hardware accelerators, and not in the decision making and control. This opens up the opportunity to take some performance hit by doing the control in software, with only marginal impact on overall system latency and throughput. Considering clock frequency of 250 MHz and 16 Gushehs in our implementation, there are about 160 clock cycles available to process each packet at 25 Mpps. This is the packet rate for average packet size of 1000 bytes at 200 Gbps. Also there can be multiple accelerators per Gusheh working in parallel which further increases the number of allowed cycles per packet by number of accelerators per Gusheh, e.g. 8 Regular Expression engines per Gusheh in our case study translates to average budget of 1280 cycles to process each packet.

Building on these insights, we make two primary contributions:

1. A framework for developing 200-Gbps hardware accelerated applications. We present Dastgāh, an open-source framework that makes it possible to hit line rate at 200 Gbps for accelerated applications by only writing C-code that calls functions on custom hardware accelerators. We implement and evaluate Dastgāh on the Xilinx XCVU9P FPGA. We demonstrate that for packet size of 128 bytes, Dastgāh can process (and generate) packets at 200 Gbps, and even for the worst case scanrio of 65 bytes, Dastgāh can process (and generate) packets at 173 Gbps. Also, at the smallest packet size Dastgāh framework adds only $\sim$0.7 μsec ($\sim$7.1 μsec worst case for jumbo packets) of latency to the packet processing. We design Dastgāh in Verilog with all open-source IP cores. We will release it as open source for the community to use at the time of publication.

2. An abstraction for software-hardware collaboration. We design Gusheh, a new hardware abstraction that makes it feasible for software and hardware developers to collaboratively

develop hardware-accelerated networked applications. This abstraction makes it possible to implement flexible, modular, and debuggable software and hardware. We also describe novel hardware subsystems that make it possible to use this abstraction at 200 Gbps, including: (1) a core-to-accelerator hybrid shared-memory subsystem, and (2) a packet subsystem that supports 32 Gbps simultaneous incoming and outgoing data rate per Gusheh. We are able to instantiate 16 Gushehs running at 250 MHz providing a 16-fold increase in clock cycles available to process each packet. We also demonstrate how Gushehs are used to implement two case studies of hardware accelerators that used to require close hardware/software developer interaction (each was implemented in less than 5 days): (1) a software-defined regex lookup for an IDS, and (2) a software-defined matching engine similar to Microsoft's Generic Flow Tables.

The outline of this chapter is as follows: we first motivate the need to improve development of hardware and software for FPGA SmartNICs (Section 5.1). Then we describe the design of Gusheh (Section 5.2), followed by the design of Dastgāh (Section 5.3). Next we outline the implementation (Section 5.4) and evaluate the system at 200 Gbps (Section 5.5). Finally we present two case studies (Section 5.6).

5.1 Motivation and design goals

In this section, we motivate the need for a software-defined framework for FPGA SmartNICs. We begin with a motivating example of adding hardware acceleration to a popular networked application: an Intrusion Detection System (IDS). This example reveals the challenges with the current procedure of jointly developing hardware and software for a networked application. Based on this set of challenges, we describe the design goals for our framework, Dastgāh, that simplifies developing hardware accelerators for networked applications.

5.1.1 Motivating example: IDS acceleration

Intrusion Detection Systems (IDS) identify suspicious network traffic by monitoring nearly all of the traffic at the border of an organization's network. If any of the traffic matches one of the IDS's rulesets, it will raise an alert to notify network operators. Most of the popular IDS systems are software packages that run on commodity servers, including Zeek, Snort, and Suricata. The flexibility of software development is important for IDSes because they need to adapt to changes in how network attacks are carried out. Often, an IDS will need to update its set of rules—a table of patterns that specifies what an attack looks like like in network traffic—in order to identify a new attack.

As bandwidth increases at the border of the network, operators have needed to divide traffic across clusters of servers running the IDS software in order to keep up with line rate [26]. However, this introduces challenges with load balancing and aggregating results [66] across the cluster. Alternatively, developers can run an IDS at line rate on one machine by offloading part of the IDS to custom hardware accelerators on FPGAs [151, 100]. These accelerators significantly speed up the most computationally expensive part of the IDS: checking to see if traffic matches the rulesets. Unfortunately, today an IDS developer has to convert a significant fraction of their application software to hardware to add a network accelerator and hit line rate on an FPGA SmartNIC.

Challenges with developing a hardware accelerated IDS

The primary portion of IDS software (generally C code) that needs to operate at line rate is as follows: (1) parsing the traffic, selecting the ruleset that needs to be matched on the traffic (or discarding the traffic if there is no ruleset for it), (2) checking the ruleset to see if the traffic matches it, and (3) generating an alert if there is a match with one or more rules. Figure 5.1 depicts how this this processing flow is accelerated from software on a host CPU to hardware running on an FPGA. As shown in this figure, offloading the computationally intensive

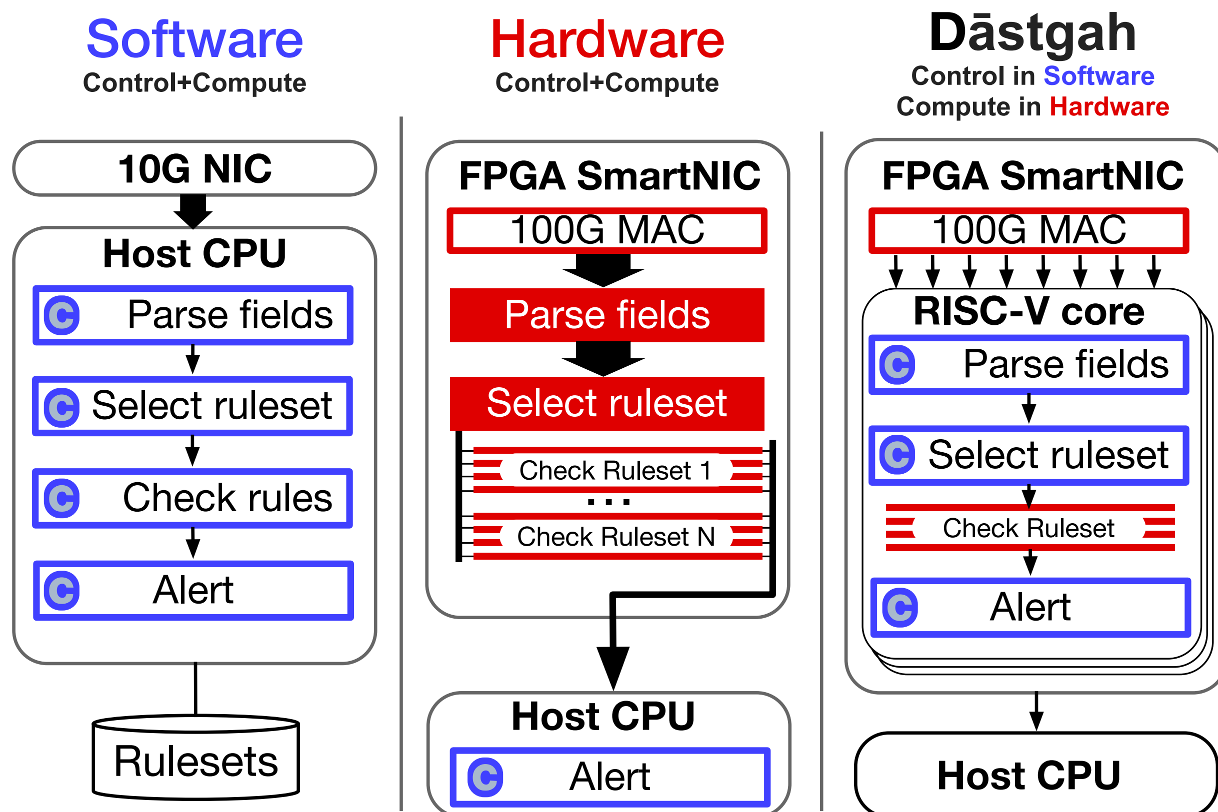


Figure 5.1: Implementing hardware acceleration for a software IDS with Dastgāh only requires porting C code and implementing custom accelerators.

function of checking the rulesets also requires offloading other less computationally intensive tasks, including: parsing and selecting the ruleset (or dropping the packet). The reason for this is, these functions must be executed at the full line rate of traffic, and sending that traffic to the host CPU first is not feasible. Sending packets back and forth between the host and FPGA multiplies the load on PCIe which already is limited to less than 100 Gbps due to OS and PCIe limitations. Unfortunately, integrating these less computationally intensive control functions into the hardware pipeline introduces several challenges:

Limited flexibility: Implementing the parsing and ruleset selection logic in hardware means that the hardware must be redesigned frequently. Namely, every time that a new protocol is being added to the ruleset. Redesigning the hardware to integrate even the smallest protocol change (e.g., parsing a new field) may cause the hardware developers to have to significantly

modify their logic in order to maintain the ability to hit line rate. These development cycles are incredibly slow: synthesising hardware for an FPGA (analogous to compiling software) often takes hours for a single run. Also it might fail meeting timing constraints and hence require more challenging updates to the hardware design. These issues also come up when modifying and adding rules to the hardware rule checker. This makes it difficult for developers to update the IDS to add new rules as attacks evolve.

Limited debuggability: Debugging important functions such as these is extremely onerous when they are implemented in hardware. There is no ability to add breakpoints and view state at runtime; instead, hardware developers need to add debugging-specific hardware modules to view what is going on in in the parser and ruleset selection logic. Adding this hardware is also likely to undermine previously met timing constraints, and also make it harder for the design to run at line rate. This results in a situation that testing may only be able to be performed on processing-intensive simulations, or at slower-than-actual line rate.

Software-defined hardware development

In this work we address these challenges by introducing a new hardware element: a set of reconfigurable RISC-V cores and partially reconfigurable hardware blocks. In Figure 5.1 we show what a hardware accelerated IDS would look like in Dastgāh. In this framework, the functions for parsing and selecting rulesets can be implemented in software on RISC-V cores instantiated on an FPGA SmartNIC, but we can still hit line rate at 200 Gbps. This software will call the hardware accelerators for checking if the traffic matches the rulesets.

5.1.2 Design goals for Dastgāh

In order to address these challenges, we design an FPGA SmartNIC development framework that can enable software developers to create functions that need flexibility and visibility, and separately hardware developers can develop accelerators for functions that are compute-

intensive. Specifically, the overarching goal of our framework, Dastgāh, is to improve the design process by separating development of control software and custom application-specific hardware accelerators; therefore, creating a framework that has the following properties:

Flexibility: Software developers should be able to implement control and modify functions by writing software. They should be able to update these control functions by deploying new software on the FPGA, even at runtime. Hardware developers should also be able to deploy new hardware accelerators on the FPGA, even at runtime.

Modularity: Software should have a well-defined interface to call hardware accelerators, and hardware accelerators should have a well-defined interface to connect to CPUs. This makes it possible for hardware developers to collaborate with software developers more effectively. As software evolves, hardware will not need to change, and as hardware is improved, software can immediately receive the performance benefit.

Debuggability: Software developers should be able to test how their software works with the hardware accelerators without deploying on an FPGA SmartNIC and running it on actual traffic. Hardware developers should be able to identify problems with their hardware at runtime by monitoring the state of accelerators and raising faults to be handled gracefully by the software when there are hardware bugs.

Performance: Achieving all of these goals should not significantly affect the performance benefits of using hardware accelerators. Specifically, we target 200 Gbps link bandwidth, as that is the more than what host CPUs can hit today.

5.2 Abstraction

In this section, we introduce Gusheh, the hardware abstraction for software-hardware collaboration which achieves the goals of flexibility, modularity, debuggability, and performance

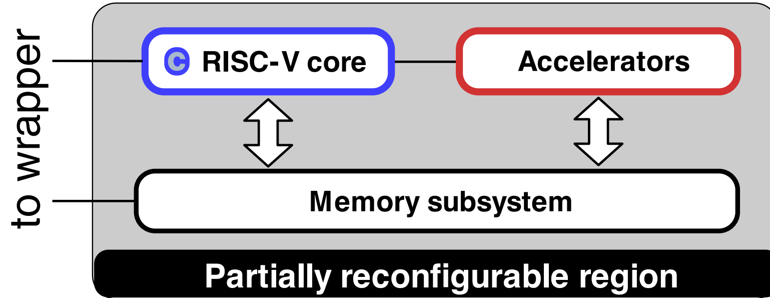


Figure 5.2: Gusheh is FPGA packet-processing hardware that contains programmable software and reconfigurable hardware.

for networked application development. Then we describe the memory subsystem inside each Gusheh, which is a critical component to achieve these goals.

5.2.1 Modular: Software and hardware collaboration

Our primary contribution is to demonstrate we can achieve the desired goals by connecting a low-end RISC-V core to hardware accelerators with a novel memory subsystem. We call this abstraction a Gusheh, and Figure 5.2 shows what components it consists of. The RISC-V core controls packet processing by (1) preparing the packets for the hardware accelerators, (2) performing the hardware-accelerated processing, and (3) transmitting a packet containing results or updating state and statistics counters within and across different Gusheh. The insight behind our idea is that most of the packet-processing cycles are spent in the hardware accelerators, and only a short time is spent deciding how to run the accelerators and outputting the result. This design allows us to separate the control and compute functions in the system into software (slow-speed control) and hardware (high-speed compute) components. We select the RISC-V architecture for the Gusheh because RISC-V is an open-source ISA with a well-supported toolchain (e.g., gcc) and several reconfigurable open-source implementations. The advantage of using an open-source core is, in addition to adding hardware accelerators, hardware developers can also improve performance for the software interface by adding new instructions to the RISC-V core. In our evaluation we demonstrate that this split does not significantly impact performance, and we are

still able to hit line rate on an 100-Gbps FPGA SmartNIC (Section 5.5).

Because the RISC-V core and the accelerators share memory, giving a received packet to an accelerator becomes as simple as passing a pointer to its memory location. Furthermore, using a memory-mapped I/O technique, the microcontroller can set the value of some registers to be used as parameters for the accelerator. One example of a parameter is the hash key for decryption or an encryption accelerator. Setting these register values and passing the pointer to the data portion is precisely equivalent to using a function in software. On the hardware developer's side, they have a memory port to get the data and use as scratch memory, as well as few registers as input ports that are mapped to the RISC-V's I/O addresses. This is a common design for most hardware accelerators: both hardware and software are transparent to each other. This makes it possible for the software developer to abstract away hardware implementation details and change the decision-making logic without regard for the hardware accelerator.

5.2.2 Flexible: Reprogrammable software and hardware

The Gusheh is flexible because it contains both reprogrammable software and hardware. Host CPUs can write to both instruction and data memory of the RISC-V core, and also read back from their data memory. This is all handled through the memory subsystem in the Gusheh, which is connected to the host through PCIe using a DMA core in the Gusheh's wrapper. This enables software updates at runtime.

To update the hardware at runtime, we build the Gusheh mostly inside a partially reconfigurable region in the Xilinx FPGA (Figure 5.2). This enables updating part of the hardware design without stopping or interfering with the rest of the logic on the FPGA. Using this capability requires defining specific regions on the FPGA for partial reconfiguration, called PR blocks. These blocks are considered to be separate during the routing stage of the synthesis of Dastgāh to keep the reconfigurable hardware regions unused and ready for a designer to drop in new logic at runtime.

The main challenge to supporting partially reconfigurability is transitioning the state in the FPGA from a running accelerator to a new accelerator. To address this challenge, we use the RISC-V core in the Gusheh to prepare the Gusheh for reconfiguration. The reconfiguration process proceeds as follows: (1) the host CPU tells the packet subsystem (Section 5.3.2) not to send any more packets to that specific Gusheh. Then it fires an interrupt on the RISC-V in that Gusheh. The Gusheh finishes processing the current packet, then it saves its state into host memory—functionality described in Section 5.3.1—and notifies the host CPU that it is ready for reconfiguration. Now that there are no packets going to or coming from that Gusheh, we can reconfigure the Gusheh. After reconfiguring, the new RISC-V core boots up and restores its state by reading it back from host memory. If the new hardware was just an optimization and not a functionality change, the host can load the corresponding binary onto the RISC-V and set the instruction pointer and other registers to return to the last instruction on the core.

5.2.3 Debuggable: Checking hardware using software

The last goal of Dastgāh is to provide hardware and software debugging support. For software debugging, hosts can inspect the data memory of each RISC-V core and determine what caused a bug. This functionality can be used for keeping track of hardware accelerator malfunctions as well—the RISC-V core can read the accelerator's state into memory when it fails. Dastgāh also supports a preventive approach for hardware debugging, which is to jointly simulate the accelerator and the RISC-V core with the corresponding software. To achieve this goal, we develop a hardware simulation framework using the open-source MyHDL project for Python. This tool provides cycle-accurate hardware simulation. Since such a test is limited to a single Gusheh, a Python-based simulator is sufficiently fast. We provide Python libraries for generating and reading packets to and from each core. Moreover, a software developer can compile their C code and provide the binary to the simulation framework. This also makes writing test-benches more straightforward; hardware developers no longer need to write a test-bench based on the

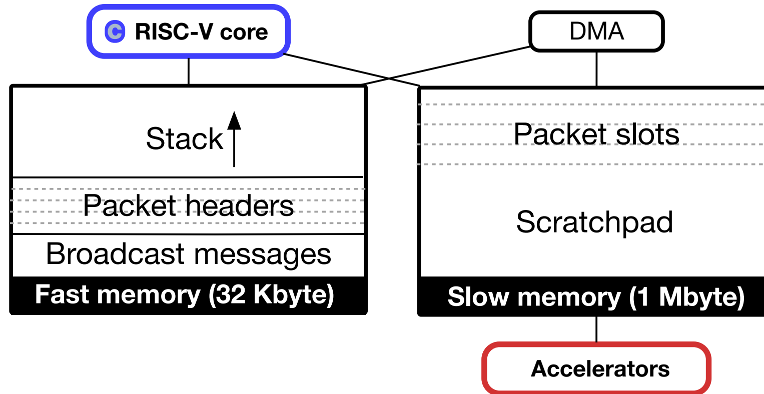


Figure 5.3: The hybrid shared-memory subsystem in each Gusheh. Cores mostly operate in fast memory but can also read payloads from slow memory. Accelerators share slow memory with cores where the DMA will automatically load packets.

provided test cases and can directly run the C code.

5.2.4 Memory subsystem

The memory subsystem plays a critical role in Gusheh's performance. As packets get larger and larger and line rates increase, being able to accommodate the packets at high throughput with low latency is more and more challenging. Fortunately, size of available memory on FPGA devices has increased which made this feasible. However, there are several challenges to achieve the desired throughput and latency. First of all, these larger memories, called Ultra-RAM, are in form of block memories with some internal line selection logic and registers that help to operate at higher clock rates. Also we need to bundle these memories to increase memory line width to achieve desired throughput, further adding logic and required registers to meet timing. These added registers adds latency to each read and can dramatically reduce system performance. Second challenge is that there are only 2 ports per memory block inside an FPGA which can cause contention and lower the throughput. We developed three design ideas to achieve the required throughput and latency.

First we realized that accelerators and RISCV core have different pattern of accessing memory. Accelerators usually read the packet data in a consecutive sequence of addresses, so

we can streamline these reads and keep up with the required throughput. However, a general purpose core usually has more random access, for example parsing the header and deciding on next read based on the last read value. This random read locations with data dependency means the core needs to stall until getting the read results back, and this can dramatically reduce core's performance.

Our second realization is that RISCV core usually only parses the header and accelerators are in charge of performing compute heavy payload processing. This opened up an opportunity for a hybrid memory system. As shown in figure 5.3 we split the memory into two parts. There is a small memory that can be accessed within single cycle, marked as Fast memory, and a large memory that has additional registers, marked as Slow memory. Next, we build our own customized Direct Memory Access (DMA) engine inside Gusheh wrapper, which is in charge of copying incoming and outgoing packets between local Gusheh's memory and switching subsystem described in Sec.5.3.1. This customized DMA copies header of incoming packet data to both Slow and Fast memory, and copies payload only to the Slow memory. Therefore, RISCV core can parse the header with very low latency, and accelerators can access the entire packet in a streaming method. Moreover, RISCV can use the remainder of Fast memory for it's local memory such as stack. Remainder of Slow memory can be used as scratchpad for accelerator, or for large look up table which are accessed sparsely from the RISCV core. In other words, we can benefit from available large memories on new FPGAs without paying any penalty for random access from RISCV cores.

Our final design idea addresses the contention issue. Fortunately, in our hybrid model RISCV core mostly accesses the Fast memory and rarely accesses the Slow memory, and accelerators only access the Slow memory. So we allocated an exclusive port of Fast memory to RISCV and an exclusive port of Slow memory to accelerators. DMA engine has exclusive access to the other port of Fast memory, which contains mostly packet data, alongside sparse broadcast messages explained in Sec. 5.3.3. The other port of Slow memory is shared by DMA

engine for packet data, and RISCV for sparse accesses. Another potential contention is between incoming and outgoing traffic if they target the same memory block. Since each packet is written to consecutive set of addresses, we can benefit from banking technique where odd and even memory line addresses are mapped to different memory blocks. If there is a contention between an incoming and outgoing packet, it would last only single cycle which one of them stalls for the other one, and all further accesses are interleaved. This also makes the available memory bandwidth to accelerators double the packet bandwidth to each Gusheh.

Note that there is also instruction memory for each core, where DMA engine only writes to this memory and core only reads from it, with no other component involved. Therefore, we gave an inclusive port to each of them and there is no contention or need for banking.

5.3 System design

In this section we describe how we implement Gushesh abstraction on FPGA SmartNICs and achieve line rates of 200 Gbps. This includes the switching subsystem that connects Gushehs to the rest of system, the packet subsystem that control packet distribution to Gushehs, and finally messaging subsystem used for communication among Gushehs. Alongside describing the functionality of each subsystem, we elaborate how limiting the scope to only networked applications helped simplify the design and make it scalable and feasible for FPGAs.

5.3.1 Switching subsystem

As stated earlier due to lower clock frequency and limited performance of each RISC-V core in our system we need to replicate them and use them in parallel. However, designing an efficient switching system at 200 Gbps link speeds is very challenging. We need to remove dependencies inside the switching system as much as possible, and also make it simple to scale efficiently for high throughputs. On top of that, there are resource and timing restrictions for

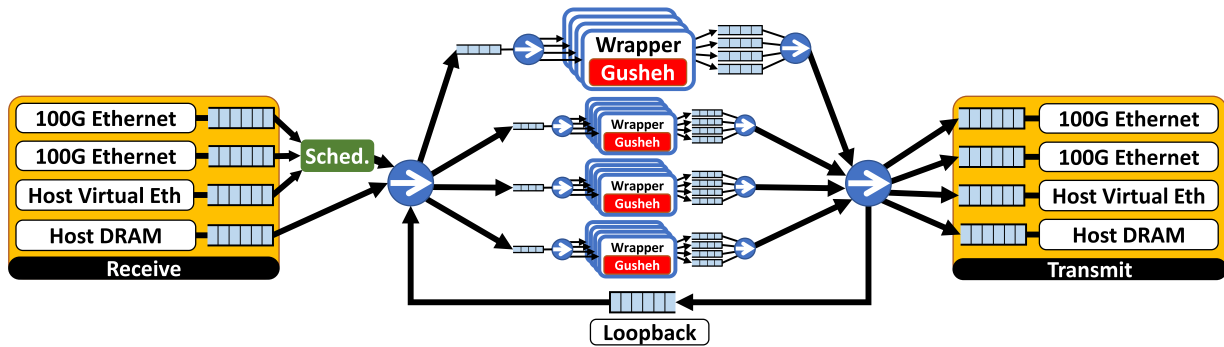


Figure 5.4: Switching subsystem overview.

FPGAs which makes it further complicated.

Figure 5.4 shows the implemented data path in Dastgāh. Data coming from physical or virtual Ethernet interfaces goes to the scheduler and are assigned a destination. Then this data goes to the RX side of switching infrastructure to get to the destination Gusheh, and then Gusheh can process the packet. When packet needs to be sent out, it goes over the TX side of switching infrastructure to one of the Ethernet interfaces. All of the paths are single directional and there is no dependency from switching subsystem between receive (RX) and transmit (TX) side. Note that there are two other possible sources and sinks for the data: 1) Host DRAM represents the communication among Gushehs and host DRAM which is also performed as packets, and 2) Loopback port which is used when a Gusheh wants to send a full packet to another Gusheh, further described in Sec.5.3.3. These two interfaces typically carry much less traffic than the Ethernet interfaces, so the same switching infrastructure is used for all of them to reduce resource utilization, without sacrificing throughout of Ethernet interfaces.

As seen in Figure 5.4, the switching is performed in two stages, first among 4 clusters, and then among 4 Gushehs. As load is distributed among parallel gushehs, each Gusheh gets less incoming rate than the Ethernet interface. If we use single stage switching system, we require to keep all the links at full throughput not to limit the source interfaces, and do bit-width conversion right before each Gusheh, which results in very inefficient resource consumption. Another approach is to do the bit-conversion at the beginning of each cluster and then use a switch

running at lower throughput and hence save resources. However, doing early bit-conversion limits the performance of each cluster to single low-throughput link. Instead we implemented a new switch for each cluster that has full throughput on one side, and 4 links running at 1/4th throughput on the other side. We used separate FIFOs for each incoming link inside this switch, which makes it fully non-blocking, that is each cluster can run at full-throughput and each Gusheh can run at 1/4th the full-throughput. These Switches are shown in the middle of Figure 5.4 with the incoming link FIFO before them. Main role of these FIFOs are bit-conversion without blocking the other interfaces.

Finally, there is arbitration in each stage. Based on our implemented scheduling scheme, described in Sec. 5.3.2, we chose round-robin-priority arbitration in each switching stage. That is each time a packet is sent most priority is given to the next destination and the lowest to the one that just sent a packet. Note that this scheme can be changed to priority per cluster or first stage switch with simply changing a parameter and resynthesizing the hardware. Using basic round-robin increases the chance of head-of-line blocking and hence there are FIFOs per incmoing and outgoing interface to be able to absorb bursts. Bit-width conversion FIFOs in each cluster can also help mitigate head-of-line blocking. Note that we implemented all of the FIFOs as packet FIFOs, meaning they would send out a packet when it is fully received. This increases the minimum latency of the system, evaluated in Sec. 5.5.3, but removes the dependency among consecutive stages.

5.3.2 Packet subsystem

The simple switching infrastructure described in the previous subsection is not achievable without some abstractions based on existence of packets in a networking application. We use concept of packet to separate memory addressing from switching subsystem. We benefit from the well-known techniques of memory reservation and descriptors to abstract the way memory addressing as slot number. Similarly, we implement communication between host DRAM and

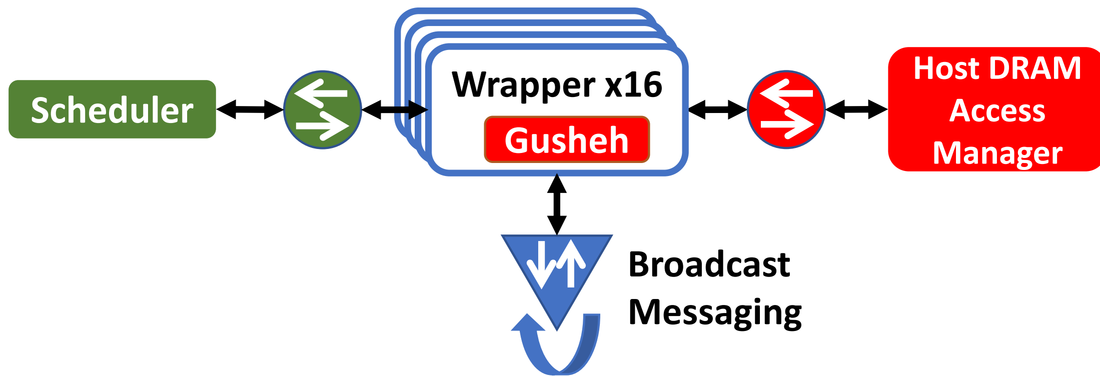


Figure 5.5: Packet subsystem overview.

Gushehs also as packets, with different slot number, aka DRAM tags. Wrapper implements address handling and interfacing with the core, as well as communicating to the scheduler and the host DRAM access manager. In other words, we split the control functionality into a central part, such as the scheduler, and a distributed part as core wrappers, to make it more scalable.

We need separate control channels for messages among Gushehs and packet scheduler, as well as request messages to host DRAM access manager for read or write requests from Gusheh or host. These control channels are shown in Figure 5.5. The switches are two stage switches similar to data-flow switches described in the previous subsection, but there is no bit-width conversion and simple switches are used. Finally, wrappers duty is to do packet to memory copy and vice versa, as well as talking to RISCV core through descriptors, both for data and request messages.

The packet flow throughout the system is as the following. At boot software allocates some slots for packets and notifies the central scheduler about number of slots and their size. When a packet arrives wrapper notifies the core as a descriptor. When core wants to send out a packet it has two options: it can ask the wrapper to send it out directly, or it can tell the scheduler which slot is ready to be sent out and scheduler sends the transmit command to the wrapper. In both cases wrapper notifies the scheduler about slot being freed after it is sent out.

We implemented a fully balancing scheduler for analysis in this chapter, where we keep track of number of slots per core and assign a new packet to the core with most available slots. This scheduler would test the infrastructure at maximum capacity by distributing to set

of homogeneous functional units, instead of being bottlenecked due to imperfect balancing algorithm. This is a simple example which is sufficient for many workloads, and it does not limit researchers to try their own scheduler within Dastgāh. There has been several designs in the literature for scheduling, such as the P4-based forwarding [[150]] or time-accurate schedulers [[130],[135],[128]]. We believe this opens up the opportunity to plug-in such designs in our system, and cleanly separate compute acceleration and scheduler.

5.3.3 Messaging subsystem

Although networking provides the abstraction of packets, there could be dependency among packets of a flow. This makes design of parallel systems for flow level systems very challenging. Multi-core SoC-based SMARTNICs such as Mellanox Bluefield, Broadcomm Stingray, Cavium Liquid IO , and Netronome Agilio use the well-known multi-level caching system to share the state among different cores. However, this caching system is already becoming the bottleneck for these systems at 40 Gbps. They suggest directly copying incoming packet to the internal cache of each core to achieve high throughput. Due to lower clock rate and limited memory and routing restrictions caching becomes more challenging and more impractical for FPGA NICs.

Our insight is most of the message passing in networked applications can be categorized in two modes: (1) copying a packet or portion of memory to another core, or (2) a short message to update other cores. In a cached system both of these can be implemented through coherency. For the first scenario we simply provided a loopback module next to network interfaces. Cores can use the provided interface to ask the scheduler for a packet slot of the destination core, and the packet can be transmitted using the same distribution system.

Note that inter-core packet passing messaging system can be used to implement a pipeline among cores. However, using fewer cores and implementing the pipeline inside each core typically would be more efficient. The exception is when cores are heterogeneous and have

different accelerators and capabilities. Specially if most packets can be processed with a single core and the scheduler can assign it properly, this type of messages can handle the other cases when the assignment is incorrect or for few case that there is need for other core's capabilities.

For the second type of messaging, in cached systems a write to a location in shared memory would evict that value. Later when another core requires that value it gets the updated value. We simplified this to a broadcast system, where portion of memory is semi-coherent. That is a write to this portion would be propagated to all the other cores, and they would receive the message at the exact same time. This system is shown in Figure 5.5. This method is sufficient for several applications. For example in a firewall or IDS application when one Gusheh receives first packet of a flow, it can broadcast the decision for rest of that flow to the other Gushehs and such messages are not that often to cause congestion on this system.

5.4 Implementation

In our implementation we used Xilinx Virtex UltraScale+ FPGA VCU1525 board which has a XCVU9P FPGA chip on it. Figure 5.6 shows layout of the implementation. There are 16 Gushehs where each of them has a unique PR allocation as well as a wrapper next to it for communicating to the rest of the system. There is a reserved area for scheduler, labeled Scheduler_block. Note that our current implementation requires very few resources, and the rest of reserved area is empty for user's scheduler. Scheduler_block is not partially reconfigurable, since reconfiguring scheduler means blocking the traffic during reconfiguration, as opposed to Gushehs which can be taken offline one at a time. If necessary for a future implementation scheduler can also become partially reconfigurable if there is a replication of it and we switch between them.

There are two main components for the incoming and outgoing interfaces. First, Physical interfaces are connected the MAC modules followed by interface FIFOs, colored in Green.

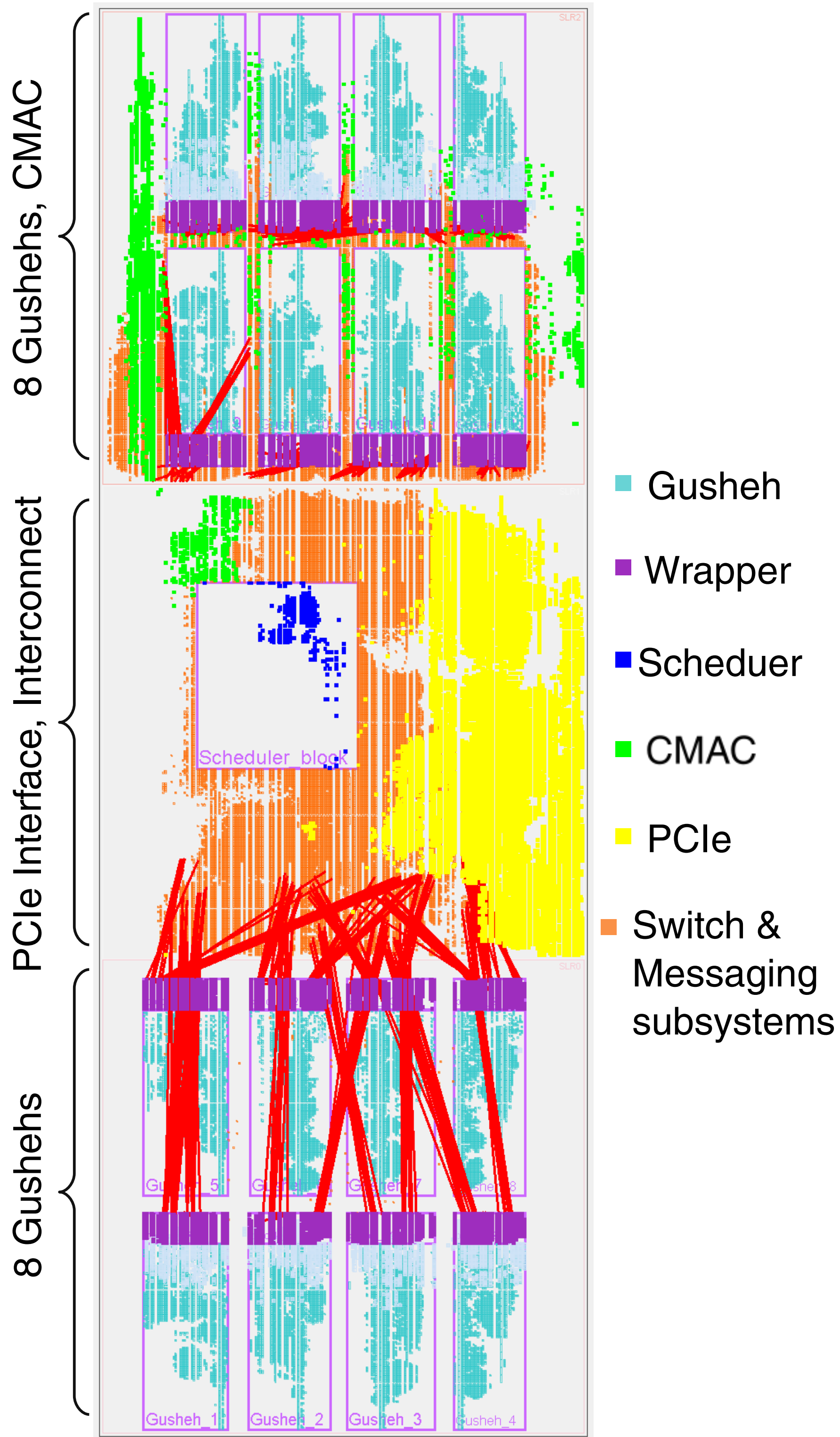


Figure 5.6: Layout of components in XCVU9P FPGA.

Second, there are PCIe modules for connection to host for control, accessing DRAM and a virtual Ethernet interface, together colored in yellow. The main component of the PCIe modules is a multi-queue PCIe DMA engine. This DMA subsystem includes a driver for the Linux networking stack, enabling operation as a NIC, aka a virtual interface. For this purpose, we integrated an open-source 100 Gbps virtual network interface[cite corundum] in our system.

The other main components of system are switching and messaging subsystems. They are marked as orange representing the switching and messaging subsystems. Note that each wrapper is connected to the switching subsystem, red lines represents few of these connections. Wider switches in 5.4 are implemented as 512-bits wide and the narrower switches as 129-bits wide, which provide max throughput of 128 Gbps and 32 Gbps respectively. There is some overhead for arbitration that reduces the performance from their maximum values, but remains above 100 Gbps for the wider interfaces.

Table 5.1: Dastgāh resource utilization, without any accelerators.

Component	LUTs	Registers	BRAM	URAM
Single Gusheh	4486 (0.4%)	3831 (0.2%)	24 (1.1%)	32 (3.3%)
Remaining (PR)	22153 (1.9%)	49449 (2.1%)	0	0
Scheduler	1822 (0.2%)	771 (0.0%)	0	0
Remaining	54720 (4.6%)	109440 (4.6%)	96 (4.4%)	64 (6.6%)
Single wrapper	2852 (0.2%)	2933 (0.1%)	0	0
CMAC	8735 (0.7%)	17753 (0.8%)	82 (3.8%)	0
PCIe	47077 (4.0%)	72657 (3.1%)	169 (7.8%)	0
Switching	115270 (9.8%)	153890 (6.5%)	51 (2.4%)	32 (3.3%)
Complete design	290334 (24.6%)	353343 (14.9%)	686 (31.8%)	544 (56.7%)
VU9P device	1182240	2364480	2160	960

Table 5.1 shows the utilization break down of the mentioned main blocks, as well as average resource utilization per Gusheh, without any accelerators in them. The average remaining resources per PR block for each Gusheh as well as the remaining resources in the reserved scheduler_block are also shown. We were able to meet timing at 250 MHz for the design. The only hard IP blocks are the network interface SERDES, PCIe and Gigabit CMAC, and the rest are our developed open source IP.

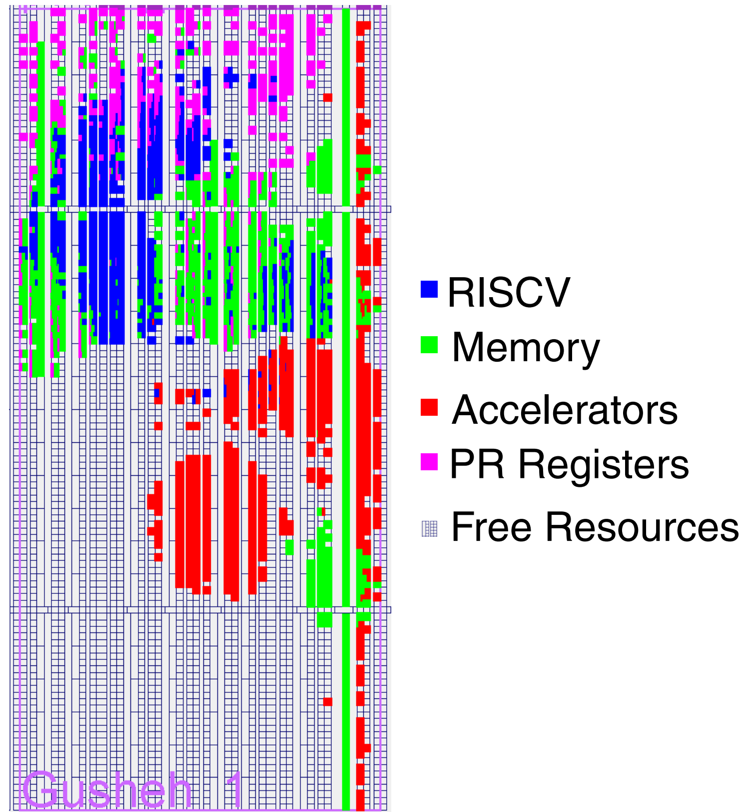


Figure 5.7: Layout inside of a Gusheh.

Figure 5.7 shows a single Gusheh in more details. RISCV core is shown in blue, memory subsystem in green and the accelerators, described in Sec.5.6, are marked as red. Finally, when using partial reconfiguration we need to add a set of register to help meeting timing constraints, shown in pink. There are many remaining resources in each Gusheh, shown by the corresponding unused primitive cells.

Table 5.1 shows the break down of average utilization inside each Gusheh. As you can see the RISCV core only consumes 6.6% of LUTs and 1.9% of registers, which we think is an acceptable resource overhead for the added functionality it brings. The memory subsystem is the largest sub-module in each Gusheh, since it has to connect several large memory units and perform arbitration among them. Also to improve system performance and avoiding blockage by different components, we implemented all memories as two banks, and there are small FIFOs for some of the ports. Accelerators have a separate port to the packet memories, where accelerators manager

Table 5.2: Gusheh resource utilization

Component	LUTs	Registers	BRAM	URAM
RISCV core	1755 (6.6%)	997 (1.9%)	0	0
Mem. subsystem	2248 (8.4%)	862 (1.6%)	16 (66.6%)	32 (100%)
Accel. manager	864 (3.2%)	374 (0.7%)	0	0
Hash accel.	1029 (3.9%)	296 (0.6%)	0	0
Regex accel.	57 (0.2%)	86 (0.2%)	0	0
PR registers	625 (2.3%)	1972 (3.7%)	0	0
Total	6977 (26.2%)	5189 (9.7%)	16 (66.6%)	32 (100%)
Gusheh	26640	53280	24	32

is in charge of arbitration among these ports and also communication between accelerators and the RISCV core. The Hash and RegEx accelerators are used for our case studies and are described in detail in Sec.5.6.

To do partial reconfiguration we used MCAP_tool from Xilinx. It has a version for Linux as a C library. The API can be called with the device ID, and the desired bitstream. Note that location of module is incorporated inside the bitstream. Note that if we only need to update the program inside each core we do not need to do partial reconfiguration and we can use PCIe to update the memory contents.

5.5 Evaluation

In this section we evaluate performance of Dastgah and what needs to be considered designing a system with Dastgah. First we describe our Experiment setup. Then we perform some basic benchmarks for Dastgāh as a bump-in-the-wire NIC or middle-box: throughput and latency at each packet size, as well as throughput of the virtual Ethernet to the host. Next we profile our messaging subsystem. Broadcast messages are for control and we evaluate their latency. Inter-core packet messaging is for sending a packet from one core to another and we evaluate its throughput. It has very similar latency to the forward latency test without physical layer serialization latency, hence we do not evaluate it here.

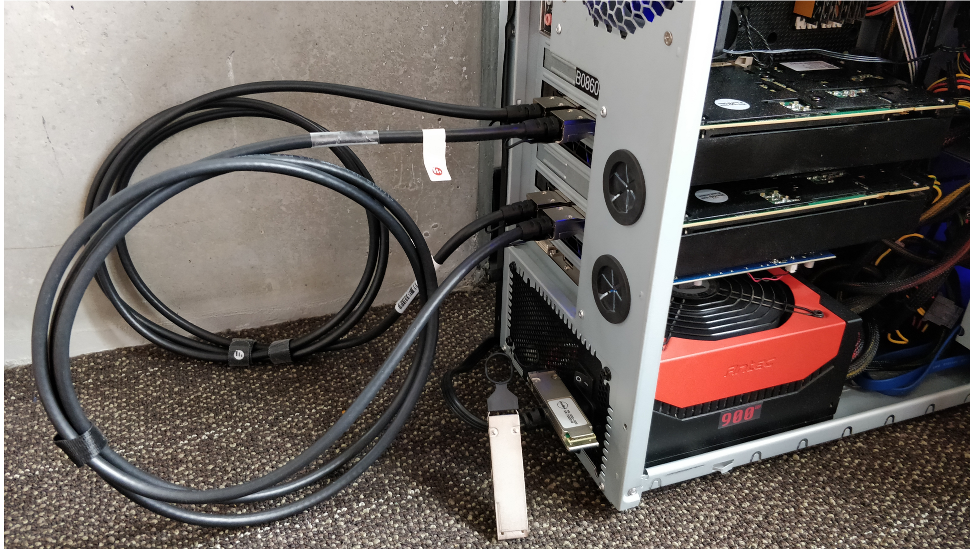


Figure 5.8: Experiment setup.

5.5.1 Experiment setup

We used a machine with Intel(R) Xeon(R) CPU E3-1230 V2 running at 3.30GHz and PCIe Gen 3 x16 as the host for our experiments. Since this machine is not capable of outputting 200 gbps we used two FPGAs, one as the tester and one to run the test design on it. We installed two Xilinx Virtex UltraScale+ FPGA VCU1525 boards on the PCIe slots of our host, whose job is to update the running code on the FPGAss and monitor performance counters. Figure 5.8 shows our setup. Note that both ports of the FPGAs are connected to each other with a 100 Gbps QSFP+ cable, so that tester FPGA can send and receive traffic at 200 Gbps.

Next we profile our tester FPGA. The first evaluation point is at what rate can we generate various packet sizes. Figure 5.9 shows the maximum bandwidth per packet size. We tested for power of 2 sizes, as well as 65 bytes which is usually the worst case, and 1500 and 9000 bytes which are typical in data-centers. The dotted lines shows the maximum achievable bandwidth at 100 Gbps and 200 Gbps. As depicted, 16 cores can fully saturate 100 Gbps for every packet size other than 65 Bytes which achieves 92% of maximum rate. They can also saturate 200 Gbps for every packet of at least size 128 bytes. 64B and 65B packets achieve lower rates of 91% and 86% of maximum rate respectively.

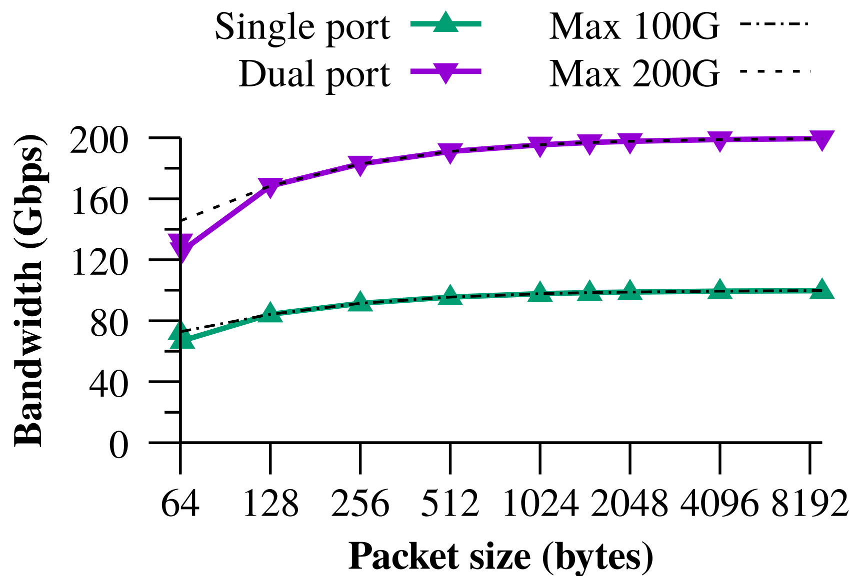


Figure 5.9: Packet generator performance on the tester FPGA.

The performance drop for 65B packets are due to the wide switches which have an interface bit-width of 512 bits or 64 bytes, causing a cycle of 1 byte after 64 bytes and almost reducing the throughput by half, from 128 Gbps to only 65 Gbps. The performance drop for 64B packets at 200 Gbps is due to the cluster switch which needs at least a cycle between start of new frames, causing similar to 65B performance drop. This bottleneck can be fixed, but there is no practical benefit in achieving 200 Gbps rate at 64B packets. The last bottleneck is when two clusters want to send to the same interface simultaneously and one of them gets stalled. This explains the slightly lower throughput for 65B packets at 200G than the 64B which the latter benefits from the gap introduced by the cluster switch. Note that for 100 Gbps the performance drop of the cluster switch is covered by all 16 cores sending to the same interface.

The second evaluation is measurements of loop-back round-trip-time (RTT) latency. This allows to separate the tester FPGA latency from the test latency running on the other FPGA. To do this we used the internal timers in each core. These timers are synced among all the cores and by putting the time-stamp inside the packet, we can measure round-trip-time latency. The measured RTT latencies are stored in local memory of the cores and portion of them are sent to

the host over virtual Ethernet interface. To measure test system RTT latency, we used two in-port loop-back modules, shown in bottom part of figure 5.9.

5.5.2 Forward throughput

To test forwarding performance of our system we send packets from the tester FPGA at maximum rate on both interfaces and observe what portion of the packets make it back. Top part of Figure 5.10 shows the full-throttle packet forwarding performance. For packet sizes below 9000B the system can fully keep up and follows the packet generator throughput in figure 5.9. For 9kB packet we have the same cluster collision as 65B packet generator, where two clusters want to send to the same destination interface and block each other. For other packet sizes the maximum rate of 128 Gbps of 512-bit wide links provides enough slack to cover this collision, but as packets get larger and larger stall times get longer and we see first performance drop at 9kB packets.

The second part of Figure 5.10 shows the number of remaining cycles at each packet size for the core. This is an example to show how we can think of time budget we have for control software in each core. For basic forwarding, it takes each core about 16 cycles to receive a descriptor and give it back to the wrapper. Hence the number of available cycles between two consecutive packets minus 16 is the available slack for each core. In practice, at 100 Gbps most packets are large to benefit from the high throughput, which translates to 1000 or more cycles budget. Moreover, if we benefit from parallel accelerators, this budget would be multiplied by the number of active accelerators, which is described in Sec.5.6.2 in more detail.

5.5.3 Forward latency

As explained in Sec. 5.5.1 we measure RTT latency, this time with the second FPGA also in loop forwarding the packets. Figure 5.11 shows the measured latencies for different packet

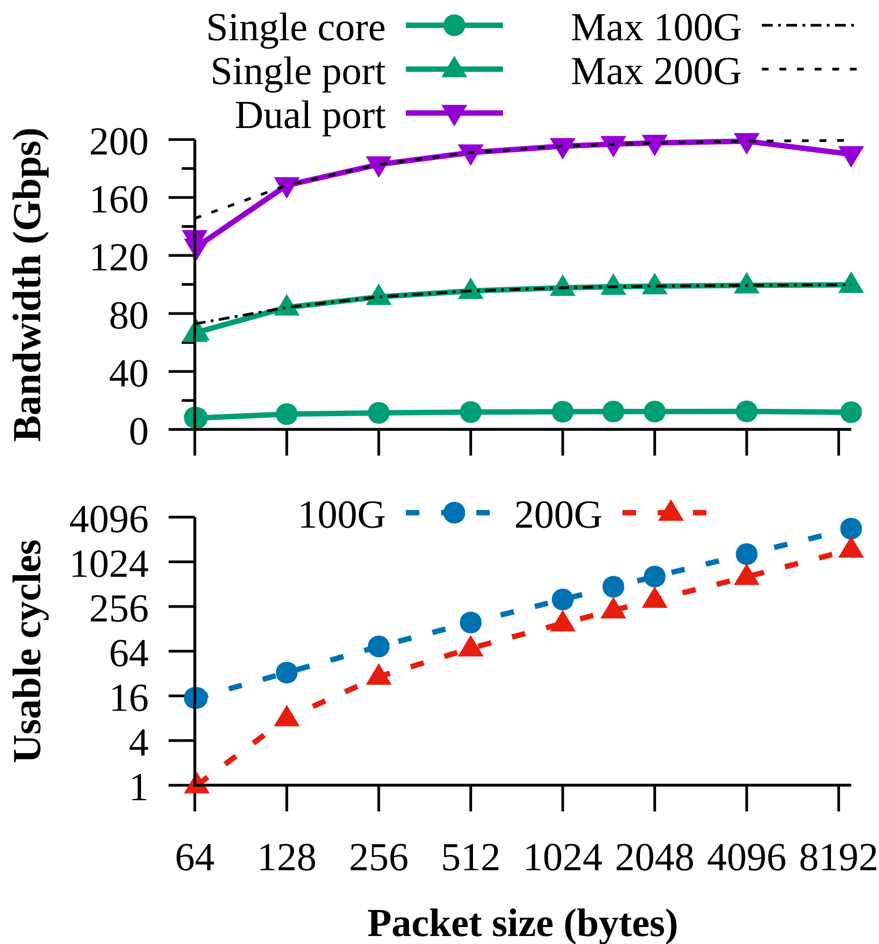


Figure 5.10: Full-throttle packet forwarding performance, and maximum spare cycles available on each Gusheh.

sizes, under both low load and maximum load scenarios. As you can see the latencies are almost linear to packet size. These latencies are introduced because of our packet FIFOs throughput our switching system, described in Sec.5.3.1, as well as serialization over physical connection between the FPGAs. Each FIFO fully receives the packet before sending it out which introduces similar serialization latency proportional to packet size. Maximum load introduces only marginal additional delay, mostly due to destination collision and resulting blockage. 64B packet size has a more significant jump, which is due to more blockage than other packet sizes. As mentioned in Sec. 5.5.1 we achieve slightly higher throughput for 64B than 65B because of the gaps inserted by the cluster switch. However, increasing the packet rate increases number of collisions and

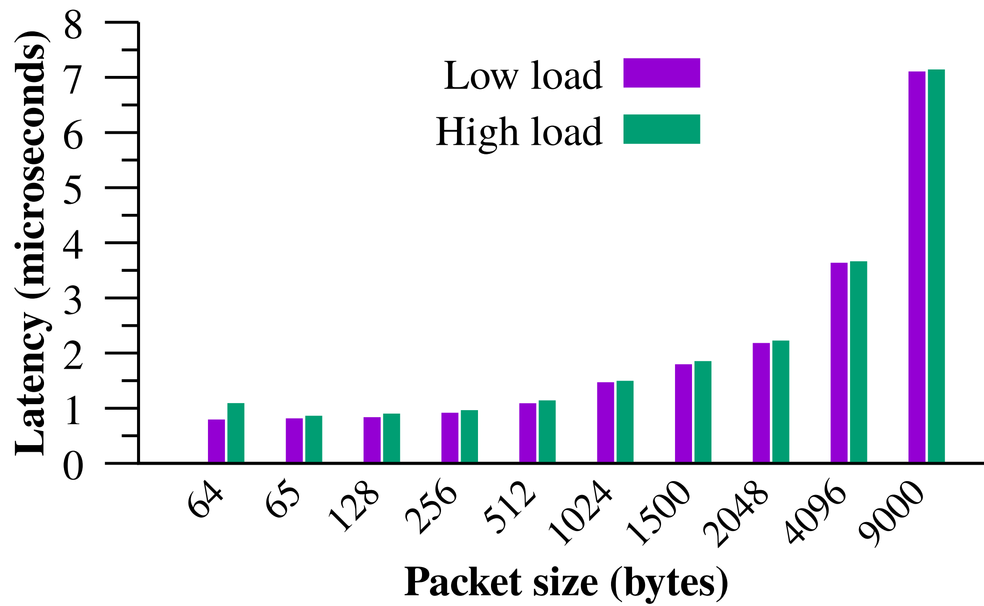


Figure 5.11: Round trip latency added by Dastgāh.

actually the single FPGA RTT latency is about double for 64B compared to 65B measurement.

5.5.4 Virtual interface to host

Next we tested the throughput of the integrated Virtual Interface to the host. First to test the transmit performance of our NIC, we simply used the same packet generator test with simple change of destination interface to the virtual interface instead of the physical one. We used 9kB packets and achieved 44 Gbps reception throughput on host, read by slurm utility. For receive side test and sending data to our FPGA NIC, we used netperf with 8 to 10 threads, 35KB UDP payload and achieved 74Gbps throughput, read by slurm utility. Doing similar test with loopback code running on FPGA resulted in 24 Gbps performance which is reasonable due to PCIe collisions when doing both directions. As a sanity check we confirmed the throughput results read by our performance counters and slurm utility match. The only difference was in case of NIC transmit test where we read 128 Gbps from performance counters but 44 Gbps from slurm, but the performance difference was justified by the drop rate reported by netstat utility.

5.5.5 Broadcast messaging latency

To measure the latency of broadcast messages, we performed two tests, one with messages distanced out and one where each core is trying to send as many messages as it can. Note that there is a FIFO inside each core for these messages, and if this FIFO gets full the write would get blocked until there is room in this FIFO. Each core only sends messages to the next core at a specific memory address, and each core looks at its assigned address for changes. We send time-stamps in this messages and compare the arrival time against transmit time. At slow rate we observed a latency between 72 to 92 *ns*. At full rate, which is not the intended use for this communication channel, we observed 1596 to 1680 *ns* of latency. This latency mostly comes from the 18 FIFO slots, 16 from FIFO and 2 from PR registers, which can be sent out every 16 cycles due to round-robin arbitration among cores, making 1152 *ns* of this latency. Rest of the latency are due to FIFOs and registers in switching subsystem and marginal added latency from the software that can have slight variation, adding up to 1680 *ns* at worse.

5.5.6 Inter-core loopback throughput

Our final benchmark is the throughput of the inter-core loopback messaging system. To do so we implemented a two step forward system: we assign half of the cores to be recipients of incoming traffic, and then each of these cores would forward the packet to a specific core in the other half, and finally that core would send it out. Since only half of cores throughput is available we used only a single physical interface for this benchmark. We achieved 60% and 61% max throughput for packet sizes 64B and 65B respectively, mainly bottlenecked by the additional computation per each core. From 128B packets forward the system could keep up with the 100 Gbps throughput.

5.6 Case studies

In this section we add hardware accelerators to Gusheh and evaluate hardware software collaboration as well as achieved performance. As our first case study we selecting packet header hashing used for load balancing and matching. We implemented Toeplitz hash [36] to hash packet header in different layers and lookup a table for the action. As second case study, we considered intrusion detection systems (IDS), explained in Sec. 5.1. One of the most challenging parts involves regular expressions rules to search for specific identifying information. We implemented a RegEx engine for detecting database insertion and deletion commands. Also we used 8 accelerators per Gusheh to demonstrate how we can benefit from parallelism inside each Gusheh. We use both of hash and RegEx accelerators to make a concrete IDS.

5.6.1 5-tuple hash accelerator

Hashing packet header fields is a very common technique in networking, enabling the use of hash tables for tasks such as matching and load balancing. A common hash function for networking applications is the Toeplitz hash. For receive-side scaling applications, the hash is computed over the IP addresses and TCP/UDP ports. Computing the hash involves concatenating these fields and then XORing segments of the hash key to produce the output hash. Computing such hashes in software is inefficient, and as such hashing benefits greatly from implementation in hardware. For example, many commercial ASIC-based NICs implement flow hashing over several fixed header fields for load-balancing flows across receive queues (e.g., Cavium LiquidIO). However, this pure hardware implementation is inflexible and cannot be modified to hash fields in arbitrary protocol headers.

Hash acceleration can be broken in to two pieces: packet parsing and hash computation. Hash computation is relatively simple to implement in hardware, but arbitrary packet header parsing and field extraction is not. For instance, Microsoft produced a hardware implementation

of packet header parsing and field extraction for their Generic Flow Table accelerator [42] that had to account for 310 possible header combinations.

```
// check eth type for IPv4
if (*((unsigned short*)(header+12))==0x0008){
 // Add IPv4 addresses to the hash
 ACC_HASH_DWORD = *((unsigned int*)(header+26));
 ACC_HASH_DWORD = *((unsigned int*)(header+30));

 // check IHL and protocol
 if((header[14]==0x45)&&(header[23]==0x06)){//TCP
   // Add ports to the hash
   ACC_HASH_DWORD=*((unsigned int*)(header+34));
   // read hash
   hash = ACC_HASH_READ;
   // Lookup action from the corresponding table
   act = *(TCP_table+(hash&0x0003ffff));
  } else ...
 }
if (act==0) desc.len = 0; else desc.port = 2;
```

Listing 5.1: Using hash accelerator on RISCV cores.

Using software to implement packet parsing and hardware to implement hashing offers a good balance of flexibility and efficiency. In order to be effective, careful consideration must be given to the design of the hardware-software interface of the hash accelerator. For example, it is more efficient to pass the packet fields to the hash accelerator instead of sending length and packet offset of each field. This is because fields are placed not-consecutively in the header and they have short length generally, and computing length and offsets are additional compute for the controller RISCV. Also the hash accelerator does not require a dedicated memory interface

anymore.

Listing 5.1 shows the simple software code that parses the packet headers, decides which fields to hash, reads those fields, and passes that data to the hash accelerator. In addition to the shown code snippet, we load the packet header pointer based on packet slot, and reset the hash accelerator per packet. Our complete code checks for TCP, UDP and also has a table for IP packets which are not TCP or UDP. Then it masks the first 18bits of hash and look-ups the corresponding table in scratch memory. In this case-study we have 3 tables of size 256 KB. Based on table value it drops the packet, or forwards it to host (port 2). Note that we can use broadcast messaging to update tables of other Gushehs, for example after processing the very first packet of a flow. The communication between core and accelerator is simply through registers: 1 for control, 1 for reading results back, and 3 for loading data of size byte, word and dword. Moreover, RISCV core only accesses low latency DRAM and reads copy of packet header, described in Sec.5.2.4.

The hash accelerator implementation with a fixed 40-byte Toeplitz hash key uses 1029 LUTs and 296 registers and can compute the hash of a block of data of up to 36 bytes, detailed in Table 5.2. Based on simulation, which is confirmed by our performance results, 17 cycles are used for initialization of each packet, 27 is used for parsing and hashing the packet header, 7 cycles for looking up the table from high latency scratch memory, and finally 14 cycles to perform the desired action. We can keep up with full 200 Gbps rate from packet size of 512B and above. Furthermore we did a test with mixed packet sizes of (64, 1024, 128, 1500, 256, 1024, 512, 1500, 1024, 1500, 65, 2048, 512, 4096, 1500, 9000) bytes which resulted in only 5 drops per seconds at rate of 197 Gbps. This shows our system can keep up at data centers with high speeds.

Note that it took us about 2 days to implement this accelerator and about 5 days to test and debug it on FPGA. Having the ability to set debug registers in software and read them back from host was very helpful in finding and fixing bugs.

5.6.2 Regular Expression accelerator

```
// Packet parsing ...
if (act!=0){
  // start regex parsing, skip Ethernet header
  ctx->regex_accel->start = ctx->desc.data+14;
  ctx->regex_accel->len = ctx->desc.len-14;
  ctx->regex_accel->ctrl = 1;
}

// RegEx match/done handler
void regex_done(struct slot_context*ctx){
  // check for match
  if (ctx->regex_accel->ctrl & 0x0200)
    ctx->desc.port=2; //send to host
  else
    ctx->desc.len=0; //drop packet
  pkt_send(&ctx->desc);
}
```

Listing 5.2: Using RegEx accelerator on RISCV cores.

Deep packet inspection (DPI) and intrusion detection systems (IDS) rely on a checking packets against a large set of rules. We used hashing and table lookup from the first case study to parse the header and detect if we need regular expression matching. Regular expression matching benefits greatly from hardware implementation. If header demands RegEx check we pass the range of bytes that needs to be checked to the RegEx accelerator and wait for a match or end signal. Listing 5.2 demonstrates how we start a RegEx engine and how we implement the action based on the results.

The rule-sets are periodically updated, requiring some form of reconfigurability. Using

software to implement the packet header parsing and rule selection improves flexibility. Additionally we can use partial reconfiguration to update the RegEx accelerators on the fly. This removes the need for a general purpose regular expression accelerator, such as reconfigurable deterministic finite automaton (DFA) which requires a significant amount of logic resources with poor timing performance, resulting in poor matching throughput in an FPGA. Instead in our implementation offline software converts the regular expression rules to Verilog, which is then placed and routed to produce a new Gusheh. The resulting accelerators are compact and have good timing performance.

We used Regular expressions to SystemVerilog compiler from [47] for our implementation. Note that other hardware implementations of RegEx engines can also be used and are complimentary to our work. Table 5.2 shows break down for each of the generated accelerators which only use 57 LUTs and 86 registers which is tiny. These accelerators can process one byte at a time and our two 100 Gbps physical interfaces can source up to 128 bytes per cycle at 250 MHz. This means we need 16×8 accelerators to run in parallel to keep up. So we used 8 accelerators per Gusheh and control the data flow among them in software.

The memory architecture of Dastgāh provides a dedicated memory port for each packet storage block to the accelerators. In our implementation we have 8 memory blocks and 8 accelerators, and thus we used a one to one matching. Each of these ports are also banked which means 16 accelerators can run in parallel without blocking each other. This allows the software to orchestrate the accelerators to work on different packets at the same time, or make a chained pipeline system among them.

To evaluate throughput of our IDS we first profiled the performance of RegEx accelerators on all incoming traffic without any header parsing. At 100 Gbps rate we were able to keep up from packets of size 512B and above, and for 200 Gbps we achieved 194 Gbps and 192 Gbps throughput at 1KB and 9kB packets respectively, and full throughput for the packet sizes in between. Next we added parsing which reduces the demand from RegEx engines but increases

max latency per packet. For mixed packet sizes (128, 1024, 128, 1500, 256, 1024, 512, 1500, 1024, 1500, 256, 2048, 512, 4096, 1500, 9000) we encountered only 28 drops per second at 197 Gbps.

Note that we could easily further improve performance of our system. Currently a bulk of cycles are spent checking status of each accelerator. Using a count lead zero instruction can make this much faster. Also in the used Vex-RISCV each branch or unconditional jump takes 3 cycles which can be tweaked. However, we wanted to simulate a more realistic application which such cycle requirements.

5.7 Discussion

This work was the initial effort in showing the additional functionality offered by small micro-controllers spread in an FPGA. There are several potential optimizations that can improve performance of this system. For some applications that require off-chip memory we can easily add well-known caching methods both outside and inside Gushehs. Another potential for optimization is higher throughput for our NIC interface to host, and adding features such as SRIoV. Also, if there is enough demand for Dastgāh architecture, FPGA manufacturing companies can provide distributed embedded processors running at higher clock rates to minimize performance overhead of Dastgāh and make FPGA development less costly and strenuous.

This work can also be used for testing new scheduler systems. The scheduler can have a very high impact in total system performance, not only by proper load balancing, but also by having early drop capabilities or using a small memory to handle head-of-line blockings. Another opportunity is to address distribution challenge for SoC-based mutlicore SmartNICs. If they do round robin for destination core the amount of shared state is very large and cannot be handled through caches, and if they do hash-based destination core selection single core's performance would bottleneck elephant flow throughput. However, our scheduler can be customized to detect

and distinguish elephant flows from other low throughput flows. Scheduler can mark packets from the elephant flows and distribute them among few cores, in contrast to lower throughput flows that go to a single core. This would reduce the required state sharing among cores to only elephant flows. Specially packets from elephant flows are large which makes state sharing very sparse in time and easy to handle with out broadcast messaging system.

Another potential target for Dastgāh is FPGA-based F1 instances from Amazon AWS. We found out they use the same FPGA chip as the one used in this chapter, and the PCIe units in their shell has a very similar interface to ours and we can potentially port our design in a short period of time. Using our system can lower the development entry cost for their customers. Also currently F1 instances are not shared and are more expensive, but if AWS adds support for Partial Reconfiguration we can implement FPGA sharing through isolation and virtualization of Gushehs.

In the future we believe that this framework can be implemented as an SoC architecture in future FPGAs. This will result in significantly faster packet processing performance that we can achieve instantiating RISC-V cores inside of an FPGA. The main challenge for using current ARM processors integrated in FPGAs is the memory bottleneck. For 40 Gbps SoC-based smartNICs such as Broadcomm's Stingray uses 16 ARM cores running at 3 GHz clock speed, but still they have memory contention problems. Moreover, as network speeds increase, designing accelerators that can keep up with the rate becomes more challenging. They would need wider input buses and more parallel logic. This is similar to going from a 32-bit processors to a 64-bit counterpart, which typically requires more than twice resources. Also, some of the accelerators cannot be fully pipelined and can become the system bottleneck. The main solution to this challenge is to distribute the load among more than one accelerator. Increasing the number of accelerators per core makes their management and control flow much harder and restraining on one or very few available ARM processors. In contrast, in out design there are more processors, and hence less number of accelerators per soft core, and even we can add more soft cores to keep

the ratio low. Moreover, since the memory is only shared between a core and it's local accelerators, there is no more memory contention problem. That being said, if FPGA chip manufacturing companies provide more high speed general purpose cores per FPGA, our design can clearly benefit from them.

Finally, note that some networked applications, particularly the congestion control algorithms, may require specialized schedulers or significantly lower latency than we can provide in Dastgāh. For instance, recently, a programmable transport protocol has been proposed for FPGA hardware [8]. This is an example of a system that may not be feasible to implement in Dastgāh.

5.8 Summary

We presented Dastgāh, a development framework that can achieve 200 Gbps while splitting hardware and software development. We reduced FPGA development cost by adding software like modularity, flexibility and debugging for networked applications, with marginal TCO overhead. To do so, we demonstrated that with a simple RISC-V core augmented with hardware accelerators it is possible to keep up with increasing line rates of networked applications, with only marginal effect on system latency. We believe many networking applications and research projects can benefit from this framework.

5.9 Acknowledgments

Chapter 5 contains reprints of M. Khazraee, A. Forencich, G. Papen, A. Snoeren, A. Schulman, "Dastgah: Software-defined FPGA SmartNICs", currently being prepared for submission for publication. The dissertation author is the primary author of this paper.

Chapter 6

Efficient Backhaul And Compute For Wireless Base Stations

Software-defined radios (SDRs) significantly reduce hardware development cost by decoupling of the radio frontend—including analog RF components and digitizers—from the signal processing backend. Now the backend portion can be deployed on general-purpose computing devices like CPUs [137], GPUs [77], or re-programmable DSPs [14]. When equipped with wideband analog-to-digital converters (ADCs), SDRs can capture a broad range of frequencies, and support for new protocols can be added by implementing the requisite signal processing in software [145, 99]. In principle, they can capture, decode, and analyze a variety of signals across arbitrary frequency ranges.

However, the universality of the SDR architecture comes at a price: raw samples need to be backhauled between the frontend and the signal-processing backend. Most existing SDR implementations are inefficient in that they require backhaul capacity and processing performance in proportion to the SDR's sampling frequency, irrespective of the (typically far more limited) bandwidth of the signal being captured. As described in the Introduction chapter, this high backhaul requirement results in limitation to use SDRs for base stations.

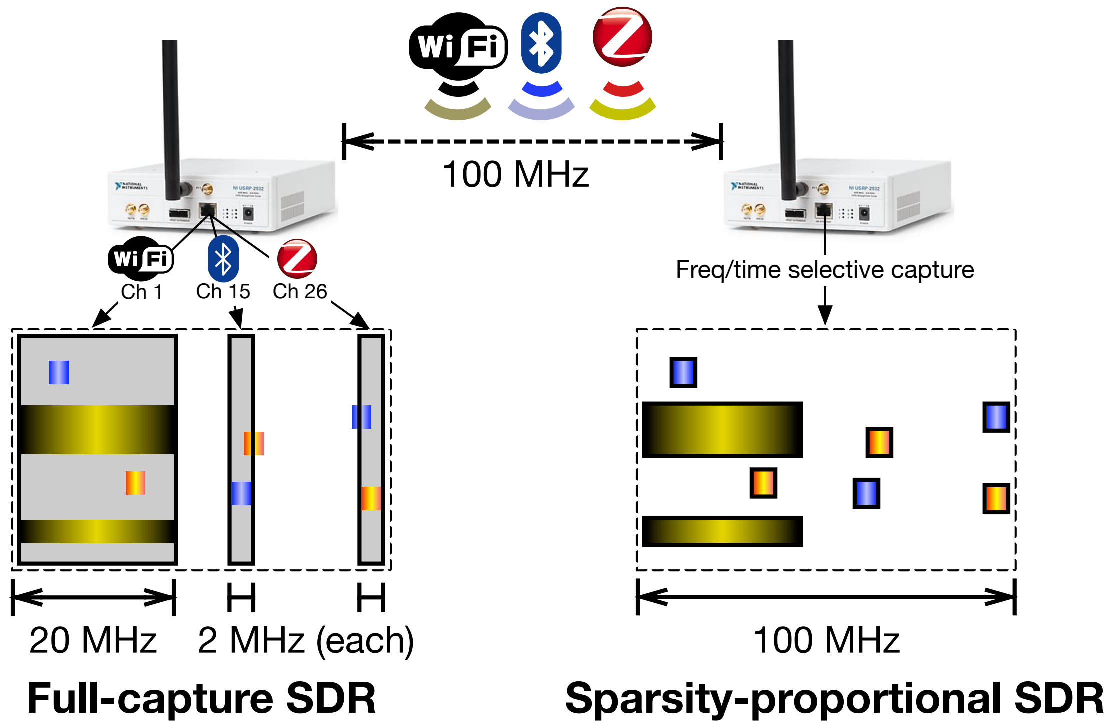


Figure 6.1: SparSDR's goal is to make SDRs capture primary transmissions rather than entire channels.

As a result, wideband (e.g., 100-MHz) SDR frontends often "throw away" information via digital downsampling in order to reduce the datarate of the backhaul stream. For example, the popular USRP N210 frontend captures 14-bit I/Q samples at 100 Msps, but must perform $4\times$ digital downsampling to 25 Msps in order to operate within the capacity of its 1-Gbps backhaul link. Even at this reduced bandwidth, signal processing requires desktop-class compute performance, foreclosing the possibility of using embedded processors (e.g., the CPU on a Raspberry Pi). Unfortunately, this limitation renders the N210 impractical for infrastructure deployments and base stations. For instance, a general-purpose IoT gateway needs to support frequency-hopping protocols such as Bluetooth and ZigBee that span 80 MHz of bandwidth (Fig. 6.1 left). Such an application requires 2.5 Gbps of backhaul capacity and server-class processing capability. (Although a single transmitter could be aliased into a narrower bandwidth, the full 80-MHz bandwidth is required to support simultaneous transmissions).

However, it is well known that the RF spectrum is sparsely occupied across both time and

frequency [96, 67, 112]. We present SparSDR, a sparsity-proportional architecture for wideband SDRs that takes advantage of this fact. In SparSDR, we propose a signal-processing extension for existing SDRs whose backhaul and compute requirements scale in proportion to the bandwidth of the captured *signal*, not the RF spectrum sampled (Fig. 6.1 right). We demonstrate that for many commonly used bands, SparSDR significantly reduces backhaul and compute requirements. Because SparSDR preserves the raw samples of the captured signal, it does not sacrifice the flexibility or protocol independence of SDR.

The added processing in SparSDR must satisfy the following requirements: (1) it must not significantly reduce overall signal quality, (2) it must allow for signals that have low SNR, such as below-the-noise transmissions from IoT protocols, and (3) it must fit in the unused processing resources of existing SDR compute platforms. We satisfy all three of these constraints by repurposing a classic signal analysis technique, short-time Fourier transform (STFT) [152, 62]. STFT is traditionally used for power-spectrum analysis: observing the primary frequency components that exist in a signal, and how they change over time. The key property of STFT that makes it possible to achieve sparsity proportionality is that only the primary frequency components need to be backhauled to reverse the transform and recover the raw samples.

Although STFT is reversible, existing algorithms generally recreate the entire raw sample stream—hence, while the backhaul requirements may be reduced, the signal processing task remains proportional to the original sample rate, not the bandwidth of primary signals in the capture. In SparSDR, we introduce a sparsity-proportional reconstruction algorithm that recovers only a portion of the captured signal. Specifically, it reverses only part of the STFT, effectively using STFT itself to identify and isolate the primary signals. The algorithm also corrects for the phase and frequency offsets introduced by STFT-based downsampling.

In this chapter, we evaluate the tradeoffs in the parametrization of STFT-based downsampling; the size of the FFT and the type of window used can both significantly impact the downsampling size, signal quality, and latency. We find that by applying a simple threshold to

detect primary signals, it is possible to detect signals below the noise floor. We also demonstrate that STFT fits in the unused processing resources of current SDR frontends. Therefore, by replacing the downsampling found in existing SDRs with STFT, we can backhaul and process signals with bandwidth and performance requirements inversely proportional to the sparsity of the signals.

Using SparSDR, we demonstrate that a USRP N210 can can continuously capture 100 MHz while sending 14-bit I/Q samples—4× its backhaul capacity. Specifically, we show that it is possible to simultaneously receive 450 Bluetooth Low Energy (BLE) transmissions per second across the full 2.4-GHz band using a Raspberry Pi 3+ connected to a SparSDR-enabled USRP N210 operating at its full 100-MHz bandwidth. We also show that SparSDR makes it possible to build a Cloud SDR platform out of low-end wideband SDRs (e.g., an Analog Devices Pluto), residential-class backhaul, and inexpensive computing platforms (i.e., the Raspberry Pi 3+). The limited backhaul bandwidth in residential networks has traditionally forced crowd-sourced SDRs to operate with only narrowband (3.2-MHz) USB SDR dongles [120, 102]. Finally, we demonstrate that battery-powered wideband SDRs have sufficient processing resources to incorporate SparSDR's additional stages, including the USRP E310 and USB-powered SDRs such as the AD Pluto and USRP B210.

In summary, our contributions in this chapter are as follows:

1. We describe how the STFT can be employed as a lightweight frequency/time-selective downsampling algorithm that produces protocol-independent raw samples that requires backhaul proportional to the bandwidth of the captured signal. We also present a lightweight time-domain reconstruction algorithm which uses partial STFTs to downsample to the primary signals in a capture (Section 6.2).

2. We evaluate the tradeoffs associated with STFT-based downsampling and demonstrate that the resource-constrained USRP N210—and even the ∼$100 AD Pluto—have sufficient unused resources to support SparSDR (Section 6.3).

3. We evaluate SparSDR in two case studies: an IoT gateway built using a USRP N210 connected to a Raspberry Pi, and a wideband Cloud SDR that operates over residential Internet-class backhaul links (Section 6.5).

SparSDR is open source. We have developed a SparSDR module for GNU Radio, making it easy to drop into existing projects. The hardware implementation is parametrized to enable porting to new SDR platforms (Section 6.4); we provide example hardware implementations for the USRP N210 and AD Pluto. The hardware and software source can be found at:

https://github.com/ucsdsysnet/sparsdr

6.1 Motivation

In this section, we motivate the need for frequency and time *sparsity-proportional* SDRs. First, we provide an example of how wideband SDR captures are sparse in frequency and time. Then, we describe how simple downsampling in today's SDRs selects only one contiguous band at a time—indiscriminately throwing away signals—and leaves many spare processing resources.

6.1.1 Popular bands are sparsely occupied

The motivation for SparSDR comes from the observation that there is a mismatch between the sparsity of spectrum usage and the full-capture design of SDRs. Commodity SDR RF frontends are often built with wide-band ADCs that can capture more than 50 MHz, making it possible for them to capture many channels and protocols simultaneously. In the following experiment, we demonstrate that the popular 100-MHz wide 2.4-GHz ISM band is sparsely occupied—even though it is shared by a variety of protocols such as WiFi (three orthogonal 20-MHz channels), Bluetooth (79 1-MHz channels), and ZigBee (sixteen 2-MHz channels).

We use the OneRadio[1] wideband SDR to capture a 45-second 125-MHz bandwidth

[1] http://www.oneradiocorp.com/oneradio-receiver/technology

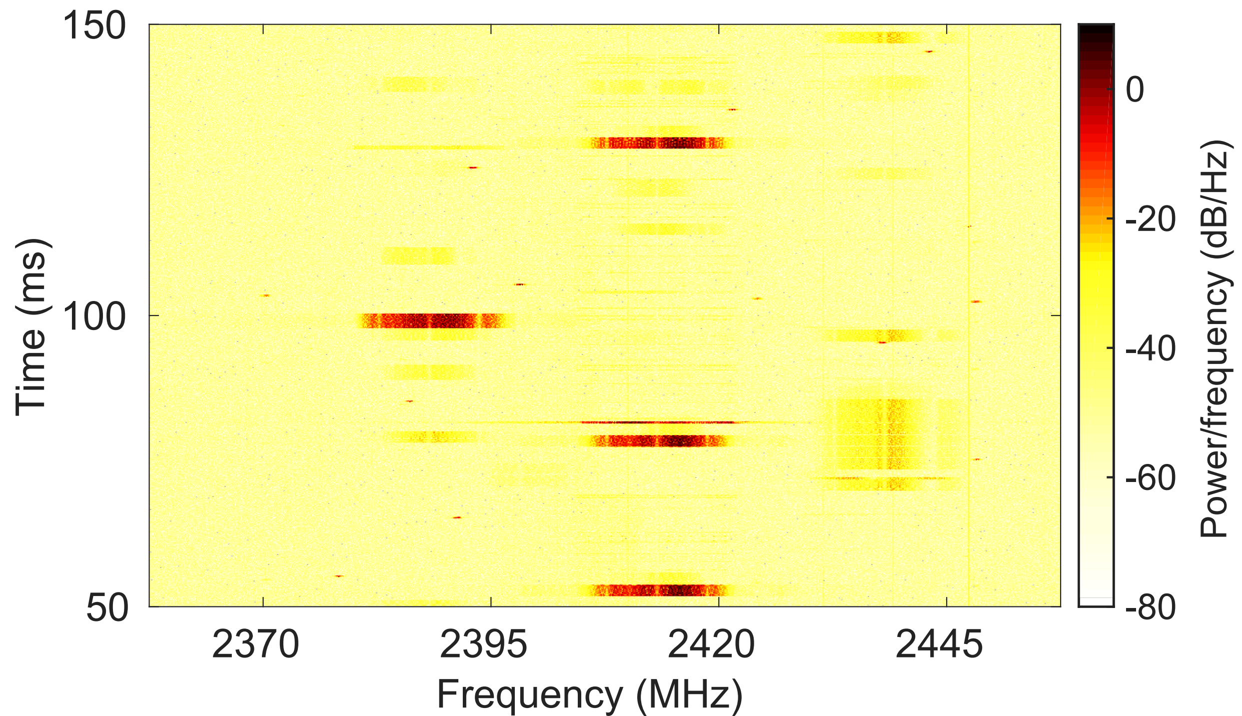


Figure 6.2: A snapshot of the 2.4-GHz band collected by a OneRadio wideband (125-MHz) SDR shows that even popular unlicensed bands are sparsely occupied.

snapshot of the entire 2.4-GHz band in an office building during business hours. Fig. 6.2 shows a 150-millisecond portion of the capture. Even in this short period, it is evident that although the 2.4-GHz band is active (there are WiFi packets being transmitted), its usage is sparse in frequency and time. Fig. 6.3 shows the distribution of instantaneous bandwidth occupancy every 10 microseconds across the entire 45-second capture. This figure shows that the wide-band (125-MHz) capture is sparse: the band is unoccupied for almost 40% of the time and the occupancy is less than 18 MHz (approximately one WiFi channel) 85% of the time.

6.1.2 SDRs need smarter downsampling

SDR frontends often downsample to reduce the capture bandwidth to allow for the use of inexpensive and widely available backhaul links. These inexpensive links limit the capture bandwidth: the USRP N210 can only capture 25 Msps on its gigabit-Ethernet backhaul, and the AD Pluto can only achieve $\sim$5 Msps over its USB 2.0 backhaul (with four bytes per I/Q sample).

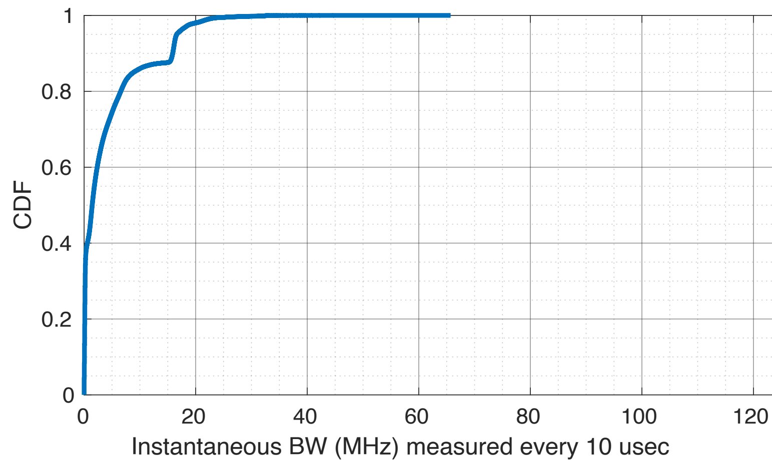


Figure 6.3: Distribution of occupied bandwidth over 10 microsecond intervals at 2.4 GHz. We use a relative power threshold of 5 dB above the maximum observed noise floor to determine if a frequency is occupied.

Downsampling is also necessary to operate within the compute resources available. For instance, processing a 100-Msps capture on a desktop-class 3-GHz CPU only allows for 30 cycles per sample and is simply infeasible on an inexpensive embedded processor like that found on the Raspberry Pi.

This is an unfortunate state of affairs, because many signals are indiscriminately thrown away during downsampling. However, the fact that the downsampling logic is often implemented on an FPGA at the SDR frontend presents an opportunity to modify the frontend to downsample more intelligently. Many SDR frontends have additional processing resources intended to support smarter downsampling, the implementation must be compact in order to fit into the FPGAs of existing SDRs.

6.2 Design

SparSDR is an add-on for existing software-defined radios, shown in Fig. 6.4, that makes them *sparsity-proportional*: they only require backhaul capacity and compute resources in inverse proportion to the sparsity of the signals they receive. The primary component of SparSDR is a frequency-and-time selective downsampling step that replaces the basic downsampling logic

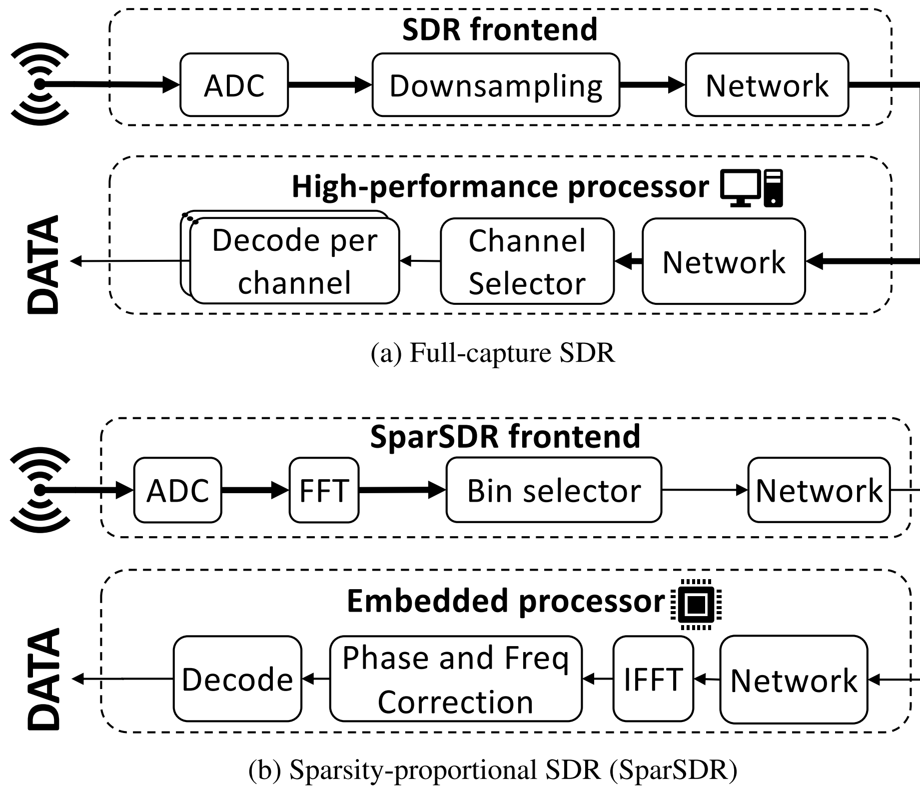
in the SDR frontend. With this modification, the SDR frontend only backhauls samples at frequencies that have active signals. To support frequency-and-time selective downsampling, SparSDR also inserts an extra processing step to reconstruct the raw time-domain samples at the backend compute platform (e.g., CPU, GPU, or DSP). We introduce an efficient reconstruction algorithm that recreates the time-domain signals while maintaining the same level of sparsity. Unlike compressed sensing, SparSDR does not require new frontend hardware (i.e., random sampling ADCs), nor any assumptions regarding the sparsity of the captured signal, and the reconstruction algorithm can operate within the constrained performance of embedded platforms (e.g., Raspberry Pi).

Here we focus exclusively on making the receiver side of SDRs sparsity-proportional. SDR receivers are are more computationally intensive than transmitters: receivers produce a constant stream of samples that must be processed in real time. We note, however, that SparSDR can be modified to operate on the transmit side of SDRs by reversing its steps: The frequency-and-time selective downsampling algorithm can be reversed to construct a signal that is sparse in the frequency domain, and upsample it to the full bandwidth of the transmit front-end (e.g., DACs). A potential application could be to efficiently transmit multiple, concurrent BLE packets (2 Msps) at varying frequencies in the 80-MHz ISM band.

6.2.1 Frequency-and-time downsampling

We begin by describing SparSDR's downsampling algorithm that generates a sample stream whose bandwidth is inversely proportional to the signal sparsity in the captured spectrum. To retain the flexibility and protocol independence of SDR, SparSDR's downsampling must be general: it cannot be specific to a particular communication protocol.

We draw inspiration from a popular spectrum analysis measurement: power spectral density. Typically observed with a spectrum analyzer, the power spectrum reveals the frequency and magnitude of received signals. A common tool for performing power spectral analysis is the



(a) Full-capture SDR

(b) Sparsity-proportional SDR (SparSDR)

Figure 6.4: (a) Full-capture SDRs require backhaul and compute resources that are equal to the bandwidth of the downsampled ADC capture. (b) SparSDR introduces new blocks that make resource usage proportional to the sparsity of the captured signals.

Short-Time Fourier Transform (STFT) [152]. STFT divides the sample stream into overlapping shorter windows of equal length and then computes the Fourier transform on each shorter window. This process reveals the frequencies of the signals that are active in that short window of time. For instance, we use the STFT to generate Fig. 6.2, showing sparse usage of the 2.4-GHz spectrum.

Our key insight is that the STFT algorithm can be repurposed to downsample an SDR frontend's ADC capture in frequency and time. We can then detect the frequency bins which are active and backhaul only those bins. The resulting downsampling is proportional to the activity in spectrum. For example, for the capture shown in Fig. 6.2 we would backhaul just the few active frequency and time components during the transmissions. Importantly, STFT is invertible and protocol independent. Even when backhauling only the active frequency and time components, we can reconstruct the active signals accurately.

A natural question is then: Can we compute STFT-based downsampling in real time on SDR frontends? The STFT transform is computed by repeatedly performing an FFT operation on short windows of a capture. The use of FFTs means the STFT computation can be hardware-accelerated to operate in real time. Specifically, we rely on pipelined streaming FFT hardware implementations that take in one time-domain sample and output one FFT frequency bin per clock cycle. Efficiency improves with increasing FFT (capture-window) length, but it also increases decoding latency (detailed in Section 6.4.1). Additional latency may be problematic if a transmitter needs to respond quickly after receiving a packet (e.g., by sending an acknowledgment).

An effective STFT-based downsampling approach must address several complexities, however, that we discuss below in turn: (1) the FFT must be parameterized to produce a sparse frequency-domain representation of each signal contained in the capture; (2) a detection algorithm is needed to select the active frequency bins to be backhauled; and (3) the frequency bins must be efficiently reconstructed back into raw samples. We discuss each of these issues in turn in the remainder of this section.

6.2.2 Sparse representation of signals with STFT

We first describe how we parameterize the STFT in SparSDR to produce a sparse frequency-domain representation of a capture without harming signal quality. In particular, we select an STFT windowing function and determine the overlap between windows.

A naive approach, commonly called rectangular windowing, would be to use no windowing function and divide the capture into non-overlapping short windows of samples. However, rectangular windowing does not produce a sparse representation in the frequency domain. Fig. 6.5 shows the frequency response curves of the rectangular window compared to other common STFT window functions. The rectangular window leaks energy into adjacent bins with its main lobe at -13db. Due to the sharp transitions at the edges of the rectangular window, the leakage does not degrade rapidly, even at bins that are far from the center. This leakage negates the benefits

of STFT-based downsampling because the signal's energy is spread over many frequency bins, each of which would need to be backhauled in SparSDR. Moreover, leakage poses a problem for signals that are close in frequency which would therefore be mapped to the same STFT bins.

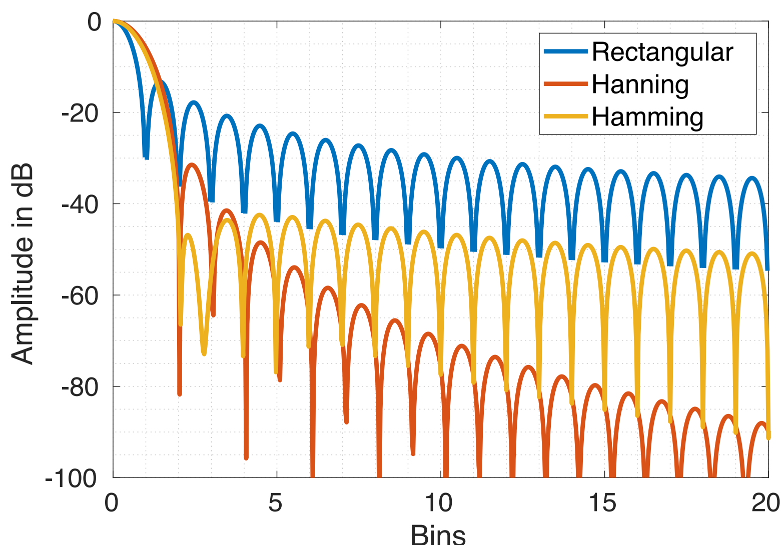


Figure 6.5: Frequency response of common window functions

In contrast to the rectangular window, generalized cosine windows attenuate the amplitude of the samples at the edges of the STFT. Fig. 6.5 plots the frequency response of two common cosine windows, Hanning and Hamming. The figure shows that cosine windows can provide sparse representation of signals in the frequency domain by reducing energy leakage into adjacent frequency bins. These window functions are preferable for SparSDR because they reduce the number of bins that must be backhauled to reconstruct the signal [107, Ch. 7, p. 468]. For example, Fig. 6.5 suggests that SparSDR need only backhaul a few bins adjacent to the signal if Hamming or Hanning windowing functions are employed.

Unfortunately, there is a complication with the direct application of cosine windows: they significantly attenuate the time-domain samples at the edges of each window. Simply inverting the window only amplifies the noise (quantization) at the edges of each window. We observe, however, that samples at the edges of a cosine window can be perfectly recovered by overlapping sequential windows by half of the window length (the overlapped region of the sinusoids adds up

to one). While overlapping windows by 50% introduces a bandwidth and computation overhead of 2× compared to a non-overlapping rectangular window, we demonstrate in Section 6.3.1 the tradeoff is worthwhile for SparSDR: overlapping cosine windows provide more than a 2× reduction in backhaul bandwidth.

Compared to the gradual decline in frequency response of the Hamming window, we find the Hanning window's steep decline is better-suited for SparSDR. Although the Hamming window has a lower first sidelobe, Hanning requires backhauling fewer bins because the energy is concentrated in the closest few bins. Therefore, we select the Hanning window as the default window function for SparSDR. If custom windowing is desired, SparSDR's software can load new windowing coefficients into the SparSDR frontend FPGA.

6.2.3 Universal signal detection for STFT

Next we describe how SparSDR selects the frequency bins in an STFT that contain signals of interest. A typical approach to detect a signal is to use time-domain correlation, namely searching for a protocol-specific synchronization sequence (e.g., a preamble) [102]. We eschew time-domain correlation in SparSDR precisely because it requires protocol-specific processing, which would undermine the flexibility of SDR frontends. Time-domain correlations are not universal: they requires the knowledge of the correlation sequence for each protocol. Said differently, it can only detect signals whose correlation sequences are known *a priori*. Furthermore, time-domain correlation requires per-protocol processing, making real-time multi-protocol frontends hard to implement.

Instead, SparSDR uses a threshold-based energy detector, an efficient and universal method of detecting signals. It is universal because it does not depend on any signal-specific patterns, and efficient because it does not require per-protocol computation. SparSDR compares the magnitude of each FFT bin with a threshold. If the magnitude is above the threshold, the bin is backhauled, otherwise the bin is dropped.

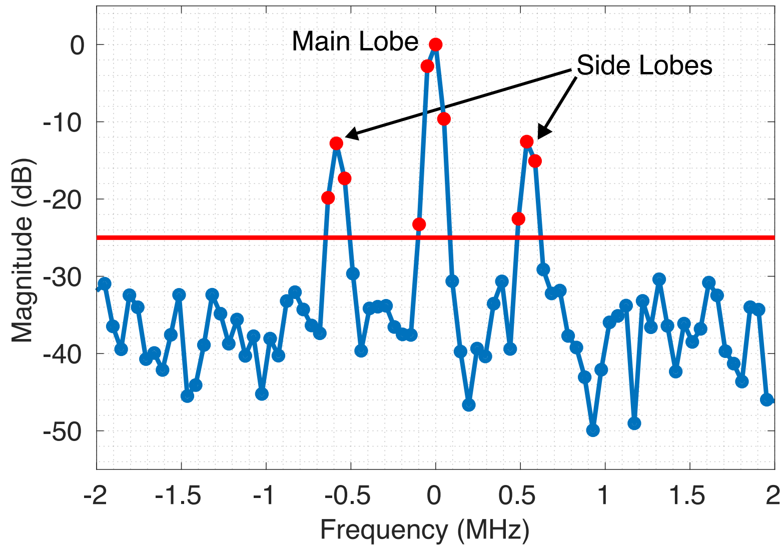


Figure 6.6: An example of SparSDR's threshold-based energy detector applied to an STFT of a Bluetooth signal. Thresholding a Bluetooth signal results in backhauling at most 10 bins (0.5%) of a 2048-bin STFT for a 100 MHz capture.

Fig. 6.6 shows an example of what SparSDR's energy-based thresholding looks like for a 1-MHz Bluetooth signal. In this example, we simulated capturing the Bluetooth with 100 MHz of bandwidth and a 2048-bin STFT. Notice that SparSDR does not need to backhaul the entire 1 MHz Bluetooth signal ($\sim$20 bins). The reason is, Bluetooth is frequency modulated, so each STFT only contains a few symbols. Therefore, SparSDR only needs to backhaul the main lobe and the side lobes for those symbols, which together only occupy only half of the Bluetooth bandwidth (10 bins).

While energy-based signal detection can miss decodable signals that are below the noise—such as distant ZigBee transmissions—we observe that this is mitigated by the natural oversampling SparSDR provides by performing the STFT on the full capture bandwidth. STFT bins are averaged over many time-domain samples (e.g., 2048) captured at a high sample rate (e.g., 100 Msps) that is several times faster than required for ZigBee. We evaluate the benefits of oversampling for energy-based detection in Section 6.3.3.

The thresholds for energy-based signal detection must be set carefully to avoid detecting spurious signals. A receiver may have non-uniform noise in the frequency domain, and harmonics

from out-of-band transmitters may alias into the capture bandwidth. To address this issue, SparSDR provides a configurable threshold for each FFT bin that can be updated constantly as conditions change.

To help select bin thresholds, SparSDR monitors the average energy of each bin. The monitoring is implemented on the SDR frontend's FPGA as an exponentially weighted moving average (EWMA). The benefit of using a weighted average is that it has a streaming implementation that requires only limited FPGA resources. When a new FFT value is available for bin n, the FPGA updates the bin average using the following formula:

$$Avg(n)_{new} = \alpha \cdot Avg(n)_{old} + (1-\alpha) \cdot Value(n).$$

The length of the averaging window (α) is configurable at the SDR backend. A short window is useful for monitoring the noise floor of a bin that has dynamic transmissions. Such monitoring is useful in ISM bands, where the noise floor must be distinguished from signals originating from many transmitters that are received at different energy levels. A longer window is effective for determining if a bin is occupied by a constant transmitter (e.g., TV and radio broadcasts). Such constant transmissions consume a large amount of backhaul resources and might not be of interest for a particular application of SparSDR. To make it possible to ignore these signals, we support a bin-masking feature: masking acts like a notch filter to avoid backhauling specific signals that are not of interest.

The final component of SparSDR is the efficient reconstruction of time-domain signals from the frequency-domain STFT bins that are backhauled to the processing backend. These time-domain signals need to be recovered at their original sample rate (e.g., 2 Msps for BLE) in order to prepare them for decoding or other application-specific signal processing. To achieve our vision of sparsity-proportional compute, this reconstruction step must be efficient: the compute resources required should be proportional to the data rate of the partial STFTs that are being

backhauled.

6.2.4 Efficient reconstruction of partial STFTs

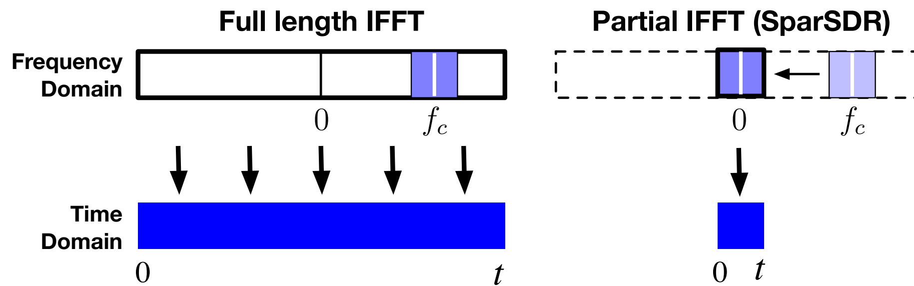


Figure 6.7: Compute-efficient reconstruction with partial IFFT

A straightforward approach to reconstruction would be to perform an IFFT that is the same length as the FFT executed on the SDR frontend. This would require performing a full-length IFFT on the bins backhauled from the SDR frontend, with other bins set to zero (Fig. 6.7 left). The time-domain samples output by this full-length IFFT must be downconverted to baseband by multiplying by a sinusoid, and then downsampled from the full capture rate to the signal's sample rate. Clearly, the amount of processing is independent of the number of bins backhauled.

A more efficient system would directly reconstruct the time-domain samples at (or close to) the desired signal sample rate by performing an IFFT with a length equal to the number of backhauled frequency-domain bins from an STFT. This approach results in a sparsity-proportional reconstruction process as well, where the computation required is proportional to the number of active bins. As shown in the right-hand side of Fig. 6.7, SparSDR, performs an IFFT on only part of the full window length, simultaneously downconverting the center frequency of the signal to baseband and downsampling to an appropriate sample rate.[2]

[2]If the frequency offset and sample rate are not an integer multiple of the frequency bin spacing, the time-domain samples must be corrected with additional downconversion and downsampling. Fortunately, these corrections do not add significant computational overhead because they are performed after the initial STFT-based downsampling.

6.2.5 Phase offset compensation

Indeed, a partial IFFT can reconstruct the time-domain signal. However, it also introduces a time-domain discontinuity in the signal. The reason is that taking the partial IFFT implicitly downconverts the signal to baseband. This change in center frequency creates phase discontinuities at the boundaries of the time-domain windows. To better understand this artifact, consider the example depicted in Fig. 6.8. A sinusoid at frequency f_c is converted to the frequency domain by an STFT across three windows (top left). The time-domain signal after an IFFT is a perfect sinusoid (top right). If we perform a simple downconversion by shifting the bins to the center frequency (lower left), changing the frequency introduces a phase discontinuity at each window boundary (lower right).

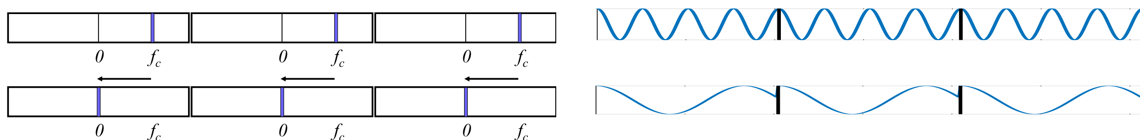


Figure 6.8: Downconverting the frequency-domain bins results in a discontinuity in the time domain.

The discontinuity in phase is due to the windowing effects of the STFT. Downconverting a signal in the time domain involves multiplying the signal by a sinusoid $e^{(-j2\pi f_c)}$ at the signal's center frequency f_c. In the frequency domain, this operation is equivalent to shifting the frequency axis by f_c as follows:

$$X(f - f_c) \leftrightarrow x(t)\,e^{-j2\pi f_c t}.$$

In a windowed FFT, the start time of each segment increments for each subsequent segment. Therefore, each segment has an additional phase term which is proportional to the time at which the segment was captured t_{start} and the shift in frequency f_c. For each segment $x_k(t) = x(t_{start} + (0 :$

$\frac{N-1}{f_s}$)), the downconversion can be computed as:

$$X_k(f-f_c) \leftrightarrow x_k(t)\, e^{-j2\pi f_c\left(0:\frac{N-1}{f_s}\right)}$$
$$= x\left(t_{start} + (0:\frac{N-1}{f_s})\right) e^{-j2\pi f_c\left(0:\frac{N-1}{f_s}\right)}.$$

When taking an IFFT of part of a full-length FFT window, this introduces a downconversion that is equivalent to multiplying each window with a sinusoid of zero initial phase $e^{j2\pi f_c(0:(l-1)/f_s)}$, where l is the length of the partial IFFT. In other words, taking a partial IFFT is equivalent to multiplying the signal in the time domain with a sinusoid at f_c whose phase begins at zero at the start of each window—independent of the time at which the window was captured. Therefore, SparSDR can compensate for the phase discontinuity by introducing an additional phase offset of $e^{-j2\pi f_c t_{start}}$. The result extends to the overlapped STFT windows that are needed to faithfully reconstruct signals; the phase correction must be applied before overlapping the windows.

6.3 Evaluation

In this section we evaluate how the parameterization of SparSDR affects the efficiency of downsampling in terms of backhaul throughput. We evaluate three parameters: STFT window type, STFT length, and threshold value. This investigation reveals the tradeoffs that must be made between efficiency and reconstruction accuracy.

6.3.1 STFT window function

In the first experiment, we evaluate how the selection of a window function affects SparSDR's ability to backhaul only the bins needed to faithfully reconstruct a signal. For each window function, we observe the decrease in the reconstruction error of a sinusoid as we increase the number of bins used for reconstruction. For all window functions, we use a fixed STFT length

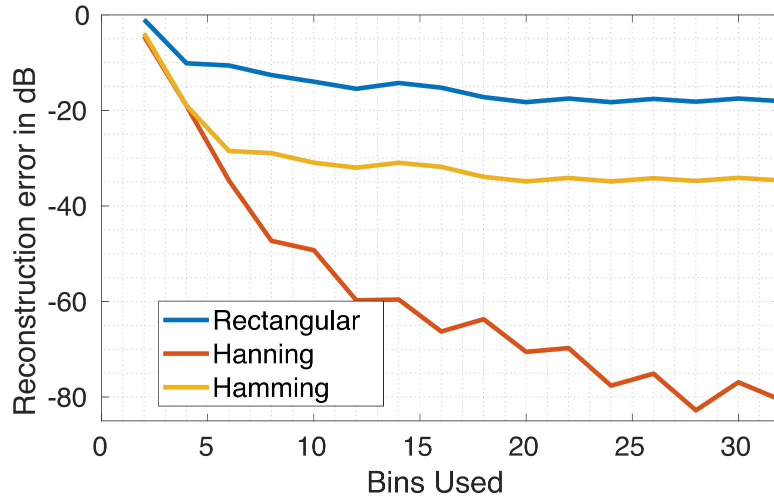


Figure 6.9: Window function affects the number of bins that must be backhauled to accurately reconstruct a sinusoid.

of 2048, and for the cosine windows we add 50% overlap. We measure the reconstruction error as the Error Vector Magnitude (EVM) [53, Chapter 5.1.3] of the ideal sinusoid compared to the reconstructed signal.

Fig. 6.9 shows the results of this experiment. As more bins are used, the rectangular and Hamming windowed STFT do not provide a significant reduction in error. However, the Hanning window provides a significant drop in reconstruction error with each new bin that is added: backhauling 30 bins for every 2048 time-domain samples results in an extremely low reconstruction error (-60 dB). (Two bins must be backhauled for each STFT bin due to the required 50% overlap.)

6.3.2 STFT length

Next we evaluate how the STFT length affects the fraction of the STFT bins that need to be backhauled to faithfully reconstruct the signal. The benefit of using a longer STFT is that it increases the STFT's frequency precision: each frequency bin represents a smaller frequency band. Although longer STFTs require backhauling more bins to represent a fixed bandwidth, these bins also represent an equally longer period of time. Therefore, the backhaul bandwidth is

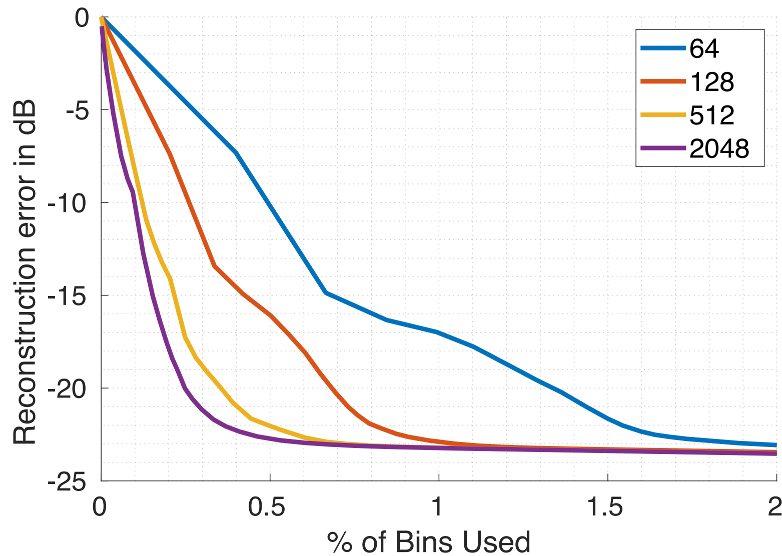


Figure 6.10: Longer STFTs result in improved backhaul efficiency without degrading reconstruction fidelity.

proportional to the fraction of the STFT bins, regardless of the STFT length.

We expect that increasing STFT length will reduce the fraction of bins that need to be backhauled. The reason is that the number of bins into which a signal falls depends mainly on the window type. For example, Hanning window's main lobe primary leaks into two adjacent bins for STFT lengths below 2048. Therefore, a signal that spans n bins would require backhauling two additional bins on each side. This results in an overhead of $4/(n+4)$. As the STFT length decreases, the number of bins needed for a signal also decreases, yielding an increase in the relative overhead.

To compare the performance of different STFT lengths, we generate a 1-MHz Bluetooth signal at 100 Msps using MATLAB. This is a realistic signal that requires backhauling multiple bins to reconstruct with high fidelity. Fig. 6.10 plots the results of capturing this signal using four STFT lengths (64, 128, 512, 2048) with a 50%-overlapped Hanning window. We reconstruct the signal with an increasing fraction of the STFT length, and compute the reconstruction error. Fig. 6.10 shows the results of this experiment. As expected, larger STFTs achieve a lower reconstruction error rate for any given fraction of bins, and achieve very low error with a small fraction of bins (-23 dB while using only half a percent in this instance).

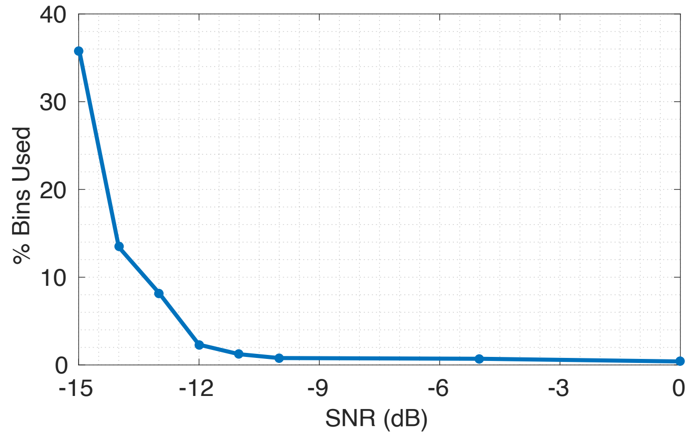


Figure 6.11: Average fraction of STFT bins that were above the threshold to achieve BER of less than 10^{-5} for different SNR values. Evaluated signal is IEEE 802.15.4 (ZigBee) PHY with 2 MHz of bandwidth.

6.3.3 Threshold value

Finally, we evaluate how the energy threshold value we select affects the fidelity of the reconstructed signals. Lowering the threshold to a level close to noise increases the rate of spuriously backhauled bins. Increasing the threshold significantly above the noise may reduce the sensitivity to weak signals.

Detecting signals in SparSDR is made easier because the SDR frontend oversamples (captures at a sample rate higher than the required for the signal). Performing an STFT on oversampled signals introduces averaging gain in the frequency domain, effectively lowering the noise floor. This separates weak signals from the noise floor, making them easier to detect. To be precise, there is $10 \cdot \log_{10}\left(\text{capture}_{\text{bw}}/\text{signal}_{\text{bw}}\right)$ averaging gain added by performing an STFT. Hence, if the capture bandwidth is wide enough, below-the-noise signals, such as weak ZigBee transmissions, can be detected.

In the following experiment, we evaluate the effectiveness of thresholding to detect below-the-noise signals (i.e., negative SNR). We generated IEEE 802.15.4 (ZigBee) signals and added noise that was equal to or higher power than the signal. We detected the signals with threshold values in steps of 1 dB for each SNR, reconstructed the signals with only these bins, and decoded the signals to measure their BER. We found the maximum threshold that resulted

in a Bit Error Rate (BER) of less than 10^{-5}: we call this the "maximum decodable threshold". Building on the results of the prior experiments in this section, these experiments are run with a Hanning-windowed 2048-bin STFT. The 2 Msps ZigBee samples were then upsampled to the full capture bandwidth of 100 Msps before applying the STFT.

In this experiment, the averaging gain results in an increase of $10 \cdot \log_{10}(50) = 17$ dB in SNR, effectively separating the signal from background noise. As described in Sec. 6.2.3, the bandwidth of the signal is 2% of the full capture bandwidth, so we expect 2% or fewer of the bins to be above the threshold.

Fig. 6.11 presents the average fraction of the STFT bins that were above the maximum decodable threshold for SNRs ranging from -15 to 0 dB. At -12 dB SNR and above, the minimum decodable threshold selects at most 2% of the bins. Between -12 dB and -15 dB SNR, there is a rapid increase in the number of bins above the minimum decodable threshold. For SNR -15 dB and below, backhauling all the bins is not sufficient to capture a decodable signal.

6.4 Implementation

In this section, we describe how to add SparSDR to an SDR's FPGA-based frontend and backend software stack. The frontend modification replaces the existing downsampling modules in the FPGA with SparSDR hardware modules, and backend reconstruction is performed on the host within GNU Radio. Specifically, we detail the downsampling hardware design, backhaul packet format, and backend reconstruction software. We demonstrate that SparSDR fits in the resources of typical SDRs, and that the design is portable to many different SDR platforms.

6.4.1 SparSDR's downsampling hardware

We begin by describing how SparSDR's frontend architecture can downsample the full capture bandwidth in real time using the limited resources of an SDR frontend's FPGA. Our

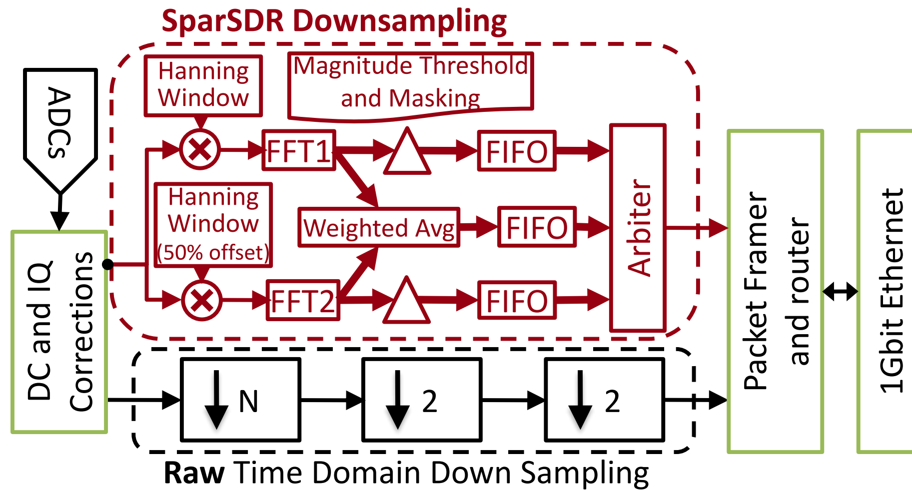


Figure 6.12: Overview of SparSDR's FPGA-based downsampling pipeline. The time-domain downsampling module is replaced by the frequency-domain SparSDR downsampling module.

implementation fits into the spare resources of two resource-limited SDRs: the classic USRP N210 and low-end AD Pluto. Further, we describe how the SparSDR implementation is portable to other SDR platforms because of its parametrized design, as well as being lightweight enough to fit in their spare resources. Finally, we describe how STFT length and other parameters affect the FPGA resource requirements.

Architecture

Fig. 6.12 shows the overall architecture of the SparSDR downsampler for the USRP N210. The first step is to transform the time-domain samples captured by the ADC into the frequency domain. The challenge is that the N210's FPGA runs at the same clock frequency as the full-capture sample rate; therefore the design needs to be fully pipelined so that at every clock cycle a new time sample is accepted, and an output frequency sample is passed through the SparSDR downsampling filter.

To address this challenge, we use the streaming version of the Xilinx FFT core. Since SparSDR requires overlapping windows by 50%, we instantiate two of these FFT modules and set their start triggers to fire half a window size apart. We store the parameters of SparSDR locally in block RAMs. These include constant values such as the Hanning-window coefficients, per-bin

threshold values, and bin masks. We implement windowing in a streaming manner as well by reading two coefficients half a window apart and multiplying them by the time-domain samples before they are input into the FFT. The outputs of the two FFT modules are also half a window apart. Each output bin is individually compared to the threshold value and mask. If the magnitude of the bin is above the threshold and the bin is not masked, then that bin is sent to the network module so it can be backhauled to the SDR host.

Due to the sparsity of SparSDR downsampling, it is not possible to predict when a bin will be above the threshold or which of the two overlapped FFTs will have an above-threshold bin. To resolve this problem and make reconstruction more straightforward, we include an arbiter module that sends all the active bins from one FFT window before switching to the next overlapped window generated by the other FFT module. On top of that, the periodic FFT bin magnitude averages have higher priority than FFT bin values. The arbiter orchestrates the order which data is sent to the networking module. This reordering and interruption by average samples, as well as potential network bottlenecks, requires FIFOs for the FFT outputs and also the averaging module.

Backhaul packet format

Next, we describe the format for backhauling STFT bins from the SDR frontend to the host. The information needed to reconstruct the signal from the frequency domain bins is as follows: alongside the I/Q value of the FFT bin, we need to send the STFT bin index and the timestamp of the STFT. The I/Q data for an FFT bin is 4 bytes, and we we use an additional 4 bytes per FFT bin for the metadata. The maximum FFT size we support in the USRP N210 implementation of SparSDR is 2048, so 11 bits are used for the FFT index, 1 bit is used to distinguish an FFT sample from an average, and 20 bits remain for the timestamp. To make it possible for the host to distinguish between the two overlapped FFTs, the timestamp includes the start time of each half window.

To make the timestamp range as large as possible, we synchronize the FFT bin 0 output to

occur when the USRP N210's internal timer lower bits are zero. Hence, the timestamp of each half window has trailing zeros that can be dropped. For an FFT of size 2048, this process results in a total timestamp size of 30 bits—20 bits of metadata and 10 bits from the zeros for each half window—sufficient for 10.7 seconds at the 10-ns sample clock interval of the USRP N210.

Portability

To make SparSDR portable to other popular SDRs that have different resource constraints, we parameterize the hardware code. There are three design parameters that can be passed to the SparSDR modules: the maximum length of the FFT that an implementation should support (fixed at synthesis time to a power of two), the size of the FIFO buffer for sending FFT samples, and the size of the FIFO buffer for sending average samples. These parameters are governed by the maximum sample rate of the SDR frontend, as well as the spare resources available on the FPGA (see below). In addition, to achieve timer and FFT output synchronization, FFT module latency values (shown in Table 6.4) need to be updated in a specific Verilog file. These latencies can be different for different FPGAs.

The primary challenge of porting SparSDR to a new platform is understanding the top module of the new platform and how the samples are backhauled. For example, porting from the USRP N210 to the AD Pluto took about two weeks. Most of the time was spent understanding the hardware design and software stack of the Pluto, followed by developing an FFT wrapper for the AXI-Stream version of the Xilinx FFT core. Additional hardware development effort would be required to use an FFT module other than the legacy or AXI-Stream versions of the Xilinx FFT module.

Resource utilization

Driven by our evaluation that demonstrates that longer FFTs produce more efficient backhaul usage (Section 6.3.2), we tried to fit the largest FFT length possible into the remaining

Table 6.1: FPGA resource requirements for different maximum FFT window sizes.

	Use CLB Logic				Use XtremeDSP Slices			
MAX FFT size	256	512	1024	2048	256	512	1024	2048
DSP	12	16	16	20	30	38	40	48
Block RAM	1	3	4	7	1	3	4	7

FPGA resources. As shown in Table 6.1, the maximum FFT length dictates the resources needed for the FFT module. Furthermore, changing the length proportionally changes the required memory for thresholds, masking, and averaging (Table 6.2) because these units use Block RAMs to hold the constant coefficients or values for each bin. Finally, the FIFO sizes determine how well the system can tolerate bursts of data for FFT output and running averages.

Table 6.2: Resource utilization for different modules in our USRP N210 implementation.

Module	Slice Reg	LUT	LUT RAM	BRAM	DSP48A	Instances
SparSDR Top	623	1022	128	0	16	1
FFT	4006	4121	1191	7	20	2
Thresholds Mem	0	0	0	4	0	1
Averages Mem	0	0	0	2	0	2
Masks mem	0	0	0	1	0	1
FFT Samples FIFO	23	61	0	4	0	2
Avg. Samples FIFO	25	67	0	7	0	1
Backhaul send FIFO	29	71	0	34	0	1

Satisfying these constraints resulted in a length-2048 FFT for the USRP N210 and length-1024 FFT for the AD Pluto. In addition to the scalable FFT-related modules, the top module requires a fixed amount of resources for a command decoder as well as windowing and average calculations which require DSP blocks. The sampling rates of the N210 and Pluto platforms are 100 Msps and 61.44 Msps respectively. The top of Table 6.3 shows the resource utilization of SparSDR on these platforms compared to their baseline implementations. The bottom of Table 6.3 shows available FPGA resources of some popular SDR modules. Our lightweight implementation can easily fit into high-end FPGAs like the one in the USRP X310.

Table 6.3: Resource requirements for our SparSDR implementations, and available FPGA resources of various popular SDR models.

Implementation / SDR module	Slice registers	LUT	BRAM	DSP	Price ($)
Baseline N210	20287	31248	41 x 18Kb	31 x 18b*18b	-
N210 + SparSDR	29066	39935	115 x 18Kb	87 x 18b*18b	-
Baseline Pluto	14365	7801	4 x 18Kb	56 x 18b*25b	-
Pluto + SparSDR	23103	13682	33 x 18Kb	74 x 18b*25b	-
AD Pluto	32500	17600	120 x 18Kb	80 x 18b*25b	150
USRP N210	47744	47744	126 x 18Kb	126 x 18b*18b	2000
USRP E310	106400	53200	280 x 18Kb	220 x 18b*25b	3050
USRP B210	184304	92152	268 x 18Kb	180 x 18b*18b	1200
USRP X310	508400	254200	1540 x 18Kb	1590 x 18b*25b	5400

Latency

Table 6.4 shows the latency, for different window sizes, added by the FFT module with a maximum FFT window size of 2048. Windowing and thresholding add another 5 cycles to these FFT latencies. The monitoring module calculates and updates the running average per bin, but it is not on the signal latency path. Since the arbiter orders the windows, there is a half window delay for sending the second half of the previous window before starting the new window. The other source of delay is periodic transmission of full average window values. As such, the primary parameter to adjust the latency added by SparSDR's downsampling hardware is FFT length. In addition, FFT and average sample FIFOs add some delay, but these delays are negligible compared to the latency introduced by the network module. For instance, the network module may not transmit a packet to the host until it is filled, and there are kernel latencies on the host side.

Table 6.4: Latency of the FFT module for different window sizes.

FFT size	8	16	32	64	128	256	512	1024	2048
Latency (cycles)	49	105	137	215	343	613	1125	2162	4210

6.4.2 SparSDR's host-based reconstruction

The software implementation of SparSDR reconstruction can provide real-time performance even on an embedded processor, and is integrated into GNU Radio. We designed a custom GNU Radio block that performs SparSDR's reconstruction. The SparSDR block produces reconstructed I/Q samples that can be passed to existing signal processing software (e.g., decoders).

SparSDR's host software can send commands to the SDR frontend to update SparSDR's parameters without resynthesizing the hardware. These parameters include FFT length, threshold and mask values, windowing coefficients, averaging weight, and interval between average samples. The software can also disable output of FFT samples or average samples. This allows each application to tailor these parameters in real time based on the performance tradeoffs described earlier in this section.

The software also provides error reporting. Namely, if the SDR frontend tries to send bins faster than the network and software can handle, the software detects the overflow. If the user desires, the software can configure the parameters to avoid overflow and resume the capture.

6.5 Case studies

In this section we demonstrate the power of SparSDR with two case studies: an SDR-based IoT gateway and a Cloud SDR for the VHF and UHF bands. We select these applications because they both involve processing sparse signals across wide capture bandwidth on platforms with limited resources. Specifically, the IoT gateway has constrained compute resources, while the Cloud SDR has constrained backhaul bandwidth. A summary of the benefits of using SparSDR instead of a full-capture SDR for these applications is as follows:

- **IoT Gateway:** Using SparSDR, Bluetooth signal processing can be performed on an SDR host with compute resources proportional to the activity rather than the 80-MHz

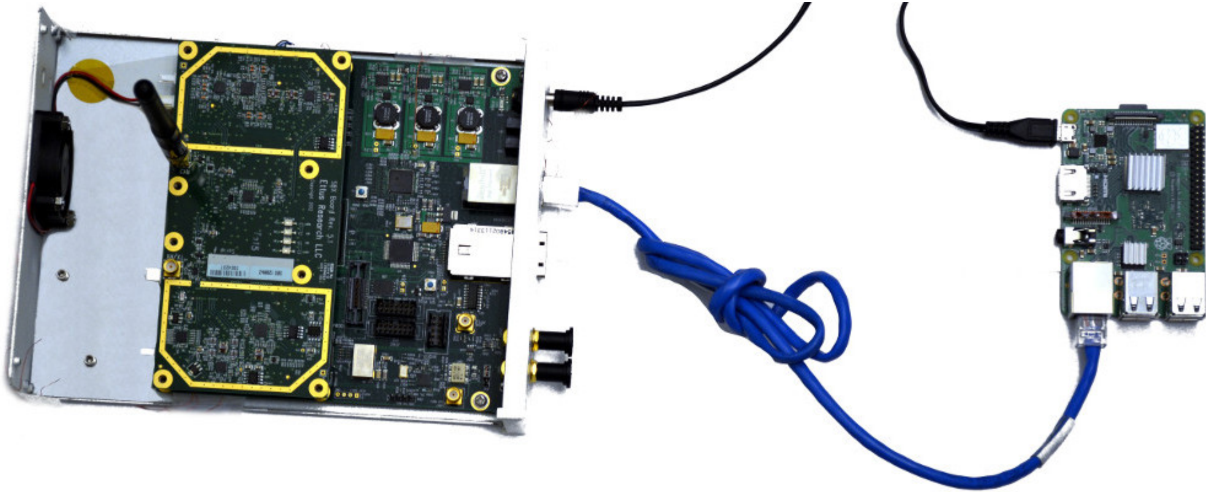


Figure 6.13: SparSDR reconstructs 100-MHz captures in near real time on a Raspberry Pi.

bandwidth of Bluetooth. We demonstrate that reconstruction processing is so lightweight that a Raspberry Pi 3+ can process 450 BLE packets per second from channels spread across the entire 80-MHz spectrum.

- **Cloud SDR:** With SparSDR, we perform an over-the-air experiment that demonstrates that most VHF and UHF bands ($\leq$ 10 MHz) are sparse, and therefore only require residential-class Internet uplink speeds to backhaul the signals to the cloud for processing.

6.5.1 IoT gateway

For many IoT applications, sensors intermittently upload data to an IoT gateway [10]. Monitoring all of the transmissions in the 100-MHz ISM band from Bluetooth [20], BLE [49], and ZigBee [165] simultaneously creates new capabilities. For instance, such a system could reduce the energy wasted by sensors channel hopping to find a channel in common with the IoT gateway. It also makes it possible to build a universal IoT gateway that is compatible with the myriad current and future IoT protocols [5, 103].

Today the only SDR platform that could achieve this would be a wideband SDR frontend with at least 10 Gbps of backhaul capacity, such as the USRP X310. Processing this sample

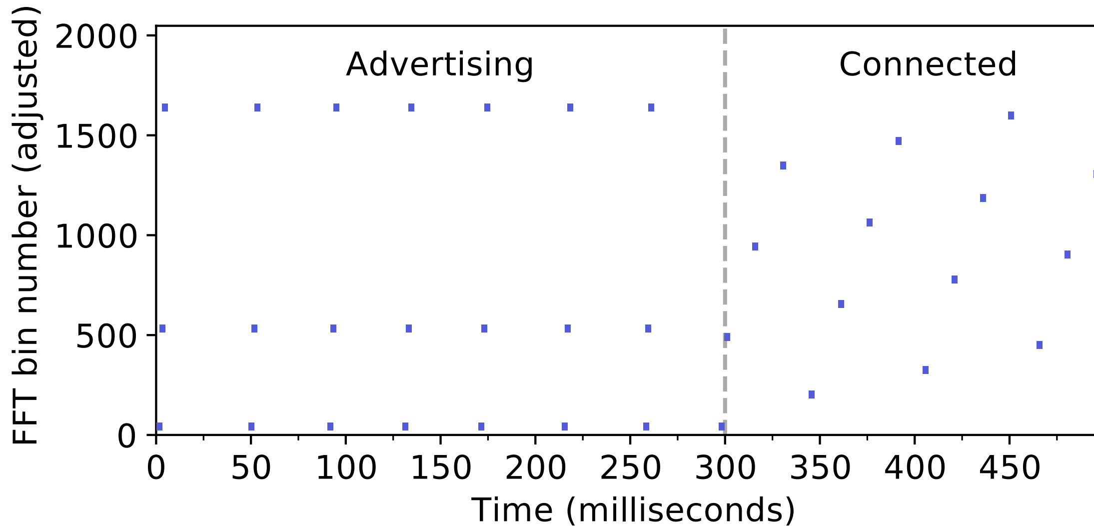


Figure 6.14: IoT transmissions are sparse in time and frequency. For both BLE modes of operation (advertising and connected) SparSDR only backhauls active FFT bins—about 45 bins—of a 2048 length FFT capturing 100 MHz.

stream would require a desktop-class processor. We demonstrate that the sparsity-proportional compute enabled by SparSDR makes it possible to monitor all of the popular IoT frequencies, and even decode signals, with the embedded-class processor of a Raspberry Pi 3+.

To demonstrate the benefits of SparSDR as an IoT gateway, we perform three controlled experiments on the following testbed: We connect a USRP N210 with the SparSDR module to a Raspberry Pi 3+ as shown in Fig. 6.13. We use the SBX-120-MHz RF frontend on the USRP N210 operating at full bandwidth and full (100-Msps) sampling rate. Our "sensors" are up to three ESP32 modules that we use to transmit BLE packets. First, we evaluate SparSDR's ability to receive BLE's wideband frequency-hopping transmissions. Then, we demonstrate how lightweight the reconstruction processing is and how much delay it adds to the system. Finally, we stress-test SparSDR's backhaul and processing capabilities.

Receiving wideband frequency-hopping transmissions

First we demonstrate that SparSDR is able to continuously monitor the full 80-MHz band for BLE packets with embedded-class compute. We set up one ESP32 as a BLE sensor that advertises a GATT service. Another ESP32 connects to the first and exchanges data. Fig. 6.14

shows the STFT bins that are backhauled to the SDR host during this experiment. SparSDR only backhauls bins during the short advertisement packets across all three advertising channels. It works similarly during the frequency-hopping exchange across all 40 BLE channels. BLE transmissions are ideal for SparSDR because they are sparse in both time and frequency. A single BLE transmitter uses at most 2 MHz of bandwidth at a time, or about 45 active FFT bins, which translates to a peak backhaul throughput of 280 Mbps. This is amortized over time due to the bursty nature of BLE signals.

Lightweight reconstruction and induced latency

To evaluate the latency introduced by reconstruction in SparSDR, we transmit BLE packets to SparSDR from an ESP32 on a fixed band. We then decode the BLE packets on the host with *gr-bluetooth*. We measure two different latencies: first, the time from the first bin backhauled for a packet to when that sample is reconstructed, and second to when the packet is fully decoded. Fig.6.15 shows the distribution of SparSDR's reconstruction latency. The reconstruction latency is $\sim$200 μs for 80% of the packets, while decoding latency is between 10 ms to 100 ms for 80% of the packets. This demonstrates that the reconstruction latency is negligible compared to the Bluetooth decode latency.

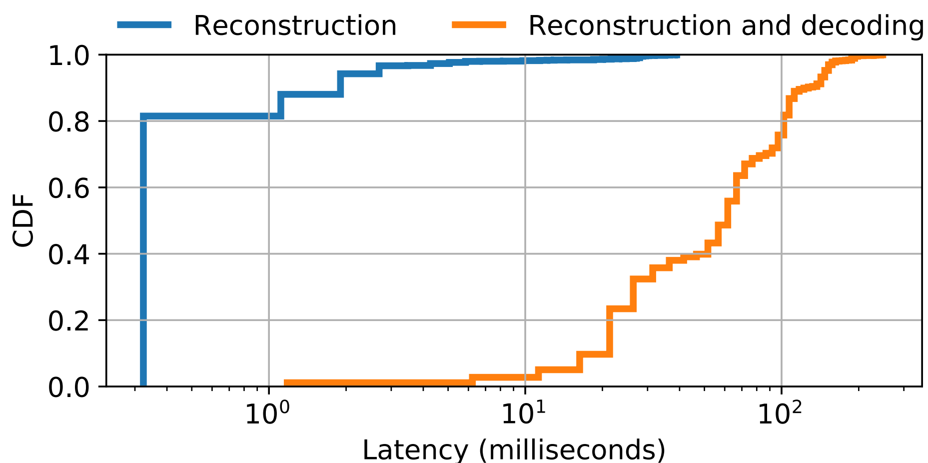


Figure 6.15: Added latency from reconstruction is significantly less than decode latency.

Backhaul and processing requirements on RPi

Next we evaluate how efficiently SparSDR reduces backhaul and processing requirements. We set up three ESP32s to advertise on the three BLE advertising channels. With just a Raspberry Pi, we monitor 100 MHz of bandwidth and decode up to 450 BLE advertisement packets per second. Fig. 6.16 shows the CPU utilization and required backhaul data rate as the BLE packet rate increases from the three transmitters. The reconstruction and decoding processes use all four CPU cores and do not fully saturate the CPU until the packet rate reaches 450 packets per second. We verify that the decoder was operating without errors. It is important to note that the Raspberry Pi 3+ has a 1-Gbps network interface, but it is connected using USB 2.0 so it can only achieve 224 Mbps.

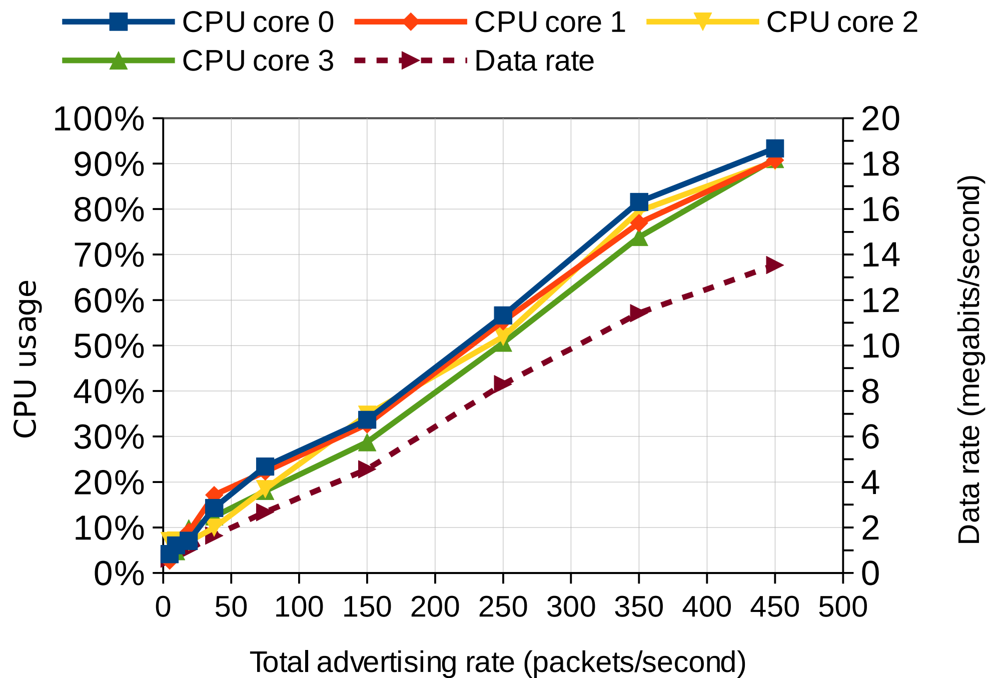


Figure 6.16: SparSDR's backhaul and compute scale linearly with the rate of received BLE advertising packets.

We acknowledge that the current system is not a full IoT Gateway. In these experiments we only focus on the receive side, and do not implement a multi-protocol scheduler. For multi-protocol decoding, there is a need to either run a process for each protocol or operate a scheduler

that distributes packets among a limited number of decoders. Designing such a system is beyond the scope of this chapter.

6.5.2 Cloud SDR

The next case study shows that SparSDR makes it possible to expand Cloud SDR deployments to wideband SDRs. We demonstrate that SparSDR significantly reduces the backhaul bandwidth requirements of a Cloud SDR: specifically, backhauling the VHF and UHF bands only requires residential-class Internet uplink speeds.

We set up a USRP N210-based SparSDR receiver to measure the backhaul data rate required to capture real-world over-the-air signals from a roof-mounted antenna. We mount the antennas on a building in a populated area near waterways, an airport, and a large military base. We use a dipole VHF antenna with a WBX-40-MHz frontend to capture signals below 400 MHz, and wideband discone antenna with an SBX-120 MHz for the UHF frequencies. We measure the backhaul data rate for each of the 10-MHz bands from 50 MHz to 1 GHz.

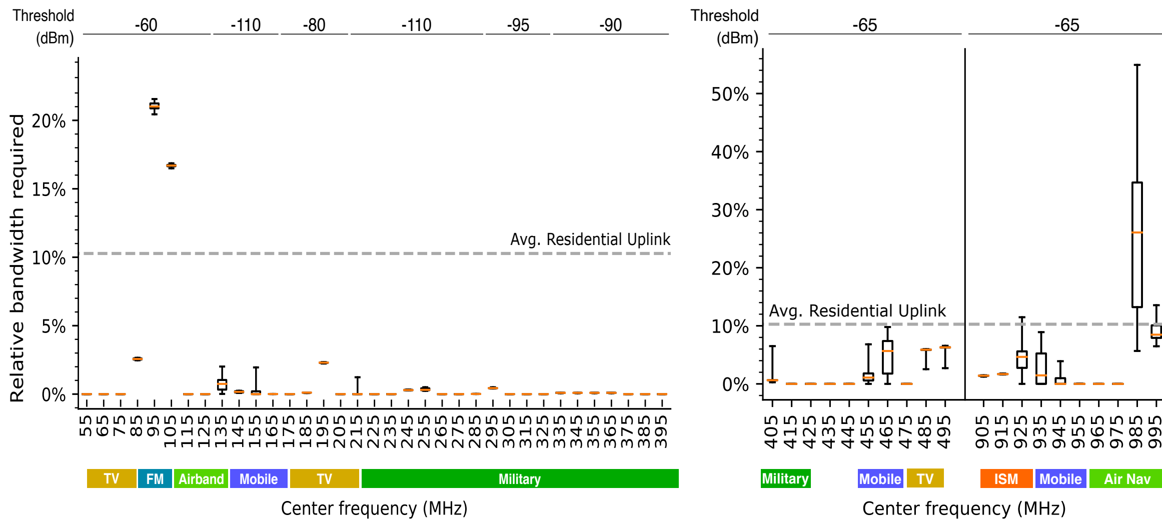


Figure 6.17: The distribution of SparSDR's backhaul for capturing transmissions in 10-MHz bands from 50 MHz to 1 GHz. Bandwidth is plotted relative to the 320 Mbps needed by a standard 10-MHz SDR. We also indicate the capacity of a typical residential-class Internet uplink.

Fig. 6.17 shows the observed distributions of the per-second backhaul data rate computed

over a period of one minute. For each 10-MHz band, the box plot shows the 0%, 25%, 50% and 75%, and 100% quartiles of the data rate. The *y* axes plots the SparSDR backhaul rate relative to the full-capture data rate (320 Mbps). The plot also shows how the backhaul data rate compares to the bandwidth of a typical U.S. residential internet connection uplink (32.88 Mbps)[3]. We break the frequency range into 10-MHz bands because that is the approximate allocation size of the different uses of the VHF and UHF bands (shown on the bottom of the figure). Fig. 6.17 also shows the absolute power threshold value that we employ for each band. These thresholds are selected by finding the minimum threshold value that does not cause overflow due to noise. We determine the absolute power by calibrating the USRP power measurements with a signal generator.

The primary result of this case study is that SparSDR's backhaul data rate is often significantly below the backhaul bandwidth required for backhauling a single 10-MHz band, or even several adjacent 10-MHz bands. The "Mobile" and "ISM" bands are the best use cases for SparSDR because they contain dynamic transmissions. For many of these bands, the entire distribution of throughput measurements are well below the typical residential uplink speed.

As expected, the bands that have constant broadcasters (e.g., FM) do not benefit much from sparsity-proportional downsampling. This is also why we do not present the bands between 500 MHz and 900 MHz because they are occupied by constant LTE downlink channels and broadcast TV stations.

6.6 Summary

SparSDR significantly reduces the backhaul and compute requirement for Software Defined Radios, which makes it possible for base stations to benefit from the hardware development cost reductions offered by SDRs. It can be implemented on many SDR front-end FPGAs avail-

[3]https://www.speedtest.net/reports/united-states/2018/fixed

able in base stations to make their backhaul capacity requirements inversely proportional to the sparsity—both in time and frequency—of the signal in any part of the RF spectrum, instead of the full ADC capture bandwidth. Further, SparSDR employs a reconstruction process whose computational requirements also scale in response to the sparsity. Hence, it significantly decreases processing demands for datacenter compute—so much so that many applications can be run on an embedded processor. SparSDR delivers near-real-time performance without sacrificing signal quality or flexibility. We have released SparSDR under an open source license to enable others to develop new applications on top of it.

6.7 Acknowledgments

Chapter 6 contains reprints of M. Khazraee, Y. Guddeti, S.Crow, A. Snoeren, K. Levchenko, D. Bharadia, A. Schulman, "SparSDR: Sparsity-proportional Wideband SDRs", *Mobisys*, 2019. The dissertation author is the primary author of this paper.

Chapter 7

Conclusions And Open Questions

In this dissertation I developed frameworks to lower development costs for different parts of the cloud infrastructure. I defended the following thesis:

"It is feasible to reduce the development costs of custom hardware accelerators in the cloud infrastructure by developing the following frameworks: 1) A framework for ASIC-based compute in datacenters which balances TCO and NRE costs and finds the optimal total cost solution, 2) A framework for FPGA-based network middle-boxes where control portion of compute is offloaded to the software running on wimpy general-purpose processors, and 3) A framework for FPGA capable base stations where backhaul and compute of software defined radios are proportional to signal sparsity."

By considering different parameters in design of ASICs, including the silicon technology, I made a framework to find the total-cost optimal solution for ASIC-based cloud hardware accelerators. The promise of total cost reduction from fully customized ASICs in cloud infrastructure suggests that both cloud providers and silicon foundries would benefit by investing in technologies that reduce the NRE of ASIC design, including open source IP such as RISC-V, new labor-saving development methodologies for hardware, and also in open source backend CAD tools. Over time, mask costs will continue to drop. In fact older silicon nodes such as 40 nm are likely to

provide suitable TCO per op/s reduction, with half the mask cost and only a small difference in performance and energy efficiency from 28 nm. Governments should also consider investing in reducing the development cost of ASIC-based cloud infrastructure, because they have the potential to reduce the exponentially growing environmental impact of datacenters across the world. Foundries should also invest in ASIC development, because ASICs' low-voltage operation leads to greater silicon wafer consumption than CPUs within the same energy limits.

For in-network compute, I built a framework to benefit from flexibility and modularity of software in FPGA-based SmartNICs. I demonstrated that for in-network hardware acceleration we can benefit from hardware code reuse, and also continuously update accelerator control flow in software. I demonstrated that this framework can significantly lower the development time and provide the required agility for network middle-boxes.

Finally, for base stations in the cloud edge that are equipped with FPGAs, I developed a framework to lower the cost of hardware-accelerated signal-processing by benefiting from sparsity in time and frequency. I showed that this framework provides the protocol-universal nature of Software Defined Radios, but with significantly lower backhaul and compute requirements. With this framework, wireless networks can benefit from the development cost reductions offered by SDRs without facing their backhaul and compute challenges.

7.1 Longevity

The methods developed for this dissertation where frameworks for developing custom hardware accelerators for ASIC and FPGA platforms. Note that none of the frameworks were tied to an specific FPGA or ASIC technology, and hence they can be applied to future technologies. Specifically, the frameworks presented in this dissertation will have longevity for the following reasons:

The NRE model in Chapter 3 was the first comprehensive model of ASIC NRE in

academia. It can add a new grounding to future evaluations of hardware accelerator designs. Designers can use our model to estimate the cost of manufacturing and use our framework to factor in design time and other missing parts of the NRE cost. The fundamental parts of NRE breakdown are likely to remain the same, and included prices can be updated as technology advances.

The ASIC cloud model was the first paper in academia to present such extreme customization. It opens up better elaboration and use of ASIC Clouds in the future, which based on improvements in TCO and increase in demands are growing rapidly. The methodology to evaluate the optimal server does not change fundamentally, as they were rooted in physics. Also, new silicon technologies or cooling systems can be added to the methodology as parameters.

The Dastgāh framework is the first in-network compute framework to demonstrate the the flexibility needed by networked systems can be achieved without incurring the full FPGA development cost for each change. The main insights of existence of packets in networks and functional acceleration taking the majority of processing time are inherent characteristics in networked applications. Therefore, I believe this work will continue to be relevant to future hardware accelerated networked systems. Also, because there is a software component, it may affect future ASIC designs.

Moreover, it is likely that non-networked applications that have high data-rates, such as machine learning, can benefit from Dastgāh framework to lower hardware acceleration development cost. For instance, the Microsoft Catapult project moved from point-to-point connection among their FPGAs to using a lightweight network protocol among their distributed accelerators for better scalability [117]. This framework can reduce development cost and effort for these previously non-networked applications that need to connect many accelerators to scale processing throughput.

Finally, SparSDR significantly increased the practicality of Software Defined Radios, and also reduced the minimum unit for a required bandwidth. This can make be highly beneficial to

development of 5G cellular networks, and also make the idea of cloud-based signal processing for wireless communications a reality.

7.2 Open questions

I now end my dissertation with a few significant open questions raised by my work:

Incorporating hardware accelerators brings an order-of-magnitude improvement for an application, but the opportunity for further improvements with silicon technology is marginal. As we get closer to physical limits for shrinking transistors [82], the cost of newer silicon technology might not be justifiable, even for massive workloads. One question is if the computation demand keeps increasing, and network bandwidth and storage can still support the increase, what will be the new cost-optimal solution? One potential solution might be lower-latency and higher-throughput communication among ASICs and FPGAs, such as new silicon photonics technologies. Benefiting from these connection among chips, we can distribute the compute more efficiently and alleviate heat dissipation problems. Also, we can make high-capacity high-bandwidth low-latency memory available for our hardware accelerators. These new technologies however will change how the chip design and ASIC server design parameters are selected. For example, based on the IP cost overhead of silicon photonics, what will be the optimal silicon technology and silicon die size per chip to balance the development cost and performance of hardware accelerators?

One question introduced by Dastgāh is, can we implement network middle-boxes with throughput in order of tera-bits per second? This maybe needed as technologies such as silicon photonics can increase link speeds dramatically, making multi-chip network processing possible. The architecture of Dastgāh was designed with this sort of scalability in mind. For example, the flow of data in system is already hierarchical and one directional. Also we can implement the scheduler in a hierarchical manner: each chip can have a local scheduler to distribute load among the processing cores, and there is a distributor chip that has a more capable scheduler and is in

charge of load distribution among the chips. The questions are, what will be the added latency of this system? and how we can design high-throughput yet efficient schedulers for this new system?

The main limitation of SparSDR framework is the added latency to detect and extract signals in hardware. For many systems the added latency of this processing, which is in order of ~100μs, is acceptable. Also, reduced backhaul bandwidth can reduce the latency added by the software running on general-purpose processor, and hence lower the overall latency. However, SparSDR does introduce some added latency that maybe unacceptable for future low-latency protocols, we do not know.

Moreover; during our experiments we found that setting the threshold properly is critical for performance and reliability of SparSDR. The question is, how we can develop a system to dynamically read the monitoring information provided by SparSDR, and automatically update the threshold values? Another direction is benefiting from the low backhaul and compute requirements of SparSDR on top of low-cost SDR devices and embedded processors, such as ADALM Pluto alongside a RPi, to make cloud-based signal processing a reality. However, there is still the question of, how we can make the software on the cloud more efficient so it can decode all active channels simultaneously?

Bibliography

[1] Mohamed S Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a chip: High performance lossless data compression on fpgas using opencl. In *IWOCL*, 2014.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*, 2015.

[3] Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Atieh Lotfi, Julian Puscar, Anuj Rao, et al. Celerity: An open source risc-v tiered accelerator fabric. In *Symp. on High Performance Chips (Hot Chips)*, 2017.

[4] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. *ACM SIGARCH Computer Architecture News*, 44(3):1–13, 2016.

[5] Gianluca Aloi, Giuseppe Caliciuri, Giancarlo Fortino, Raffaele Gravina, Pasquale Pace, Wilma Russo, and Claudio Savaglio. A mobile multi-technology gateway to enable IoT interoperability. In *2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 259–264. IEEE, 2016.

[6] Gianni Antichi, Muhammad Shahbaz, Yilong Geng, Noa Zilberman, Adam Covington, Marc Bruyere, Nick McKeown, Nick Feamster, Bob Felderman, Michaela Blott, Andrew W. Moore, and Philippe Owezarski. OSNT: Open source network tester. *IEEE Network*, September 2014.

[7] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20). USENIX Association, Santa Clara, CA*, 2020.

[8] Mina Tahmasbi Arashloo, Alexy Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[9] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The

rocket chip generator. *EEECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[10] Luigi Atziori, Antonio Iera, and Giacomo Morabito. The Internet of things: A survey, 2010.

[11] AyarLabs. *Enabling the next phase of Moore's Law through optical connectivity*, 2020. https://ayarlabs.com/.

[12] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[13] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, et al. Openpiton: An open source manycore research framework. *ACM SIGPLAN Notices*, 51(4):217–232, 2016.

[14] Manu Bansal, Aaron Schulman, and Sachin Katti. Atomix: A framework for deploying signal processing applications on wireless infrastructure. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[15] J. Barkatullah and T. Hanke. Goldstrike 1: Cointerra's first-generation cryptocurrency mining processor for bitcoin. *Micro, IEEE*, 35(2), Mar 2015.

[16] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8, 2013.

[17] Luiz Andre Barroso and Urs Hoelzle. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[18] John Beetem, Monty Denneau, and Don Weingarten. The gf11 supercomputer. In *ISCA*, 1985.

[19] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. In *ACM Computer Communication Review (CCR)*, 2014.

[20] Bluetooth SIG. *Bluetooth Core Specification*, 12 2016. Rev. 5.0.

[21] Mahdi Nazm Bojnordi and Engin Ipek. Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–13. IEEE, 2016.

[22] Manuel Bronstein. *A more helpful Google Assistant for your every day*. urlhttps://blog.google/products/assistant/ces-2020-google-assistant/.

[23] Grace Caffyn. Bitfury announces 'record' immersion cooling project. In *Coindesk*, Oct 2015.

[24] R. Calvo-Palomino, D. Giustiniano, V. Lenders, and A. Fakhreddine. Crowdsourcing spectrum data decoding. In *INFOCOM*, 2017.

[25] Roberto Calvo-Palomino, Fabio Ricciato, Domenico Giustiniano, and Vincent Lenders. Ltess-track: A precise and fast frequency offset estimation for low-cost sdr platforms. In *WINTECH*, 2017.

[26] Scott Campbell and Jason Lee. Intrusion detection at 100G. In *Proc. The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2011.

[27] Emmanuel Candes and Justin Romberg. Sparsity and incoherence in compressive sampling, 2006.

[28] Adrian M. Caulfield et al. A cloud-scale acceleration architecture. In *Proc. IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[29] R Chellappa. Cloud computing—emerging paradigm for computing. *INFORMS 1997, Dallas, TX*, 1997.

[30] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ACM SIGPLAN Notices*, 51(4):681–696, 2016.

[31] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.

[32] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

[33] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *MICRO*, 2014.

[34] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. *ACM SIGARCH Computer Architecture News*, 44(3):27–39, 2016.

[35] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor. The celerity open-source 511-core risc-v tiered accelerator fabric: Fast architectures and design methodologies for fast chips. *IEEE Micro*, 38:30–41, 2018.

[36] PP Deepthi and PS Sathidevi. Design, implementation and analysis of hardware efficient stream ciphers using lfsr based hash functions. *Computers & Security*, 28(3-4):229–241, 2009.

[37] David L. Donoho. Compressed sensing. *IEEE Trans. Inform. Theory*, 52:1289–1306, 2006.

[38] Prabal Dutta and Iqbal Mohomed. emergence of the IoT gateway platform. *GetMobile: Mobile Computing and Communications*, 19(3):33–36, 2015.

[39] Economist. *After Moore's law*, 2016. https://www.economist.com/technology-quarterly/2016-03-12/after-moores-law.

[40] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. Nica: An infrastructure for inline acceleration of network applications. In *Proc. USENIX Annual Technical Conference*, 2019.

[41] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[42] Daniel Firestone et al. Azure accelerated networking: SmartNICs in the public cloud. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[43] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. A scalable high-bandwidth architecture for lossless compression on FPGAs. In *Proc. IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2015.

[44] Eitan Frachtenberg, Ali Heydari, Harry Li, Amir Michael, Jacob Na, Avery Nisbet, and Pierluigi Sarti. High-efficiency server design. In *SC*, 2011.

[45] Vinay Gangadhar, Raghu Balasubramanian, Mario Drumond, Ziliang Guo, Jai Menon, Cherin Joseph, Robin Prakash, Sharath Prasad, Pradip Vallathol, and Karu Sankaralingam. Miaow: An open source gpgpu. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–43. IEEE, 2015.

[46] Glassdoor salaries, 2016. *Glassdoor*. https://www.glassdoor.com.

[47] Gleb Gagarin. Regular expressions to SystemVerilog compiler. https://github.com/p12nGH/re2v.

[48] Vaibhav Gogte, Aasheesh Kolli, Michael J Cafarella, Loris D'Antoni, and Thomas F Wenisch. Hare: Hardware accelerator for regular expressions. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[49] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and evaluation of Bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753, 2012.

[50] Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Jonathan Babb, Michael B Taylor, and Steven Swanson. Greendroid: A mobile application processor for a future of dark silicon. In *2010 IEEE Hot Chips 22 Symposium (HCS)*, pages 1–39. IEEE, 2010.

[51] Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha Nath, Vikram Bhatt, Jonathan Babb, et al. The greendroid mobile application processor: An architecture for silicon's dark future. *IEEE Micro*, 31(2):86–95, 2011.

[52] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. Biscuit: A framework for near-data processing of big data workloads. *ACM SIGARCH Computer Architecture News*, 44(3):153–165, 2016.

[53] Qizheng Gu. *RF System Design of Transceivers for Wireless Communications*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[54] Anthony Gutierrez, Michael Cieslak, Bharan Giridhar, Ronald G. Dreslinski, Luis Ceze, and Trevor Mudge. Integrated 3d-stacked server designs for increasing physical density of key-value stores. In *ASPLOS*, 2014.

[55] Mesud Hadzialic, Branko Dosenovic, Merim Dzaferagic, and Jasmin Musovic. Cloud-ran: Innovative radio access network architecture. In *Proceedings ELMAR-2013*, pages 115–120. IEEE, 2013.

[56] James Hale. *More Than 500 Hours Of Content Are Now Being Uploaded To YouTube Every Minute*, 2019. urlhttps://www.tubefilter.com/2019/05/07/number-hours-video-uploaded-to-youtube-per-minute/.

[57] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[58] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding

sources of inefficiency in general-purpose chips. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 37–47, 2010.

[59] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[60] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proc. ACM SIGCOMM*, 2017.

[61] Quentin Hardy. Intel betting on (customized) commodity chips for cloud computing. In *NY Times*, Dec 2014.

[62] F. J. Harris. On the use of windows for harmonic analysis with the discrete Fourier transform. *Proceedings of the IEEE*, 66(1):51–83, jan 1978.

[63] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. Simple and practical algorithm for sparse Fourier transform. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012.

[64] Haitham Hassanieh, Lixin Shi, Omid Abari, Ezzeldin Hamed, and Dina Katabi. GHz-Wide sensing and decoding using the sparse Fourier transform. In *Proc. IEEE Conference on Computer Communications (INFOCOM)*, 2014.

[65] Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, et al. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–238, 2015.

[66] Victor Heorhiadi, Michael K. Reiter, and Vyas Sekar. New opportunities for load balancing in network-wide intrusion detection systems. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012.

[67] Md Habibul Islam, Choo Leng Koh, Ser Wah Oh, Xianming Qing, Yoke Yong Lai, Cavin Wang, Ying-Chang Liang, Bee Eng Toh, Francois Chin, Geok Leng Tan, et al. Spectrum survey in singapore: Occupancy measurements and analyses. In *Cognitive Radio Oriented Wireless Networks and Communications, 2008. CrownCom 2008. 3rd International Conference on*, pages 1–7. IEEE, 2008.

[68] Anand Padmanabha Iyer, Krishna Chintalapudi, Vishnu Navda, Ramachandran Ramjee, Venkata N Padmanabhan, and Chandra R Murthy. Specnet: Spectrum sensing sans frontieres. In *Symposium on Networked Systems Design and Implementation (NSDI 11)*. USENIX, 2011.

[69] Yu Ji, YouHui Zhang, ShuangChen Li, Ping Chi, CiHang Jiang, Peng Qu, Yuan Xie, and WenGuang Chen. Neutrams: Neural network transformation and co-design under neuromorphic hardware constraints. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[70] John Lockwood. Low-Latency Library in FPGA Hardware for High-Frequency Trading, 2012. Hot Interconnects, https://www.youtube.com/watch?v=nXFcM1pGOIE.

[71] H Jones. Strategies in optimizing market positions for semiconductor vendors based on ip leverage. *IBS White Paper*, 2014.

[72] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[73] Ju et al. 18.6 a 0.5 nj/pixel 4k h. 265/hevc codec lsi for multi-format smartphone applications. In *ISSCC*, 2015.

[74] Sang Jun, Ming Liu, Sungjin Lee, Hicks, Ankcorn, Myron King, Shuotao Xu, and Arvind. Bluedbm: An appliance for big data analytics. In *ISCA 2015*, 2015.

[75] Alex Kampl. Bitcoin 2-phase immersion cooling and the implications for high performance computing. In *Electronics Cooling Magazine*, February 2014.

[76] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. *ACM SIGARCH Computer Architecture News*, 44(3):380–392, 2016.

[77] June Kim, Seungheon Hyeon, and Seungwon Choi. Implementation of an SDR system using graphics processing unit. *IEEE Communications Magazine*, March 2010.

[78] Martha Mercaldi Kim, Mojtaba Mehrara, Mark Oskin, and Todd Austin. Architectural Implications of Brick and Mortar Silicon Manufacturing. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 244–253. ACM, 2007.

[79] MICHAEL KING. Raspberry Pi to production. https://www.rigado.com/download/raspberry-pi-to-production-whitepaper/.

[80] Nikolaus Kleber, Jonathan D. Chisum, Aaron Striegel, Bertrand M. Hochwald, Abbas Termos, J. Nicholas Laneman, Zuohui Fu, and John Merritt. RadioHound: A pervasive sensing network for sub-6 GHz dynamic spectrum monitoring. *CoRR*, 2016.

[81] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *MICRO*, 2013.

[82] Suhas Kumar. Fundamental limits to moore's law, 2015.

[83] Jason Laska, Sami Kirolos, Yehia Massoud, Richard Baraniuk, Anna Gilbert, Mark Iwen, and Martin Strauss. Random sampling for analog-to-information conversion of wideband signals. In *IEEE Dallas Circuits and Systems Workshop (DCAS)*, volume 1, pages 119–122, 2006.

[84] Li et al. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ISCA*, 2015.

[85] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[86] Bojie Li, Kun Tan, Larry Luo, Renqian Luo, Yanqing Peng, Ningyi Xu, Yongqiang Xiong, and Peng Cheng. ClickNP: Highly flexible and high-performance network processing with reconfigurable hardware. In *Proc. ACM SIGCOMM*, 2016.

[87] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High precision congestion control. In *Proc. ACM SIGCOMM*, 2019.

[88] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin servers with smart pipes: Designing soc accelerators for memcached. In *ISCA*, 2013.

[89] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *ISCA*, 2008.

[90] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto SmartNICs using IPipe. In *Proc. ACM SIGCOMM*, 2019.

[91] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Cambricon: An instruction set architecture for neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 393–405. IEEE, 2016.

[92] Liu et al. Pudiannao: A polyvalent machine learning accelerator. In *ASPLOS*, 2015.

[93] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. NetFPGA–an open platform for gigabit-rate network switching and routing. In *Proc. IEEE International Conference on Microelectronic Systems Education (MSE)*, 2007.

[94] E. J. McDonald. Runtime fpga partial reconfiguration. In *2008 IEEE Aerospace Conference*, pages 1–7, 2008.

[95] Caitlin McGarry. *5G Speed: 5G vs 4G Performance Compared*, 2019. urlhttps://www.tomsguide.com/features/5g-vs-4g.

[96] Mark A. McHenry, Peter A. Tenhula, Dan McCloskey, Dennis A. Roberson, and Cynthia S. Hood. Chicago spectrum occupancy measurements & analysis and a long-term studies proposal. In *Proceedings of the First International Workshop on Technology and Policy for Accessing Spectrum*, TAPAS '06, New York, NY, USA, 2006. ACM.

[97] Lucas Mearian. *Data storage goes from $1M to 2 cents per gigabyte*, 2017. urlhttps://www.computerworld.com/article/3182207/cw50-data-storage-goes-from-1m-to-2-cents-per-gigabyte.html.

[98] Microsoft. *Project Catapult*, 2018. urlhttps://www.microsoft.com/en-us/research/project/project-catapult/.

[99] Joe Mitola. The software radio architecture. *IEEE Communications Magazine*, May 1995.

[100] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling PCRE to FPGA for accelerating Snort IDS. In *Proc. ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2007.

[101] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan. Flow-level state transition as a new switch primitive for SDN. In *Proc. ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014.

[102] Revathy Narayanan and Swarun Kumar. Revisiting software defined radios in the IoT era. In *Proc. Workshop on Hot Topics in Networks (HotNets)*. ACM, 2018.

[103] Revathy Narayanan and C Siva Ram Murthy. A probabilistic framework for protocol conversions in IIoT networks with heterogeneous gateways. *IEEE Communications Letters*, 21(11):2456–2459, 2017.

[104] Lauren E. Nelson and Richard Fichera. Vendor landscape: Private cloud overview. In *Forrester Research Report*, Oct 2015.

[105] William B. Norton. *Internet Transit Prices - Historical and Projected*. urlhttps://drpeering.net/white-papers/Internet-Transit-Pricing-Historical-And-Projected.php.

[106] S. O'Dea. *Number of mobile wireless cell sites in the United States from 2000 to 2018*. urlhttps://www.statista.com/statistics/185854/monthly-number-of-cell-sites-in-the-united-states-since-june-1986/.

[107] Alan V Oppenheim and Ronald W Schafer. Digital signal processing (book). *Research supported by the Massachusetts Institute of Technology, Bell Telephone Laboratories, and Guggenheim Foundation. Englewood Cliffs, N. J., Prentice-Hall, Inc., 1975. 598 p*, 1975.

[108] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric Chung. Toward accelerating deep learning at scale using specialized hardware in the datacenter. In *Proc. IEEE HotChips Symposium on High-Performance Chips (HotChips)*, 2015.

[109] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. Energy efficient architecture for graph analytics accelerators. *ACM SIGARCH Computer Architecture News*, 44(3):166–177, 2016.

[110] Packan et al. High performance 32nm logic technology featuring 2nd generation high-k + metal gate transistors. In *IEDM*, Dec 2009.

[111] Ehsan Pakbaznia and Massoud Pedram. Minimizing data center cooling and server power costs. In *ISLPED*, 2009.

[112] Kishor Patil, Ramjee Prasad, and Knud Skouby. A survey of worldwide spectrum occupancy measurement campaigns for cognitive radio. In *Devices and Communications (ICDeCom), 2011 International Conference on*, pages 1–5. IEEE, 2011.

[113] Ardavan Pedram, Stephen Richardson, Mark Horowitz, Sameh Galal, and Shahar Kvatinsky. Dark memory and accelerator-rich system optimization in the dark silicon era. *IEEE Design & Test*, 34(2):39–50, 2016.

[114] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for nic-accelerated network applications. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[115] Salvatore Pontarelli et al. FlowBlaze: Stateful packet processing in hardware. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[116] Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, 2014.

[117] Andrew Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proc. ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2014.

[118] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark Horowitz. Convolution engine: Balancing efficiency and flexibility in specialized computing. *Communications of the ACM*, 58(4):85–93, 2015.

[119] S. Rajendran, W. Meert, V. Lenders, and S. Pollin. SAIFE: Unsupervised wireless spectrum anomaly detection with interpretable features. In *Proc. IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, 2018.

[120] Sreeraj Rajendran, Roberto Calvo-Palomino, Markus Fuchs, Bertold Van den Bergh, Héctor Cordobés, Domenico Giustiniano, Sofie Pollin, and Vincent Lenders. Electrosense: Open and big spectrum data. *IEEE Communications Magazine*, January 2018.

[121] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278. IEEE, 2016.

[122] Jack Sampson, Ganesh Venkatesh, Nathan Goulding-Hotta, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor. Efficient Complex Operators for Irregular Codes. In *HPCA*, 2011.

[123] Richard Sampson, Ming Yang, Siyuan Wei, Chaitali Chakrabarti, and Thomas F Wenisch. Sonic millip3de: A massively parallel 3d-stacked accelerator for 3d ultrasound. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 318–329. IEEE, 2013.

[124] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44(3):14–26, 2016.

[125] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 97–108. IEEE, 2014.

[126] David E Shaw, Martin M Deneroff, Ron O Dror, Jeffrey S Kuskin, Richard H Larson, John K Salmon, Cliff Young, Brannon Batson, Kevin J Bowers, Jack C Chao, et al. Anton, a special-purpose machine for molecular dynamics simulation. *Communications of the ACM*, 51(7):91–97, 2008.

[127] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 367–379, 2019.

[128] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 367–379, 2019.

[129] Singularity. *Average Transistor Price*, 2002. url-http://www.singularity.com/charts/page59.html.

[130] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 44–57, 2016.

[131] Matt Skach, Manish Arora, Chang-Hong Hsu, Qi Li, Dean Tullsen, Lingjia Tang, and Jason Mars. Thermal time shifting: Leveraging phase change materials to reduce cooling costs in warehouse-scale computers. In *ISCA*, 2015.

[132] Alex Solomatnikov, Amin Firoozshahian, Wajahat Qadeer, Ofer Shacham, Kyle Kelley, Zain Asgar, Megan Wachs, Rehan Hameed, and Mark Horowitz. Chip multi-processor generator. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 262–263, New York, NY, USA, 2007. ACM.

[133] N Sornin, M Luis, T Eirich, T Kramp, and O Hersent. Lora specification 1.0. *Lora Alliance Standard Specification. Available online: https://lora-alliance. org/sites/default/files/2018-04/lorawantm_specification_-v1*, 1, 2015.

[134] Stephen Weston. FPGA Accelerators at JP Morgan Chase, 2011. Stanford Computer Systems Colloquium, https://www.youtube.com/watch?v=9NqX1ETADn0.

[135] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and efficient {NIC} packet scheduling. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 33–46, 2019.

[136] Brent Stephens, Aditya Akella, and Michael M Swift. Your programmable nic should be a programmable switch. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 36–42, 2018.

[137] Kun Tan, He Liu, Jiansong Zhang, Yongguang Zhang, Ji Fang, and Geoffrey M. Voelker. Sora: High-performance software radio using general-purpose multi-core processors. *Commun. ACM*, 54(1):99–107, January 2011.

[138] Prateek Tandon, Jichuan Chang, Ronald G. Dreslinski, Vahed Qazvinian, Parthasarathy Ranganathan, and Thomas F. Wenisch. Hardware acceleration for similarity measurement in natural language processing. In *ISLPED*, 2013.

[139] M. B. Taylor. Basejump stl: Systemverilog needs a standard template library for hardware design. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[140] Michael Taylor. Bitcoin and the age of bespoke silicon. In *CASES*, 2013.

[141] Michael Taylor. A landscape of the new dark silicon design regime. *Micro, IEEE*, Sept-Oct. 2013.

[142] Michael B. Taylor. *BaseJump ASIC MotherBoards*. http://bjump.org/#motherboards.

[143] Michael B. Taylor. *BSG DoubleTrouble*. http://bjump.org/basejump_motherboards_socket_352.html.

[144] Michael B. Taylor. Is dark silicon useful? harnessing the four horesemen of the coming dark silicon apocalypse. In *DAC*, 2012.

[145] David L. Tennenhouse and Vanu G. Bose. SpectrumWare - A software-oriented approach to wireless signal processing. In *Proc. ACM Conference on Mobile Computing and Networking (MobiCom)*, 1995.

[146] Stephen M Steve Trimberger. Three ages of fpgas: A retrospective on the first thirty years of fpga technology: This paper reflects on how moore's law has driven the design of fpgas through three epochs: the age of invention, the age of expansion, and the age of accumulation. *IEEE Solid-State Circuits Magazine*, 10(2):16–29, 2018.

[147] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. *ACM Sigplan Notices*, 45(3):205–218, 2010.

[148] Venkatesh et al. Qscores: Configurable co-processors to trade dark silicon for energy efficiency in a scalable manner. In *MICRO*, 2011.

[149] M. Wachs, O. Shacham, Z. Asgar, A. Firoozshahian, S. Richardson, and M. Horowitz. Bringing up a chip on the cheap. *IEEE Design Test of Computers*, 29(6):57–65, Dec 2012.

[150] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4FPGA: A rapid prototyping framework for P4. In *Proc. ACM Symposium on SDN Research (SOSR)*, 2017.

[151] Nicholas Weaver, Vern Paxson, and Jose M Gonzalez. The shunt: An fpga-based accelerator for network intrusion prevention. In *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2007.

[152] P. Welch. The use of fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. *IEEE Transactions on Audio and Electroacoustics*, 15(2):70–73, jun 1967.

[153] J. L. Wong, F. Kourshanfar, and M. Potkonjak. Flexible asic: shared masking for multiple media processors. In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 909–914, June 2005.

[154] Kun-Cheng Wu and Yu-Wen Tsai. Structured asic, evolution or revolution? In *Proceedings of the 2004 International Symposium on Physical Design*, ISPD '04, pages 103–106, New York, NY, USA, 2004. ACM.

[155] Lisa Wu, Andrea Lottarini, Timothy Paine, Martha Kim, and Kenneth Ross. Q100: The architecture and design of a database processing unit. In *ASPLOS*, 2014.

[156] Dirk Wubben, Peter Rost, Jens Steven Bartelt, Massinissa Lalam, Valentin Savin, Matteo Gorgoglione, Armin Dekorsy, and Gerhard Fettweis. Benefits and impact of cloud computing on 5g signal processing: Flexible centralization through cloud-ran. *IEEE signal processing magazine*, 31(6):35–44, 2014.

[157] Shaolin Xie and Michael Bedford Taylor. The basejump manycore accelerator network, 2018.

[158] Xilinx. *Xilinx Announces the World's Largest FPGA Featuring 9 Million System Logic Cells*, 2019. urlhttps://www.xilinx.com/news/press/2019/xilinx-announces-the-world-s-largest-fpga-featuring-9-million-system-logic-cells.html.

[159] Xilinx. *Xilinx Versal AI Core Series*, 2019. urlhttps://www.xilinx.com/products/silicon-devices/acap/versal-ai-core.html.

[160] Ning-Yi Xu, Xiong-Fei Cai, Rui Gao, Lei Zhang, and Feng-Hsiung Hsu. Fpga acceleration of rankboost in web search engines. *TRETS*, 1(4), January 2009.

[161] Reza Yazdani, Albert Segura, Jose-Maria Arnau, and Antonio Gonzalez. An ultra low-power hardware accelerator for automatic speech recognition. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[162] Yeh et al. A low operating power finfet transistor module featuring scaled gate stack and strain engineering for 32/28nm soc technology. In *IEDM*, Dec 2010.

[163] B. Zahiri. Structured asics: opportunities and challenges. In *Computer Design, 2003. Proceedings. 21st International Conference on*, pages 404–409, Oct 2003.

[164] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[165] ZigBee Alliance. *ZIGBEE SPECIFICATION*, 9 2012. Rev. 20.

[166] N. Zilberman, Y. Audzevich, G. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 gbps as research commodity. *IEEE Micro*, September 2014.