Rapid SoC Design: On Architectures, Methodologies and Frameworks

by

Tutu Ajayi

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical and Computer Engineering)
in the University of Michigan
2021

Doctoral Committee:

Assistant Professor Ronald Dreslinski Jr, Chair
Professor David Blaauw
Professor Chad Jenkins
Professor David Wentzloff

Tutu Ajayi

ajayi@umich.edu

ORCID iD: 0000-0001-7960-9828

© Tutu Ajayi 2021

ACKNOWLEDGMENTS

Foremost, I wish to recognize Ronald Dreslinksi Jr., my thesis advisor, who took a chance on me as an incoming graduate student. He ushered me into academic research and made my transition from industry seamless. He taught me to think differently, perform research, and exposed me to a broad array of research topics. His advice, research, and network were important in my development as a student and researcher.

I would like to thank the faculty members at the University of Michigan that have provided additional advice and guidance. Several collaborations with David Wentzloff, David Blaauw, and Dennis Sylvester enriched my research experience. I would also like to thank Mark Brehob and Chad Jenkins for their mentorship and guidance as I navigated my way through the program.

I am thankful for the opportunity to work with the good people at Arm; Balaji Venu and Liam Dillon provided me with a glimpse into industry research. I am also grateful to my collaborators on the OpenROAD project for exposing me to open-source world. Additional thanks to my many collaborators at Arizona State University, Cornell University, University of California - San Diego, and the University of Washington.

I am also thankful to the many colleagues, friends, and students at the University of Michigan, without whom I could not have completed this work. Also thankful for my squash buddies who challenged me to keep my mind clear and body active. I also wish to recognize my friends still back at Austin, Texas.

I would like to thank my family for their continued support and for helping me out through all the years. I am forever grateful to my parents for their unconditional love and for teaching me the importance of education; and to my brothers for the backing and encouragement.

Finally, I dedicate this thesis to my wife, Erin Nicole Ajayi, for the patience, support, and love she has given me all these years.

TABLE OF CONTENTS

Acknowledgments . . . . **ii**

List of Figures . . . . **vi**

List of Tables . . . . **viii**

Abstract . . . . **ix**

Chapter

1 Introduction **1**

1.1 Contributions 3

2 Rapid Architectures: Celerity **5**

2.1 Introduction 6
2.2 The Celerity Architecture 8
2.2.1 Partitioned Global Address Space 9
2.3 The General-purpose Tier 9
2.4 The Massively Parallel Tier 10
2.4.1 NoC Design 10
2.4.2 Remote Stores 11
2.4.3 LBR 11
2.4.4 Token Queue 11
2.4.5 Programming Models 12
2.5 The Specialization Tier 12
2.5.1 Choosing the Neural Network 12
2.5.2 Performance Target 13
2.5.3 Creating and Optimizing the Specialization Tier 14
2.5.4 Establishing the Functionality of the Specialization Tier 14
2.5.5 Designing the Specialization Tier 14
2.5.6 Combining the Massively Parallel and Specialization Tiers 15
2.5.7 The Benefits of HLS 15
2.6 Performance Analysis Of The Specialization Tier 15
2.7 New Directions For Fast Hardware Design 16
2.8 Achieving Celerity With Celerity: Fast Design Methodologies For Fast Chips 17
2.8.1 Reuse 17
2.8.2 Modularization 18

2.8.3 Automation . . . 19

3 Rapid Architectures: FASoC . . . 20

3.1 Introduction . . . 20
3.2 Framework Architecture . . . 21
3.2.1 Process Setup and Modeling . . . 22
3.2.2 SoC Generator . . . 23
3.3 Analog Generator Architecture . . . 24
3.3.1 PLL . . . 25
3.3.2 LDO . . . 26
3.3.3 Temperature Sensor . . . 26
3.3.4 SRAM . . . 27
3.4 Evaluation . . . 28
3.4.1 Analog Generation Results . . . 28
3.4.2 Prototype Chip Results . . . 30
3.5 Conclusion . . . 34

4 Rapid Methodologies: Cadre Flow . . . 35

4.1 Introduction . . . 35
4.2 Design . . . 37
4.3 Goals . . . 38
4.4 Evaluation . . . 39
4.5 Conclusion . . . 40

5 Rapid Methodologies: OpenROAD . . . 41

5.1 Introduction . . . 41
5.1.1 IDEA and the OpenROAD Project . . . 42
5.1.2 A New Paradigm . . . 44
5.2 Layout Tool Chain . . . 44
5.2.1 Logic Synthesis . . . 44
5.2.2 Floorplan and PDN . . . 46
5.2.3 Placement . . . 46
5.2.4 Clock Tree Synthesis . . . 47
5.2.5 Routing . . . 48
5.3 Other Elements . . . 48
5.3.1 Static Timing Analysis . . . 49
5.3.2 Parasitic Extraction . . . 49
5.3.3 Power Integrity . . . 50
5.3.4 Cloud Infrastructure . . . 50
5.3.5 METRICS 2.0 . . . 51
5.3.6 Early SoC Planning . . . 51
5.3.7 Integration and Testing . . . 52
5.4 Looking Forward . . . 53
5.5 Conclusion . . . 54

6 Rapid Frameworks: ACAI . . . 55

6.1 Introduction 56
6.2 Background 58
6.3 ACAI Architecture 60
6.3.1 Base Hardware 61
6.3.2 ACAI Operation 62
6.3.3 Advanced Scheduling 64
6.4 Methodology 66
6.4.1 Hardware Setup 66
6.4.2 Software Setup 69
6.4.3 Kernels and Applications 71
6.5 Evaluation 73
6.5.1 Latency and Bandwidth 73
6.5.2 Single Kernel Analysis 75
6.5.3 Application Analysis 76
6.6 Future work 76
6.7 Related Work 77
6.8 Conclusion 78

7 Conclusions 79

Bibliography 80

LIST OF FIGURES

2.1 Celerity block diagram 7
2.2 Detailed Celerity images 8
2.3 BNN accelerator 13
2.4 BaseJump open-source hardware components 18

3.1 FASoC Framework Overview 21
3.2 Aux-cell and model file generation flow 22
3.3 Schematic for aux-cells used across PLL, LDO and temperature sensor generators 23
3.4 Analog generator flow 24
3.5 DCO architecture indicating the aux-cells and designs parameters 26
3.6 LDO architecture indicating the aux-cells and design parameters 27
3.7 Temperature sensor architecture indicating the aux-cells 27
3.8 SRAM architecture showing macros and bank strategy 28
3.9 Generated PLL designs for eight different input specifications 29
3.10 $I_{load,max}$ vs. array size, for multiple LDO designs generated 29
3.11 Normalized energy and delay plots for generated SRAM blocks 30
3.12 Power and Error results against temperature for various temperature sensor designs (each fitted plot represents a unique design) 31
3.13 Simplified block diagram for the 65nm prototype SoC 33
3.14 Annotated die photo for the 65nm prototype SoC 33
3.15 Measured and simulated performance and power results of SRAM across VDD 34

4.1 Cadre flow module abstractions 37
4.2 Timeline of SoCs taped out using Cadre Flow 39

5.1 Design technology crisis. 42
5.2 Design complexity. 43
5.3 The OpenROAD flow. 45
5.4 16nm RISC-V based design block design 47
5.5 Comparison between OpenSTA and a leading signoff timer 49
5.6 Example PDN templates and validation of the regression mode 51
5.7 Overall METRICS 2.0 system architecture. 52
5.8 Illustration of Unified Planning Tool. 53

6.1 Example SoC platform showing ACAI hardware interface with the integration of (a) an on-chip HA and (b) an off-chip HA 57

6.2 ACAI hardware interface design 61
6.3 An example architectural state of the ACAI framework 63
6.4 Example application using framework 65
6.5 ACAI$_{opt}$ emulation for optimized for bandwidth 67
6.6 Hardware prototype platform showing HA integration using ACAI and other interface options 68
6.7 Pseudo-code for sequential bulk-data copy kernel 71
6.8 Pseudo-code for random bulk-data copy kernel 72
6.9 Pseudo-code for linked-list data traversal kernel 72
6.10 Memory latency distribution for random memory requests 73
6.11 Synthetic benchmark performance normalized to Shared-NC 74
6.12 Machsuite performance results across various kernels 75
6.13 Rosetta Benchmark Results across various applications 76

LIST OF TABLES

2.1 Performance comparison of optimized BNN implementations 16

3.1 PLL Simulation vs Measurement Results 32
3.2 LDO Simulation vs Measurement Results @ 200MHz control clock 32

4.1 List of supported platforms in the Cadre Flow 40

6.1 Prototype system configuration and details 70
6.2 Machsuite kernel details 73
6.3 Rosetta application details 74

ABSTRACT

Modern applications like machine learning, autonomous vehicles, and 5G networking require an order of magnitude boost in processing capability. For several decades, chip designers have relied on Moore's Law - the doubling of transistor count every two years - to deliver improved performance, higher energy efficiency, and an increase in transistor density. With the end of Dennard's scaling and a slowdown in Moore's Law, system architects have developed several techniques to deliver on the traditional performance and power improvements we have come to expect. More recently, chip designers have turned towards heterogeneous systems comprised of more specialized processing units to buttress the traditional processing units. These specialized units improve the overall performance, power, and area (PPA) metrics across a wide variety of workloads and applications. While the GPU serves as a classical example, accelerators for machine learning, approximate computing, graph processing, and database applications have become commonplace. This has led to an exponential growth in the variety (and count) of these compute units found in modern embedded and high-performance computing platforms.

The various techniques adopted to combat the slowing of Moore's Law directly translates to an increase in complexity for modern system-on-chips (SoCs). This increase in complexity in turn leads to an increase in design effort and validation time for hardware and the accompanying software stacks. This is further aggravated by fabrication challenges (photo-lithography, tooling, and yield) faced at advanced technology nodes (below 28nm). The inherent complexity in modern SoCs translates into increased costs and time-to-market delays. This holds true across the spectrum, from mobile/handheld processors to

high-performance data-center appliances.

This dissertation presents several techniques to address the challenges of rapidly birthing complex SoCs. The first part of this dissertation focuses on foundations and architectures that aid in rapid SoC design. It presents a variety of architectural techniques that were developed and leveraged to rapidly construct complex SoCs at advanced process nodes. The next part of the dissertation focuses on the gap between a completed design model (in RTL form) and its physical manifestation (a GDS file that will be sent to the foundry for fabrication). It presents methodologies and a workflow for rapidly walking a design through to completion at arbitrary technology nodes. It also presents progress on creating tools and a flow that is entirely dependent on open-source tools. The last part presents a framework that not only speeds up the integration of a hardware accelerator into an SoC ecosystem, but emphasizes software adoption and usability.

CHAPTER 1

Introduction

Modern applications like machine learning, autonomous vehicles, and 5G networking need an order of magnitude boost in processing capability. For several decades, chip designers have relied on Moore's Law - the doubling of transistor count every two years - to deliver improved performance, higher energy efficiency, and an increase in transistor density. With the end of Dennard's scaling and a slowdown in Moore's Law, system architects have developed several techniques to deliver on the traditional performance and power improvements we have come to expect. A prime example is the shift to multicore processors that has become pervasive in present-day computers (and even mobile phones). More recently, chip designers have turned towards heterogeneous systems comprised of more specialized processing units to buttress the traditional processing units. These specialized units improve the overall performance, power, and area (PPA) metrics across a wide variety of workloads and applications. While the GPU serves as a classical example, accelerators for machine learning, approximate computing, graph processing, and database applications have become commonplace. This has lead to an exponential growth in the variety (and count) of these compute units found in modern embedded and high-performance computing platforms. In 2010, Apple's A4 mobile SoC was estimated to have 9 accelerator blocks and by 2014, the A8 had pushed that number to 29 [1]. These counts are expected to be dwarfed by the A14 Bionic SoC announced in 2020.

The various techniques adopted to combat the end of Moore's Law directly translates to an increase in complexity for modern system-on-chips (SoCs). This increase in complexity in turn leads to an increase in design effort and validation time for hardware and the accompanying software stacks. This is further aggravated by fabrication challenges (photo-lithography, tooling, and yield) faced at advanced technology nodes (below 28nm). The inherent complexity in modern SoCs translates into increased costs and time-to-market delays. This holds true across the spectrum, from mobile/handheld processors to high-performance data-center appliances. The skyrocketing costs of modern SoC projects can directly be attributed to design complexity, and it varies widely depending on the nature

of the chip. Extremely complex SoCs can be an order of magnitude more costly when compared to a traditional SoC optimized for high-performance. While the design and fabrication costs of complex SoCs can be ameliorated by high-volume and increased life-time, the total cost remains dominated by non-recurring engineering costs (NREs). These one-time overhead costs (often applicable to prototype runs) can be prohibitively expensive for small companies and academic researchers.

This dissertation presents several techniques to address the challenges of rapidly birthing complex SoCs. It proposes a series of architectures, methodologies and frameworks that aid in the swift construction of complex SoC architectures and approaches to rapidly walk it through the realization process.

Despite the availability of a range of silicon IP cores from vendors such as Arm and Synopsys, the designer productivity for such chips has remained flat. Other companies have devised ASIC platforms with qualified function blocks and interfaces to cut down on design time and verification, however, these are often too rigid for the rapidly changing landscape and only address a small percentage of the challenge. Complex chips rely on expert designers at the different phases of the design to fill this gap and achieve the necessary performance and cost goals. The process essentially remains an unintuitive black-art and filled with excessive trial-and-error. Chapter 2 presents a variety of architectural techniques that were developed and leveraged to rapidly construct *Celerity* - a complex SoC design targeted at a 16nm process. This addresses the first step of the challenge - quickly creating a complex SoC architecture that will seed the rest of the chip realization process. It fleshes out design reuse, modularization, and high-level synthesis (HLS) techniques that were used to rapidly develop the celerity chip.

Chapter 3 focuses on SoCs that require additional analog components. An analog-mixed-signal SoC (AMS-SoC) is a combination of analog circuits, digital circuits and/or mixed-signal circuits on the same chip. Example AMS-SoCs include smart sensors, wireless/RF devices, and voltage/power controllers. The increased use of cellular phones and other portable devices has propelled growth in this category of SoCs. The design of high-performance analog components is often labor-intensive and involves some hand-crafting or rigid block generation. Furthermore, the inner workings of these components have historically been opaque to digital designers responsible for the overall SoC architecture. This co-habitation often presents challenges for the overall system since the blocks often have different power and clocking needs. It often leads to a stringent interface between the blocks that typically affects overall performance and is often a source of chip re-spins due to poorly matched circuit models. The proposed Fully Autonomous SoC (FASoC) framework is process-agnostic approach that leverages a catalog of synthesizable analog blocks

to generate complete SoC designs ready for fabrication.

Chapter 4 focuses on the gap between a completed design model (in RTL form) and its physical manifestation (a GDS file that will be sent to the foundry for fabrication). These challenges are as much SoC design complexity as they are due to methodologies binding CAD tools and thte workflow from an architectural design to a finished product. The recent proliferation of commercial tools, from formal verification, logic synthesis, timing optimization to physical design have made it possible to deploy mature algorithms to scale design blocks before being composed into systems. Using these tools usually requires calibrating and setting a large number of parameters. Further downstream towards the manufacturing process, an even larger number of parametric adjustments must be made to achieve at-performance/at-yield manufacturing results. This chapter also presents Cadre Flow, a robust flow methodology that targets addresses the challenge of pushing a completed design through the CAD tools on an arbitrary technology node.

To break through the shackles forged from NDAs and IP restrictions (imposed by EDA tool vendors and IP providers alike), Chapter 5 presents work that reboots this effort using a completely open-source approach.

Chapter 6 progress further through the chip's life cycle with ACAI, a framework that not only speeds up the integration of a hardware accelerator, but emphasizes software adoption and usability. It provides the hardware accelerator with a shared view of system memory, shared virtual addressing, and data caching with full hardware coherency. ACAI simplifies the software programming experience, reduces integration effort, and orchestrates job scheduling.

1.1 Contributions

In this thesis, I present five projects focused on rapid SOC design. Many of the projects are large undertakings with significant contributions by several individuals at different institutions. For clarity, my contributions to the individual projects are as follows:

1. **Celerity**: On this project, my contributions primarily revolved around physical design. This includes developing the physical design flow and mythologies. I provided feedback on logic design and architecture, performed block integration, and handled all chip sign-off tasks.

2. **FASoC**: On this project, I was primarily involved in architecting the infrastructural pieces for the framework. This includes the general scaffolding and API/interface definitions. I also performed physical design and lead the chip tape-out.

3. **Cadre Flow**: I was the primary developer of the flow.
4. **OpenROAD**: I was primarily responsible for creating test cases, steering the development of the physical design tools, creating a flow that stitches all off the tools coherently, and developing any intermediate tools/workarounds.
5. This was also ongoing work as the tools matured
6. **ACAI**: This work was performed in collaboration with the research team at Arm while I interned there.

CHAPTER 2

Rapid Architectures: Celerity

The recent trend towards accelerator-centric architectures has renewed the need for demonstrating new research ideas in prototype systems with custom chips. Unfortunately, building such research prototypes is tremendously challenging. This chapter presents an open-source software and hardware ecosystem partly addresses this challenge by reducing design, implementation, and verification effort. It briefly describes the Celerity system-on-chip (SoC), a 5×5 mm, 350M-transistor chip in TSMC 16nm, which uses a tiered parallel architecture to improve both the performance and energy efficiency of embedded applications.

The Celerity SoC includes five RV64G cores, a 496-core RV32IM tiled manycore processor, and a complex BNN (binarized neural network) accelerator implemented as a Rocket custom co-processor (RoCC). It was developed in nine months, from PDK access to tapeout. The project is a part of the DARPA CRAFT (Circuit Realization At Faster Timelines) program which seeks to develop new methodologies for rapid chip development. At the time of fabrication in 2018, Celerity was the most complex SoC developed to date in academia in terms of transistor count (385 Million).

The work presented in this chapter was completed in collaboration with researchers from multiple universities including the University of Michigan, the University of California - San Diego, Cornell University, and with support from Arm.

Other major accomplishments from this work include:

- The coordination of researchers and resources spread across the four universities
- The development of an agile workflow in an advanced 16nm FinFET node
- Satisfying the CRAFT program constraints with only $1.3 million in NRE costs

Results from this chapter first appeared at The Symposium on High Performance Chips (Hot Chips) [2]. It has also appeared in First Workshop on Computer Architecture Research

with RISC-V [3], IEEE Micro Journal [4], 2019 Symposium on VLSI Circuits [5], and IEEE Solid-State Circuits Letters [6].

2.1 Introduction

Rapidly emerging workloads require rapidly developed chips. The Celerity 16nm open-source SoC was implemented in nine months using an architectural trifecta to minimize development time: a general-purpose tier comprised of open-source Linux-capable RISC-V cores, a massively parallel tier comprised of a RISC-V tiled manycore array that can be scaled to arbitrary sizes, and a specialization tier that uses high-level synthesis (HLS) to create an algorithmic neural-network accelerator. These tiers are tied together with an efficient heterogeneous remote store programming model on top of a flexible partial global address space memory system.

Emerging workloads have extremely strict energy-efficiency and performance requirements that are difficult to attain. Increasingly, we see that specialized hardware accelerators are necessary to attain these requirements. But accelerator development is time-intensive, and accelerator behavior cannot be easily modified to adapt to changing workload properties. These factors motivate new architectures that can be rapidly constructed to address new application domains, while still leveraging specialized hardware and offering high performance and energy efficiency even as applications evolve post-tapeout.

We propose a chip architecture called Celerity, meaning "swiftness of movement," that embodies an architectural design pattern called the tiered accelerator fabric (TAF). TAF minimizes time-to-market and allows the chip to maintain high performance and energy efficiency on evolving workloads.

A TAF has three key architectural tiers:

- The general-purpose tier is a set of OS-capable cores for executing complex codes like networking, control, and decision making.
- The specialization tier is made of highly specialized algorithmic accelerators to target specific computations with extreme energy-efficiency and performance requirements.
- The massively parallel tier is made of scalable programmable arrays of small, tightly coupled cores that attain high energy efficiency and flexibility for evolving workloads.

In response to our target application domain—autonomous vision systems—the Celerity SoC implements the general-purpose, specialization, and massively parallel tiers using

five Linux-capable RISC-V cores, a binarized neural network (BNN) accelerator generated with HLS, and a “GPU-killer” 496-core RISC-V manycore array, respectively. Figure 1 shows a block diagram of Celerity highlighting the general-purpose tier in green, the specialization tier in blue, and the massively parallel tier in red. To bind these components together, we support a heterogeneous remote store programming model that allows core and accelerators to write to each other’s memories through a partitioned global address space. Layered upon this model are two novel synchronization mechanisms: load-reserved, load-on-broken-reservation (LR-LBR), which extends load-reserved store conditional for efficient producer-consumer synchronization; and the token queue, which uses LR-LBR to achieve efficient producer-consumer transfer of resource ownership. This architecture enabled us to design and implement Celerity in only nine months through open-source and agile hardware techniques.

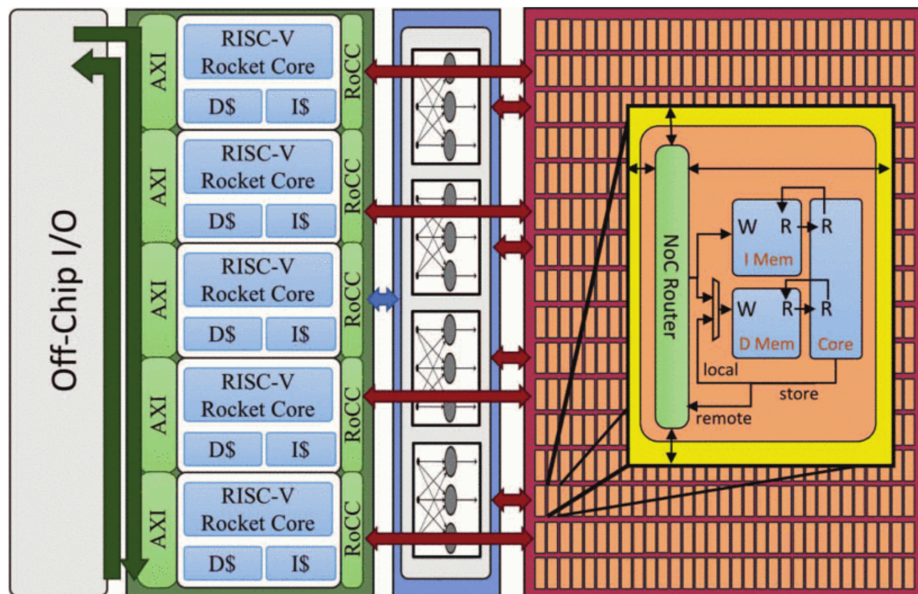


Figure 2.1: Celerity block diagram. The general-purpose tier (shown in green) has a five-core Rocket core complex, the specialization tier (shown in blue) has a BNN accelerator, and the massively parallel tier (shown in red) has a 496-core tiled manycore array

Celerity is an open-source 5x5-mm tiered accelerator fabric SoC taped out in Taiwan Semiconductor Manufacturing Company (TSMC) 16nm Fin field-effect transistor (FinFET) Compact (FFC) with 385 million transistors. In addition to the previously mentioned 501 RISC-V cores, it features an ultra-low-power 10-core RISC-V manycore array powered by an on-chip DC/DC low-dropout (LDO) regulator [2]. The 10-core array shares the same code as the larger 496-core array. The architecture has separate clock domains for I/O (400 MHz), the manycore (1.05 GHz), and the rest of the chip (625 MHz). Figure 2(a)

shows the SoC's floorplan image from our CAD tools. Figure 2(b) shows the layout of Celerity. Finally, Figure 2(c) shows a photomicrograph.

The design's entire source base is available at http://opencelerity.org. See the sidebar, "Achieving Celerity with Celerity," for the methodologies used to design and tapeout the Celerity chip in less than nine months.

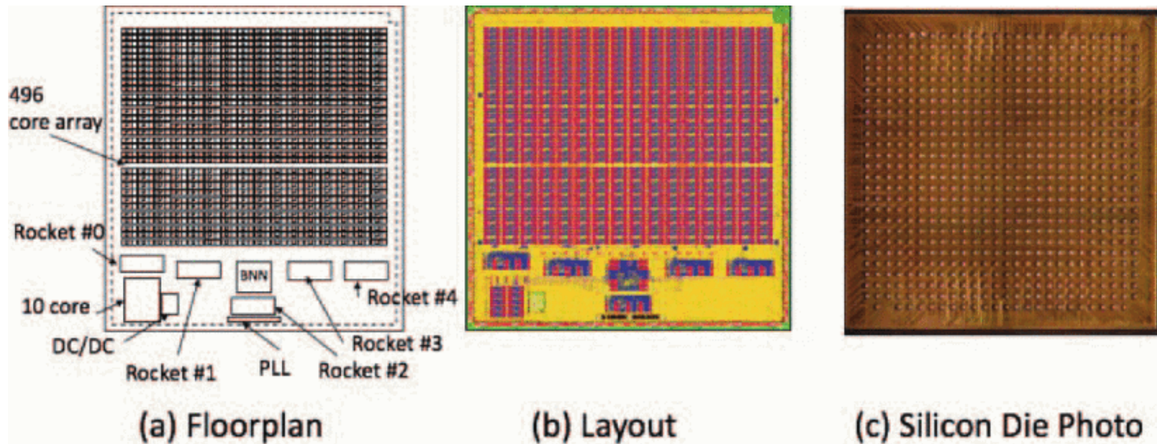


Figure 2.2: Detailed Celerity images. The floorplan (a) shows the relative sizes and positions of the various blocks in the SoC. The layout (b) shows the physical chip attributes where the red area represents SRAM, the blue area represents logic, and the yellow area represents interconnection between blocks. The silicon die photo (c) shows the real chip taken from a photomicrograph.

2.2 The Celerity Architecture

When addressing emerging application domains with a tiered accelerator fabric, a number of key decisions must be made. The specialization tier is among the most important, because it is the most integral in determining the chip's super-capabilities, and it requires the most effort to design. The choice of general-purpose tier will be determined by feature set (for example, security, debugging features, or raw irregular computation) but also by the availability and expense of processor IP cores. ARM offers many variants, but low-non-recurring-engineering (NRE) open-source versions of RISC-V are becoming available, like the Berkeley Rocket processor core used in our design. The massively parallel tier could be comprised of ARM or Advanced Micro Devices (AMD) GPU IPs. Alternatively, our open-source tiled manycore architecture, BaseJump Manycore, is free and allows for fast and flexible scaling from one to 1 million+ cores, at an area cost of 1 mm2 per 42 cores in 16 nm. We explore each tier in the following sections, but first discuss how these components

are tied together.

2.2.1 Partitioned Global Address Space

Communication among accelerators and cores in the three tiers is accomplished through a partitioned global address space over a unified mesh network-on-chip (NoC). When a remote store is performed, a wide single-word packet is injected into the NoC, which contains x,y coordinates of the destination core, the local word address at that core, 32 bits of data to store, and a byte mask. When the message arrives at the destination, the address is translated and the store is performed. Ordering of messages sent from one node to another is maintained. The parameterized NoC in Celerity was configured for 512 coordinates (x = 0..15, y = 0..31) and 22-bit addresses. The manycore's cores map one-to-one to all of these addresses except y = 31, which demarcates the south edge of the manycore. The remaining 16 positions on the south edge are used for four parallel connections to the BNN and four connections to the Linux-capable Rocket cores.

While remote loads, such as those found in the Adapteva parallel architecture [7], are easy to add and could arguably make the system more programmable, they have high round-trip latency costs and lead users astray by offering a high-convenience, low-performance mechanism. Remote stores do not incur such a latency penalty because they are pipelined and can therefore be issued once per cycle. When a remote store is performed, a local credit counter will be decremented at the sender. When the store is successful at the remote node, a store credit is placed on the store network that is routed back to the original tile on a separate 9-bit physical network, incrementing the counter. A RISC-V fence instruction on either manycore or Rocket core is used to detect whether any outstanding remote stores exist, allowing a core to pause for memory traffic to finish during a barrier.

2.3 The General-purpose Tier

For our SoC to support complex software stacks, exception handling, and memory management, we instantiated five Berkeley RISC-V Rocket cores running the RV64G ISA. The Rocket core is an open-source [8], five-stage, in-order, single-issue processor with a 64-bit pipelined FPU and size-configurable non-blocking caches. Each Rocket core can run an independent Linux image. This gives us the flexibility to run SPEC-style applications and network stacks like TCP/IP. Four Rocket cores connect directly to the massively parallel tier using four parallel remote store links on the global mesh NoC. One Rocket core connects directly to the specialization tier through a dedicated Rocket custom coprocessor

(RoCC) interface. These connections are made using the Berkeley RoCC interface. L1 data and instruction caches are configured at 16 KBs each. When remote stores are done to the Rocket cores, they go directly into the four Rocket cores' caches, potentially causing cache misses to DRAM. Remote store addresses are translated using a segment address register that maps the 22-bit address space into the Rocket's 40-bit address space. Rocket cores issue remote stores through a single RoCC instruction and can, for example, do remote stores to other Rocket cores, to any manycore, or to any of the BNN input links. Remote stores to manycore tiles are used to write instruction and data memories, as well as to set configuration registers, such as freeze registers and arbitration policies for the local data memory.

2.4 The Massively Parallel Tier

To achieve massive amounts of programmable energy-efficient parallel computation, we wanted an architecture with a high density of physical threads per area. Therefore, we implemented a 496-core tiled manycore array [9] that interconnects low-power RISC-V Vanilla-5 cores using a mesh interconnection network. Each tile contains a simple router and a Vanilla-5 core. Our in-house-developed Vanilla-5 cores are five-stage, in-order, single-issue processors with 4-KB instruction and data memories that use the RV32IM ISA. The manycore uses a strict remote store programming model [10], giving us a highly programmable array to maintain high performance as workloads evolve post-tapeout. A key contribution of our work is to extend the remote store programming model to incorporate heterogeneous processor types and to support fast producer-consumer synchronization.

2.4.1 NoC Design

The manycore's mesh NoC design, which facilitates the remote store fabric that ties the chip together, targets extreme area efficiency using only a single physical network for data transfer, no virtual channels, single-word/single-flit packets, deterministic x,y dimension-ordered routing, and two-element router input buffers. Head-of-line blocking and deadlock are eliminated because remote stores can always be written to a core's local memory, removing the word from the network. Connections between neighboring tiles are 80-bit wide full duplex, running at 1 GHz, allowing address, command, and data information to be routed in a single wide word, and each hop takes one cycle. To generate packets that go off the array's south side, to the specialized and general-purpose tiers, a NoC client performs a store to a memory address whose x,y coordinate is beyond the coordinates of manycore.

Both local and remote stores use the same standard store word, half-word, and byte instructions from the ISA.

2.4.2 Remote Stores

Each time a store is about to be performed, the high bit of the address determines if the store address is local (0) or remote (1). The local address space uses the remaining 31 bits to determine the memory address. The remote address space uses the next 9 bits as a destination coordinate (x= 0..15, y = 0..31) of the target core on the NoC. The remaining 22 bits are translated at the destination into a local address, and the store is performed.

2.4.3 LBR

The manycore features an extension to the LR store-conditional (LR-SC) atomic instructions called LR-LBR. LR operates much like in LR-SC by performing a load and then adding the target address to a reservation register, which is then cleared if an external core writes to that address. LBR is a new instruction that places the core's pipeline in a low-power mode until another core remote stores to that address and breaks the reservation, at which point the core will wake up and perform a load on the target address. Typically, user code will load a memory location's value with LR, branch away if it is satisfied with the value (a ready flag is set, or a FIFO pointer has sufficiently advanced), and otherwise fall through to a LBR to wait for it to change, so it can be rechecked.

2.4.4 Token Queue

Our design shows that tight producer-consumer synchronization can be layered on top of remote store programming. By using the LR-LBR instruction extension, we implemented the token queue, a software construct used to asynchronously transfer control of buffer address between producer and consumer tiles. The consumer will allocate a circular buffer to which tokens can be enqueued and dequeued. A token can be a simple data value, a pointer to a memory buffer, or identifiers for more abstract resources. Producer and consumer can consume different quantities of tokens at each step. By enqueuing a set of tokens, the producer is transferring read/write ownership of those resources to the consumer. By dequeuing a set of tokens, the consumer is transferring write ownership of the resource back to the producer. The producer and consumer each have local copies of head and tail pointers to the circular buffer, but only the producer will modify the head pointers, and only the consumer will modify the tail pointers. The remote versions of the pointers will be up-

dated after the local versions, similar to a clock-domain-crossing FIFO. The producer tile confirms there is enough space in the token queue to enqueue a particular group of tokens, using LR-LBR to wait in low-power mode for remote updates to the local tail pointers if there is not enough space in the queue. Then, it will send the corresponding data through remote stores. After that is done, the producer will update the head pointers through local and remote stores. The consumer confirms that it has enough tokens in the token queue to proceed, using the LR-LBR instructions to wait in low-power mode until the head pointer is updated by the producer, and checking if enough tokens have been enqueued. When there is enough, the consumer will wake up and start accessing the data represented by the new tokens in the buffer. When done, the consumer will dequeue the tokens by updating the tail pointers and proceeding back to consuming the next set of tokens.

2.4.5 Programming Models

The token queue and remote store programming models are particularly well suited for programming with the StreamIt [11] programming model. We are also investigating libraries that will enable CUDA-style applications to be ported more easily, but emphasizing an execution model that is better able to leverage the inherent locality in parallel computation rather than using double datarate type five synchronous graphics random-access memory (GDDR5) DRAM as the primary communication mechanism between cores.

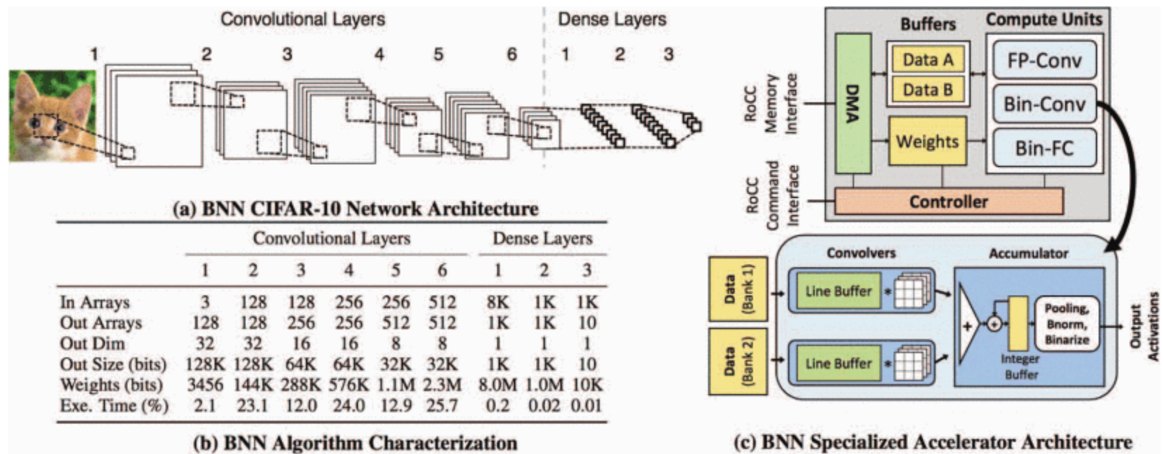
2.5 The Specialization Tier

Deciding which workload parts get implemented in the specialization tier takes careful consideration. In Celerity, we chose to implement a BNN accelerator. The architecture and reasoning for implementing a BNN in the specialization tier are discussed here.

2.5.1 Choosing the Neural Network

Deep convolutional neural networks (CNNs) are now the state of the art for image classification, detection, and localization tasks. However, using CNN software implementations for real-time inference in embedded platforms can be challenging due to strict power and memory constraints. This has sparked significant interest in hardware acceleration for CNN inference, including our own prior work on FPGA-based CNN accelerators [12]. Given this context, we chose to use flexible image recognition as a case study for demonstrating the potential of tiered accelerator fabrics in general, and the Celerity SoC specifically. Most prior work on CNN accelerators uses carefully hand-crafted digital VLSI architectures and

represent the weights and activations in 8 to 16-bit fixed-point precision. Recent work on BNNs has demonstrated that binarized weights and activations (+1, -1) can, in certain cases, achieve accuracy comparable to full-precision floating-point CNNs [13]. BNNs' key benefit is that the computation in convolutional and dense layers can be realized with simple exclusive-negated-OR (XNOR) and pop-count operations. This removes the need for more expensive multipliers and adder trees, saving area and energy. BNNs can also achieve a substantial gain (8-16X) in the memory size of weights compared to a fixed-point CNN using the same network structure, making the model easier to fit on-chip. Additionally, there is an active body of research on BNNs attempting to further improve classification performance and reduce training time. We employ the specific BNN model shown in Figure 3(a) based on Courbariaux et al [13]. This model includes six convolutional, three max-pooling, and three dense (fully connected) layers. The input image is quantized to 20-bit fixed-point, and the first convolutional layer takes this representation as input. All remaining layers use binarized weights and activations. BNN-specific optimizations include eliminating the bias, reducing the batch norm calculation's complexity, and carefully managing convolutional edge padding. This network achieves 89.8-percent accuracy on the CIFAR-10 dataset.



	Convolutional Layers						Dense Layers		
	1	2	3	4	5	6	1	2	3
In Arrays	3	128	128	256	256	512	8K	1K	1K
Out Arrays	128	128	256	256	512	512	1K	1K	10
Out Dim	32	32	16	16	8	8	1	1	1
Out Size (bits)	128K	128K	64K	64K	32K	32K	1K	1K	10
Weights (bits)	3456	144K	288K	576K	1.1M	2.3M	8.0M	1.0M	10K
Exe. Time (%)	2.1	23.1	12.0	24.0	12.9	25.7	0.2	0.02	0.01

(b) BNN Algorithm Characterization

Figure 2.3: BNN accelerator

2.5.2 Performance Target

We target ultra-low latency, requiring a batch size of one image and a throughput target of 60 classifications per second to enable real-time operation.

2.5.3 Creating and Optimizing the Specialization Tier

We use a three-step process to map applications to tiered accelerator fabrics. First, we implement the algorithm using the general-purpose tier for initial workload characterization and to identify key kernels for acceleration. Second, we can choose to accelerate the algorithm using either the specialization tier or the massively parallel tier. Finally, we can further improve performance and/or efficiency by cooperatively using both the specialization tier and the massively parallel tier.

2.5.4 Establishing the Functionality of the Specialization Tier

In the first step, we implemented the BNN using the general-purpose tier to characterize the computational and storage requirements of each layer. Figure 3(b) shows the number of binary weights and binary activations per layer in addition to the execution time breakdown, assuming a very optimistic embedded microarchitecture capable of sustaining one instruction per cycle. The total estimated execution time for the BNN software model (estimated to be around 2 billion instructions) on the general-purpose tier would be approximately 200X slower than the performance target. Although the binarized convolutional layers require more than 97 percent of the dynamic instructions, preliminary analysis suggests that all nine layers must be accelerated to meet the performance target. The storage requirements for activations are relatively modest, but the storage requirements for weights are non-trivial and require careful consideration.

2.5.5 Designing the Specialization Tier

In the second step, we implemented the BNN using a configurable application-specific accelerator in the specialization tier. This accelerator was designed to integrate with a Rocket core in the general-purpose tier through the RoCC interface. Although the massively parallel tier could be used to implement the BNN at speed, superior energy efficiency could be attained through specialization. Figure 3(c) shows the BNN accelerator architecture. The BNN accelerator consisted of modules for fixed-point convolution (first layer), binarized convolution, dense layer processing, weight and activation buffers, and a DMA engine to move data in and out of the buffers. The BNN accelerator processes one image layer at a time and can perform 128 binary multiplications (XNORs) per cycle using two convolvers. Any non-binarized computation is performed completely within each module to limit the amount of non-binarized intermediate data stored in the accelerator buffers and/or memory system. The activation buffers are large enough to hold all activations; however, in

this design, the sizeable binarized weights necessitated off-chip storage using the general-purpose RoCC memory interface. The binarized convolution unit includes two convolvers implemented with a flexible line buffer based on Zhao et al [12].

2.5.6 Combining the Massively Parallel and Specialization Tiers

In the third step, we explored the potential for cooperatively using both the specialization tier and the massively parallel tier. Our early analysis suggested that repeatedly loading the weights from off-chip would significantly impact both performance and energy efficiency. We implemented a novel mechanism that enables cores in the massively parallel tier to send data directly to the BNN. To classify a stream of images, we first load all data memories in the massively parallel tier with the binarized weights. We then repeatedly execute a small remote-store program on the massively parallel tier; each core takes turns sending its portion of the binarized weights to the BNN in just the right order. The BNN can be configured to read its weights from queues connected to the massively parallel tier instead of from the general-purpose tier.

2.5.7 The Benefits of HLS

We employed HLS to accelerate time-to-market and to enable significant design-space exploration for the BNN algorithm. The BNN model was first implemented in C++ for rapid algorithmic development, before adding HLS-specific pragmas and cycle-accurate SystemC interface specifications. Cadence Stratus HLS transformed the SystemC code into cycle-accurate RTL. Very similar C++ test benches were used to verify the BNN algorithm, the SystemC BNN accelerator, the generated BNN RTL, and the Rocket core running the BNN accelerator. This HLS-based design methodology enabled three graduate students with near-zero neural-network experience to rapidly design, implement, and verify a complex application-specific accelerator.

2.6 Performance Analysis Of The Specialization Tier

Table 1 shows the performance and power of optimized BNN implementations on the Celerity SoC and other platforms. Although each platform uses a different implementation methodology, technology, and memory system, these results can still provide a rough high-level comparison. These results suggest that the Celerity SoC can potentially improve performance/Watt by more than 10X compared to our prior FPGA implementation [12] and more than 100X compared to a mobile GPU.

	GPT	SpT	SpT+MPT	mGPU [6]	CPU [6]	GPU [6]	FPGA [6]
Runtime (ms)	4,024.0	20.0	3.2	90.0	14.8	0.7	5.9
Performance (images/sec)	0.3	50.0	312.5	11.1	67.6	1428.6	168.4
Power (Watts)	0.5	0.5	1.9	3.6	95	235.0	4.7
with aggressive clock gating	0.1	0.2	0.4				
Efficiency (images/J)	0.5	100.0	164.5	3.0	0.7	6.0	35.8
with aggressive clock gating	2.5	250.0	625.0				
Relative Efficiency vs. GPT	1×	100×	250×				
Relative Efficiency vs. mGPU			208×	1×	0.2×	2×	12×

*GPT = general-purpose tier. SpT = specialization tier with the weights stored in the general-purpose tier's cache. SpT + MPT = specialization tier with the weights stored in the massively parallel tier. mGPU = Nvidia Jetson TK1 embedded GPU board. CPU = Intel Xeon E5-2640. GPU = Nvidia Tesla K40. FPGA = Xilinx Zynq-7000 SoC.

Table 2.1: Performance comparison of optimized BNN implementations on different platforms.

In the table, runtimes measure processing a single image from the CIFAR-10 dataset. The power of GPT, SpT, and SpT + MPT is estimated using post-place-and-route gate-level simulations with limited clock-gating, as provided in the Celerity SoC (only gating the entire MPT when unused). Aggressive clock-gating assumes an alternate design that can gate unused cores/accelerators in the GPT, SpT, and MPT. Celerity SoC power estimates do not include DRAM power.

2.7 New Directions For Fast Hardware Design

Our research examines the speedy construction of new classes of chips in response to emerging application domains. Our approach was successful due to a heterogeneous architecture that offers fast construction, scalability, and heterogeneous interoperability through the remote store programming model and advanced producer-consumer synchronization methods like LR-LBR and token queues. At the same time, our design methodology combines HLS for specialized tier accelerator development, open-source technology like Rocket and BaseJump for key IP blocks, fast motherboard and socket development and FPGA firmware, and principled SystemVerilog parameterized component libraries like BaseJump Standard Template Library (STL). Finally, our agile chip development techniques enabled us to quickly tape out a 16nm design with a team of graduate students geographically distributed across the US. Each approach targets the key goal of creating new classes of chips quickly and with low budgets. We hope that the lessons from our experience will inspire new classes of chips, unlocking the creativity of future students, architects, and chip designers alike.

2.8 Achieving Celerity With Celerity: Fast Design Methodologies For Fast Chips

Celerity was designed under the DARPA Circuit Realization at Faster Timescales (CRAFT) program, whose goal was to reduce the design time for taping out complex SoCs. To meet the aggressive schedule for Celerity, we developed three classes of techniques to decrease design time and cost: reuse, modularization, and automation.

2.8.1 Reuse

Reuse for hardware design accelerates both design and implementation time, as well as testing and verification time. For Celerity, we made heavy reuse of open-source designs and infrastructures. We leveraged the Berkeley RISC-V Rocket core generator3 to implement the SoC's general-purpose tier, allowing the reuse of Rocket's testing infrastructure and the RISC-V toolchain. The same infrastructure was used for the manycore array's Vanilla-5 core. Because validation is usually more work than design, inheriting a robust test infrastructure greatly reduced overall design time. We leveraged the RoCC interface for all connections to the general-purpose tier. As part of our learning process with RoCC, we created the "RoCC Doc," located at http://openceler- ity.org. Beyond the RISC-V ecosystem, we leveraged the BaseJump open-source hardware components, which can be found at http://bjump.org. BaseJump provides open-source infrastructure and frameworks for designing and building SoCs, including the BaseJump STL [14] for SystemVerilog, the BaseJump SoC framework, BaseJump Socket, BaseJump Motherboard, BaseJump FPGA bridge, and BaseJump FMC bridge, as seen in Figure 4. In Celerity, we built all of our RTL using the Basejump STL and SoC framework's pre-validated components and unit testing suite. We ported the BaseJump Socket to the CRAFT flip-chip package and will use the BaseJump Motherboard for the final chip. By leveraging the unit testing suite from BaseJump and RISC-V testing infrastructure, we could focus our verification efforts primarily on integration testing. Using an FPGA in place of the SoC, the BaseJump infrastructure allows for designs to be simulated in the same two board environment they will be running in post-tapeout. All firmware and test-bench code written during simulation will be reused during bring-up once the chip returns from fabrication, giving us a robust verification and validation suite. Reuse is also enabled by extensibility and parameterization. Due to the scalable nature of tiled architectures, BaseJump STL's parameterization, and the flexibility of our backend flow methodology, we were able to extend the BaseJump manycore array from 400 cores to 496 to absorb free die area. By changing just nine lines of code, we could

fully synthesize, place, route, and sign off on the new design in a span of three days.

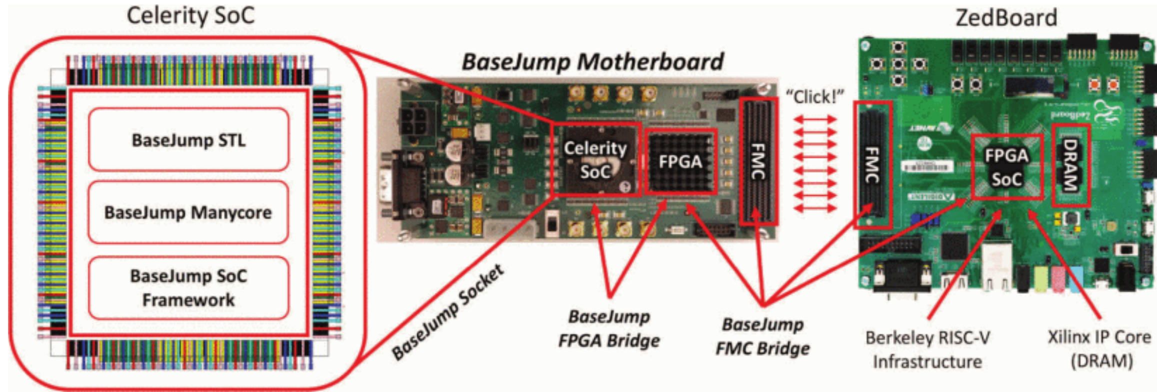


Figure 2.4: BaseJump open-source hardware components. The SoC framework, including NoC and high-speed off-chip double data-rate (DDR) interface, were implemented using STL, as was the manycore. The fabricated chip conforms to the socket definition and is placed in the motherboard's socket. The motherboard connects through an FMC connector to a ZedBoard hosting RISC-V testing infrastructure. Communication between the motherboard and ZedBoard is handled with the open-source FMC bridge code.

2.8.2 Modularization

One key challenge for this project was that our design teams were spread across four physical locations. Fine-grained synchronization between teams was not feasible, so we developed techniques to modularize both our design interfaces on chip and our interfaces between teams. Many techniques we used can be compared to an agile design methodology as it applies to hardware. We used a bottom-up design flow to build, iterate, and integrate smaller components into a larger design. We also used a SCRUM-like task management system, where we clearly identified and prioritized various tasks and issues, minimized synchronization issues, and distributed tasks across team members without assigning rigid specialized roles. We also defined tape-in [15] deadlines. These are simpler designs that were tapeout ready before the deadline. This allowed us to stress-test our physical design flow early in the design cycle, in addition to identifying big-picture problems early on, which we found particularly useful when dealing with an advanced technology node. Each successive tape-in incorporated an additional IP block, building up to what we see in Celerity. We performed daily chip builds to ensure no changes broke the overall design and that we always had a working design to tapeout. To help modularize the RTL, chip component interfaces were established early. We selected RoCC early on for on-chip communication and BaseJump for off-chip communication. Because we used BaseJump STL's pervasive

latency-insensitive interfaces, our architecture-specific dependencies between components were minimized.

2.8.3 Automation

CRAFT's tight time constraints required that we employ higher degrees of automation to accelerate the design cycle. We developed an abstracted implementation flow to minimize the changes necessary for different designs to go from synthesis through sign-off. We combined vendor reference scripts with an integration layer to coalesce implementation parameters and separate scripts into design-specific and process-specific groups. We could then quickly identify which scripts needed to be modified between designs.

We also took advantage of emerging tools and methodologies. We used the PyMTL framework for rapid test-bench development using high-level languages and abstractions rather than lowlevel SystemVerilog. In our BNN accelerator development, we used HLS to drastically improve design space exploration and implementation time.

CHAPTER 3

Rapid Architectures: FASoC

FASoC is a fully autonomous mixed-signal SoC framework, driven entirely by user constraints, along with a suite of automated generators for analog blocks. The process-agnostic framework takes high-level user intent as inputs to generate optimized and fully verified analog blocks using a cell-based design methodology.

Our approach is highly scalable and silicon-proven by an SoC prototype which includes 2 PLLs, 3 LDOs, 1 SRAM, and 2 temperature sensors fully integrated with a processor in a 65nm CMOS process. The physical design of all blocks, including analog, is achieved using optimized synthesis and APR flows in commercially available tools. The framework is portable across different processes and requires no-human-in-the-loop, dramatically accelerating design time.

Work presented in this chapter has been presented in Government Microcircuit Applications and Critical Technology Conference [16], a poster at The Symposium on High Performance Chips (Hot Chips), and at IFIP/IEEE International Conference on Very Large Scale Integration [17].

3.1 Introduction

There is an ever-growing need for automation in analog circuit design, validation, and integration to meet modern-day SoC requirements. Time-to-market constraints have become tighter, design complexity has increased, and more functional blocks (in number and variety) are being integrated into SoCs. These challenges often translate to increased manual engineering efforts and non-recurring engineering (NRE) costs. We present FASoC, an open-source[1] framework for Fully-Autonomous SoC design. Coupled with a suite of analog generators, FASoC can generate complete mixed-signal system-on-chip (SoC) designs

[1] Source code for the framework and all generators developed as part of this work can be downloaded from https://fasoc.engin.umich.edu

from user specifications. The framework leverages differentiating techniques to automatically synthesize correct-by-construction RTL descriptions for both analog and digital circuits, enabling a technology-agnostic, no-human-in-the-loop implementation flow.

Analog blocks like PLLs, LDOs, ADCs, and sensor interfaces are recasted as structures composed largely of digital components while maintaining analog performance. They are then expressed as synthesizable Verilog blocks composed of digital standard cells and auxiliary cells (aux-cells). We employ novel techniques to automatically characterize aux-cells and develop models required for generating bespoke analog blocks. The framework is portable across processes, EDA tools and scalable in terms of analog performance, layout, and other figures of merit.

The SoC generation tool translates user intent to low-level specifications required by the analog generators. To achieve full SoC integration, we leverage the IP-XACT [18] standard and added vendor extensions to capture additional meta-data from the generated blocks. This enables the composition of vast numbers of digital and analog components into a single correct-by-construction design. The fully composed SoC design is finally realized by running the Verilog through synthesis and automatic place-and-route (APR) tools to realize full design automation.

3.2 Framework Architecture

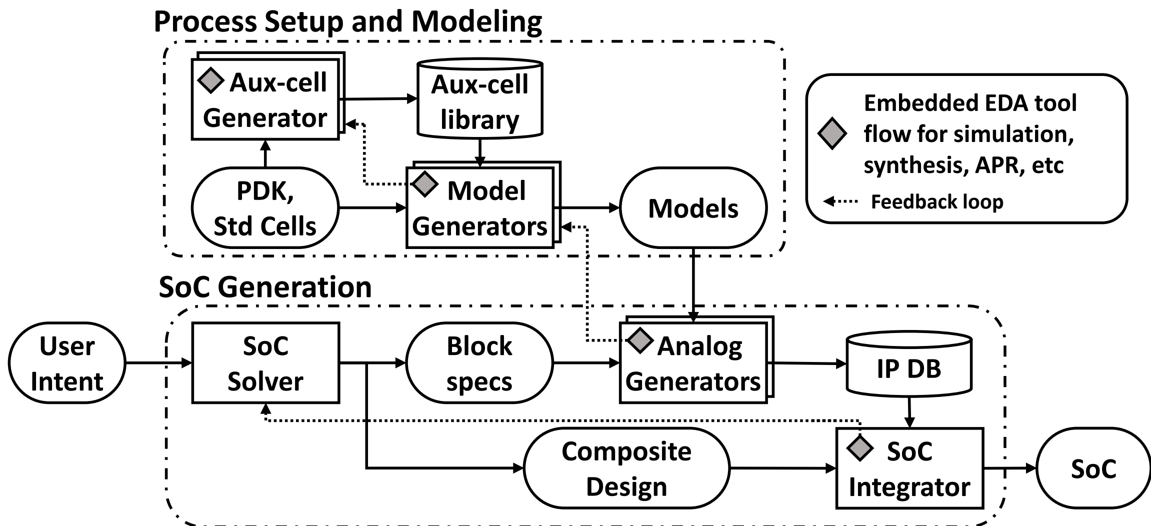


Figure 3.1: FASoC Framework Overview

A high-level representation of the framework is shown in Fig. 3.1. The *Process setup and modeling* phase is performed once for the process design kit (PDK), and it involves

the generation of the aux-cells and models for the generator. The *SoC generation* phase begins by translating high-level user-intent into analog specifications that satisfy the user constraints. The block generators are invoked as needed and the SoC integrator stitches the composed design and walks it through a synthesis and APR flow to create the final SoC layout. The FASoC framework is tightly integrated with analog generators for PLL, LDO, temperature sensor, and SRAM blocks. Section 3.3 describes the circuit architecture adopted by the different generators.

3.2.1 Process Setup and Modeling

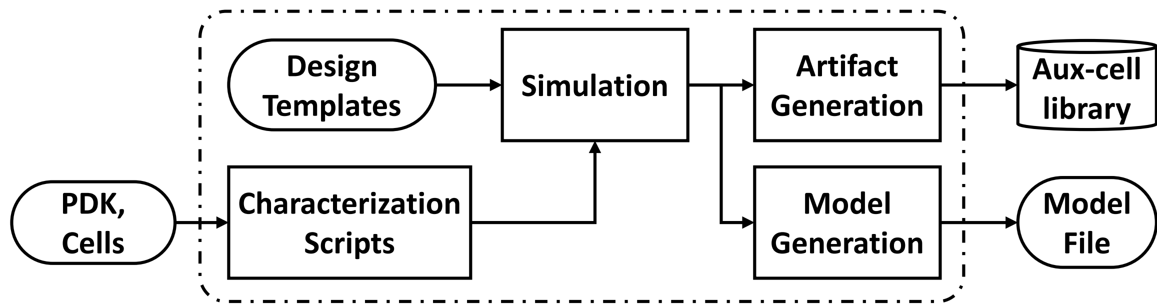


Figure 3.2: Aux-cell and model file generation flow

FASoC employs a synthesizable cell-based approach for generating analog blocks, significantly cutting back on manual layout and verification efforts. Aux-cells are small analog circuits that buttress the standard cell library and provide specific analog functionality required by the generators. Each cell is no larger than a D flip-flop and can be placed on the standard cell rows. We simplify the creation of aux-cells by using a suite of design templates in tandem with PDK characterization scripts. The templates capture the aux-cell's precise circuit behavior without including any PDK-specific information. The characterization scripts operate on the PDK to derive technology-specific parameters required to set knobs within the templates. Example parameters extracted from the PDK include threshold voltage, metal parasitics, MOSFET behavior, and Fan-out of 4. The knobs set within the template include device type, transistor sizing, and other circuit design options. The results from aux-cell generation include the netlist, layout, timing library, and other files required to proceed with conventional synthesis and APR. At present, the layouts for the aux-cells are manually created, however, we are currently evaluating several layout automation tools [19–21] that are showing promising results. We find our template-based methodology for creating aux-cells enhances process-portability and significantly cuts down on design time. All of the generators presented in this work leverage 8 aux-cells that are depicted in

Fig. 3.3.

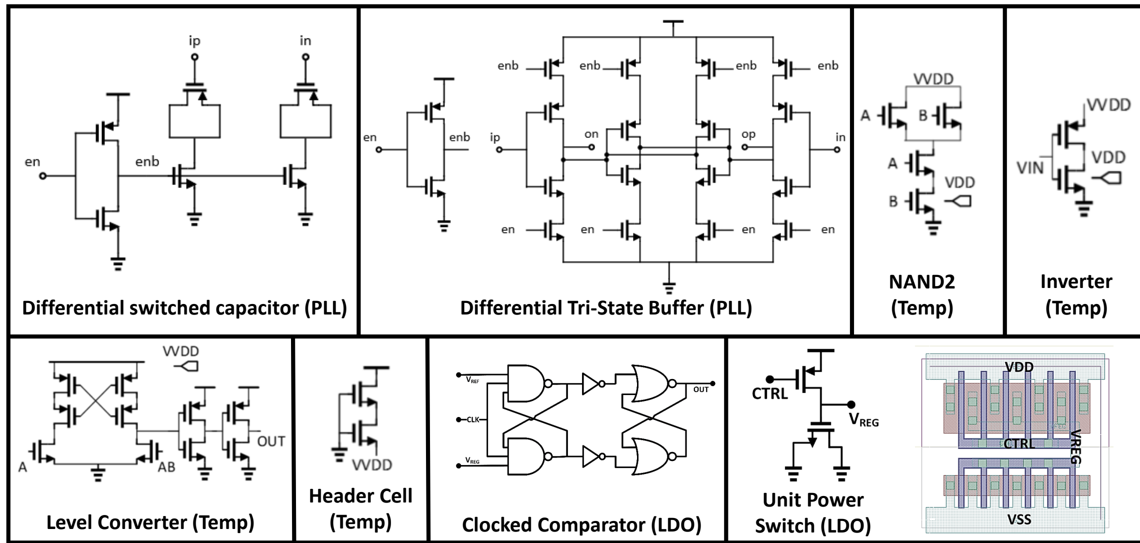


Figure 3.3: Schematic for aux-cells used across PLL, LDO and temperature sensor generators

The analog generators use models to predict performance and select design parameters to create optimized block designs that satisfy the input specifications. The models are derived from the parameterized templates that incorporate the aux-cells. The models for each generator vary and are developed from a combination of mathematical equations, machine learning, and design space exploration. The modeling exercise is also performed once per PDK and the results are saved into a model file. Sections 3.3 briefly describes the modeling approach adopted by each generator integrated into the framework.

3.2.2 SoC Generator

This stage begins with an iterative *SoC solver* to determine the optimal *composite design* description which is a combination of blocks, analog specifications, and connections. The strategy is guided by high-level user intent (i.e. target application and power/area budgets), available analog block generators, and a database of IPs. Analog generators are invoked as necessary to generate bespoke blocks required to satisfy the specifications within the composite design. The generator outputs include all artifacts required to push the block through standard synthesis and APR tools. The outputs are also cached in an *IP database*, allowing for block generation to be skipped if a matching entry already exists. Entries in the database can also be populated with 3rd party IPs such as processors and other peripherals.

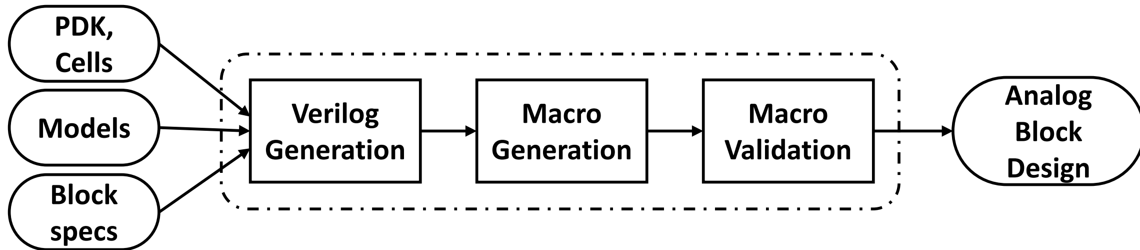


Figure 3.4: Analog generator flow

We adopt the IP-XACT format to describe the composite design as well as the block designs stored in the database. We use an extended format [16] to capture additional analog data, simulation, and verification information.

The *SoC integrator* begins by stitching the composite design together and translating it to its structural Verilog equivalent that can be run through digital simulation tools. The structural Verilog, along with all required artifacts from the database, is then passed through the embedded tool flow to generate the final verified GDS. This same flow is pervasive across the framework and is also used by all generators (aux-cell, model, and analog). Tools within the flow cover all aspects of chip design including SPICE simulations, digital simulations, synthesis, APR, DRC, LVS, and extraction.

3.3 Analog Generator Architecture

Synthesizable analog blocks were introduced a few decades ago and have continued to evolve, closely matching the performance obtainable by full custom designs. Prior works have described techniques for synthesizing analog blocks for UWB transmitters [22], PLLs [23], DACs [24], and other types of analog blocks [25–27]. This approach lowers engineering design costs, increases robustness, eases portability across PDKs, and continues to show promise even at advanced process nodes [5,28,29]. The analog generators developed as part of this work can be likened to ASIC memory compilers that take in a specification file and produce results in industry-standard file formats, which can then be used in standard synthesis and APR tools. Unlike typical memory compilers, our generators are open-source, process agnostic, and share a scalable framework amenable to different types of blocks. The framework is modular and share a similar process as depicted in Fig. 3.4. The full generation process is broken down into three steps:

Verilog Generation: This step leverages models to produce a synthesizable Verilog description of the block that conforms to the input specifications. It also generates guid-

ance information in a vendor-agnostic format. The guidance includes synthesis constraints, placement instructions, and other information that may be required by the synthesis and/or APR tool to generate blocks that achieve the desired performance. In addition, this step also reports early estimates on performance and the characteristics of the block to be created.

Macro Generation: The Verilog and guidance information is passed to a digital flow to create macros that can be embedded into larger SoC designs. The digital flow in this step performs synthesis, APR, DRC, and LVS verification. The digital flow includes an adapter to translate the guidance into vendor-specific commands used in synthesis and APR. The adapter abstraction allows us to (1) express additional design intent without exposing protected vendor-specific commands and (2) easily support multiple cad tools including open-source alternatives [30–32]

Macro Validation: The last step is a comprehensive verification and reporting of the generated block. The full circuit goes through parasitic extraction, SPICE simulations, requirement checks and other verification to culminate in a detailed datasheet report.

The generators can be invoked standalone, outside of the full SoC generator flow. To simplify the system integration, we adopt the AMBA™ APB protocol as the register interface to all blocks. The following sub-sections briefly describe the analog generators currently integrated into the FASoC framework.

3.3.1 PLL

The generated PLLs (Fig. 3.5) share the same base architecture as ADPLL [33]. The phase difference of the reference and output clocks are captured by the embedded time-to-digital converter (TDC), while the digital filter calculates the frequency control word for the digitally controlled oscillator (DCO). The input specification to the generator defines the nominal frequency range and in-band phase noise (PN). The PLL generator uses a physics-based mathematical model [34] for characterization. We first build a mathematical relationship between DCO design parameters (number of aux-cells and stages) and the required DCO specifications. Using simulation results from a parametric sweep, we then find the effective ratio of drive strength and capacitance for each aux-cell. This ratio enables us to predict frequency and power results (frequency range, frequency resolution, frequency gain factor, and power consumption) given a set of input design parameters.

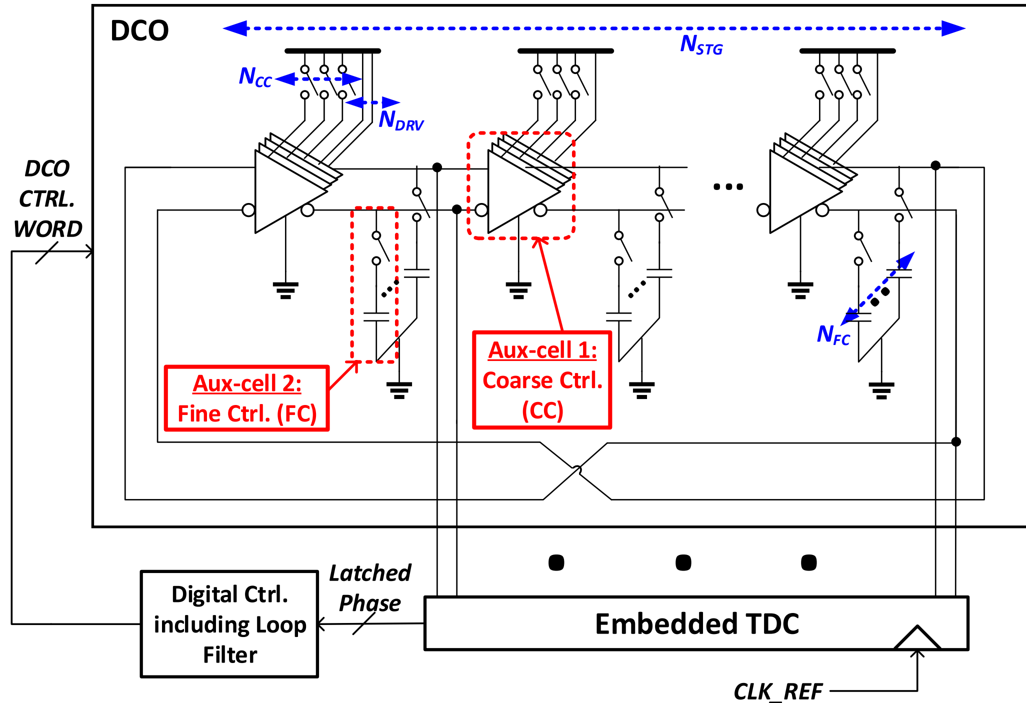


Figure 3.5: DCO architecture indicating the aux-cells and designs parameters

3.3.2 LDO

The generated LDOs (Fig. 3.6) share the same base architecture as DLDO [35]. The LDO leverages an array of small power transistors that operate as switches for power management. Based on design requirements, the generator can swap the clocked comparator with a synthesizable stochastic flash ADC [36] to improve transient response. The input specifications to the LDO generator are the V_{IN} range, $I_{load,max}$ range, and the dropout voltage. The generator uses a poly-fit model of the load current ($I_{load,max}$) performance with respect to various combinations of aux-cell connections (connected in parallel and for different VDD inputs) in both ON and OFF states. We create the model by simulating various test circuits after parasitic extraction.

3.3.3 Temperature Sensor

The generated sensors (Fig. 3.7) share the same base architecture as [37]. The sensor relies on a temperature-sensitive ring oscillator and stacked zero-VT devices for better line sensitivity. The input specifications include the temperature range and optimization strategy, for either error or power. For a given temperature range, the models attempt to select the optimal circuit topology that minimizes error and/or performance. The generator

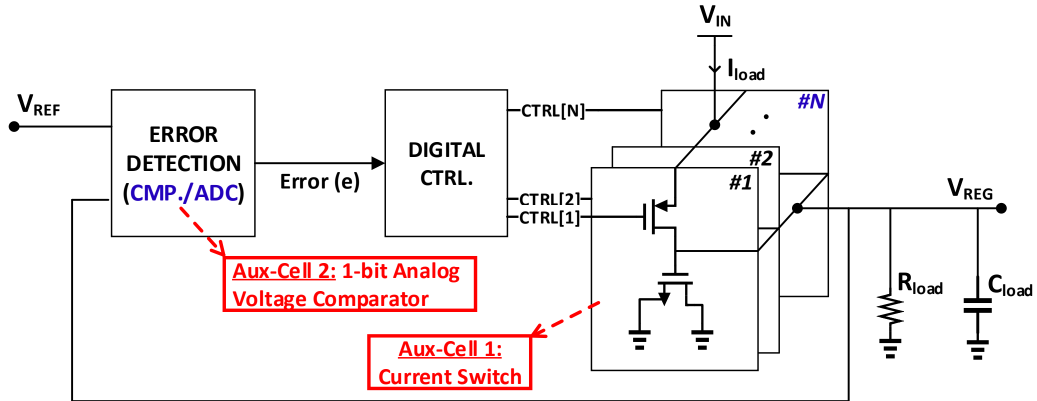


Figure 3.6: LDO architecture indicating the aux-cells and design parameters derived from input specifications of V_{IN} , I_{load} and desired transients

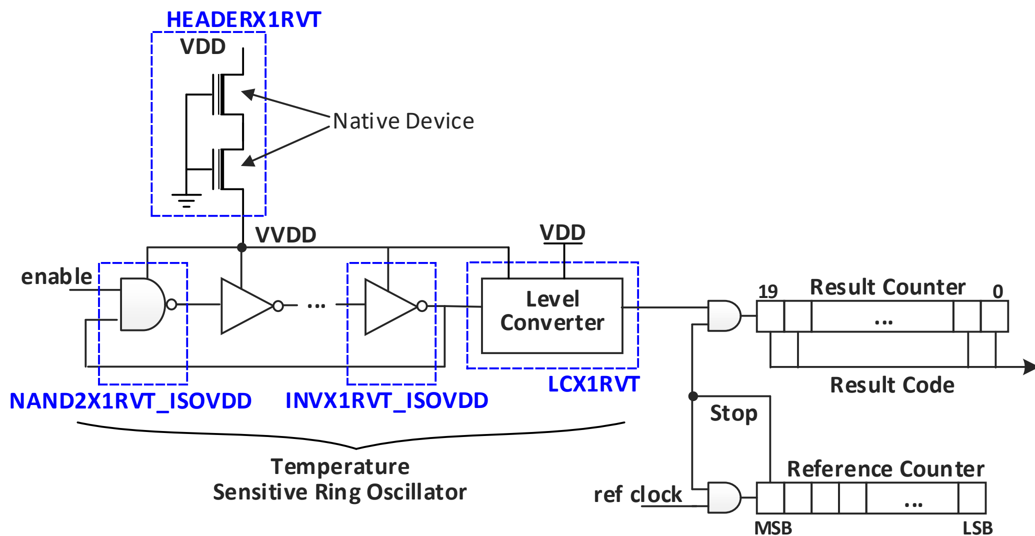


Figure 3.7: Temperature sensor architecture indicating the aux-cells

relies on a predictive Bayesian neural network model to select design parameters that satisfy the input specifications.

3.3.4 SRAM

The compiled SRAMs (Fig. 3.8) follow a standard multi-bank memory architecture. Unlike other generators in the framework, the memory generator uses a combination of macros instead of aux-cells. The macros used include a 6T bitcell, a row decoder, column mux, wordline driver, sense amplifier, write driver, and a pre-charge circuit. The macros are stitched together, bottom-up, to form a bank. The user input specifications are capacity, word size, operating voltage, and operating frequency. The generator adopts a hierarchi-

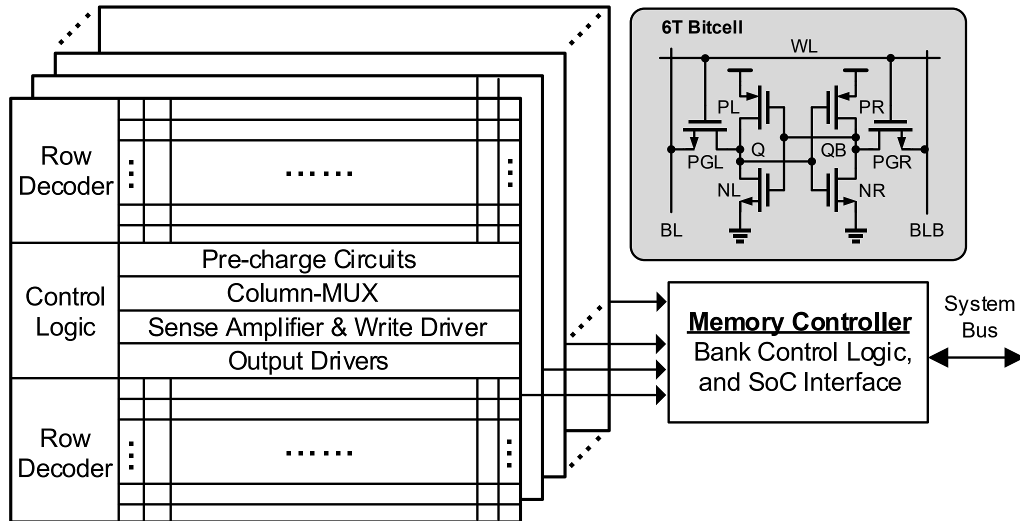


Figure 3.8: SRAM architecture showing macros and bank strategy

cal meta compiler (HMC) [38] for technology characterization and a hierarchical memory model to determine the optimal row and column periphery. The model helps to select the SRAM architecture and the leaf-level components that best satisfy the user specifications while minimizing energy consumption and delay.

3.4 Evaluation

The framework has been fully validated in a 65nm process. Our evaluation begins with a focus on the individual generators. We present results that explore the design-space possible with each generator and demonstrate full adherence to the user input specification. We then present results from a prototype SoC created using this framework.

3.4.1 Analog Generation Results

Fig. 3.9 presents the results of several PLLs generated using different input specifications. It compares the input requirements against the simulated results after parasitic extraction. The results show that the generated frequency ranges cover that of the input requirements and with better phase noise levels. The highlighted PLL 8, corresponds to one of the PLLs integrated into the SoC prototype and also shows measured results that satisfy the given specifications

Fig. 3.10 shows the spice simulation results of multiple LDO designs after parasitic extraction. The graph shows the maximum load current at different input voltages corre-

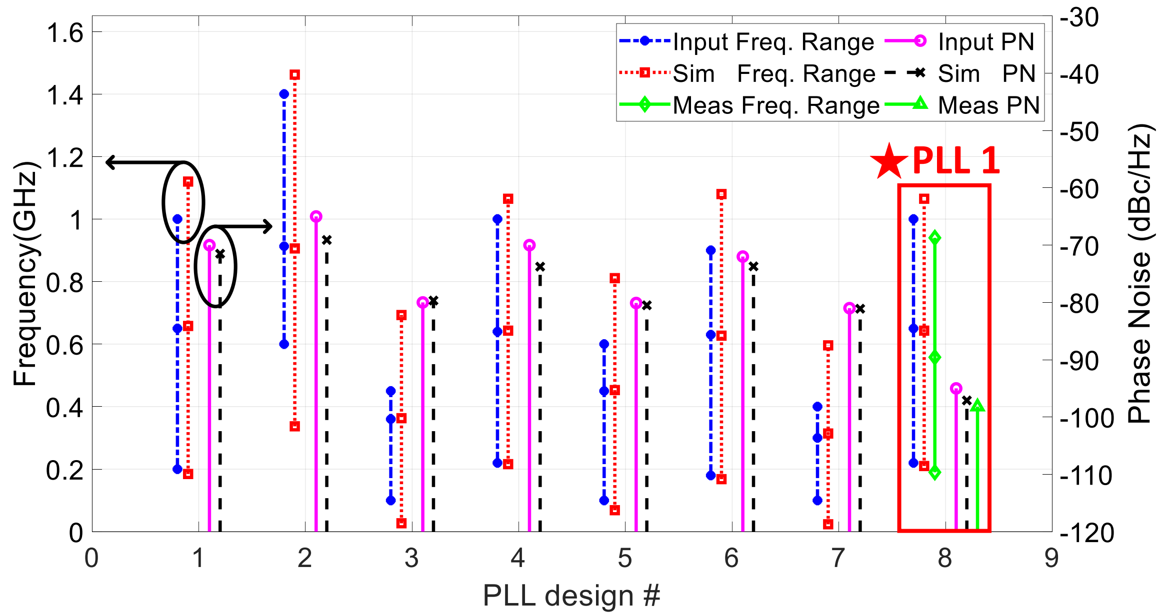


Figure 3.9: Generated PLL designs for eight different input specifications. PLL1 is taped-out in the SoC prototype

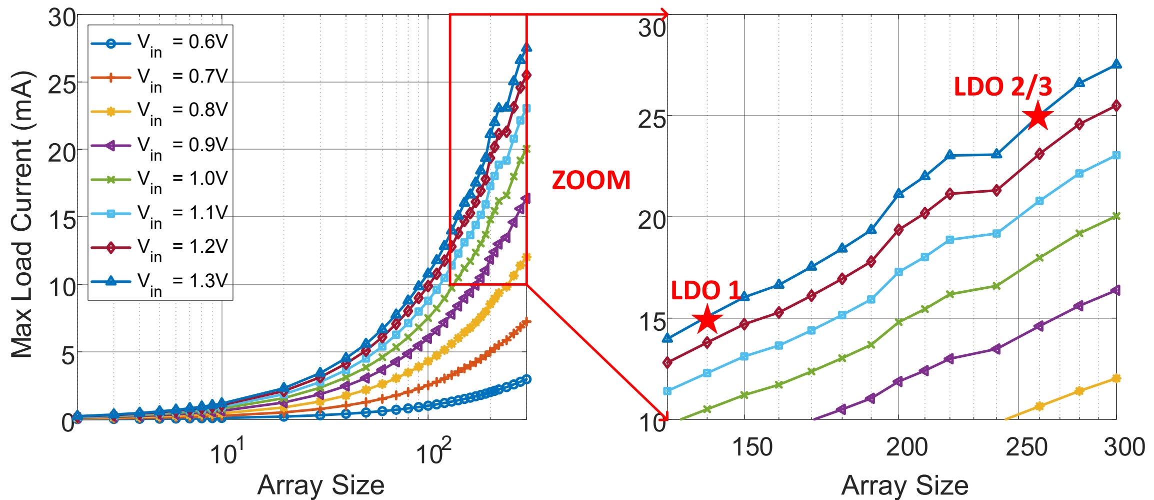


Figure 3.10: $I_{load,max}$ vs. array size, for multiple LDO designs generated

sponding to the input parameter array size for a dropout voltage of 50mV. The highlighted measurements correspond to the input specification for blocks integrated into the SoC prototype with $V_{IN} = 1.3V$ and $V_{REG} = 1.2V$.

Fig. 3.11 presents the simulation results of various memory capacities across a broad range of architectural options and operating voltages (VDD). Each point on the curve corresponds to an energy-delay pair specific to an architecture (rows, columns, and banks) and

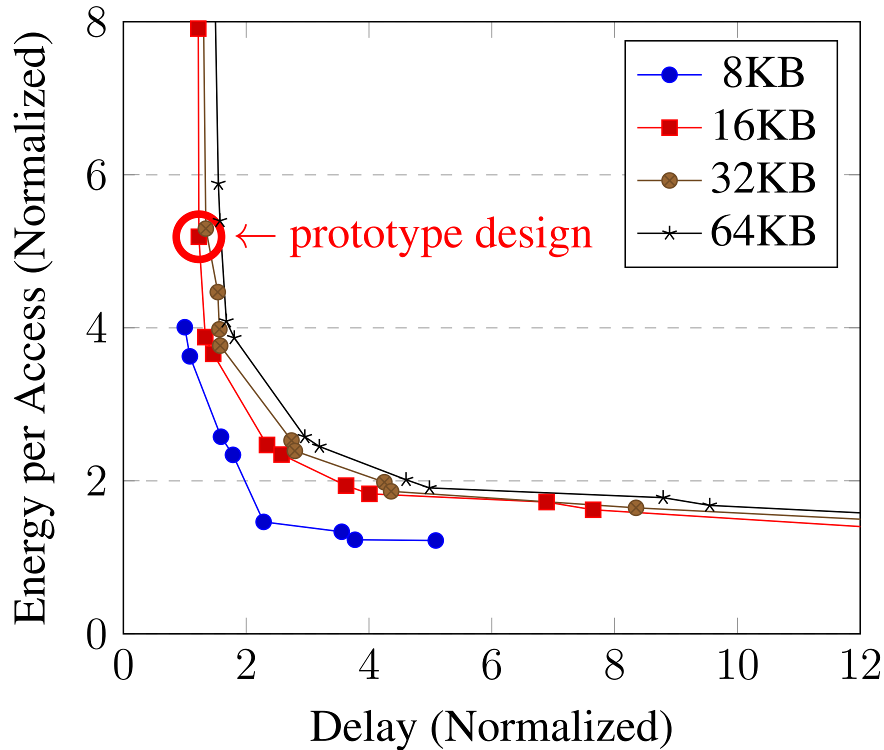


Figure 3.11: Normalized energy and delay plots for various memory sizes while sweeping VDD. The results are normalized with respect to the 8KB memory.

VDD combination. The generator selects the Pareto-optimal design that satisfies the user requirements. The highlighted point on the 16KB curve corresponds to the memory block integrated into the SoC prototype.

Fig. 3.12 shows the spice simulation results of multiple temperature sensor designs after parasitic extraction.

3.4.2 Prototype Chip Results

The prototype SoC design (Fig. 3.14) includes 2 PLLs, 3 LDOs, 1 16KB SRAM, and 2 temperature sensors fully integrated with an Arm® Cortex™-M0 in a 65nm CMOS process. Using off-chip connections, we were successfully able to power the entire SoC using one of the LDOs and clock it using the PLLs, while monitoring the temperature of the chip.

Fig. 3.9 presents results for 8 PLL designs generated from different input specifications, including one from the prototype, and the results show output performances in-line with the input specifications. The measured frequency is 10% slower while the phase noise matches

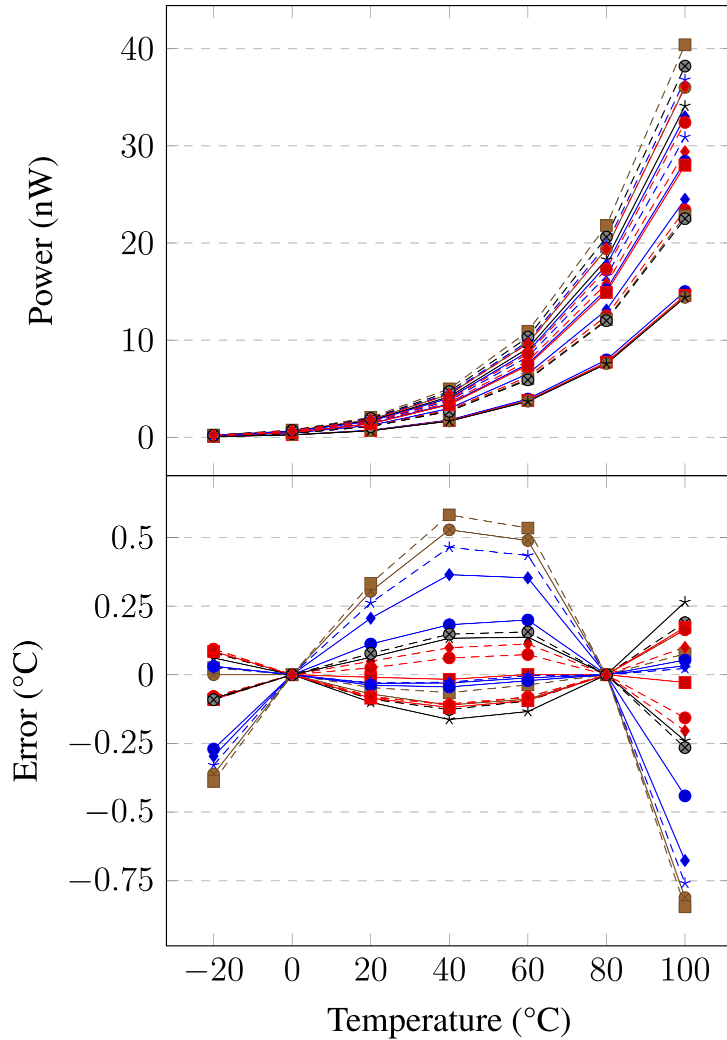


Figure 3.12: Power and Error results against temperature for various temperature sensor designs (each fitted plot represents a unique design)

the simulation and specification requirement. Table 3.1 summarizes the results for all PLLs in the prototype.

Table 3.2 shows the LDO $I_{load,max}$ measurements closely matching the input specification requirements. Compared to the comparator-based architecture (LDO1/2), the ADC based controller architecture (LDO3) achieves better transient performance with a 10x and 7x improvement in settling time and undershoot voltage respectively. The line and load regulation values are measured at V_{IN}=1.3V, V_{REF}=1.2V, and I_{load}=10mA. LDO3 load regulation is comparatively worse due to the high gain of the ADC based controller. As we operate at lower V_{REF} and I_{load} conditions, the line/load regulation degrades for all the LDOs because of the increase in relative switch strength.

Table 3.1: PLL Simulation vs Measurement Results

Output Specifications	**PLL1**		**PLL2**	
	Sim	Meas	Sim	Meas
Min Freq (MHz)	200	190	170	150
Max Freq (MHz)	1,060	920	1,080	930
F_{nom} (Mhz)	643	558	627	548
Power@F_{nom} (mW)	7.20	6.90	8.06	7.70
Area (µm^2)	167,639.04		167,639.04	

Table 3.2: LDO Simulation vs Measurement Results @ 200MHz control clock

Output Specifications	**LDO1**		**LDO2**		**LDO3**	
	Sim	Meas	Sim	Meas	Sim	Meas
Dropout Voltage (mV)	50	70	50	80	50	80
$I_{load,max}$ (mA)	15.00	15.38	25.00	24.84	25.00	23.72
Settling Time - Ts (µs)	1.1	1.8	2.1	2.9	0.12	0.19
Max Undershoot (V)	0.35	0.98	0.57	0.98	0.38	0.14
Max Current Eff. (%)	94.2	96.4	95.7	94.5	81.9	74.0
Load Regulation (mV/mA)	-	-1.00	-	-0.35	-	-3.6
Line Regulation (V/V)	-	0.180	-	0.004	-	0.950
Area (µm^2)	17,318.56		31,187.56		127,163.56	

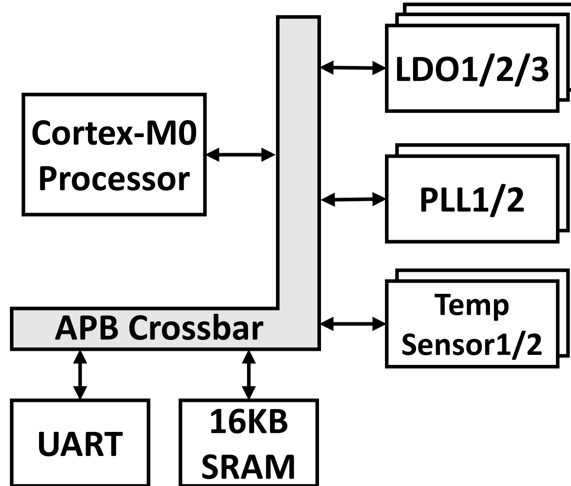


Figure 3.13: Simplified block diagram for the 65nm prototype SoC

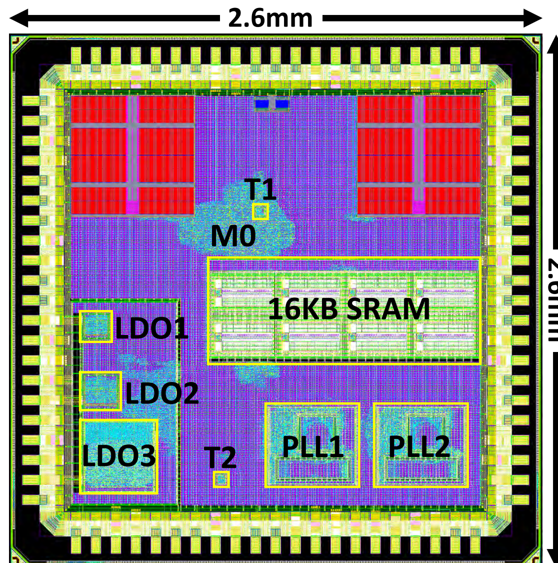


Figure 3.14: Annotated die photo for the 65nm prototype SoC

The temperature sensor has an area of 2,620μm^2. A 2-pt calibration is performed at 0°C and 80°C. Measured results show a sensing range between -20°C and 100°C with an accuracy of ±4°C.

Fig. 3.15 summarizes the SRAM measured and simulated performance across the input operating voltage range of 0.8V to 1.2V. The SRAM peak performance is at 65MHz with the power consumption of 2.09mW at 1.2V, which exceeds the targeted frequency of 50MHz. The measured power for the SRAM also includes the leakage power of the processor and peripheral interface. The generated SRAM has an area of 0.68mm^2 with the

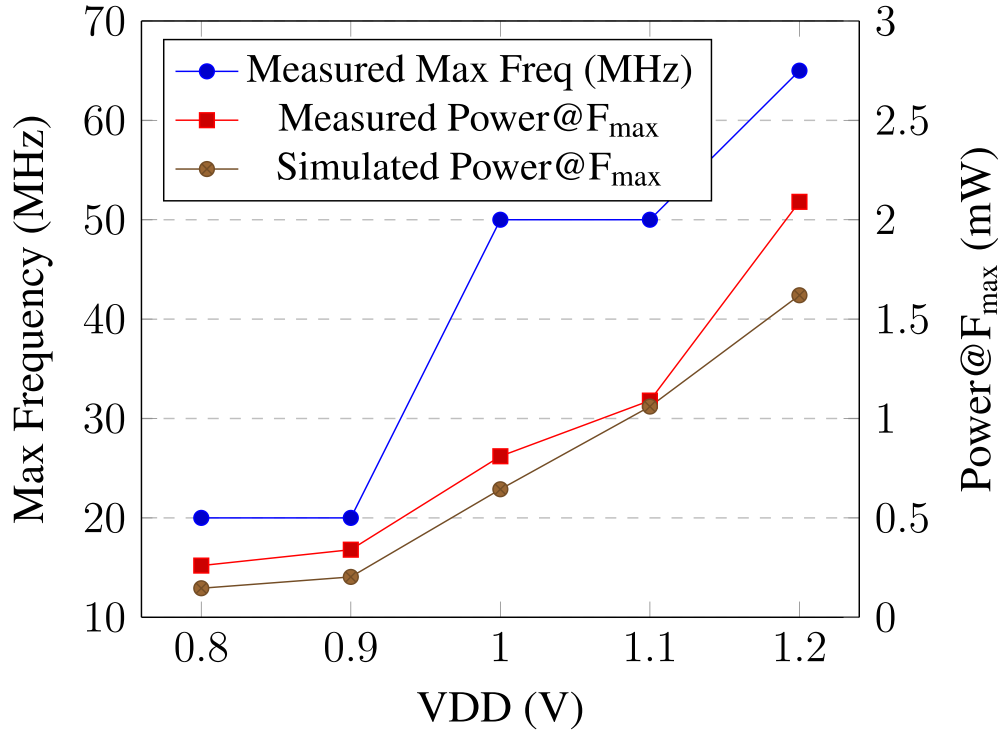


Figure 3.15: Measured and simulated performance and power results of SRAM across VDD

custom bitcell area occupying $0.4mm^2$.

3.5 Conclusion

We presented an autonomous framework that generates a completely integrated SoC design based on user input specifications. This framework is PDK agnostic and allows for faster turn-around times when building custom analog blocks and integrating them into larger SoC designs. The framework includes generators for PLL, LDO, temperature sensor and SRAM blocks. The framework can easily be extended to support more generators and different PDKs. We fabricated an SoC prototype in a 65nm process and presented silicon measurements to validate the framework's accuracy. Our work establishes a new milestone in creating a silicon compiler [39] that further reduces the complexity of realizing modern SoCs and cuts down on design time.

CHAPTER 4

Rapid Methodologies: Cadre Flow

The Cadre Flow is an offshoot project from celerity and aims to deliver on a process agnostic methodology to accelerate the physical implementation of chips using commercial tools. It focuses on the workflow for developing integrated circuits, moving designs from Register Transfer Level (RTL) to a fully implemented physical design (in GDS file format). This phase in the SoC design is usually referred to as RTL-to-GDS or *Back-end* flow

Cadre Flow is robust and adaptable to arbitrary designs across different technology nodes. It has support for 13 process development kits (PDKs) and has been validated with 6 different tape-outs across different 6 PDKs. The flow has been shared to over 7 institutions and validated across 3 different premises.

4.1 Introduction

A major milestone in the course of developing an SoC is the *completion* of the behavioral RTL. The steps to achieve this is often referred to as the *Front-end* flow. The RTL is a representation of the design that can be simulated and accurately describes the full behavior of the chip as intended by the designers. This representation, however, contains little to no information regarding the physical manifestation of the chip. Beyond this step is physical implementation (often referred to as *Back-end* flow) which focuses on the realization of the circuit into a physical implementation, whether on an ASIC or FPGA.

The behavioral model lacks several critical information including timing constraints (clock structures, etc.), physical constraints (ports, pads, placement, guides, packaging, etc.), and information relating to the target technology process/kit. To create any fabricatatable chip, all this information and more need to be passed on to the CAD tools and then evaluated for adherence and effectiveness.

While CAD tools have advanced greatly over the last few decades, there is a myriad of physical design choices and decisions that the designer is required to make (and analyze)

in order to improve the quality of results (QoR). This is often a black art that requires experience and expertise that is built up over time. There is a myriad of knobs, techniques, corner cases, intermediate outputs (with various formats depending on commercial tool vendor) and requires significant tool knowledge and technology familiarity.

The traditional back-end workflow involves several steps performed in sequential order in a waterfall approach. The steps constituting physical implementation can be broadly categorized into 3 groups:

Synthesis: This is the synthesis, optimization, and mapping of behavioral logic into a gate-level netlist that is mapped to the standard cell libraries.

Place and Route (PAR): This encompasses the cell placement, clock-tree synthesis, net routing, and optimization of the physical design.

Verification: This verifies timing closure, power analysis, design rule checks (DRCs), and layout-vs-schematic (LVS) comparison.

Although the steps are executed sequentially, design decisions and subtleties from one step (and even the RTL) significantly affect the QoR of downstream steps. Often, designers will want to quickly race their designs through the flow to foreshadow decision impacts and address them early in the design phase. This often leads to a complex negotiation on all aspects of the chip design in order to maximize the QoR.

The primary objective of the project is to develop *best practices*, build-up institutional expertise, increase code re-usability, and allow for quicker iterations on physical design implementation. The workflow will also serve as a repository of knowledge applicable to all designs and technology nodes. The project specifically targets state of the art designs at advanced silicon nodes (less than 28nm) with industry level figures of merit (Power, Performance, and Area) and verification.

The cadre flow is a set of scripts and methodologies that leverage commercial CAD tools to accomplish the task of walking an arbitrary design through the implementation process as quickly as possible. It forms a design methodology that leverages re-use to create a *turn-key* solution once the RTL design is complete. We take a modular approach to address the challenges and eliminate redundant work which allows for *agile hardware design*. While CAD tool vendors often provide example flows or reference methodologies, these are often too rigid to adapt easily to arbitrary designs and technologies. Additionally, the references provided are often disjoint and/or siloed to a specific vendor tool step that does not encompass the entire back-end flow.

4.2 Design

The flow was initially put together as part of the CRAFT program to tape-out celerity in a 16nm TSMC process, but has since been adapted to work on several platforms and designs. The flow is primarily driven by GNU Make recipes and includes several configuration and parameterizations to target specific designs, technologies, and sites (IT environment).

Figure 4.1 shows the primary abstractions made in the architecture of the flow. Clear boundary abstractions are made within the flow to increase robustness and address the project goals.

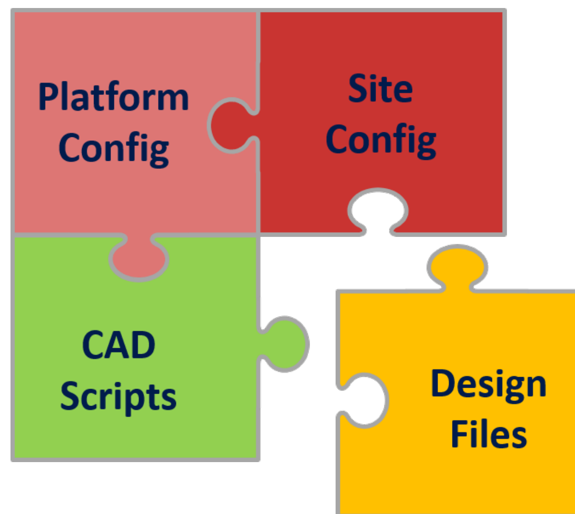


Figure 4.1: Cadre flow module abstractions

- **CAD scripts**: These are robust reference scripts for all the steps in the flow. The collection of scripts span several CAD vendors used throughout the steps. They contain ”best practices” and recommendations from the tool providers and have been stitched together to form a generic end-to-end flow that can easily be customized based on several factors.

- **Platform Configuration**: These constitute the PDK specific information, configurations, rules, and requirements for a specific technology node. There is a one-time effort to create this customization for newly supported technologies, however, subsequent designs can re-use that effort almost transparently

- **Site Configuration**: These are configuration parameters that establish pointers to the required site-specific information. These are file paths to the PDK, standard cells, tool binaries, and license information for the CAD tools.

- **Design Files**: This is the design-specific information that will contain significantly less information about the CAD tools, PDK and Site location since those have mostly been abstracted away. It will still contain a fair amount of designer intent which are expressed as customization to the generic flow in the CAD scripts.

Combined, these modular abstractions are be customized and combined together to form a chip-specific flow.

4.3 Goals

This section presents the objective goals Cadre flow set out to address and the way the challenges were addressed.

Design Agnostic Approach: Physical design can be particularly challenging because the design intent is very tightly coupled with the technology kit and especially when trying to achieve high QoR. We created a set of abstractions that encapsulates all of the common technology information and practices so they can be shared across designs. We also created several hooks and mechanisms so designers can fully overwrite and customize the scripts to express any needed intent to the tools.

Process Agnostic Approach: A big goal of the project was to be able to quickly retarget a design from one technology kit to another with minimal effort. This is accomplished using the Design Configuration abstraction which is selected by the design. The Design file is intended to have minimal information coupled with the technology it targets.

High Barrier to entry: We set out to create a flow that can easily be adopted by a developers not familiar with the intricacies and CAD tools relating to back-end development. Although a fair amount of explicit back-end design intent is required for any successful tapeout, the generic flow in the CAD tools provides a quick path to obtaining an initial implemented design. The flow can then be subsequently customized with explicit intent to improve QoR and package requirements. It allows for re-use of the CAD Scripts as well as the pre-configured design platform.

Tool/Vendor Agnostic Approach: The flow intentionally uses a mix of tools from different CAD vendors (the most reputable for each step of the flow) but can also be adapted to support multiple tools per step. Most major steps can be customized to use a competing tool for different CAD company (e.g. Swap from synthesis tool from vendor A to vendor B). Additionally, the scripts are written robustly to support non-strict versions of the CAD tools.

Site Agnostic: We set out to create a set of configurations that can be migrated from one

computing location or environment to another provided the site configuration can properly point to all the required file and tool dependencies.

Version Control: We ensure a full git-based version control mechanism for all the associated scripts, configurations, and design files.

Sharing: There are several NDAs and restrictions inherent to the tools, kits, and IP which prevents us from fully open sourcing the workflow. The workflow was modularized into several components and that can have separate access control privileges. This has allowed us to share certain components in a targeted way based on access to the process kit and or design IP.

Agile development: The flow is amenable to agile development and supports a hierarchical block-based structure for large designs.

4.4 Evaluation

The flow currently has support for 13 process development kits (PDKs) and has been validated with 6 different tape-outs across different 6 platforms. Figure 4.2 shows a timeline of all the SoCs that have been taped out using this approach. Table 4.1 also presents the list of supported technology platforms and their status. The flow has been shared with over 7 institutions and validated across 3 different premises.

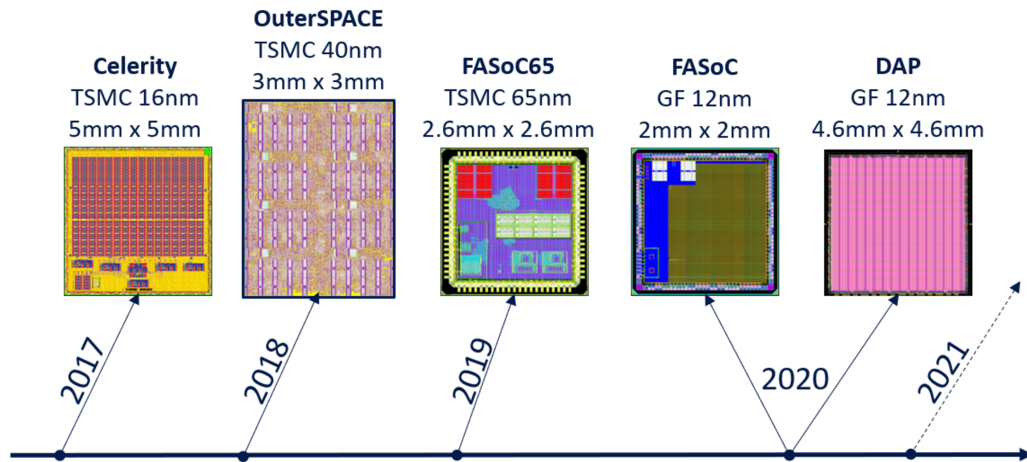


Figure 4.2: Timeline of SoCs taped out using Cadre Flow

It has been set up and used by several research groups at The University of Michigan in the fabrication of x chips. Outside of Michigan it has been set up and evaluated at Cornell University, University of Virginia, and the University of California - San Diego.

#	Platform	Description	Tapeout Validated
1	asap7	ASAP7: 7-nm Predictive PDK	
2	gf12lp	Global Foundries 12nm LP	Yes
3	gf14lpp	Global Foundries 14nm LPP	Yes
4	Gf14lppxl	Global Foundries 14nm LPPXL	Yes
5	tsmc16	TSMC 16nm FFC	Yes
6	gpdk45	GPDK 45nm Mixed Signal GPDK	
7	ibm45	IBM 45nm	Yes
8	freepdk45	Nangate45/FreePDK45	
9	fujitsu55	Fujitsu 55nm	
10	tsmc45	TSMC 45nm	Yes
11	tsmc65lp	TSMC 65nm LP	Yes
12	Skywater130	SkyWater 130nm	
13	gfbicmos8hp	Global Foundries SiGe 8HP	

Table 4.1: List of supported platforms in the Cadre Flow

Cadre flow has proved to be flexible enough to handle large complex designs like celerity and even mixed-signal designs generated by FASoC.

4.5 Conclusion

Hardware design has become more challenging with increased demand for performance and added complexity from techniques adopted to battle the slowdown of Moore's law. Creating CAD flow scripts specific to tools, technologies, platforms severely limit productivity and portability. To address these challenges, we created the Cadre flow methodology, a modular platform consisting of 1) proven CAD flow scripts that retain best-practices for a generic tape-out; 2) platform configuration scripts that customize the flow to any process node/kit; 3) Site configurations to adapt the flow to any computing environment; 4) Design specific files and customization that have most of the flow scripts and process specific information removed. The design files are able to re-use a lot of exiting work based on the process and can quickly re-target other nodes. Additionally, the design files can fully customize the chip flow to include any needed to customize the flow. The methodology proposed by the workflow allows for agile hardware development and quick iterations on chip design.

CHAPTER 5

Rapid Methodologies: OpenROAD

The OpenROAD (”Foundations and Realization of Open, Accessible Design”) project was launched in June 2018 within the DARPA IDEA program. OpenROAD aims to bring down the barriers of cost, expertise and unpredictability that currently block designers’ access to hardware implementation in advanced technologies. The project team is developing a fully autonomous, open-source tool chain for digital layout generation across die, package and board, with initial focus on the RTL-to-GDSII phase of system-on-chip design. Thus, OpenROAD holistically attacks the multiple facets of today’s design cost crisis: engineering resources, design tool licenses, project schedule, and risk.

The IDEA program targets no-human-in-loop (NHIL) design, with 24-hour turnaround time and eventual zero loss of power-performance-area (PPA) design quality. No humans means that tools must adapt and self-tune, and never get stuck: thus, machine intelligence must replace today’s human intelligence within the layout generation process. 24 hours means that problems must be aggressively decomposed into bite-sized problems for the design process to remain within the schedule constraint. Eventual zero loss of PPA quality requires parallel and distributed search to recoup the solution quality lost by problem decomposition.

Work presented in this chapter has been presented in 2019 Government Microcircuit Applications and Critical Technology Conference [40] and 2019 Design Automation Conference [41] and 2020 International Conference on Computer-Aided Design [42].

5.1 Introduction

Even as hardware design tools and methodologies have advanced over the past decades, the semiconductor industry has failed to control product design costs, as depicted in Figure 5.1. Today, barriers of cost, expertise, and unpredictability (risk) block designers’ access to hardware implementation in advanced technologies. Put another way: hardware system

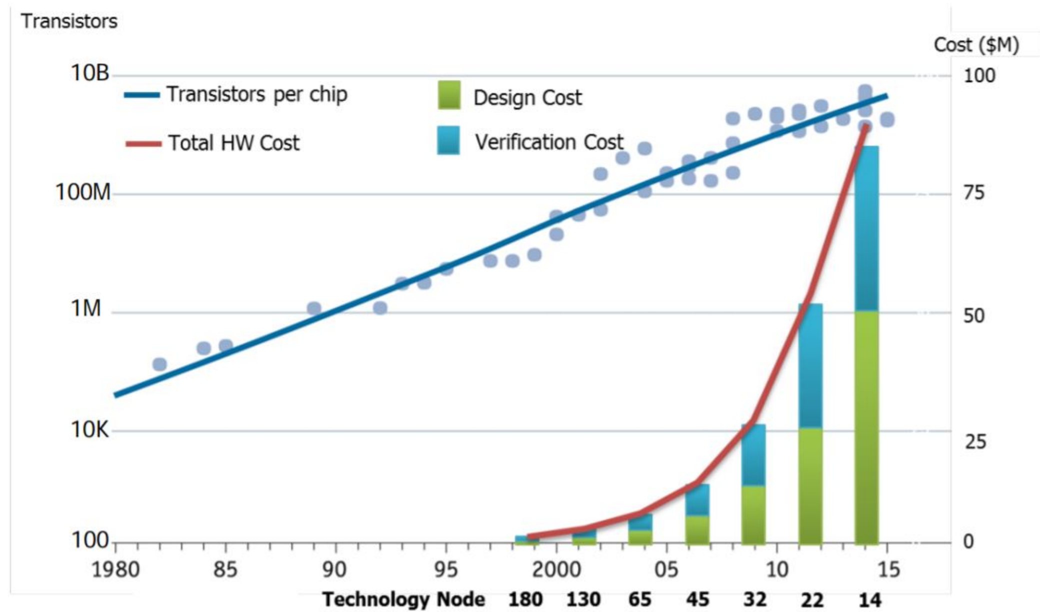


Figure 5.1: Design technology crisis.

innovation is stuck in a local minimum of (i) complex and expensive tools, (ii) a shortage of expert users capable of using these tools in advanced technologies, and (iii) enormous cost and risk barriers to even attempting hardware design.

Particularly in the digital integrated-circuit (IC) domain, layout automation has been integral to the design of huge, extremely complex products in advanced technology nodes. However, a shortfall of *design capability* – i.e., the ability to scale product quality concomitant with the scaling of underlying device and patterning technologies – has been apparent for over a decade in even the most advanced companies [43]. Thus, to meet product and schedule requirements, today's leading-edge system-on-chip (SoC) product companies must leverage specialization and divide-and-conquer across large teams of designers: each individual block of the design is handled by a separate subteam, and each designer has expertise in a specific facet of the design flow. DoD researchers and development teams do not have resources to execute such a strategy, and hence see typical hardware design cycles of 12-36 months.

5.1.1 IDEA and the OpenROAD Project

To overcome the above limitations and keep pace with the exponential increases in SoC complexity associated with Moore's Law, the DARPA IDEA program aims to develop a fully automated "no human in the loop" circuit layout generator that enables users with no electronic design expertise to complete physical design of electronic hardware. The

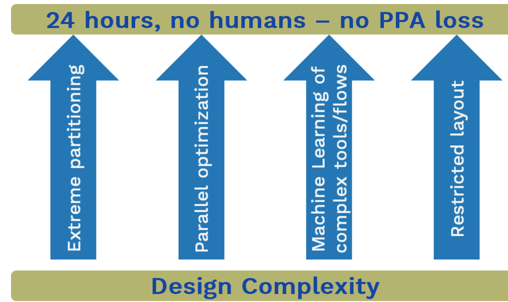


Figure 5.2: Design complexity.

OpenROAD ("Foundations and Realization of Open, Accessible Design") project [44] was launched in June 2018 as part of the DARPA IDEA program. OpenROAD's overarching goal is to bring down the barriers of cost, expertise and unpredictability that currently block system creators' access to hardware implementation in advanced technologies. With a team of performers that includes Qualcomm, Arm, and multiple universities led by UC San Diego, OpenROAD seeks to develop a **fully autonomous**, ***open-source*** tool chain for digital layout generation across die, package, and board, with initial focus on the RTL-to-GDSII phase of system-on-chip design. More specifically, we aim to deliver tapeout-capable tools in source code form, with permissive licensing, so as to seed a future "Linux of EDA" (i.e., *electronic design automation*).

Three innovative base technologies underlie the OpenROAD team's strategy to achieve no-human-in-loop (NHIL), 24-hour turnaround time (TAT). First, ***machine learning*** based modeling and prediction of tool and flow outcomes will enable the tool auto-tuning and design-adaptivity required for NHIL, new optimization cost functions in EDA tools, and new tool knobs that tools may expose to users. Second, ***extreme partitioning*** strategies for decomposition will enable thousands of tool copies running on cloud resources to maximize success within human, CPU, schedule bounds. Quality loss from decomposition is recovered with improved predictability of flow steps, along with stronger optimizations. Third, ***parallel/distributed search and optimization*** will leverage available compute resources (e.g., cloud) to maximize design outcomes within resource limits, and in the face of noise and chaos in the behavior of complex metaheuristics. A complementary precept is to reduce design and tool complexities through "freedoms from choice" in layout generation; this can increase predictability and avoid iterations in the design process. The synergy of base technologies and restrictions of the layout solution space is illustrated in Figure 5.2.

5.1.2 A New Paradigm

The contributions and approach of OpenROAD seek to establish a new paradigm for EDA tools, academic-industry collaboration, and academic research itself. OpenROAD aims to finally surmount ingrained, "cultural" and "critical mass / critical quality" barriers to establishing an open-source ethos in the EDA field. To start the project, we bring (i) significant initial software IP including donated source code bases, and a commercial static timing analysis tool; (ii) a significant set of academic software IP and skillsets; (iii) leading SoC and IP know-how and guidance from industry partners Qualcomm and Arm; (iv) an in-built Internal Design team (U. Michigan) to provide de facto product engineering and alpha customer functions; and (v) a broad agenda of industry and academic outreach. Furthermore, OpenROAD derives its "Base Technologies" efforts directly from the IDEA program requirements (no-humans, 24-hours, no loss of PPA quality). We view the cohesive integration of machine learning, problem partitioning and decomposition, and parallel/distributed search and optimization as essential to reaching the IDEA target.

The remainder of this paper will outline the current status of OpenROAD's GitHub-deployed tools and flow. Early proof points and calibrations in the realm of digital IC layout generation (RTL-to-GDSII) have been obtained in multiple foundry design enablements including 16nm FinFET technology.

5.2 Layout Tool Chain

OpenROAD's layout generation tool chain consists of a set of open-source tools that takes RTL Verilog, constraints (.sdc), liberty (.lib) and technology (.lef) files as input, and aims to generate tapeout-ready GDSII file. Figure 5.3 illustrates the flow of tools corresponding to individual OpenROAD tasks. These include logic synthesis (LS), floorplan (FP) and power delivery network (PDN) generation, placement, clock tree synthesis (CTS), routing and layout finishing.

5.2.1 Logic Synthesis

The major gap in open-source LS is timing awareness and optimization. OpenROAD has explored two avenues toward enablement of timing-driven synthesis. First, we use machine learning techniques to enable autonomous design space explorations for timing-driven logic optimization. It is often the case that synthesis scripts contain tens of commands in order to make a design meet its timing and area goals. These scripts are crafted by human experts. To produce best synthesis scripts that are tuned to individual circuits, we design machine

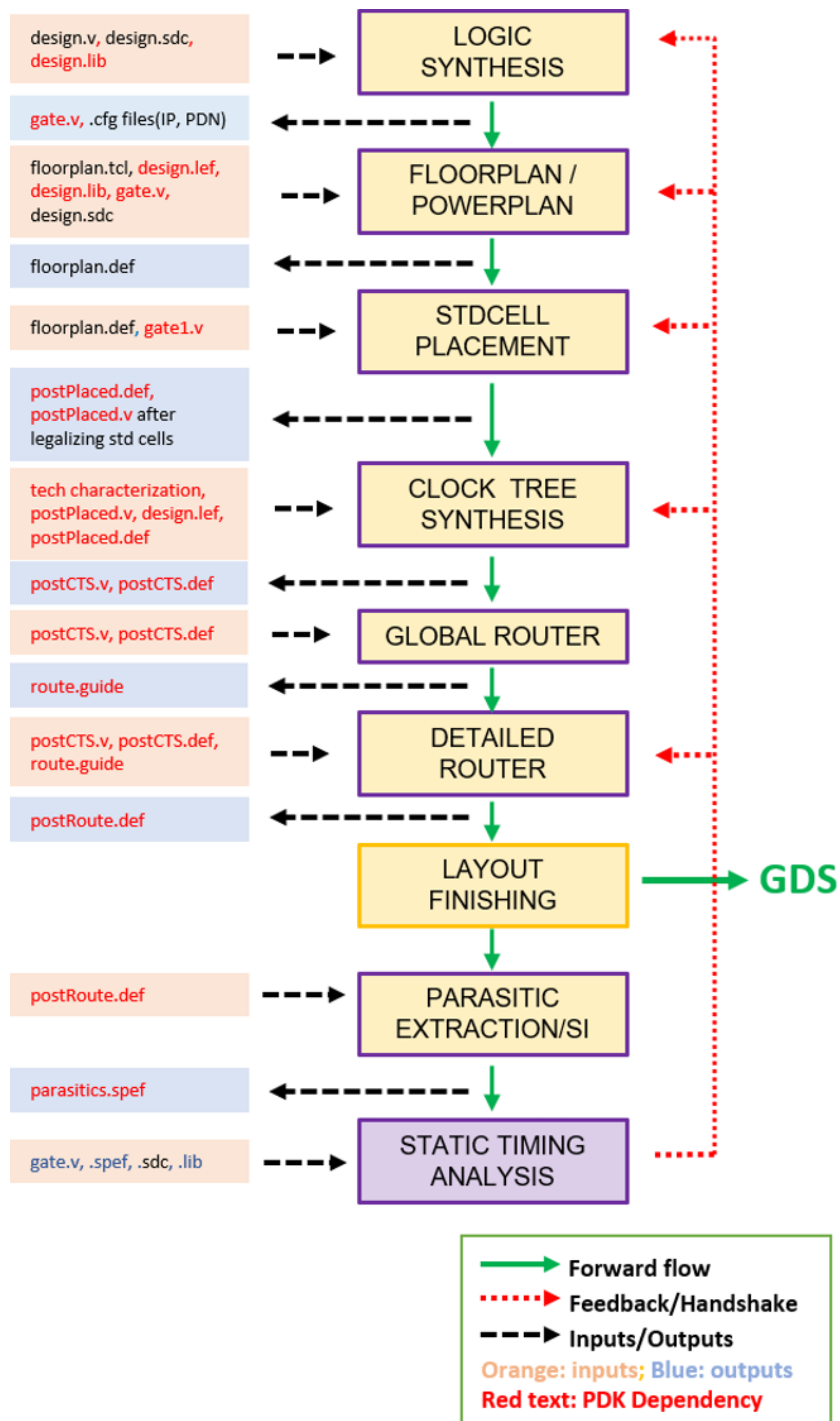


Figure 5.3: The OpenROAD flow.

learning agents that automatically generate step-by-step synthesis scripts to meet target timing and delay goals. Second, we enable physical-aware logic synthesis by integrating the RePlAce [45] placement tool into the logic synthesis flow, whereby global placement-based wire capacitance estimates are used within logic synthesis to improve timing results. Existing academic tools are oblivious to the outcomes of subsequent steps in the design flow, and our ultimate goal is to feed back wiring estimates as they are refined in physical design steps (e.g., standard-cell placement and global routing) to improve synthesis results.

5.2.2 Floorplan and PDN

Floorplanning and power delivery network synthesis are performed by TritonFPlan, which has two major components. The first component is integer programming-based macro block packing that is aware of macro-level connectivity and is seeded by a mixed-size (blocks and standard cells) global placement. The second component is Tcl-based power delivery network (PDN) generation following a safe-by-construction approach. TritonFPlan requires the user to specify several *config* files, e.g., IP_global.cfg and IP_local.cfg capture macro packing rules, and PDN.cfg captures safe-by-construction metal and via geometry information. These *config* files are necessitated by the inability of academic open-source tool developers (or, their tools) to see complete unencrypted design enablements from the foundry. We discuss this below in Section 5.4. The TritonFPlan tool uses mixed-size placer (RePlAce) for its initial global placement. The generated macro global locations provide a starting point from which multiple floorplan solutions are created. For each of the generated floorplan solutions with fixed macros and PDN, we use our placer (RePlAce) again, to determine the best floorplan according to an estimated total wirelength criterion. Limitations include support of only rectangular floorplans, and macro counts less than 100.

5.2.3 Placement

RePlAce [45, 46] is a BSD-licensed open-source analytical placer based on the electrostatics analogy. In OpenROAD, RePlAce is used for mixed-size (macros and cells) placement during floorplanning, for standard-cell placement within a given floorplan, and during clock tree synthesis (CTS) [47] for clock buffer legalization. Timing-driven placement is achieved with integration of FLUTE [48] and OpenSTA [49], along with a signal net reweighting iteration [50]. The timing-driven TD-RePlAce tool takes input in standard LEF/DEF, Verilog, SDC and Liberty formats, and incorporates a fast RC estimator for parasitics extraction. Ongoing efforts aim to enable routability-driven mode using commercial

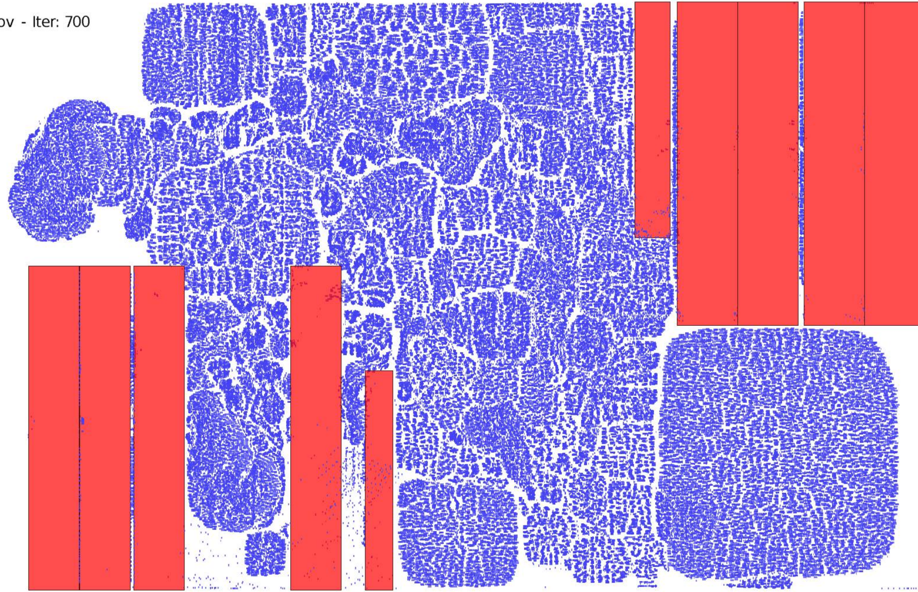


Figure 5.4: Foundry 16nm RISC-V based design block from the University of Michigan, after RePlAce mixed-size placement. Red color indicates macros and blue color indicates standard cells.

format (LEF/DEF/Verilog). Figure 5.4 shows the RePlAce placement of a small RISC-V based block (foundry 16nm technology) produced by the University of Michigan internal design advisors subteam.

5.2.4 Clock Tree Synthesis

TritonCTS [47, 51] performs clock tree synthesis (CTS) for low-power, low-skew and low-latency clock distribution, based on the GH-Tree (generalized H-Tree) paradigm of [51]. A dynamic programming algorithm finds a clock tree topology with minimum estimated power, consistent with given latency and skew targets. Linear programming is used to perform sink clustering and clock buffer placement. Leaf-level routing may be performed using either the single-trunk Steiner tree or the Prim-Dijkstra [52] algorithm.

In the layout generation flow, TritonCTS has interfaces with the placer (RePlAce) and the router (TritonRoute [53]). The placer is used for legalization of inserted clock buffers. The router maps sink pins to GCELLs that should be used for clock tree routing. TritonCTS inputs are LEF, placed DEF, placed gate-level Verilog, a configuration file and library characterization files. (For each foundry enablement, a one-time library characterization is needed. Currently, this library characterization is expected to be performed by some outside entity (foundry or tool user) using commercial EDA tools.) TritonCTS out-

puts are "buffered" placed DEF, "buffered" gate-level Verilog, and clock tree global routing in ISPD18 route guides format [54]. TritonCTS is publicly available on GitHub [47]. Early validations have been made using 16nm and 28nm foundry enablements. Improvements to handle multiple clock sources, non-default routing rules, etc. are ongoing.

5.2.5 Routing

TritonRoute [53] consumes LEF and placed DEF, then performs detailed routing for both signal nets and clock nets given a global routing solution in route guide format [54]. Prior to the detailed routing, TritonRoute preprocesses the global routing solution using a fast approximation algorithm [55] to ensure a Steiner tree structure for each net. Thus, congestion and wirelength are minimized while net connectivity is preserved in detailed routing stage. The detailed routing problem is then iteratively solved on a layer-by-layer basis, and each layer is partitioned into disjoint routing panels. The panel routing is formulated as a maximum weighted independent set (MWIS) problem and solved in parallel using a mixed integer linear programming (MILP)-based approach. The MWIS formulation optimally assigns tracks considering (i) intra- and inter-layer connectivity, (ii) wirelength and via minimization, and (iii) various design rules. By an alternating panel routing strategy with multiple iterations, inter-panel and inter-layer design rules are properly handled and track assignments are maximized. To date, TritonRoute supports major TSMC16 metal and cut spacing rules, i.e., LEF58_SPACING, LEF58_SPACINGTABLE and LEF58_CUTCLASS. An early evaluation shows approximately 10× reduction of spacing rule violations in a TSMC16 design block. Detailed routing flow with integration and optimization of local net routing is the next step towards a 100%-completion, DRC-clean layout capability.

5.3 Other Elements

Other elements of the OpenROAD project under development include the above-mentioned "base technologies" (machine learning, partitioning, parallel optimization), a design performance analysis backplane (parasitic extraction, static timing analysis, and power/signal integrity), cloud infrastructure for tool/flow deployment and machine learning, the "internal design advisors" task, and corresponding self-driving layout generation capability in the package and PCB domains. This section outlines the status of several of these project elements.

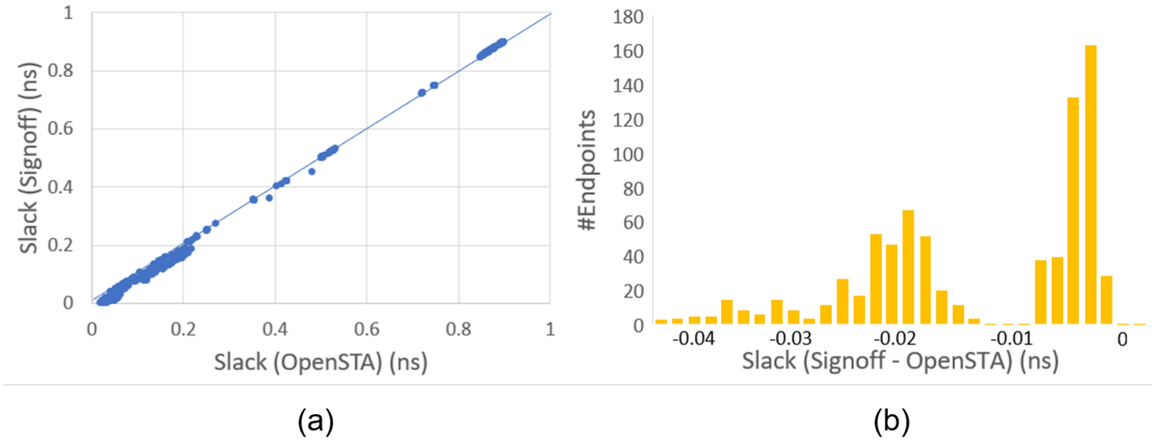


Figure 5.5: Comparison between OpenSTA and a leading signoff timer (Signoff) for a small 28nm testcase. (a) Endpoint slacks of OpenSTA vs. Signoff timer. (b) Histogram of endpoint slack differences between OpenSTA and Signoff timer.

5.3.1 Static Timing Analysis

OpenSTA [49] is a GPL3 open-sourced version of the commercial Parallax timer. The Parallax timing engine has been offered commercially for nearly two decades, and has been incorporated into over a dozen EDA and IC companies' timing analysis tools. OpenSTA is publicly available on GitHub [49] since September 2018. The developer, James Cherry, has added Arnoldi delay calculation, power reporting and other enhancements since the original release. OpenSTA has been confirmed to support multiple advanced foundry nodes, and it supports standard timing report styles. To date, the OpenSTA timer has been integrated into TD-RePlAce (timing-driven enhancement of RePlAce), physical-aware synthesis (Yosys [56]) and a gate-sizing tool (TritonSizer [57]). Figure 5.5(a) shows a comparison of endpoint timing slacks from OpenSTA and a commercial signoff timer. Figure 5.5(b) shows the distribution of endpoint slack differences between OpenSTA and the commercial signoff timer.

5.3.2 Parasitic Extraction

In OpenROAD's approach, the parasitic extraction (PEX) tool processes a foundry process design kit (PDK) to build linear regression models for wire resistance, ground capacitance, and coupling capacitances to wires on the same layer, or in the adjacent layers above and below. A basic use case is for another tool in the flow (e.g., CTS, global routing, timing analysis) to call PEX, providing an input DEF file that consists of the wire of interest and

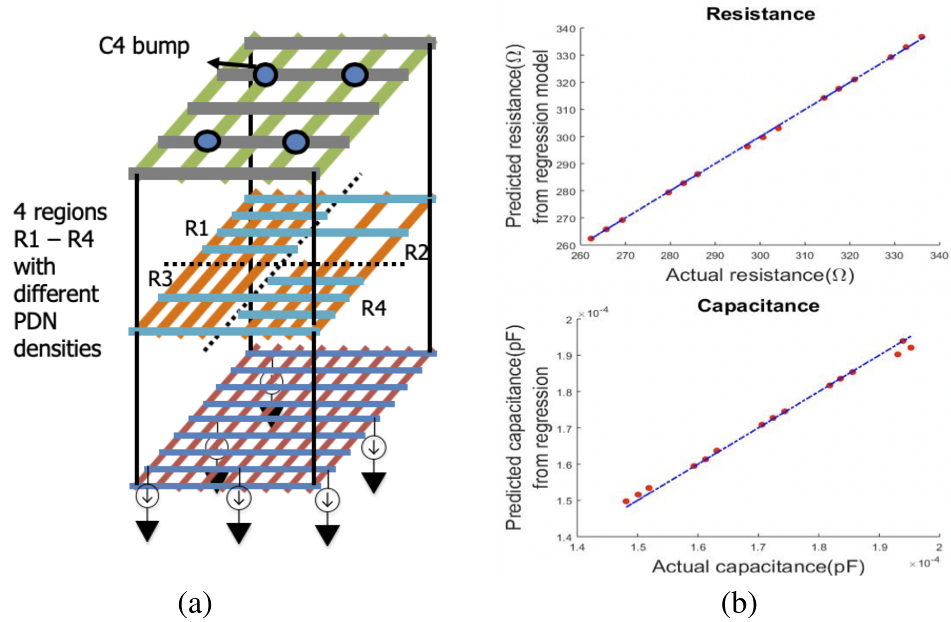
its neighbors. The output is provided as a SPEF file that contains the extracted parasitics. Figure 5.6(b) compares the actual and predicted values of the resistance and capacitance obtained from test nets to validate the regression model, and shows a good fit. Anticipated evolutions include interfacing the PEX functions to a possible future IDEA-wide physical design database, and extending the model-fitting approach to achieve low-overhead parasitic estimators for use in timing-driven placement, crosstalk estimation during global routing, etc.

5.3.3 Power Integrity

A key goal of our power integrity analysis effort is to enable single-pass, correct-and-safe-by-construction specification of the power delivery network (PDN) layout strategy across the SoC. Our power delivery network (PDN) synthesis tool tiles the chip area into regions, and selects one of a set of available PDN wiring templates (cf. the "config" files noted in the Floorplanning discussion, above) in each region. These templates are stitchable so that they obey all design rules when abutted. The PDN tool takes in a set of predefined templates (Figure 5.6(a)), an early (floorplanning-stage) placed DEF for a design, and available power analysis information (e.g., our OpenSTA tool can provide instance-based power reporting). A trained ML model then determines a safe template in each region. An early prototype shows that the ML-based approach can successfully deliver a PDN to satisfy a given (e.g., 1mV static) IR drop specification.

5.3.4 Cloud Infrastructure

For users to take advantage of OpenROAD tools as well as tools developed by other collaborators, a cloud infrastructure effort aims to provide an end-to-end seamless user experience. In our cloud deployment, users subscribe their Git repo to our cloud system. Once a design change is pushed to the Git repo, the design is automatically compiled by the OpenROAD flow and the user receives a notification by email when the flow is complete. The user can then download the outcome files through a web browser. If needed, the user can also monitor the progress of the flow on our web-based front end. Our cloud deployment is elastic as it leverages more computing resources when more users log into the service, or when a user requests parallel processing capabilities. For instance, the service can elastically deploy multiple machines in order to run a tool (e.g., placer) with multiple random seeds to obtain a better result within a given wall time budget. Or, in conjunction with global design partitioning, the cloud deployment can run each design partition in parallel on a cloud instance, to maximize parallel speedup and minimize design turnaround time.



(a) (b)

Figure 5.6: (a) Example PDN templates; and (b) validation of the regression model for R, C in PEX.

5.3.5 METRICS 2.0

To enable large-scale applications of machine learning (ML) and ultimately a self-driving OpenROAD flow, we are developing METRICS 2.0 [58], which can serve as a unified, comprehensive design data collection and storage infrastructure (see [59]). A METRICS 2.0 dictionary provides a standardized list of metrics suitable for collection during tool/flow execution, to capture key design parameters as well as outcomes from various tools in the design flow. We also propose a system architecture based on JavaScript Object Notation (JSON) for data logging, and MongoDB database [60] for data storage and retrieval of the metrics. Figure 5.7 illustrates the METRICS 2.0 system architecture. The proposed architecture eliminates the need to create database schemas and enables seamless data collection. METRICS 2.0 is tightly coupled with machine-learning frameworks such as TensorFlow, which provides easy interfaces to read and write into MongoDB, and enables fast deployment of machine learning algorithms.

5.3.6 Early SoC Planning

In light of NHIL and 24-hour turnaround time requirements, it is important to initiate the OpenROAD tool chain with with reliable tentative floorplans as flow starting points, to min-

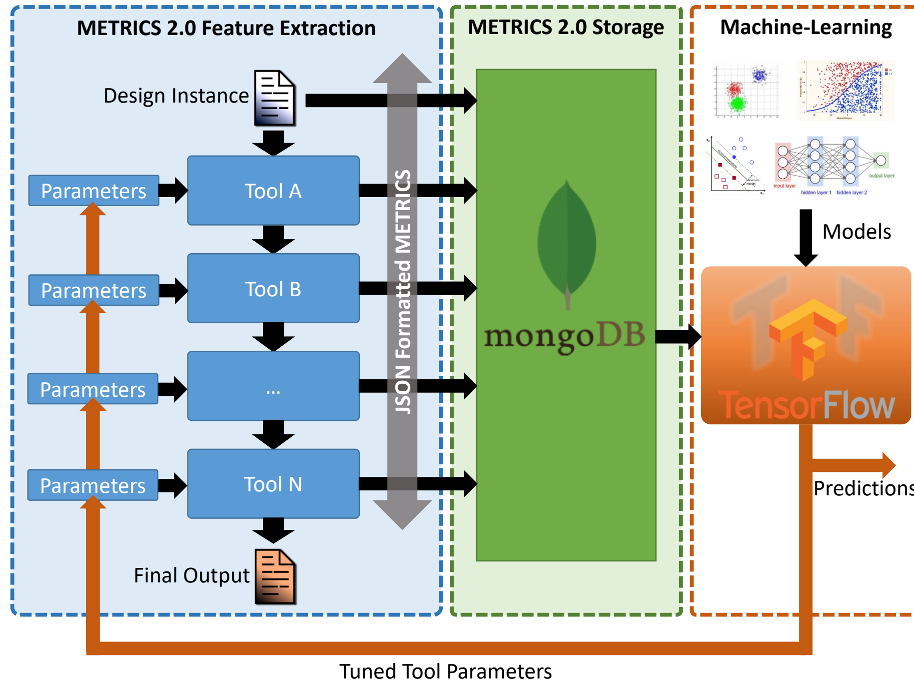


Figure 5.7: Overall METRICS 2.0 system architecture.

imize the likelihood of run failures. This is a key link between the "system-level design" (IDEA TA-2) and "layout generation" (IDEA TA-1, which we address in OpenROAD). Early floorplan estimates for the SoC can be enhanced by embedding physical implementation information in each IP (e.g., using the vendor extension mechanism within industry-standard IP-XACT descriptions), and by making use of technology- and tool chain- specific parameters and statistical models. Combining and elaborating such information enables early area and performance estimates that can indicate doomed-to-fail floorplan candidates or suggest design implementation fine-tuning (hard-macro placement, grouping, register slice insertions, etc.) in viable floorplans.

5.3.7 Integration and Testing

The individual tools described above comprise a tool chain that produces an implemented design ready for final verification and fabrication. Initial platform support is targeted for CentOS 6, with tool- and flow-specific support maintained at [44]. To evaluate the flow, non-tool developer entities in our team (i.e., U. Michigan, Qualcomm and Arm) perform fine-grained analyses on our tool outputs and provide target calibration metrics for tool developers. Here, we leverage a testcase suite based around existing designs that have

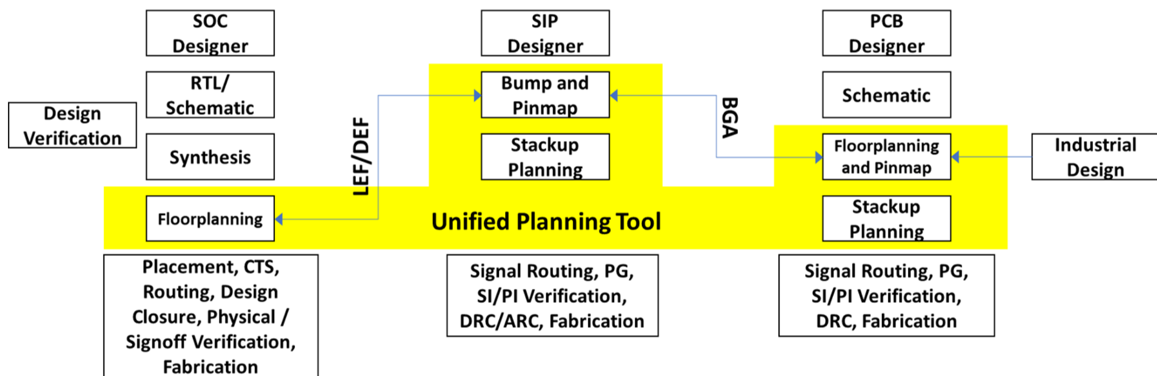


Figure 5.8: Illustration of Unified Planning Tool.

previously been taped out; these designs range across complexity (from small blocks to whole chips) and process (e.g., 16nm and 65nm). Our suite of testcases also includes cutting-edge complex SoCs that are currently in development. A continuous integration test suite validates the tools individually during development and tracks regression metrics and feature impact.

5.4 Looking Forward

Our near-term efforts will continue development of the tools and flow described above. More broadly, we will also seek to address various technical, structural and cultural challenges that have become apparent even at project outset.

One key technical challenge is to develop design automation technologies as well as layout generation flows that can co-optimize across the SoC, package (PKG) and PCB domains. Today, SoC, PKG and PCB tools and flows are largely disjoint; weeks if not months are required to converge across the three designs with manual iterations. To deliver NHIL, 24-hour turnaround time in the PKG and PCB domains, a Unified Planning Tool that seamlessly coordinates among the three databases and enables quick iterations is essential. Figure 5.8 illustrates our envisioned Unified Planning Tool. The Unified Planning Tool would also include optimization engines, using analytical and ML approaches to evaluate the complex tradeoffs across the three design spaces.

Some other technical challenges include the following. (1) The "small and expensive" nature of design process data in IC design – where obtaining a single data point might require three weeks to run through a tool flow – challenges machine learning and devel-

opment of “intelligent” tools and flows. (2) The need for new, common standards for measuring and modeling of hardware designs and design tools must be compatible with the IP stances of foundries and commercial EDA; this may shape the future opening of a “Linux of EDA” to broad participation. And, (3) it will be difficult to illuminate the critical junctures where “human intelligence” is now required, yet must be replaced by “machine intelligence”, in the hardware design process.

Several structural challenges stem from our status as academic tool developers of a tool chain that must produce tapeout-ready GDSII. (1) OpenROAD tools will likely not be foundry-qualified, which implies that OpenROAD tools and tool developers will not be able to read encrypted advanced-node PDKs. To achieve safety and correctness by construction of the tapeout database, OpenROAD tools require *config* files and one-time generation of “OpenROAD kit” elements, for each foundry enablement. (2) OpenROAD’s analyses and estimators for timing, parasitics and power/signal integrity are not “signoff” verifiers. Thus, additional performance guardbands are required throughout the layout generation flow. And, post-OpenROAD verifications may be performed by designers and/or foundries. (3) OpenROAD tools are developed and released by non-commercial entities. Commercial EDA vendors receive bug/enhancement requests accompanied by a testcase that exhibits the bug or behavior at issue. By contrast, bug reports that we receive are unlikely to be accompanied by testcases due to blocking NDA / IP restrictions. This complicates the bug-fixing and enhancement process.

Finally, our outreach efforts seek culture change and engagement across the community of potential developers and tool users. For example, in the academic research world, a lab’s code is its competitive advantage (or, potential startup), and liberal open-sourcing is still rare (cf. [61]). We hope that OpenROAD and the IDEA/POSH programs help drive culture change in this regard. With regard to tool users, we observe that commercial EDA tools are invariably driven to production-worthiness by “power users” – i.e., paying customers who have deep vested interests in the capability and maturation of a given tool. Traditionally, power users expose a new tool to leading-edge challenges and actively drive tool improvement. For OpenROAD, finding our “power users” is a critical need, especially since they would be able to improve tools and flows at the source-code level.

5.5 Conclusion

In this paper, we have reviewed the scope and status of OpenROAD, a DARPA IDEA project that aims to develop a self-driving, open-source digital layout implementation tool chain. The above is only a snapshot, taken six months into a four-year project.

CHAPTER 6

Rapid Frameworks: ACAI

In heterogeneous systems, various computational units such as GPUs, co-processors and other hardware accelerators (HAs) work alongside general purpose cores in order to improve the performance, power and/or energy metrics. With Dennard's scaling at an end, the pervasiveness of fixed-function HAs on modern SoCs have increased, as system architects strive to maximize efficiency growths on embedded and high-performance platforms. Integrating HAs into the system introduces additional challenges such as: 1) increased complexity in programming and data models; 2) increased attack surface; 3) increased hardware integration and verification costs; and 4) fair scheduling of jobs from multiple processes to HA resources.

To address these problems, we present *ACAI*, a hardware and software framework for HA integration. It provides the HA with a shared view of system memory, shared virtual addressing and data caching with full hardware coherency. ACAI simplifies the software programming experience, reduces integration effort and scheduling of jobs. It is prototyped on real hardware using the Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation board and user space applications running in Linux. We explore the benefits of full cache coherency on an optimized implementation of ACAI across several kernels and alternate interface options.

Our results show average read latency improvements of 12.1x and 5.7x when compared to non-coherent and IO-coherent interfaces respectively. ACAI allows for HAs to sustain higher data bandwidth for sequential bulk-data copy and vastly outperforms other interfaces for small data sizes, random data access and link-list traversal. We also demonstrate an average performance improvement of 4.3x across Machsuite [62] kernels and 1.4x across several hardware applications from Rosetta [63].

Work presented in this chapter has been presented in at 2018 Design Automation Conference - IP Track [64] and 2019 Arm Research Summit [65].

6.1 Introduction

Dennard's scaling is at an end [66], and transistor density scaling is wavering at more advanced technology nodes. In order to deliver the traditional performance and power improvements required by future workloads, computer architects are turning towards heterogeneous architectures comprised of more specialized compute units. These specialized units or hardware accelerators (HAs) improve on the system's performance, power and/or energy metrics across a wide variety of workloads and applications. While the GPU serves as a classical example of an accelerator, there has been an explosion in the variety of HAs designed for embedded and high performance computing platforms.

There is growing adoption of fixed-function HAs for computationally intensive kernels in approximate computing [67], neural computing [68–70], graph processing [71, 72], and database applications [73, 74]. Prior works [75–77] have proposed mechanisms to extract kernel regions from existing software programs for offload to HAs. Adding to the assortment of accelerator workloads, are techniques employed in creating HAs. In addition to hand crafted RTL, high levels synthesis (HLS) and domain specific languages (DSL) are growing in popularity [78–80]. The ubiquity of HAs drives the need for robust system architectures and accelerator interfaces capable of handling diverse memory demands and irregular access patterns [81].

Integrating HAs into existing system architectures can be effort intensive and it introduces additional challenges relating to data management. Loosely coupled accelerators (separated from the processor pipeline) can be attached at different levels of the memory system hierarchy. Integration over an I/O bus requires expensive data movement to accelerator buffers using specialized drivers. Other techniques employ a shared memory space and offer varying levels of data coherency and virtual addressing. Factors within specific SoC platforms - such as coherency protocols and interconnects - drastically affect the usability and performance of the attached HA. Even with an intimate understanding of the SoC architecture and HA, the various programming considerations make it challenging to quickly develop software applications in a manner that allows for maximum utilization of the HA. The proliferation of HAs on modern SoCs also increases the attack surface, presenting additional security, isolation, and protection concerns. All of these challenges negatively impact the development effort (in software and hardware), non-recurring engineering (NRE) cost, validation time and time-to-market.

In this work, we present ACAI (Figure 6.1), a novel hardware and software framework for integrating HAs into heterogeneous SoC architectures. ACAI is designed to ease HA integration and deliver improved performance for a wide variety of accelerator workloads.

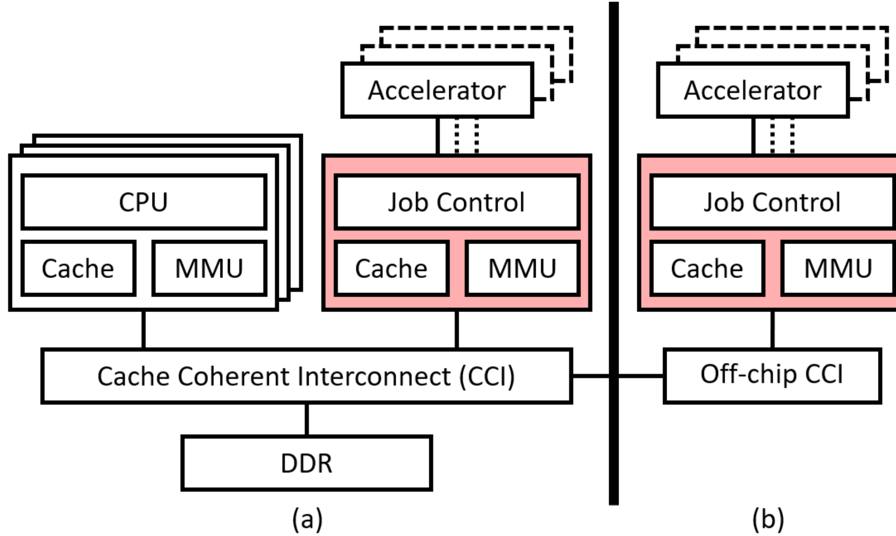


Figure 6.1: Example SoC platform showing ACAI hardware interface with the integration of (a) an on-chip HA and (b) an off-chip HA

The hardware interface provides accelerators with a unified view of system memory, a virtual addressing context that is shared with the user space application, data caching and full data coherency. Paired with kernel drivers and software libraries, it serves as a configurable, robust and scalable framework for integrating HAs with reduced design effort. ACAI also enables fine-grained kernel acceleration and job scheduling between multiple processes and accelerator resources. For our evaluation, we developed a hardware prototype using the Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation board and user space applications running in Linux. We also implemented and explored two alternate HA interfaces for performance comparisons: a shared non-coherent (Shared-NC) and a shared IO coherent (Shared-IO) option.

We present quantitative analysis on the different interfaces and show average read latency improvements of 12.1x and 5.7x when compared to Shared-NC and Shared-IO respectively. In our optimized implementation, ACAI allows for the HA to sustain improved data bandwith for sequential bulk-data copy when compared to the alternate interfaces and vastly outperforms them for fine-grained acceleration, random data access and link-list traversal. We also demonstrate an average performance improvement of 4.3x across several accelerated kernels from the Machsuite benchmarks [62] and 1.4x across several hardware applications from Rosetta benchmark suite [63].

Paper Outline: Following this introduction, Section 2 presents background information for shared memory, virtual addressing and caching as it relates to HAs. It also explores different accelerator implementations and memory demands. Section 3 describes the goals

of our framework, introduces the baseline hardware design and walks through the mechanism for dispatching jobs to HAs. Section 4 details the methodology and the HA workloads we used in our evaluation. In section 5, we present and analyze performance results from our experiments. Finally, we discuss future work, related work and present conclusions in the sections that follow.

6.2 Background

Fixed-function HAs typically improve performance by exploiting instruction level parallelism (ILP) and memory level parallelism (MLP) present in kernel routines. They are often implemented using complex functional units and take advantage of pipelining techniques. Accelerator performance is often limited by the ability of the system to adequately satisfy its memory demands.

Shared Memory: Early HA implementations had a separate memory space (or scratchpads) which relied on specialized software and drivers to move data between the CPU and scratchpad over dedicated I/O buses (e.g. PCIe). GPUs and GPGPUs are commonly implemented this way because they have high memory bandwidth demands and the costly offload overheads can be amortized over large data computations. To ease the offload burden and improve programmability, more HA implementations have adopted a view of memory shared with the CPU. This shared view reduces data movement and eliminates data copies between buffers. Hardware coherence further simplifies the software stack by eliminating the need to perform explicit software synchronization (cache flushes and invalidations) for data accessed by the HA. Prior research in academia [82–84] and products from industry [85, 86] demonstrate promising results for shared memory implementations.

Virtual Addressing: We find that shared virtual addressing is another key to ease programming by allowing the HA to access memory in the same context as the application process. The HA will be able to operate independently on application memory without special buffers, kernel-level data copies, or pinned pages. Large regions of memory can be allocated without the need of having a physically contiguous address space. Virtual addressing also enables the HA to take advantage of pointer semantics and dereference dynamic software data structures. Overall system security is improved by eliminating the HA's ability to access memory using physical addresses. This isolates the HA, preventing it from accessing data belonging to the kernel or other processes. Virtual addressing increases interoperability between the accelerator, CPU and other memory masters in the system.

Virtual addressing does come with some pitfalls. System performance may be degraded

due to overheads from address translation and page table management. The overheads are strongly tied to the address translation implementation, coherence protocol and the memory access pattern. These effects have been extensively studied and prior works have described techniques for significantly reducing page-translation overheads for accelerators [87–89].

Caching: HAs can benefit from caches in the same way CPUs do. Caches can reduce memory latency by exploiting spatial and temporal locality in the memory access pattern. This can yield significant performance improvements for HAs that are sensitive to memory latency. IO coherency (one-way coherence without cache support) can reduce the HA's read latency by *snooping* data from neighboring caches, instead of accessing the DDR controller. Full cache coherency extends this by allowing cache-line migration (reducing invalidation/flushes from neighboring caches) and the transfer of results back to the CPU cache. A reduction in overall latency enables fine-grained acceleration since the overheads for job dispatch are substantially reduced. Caches can also improve performance when sharing data between accelerators behind the same cache hierarchy [90].

Bandwidth vs Latency: The performance of conventional accelerators is typically limited by memory bandwidth, however as workloads become more diverse, memory latency and data synchronization are beginning to play a bigger role [81, 91–95]. Many applications in data analytics and machine learning rely on very large data structures that use pointers and exhibit irregular memory patterns. These structures include linked-lists, trees, hashes and graphs. HAs that operate on software data structures are more likely to be latency sensitive since the data structures may need to be traversed. In order to work around the problem, programmers often manipulate the data in software before dispatching to the HA. This manipulation, which occurs before and after hardware dispatch, will negatively impact the application performance and programming experience. We find it important to deliver a platform that caters to both types of memory demands.

HA Implementations: In our research, we studied and categorized various HA implementations based on the data usage model. HAs that use a *Buffered* data model divide the kernel execution into three phases: data load, compute and data unload. The phases may be explicit or pipelined. These types of HAs require large scratchpad buffers and often have separate DMA engines to perform data transfer. The performance of these accelerators are often handicapped by bandwidth, especially for accelerators with shorter compute phases. HAs that use an *unbuffered* data model fetch data on demand, interleaving memory access with computation. These applications typically work on larger data sets and may expose data locality.

Real-world applications often require several invocations of an accelerator or even continuous activations for streaming data. For example, many applications in signal processing

require successive invocations of different HAs to form a data pipeline [96]. We again categorized HAs by their execution model. In *Burst* execution models, the HA must complete the current job before starting a subsequent one. In *Pipelined* execution models, the HA can begin working on the next job before the current job has completed.

The HAs studied in work are capable of mastering the memory bus; they can issue read/write memory requests and tolerate variable memory latency. In addition to the kernel algorithm, system architects tune their data and execution models to match the memory system so as to maximize utilization of the HA.

6.3 ACAI Architecture

The ACAI framework is comprised of hardware and software components that simplify the integration of HAs and the offloading of jobs. The hardware is an interface that configures HA resources and provides them with access to shared memory. It provides virtual address translation and a coherent data cache. The software components - kernel driver and software library - ease programmability, configuration, scheduling and dispatch of jobs to the HAs managed by the interface. In designing ACAI, we had the following goals in mind:
Programmability: ACAI adopts virtual addressing for Accelerators. The framework provides the HA with a shared view of virtual memory, as seen by the CPU application process. Software developers can use familiar software data structures without needing to manipulate data. There are no special buffer allocations or data copies between user and kernel spaces. The design transparently supports application processes running on hypervisors (a growing trend in data-centers) since there no physical addresses are involved. The HA is also able to perform full pointer dereferencing. ACAI also introduces the notion of a *job chain*, a linked-list of jobs that are intended to be executed serially. This feature accommodates applications needing to schedule back-to-back executions of accelerated kernels.
Scalability/Robustness: The ACAI interface serves to abstract coherence protocols from HAs. It supports different system topologies and various arrangements of CPUs, caches, HAs, memory controllers, interconnects and other nodes in the design. It also enables HA nodes to participate in off-chip coherence. The interface supports multiple data lanes and lane sizes to increase throughput and support multiple HA resources. Figure 6.1-b shows an example of a more complex system topology that supports an additional set of accelerators implemented off-chip.
Performance: Today, most HAs have buffered implementations causing application performance to be proportional to bandwidth for memory intensive kernels. For this reason, maximizing data bandwidth was a key goal during the design and implementation of ACAI.

ACAI adopts full cache coherency to reduce the job offload costs. The framework also aims to improve performance on workloads that are more sensitive to latency by enabling cache-to-cache data transfer and the ability to exploit locality.

Security: ACAI preserves full memory protection semantics and isolation of the virtual memory system to prevent malicious or erroneous access to data. The HA will be restricted to regions of memory (or pages) specific to the application context from which it is being invoked. If memory access occurs outside the permitted region or without the correct attributes, the OS can be notified to respond appropriately.

Design Effort: In addition to easier programmability, the hardware simplifies HA development by leveraging standard coherence protocols from the *ARM AMBA 4* protocol specifications [97]. This allows the HA to be designed and tested in isolation, generated from HLS tools or obtained from third-party vendors. This in turn reduces NRE costs and the time-to-market. The framework can be emulated on commercially available boards allowing accelerator designers to iterate across different implementations to further increase accelerator utilization.

6.3.1 Base Hardware

At a high level, ACAI's hardware micro-architecture is similar to a processor, containing recognizable functional units. The basic components are shown in figure 6.2 and are described in this subsection.

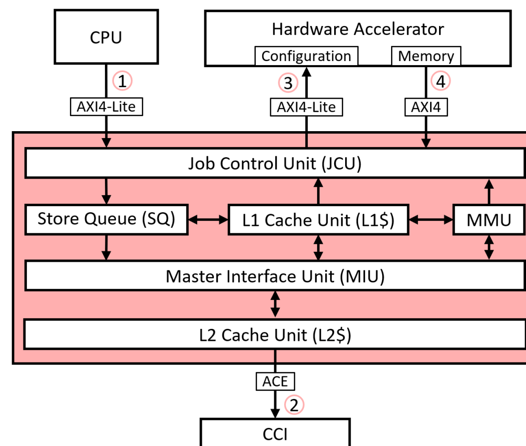


Figure 6.2: ACAI hardware interface design

L2 Cache Unit (L2$) connects the hardware (and HA) to the cache coherent interconnect (CCI) using the CCI slave interface②. The data states are mapped directly to the MOESI cache coherency model. It has an integrated snoop controller that responds to broadcasted

snoop requests and *distributed virtual memory* (DVM) synchronization messages from the CCI. The L2 cache is *physically indexed, physically tagged* (PIPT), has a configurable cache size and is fully inclusive of the L1 cache for coherency support.

Master Interface Unit (MIU) is responsible for decoupling and aggregating requests to the CCI (through the L2$) from the SQ, L1$ and MMU. The MIU contains buffers, tracks outstanding requests upstream, performs significant hazard checking and coordinates cache line fills/evictions.

L1 Cache Unit (L1$) interfaces with the HA (through the job control unit) and the MIU. The L1$ is *physically indexed, physically tagged* (PIPT) and has a configurable cache size. Like a CPU's cache, the L1$ provides the accelerator with extremely low latency for access patterns that have spatial or temporal locality.

Store Queue (SQ) holds pending store requests from the HA. The SQ can access data in the L1$, initiate linefills through the MIU, or write the data out to the system through the MIU. The SQ can merge several store transactions into a single transaction if they are aligned and is also capable for merging multiple writes into a write burst. Store coalescing close to the accelerator significantly improves write performance.

MMU performs pagewalks to translate virtual addresses into physical addresses, and caches the results of those translations in the TLB for future use. The TLB can accommodate 256 entries. The MMU follows the *Virtual Memory System Architecture* (VMSA) specification [98] which describes address translation, access permissions and privileges for the ARMv7 architecture.

Job Control Unit

communicates directly with the software driver through the *CPU peripheral interface*① and is responsible for fetching, scheduling and launching jobs on the HA. The job control unit has two interfaces to the HA: The *HA configuration interface*③ configures the HA with information obtained from the job description before execution. The *HA memory interface*④ is mastered by the accelerator for access to shared memory. The job control unit also performs fair arbitration of job requests from multiple processes to multiple HA resources.

6.3.2 ACAI Operation

This subsections describes the software routines and the procedures by which compute kernels are offloaded to HAs integrated using the framework. A *job* is a well-defined piece of work which can be computed in software or can be offloaded to the HA to improve efficiency. The computation to be performed by the HA is specific to the accelerator rou-

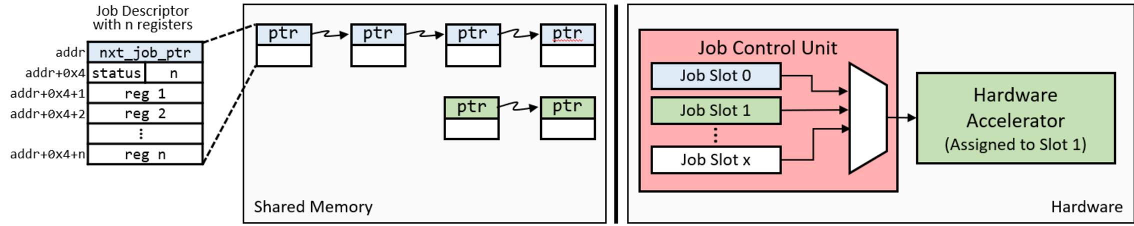


Figure 6.3: An example framework state showing job descriptors linked together to form two job chains in shared memory and the job control unit performing arbitration and execution in hardware

tine/kernel as well as the associated input and output data. A *job descriptor* is a data structure that stores the job related information and is required for configuring and invoking the HA in the framework. It primarily contains a number of job specific register values (*or descriptor values*) that will be written to the accelerator before launch. A *job chain* is a linked list containing one or more job descriptors scheduled for sequential execution by the HA resource. A *job slot* is a set of resources within the job control unit used to store and manage the execution of a job chain. The job slot keeps track of the address space information associated with the context of an assigned job chain. Each job slot has buffers used to store job descriptor values required for dispatch. The hardware can be configured to have an arbitrary amount of job slots to co-ordinate requests from multiple processes.

Figure 6.3 (left) depicts the structure of the job descriptor. It also shows an example framework state with two job chains - created by different processes - scheduled for execution on a single HA resource. The following steps walk through the process of creating and launching jobs using the ACAI framework:

1. Application Initialization [CPU] Each application process has its own unique address space information (PID, base register, etc) stored in memory. The application shares the address space information with the HA using the initialization routines provided by the kernel driver. The driver reserves a job slot within the job control unit and sets the translation base address. This configuration is performed over the CPU peripheral interface①.

2. Job Offload [CPU] As the process execution proceeds, the data structures to be operated on by the HA are allocated and initialized. Since the HA will be able to access all user mode data (static and dynamic), there is no need to manipulate the data or move it across special buffers. The application begins the offload process by creating a chain of job descriptors. This is a simple linked list stored in the application's memory. The job is offloaded, with the help of the driver, by writing the base address of the first job in the chain to the already reserved job slot. Hardware coherence, slot reservation and single-write dispatch avoids several data race conditions associated with accelerator dispatch.

3. HA Configuration and Launch [Hardware] The job slot becomes active when the base address for the first job in the chain is written. Once the job control unit completes arbitration (from active slots to available HA resources), the MMU is configured for the slot's context and the job descriptor is copied from main memory into buffers in the job slot. The job descriptor values are read from the CCI interface② and is immediately written out over the HA configuration interface③. The job control unit will invoke the execution of the HA once the required number of descriptor values have been written. The job control unit pipelines the chain execution by fetching the next linked job descriptor ahead of execution.
4. Job Completion [Hardware] When the HA signals completion, the job control unit updates the status bits in the job descriptor and begins executing on the next job in the chain. The chain execution completes when the status of the last job descriptor is marked complete. There are several supported mechanisms for the application process to determine when a job or chain is complete:

1. The application can poll the status of individual jobs or the last job in the chain to determine completion state. The polling traffic will be filtered in the CPU cache since the job chain is stored in a hardware coherent cache line that will be invalidated by the interface on completion.

2. The application can poll the status of the job slot status using the driver.

3. The application can wait for an interrupt triggered by the interface on the completion of the job chain.

4. The application can wait for an event (WFE) [98] triggered by the interface on the completion of the job chain. This is preferred over interrupts, which have longer latencies due to the necessary interrupt service routines. Interrupt latency may significantly affect performance for fine-grained acceleration.

5. Retrieving results [CPU] After the job(s) are completed, the application process can access the resulting data directly since the framework relies on hardware coherence. The slot reservation can also be relinquished at this time.

An example program that invokes a simple data copy accelerator is presented in figure 6.4.

6.3.3 Advanced Scheduling

Data Reuse: Working data from job executions remain in the interface cache through the launch of subsequent jobs in the chain. The application may realize further performance

```
#include "acai.h"
#define N 1024
void main() {
  // 1. initialize application data
  int src[N], dst[N];
  for (i = 0; i < N; i++)
    src[i] = rand();
  memset(dst, 0, sizeof(dst))

  // initialize acai framework and reserve job slot
  acai *p_acai = new acai();
  p_acai->init();

  // 2. setup job chain with a single job
  vector<acai_jd> job_chain;
  job_chain.reserve(1);

  // set descriptor with 2 values: src and dst address
  job_chain.push_back(acai_jd(3, 0));
  job_chain[0][0] = src;
  job_chain[0][1] = dst;

  // 3/4. start and wait for job to complete
  p_acai->start_job(job_chain);
  p_acai->wait_job(job_chain);

  // 5. cpu reads results
  for (i = 0; i < N; i++)
    printf("dst[%d]: %d\n", i, dst[i]);
};
```

Figure 6.4: Example application using framework

gains by reducing data movement in the system. This is accomplished through the careful construction of job chains that reuse data across jobs. Reuse may take place between invocations of the same kernel with partially new data or between different kernels that access the same data (e.g. signal processing pipeline).

Multiple Processes: The framework performs fair arbitration and job scheduling from an arbitrarily configured number of job slots. Applications processes running in the CPU can reserve as many jobs slots as are implemented in hardware. The execution will be performed in a round-robin fashion, either on a job or job chain granularity.

Multiple Accelerators: Currently, ACAI is only able to perform arbitration granting a process access to all the available HA resources. Although the HA resources can have multiple

personalities (implement different kernels), only a single job slot can can be scheduled for execution. Future extensions of the design will support multiple executing across the available HA resources. This will require updates to the job control unit and an MMU that can service multiple contexts simultaneously. It will also lead to cache interactions between the active HA resources.

6.4 Methodology

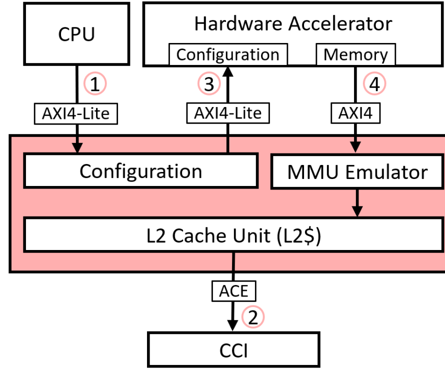
The system architecture is prototyped on the Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation board. Central to the board is a programmable MPSoC device with a quad-core ARM Cortex-A53 processor, a cache cache coherent interconnect and a re-configurable FPGA fabric capable of hosting the the framework as well as the HAs of interest.

6.4.1 Hardware Setup

The MPSoC chip features several interconnects, switches and data-paths that allows us to properly emulate the complex interactions between all the different data nodes in our design. More specifically, it includes a cache coherent interconnect (CCI-400) that supports full cache coherency between the memory masters in our proposed hardware architecture. In our experience, it is difficult to accurately simulate cache coherent interconnects in a way that captures the complex interactions and nuances of data movement between all of the system resources. This includes adherence to coherence protocol specifications, buffers, hazard management, protection, speculation and other low level behaviours. Although the hardware emulation platform limits our ability to perform design space exploration and "what-if" experiments, it provides us with a more realistic baseline configuration for heterogeneous SoCs used in low-power embedded applications.

ACAI's hardware components are implemented in Verilog, synthesized using Xilinx Vivado 2017.2 and programmed to the FPGA fabric on the MPSoC. For evaluation, we use a variant of the hardware design that has been extensively optimized for improved data bandwidth. Due to time constraints, this variant only models data movement across the HA memory interface. Compared to the base design, ACAI_{opt} has a single cache and can accommodate a large amount of outstanding requests to the CCI. This overcomes some bandwidth limitations in our base implementation. Accelerator configuration time and address translation is emulated using measured delays from the base design. The MMU emulator tracks the number 4K memory regions (N) accessed by the accelerator and adds delays based on latency measured from ACAI_{base}. A block diagram of ACAI_{opt} is presented in

figure 6.5.



Delay	**ACAI$_{base}$ measurement**	**Value**
τ_{JD}	Job descriptor fetch and configuration	294 Cycles
τ_{PTW}	MMU page table walk latency (2-level)	167 Cycles

$$Performance = Performance_{opt} + \tau_{JD} + N \times \tau_{PTW}$$

Figure 6.5: ACAI$_{opt}$ emulation for optimized for bandwidth

To compare ACAI's performance, we developed other interface options with the same HAs attached. The shared interface (Shared-NC) has no hardware data coherency and the user application is required to perform flushes and/or invalidations to synchronize data. The time spent conducting synchronization operations are included in the performance numbers presented for the Shared-NC interface. The shared IO-coherent interface (Shared-IO) features one-way coherency and virtual addressing through an IOMMU implemented in the ASIC. Although this interface allows the HA to snoop data from the CPU cache (similar to ACAI), data generated from the HA will be posted to the DDR controller (DDRC). Write from the HA will cause hardware invalidation messages to be sent to the CPU cache.

A block diagram showing the interactions between the various hardware nodes is presented in figure 6.6.

6.4.1.1 Interface Protocols

The prototype setup leverages a combination of interface protocols from the *ARM AMBA AXI and ACE Protocol Specification* [97]. The *Advanced eXtensible Interface* (AXI) protocol is adopted as the means to provide HA access to memory through the HA memory interface④. It is a point-to-point handshake interface mastered by the HA. Memory requests from the HA propagate through intermediate nodes until the transaction completes.

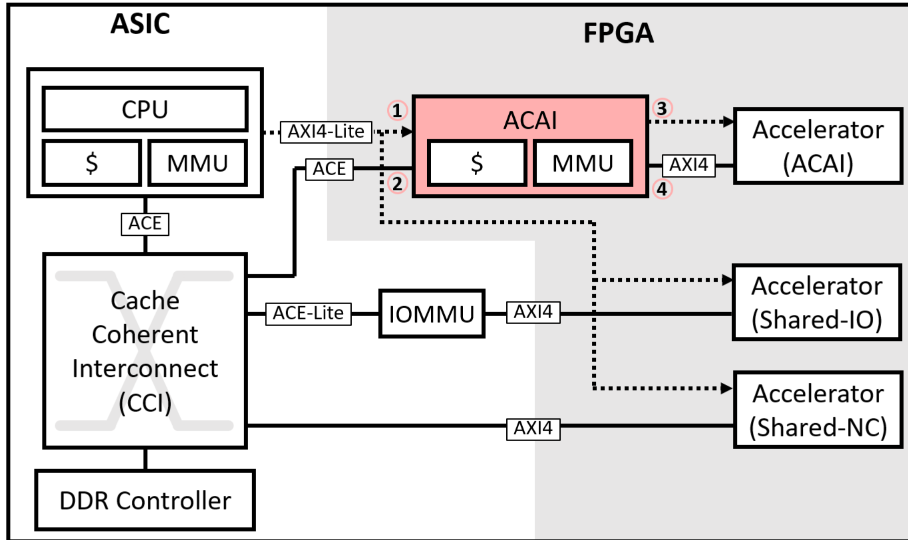


Figure 6.6: Hardware prototype platform showing HA integration using ACAI and other interface options

AXI transactions have a variable burst size (the number of data samples returned per requests) and supports an arbitrary number of outstanding requests from the master. It also defines a set of request signals (AxCACHE) that specify the *bufferable*, *cacheable* and *allocate* attributes of the transaction. The protocol grants the HA the ability to control the data granularity, bandwidth and other coherence attributes based on need. Support for variable burst size, number of outstanding requests and transaction attributes are dependent on the capability of the nodes (master, slave, caches, interconnect and switches) along the path of the transaction.

The *AXI Coherency Extension* (ACE) protocol builds on AXI, adding support for full-coherency. The interfaces that connect the CPU and the ACAI hardware ② to the CCI leverage the ACE protocol. The ACE-Lite protocol is a subset of ACE that only supports IO Coherency (one-way coherency) and is used between the interconnect and the IOMMU present on the ASIC. The light-weight AXI4-Lite protocol is used for the CPU peripheral interface① and the HA configuration③ interface.

6.4.1.2 Clocking

All of the designs and logic running on the FPGA fabric are clocked at 200 MHz. Although we expect the ACAI to run at much higher frequencies in ASIC implementations, this was a comfortable frequency to close timing on the FPGA across various HA designs. For

proper ASIC emulation, the rest of the system clocks have been scaled down to match the hardware implemented in the FPGA. This is achieved by re-programming the PLLs built into the MPSoC chip. Clock scaling is necessary for two reasons: firstly, it provides more accurate results when comparing execution speedups of kernels running on the HA against the CPU. Secondly and more importantly, it reflects the expected clock ratios on real ASIC SoC designs. This will preserve the latency ratio between nearby memory hits (in ACAI caches) and those further away (in CPU caches or DDRC). The frequency of the CPU is scaled by 6x ($1.2\,\mathrm{GHz} \rightarrow 200\,\mathrm{MHz}$), CCI by 6x ($533\,\mathrm{MHz} \rightarrow 88\,\mathrm{MHz}$) and the DDRC by 3x ($1067\,\mathrm{MHz} \rightarrow 355\,\mathrm{MHz}$). The DDRC is not scaled similarly due to minimum operating frequency requirements for the PLLs in the PHY to lock. This slightly skews the performance comparisons in favor of Shared-NC, since it changes the latency ratio between requests fulfilled by snoops and the DDRC.

A summary of the prototype setup is presented in table 6.1.

6.4.2 Software Setup

Performance is measured in CPU cycles using counters in the *Performance Monitoring Unit* (PMU) of the ARM Cortex-A53 processor. It includes the time taken to configure the HA, execute and receive a notification of completion by polling. We also include the time it takes to marshal the results back to the CPU. Our benchmark applications simply reads the result data back after execution. The marshalling time will vary based on the result data size and where it is retrieved from (ACAI cache vs. DDRC). The memory latency, as observed by the HA, is measured using Xilinx AXI Performance Monitor (APM) [99] cores placed along the HA memory interface④. The reported latency measures the number of cycles from when the HA declares intent (e.g issuing a read request, before acceptance from the downstream node) to when the request is completed (e.g. last data sample of a burst request is received).

The tests are setup and deployed from a user space applications running in Linux 4.9 on the ARM Cortex-A53 processor. Before offloading the kernels to the HA, the application performs memory allocations on any data structures that will be accessed by the HAs. The Linux kernel uses lazy (on-demand) allocation of physical pages, deferring the assignment of physical memory until first access. To ensure the page table entries have been created for addressing by the HA, we also initialize the data structures. Kernels executing on a HA attached to ACAI will start with a cold cache. Any required data will be snooped from CPU cache. We believe that our measurement procedures (data marshalling and cold caching) is more representative of real-world applications attempting to offload a kernel routine to a

Evaluation System Details	
CPU	Single Core Arm Cortex-A53 Processor 200 MHz (Scaled down from 1.2 GHz) 32 KiB L1 Cache, 1 MiB L2 Cache 64 B Cache Line Size Linux 4.9 Operating System
CCI	88 MHz (Scaled down from 533 MHz) Snoop-based Coherence Speculative Fetch Disabled
DDR Controller	355 MHz (Scaled down from 1067 MHz)
FPGA Fabric	200 MHz
Accelerator Interface Details	
$ACAI_{base}$	Full Memory Coherence 64 KiB L1 Cache, 512 KiB L2 Cache Virtual Addressing 64-bit data bus width Min Latency: 4 Cycles (to FPGA L1$)
$ACAI_{opt}$	Full Memory Coherence 512 KiB L2 Cache Emulated Virtual Addressing 64-bit data bus width Min Latency: 7 Cycles (to FPGA L2$)
Shared-IO	I/O Memory Coherence Virtual Addressing 64-bit data bus width Min Latency: 79 Cycles (to CPU L2$)
Shared-NC	No Memory Coherence Physical Addressing 64-bit data bus width Min Latency: 98 Cycles (to DDRC)

Table 6.1: Prototype system configuration and details

HA. Input data will likely have been accessed recently and the results will likely be used upon kernel completion.

For benchmarks running over the Shared-IO interface, we leveraged existing *Virtual Function I/O* (VFIO) [100] and ARM SMMU [101] drivers. The application creates special buffers with virtual addresses that can be translated by the SMMU servicing the interface. For benchmarks running over the Shared-NC interface, we leveraged custom device drivers that allocate contiguous memory blocks in the kernel space and makes them available to user space (zero-copy). The Shared-NC driver allows the blocks to be cached in the CPU and provides mechanisms for performing data synchronization.

6.4.3 Kernels and Applications

Synthetic kernels: We first evaluate our proposed framework using a set of synthetic kernels that aim to provide us with an understanding of it's memory latency and bandwidth. These pedagogical HAs perform little to no computation but exhibit various memory access patterns that are representative of memory demanding kernels. These kernels are written in C and the HA is generated from Xilinx Vivado HLS 2017.2. The first synthetic HA performs *sequential bulk-data copy* from one buffer to another. The pseudo-code for this HA is presented in figure 6.7. This HA models accelerators with buffered implementations where the performance may be limited by bandwidth. The second synthetic kernel per-

```
1 void memcopy(u64 src[N], u64 dst[N]) {
2   u64 buff[N];
3   memcpy(buff, src, N * sizeof(u64));
4   memcpy(dst, buff, N * sizeof(u64));
5 }
```

Figure 6.7: Pseudo-code for sequential bulk-data copy kernel

forms *random bulk-data copy* from one buffer to another. The randomness is implemented using a linear feedback shift register (LFSR), ensuring that each memory address is visited only once. The pseudo-code for this HA is presented in figure 6.8. This HA models newer workloads that have little to no buffering, may be operating on large data sets and have interleaved read and write memory requests. The third synthetic kernel performs a linked-list traversal, summing up the data in each node as it progresses. The pseudo-code for this HA is presented in figure 6.9. This HA models kernels that utilize software data structures and are sensitive to memory latency. All synthetic HAs are implemented with optimization directives to ensure maximum bus utilization.

```
#include "lfsr.cpp"
void memcopy(u64 src[N], u64 dst[N]) {
  for (int i = 0; i < N; ++i)
    dst[LFSR1_N()] = src[LFSR2_N()];
  dst[0] = src[0]; // LFSR is not inclusive of 0
}
```

Figure 6.8: Pseudo-code for random bulk-data copy kernel

```
typedef struct nodeType {
   nodeType* next;
   u64 payload[PAYLOAD_SIZE];
 } nodeType;

void linklist(nodeType *head, u64 *result) {
  u64 tmp_result = 0;
  nodeType* curNode=head;
  while (curNode != NULL) {
    for (int i = 0; i < PAYLOAD_SIZE; ++i)
      tmp_result += curNode.payload[i];
    curNode = curNode.next;
  }
  *result = tmp_result;
}
```

Figure 6.9: Pseudo-code for linked-list data traversal kernel

Micro-kernels: We also study the performance of the framework across a series of micro-kernels selected from Machsuite [62]. Although the HA generated from these kernels are not optimized for performance, they are representative of workloads with diverse memory demands (e.g. data stride, data size, cache locality, etc). They are also representative of hardware accelerators generated using High Level Synthesis (HLS) techniques. Table 6.2 summarizes the different selected kernels and their characteristics.

Applications: Lastly, we explore the performance of ACAI across a few applications from the Rosetta [63] benchmark suite. These are full-blown applications comprised of multiple kernels and is targeted to meet realistic performance requirements. They also exhibit different data use and execution models. The HAs generated for these applications are already optimized for performance and show significant speedups over embedded CPU implementations. Table 6.3 summarizes the different Rosetta applications and their properties.

Kernel	Description	Locality	Data Size	N
backprop	Neural network training	High	61.4 KiB	16
bfs/bulk	Breadth-first search	Low	38.1 KiB	10
bfs/queue	Breadth-first search	Low	38.1 KiB	10
fft/strided	Fast Fourier transform	High	24 KiB	6
fft/transpose	Fast Fourier transform	High	8 KiB	2
gemm/blocked	Matrix multiplication	High	384 KiB	96
gemm/ncubed	Matrix multiplication	High	384 KiB	96
md/knn	Molecular dynamics	Low	44 KiB	11
sort/merge	Sorting	High	16 KiB	4
sort/radix	Sorting	High	49 KiB	13
spmv/crs	Sparse matrix multiplication	Low	37.6 KiB	10
stencil2d	Stencil computation	High	128.1 KiB	33
stencil3d	Stencil computation	High	256 KiB	64

Table 6.2: Machsuite kernel details

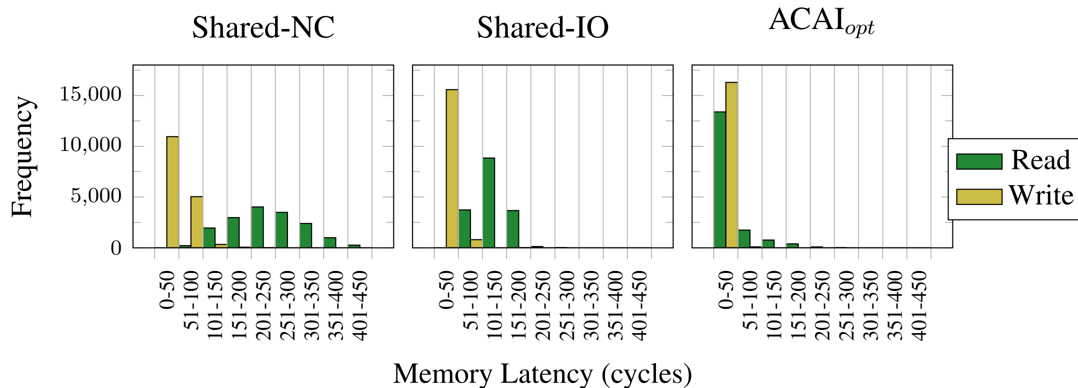


Figure 6.10: Memory latency distribution for random memory requests

6.5 Evaluation

6.5.1 Latency and Bandwidth

Figure 6.10 shows the memory latency distribution for read and write requests as observed by the synthetic bulk random data copy HA across different interfaces. The transaction latencies are presented in 50-cycle bin sizes. The HA is configured to access a total data size of 256 KiB. Shared-IO and ACAI experience significantly better read latencies compared to Shared-NC because they can snoop data directly from the neighboring CPU cache. ACAI is able to experience even lower latency because it is able to exploit spatial locality within its cache. The minimum observed read latencies are 98 cycles on Shared-NC (to the DDRC), 79 cycles on Shared-IO (to CPU L2$) and 7 cycles (to ACAI cache). On average, ACAI has

Application	Property	Test Size	N
Digit Recognition	Compute Bound	627.0kB	157
Face Detection	Compute Bound	76.6kB	20
3D Rendering	Compute Bound	101.4kB	26
Spam Filter	Memory Bound	8.80MB	2253

Table 6.3: Rosetta application details

a read latency improvement of 12.1x and 5.7x over Shared-NC and Shared-IO respectively, for random bulk data access. The disparity in latency is not as prominent for write requests because the intermediate nodes are able to buffer the data before forwarding to the final destination.

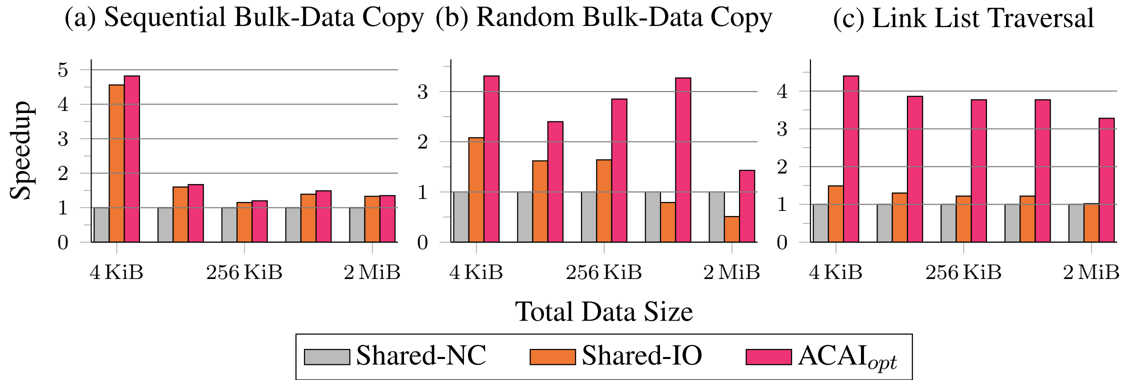


Figure 6.11: Synthetic benchmark performance normalized to Shared-NC

Figure 6.11-a shows the performance results for sequential bulk-data copy normalized to Shared-NC. The results are presented for different data sizes. At small data sizes, the performance of the Shared-NC suffers from software coherence overheads. The overheads become less pronounced as the data size increases, demonstrating the importance of hardware coherence for fine-grained accelerators. The performance gap across all interfaces begins to close at larger data sizes. There are no boosts from spatial locality because the HA performs burst memory requests (each read request returns 2 cache lines) across all interfaces. ACAI has a speedup of 4.5x to 1.4x over Shared-NC for data sizes ranging from 4 KiB to 2 MiB. The primary factor that affects bandwidth for sequential bulk-data reads, is the ability to pipeline memory transactions at all participating nodes. This was also the primary driver for developing a variant that supports multiple outstanding cache line fills. Our results also show that HAs operating with virtual addresses and caches are able to maximize the full memory bandwidth with properly pipelined implementations.

Figure 6.11-b shows the performance results for random bulk-data copy normalized to

Shared-NC. The results are also presented for different data sizes. The interleaved non-sequential access pattern prevents the HA from leveraging burst AXI transactions. Shared-IO outperforms Shared-NC at smaller data sizes but falls behind at larger data sizes (0.5x at 2 MiB). This is due to the increased network traffic at all nodes along the requests and to a larger number of granular (non-burst) requests in the network. Write requests from the the Shared-IO interface will also force CPU cache lines to be invalidated, creating even more traffic as the lines are flushed to DRAM. Negative effects from address translation, buffer pressure, data hazard management, and CPU snoop response latency become more pronounced and begin to affect Shared-IO performance. ACAI is able to overcome this by using its cache to filter traffic and performing cache line migrations. ACAI outperforms Shared-NC by 2.5x to 3.3x for data sizes less than its cache (512 KiB) and continues to perform at 1.4x for data sizes beyond that.

Figure 6.11-c shows the performance result for linked-list data traversal. Each node has a 24 B payload and the length of the list is increased to sweep through a total data size of 4 KiB to 2 MiB. This HA uses a mixture of non-burst and burst transactions as it executes. ACAI is able to outperform other interfaces by 3.2x-4.4x across different data sizes due to reduced memory latency.

6.5.2 Single Kernel Analysis

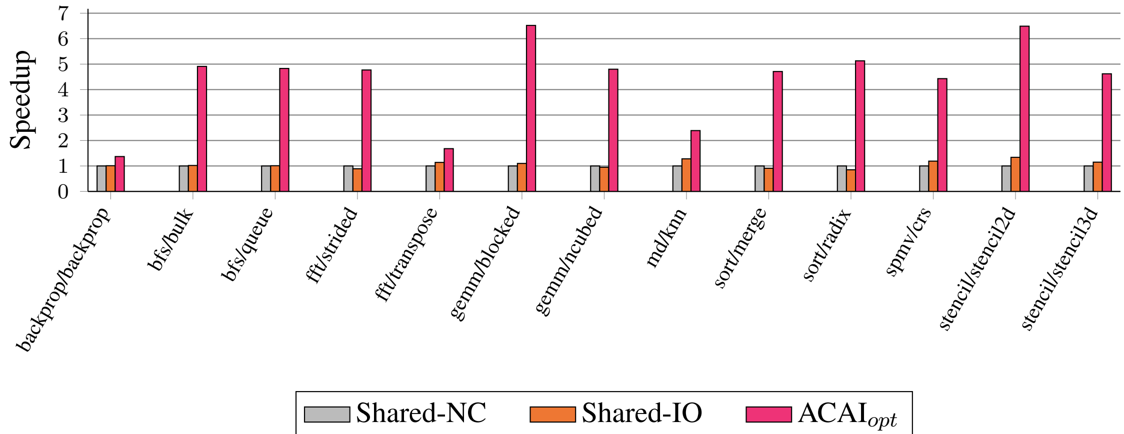


Figure 6.12: Machsuite performance results across various kernels

Figure 6.12 shows the performance result for several HAs derived from machsuite kernels normalized to Shared-NC. In every kernel, ACAI is able to outperform both Shared-IO and Shared-NC. ACAI has an average speed up of 4.3x and a max speedup of 6.5x across 13 kernels.

6.5.3 Application Analysis

Figure 6.13 shows the performance result for several application HAs generated from Rosetta and normalized to Shared-NC. In every application, ACAI is able to outperform both Shared-IO and Shared-NC accelerators. ACAI has an average speed up of 1.4x and a max speedup of 1.8x across 4 applications.

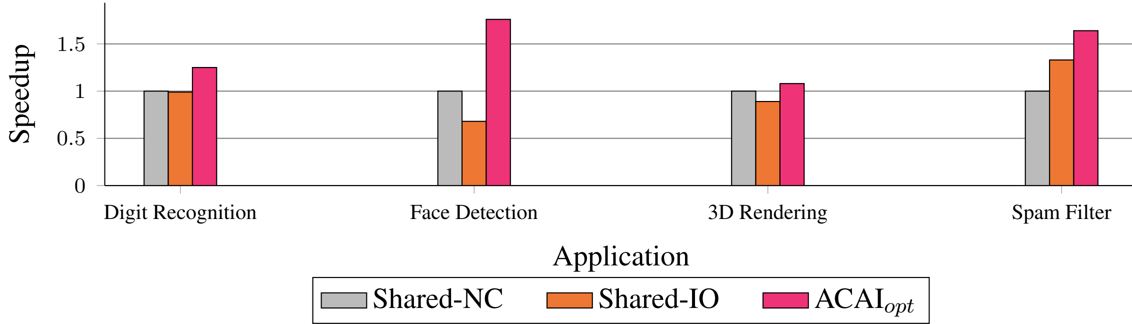


Figure 6.13: Rosetta Benchmark Results across various applications

6.6 Future work

Implementation: From our experiments, we observed that performance can be impeded by low-level implementation details that manifest across the system. Details such as the number of outstanding memory requests supported by the memory nodes and interconnect, buffers within each node, hazard control mechanisms, and snoop response behavior can significantly changed the latency and/or performance results. We plan to fully integrate the optimizations made for ACAI$_{opt}$. ACAI improves accelerator performance by employing efficient cache-to-cache transfers through the interconnect. The interconnect in the MPSoC device implements a snoop-based coherence protocol that broadcasts coherence messages to all coherent masters. A dictionary-based coherence protocol would allow the framework scale more efficiently when the working data size is larger than the CPU cache or to systems with many masters.

Recently released interconnect architectures [102, 103] have much higher bandwidth and can cater to high performance workloads. Our future goal is to accurately simulate many of these low-level implementation details and perform a design-space exploration for systems with different topologies (e.g off-chip accelerators, storage and caches).

Power/Energy: We are yet to perform a complete power and energy analysis on the framework. Although the cache and MMU units in ACAI will consume significant amounts of

energy, we anticipate some of this to be offset by reduced data movement. After the job chains have been setup, the job control unit is capable of independently executing all jobs on the HA resources without CPU intervention. Additional power savings may be realized by putting the CPU into a low-power state until signaled appropriately.

Protocol Updates: Full cache coherency promotes the HA to the same rank as the CPU, with ACAI serving to abstract away the complexities of maintaining coherence. This abstraction layer can easily be updated to exploit advanced coherence messages supported by the interconnect protocol. This includes cache stashing [104], where result data from the interface can be pushed directly into the CPU's cache. A similar optimization can be performed in the opposite direction by having the user application issue instructions to preemptively demote cache lines to a lower hierarchy, thereby moving data closer to the consumer [105].

6.7 Related Work

This work primarily focuses on designing a robust hardware interface for fixed-function HAs that reduces data movement and improves the programming experience for applications attempting to exploit hardware acceleration. Section 6.2 describes and references prior work that has explored shared memory, virtual addressing and caches for heterogeneous systems.

There are emerging bus/interconnects standards being developed in industry that also aim to foster tighter coupling between processors and accelerators (GPUs, HAs, FPGAs, etc). An in-work specification for OpenCAPI [106] adds support for full cache coherency. Cache Coherent Interconnect for Accelerators (CCIX) [107] is a multichip standard that aims to extend cache coherency to a large number of acceleration devices. Gen-Z [108] provides memory-semantic access to data and aims to scale to pooled memory across a server rack.

Prior work have proposed various coherence protocols and system topologies that focus on different HA integration challenges. The Fusion cache hierarchy [90] relies on a timestamp based, self-invalidation protocol for accelerators sharing a cache. Spandex [81] is an interface that relies on a DeNovo-based LLC to connect devices with diverse protocols and memory demands. Heterogeneous System Coherence (HSC) [109] uses a modified directory/interconnect and memory hierarchy for CPU-GPU systems. Crossing Guard integrates cached accelerators using a simplified coherence protocol and adapts it to the host coherence protocol. Past research have also proposed software backed mechanisms for address translation and memory protection for accelerators [110, 111].

Many efforts have been made to improve the programming experience for applications accelerated in FPGA hardware. Xilinx SDx development environment [112] uses high level programming languages to provide system-level abstractions. The Acceleration Stack for Intel Xeon CPU with FPGAs [113] provides software APIs intended to simplify application development and deployment.

6.8 Conclusion

Efficient hardware acceleration is a hard problem that impacts designers at every layer of the architecture. Application designers are often faced with rigid implementation requirements and are unfamiliar with the underlying complexities in data management across the hardware platform. SoC architects face complex design trade-offs when weighing hardware acceleration needs against other platform demands. Hardware accelerator developers face design restrictions and memory limitations that impact the utilization of the HA.

ACAI addresses this problem by simplifying the software application stack, transparently sharing the process address space and coordinating the execution of jobs across HA resources. With full cache coherency, it eases HA integration into the platform while abstracting away the intricacies of coherence management. ACAI's hardware interface maximizes memory bandwidth and significantly improves latency using its cache. Lower latency enables fine-grained acceleration and performance improvements across diverse workloads.

CHAPTER 7

Conclusions

With the end of Dennard's scaling and a slowdown in Moore's Law, system architects have developed several techniques to deliver on the traditional performance and power improvements we have come to expect. The various techniques adopted to combat this phenomenon has directly translated to an increase in complexity for modern SoCs. This increase in complexity has in-turn lead to an increase in design effort and validation time.

This dissertation presented several techniques to address the challenges to rapidly birthing complex SoCs. The first part of this dissertation focused on the foundations and architectures that aid in rapid SoC design. It presented a variety of architectural techniques that were developed and leveraged to rapidly construct complex SoCs at advanced process nodes. The next part of the dissertation focused on the gap between a completed design model(in RTL form) and its physical manifestation (a GDS file that will be sent to the foundry for fabrication). It presented methodologies and a workflow for rapidly walking a design through to completion at arbitrary technology nodes. It also presented progress on creating tools and a flow that is entirely dependent on open-source tools. The last part focused framework that not only speeds up the integration of a hardware accelerator into an SoC ecosystem, but emphasizes software adoption and usability

BIBLIOGRAPHY

[1] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, “The aladdin approach to accelerator design and modeling,” *IEEE Micro*, vol. 35, no. 3, pp. 58–70, 2015.

[2] T. Ajayi, K. Al-Hawaj, A. Amarnath, S. Dai, S. Davidson, P. Gao, G. Liu, A. Lotfi, J. Puscar, A. Rao, *et al.*, “Celerity: An open source risc-v tiered accelerator fabric,” in *Symp. on High Performance Chips (Hot Chips)*, 2017.

[3] T. Ajayi, K. Al-Hawaj, A. Amarnath, S. Dai, S. Davidson, P. Gao, G. Liu, A. Rao, A. Rovinski, N. Sun, C. Torng, L. Vega, B. Veluri, S. Xie, C. Zhao, R. Zhao, C. Batten, R. G. Dreslinski, R. K. Gupta, M. B. Taylor, and Z. Zhang, “Experiences using the risc-v ecosystem to design an accelerator-centric soc in tsmc 16nm,” in *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, ACM, 2017.

[4] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, *et al.*, “The celerity open-source 511-core risc-v tiered accelerator fabric: Fast architectures and design methodologies for fast chips,” *IEEE Micro*, vol. 38, no. 2, pp. 30–41, 2018.

[5] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, *et al.*, “A 1.4 ghz 695 giga risc-v inst/s 496-core manycore processor with mesh on-chip network and an all-digital synthesized pll in 16nm cmos,” in *2019 Symposium on VLSI Circuits*, pp. C30–C31, IEEE, 2019.

[6] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, *et al.*, “Evaluating celerity: A 16-nm 695 giga-risc-v instructions/s manycore processor with synthesizable pll,” *IEEE Solid-State Circuits Letters*, vol. 2, no. 12, pp. 289–292, 2019.

[7] A. Olofsson, “Epiphany-v: A 1024 processor 64-bit risc system-on-chip,” *arXiv preprint arXiv:1610.01832*, 2016.

[8] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, *et al.*, “The rocket chip generator,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[9] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, *et al.*, “The raw microprocessor: A computational fabric for software circuits and general-purpose programs,” *IEEE micro*, vol. 22, no. 2, pp. 25–35, 2002.

[10] H. Hoffmann, D. Wentzlaff, and A. Agarwal, "Remote store programming: A memory model for embedded multicore," *HiPEAC'10*, p. 3–17, 2010.

[11] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *International Conference on Compiler Construction*, pp. 179–196, Springer, 2002.

[12] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable fpgas," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 15–24, 2017.

[13] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[14] M. B. Taylor, "Basejump stl: Systemverilog needs a standard template library for hardware design," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2018.

[15] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic, *et al.*, "An agile approach to building risc-v microprocessors," *IEEE Micro*, vol. 36, no. 2, pp. 8–20, 2016.

[16] R. Dreslinski, D. Wentzloff, M. Fayazi, K. Kwon, D. Blaauw, D. Sylvester, B. Calhoun, M. Coltella, and D. Urquhart, "Fully-autonomous soc synthesis using customizable cell-based synthesizable analog circuits," tech. rep., University of Michigan Ann Arbor United States, 2019.

[17] T. Ajayi, S. Kamineni, Y. K Cherivirala, M. Fayazi, K. Kwon, M. Saligane, S. Gupta, C.-H. Chen, D. Sylvester, D. Blaauw, R. Dreslinski Jr, B. Calhoun, and D. Wentzloff, "An open-source framework for autonomous soc design with analog block generation," in *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SoC)*, IEEE, 2020.

[18] Accellera, "IP-XACT - Accellera." https://www.accellera.org/downloads/standards/ip-xact. Last accessed 2020-05-03.

[19] C.-Y. Wu, H. Graeb, and J. Hu, "A pre-search assisted ilp approach to analog integrated circuit routing," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pp. 244–250, IEEE, 2015.

[20] K. Kunal, M. Madhusudan, A. K. Sharma, W. Xu, S. M. Burns, R. Harjani, J. Hu, D. A. Kirkpatrick, and S. S. Sapatnekar, "ALIGN: Open-source analog layout automation from the ground up," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–4, 2019.

[21] B. Xu, K. Zhu, M. Liu, Y. Lin, S. Li, X. Tang, N. Sun, and D. Z. Pan, "MAGICAL: Toward fully automated analog ic layout leveraging human and machine intelligence," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2019.

[22] Y. Park and D. D. Wentzloff, "An all-digital 12pj/pulse 3.1–6.0 ghz ir-uwb transmitter in 65nm cmos," in *2010 IEEE International Conference on Ultra-Wideband*, vol. 1, pp. 1–4, IEEE, 2010.

[23] Y. Park and D. D. Wentzloff, "An all-digital pll synthesized from a digital standard cell library in 65nm cmos," in *2011 IEEE Custom Integrated Circuits Conference (CICC)*, pp. 1–4, IEEE, 2011.

[24] E. Ansari and D. D. Wentzloff, "A 5mw 250ms/s 12-bit synthesized digital to analog converter," in *Proceedings of the IEEE 2014 Custom Integrated Circuits Conference*, pp. 1–4, IEEE, 2014.

[25] S. Bang, A. Wang, B. Giridhar, D. Blaauw, and D. Sylvester, "A fully integrated successive-approximation switched-capacitor dc-dc converter with 31mv output voltage resolution," in *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pp. 370–371, IEEE, 2013.

[26] W. Jung, S. Jeong, S. Oh, D. Sylvester, and D. Blaauw, "A 0.7 pf-to-10nf fully digital capacitance-to-digital converter using iterative delay-chain discharge," in *2015 IEEE International Solid-State Circuits Conference-(ISSCC) Digest of Technical Papers*, pp. 1–3, IEEE, 2015.

[27] M. Shim, S. Jeong, P. Myers, S. Bang, C. Kim, D. Sylvester, D. Blaauw, and W. Jung, "An oscillator collapse-based comparator with application in a 74.1 db sndr, 20ks/s 15b sar adc," in *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, pp. 1–2, IEEE, 2016.

[28] S. Bang, W. Lim, C. Augustine, A. Malavasi, M. Khellah, J. Tschanz, and V. De, "A fully synthesizable distributed and scalable all-digital ldo in 10nm cmos," in *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 380–382, IEEE, 2020.

[29] S. Kundu, L. Chai, K. Chandrashekar, S. Pellerano, and B. Carlton, "A self-calibrated 1.2-to-3.8 ghz 0.0052mm^2 synthesized fractional-n mdll using a 2b time-period comparator in 22nm finfet cmos," in *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 276–278, IEEE, 2020.

[30] C. Wolf, "Yosys open synthesis suite." http://www.clifford.at/yosys/. Last accessed 2020-05-08.

[31] "Ngspice, the open source spice circuit simulator." http://ngspice.sourceforge.net/. Last accessed 2020-05-08.

[32] S. N. Laboratories, "Xyce parallel electronic simulator (xyce)." https://xyce.sandia.gov/. Last accessed 2020-05-08.

[33] D. M. Moore, T. Xanthopoulos, S. Meninger, and D. D. Wentzloff, "A 0.009 mm 2 wide-tuning range automatically placed-and-routed adpll in 14-nm finfet cmos," *IEEE Solid-State Circuits Letters*, vol. 1, no. 3, pp. 74–77, 2018.

[34] M. H. Perrott, M. D. Trott, and C. G. Sodini, "A modeling approach for Σ-Δ fractional-N frequency synthesizers allowing straightforward noise analysis," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 8, pp. 1028–1038, 2002.

[35] Y. Okuma, K. Ishida, Y. Ryu, X. Zhang, P.-H. Chen, K. Watanabe, M. Takamiya, and T. Sakurai, "0.5-v input digital ldo with 98.7% current efficiency and 2.7-μa quiescent current in 65nm cmos," in *IEEE Custom Integrated Circuits Conference 2010*, pp. 1–4, IEEE, 2010.

[36] S. Weaver, B. Hershberg, P. Kurahashi, D. Knierim, and U.-K. Moon, "Stochastic flash analog-to-digital conversion," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 57, no. 11, pp. 2825–2833, 2010.

[37] M. Saligane, M. Khayatzadeh, Y. Zhang, S. Jeong, D. Blaauw, and D. Sylvester, "All-digital soc thermal sensor using on-chip high order temperature curvature correction," in *2015 IEEE Custom Integrated Circuits Conference (CICC)*, pp. 1–4, IEEE, 2015.

[38] S. Nalam, M. Bhargava, K. Mai, and B. H. Calhoun, "Virtual prototyper (vipro): An early design space exploration and optimization tool for sram designers," in *Design Automation Conference*, pp. 138–143, IEEE, 2010.

[39] D. Johannsen, "Bristle blocks: A silicon compiler," in *16th Design Automation Conference*, pp. 310–313, IEEE, 1979.

[40] T. Ajayi, D. Blaauw, T. Chan, C. Cheng, V. A. Chhabria, D. K. Choo, M. Coltella, R. Dreslinski, M. Fogaça, S. Hashemi, A. Ibrahim, A. B. Kahng, M. Kim, J. Li, Z. Liang, U. Mallappa, P. Penzes, G. Pradipta, S. Reda, A. Rovinski, K. Samadi, S. Sapatnekar, L. Saul, C. Sechen, V. Srinivas, W. Swartz, D. Sylvester, D. Urquhart, L. Wang, M. Woo, and B. Xu, "Openroad: Toward a self-driving, open-source digital layout implementation tool chain," 2019.

[41] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem, *et al.*, "Toward an open-source digital flow: First learnings from the openroad project," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–4, 2019.

[42] A. Rovinski, T. Ajayi, M. Kim, G. Wang, and M. Saligane, "Bridging academic open-source eda to real-world usability," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–7, IEEE, 2020.

[43] W.-T. J. Chan, A. B. Kahng, S. Nath, and I. Yamamoto, "The itrs mpu and soc system drivers: Calibration and implications for design-based equivalent scaling in the roadmap," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pp. 153–160, IEEE, 2014.

[44] "The openroad project." https://theopenroadproject.org. Accessed: 2020-09-12.

[45] "Replace." https://github.com/abk-openroad/RePlAce. Accessed: 2020-09-12.

[46] C.-K. Cheng, A. B. Kahng, I. Kang, and L. Wang, "Replace: Advancing solution quality and routability validation in global placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 9, pp. 1717–1730, 2018.

[47] "Tritoncts." https://github.com/abk-openroad/TritonCTS. Accessed: 2020-09-12.

[48] C. Chu and Y.-C. Wong, "Flute: Fast lookup table based rectilinear steiner minimal tree algorithm for vlsi design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 1, pp. 70–83, 2007.

[49] "Opensta." https://github.com/abk-openroad/OpenSTA. Accessed: 2020-09-12.

[50] M. Fogaça, G. Flach, J. Monteiro, M. Johann, and R. Reis, "Quadratic timing objectives for incremental timing-driven placement optimization," in *2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 620–623, IEEE, 2016.

[51] K. Han, A. B. Kahng, and J. Li, "Optimal generalized h-tree topology and buffering for high-performance and low-power clock distribution," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[52] C. J. Alpert, W.-K. Chow, K. Han, A. B. Kahng, Z. Li, D. Liu, and S. Venkatesh, "Prim-dijkstra revisited: Achieving superior timing-driven routing trees," in *Proceedings of the 2018 International Symposium on Physical Design*, pp. 10–17, 2018.

[53] A. B. Kahng, L. Wang, and B. Xu, "Tritonroute: an initial detailed router for advanced vlsi technologies," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2018.

[54] "Ispd-2018 initial detailed routing contest." http://www.ispd.cc/contests/18/index.html. Accessed: 2020-09-12.

[55] L. Kou, G. Markowsky, and L. Berman, "A fast algorithm for steiner trees," *Acta informatica*, vol. 15, no. 2, pp. 141–145, 1981.

[56] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[57] "Tritonsizer." https://github.com/abk-openroad/TritonSizer. Accessed: 2020-09-12.

[58] S. Hashemi, C. Ho, A. Kahng, H. Liu, and S. Reda, "Metrics 2.0: A machine-learning based optimization system for ic design," in *Workshop on Open-Source EDA Technology*, p. 21, 2018.

[59] "The metrics initiative." https://vlsicad.ucsd.edu/GSRC/metrics/. Accessed: 2020-09-12.

[60] K. Chodorow, *MongoDB: the definitive guide: powerful and scalable data storage*. " O'Reilly Media, Inc.", 2013.

[61] "Vlsi cad bookshelf 2." http://vlsicad.eecs.umich.edu/BK/. Accessed: 2020-09-12.

[62] B. Reagen, R. Adolf, Y. S. Shao, G. Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *IISWC 2014 - IEEE International Symposium on Workload Characterization*, 2014.

[63] Y. Zhou, G. A. Velasquez, W. Wang, Z. Zhang, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, and G. Liu, "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs," *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '18*, 2018.

[64] Design Automation Conference (DAC), 2018 55th ACM/EDAC/IEEE, "Acai: Arm coherent accelerator interface." https://dac2018.zerista.com/event/member?item_id=7860515. Last accessed 2020-05-08.

[65] Arm Ltd, "Enabling hardware accelerator and soc design space exploration." https://community.arm.com/developer/research/b/articles/posts/enabling-hardware-accelerator-and-soc-design-space-exploration. Last accessed 2020-05-08.

[66] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011.

[67] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 449–460, IEEE Computer Society, 2012.

[68] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 243–254, IEEE Press, 2016.

[69] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[70] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 269–284, ACM, 2014.

[71] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–13, IEEE, 2016.

[72] S. Pal, R. Dreslinski, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, and T. Mudge, "OuterSPACE: An Outer Product based Sparse Matrix Multiplication Accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 724–736, IEEE, 2018.

[73] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the Walkers: Accelerating Index Traversals for In-memory Databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 468–479, ACM, 2013.

[74] D. Sidler, M. Owaida, Z. István, K. Kara, and G. Alonso, "doppiodb: A hardware accelerated database," in *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*, p. 1, 2017.

[75] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation Cores: Reducing the Energy of Mature Computations," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 205–218, ACM, 2010.

[76] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A Pre-RTL, Power-performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures," *SIGARCH Comput. Archit. News*, vol. 42, pp. 97–108, June 2014.

[77] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "SNNAP: Approximate Computing on Programmable SoCs via Neural Acceleration," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 603–614, IEEE, 2015.

[78] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, (New York, NY, USA), pp. 33–36, ACM, 2011.

[79] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pp. 69–70, IEEE, 2004.

[80] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pp. 1212–1221, IEEE, 2012.

[81] J. Alsop, M. D. Sinclair, and S. V. Adve, "Spandex: A Generalized Interface for Flexible Heterogeneous Coherence," in *Proceedings of the 45th International Symposium on Computer Architecture*, ISCA, June 2018.

[82] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache coherence for gpu architectures," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, (Washington, DC, USA), pp. 578–590, IEEE Computer Society, 2013.

[83] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 457–467, ACM, 2013.

[84] M. D. Sinclair, J. Alsop, and S. V. Adve, "Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 161–174, ACM, 2017.

[85] G. Kyriazis, "Heterogeneous System Architecture: A Technical Review," *AMD Fusion Developer Summit*, p. 21, 2012. https://developer.amd.com/wordpress/media/2012/10/hsa10.pdf.

[86] R. Mijat, "Take GPU Processing Power Beyond Graphics with Mali GPU Computing." White Paper, Aug. 2012. https://developer.arm.com/-/media/Files/pdf/graphics-and-multimedia/WhitePaper_GPU_Computing_on_Mali.pdf.

[87] P. Vogel, A. Marongiu, and L. Benini, "Lightweight virtual memory support for zero-copy sharing of pointer-rich data structures in heterogeneous embedded SoCs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1947–1959, 2017.

[88] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting Address Translation for Accelerator-Centric Architectures," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 37–48, Feb. 2017.

[89] A. Kegel, P. Blinzer, A. Basu, and M. Chan, "Virtualizing IO through THE IO Memory Management Unit (IOMMU)," *ASPLOS-XXI Tutorials*, 2016.

[90] S. Kumar, A. Shriraman, and N. Vedula, "Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators," *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15*, 2015.

[91] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. C. Hoe, "A Study of Pointer-Chasing Performance on Shared-Memory Processor-FPGA Systems," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 264–273, ACM, 2016.

[92] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-level synthesis of dynamic data structures: A case study using Vivado HLS," in *Field-Programmable Technology (FPT), 2013 International Conference on*, pp. 362–365, IEEE, 2013.

[93] J. Gómez-Luna, I. El Hajj, L.-W. Chang, V. García-Floreszx, S. G. de Gonzalo, T. B. Jablin, A. J. Pena, and W.-m. Hwu, "Chai: Collaborative Heterogeneous Applications for Integrated-architectures," in *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*, pp. 43–54, IEEE, 2017.

[94] J. Kim and C. Batten, "Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 75–87, Dec. 2014.

[95] M. S. Orr, S. Che, A. Yilmazer, B. M. Beckmann, M. D. Hill, and D. A. Wood, "Synchronization Using Remote-Scope Promotion," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, (New York, NY, USA), pp. 73–86, ACM, 2015.

[96] K. Barker, T. Benson, D. Campbell, D. Ediger, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent, and A. Tumeo, *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. Pacific Northwest National Laboratory and Georgia Tech Research Institute, Dec. 2013. http://hpc.pnnl.gov/projects/PERFECT/.

[97] Arm, "AMBA AXI and ACE Protocol Specification - ARM Infocenter." http://infocenter.arm.com/help/topic/com.arm.doc.ihi0022e.

[98] Arm, "ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition - ARM Infocenter." http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html.

[99] Xilinx, "Xilinx AXI Performance Monitor." https://www.xilinx.com/products/intellectual-property/axi_perf_mon.htmll.

[100] Linux, "VFIO - "Virtual Function I/O." https://www.kernel.org/doc/Documentation/vfio.txt.

[101] Linux, "ARM System MMU Architecture Implementation." https://www.kernel.org/doc/Documentation/devicetree/bindings/iommu/arm%2Csmmu.txt.

[102] Arm, "AMBA 5 CHI Architecture Specification - ARM Infocenter." http:/infocenter.arm.com/help/topic/com.arm.doc.ihi0050c/index.html.

[103] W. J. Starke, J. Stuecheli, D. M. Daly, J. S. Dodson, F. Auernhammer, P. M. Sagmeister, G. L. Guthrie, C. F. Marino, M. Siegel, and B. Blaner, "The cache and memory subsystems of the ibm power8 processor," *IBM Journal of Research and Development*, vol. 59, pp. 3:1–3:13, Jan. 2015.

[104] Arm, "ARM DynamIQ Shared Unit Technical Reference Manual - ARM Infocenter." http://infocenter.arm.com/help/topic/com.arm.doc.100453_0002_00_en/rob1444131023516.html.

[105] Intel, "Intel Architecture Instruction Set Extensions and Future Features." Programming Reference, May 2018. https://software.intel.com/sites/default/files/managed/c5/15/architecture%2dinstruction%2dset%2dextensions%2dprogramming%2dreference.pdf.

[106] O. Consortium, "What is OpenCAPI?." https://opencapi.org/about/.

[107] C. Consortium *et al.*, "Cache Coherent Interconnect for Accelerators (CCIX)." http://www.ccixconsortium.com.

[108] G.-Z. Consortium, "Computer Industry Alliance Revolutionizing Data Access." https://genzconsortium.org.

[109] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous system coherence for integrated cpu-gpu systems," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 457–467, ACM, 2013.

[110] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: Sandboxing accelerators," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 470–481, ACM, 2015.

[111] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing memory in heterogeneous systems," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, (New York, NY, USA), pp. 637–650, ACM, 2018.

[112] Xilinx, "Programmable Abstractions." https://www.xilinx.com/products/design-tools/all-programmable-abstractions.html.

[113] Intel, "Intel FPGA Acceleration Hub - Acceleration Stack for Intel Xeon CPU with FPGAs?." https://www.intel.com/content/www/us/en/programmable/solutions/acceleration-hub/acceleration-stack.html.