Streamlining High-Performance Heterogeneous Hardware Design

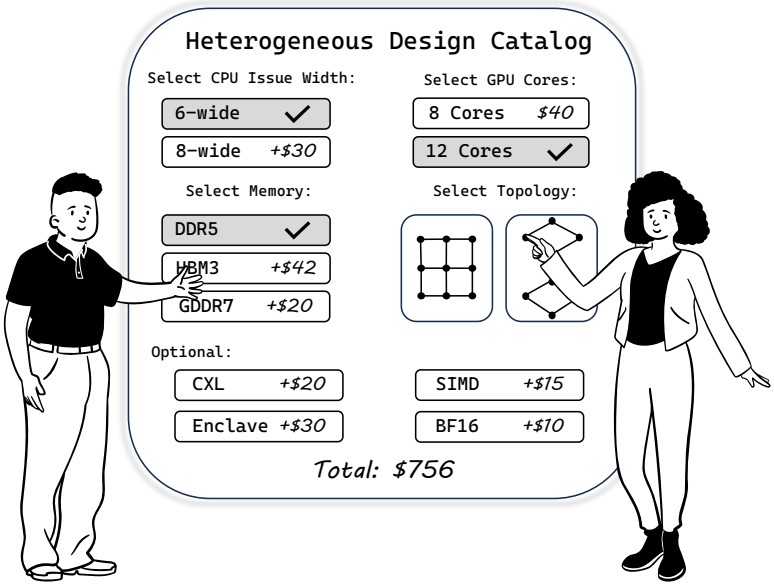
by

Shibo Chen

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2025

Doctoral Committee:

Professor Todd Austin, Chair
Professor Valeria Bertacco
Associate Professor Jean-Baptiste Jeannin
Assistant Professor Georgios Tzimpragos, University of Wisconsin - Madison


Heterogeneous Design Catalog
Select CPU Issue Width:
6-wide
8-wide +$30
Select GPU Cores:
8 Cores $40
12 Cores
Select Memory:
DDR5
HBM3 +$42
GDDR7 +$20
Select Topology:
Optional:
CXL +$20
Enclave +$30
SIMD +$15
BF16 +$10
Total: $756

Shibo Chen
chshibo@umich.edu
ORCID iD: 0000-0002-9522-8934

DEDICATION

To my family——Suicun Chen, Haiyan Chen, and Tong Chen——whose love and encouragement have nurtured my growth and inspired me to pursue excellence.

To my homestay family——Weidong Chu, Yingying Zhong, and Lin Chu——whose early influence ignited my passion for learning and helped shape the person I am today.

To my dear friend Ao Wang (1996–2019), whose kindness and companionship guided me through some of my dark days. I carry her memory with me, and I know she would be happy to see how far I have come.

May this dissertation stand as a tribute to our shared perseverance and the deep bonds of love that unite us.

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to everyone who has supported and contributed to this dissertation.

Above all, I am profoundly grateful to my advisor, Professor Todd Austin, for granting me the invaluable opportunity to join his research group in 2018. My journey began in winter 2017, when I attended his inspiring lecture on the value of graduate education—an experience that motivated me to pursue this degree. His exceptional guidance, patience, and unwavering support have been fundamental in my exploration of this research field. Professor Austin's mentorship, encouragement, and wisdom have profoundly shaped this dissertation and my growth as a researcher. I am particularly thankful for his understanding and patience when I made mistakes and faced challenges. Beyond academics, his charisma and optimism have had a significant influence on my personal growth; he remains an inspiring role model whom I deeply admire and strive to emulate.

I extend my sincere appreciation to my dissertation committee members, Professor Valeria Bertacco, Professor Jean-Baptiste Jeannin, and Professor Georgios Tzimpragos, for their consistent support and insightful feedback. Professor Bertacco's guidance and generosity, especially during critical moments, have played an indispensable role in my academic journey. Although she is not officially my advisor, she has considerably influenced my research perspectives through our weekly lab meetings, where she generously shares her ideas and industry insights. Her extensive experience as the university's vice provost has significantly enhanced my understanding of academia and contributed substantially to my professional growth. Professor Jeannin's collaborative spirit and resourcefulness have enriched multiple projects, some of which are included in this dissertation. His mathematical approach to problem-solving has effectively complemented my research style, which is more empirical and experiment-driven. Professor Tzimpragos introduced me to the Structural Simulation Toolkit, an invaluable resource in my research. Although our overlap at Michigan was brief, his consistent support and unique perspectives as a new faculty member transitioning from a recent PhD graduate have been greatly beneficial.

I am deeply grateful to my collaborators, including Yonathan Fisseha, Hailun Zhang, Yunjie Zhang, Qinhan Tan, Meron Demissie, Duo Xu, Yishen Zhou, Mark Gallagher, Professor Lauren Biernacki, and Professor Sharad Malik, whose contributions have substantially enriched both this

dissertation and my broader research outlook. My work would not have been possible without their invaluable input. Some collaborators were my undergraduate mentees; I sincerely wish them success in their future careers, as I have learned just as much from them as they have from me. I particularly thank Jeremy Erikson, who introduced me to research and guided me through my first publication, laying a crucial foundation for my academic path.

I warmly thank my friends and peers in the field whose ideas and feedback continuously inspired me throughout my PhD journey, especially Jiacheng Ma, Yunjie Pan, Yufeng Gu, Juechu Dong, Tarunesh Verma, and all my labmates in the ABLab Research Group. Heartfelt appreciation also goes to my close friends and cohorts——Minxue Niu, Jinyang Li, Tianji Cong, Junjie Xing, Yin Lin, and Shengpu Tang——for sharing memorable times and experiences during this journey. I am profoundly grateful for the support of my numerous friends from college, high school, middle school, and even childhood——Yu Huang, Mingzhe Shao, Heliu Dong, Yanyi Zheng, Jia Jia, Yuhan Liu, Zixiang Jin, Yin Li, Xiaoou Wang, Alex Kisil, Zhu Jing, and Yunfei Song. Although we cannot easily meet in person, they remain my go-to individuals for sharing both achievements and challenges. May our friendships endure forever! I recognize that it is impossible to list every person who has supported me; therefore, I extend my sincere gratitude to all friends whose names may not appear here.

I would also like to thank Professors Baris Kasikci and Morley Mao, with whom I worked during my undergraduate years. Their kind recommendation letters ultimately secured my admission into the PhD program at the University of Michigan. This academic journey would have been impossible without their generous support and encouragement.

My deepest appreciation goes to the University of Michigan, especially the Department of Computer Science and Engineering, for providing exceptional resources and a nurturing research environment. I am also grateful to the members and sponsors of the CELab Reading Group, the Security Reading Group, and the System Reading Group for the stimulating discussions that broadened my insights across diverse research fields.

A special thank you to my fellow CSEG officers——particularly Madelyn Gatchel, Noah Curran, and Chen Liang——for their dedicated service alongside me during my term as President in the 2022-2023 academic year, as we collectively revitalized our community following challenging times. I also greatly appreciate the steadfast support from the CSE leadership, including Professors Michael Wellman, Emily Provost, and Westley Weimer.

Additionally, I acknowledge the dedicated CSE staff whose tireless efforts sustain our community, particularly Magda Calvillo, Jasmin Stubblefield, Christa Carr, Kelly Cormier, Cindy Estell, Jamie Goldsmith, Stephanie Jones, Emily Johnson, Karen Liska, Alice Melloni, Stephen Reger, Sarah Snay, Erika Hauff, Ashley Andreae, among others. Special thanks to the custodial staff,

whose early-morning diligence ensures our research and meeting spaces remain welcoming and comfortable.

Thank you to the funding organizations that supported my research, notably the Center for Applications Driving Architectures (ADA), a Semiconductor Research Corporation (SRC) program, whose events provided valuable feedback and opportunities for interaction with esteemed scholars and industry sponsors.

My heartfelt thanks go to the members of the Ann Arbor Next Level Running Club, especially Bin Xu, Hongzhi Miao, Zaiyi Jiang, Zhengke Wu, Shujun Dong, Pai Liu, Xiong Zeng, Qingzhuo Liu, and Yiqi Li. I began endurance training in my third year of the PhD program, and completing this long and demanding process would have been impossible without their encouragement and companionship.

I deeply appreciate Wei-Han Lien, Liang Yin, and Xin He from Tenstorrent for offering me internship opportunities during the summers of 2022 and 2023. These experiences significantly deepened my understanding of the field and were instrumental in shaping crucial early concepts of this dissertation.

Finally, I am profoundly grateful to my family for their unwavering support, patience, and encouragement throughout this journey. I thank my parents for giving me the name Shibo, which reflects their hope and aspiration that I would earn a doctoral degree. This has served as my ultimate inspiration, and their unwavering belief in me has made all the difference.

PREFACE

This thesis represents not only the culmination of years of academic research but also a deeply personal journey toward realizing a lifelong aspiration. Pursuing a doctoral degree has been a longstanding ambition of mine, and completing this thesis fulfills that dream.

My research journey began in computer security, but gradually, my interests shifted toward computer architecture, greatly inspired by the pioneering work of my advisor, Professor Todd Austin. Over time, I discovered the immense potential that working at the architectural level offers, particularly the ability to approach complex problems from diverse perspectives. My transition into this field was sparked by an early, challenging experience collaborating with my friend and peer, Alex Kisil, on a course project. We sought to integrate a new security feature into a widely used open-source CPU design implemented in Chisel. However, we quickly realized the existing codebase was exceptionally difficult to comprehend, let alone modify. Driven by this frustration and informed by our combined experience with Chisel and SystemVerilog, I developed Twine (Chapter 3), a tool designed specifically to simplify the generation of heterogeneous hardware designs.

As my research advanced, I encountered further challenges in efficiently integrating heterogeneous components, prompting the development of novel architectural solutions. Zipper (Chapter 4) was created to address the communication overhead between host systems and coprocessors——a problem I confronted directly while contributing to the Sequestered Encryption project, led by my then-labmate Lauren Biernacki. Later, during an internship at Tenstorrent, my work on MPSoC interconnects highlighted issues related to resource contention and interference among compute hosts. These observations informed the development of Overpass (Chapter 5), an innovative design that aims to enhance resource sharing and efficiency.

Throughout my PhD, my work has consistently emphasized the creation of tools, architectures, and engineering principles that facilitate high-performance, high-quality hardware design without compromising flexibility and ease of use. As computer systems continue to evolve into more heterogeneous and complex architectures, ensuring their accessibility becomes increasingly critical, particularly given that human cognitive abilities are not advancing at a comparable pace. While AI may eventually become instrumental in designing superior systems, this vision remains speculative at the time of writing.

It is my sincere hope that this thesis makes a significant contribution to the field of heterogeneous hardware design and computer architecture and that it inspires and guides future research efforts.

Shibo Chen

Ann Arbor, Michigan, USA

April 2025

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

LIST OF PROGRAMS

LIST OF ACRONYMS

AES Advanced Encryption Standard

AI Artificial Intelligence

ALM Adaptive Logic Modules

ALU Arithmetic Logic Unit

ASIC Application-Specific Integrated Circuit

AXI Advanced eXtensible Interface

BRAM Block RAM

CCI-P Core Cache Interface

CCD Core Complex Die

CPU Central Processing Unit

CXL Compute Express Link

DCOH Device Coherency Agent

DSP Digital Signal Processing

DSL Domain-Specific Language

FIFO First-In, First-Out

FPGA Field Programmable Gate Arrays

FPU Floating-Point Unit

GCP Graphics Processing Unit (GPU) Command Processor

GEMM General Matrix Multiply

GPU Graphics Processing Unit

HDL Hardware Design Language

HLS High-Level Synthesis Language

HPC High-Performance Computing

IOD Input Output Die

I/O Input-Output

IP Intellectual Property

ISA Instruction Set Architecture

LLM Large Language Model

LRU Least Recently Used

MBA Memory Bandwidth Allocation

MMIO Memory Mapped Input Output

MMPerf Single CPU Core Matrix Multiplication Benchmarks

NoC Network-on-Chip

NPB The NAS Parallel Benchmarks

PCIe Peripheral Component Interconnect Express

QoS Quality-of-Service

QPI QuickPath Interconnect

RISC Reduced Instruction Set Computing

RLP Request-Level Parallelism

RR Round-Robin

RTL Register Transfer Level

SE Sequestered Encryption

SKU Stock Keeping Unit

SoC System-on-Chip

SQL Structured Query Language

SST The Structural Simulation Toolkit

STL Standard Template Library

TPU Tensor Processing Unit

TSMC Taiwan Semiconductor Manufacturing Company

UCIe Universal Chiplet Interconnect Express

UPI UltraPath Interconnect

VC Virtual Channel

VHDL VHSIC Hardware Description Language

XBar Crossbar

ABSTRACT

With the stagnation of Moore's Law and the breakdown of Dennard Scaling, hardware designers are increasingly turning to heterogeneous architectures to achieve higher performance and energy efficiency. Heterogeneous design composes complete systems from specialized, modular components optimized for specific applications or markets. This modular approach contrasts with traditional homogeneous architectures, which use identical processors to handle diverse workloads. By leveraging the unique strengths of each component, heterogeneous systems not only outperform their homogeneous counterparts but also enable faster time-to-market across a broad range of use cases. Such specialization has been key to advances in low-power embedded systems and data-intensive machine learning applications.

However, these benefits come at a cost. Heterogeneous systems incur greater non-recurring engineering (NRE) effort, higher communication overhead, and increased memory bandwidth contention—factors that limit scalability and adoption. A primary contributor is inefficient communication among heterogeneous components. Specifically, mismatched I/O interfaces hinder reusability and drive up NRE; poor latency tolerance creates performance bottlenecks; and inadequate resource allocation leads to contention and interference. These challenges arise at multiple stages of system design and collectively slow or block deployment in real-world scenarios.

This dissertation presents minimally invasive solutions that streamline heterogeneous system design by directly addressing these communication challenges. First, it introduces Twine, a design language for heterogeneous design that standardizes communication interfaces and automates control logic generation. Twine reduces design specification size by 3×, enhancing reusability and reducing engineering overhead. Second, it proposes Zipper, a set of latency-tolerant bus optimizations that enable systems to tolerate microsecond-level delays without drastic redesign. By exploiting the temporal locality and parallelism that exist in applications, Zipper delivers up to 8× performance gains. Finally, it introduces Overpass, a flexible interconnect system with distributed resource allocation that optimizes bandwidth utilization. Overpass increases system performance by 35%, enabling efficient communication across diverse components.

Together, Twine, Zipper, and Overpass complement each other, forming a cohesive framework to help developers address the core communication bottlenecks of heterogeneous hardware at various design stages. These solutions help developers extract greater performance from their designs

while conserving valuable engineering effort. By directly addressing the fundamental barriers to adoption, this dissertation advances the practicality and effectiveness of heterogeneous system design and lays the groundwork for broader deployment and continued innovation in the field.

CHAPTER 1

Introduction

Heterogeneous hardware architectures integrate diverse processing elements, such as CPUs, GPUs, FPGAs, and specialized accelerators, into a unified system. This multi-faceted design allows each component to execute the tasks it is best suited for, thereby enhancing overall performance, energy efficiency, and responsiveness in handling diverse and demanding applications. This is in contrast to traditional homogeneous architectures, where all applications are deployed to identical processing components. Moreover, by permitting components to be produced separately and sourced from different vendors, heterogeneous systems can be more cost-effective and flexible than their homogeneous counterparts, which typically rely on identical processing elements manufactured using a single process node.

Despite their clear advantages, developing heterogeneous systems remains a formidable challenge. Manually generating design candidates for extensive design space exploration is both laborious and inefficient. Furthermore, performance is frequently hampered by communication overhead between diverse components and by resource contention. At the root of these issues lies inadequate communication and a lack of effective coordination among heterogeneous elements.

To address these challenges, this dissertation investigates and enhances communication channels between heterogeneous components. It introduces a suite of tools and methodologies designed to automate key design processes, reduce engineering overhead, and ultimately deliver higher-performing hardware systems. Importantly, all solutions proposed require only minimal modifications to existing hardware designs while incurring only a trivial area overhead.

In the rest of this section, we first explore the emerging trends that are driving the adoption of heterogeneous architectures, providing a detailed account of the associated design challenges. We then introduce the solutions developed in this work and outline the organization and key contributions of this dissertation.

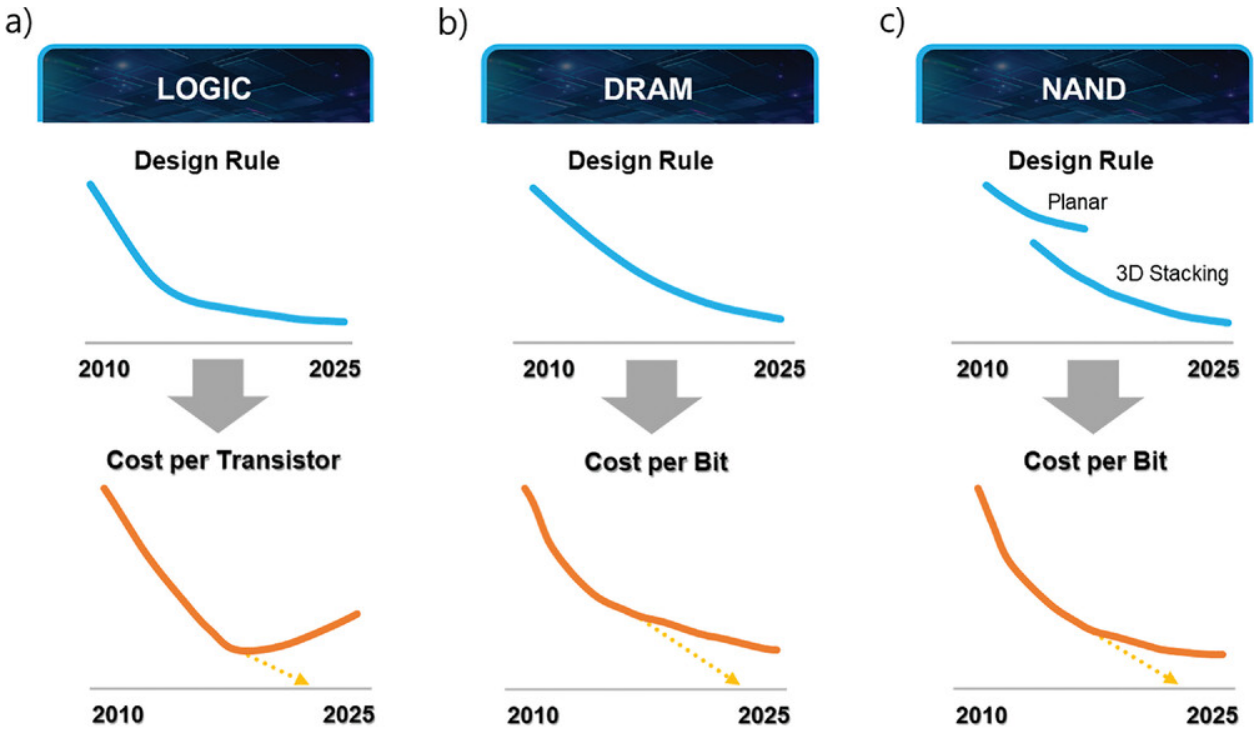


Figure 1.1: The manufacturing cost trend for various integrated circuits technologies and components over the years by ES Jung [83]. The solid line depicts the recorded data from 2010 to 2015. The dotted line is the projection of Moore's Law.

1.1 The End of Moore's Law and Dennard Scaling

Gordon Moore, co-founder of Intel, co-authored a seminal projection in 1965, later known as **Moore's Law** [111], which stated that the number of transistors within a given unit area on a microchip would double every two years while manufacturing costs for the same unit area would remain constant. His colleague, David House, refined this prediction by suggesting that transistor counts would double approximately every 18 months——a formulation widely regarded as the practical speed of Moore's Law [85].

In 1974, Robert Dennard introduced another critical observation, later termed **Dennard Scaling** [45] or MOSFET Scaling, which posited that as transistors became smaller and their density increased, the power consumption per unit area would remain constant. This principle was instrumental in driving the continuous advancement of increasingly complex microprocessors, as engineers were able to integrate more transistors while achieving higher operating frequencies without exceeding thermal constraints.

However, these trends have largely stagnated. Figure 1.2 shows that the power required per nm^2 increases as the process node shrinks rather than staying constant as predicted by Dennard Scaling. Figure 1.1 illustrates the deceleration and eventual breakdown of Moore's Law: not only do the

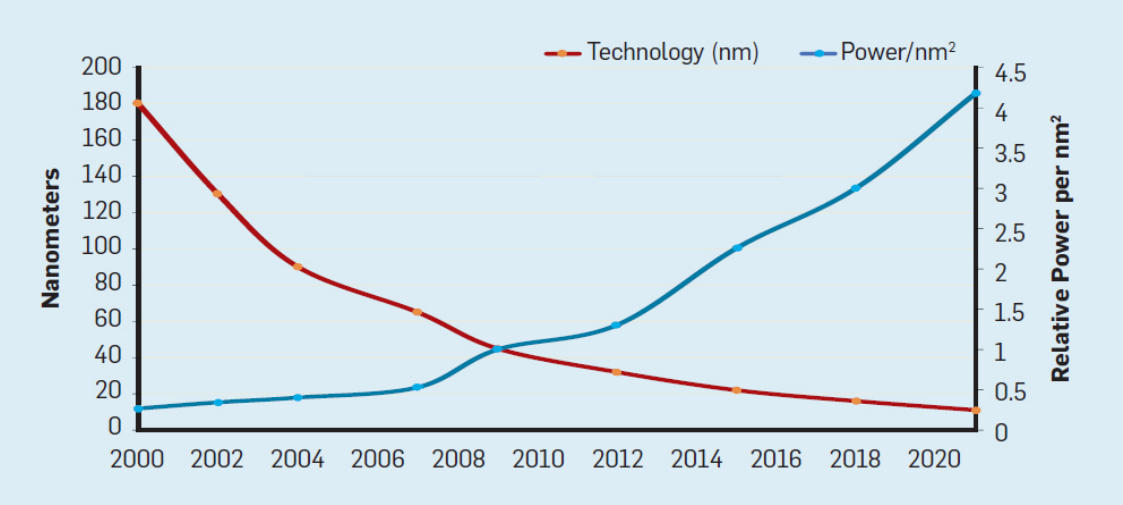


Figure 1.2: The power consumed per nm^2 increases as the process node thrinks by Hennessy and Patterson [61].

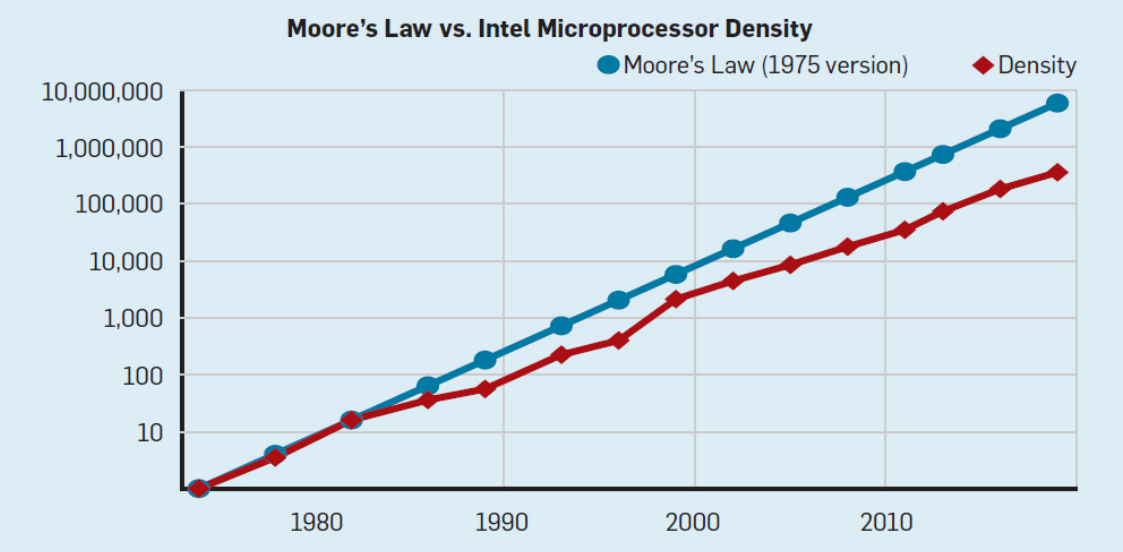


Figure 1.3: Transistors on Intel processors versus Moore's Law by year by Hennessy and Patterson [61].

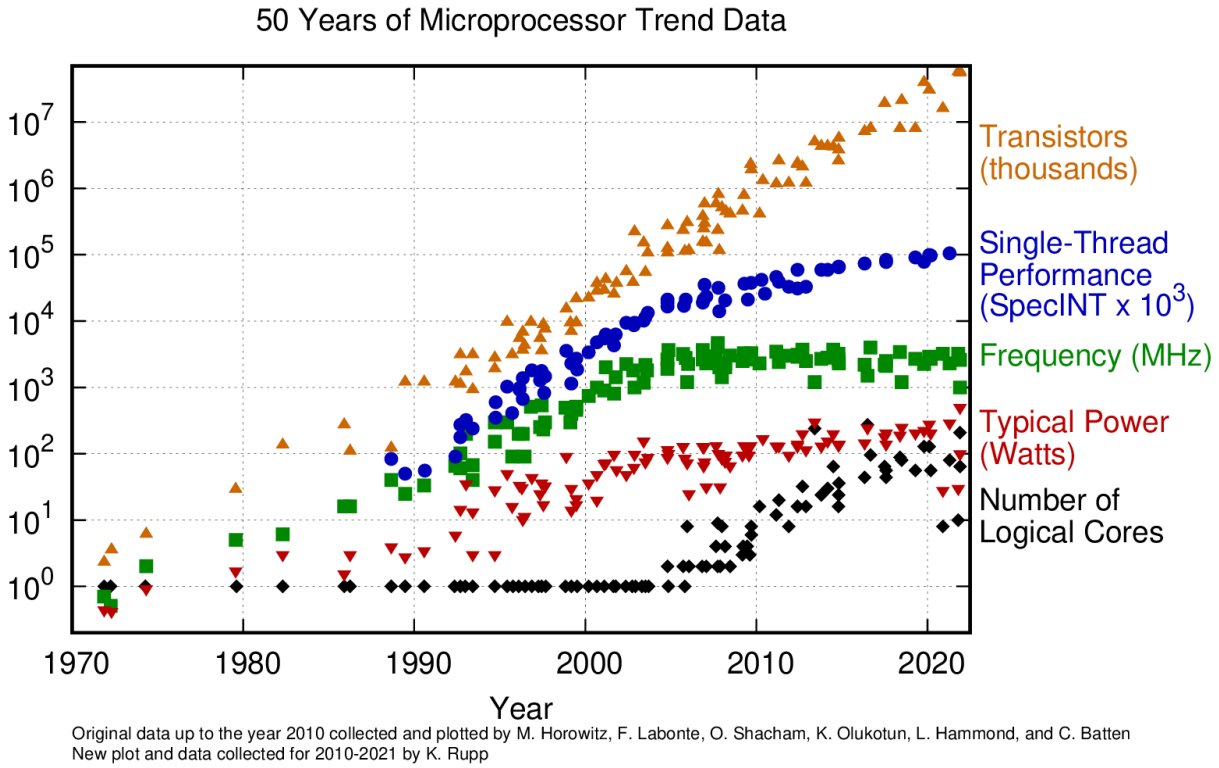


Figure 1.4: The trend of microprocessor development collected and managed by K. Rupp, et al. [135].

design rule sizes shrink slowly, but also, the cost per transistor decreases slowly and deviates from the projection of Moore's Law. The cost of logic transistors even goes up as the process node shrinks. Figure 1.3 reinforces such a trend, showing that the number of transistors on their most recent chips lags behind the projection of Moore's Law.

The ripple of such stagnation has caused real disruption in commercial products. Rupp, et al.[135] collected the data on microprocessor features and parameters of the past 50 years, illustrated in Figure 1.4. The stagnation of single-thread performance, processor frequency, and power envelope is a direct victim of the breakdown of Moore's Law and Dennard Scaling. Hennessy and Patterson arrived at a similar conclusion with a similar trend shown in Figure 1.5.

At the time of writing, Taiwan Semiconductor Manufacturing Company (TSMC), the lead foundry company with the most advanced manufacturing process, has not yet mass-produced its next-generation 2nm process more than three years after introducing its 3nm process. Furthermore, the projected 2nm process will cost more than a 1.5x price increase for a mere 1.15x density improvement [137].

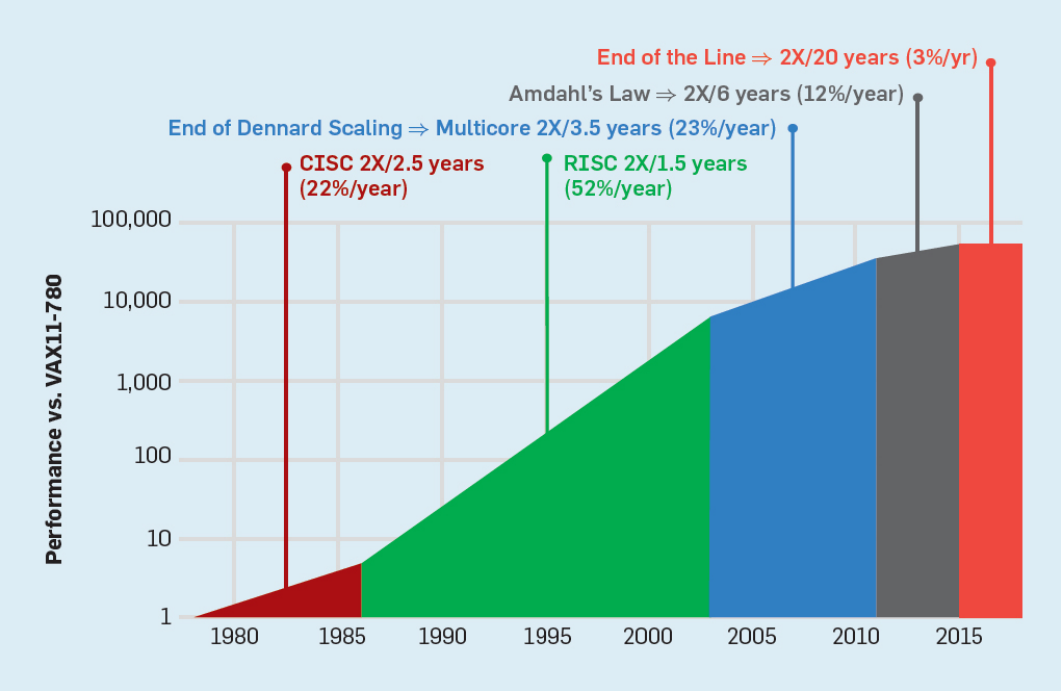


Figure 1.5: Growth of computer program using integer programs (SPECCPU) by Hennessy and Patterson [61].

This slowdown has profound implications for the semiconductor design industry. Historically, engineers could leverage process advancements to reduce power consumption while maintaining existing designs or achieve higher performance by increasing design complexity without exceeding thermal and die area constraints. However, these assumptions no longer hold. Designers now face the challenge of downscaling their designs to meet power efficiency requirements for edge devices while simultaneously producing larger, more power-intensive architectures to sustain performance gains, both of which introduce significant power and yield challenges.

1.2 Increasingly Diverse Computing Needs

The rapid diversification of computing needs contrasts with the stagnation in high-performance computing advancements. Over the past two decades, the computational requirements for training machine learning models have quadrupled annually, as depicted in Figure 1.6. For instance, xAI deploys over 200,000 GPUs to train its largest large language model, Grok 3 [63], which demands more than 400 billion petaFLOPs. This exponential growth in demand underscores an urgent need for increased computational power.

Conversely, developers are also pushing the boundaries on the low-power end of the spectrum. Emerging ultra-low-power devices are being designed for applications ranging from human im-

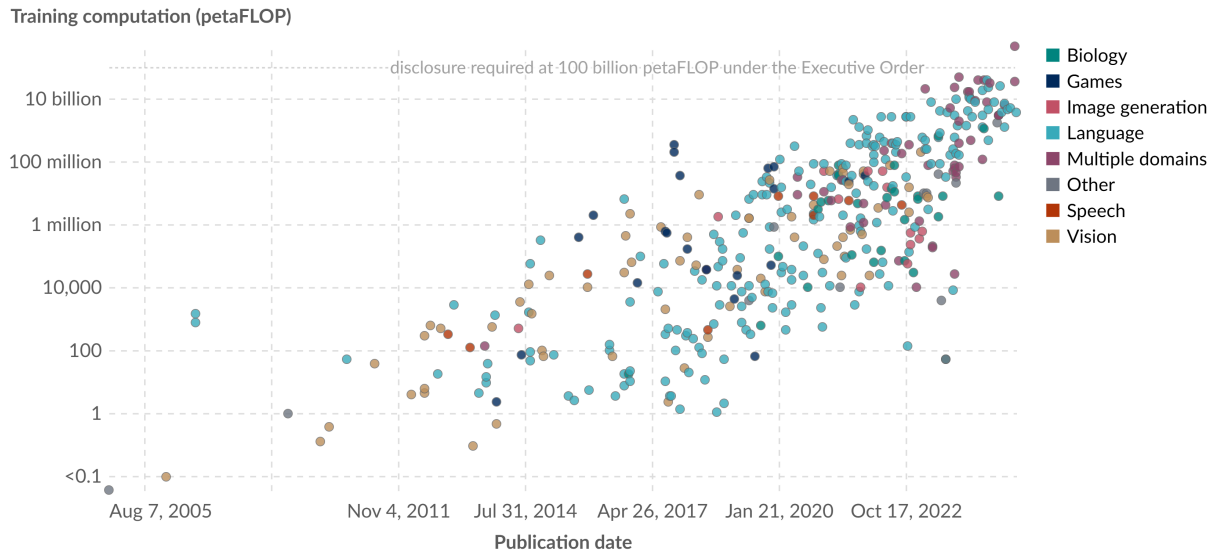


Figure 1.6: The compute needed to train a machine learning model over the years by domain[53].

plants to outdoor environments where external power is scarce [106, 78]. These devices must operate efficiently, drawing only tens of microwatts—and even as little as picowatts when idle.

While performance and power remain the most frequently discussed metrics, other emerging demands are also fueling innovations, necessitating novel architectures and designs. OpenTitan [119] and Sequestered Encryption (SE)[23] are two architectures that emphasize security and privacy, whereas Tenstorrent's Tensix Neo architecture[152] targets reducing AI inference costs by exclusively using commodity components.

Collectively, these developments demonstrate that a single, uniform architectural approach can no longer adequately address the increasingly diverse and specialized computing demands of today's technological landscape, driving the inevitable shift toward heterogeneous architectures.

1.3 Emergence and Potential of Heterogeneous Hardware

> "What we're really trying to do is have heterogeneous systems really become the foundation of our computing going forward. And that's the idea that you make every processor and every accelerator a peer processor. "
>
> Lisa Su, CEO of Advanced Micro Devices, Inc., 2013

In response to the emerging diverse demands of computing, researchers and companies started

to study and design heterogeneous architectures in the post-Moore's Law and Dennard Scaling era.

Traditionally, hardware architectures have been homogeneous, deploying diverse workloads onto identical cores built on the same microarchitecture. Developers scale these systems by replicating the same components until they meet the performance targets. This approach simplifies design and reduces engineering effort through extensive reuse. However, using powerful cores for low-intensity tasks wastes power and resources. Furthermore, homogeneous systems are ultimately limited by the serial portions of computation, a bottleneck that is worsened by the stagnation of single-thread performance gains.

In contrast, heterogeneous architectures account for the diverse and evolving nature of computational demands. Rather than replicating identical processing units, they integrate a range of specialized components, each optimized for specific workload characteristics or market requirements. This targeted approach enables the elimination of unnecessary general-purpose logic, resulting in improved power and area efficiency relative to homogeneous systems. The resources saved can be reallocated to increase the density of compute units on a single chip, thereby enhancing parallel processing capabilities. Moreover, for latency-critical or serial workloads, heterogeneous systems can incorporate dedicated circuits that execute tasks directly, avoiding the inefficiencies associated with general-purpose instruction pipelines and reducing overall latency.

Heterogeneous designs primarily manifest in two forms:

- **Reconfigurable Designs**: These architectures can adapt to various market segments or computing requirements by upscaling or downscaling. By adopting modular designs, developers can quickly generate new designs during the architectural exploration phase and rapidly produce new Stock Keeping Unit (SKU)s. Additionally, by employing components like Field Programmable Gate Arrays (FPGA)s, systems can be tailored post-manufacturing for specific tasks, offering flexibility and prolonged hardware relevance. A notable example is Google's Tensor Processing Unit (TPU), which is implemented using systolic array architecture. The TPU lineup includes high-performance cloud designs and smaller edge designs downscaled for low-power devices such as smartphones.

- **Hybrid Architectures**: These systems combine different types of processors, such as Central Processing Unit (CPU)s, GPUs, and specialized accelerators, to handle complex, multitasking environments efficiently. By assigning workloads to the most suitable processing units, these architectures enhance performance and energy efficiency. A notable in this form is ARM big.Little [16] architecture, which pairs high-performance cores for demanding tasks with power-efficient cores for background activities to optimize workload distribution. Modern smartphone System-on-Chip (SoC) designs [15, 71] also exemplify this design philosophy. Modern mobile SoCs integrate CPU, GPU, and Digital Signal Processing (DSP)s

in a single package where CPU is responsible for control logic and sequential tasks, GPU is responsible for image rendering, and DSPs are for processing audio and video related data.

In addition to the well-recognized advantages of enhanced performance and improved power efficiency resulting from division of labor, heterogeneous design offers the often-overlooked benefits of reduced manufacturing costs and increased supply chain resilience. By allowing components to be manufactured using the most cost-effective process nodes that meet design requirements, companies can avoid the substantial expenses associated with premium fabrication technologies.

A prime example of this approach is AMD's EPYC "Genoa" processors [12], which utilize a chiplet-based architecture. In this design, the processor is segmented into Core Complex Die (CCD)s and an Input Output Die (IOD). Each CCD is fabricated using TSMC's 5nm process technology, optimizing performance, while the IOD employs TSMC's 7nm process, balancing cost and functionality.

This strategic partitioning not only reduces manufacturing costs but also enhances supply chain resilience. By decoupling the production of different components, manufacturers can more readily adapt to supply chain disruptions. If certain components become unavailable, alternative solutions can be sourced or produced using different process nodes without necessitating a complete redesign. This flexibility allows for quicker responses to unforeseen challenges, thereby maintaining production continuity and market competitiveness.

1.4 Zero to One: Building Heterogeneous Hardware from Scratch

To successfully design and manage the development of a large system-level heterogeneous chip, developers typically decompose the chip into smaller parts or subsystems. Figure 1.7 illustrates how a heterogeneous chip is composed hierarchically: from low-level (module-level) components to the final product—the chip itself. Notably, heterogeneity manifests at different levels of the design hierarchy:

- **Modules:** Modules sit just above transistors and logic gates and serve as the minimal functional building blocks of a design. To support heterogeneous architectures, modules must be reconfigurable to meet varying requirements for timing, throughput, area, and power. Common reconfigurable parameters include buffer sizes, pipeline depths, and vectorization degrees. Designers also have access to a diverse library of modules, especially for arithmetic operations with different data formats (*e.g.*, integers, floating-point, brain floating-point [26], *etc.*). Representative examples at this level include Arithmetic Logic Unit (ALU)s, Floating-Point Unit (FPU)s, and simpler components like First-In, First-Out (FIFO) buffers.

- **Intellectual Property (IP)s:** IPs are typically the smallest reusable or licensable design units, composed of multiple modules and conforming to standard interfaces for integration. Designing a high-quality IP involves selecting the optimal modules to fulfill functional requirements and tuning their parameters to meet performance, cost, area, and power constraints. Examples include compute IPs such as CPU cores (with varying issue widths), AI accelerators, and memory controllers.

- **Dies/Chiplets:** Dies or chiplets integrate multiple IPs through interconnects like crossbars or more scalable solutions such as Network-on-Chip (NoC)s. A single die may contain a diverse set of IPs or multiple instances of the same IP. Typically, compute dies are fabricated using advanced process nodes to maximize performance and energy efficiency, while memory and I/O dies are manufactured separately using more cost-effective technologies.

- **Chips:** With chiplets available, developers can mix and match them to build larger systems using advanced packaging and interconnect techniques. For instance, NVIDIA's Grace-Hopper superchip [116] combines a CPU chiplet with a GPU chiplet, and its Blackwell AI accelerator [117] integrates two GPU chiplets. Generally, a fully functional system-level chip would include compute dies that shared I/O chiplets or memory controller chiplets for best cost efficiency.

Hierarchical decomposition enables developers to balance the need for performance and flexibility with the challenge of managing complexity. Transitioning between levels in the design hierarchy involves hundreds of design decisions. Developers must also contend with the overhead and intricacies of integrating heterogeneous components.

1.5 Challenges on the Road to Heterogeneity

While heterogeneous designs hold significant promise for addressing evolving computational demands, designing and assembling a high-performance heterogeneous system presents considerable challenges.

Figure 1.8 illustrates an example of a heterogeneous system architecture. This system includes multiple compute dies interconnected to a shared pool of memory and I/O resources. Each compute die integrates one or more compute agents, such as CPUs, GPUs, FPGAs, and other specialized accelerators, designed to handle specific computational workloads efficiently.

The first key challenge, indicated by ❶, is generating comprehensive designs to explore the vast design space and identify optimal configurations thoroughly. Modern designs typically involve hundreds or even thousands of parameters or design options. The designs may also need to add or

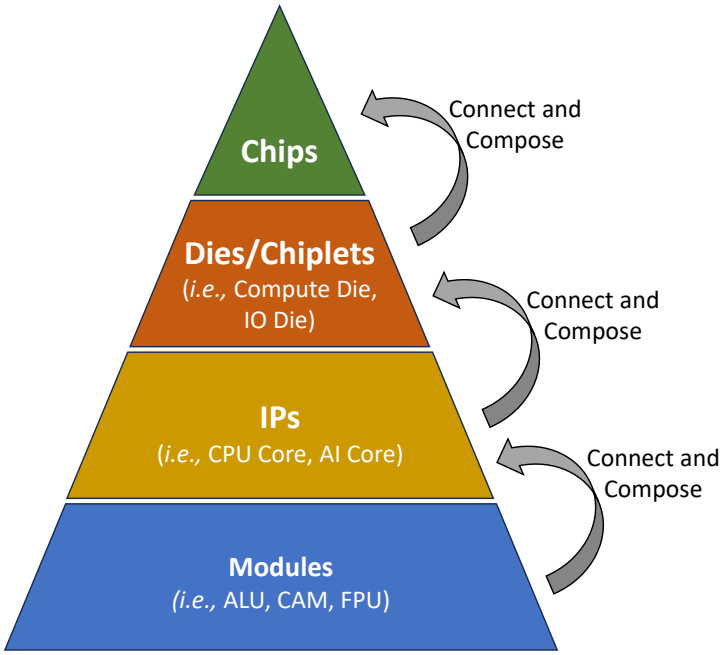


Figure 1.7: Different levels of component in heterogeneous designs. The higher-level components are composed of lower-level components.

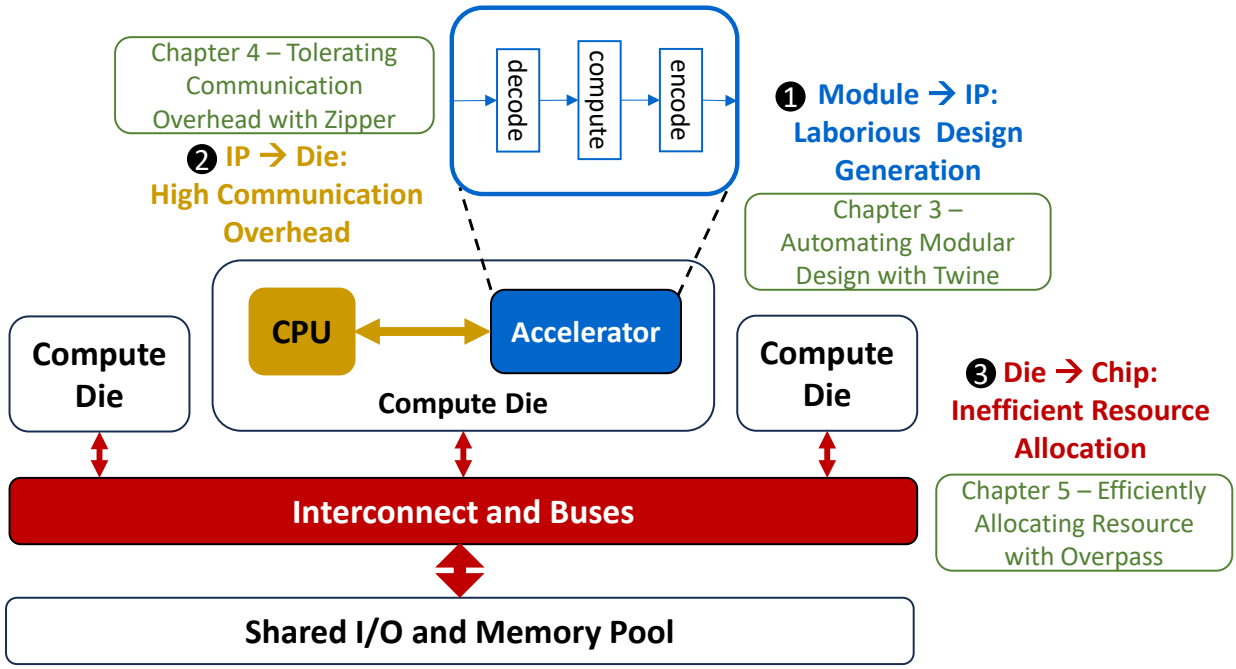


Figure 1.8: The challenges developers face when designing and assembling a heterogeneous hardware design and the solution to each challenge presented in this dissertation, demonstrated with an example heterogeneous system.

subtract one or more pipeline stages and functions to meet the unique demands. The combinatorial explosion of design possibilities makes it exhaustively generate these designs. While reusable and modular components offer promising improvements to mitigate the engineering overhead of generating such designs, integrating these components into functional systems remains laborious, reduces productivity, and thus incurs substantial engineering costs.

The second challenge arises after accelerator designs have been realized, marked as ❷ in the figure. To handle complex tasks efficiently, it often requires frequent inter-component communication between the hosts and accelerators. However, communication latency between CPUs and accelerators is orders of magnitude greater than what can be tolerated by modern computing architectures and software techniques. This substantial latency overhead significantly hampers heterogeneous system performance and, in extreme cases, precludes the deployment of specific designs altogether.

The third significant challenge emerges when integrating multiple compute agents that concurrently access shared external resources, indicated by ❸. Resource contention in heterogeneous systems negatively impacts overall system performance, leading to performance degradation. Conventional resource allocation methods predominantly assume a homogeneous system environment, rendering them inefficient when applied to heterogeneous contexts. A heterogeneous-aware, optimized resource allocation strategy is crucial for realizing the full potential of new designs.

Chapters 3, 4, and 5 provide an in-depth discussion on the root causes of these challenges, along with a detailed analysis of the shortcomings in current industry standards and state-of-the-art approaches.

1.6 Streamlining Heterogeneous Hardware Design

> ”Good design is obvious. Great design is transparent.”
>
> Joe Sparano, Instructional and Graphic Designer

This dissertation aims to streamline high-performance heterogeneous designs by addressing each of the challenges described in §1.5. Collectively, this dissertation assists developers in constructing high-performance heterogeneous systems, ranging from Register Transfer Level (RTL) modules and interconnected host-accelerator setups to intricate designs involving multiple agents sharing resources. Each solution is described, studied, evaluated, and thoroughly discussed in Chapters 3, 4, and 5, respectively.

Design Principle. The solutions presented in this dissertation are designed to be straightforward to implement and transparent to both hardware components and application software. Crucially, these solutions require minimal alterations to existing user designs, which reduces the adoption barrier and minimizes friction between proposed approaches and current practices. Solutions developed under this principle possess the highest likelihood of adapting seamlessly into the continuously evolving landscape of heterogeneous designs.

Chapter 3 explores the generation of RTL designs intended for design space exploration and evaluation. It identifies that challenges in reusing modules arise primarily from fragmented design assumptions, which cause discrepancies in the control interfaces of modules. Consequently, engineers spend excessive time deciphering control logic and creating glue logic for various scenarios. To mitigate these challenges, this dissertation proposes standardizing common control behaviors into four unified interfaces, enabling fully automated integration of diverse modules without manual intervention. We introduce Twine, a language embodying these insights, and demonstrate its effectiveness. Our evaluations show that Twine significantly enhances productivity by reducing code changes across designs. Early adopter experiments further confirm that Twine is intuitive, easy to learn, and user-friendly.

Chapter 4 addresses the substantial communication overhead between hosts and accelerators. Existing solutions typically involve hardware modifications or compiler adjustments to mask communication latency, both of which entail significant engineering effort. This dissertation uncovers previously untapped opportunities for exploiting locality and parallelism within applications without requiring intrusive hardware or compiler modifications. We validate these findings through two case studies and introduce Zipper——a set of optimizations utilizing only software libraries and drop-in hardware infrastructure. Zipper achieves up to an 8x performance enhancement while requiring minimal engineering effort and hardware changes.

Chapter 5 investigates performance degradation caused by resource contention and the inherent complexity of optimizing bandwidth allocation. We argue that the interconnect——the central component that connects all agents——should be responsible for monitoring system performance and optimally managing bandwidth distribution. To this end, we present Overpass, a flexible interconnect design that dynamically tracks bandwidth usage and agent performance metrics. Overpass uses a distributed management mechanism that is adaptable to various network topologies, eliminating the need for additional engineering steps. Evaluation on an eight-agent system demonstrates that Overpass yields a 35% overall performance improvement.

Figure 1.9 offers a high-level overview of the heterogeneous system design landscape by mapping key design attributes, such as productivity, quality, and cost, against primary stages of the hardware development process, from module generation to system-level integration. The green-

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="3">Design Stages</td></tr>
<tr><td>Module → IP</td><td>IP → Die</td><td>Die → Chip</td></tr>
<tr><td rowspan="7">Design Attributes</td><td>Productivity</td><td rowspan="3">Generating high-performance accelerator: Twine</td><td colspan="2" rowspan="3">Addressing high latency overhead: Zipper
Addressing bandwidth contention: Overpass</td></tr>
<tr><td>Quality</td></tr>
<tr><td>Cost</td></tr>
<tr><td>Programmability</td><td></td><td colspan="2"></td></tr>
<tr><td>Security</td><td></td><td></td><td></td></tr>
<tr><td>Reliability</td><td></td><td></td><td></td></tr>
<tr><td>Verifiability</td><td></td><td></td><td></td></tr>
</table>

Figure 1.9: The table shows various design attributes and design stages of heterogeneous systems. Green cells mark the areas that are addressed in this dissertation; gray cells are out of the scope of this dissertation.

highlighted cells indicate the areas directly addressed in this dissertation, reflecting a targeted effort to improve design productivity at the module level (via Twine), reduce communication overhead during IP integration (via Zipper), and optimize resource allocation among IP blocks or chiplets(via Overpass). All three solutions are designed to be transparent to programmers, require minimal changes to existing designs, and are portable across platforms—thereby enhancing productivity, preserving high programmability, and reducing engineering and manufacturing costs.

Although Zipper and Overpass are each presented in the context of a specific design stage, both address recurring challenges that span multiple phases of the design process. Their underlying techniques are broadly applicable and require little to no adaptation when deployed across different stages. Overall, the figure illustrates the complementary nature of Twine, Zipper, and Overpass, each resolving a distinct bottleneck while collectively streamlining the end-to-end design flow.

A recurring insight throughout this dissertation is the observation that friction in heterogeneous hardware design primarily arises from the loss of design information between compartmentalized components and during integration phases. Defining clear interfaces and semantics to share or reconstruct this information significantly enhances both design productivity and system quality. Complexity in generating heterogeneous designs is substantially reduced by sharing standardized control information among modular components. Communication latency is mitigated by capitalizing on shared locality and parallelism information between hosts and accelerators. Resource

allocation is optimized by enabling components to communicate performance metrics within the broader system. This dissertation identifies these information gaps and provides concrete solutions, enabling smoother design and assembly processes tailored to unique user requirements.

> **General Applicability.** Although the solutions in this dissertation are primarily motivated by the challenges of heterogeneous hardware design, they are not restricted to heterogeneous systems. Twine, Zipper, and Overpass address fundamental design, communication, and resource allocation bottlenecks that are also prevalent in large-scale homogeneous architectures. For example, Twine's modular interface abstractions improve design reuse and integration across any complex system. Zipper's latency-tolerant mechanisms are effective in scenarios with host-device communication bottlenecks, regardless of component heterogeneity. Overpass's decentralized bandwidth arbitration can enhance performance in homogeneous systems with shared interconnects and dynamic resource contention. As such, these tools and methodologies offer value beyond heterogeneous contexts, making them broadly applicable to modern system design.

Ultimately, the solutions presented herein help developers attain high performance with minimal engineering overhead. The findings and detailed case studies provided in this dissertation lay the groundwork and inspire future research and advancements in heterogeneous hardware design.

1.7 Dissertation Contributions and Organization

Through the following contributions, this dissertation provides innovative tools, techniques, and architectural enhancements that significantly streamline the complex design process of high-performance heterogeneous hardware. By addressing critical challenges in heterogeneous design, the presented work reduces development complexity, automates essential design tasks, and unlocks substantial performance improvements. Additionally, the dissertation shows that heterogeneous systems require distinct design principles and novel solutions to harness their full performance potential effectively.

The specific contributions of this dissertation include:

- Identifying and characterizing the complexity involved in coordinating control logic during the design space exploration phase of heterogeneous hardware design.
- Analyzing and profiling latency-sensitive applications whose performance significantly depends on effective communication between host processors and hardware accelerators.

- Investigating deficiencies in existing communication resource allocation approaches that impede performance in heterogeneous hardware systems.
- Proposing Twine HDL [33], a new design extension that standardizes control interfaces and automates control logic generation through modular, reusable components. Twine accelerates design space exploration, a crucial step in designing heterogeneous systems.
- Evaluating Twine against industry-standard design languages, demonstrating that automating control logic dramatically reduces engineering overhead and enhances developer productivity.
- Identifying critical optimization opportunities that can exploit inherent application features to improve data locality and parallelism within heterogeneous systems significantly.
- Introducing Zipper [34], a set of latency-tolerant optimizations targeting communication buses to mitigate latency overhead without altering the underlying physical channel characteristics.
- Demonstrating Zipper's effectiveness through two comprehensive case studies, achieving performance improvements of up to 8x with less than 5% additional area overhead, thereby highlighting substantial opportunities for latency optimization based on application-specific characteristics.
- Expanding the practical design space of heterogeneous systems through the optimizations provided by Zipper, facilitating previously unattainable system setups and architecture designs.
- Proposing Overpass, an advanced flexible interconnect architecture featuring distributed bandwidth allocation and prefetch throttling mechanisms designed to optimize overall system throughput.
- Validating Overpass through empirical evaluation in an eight-agent system configuration, resulting in a 35% improvement in overall system performance.
- Providing insights into the decision-making processes within Overpass, offering foundational guidance and inspiration for future research in heterogeneous system interconnect design.
- Summarizing the limitations of this dissertation and presenting potential future directions of heterogeneous system research.

Dissertation Organization:

In **Chapter 2**, we provide the foundational background necessary for understanding the contributions of this dissertation, covering essential tools, technologies, protocols, target architectures, and relevant applications. This background sets the stage for the solutions presented in subsequent chapters.

In **Chapter 3**, we introduce Twine [33], a design language developed to enable the rapid generation of heterogeneous hardware designs, thereby facilitating effective design space exploration.

Chapter 4 presents Zipper [34], a series of latency-tolerant optimizations for high-performance buses. These optimizations are designed to significantly reduce communication overhead, enabling more efficient integration of heterogeneous components.

In **Chapter 5**, we describe Overpass, a flexible interconnect solution developed to manage resource allocation challenges within heterogeneous systems efficiently.

Chapter 6 compares the related works with the solutions presented in Chapters 3, 4, and 5 and underpins the novelty and significance of the solutions proposed in this dissertation.

In **Chapter 7**, we present and discuss potential future directions to foster the further development of heterogeneous designs.

Finally, **Chapter 8** concludes this dissertation with a summary of contributions to the heterogeneous hardware design process and practice.

CHAPTER 2

Background: Design Languages, Technologies, Protocols, and Applications of Heterogeneous Design

This chapter serves as a practical guide to foundational concepts and tools in heterogeneous hardware design and further motivates the importance of improving the heterogeneous design process.[1] We first discuss the design languages and tools developers use to specify hardware designs. Next, we provide an overview of technologies that enable heterogeneous designs, followed by an exploration of the communication protocols and conventions used by heterogeneous components during their design and deployment phases. Finally, we present selected applications studied or evaluated throughout this dissertation.

2.1 Design Languages and Tools

To build any hardware system, the first step is to translate a functional idea into an actual design. Developers use various design languages to translate functional specifications into designs that can be synthesized into hardware. Some languages explicitly describe hardware cells, gates, and wires, such as SystemVerilog [68]. Others rely on accompanying tools to interpret and convert high-level descriptions into synthesizable designs, like Vivado HLS [162]. This section explains the major language categories engineers rely on, their benefits, and where they fall short, especially in heterogeneous contexts.

2.1.1 HDL

For decades, digital circuits have been designed using HDLs such as SystemVerilog [68] and VHDL [67]. These languages allow engineers to define hardware at the RTL, explicitly capturing concurrency and timing necessary for accurate simulation and synthesis, and remain widely

[1] Given the vast scope of the heterogeneous design landscape, this chapter focuses specifically on information relevant to the dissertation's core topics rather than exhaustively covering all existing works or technologies.

used in industry. However, traditional HDLs offer limited abstraction, making code reuse challenging and resulting in verbose, repetitive code. Additionally, developers must manually manage low-level details such as clock/reset handling, signal connections, and testbench separation. Code reuse in HDLs is minimal. For complex or rapidly evolving designs, HDLs often become productivity bottlenecks.

2.1.2 HLS

HLS languages, such as VivadoHLS [162], provide designers with a more software-like development environment. Instead of writing detailed RTL, engineers describe the intended hardware behavior using high-level languages, typically C or C++. The HLS tools then generate corresponding hardware logic, enabling designers to concentrate on algorithmic concerns while the tool handles scheduling, parallelism, and resource allocation. Nevertheless, HLS-generated designs often achieve lower performance compared to hand-optimized HDL implementations. HLSs also lack sufficient expressiveness for certain hardware designs, limiting their applicability in performance-critical systems.

2.1.3 DSL

DSLs, such as Halide [129] and Taco [88], offer specialized abstractions explicitly designed to accelerate computationally intensive tasks. DSLs separate computational algorithms from optimization strategies, allowing engineers to quickly generate hardware designs and explore different performance trade-offs. However, the narrow scope of DSLs limits their applicability. Tasks slightly outside their targeted domain often require complex workarounds or integration with other tools. Furthermore, deploying DSL-generated designs within larger heterogeneous systems can present significant challenges, particularly when combining multiple specialized DSLs or interfacing with conventional RTL designs.

2.2 Communication Technologies that Enable Heterogeneity

Modern heterogeneous systems are more than just a collection of processing elements—they rely on a new class of technologies that make component-level integration viable. In general, integrating heterogeneous components demands more sophisticated technologies compared to homogeneous systems, as inter-component communication and resource sharing become critical. Chiplets provide a foundation for breaking down monolithic chips into modular components, and interconnect technologies facilitate essential communication capabilities both within and between chips.

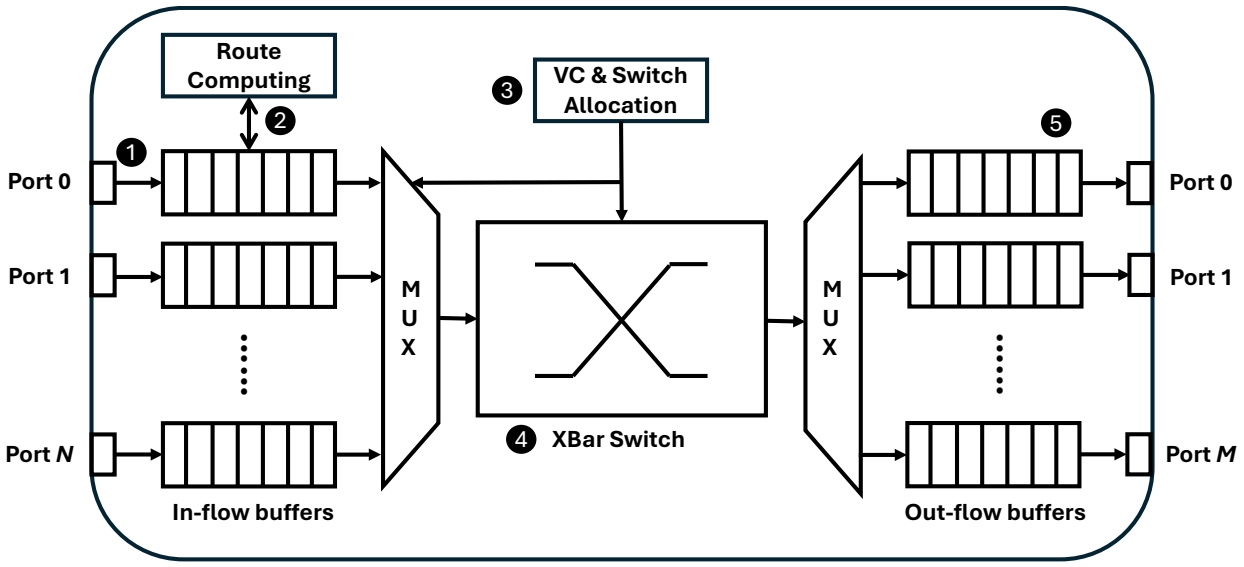


Figure 2.1: A high-level diagram of a router node. Credit return is omitted in the figure. VC = Virtual Channel. XBar = Crossbar.

2.2.1 Chiplets

Chiplets break large monolithic chips into smaller functional units that can be mixed and matched. Chiplets are smaller dies that can be integrated on silicon interposer substrates [124] to form a large chip. For instance, a CPU chiplet can be paired with a GPU or AI accelerator on the same package using silicon interposers. Chiplet-based designs lower manufacturing losses caused by variations and yield issues inherent in advanced node processes [52]. Additionally, chiplets enable manufacturers to combine different IPs flexibly, generating diverse SKUs targeting multiple markets with similar designs. In chiplet architectures, compute units with varied performance characteristics communicate through shared I/O dies to efficiently access off-chip resources, such as memory. Effective resource sharing is essential to maintain overall system performance.

2.2.2 Interconnect

Interconnect technologies are pivotal for enabling efficient communication among heterogeneous computing agents within and across integrated circuits. XBar switches [126] were used for interconnection in the early days and are now gradually replaced by NoC [22] systems as the size of the system scales larger. NoCs typically consist of interconnected routers arranged in defined topologies, routing communication packets dynamically to their destinations.

2.2.2.1 Router Design

Figure 2.1 shows the internals of a simple network router, similar to the one described in [42]. When a packet arrives at an input port of a router, it will first be buffered in the corresponding buffer, as shown with ❶ in the figure. ❷ While it resides in the buffer before being routed to the next node, the router will compute its next stop based on the routing and destination information. ❸ The virtual channel and the switch allocators will determine the idle channels and decide which direction or virtual channel has priority to send packets through the crossbar switch at the next cycle. If a packet wins the allocation, ❹ it can traverse the crossbar switch, ❺ arrives at the output buffer before it gets sent to the next node.

2.2.2.2 Arbitration in Interconnect Router

Various strategies exist for allocating router resources and prioritizing packets during arbitration:

- **Static arbiters:** These arbiters follow fixed policies, such as Round-Robin (RR), LRU, or strict priority. They are typically not programmable and cannot adapt to changing system or application-level requirements.
- **Dynamic adaptive arbiters:** These arbiters monitor recent traffic patterns and dynamically adjust priorities to achieve desired bandwidth partitions. While more flexible, the partitioning policies must be pre-configured or set by low-level firmware.
- **Quality-of-Service (QoS)-based arbiters:** These arbiters assign a QoS tag to each data flow or packet, indicating its urgency. Higher-urgency packets are prioritized. QoS status may be generated by the application or by system-level initiators. While QoS can be used to enforce bandwidth guarantees or perform operations like pipeline flushing [77], the static nature of QoS tagging—typically set at request initiation—limits its responsiveness to overall system dynamics.

2.3 Communication Protocols Between Hardware Components

Efficient and well-defined communication protocols are fundamental to harnessing the potential of heterogeneous hardware components. In this context, the term ”protocol” broadly refers to communication conventions followed during the design and operational phases of hardware.

2.3.1 Host-Accelerator Communication Conventions

Host-accelerator communication, demonstrated in Figure 2.2, typically employs shared memory and Memory Mapped Input Output (MMIO) to manage data transfer and execution control. We describe the conventions that hardware follows for intercommunication. Note that the conventions here are agnostic to physical implementations (*e.g.*, UltraPath Interconnect (UPI) [73], PCIe [6], Infinity [11], etc.) or data transfer protocols (*e.g.*, Core Cache Interface (CCI-P) [74], CXL [64], Advanced eXtensible Interface (AXI) [17], etc.).

After the host connects to the accelerator, as indicated in Step ❶ in Figure 2.2, the developer creates a shared memory space between the host and the accelerator for them to pass inputs and compute results. To kick off the kernel in Step ❸, the host writes inputs into the shared memory and issues instructions with metadata (*i.e.*, input starting address, result write back address, etc.) to the accelerator through (typically) MMIO. After receiving the instruction, the accelerator fetches the input data from the shared memory, writes back the result to the specified write-back address, and notifies the host.

A large kernel usually takes a relatively long time (on the order of milliseconds) to compute. The host can usually context-switch to another thread while waiting for results, thus keeping the host system busy and increasing the overall efficiency. The communication overhead induced by kernel launch and result fetch, which is on the microsecond level, is low compared to the kernel itself.

In contrast, the system is inefficient for instruction-level acceleration with the same approach. For each request, the system pays the round-trip latency of host-acceleration communication for a compute kernel that takes only tens of cycles. Meanwhile, the program on the host cannot proceed until it receives the result. While the overhead for large kernels remains proportionally low due to their longer execution times (milliseconds), instruction-level kernels incur prohibitive communication overhead (microseconds) relative to their brief execution durations (tens of cycles). This discrepancy creates execution inefficiencies, as conventional latency-hiding mechanisms such as context switching prove inadequate at these timescales, resulting in detrimental execution pipeline stalls and reduced overall system utilization.

2.3.2 Computer eXpress Link (CXL)

CXL [64] is an emerging protocol that bridges the gap between high-bandwidth access and cache coherence for heterogeneous hardware. Unlike traditional cache coherency models, which necessitate uniformity in coherency implementations or software-based coherency management, CXL centralizes coherency control at memory controllers, significantly simplifying integration and reducing overhead. This approach allows heterogeneous devices to implement local coherency

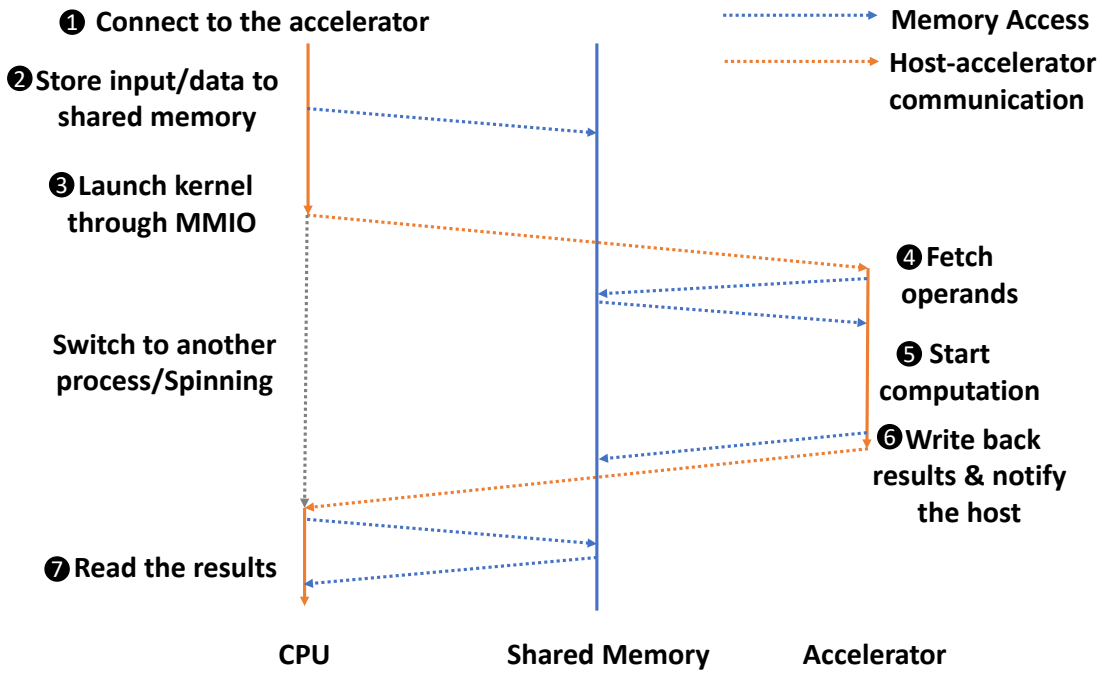


Figure 2.2: Host-accelerator communication conventions.

mechanisms internally, translating coherency messages into standard CXL format only when interacting with shared memory resources. Thus, CXL streamlines coherent memory access in complex heterogeneous environments.

2.4 Applications of Heterogeneous Design

To ground the discussion, we highlight real-world systems that exemplify the power and complexity of heterogeneous design. These heterogeneous design applications are studied and evaluated throughout this dissertation.

2.4.1 Q100 Accelerators

Q100 [159] is a specialized Database Processing Unit designed to efficiently accelerate database applications by executing standard relational operations like joins, sorts, and aggregations directly in hardware. It consists of specialized Application-Specific Integrated Circuit (ASIC) tiles, each tailored to different database operators. By leveraging data streams, Q100 maximizes parallelism and pipeline efficiency, greatly reducing the need for intermediate data storage and significantly accelerating query execution. It's particularly optimized for analytic database workloads, such as those used in big data analytics. Q100 demonstrates substantial improvements in both energy

efficiency (up to three orders of magnitude less energy consumption) and performance (up to 70 times faster) compared to traditional software database management systems running on general-purpose CPUs.

2.4.2 Posit Number Format and Its Acceleration

Emerging numerical representations, such as Posit numbers [44], large-number arithmetic [105], and residue number systems [91], aim to improve algorithm efficiency and accuracy. The Posit number system provides superior precision and dynamic range compared to the IEEE-754 floating-point standard [84]. However, widespread support for Posit numbers remains limited in mainstream hardware platforms. Implementing Posit arithmetic directly in hardware circuits, typically using FPGAs or embedded ASICs, can offer developers significant performance advantages. Interaction between the host and these specialized circuits would ideally occur at the instruction level, leveraging high-performance communication interfaces.

2.4.3 Sequestered Encryption

Spectre [90] and Meltdown [98], researchers have sought robust solutions for secure hardware microarchitectures. SE provides isolation between secure computation enclaves and conventional, potentially vulnerable circuits [23, 171, 114]. SE isolates sensitive data in encrypted form within a dedicated hardware enclave, decrypting data only inside the enclave for secure computation. Results are re-encrypted before leaving the secure environment. The host CPU interacts with the isolated SE enclave via secure, simplified Reduced Instruction Set Computing (RISC)-style commands transmitted over high-performance data buses. This secure partitioning ensures vulnerabilities in the general-purpose system do not compromise sensitive computations.

2.5 Implications for This Dissertation

The concepts and limitations outlined in the previous sections provide foundational motivation and context for the contributions of this dissertation. Specifically, the constraints identified in existing design languages and associated tooling (§2.1) inspired the development of Twine, which addresses these deficiencies through enhanced abstractions and integration mechanisms, with evaluations including the Q100 accelerator (§2.4.1) presented in Chapter 3.

Moreover, inefficiencies in host-accelerator communication conventions (§2.3.1) directly influenced the design and motivation behind Zipper. Zipper's efficacy was assessed using implementations of instruction-level accelerators for Posit number arithmetic (§2.4.2) and SE enclaves (§2.4.3), described in Chapter 4.

Finally, emerging chiplet technologies (§2.2.1) and communication standards, such as CXL (§2.3.2), together enable heterogeneous systems that were previously impractical, allowing the plug-and-play integration of compute components with differing coherence mechanisms to share memory resources transparently. They underpin the assumptions and design choices incorporated into Overpass. Overpass builds upon and advances conventional interconnect designs (§2.2.2), improving system performance, as detailed further in Chapter 5.

CHAPTER 3

Automating Modular Design for Rapid Generation of Heterogeneous Architectures

"This is a language for mere mortals."

Todd Austin, Professor at the University of Michigan

3.1 Introduction

As silicon scaling benefits wane, performance improvements in homogeneous systems are reaching a plateau. Therefore, developers are building heterogeneous systems to meet distinct demands for various workloads. However, a heterogeneous design inevitably increases complexity and leads to exponential growth in design costs [118]. Reusing existing components and modules across new systems is a promising strategy to mitigate this complexity explosion.

Although reusing modules can effectively reduce cost and enable fast design iteration, current hardware design tools generally fail to facilitate creating modules that are easily reusable outside their initial contexts. In existing workflows, a module is tightly bound to the particular component it serves: the composing module's timing behaviors and control signals are tailored solely to the current design. Reusing that module outside the original component demands adjusting the timing behavior of that module to satisfy new constraints, forcing developers to delve deeply into design specifics. The necessity to understand every detail significantly lowers productivity and module reusability, a phenomenon that software engineering research has long recognized [57]. Additionally, in scalable systems, the increasing size of components and numerous modules complicate the manual coordination of control signals. Design complexity further escalates if the composing

[1]The work presented in this chapter has been published in *S. Chen, Y. Fisseha, J. -B. Jeannin and T. Austin, "Twine: A Chisel Extension for Component-Level Heterogeneous Design," 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), Antwerp, Belgium, 2022, pp. 466-471, doi: 10.23919/DATE54114.2022.9774555.*

modules utilize different data formats. Such conditions create substantial challenges for hardware designers, making designs intellectually burdensome, costly, and prone to errors.

We argue that existing popular HDL do not sufficiently address the above problem, leaving designers to toil through the construction, debugging, and deployment of complex heterogeneous designs.

Conventional HDLs, such as SystemVerilog [68] and VHSIC Hardware Description Language (VHDL) [67], abstract I/O interfaces without differentiating clearly between data and control. The low-level semantics of HDLs pose no requirement on a module's behavior, shifting the responsibility entirely onto designers to understand module behaviors and reconcile incompatibilities. While some synthesis tools offer reusable building blocks, their highly vendor-dependent nature severely restricts design portability across different target devices. On the other hand, HLS, such as Vivado HLS [162] and SystemC [2], allows designers to translate software designs directly to hardware representations. Although HLS significantly improves productivity by abstracting away low-level hardware details, recent studies show that, in most use cases, HLS designs often fall short of HDL-based designs in quality [93]. Despite the productivity advantages of HLS, HDLs remain prevalent. Therefore, the solution presented in this chapter aims to improve heterogeneous design capabilities specifically within the popular HDL Chisel.

Chisel[19] is a Scala-based hardware generation language that has been drawing much attention recently. While Chisel introduces polymorphism to hardware design and provides standard modules to enhance productivity, it still does not sufficiently raise abstraction levels to substantially ease the challenges of heterogeneous design. As a result, designers must manually manage control signals, facing many of the same difficulties encountered with traditional HDL.

To effectively address these challenges in heterogeneous system design, we introduce Twine. Built upon Chisel, Twine retains the strengths of Chisel while adding powerful new features designed to ease the design burden of scalable heterogeneous systems significantly:

To better address the challenges posed to the designers of heterogeneous systems, we designed Twine. Built upon Chisel, Twine preserves all the power and strength that Chisel already carries while also introducing the following features that help alleviate much of the burden of designing scalable heterogeneous systems:

- Standardized module-level control interfaces with built-in parameterization, buffering, and reordering capabilities.
- High-level specifications for module-level producer-consumer relationships and dataflow management.
- Automated system-level coordination of control signals and flexible data format conversions.

These standard interfaces generalize modules, satisfying most system design requirements. High-level semantics combined with automation facilitate easier scaling and reconfiguration of systems. Our evaluation demonstrates that Twine simplifies heterogeneous design processes without sacrificing the design quality typically associated with manual, low-level HDL implementations.

3.1.1 Chapter Organization

The remainder of the chapter is organized as follows: §3.2 presents a motivating design challenge that illustrates the complexities of modular, heterogeneous hardware design. §3.3 introduces Twine, detailing its abstractions for control interfaces, producer-consumer relations, and interconnection automation. §3.4 describes the implementation of Twine, and §3.5 evaluates its impact on design productivity, design quality, and user experience. The chapter concludes with a summary in §3.6.

3.1.2 Chapter Contributions

We summarized the contributions of this chapter as follows:

- Identifying and characterizing the deficiency of current hardware design languages and the root cause of low reusability of designs. We further identified the complexity of coordinating control logic during the design space exploration phase of heterogeneous hardware design.
- Proposing Twine HDL, a new design extension that standardizes control interfaces and automates control logic generation through modular, reusable components.
- Evaluating Twine against industry-standard design languages, demonstrating that automating control logic dramatically reduces engineering overhead and enhances developer productivity.

3.2 A Motivating Design Challenge

In this section, we break down the process of designing a highly heterogeneous data query architecture that can be reconfigured to provide different functionalities and meet various performance requirements. We study the tasks a developer needs to complete while integrating various modules as motivation for the Twine HDL capabilities.

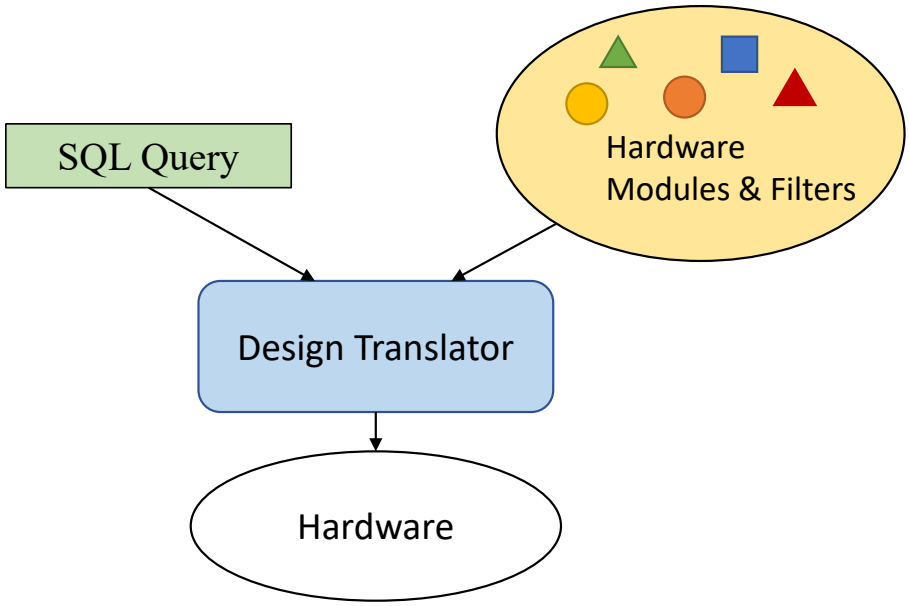
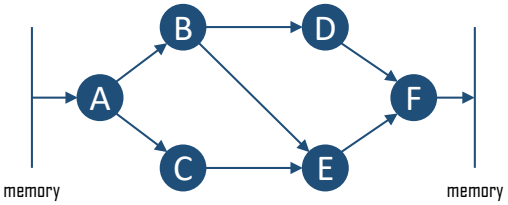


Figure 3.1: A Reconfigurable Data Query Accelerator Generator

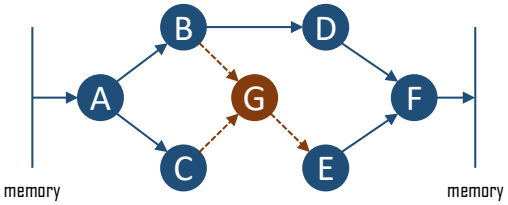
3.2.1 Heterogeneous Data Query Architecture

The heterogeneous data query architecture, which we demonstrate in a case study, is on the Q100 architecture [159]. Q100 is a well-recognized work in the database acceleration domain. It is a highly modular design: it is composed of 7 functional tiles (*e.g.*, `Aggregrator`, `ColFilter`, `BoolGen`) and 4 auxiliary tiles (*e.g.*, `ColSelect`, `Append`), all of which are highly configurable to the specific query. When generating a hardware design for a query or a class of queries, the generator maps and connects Q100 tiles to achieve optimal performance-area-power trade-offs, as shown in Figure 3.1. There are three major aspects to consider during the design optimization process:

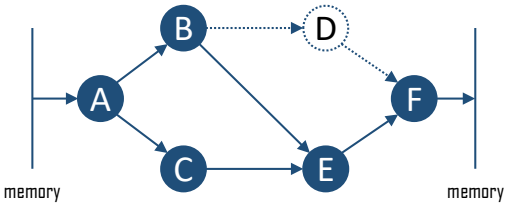
- **Depth:** The depth of the accelerator is the number of stages from the input to the output, which is mainly determined by query complexity. A complex query usually requires multiple filters and functional stages that have to be placed sequentially. Synchronization must be achieved at the end of each stage for valid data to arrive at the next stage with proper timing.
- **Width:** The Width of the accelerator determines the level of parallelism. As a rule of thumb, data streams usually can be processed in parallel, and we can expect higher performance if the architecture supports a high level of parallelism. However, the breadth of the architecture is bounded by memory bandwidth, power, and area constraints, so that an optimal balance may require Pareto analysis.



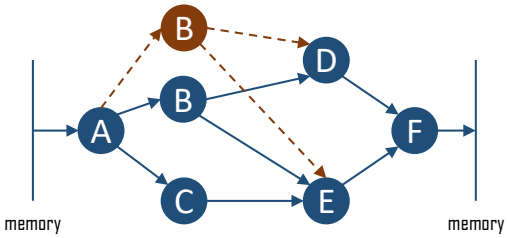
(a) Original Pipeline



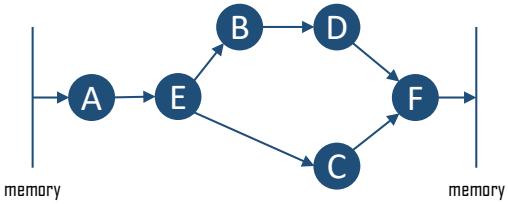
(b) Inserting New Modules



(c) Removing Modules



(d) Replicating Modules



(e) Reordering

Figure 3.2: Various reconfiguration decisions are possible during the design stage. Each letter represents a different type of module.

- **Ordering:** The ordering of tiles affects the ordering of processing stages, which could have a dramatic impact on overall performance. For example, aggregating data before ordering it would reduce the latency of the ordering stage because there are fewer entries to process after the data is aggregated. However, the system may need another aggregation stage after ordering if duplicate keys exist in the data stream. The designer likely needs to consider a variety of scenarios to make the optimal design choice.

3.2.2 Heterogeneous Hardware Design Process

3.2.2.1 Hardare Modules

To achieve high productivity, developers reuse and reconfigure pre-designed modules and integrate them together to craft a suitable computation pipeline. Module reusability is essential for large-scale designs whose size would otherwise necessitate a prohibitive amount of design effort. To reuse existing modules effectively, developers need to consider and fully understand two aspects of the module interface:

- **Control interface:** The control interface orchestrates the movement of data and maintains the control status of the module itself. In practice, different designs usually have different timing and flow control assumptions, and thus, different control interfaces. This interface diversity requires developers to devote a lot of effort to thoroughly understanding the interface. For example, the Q100 `filter` module consumes input from at least two modules simultaneously and produces its result within the same cycle. The `aggregate` module can take new input each cycle but only produces outputs after all records belonging to the current category are in. The `sort` module takes a vector of values and cannot consume new input during operation. Each distinct control interface dictates how developers should orchestrate control logic and communications of the system.

- **Data interface:** The data interface defines the type and format of input and output data. In hardware design, a data interface specifies the bitwidth of the ports, the semantic type of the data, and the degree of vectorization of the data. In Q100, values need to be converted from one format to another frequently, *i.e.*, a signed integer needs to be converted into a floating-point in floating-point multiplication and rounded back to a signed integer before being delivered to the next module. Other modules such as `sort` and `scalar ALU` take in a vector of data. Therefore, the data must be transformed between its parallel and serial forms.

In practice, it is difficult to completely separate the control interface from the data interface, making changes to the data interface would necessitate changes to the control interface and vice versa.

3.2.2.2 System-Level Configuration of Reusable Modules

Developers need to generate and connect all reusable modules in a specific order to achieve correct functionality and optimal performance. Figure 3.2 illustrates the developer assembling a pipeline and the various adjustments they can make in later stages of the design space exploration. In Figure 3.2, each node represents a hardware module.

The developer can change the system's functional behaviors by inserting or removing modules, as shown in Figure 3.2b and Figure 3.2c. An example scenario that would necessitate this task is when developers want to apply a filter to the data stream to process rows within a table selectively. To implement this functionality in Q100, we need to use multiple `BoolGen` modules that determine the selectivity based on a comparison between the key value and the reference value, and then a set of `Filter` modules that select or drop the input value based on the selectivity. To insert these new modules into the pipeline, developers need to break existing connections within the system, wire data ports of the new modules to the rest of the system, and re-design the control logic to ensure correct dependency and synchronization between the new modules and the rest of the system. Removing modules follows the same process.

Performance requirements and optimizations determine another category of configuration. The developers may need to replicate or reorder the modules to alleviate the architectural bottleneck, as shown in Figures 3.2d and 3.2e, respectively. In Q100, the scalar ALU can process multiple records simultaneously, but connecting non-scalar modules with the scalar ALU interface requires buffers and serialization modules. The exact control logic and buffer arrangements constantly change as the position of ALU and the data interface change. Thus, the order of the modules would also significantly impact performance. In a Structured Query Language (SQL) query where the users decide to filter out certain records and calculate the sum of those records by different categories, Q100 would have a filter layer and an aggregation layer. Both layers can reduce the number of records for the later stage of the pipeline, so which one is applied first should be determined by design space exploration. To explore a range of possible configurations, developers must correctly modify the control behaviors and connections between multiple modules.

3.2.2.3 Design Steps

To assemble all the modules into a functional module with tunable parameters and adjust them to achieve the desired performance, developers need to go through the following time-consuming and error-prone steps:

1. Fully understanding the interfaces of the modules. Many bugs and design errors result from misunderstandings or misinterpretations of the module interfaces.

2. Adding glue logic between hardware modules. This includes inserting type conversion modules between module ports of different types, serializing and de-serializing between vectorized modules, and padding buffers between modules to accommodate back pressure. This process would become prohibitively complex if developers needed to iterate and consider the appropriate transformation for each possible case.

3. Coordinating and tweaking control logic. Developers need to consider all the possible combinations of modules where the behavior of one module can affect the rest of the system. The fact that data ports between modules also need to be adjusted based on the configuration, as described in step ❷, significantly increases the overall complexity and the number of necessary changes for each distinct configuration.

We argue that to improve developers' productivity, modern hardware design languages should address these challenges and help developers complete these tasks. We will demonstrate how Twine solves these challenges in the rest of the chapter and compare Twine with other HDLs with regard to these tasks.

3.3 Twine Overview

Listing 3.1: An example of Twine in action. The highlighted lines show the features of Twine extensions, with the rest of the lines showing syntax and semantics of Chisel.

```
// Input operands for FMA operation
class FPFMAOps(val numBit:Int, val entries:Int) extends Bundle{
   val op1 = Vec(entries, new FP(numBit))
   val op2 = Vec(entries, new FP(numBit))
   val op3 = Vec(entries, new FP(numBit))
}

// Input operands for Sqrt operation
class FPSqrtOp(val numBit:Int, val entries:Int) extends Bundle{
   val op = Vec(entries, new FP(numBit))
}

// Output Result
class FPResult(val numBit:Int, val entries:Int) extends Bundle{
   val result = Vec(entries, new FP(numBit))
}
```

```
class FPFMA(val numBit:Int, val entries:Int) extends TwineModule{
   ❶ val in = Input(new FPFMAOps(numBit, entries))
   val out = Output(new FPResult(numBit, entries)
   val ctrl = new DecoupledIOCtrl(1,2)
   /*Computation logic for FMA(fused multiply-add) */
}

class FPSqrt(val numBit:Int, val entries:Int) extends TwineModule{
   ❷ val in = Input(new FPSqrtOp(numBit, entries))
   val out = Output(new FPResult(numBit, entries))
   val ctrl = new ValidIOCtrl(2)
   /*Computation logic for Sqrt */
}

class TopLevelDesign extends TwineModule{
   ❸ val in = Input(new FPFMAOps(32,8))
   val out = Output(new FPSqrtOps(16,4))
   val ctrl = new DecoupledIOCtrl(3,2)
   val fma1 = Module(new FPFMA(32,4))
   val fma2 = Module(new FPFMA(32,2))
   val sqrt1 = Module(new FPSqrt(16,2))
   val sqrt2 = Module(new FPSqrt(16,2))
   ❹ in >>> fma1 >>> sqrt1
   ❺ in.op1 >>> sqrt2
   ❻ TwineBundle(sqrt1, sqrt2, sqrt2) >>> fma2
   ❼ fma2 >>> ctrl
}
```

In this section, we will demonstrate the features of Twine with a running example shown in Listing 3.1. In this example, the top-level module is computing

$$out := \sqrt{in.op1 \cdot in.op2 + in.op3} \cdot \sqrt{in.op1} + \sqrt{in.op1} \tag{3.1}$$

The top level design is composed of the four modules `fma1`, `fma2`, `sqrt1`, and `sqrt2` as shown in Figure 3.3. Module `fma1` and `fma2` both take 32-bit floating point inputs and have vectorization of 4 and 2, respectively. The modules `sqrt1` and `sqrt2`, on the other hand, implement 16-bit floating-point operations and can take two inputs per cycle.

In a traditional HDL, the developers would be concerned about serialization, type conversion

between module interfaces, and excessive control complexity caused by the additional auxiliary codes. To relieve developers from complex control signal coordination and timing analysis at the high level, Twine has the following enhancements.

Twine standardizes I/O and control interface. Twine provides four standard interfaces and requires designers to choose one of them when designing their modules. The four standard interfaces are `TightlyCoupledIOCtrl`, `ValidIOCtrl`, `DecoupledIOCtrl`, and `OutOfOrderIOCtrl`. Twine imposes timing requirements on how modules use the interfaces through sanity checks. ❶, ❷, and ❸ in Listing 3.1 show examples of Twine interfaces. In Twine, each hardware module is a `TwineModule` which defines three variables:

- ***in*:** all input ports of the module
- ***out*:** all output ports of the module
- ***ctrl*:** the standard control interface used

Such requirements make understanding module behaviors considerably easier. The standard interface enables Twine to implement high-level semantics and automation to simplify the design process. We describe the details of the standard interfaces in §3.3.1.

Twine provides expressive high-level semantics to specify dataflow and producer/consumer relations. As mentioned, Twine abstracts away the high-level design specification's control logic and timing behaviors. As such, Twine offers a set of semantics that are more expressive than those in current design languages. It is more intuitive for developers to directly define producer/-consumer relations at a high level, rather than connecting low-level ports and control signals. In Twine, a new operator >>> directly specifies the relationships between modules and data ports. ❹, ❺, ❻, and ❼ in Listing 3.1 show the use of Twine's high-level connection operator between modules and between separate values and a module.

Twine automates system-level control signal coordination and data format conversion between modules. There are two key challenges when assembling pipelines, especially for reconfigurable accelerators: coordinating control signals and converting data between module boundaries. These two tasks are often considered 'busy work,' where designers need to put in considerable effort to implement them correctly, but get little value out of doing so.

Twine automates the assembly stage for developers. Based on the producer-consumer relations and the interface of each module, Twine first inserts necessary buffers and converters as intermediate modules between them. Twine then connects the data ports to the corresponding modules or its newly generated intermediate modules. Lastly, Twine coordinates the control signals through the standard interfaces to finish the last step of interconnection. As shown in Listing 3.1, developers do not need to insert auxiliary logic to adapt to different interfaces or re-specify control logic; it

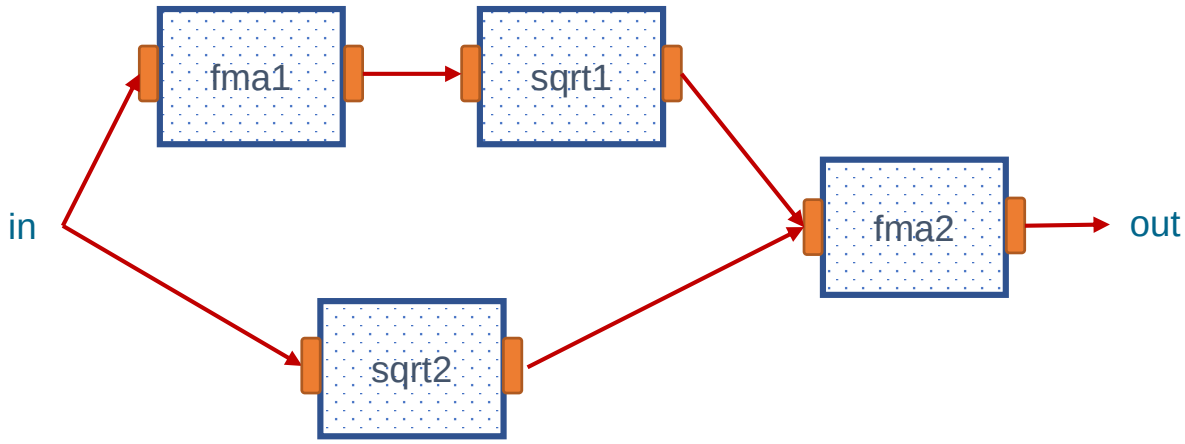


Figure 3.3: An example of producer/consumer relations of modules.

is automated in Twine. This feature is particularly useful during design space exploration with parameterization.

3.3.1 Control Interface Abstraction

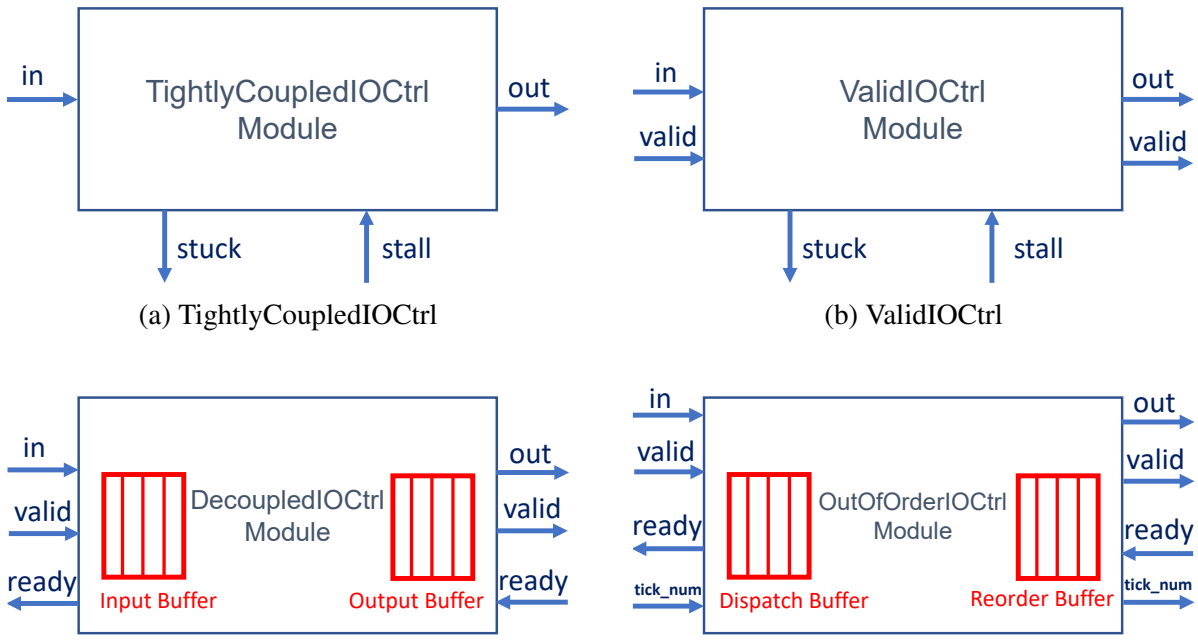
In this section, we describe the four standard control interfaces in Twine and the differences between each interface.

Standard interfaces, such as AXI, have been widely adopted in the industry as a common practice. However, most standard interfaces, like AXI, are overly complex and inflexible for intra-core communication. Other interfaces (*e.g.*, `DecoupledIO` in Chisel) only define a set of I/O ports. The language neither requires users to use standard interfaces in their designs nor enforces the way those standard interface ports should be used, which limits the utility of having standard interfaces in the first place.

Enforcing the use of standard interfaces makes hardware modules more predictable when reusing them in higher-level designs. Twine provides four standard control interfaces that free developers from needing to understand each module in detail. We describe the four interfaces below.

3.3.1.1 TightlyCoupledIOCtrl

`TightlyCoupledIOCtrl` is designed for the modules with constant latency and simple timing behaviors (*e.g.*, a pipelined multiplier that always takes four cycles to compute the result). It has the simplest interface and the lowest overhead. Since the input and output of this interface are tightly coupled, it cannot accommodate variable latency. As shown in Figure 3.4a,



(a) TightlyCoupledIOCtrl

(b) ValidIOCtrl

(c) DecoupledIOCtrl

(d) OutOfOrderIOCtrl

Figure 3.4: Four standard interfaces in Twine.

`TightlyCoupledIOCtrl` has two control signals: ***stall*** and ***stuck***. The module neither consumes nor produces if ***stall*** or ***stuck*** signal is asserted; otherwise, it takes inputs and produces outputs every cycle.

3.3.1.2 ValidIOCtrl

`ValidIOCtrl` is designed to be more flexible than `TightlyCoupledIOCtrl` to support non-deterministic latencies during execution while maintaining low overhead. As shown in Figure 3.4b, `ValidIOCtrl` implements an additional pair of ***valid*** bits at both ends, which allows the input side to be only loosely coupled with the output side.

The two new signals introduced in `ValidIOCtrl` are ***valid*** signals. The input ***valid*** bit notifies the module of new available inputs, and the output ***valid*** bit signals the external system to handle the new results produced by the module.

3.3.1.3 DecoupledIOCtrl

While `ValidIOCtrl` has increased flexibility, it cannot locally buffer the backpressure from downstream modules, so the backpressure will propagate further upstream and stall more modules.

Table 3.1: Comparison between different interfaces. *Req. fixed lat.: Require Fixed Latency, Intra.: Intra-module, Inter.: Inter-module

	TightlyCoupled	Valid	Decoupled	OutOfOrder
Flexibility	Very low	Low	High	High
Overhead	Low	Low	High	High
Req. fixed lat.*	Yes	No	No	No
Out-of-Order	No	Intra.*	Intra.	Inter.*
Backpressure	Yes	Yes	No	No

`DecoupledIOCtrl` is the first control interface in Twine where the inputs and outputs are entirely decoupled. `DecoupledIOCtrl` supports FIFO buffers at both ends of the module to accommodate backpressure locally. Those buffers can be easily customized and generated through parameters during declaration. As shown in Figure 3.4c, `DecoupledIOCtrl` implements ***valid/ready*** pairs at both ends of the module. ***valid/ready*** pairs provide a handshake mechanism to adapt to more complex control logic. ❶ in Listing 3.1 shows the declaration of a `DecoupledIOCtrl` with 1 and 2 buffer entries on the input end and the output end of the module, respectively.

3.3.1.4 OutOfOrderIOCtrl

For operations that have large latency variances (*e.g.*, memory access), out-of-order execution is necessary to exploit parallelism. `OutOfOrderIOCtrl`, shown in Figure 3.4d, is the interface designed for such use cases. It is one of the most flexible, but also one of the most complex, interfaces. The request that goes into an out-of-order module would be assigned a ticket number `tick_num` to enable out-of-order processing and completion. At the end of execution, the requests would be either automatically reordered with `tick_num` or passed on to another out-of-order module.

Buffer management and reordering are transparent to developers so they only need to focus on handling incoming requests.

3.3.1.5 Comparison Between Interfaces

Table 3.1 provides a qualitative comparison of interfaces. As a rule of thumb, modules with naive timing behaviors should use low-overhead interfaces. Those with complex functionalities and long latency operations should implement the more flexible interfaces.

3.3.1.6 Adapting and Reusing Existing Designs in Twine

There are two ways to adapt and reuse existing designs in Twine: either in Verilog or Chisel. First, the developers can extend a `TwineAdaptionModule` and implement one of the four standard Twine

control interfaces without changing the internal logic of the existing designs. The extended module can take the fullest advantage of Twine's automation framework and be integrated seamlessly into the existing Twine ecosystem. The second way is to integrate the existing modules manually. Developers can benefit from the part of the system that implements Twine interfaces without sacrificing functionality in the parts that do not.

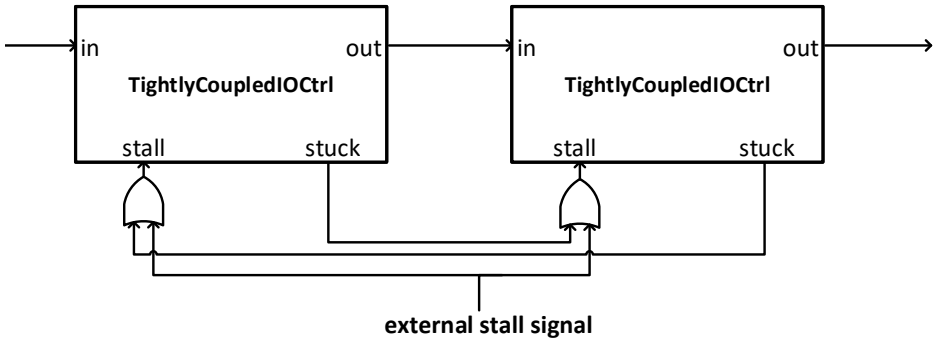
3.3.2 Specifying Producer/Consumer Relations and Dataflow

To improve the efficiency of specifying designs, Twine provides a set of semantics to express producer-consumer relations rather than low-level port connections. The producer/consumer model abstracts away the timing behaviors and control signals and allows users to focus entirely on data dependency. Such a design philosophy makes design much more intuitive and enables high-level design automation. Twine needs to collect information about the data dependencies between each module to synthesize system control logic. To analyze the data dependency, we label each module with ***producer***, ***consumer***, and/or ***stakeholder***, relative to its neighboring modules. The consumers are the modules that take in values directly from the current module. The producers are the modules from which the input data originates. A stakeholder is a module that needs to monitor the status of the other modules to make control decisions. For example, when there is one producer with multiple consumers, the producer can only release the results when all consumers are ready to consume. In such a case, each consumer needs to monitor the status of other consumers to avoid duplication and ensure correctness. There are two types of stakeholders: producer-stakeholder (*p-stakeholder*) and consumer-stakeholder (*c-stakeholder*). P-stakeholders share at least one common consumer, and c-stakeholders share at least one common producer. Taking the design shown in Listing 3.1 and Figure 3.3 as an example, relations between the modules are shown in Table 3.2. Module `sqrt1` and `sqrt2` are p-stakeholders for sharing a common consumer `fma2`; `fma1` and `sqrt2` are c-stakeholders for sharing a common producer `top`.

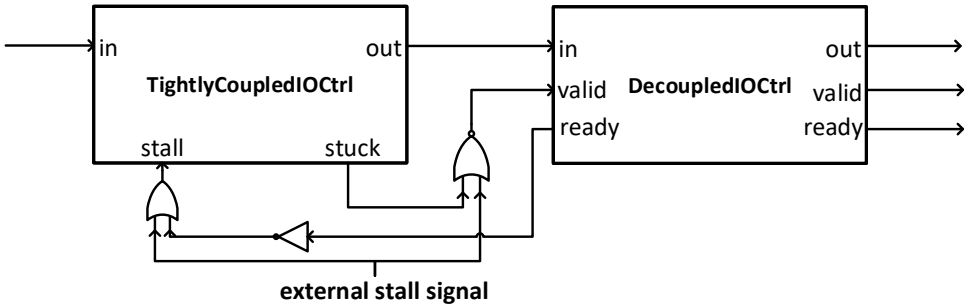
Users only specify the producer/consumer relations in the specification. Twine determines stakeholder relations during the elaboration process. Users can specify a producer/consumer relationship between any two Twine objects with `>>>` operator as *producer* `>>>` *consumer*. Twine will match the ports on both sides if a bundle of ports is provided and infer the producer/consumer relations from the matching ports.

3.3.3 Component Interconnection Automation

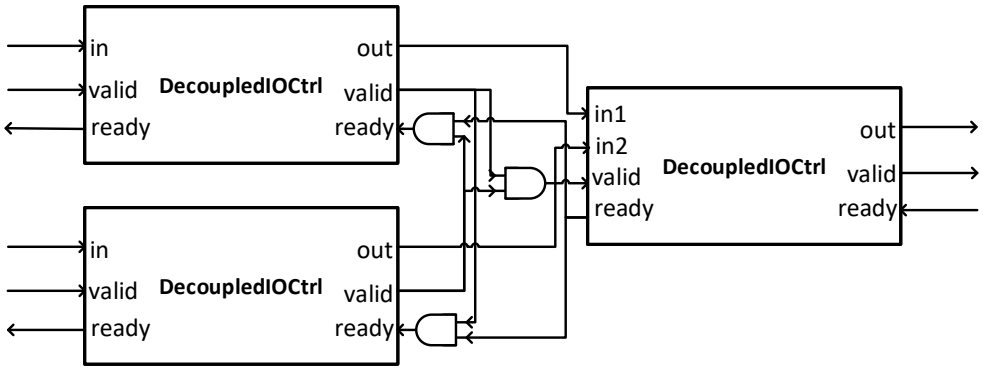
Based on the module interfaces and high-level specification of producer/consumer relations, Twine automates two challenging parts of the design. The first part is control signal coordination. For a large-scale and reconfigurable design, the modules and their topology are not concrete or material-



(a) Lock-step connection.



(b) Cross module connection.



(c) Connection with multiple producers.

Figure 3.5: Examples showing interconnections between different modules.

Table 3.2: The relationship between modules in a system shown in Figure 3.3 and specified in Listing 3.1.

module	consumer	producer	p-stakeholder	c-stakeholder
top	fma1,sqrt2	fma2	N/A	N/A
fma1	sqrt1	top	N/A	sqrt2
fma2	top	sqrt1, sqrt2	N/A	N/A
sqrt1	fma2	fma1	sqrt2	N/A
sqrt2	fma2	top	sqrt1	fma1

ized at the design stage because the modules of the full design and the connections between them will only be fully known after a design space search is complete and all parameters have been set. It is challenging for designers to identify the producer/consumer pairs with potential stakeholders and coordinate the control signals correctly. The second part is data format conversion between modules. It is not uncommon for a heterogeneous design to reconfigure module bandwidth with different data types. It is important to accommodate these challenges at the language level.

3.3.3.1 Coordination of Control Signals

Control signals in a hardware design determine when a module should consume and release the data. Twine standardizes the control interface. For each module, Twine checks the module's status and controls its behavior through a set of pre-defined signals. Twine then synthesizes the system control logic based on the interface each module uses and the high-level specification that describes the interconnection between modules. All the standard interfaces defined in Twine can be mixed and matched with each other, either on the same level or across levels (*e.g.*, a parent module connecting with a child module). Here, we include several examples to demonstrate how this process works. The complete connection rules are available in the Twine documentation.

Figure 3.5a shows an interconnection between two `TightlyCoupledIOCtrl` modules. In this case, two modules would proceed in a lockstep manner—either proceeding together or stalling together. Figure 3.5b shows an interconnection between a `TightlyCoupledIOCtrl` module and a `DecoupledIOCtrl` module. The `DecoupledIOCtrl` module only needs to bind its input side to this relationship, thanks to its flexibility. In contrast, the `TightlyCoupledIOCtrl` module is locked in step with the input side of the `DecoupledIOCtrl` module. Figure 3.5c shows an interconnection when there are multiple producers and one consumer. In this case, two producers are p-stakeholders to each other; therefore, they need to listen to both the consumer and the other stakeholder to sync their data transactions. All connections can be generated automatically by Twine.

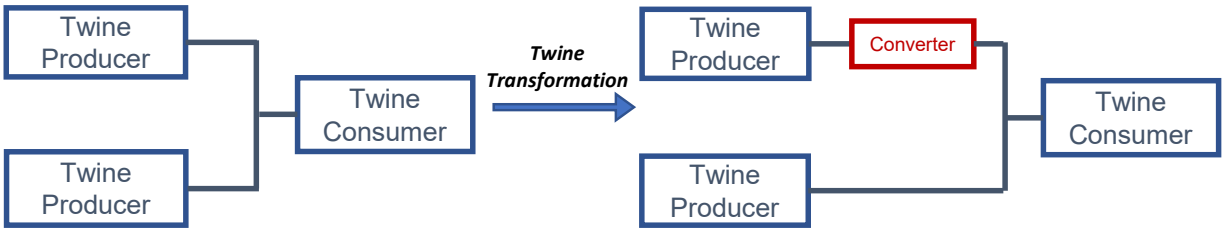


Figure 3.6: Twine inserts a converter between modules to adapt different data types.

3.3.3.2 Data Format Conversion

Twine can automatically convert data formats between modules. There are two types of conversions between module boundaries: type and width.

A type conversion occurs when the two connecting ports have different data types. At the current stage of development, Twine supports automatic conversion between unsigned integers, signed integers, and common formats of floating-point data (*i.e.,* 8-bit, 16-bit, 32-bit, 64-bit). The framework can be easily extended to support more types as needed. When Twine detects mismatched data types at the two ends of the connection, Twine will automatically insert a converter in between to adapt different types.

A width conversion happens when the two connecting ports have different numbers of units produced and consumed each cycle. This usually occurs when there are vector modules in the design. Twine will insert serializers or de-serializers between modules based on the width difference between interfaces.

If the converter is fully combinational, it is transparent to the surrounding modules and does not change the control logic. If the converter takes multiple cycles, Twine will treat it as a complete module and update the producer/consumer relation map to rewire the control logic, as shown in Figure 3.6.

3.4 Implementation

In this section, we describe the implementation of Twine. Twine is completely backward-compatible with Chisel and supports all Chisel functionalities. Developers can opt in to Twine for their design by simply inheriting from `TwineModule` when declaring modules. A `TwineModule` can be used as a normal Chisel module as well. When a Twine operation is evaluated, it is translated into a set of Chisel operations, inserts conversion modules where necessary, and registers the connection information in the Twine profile——a collection of information that guides the system's synthesis at later stages. The connections between modules are only complete after all statements

have been evaluated and the context information has been fully collected. We add hooks into the Chisel elaboration phase to initiate Twine synthesis after all statements have been evaluated. Twine then analyzes the producer/consumer relations to finalize connections for control coordination.

The workflow from Twine to FIRRTL are as follows(*Twine-specific steps are marked with* $*$):

1. Generate low-level modules using Chisel. *Twine modules are implemented with a standard Twine interface.**

2*. Collect high-level information from Twine specification.

3*. Generate necessary buffers and converters. Insert them into the right locations based on the high-level specification.

4*. Reconstruct the roles and relations of Twine modules.

5*. Coordinate control signals. Interconnect all modules.

6. Compile Chisel primitives into FIRRTL representation.

7*. Checks that modules meet the required standards to generate correct Twine design (*e.g.*, `TightlyCoupledIOCtrl` module has fixed latency, `ready` and `valid` signals in `DecoupledIOCtrl` are not interdependent).

8. Emit finalized FIRRTL files.

3.5 Evaluation

A hardware design language should be easy to learn and easy to use. It should make users more productive at what they do without compromising the design quality. Therefore, we evaluate Twine across three key metrics: design productivity, design quality, and early adopter experience.

3.5.1 Design Productivity

To evaluate productivity, we designed a library of data processing modules. The library includes modules essential to many data processing workloads (*e.g.*, *aggregate*, *column filter*, and *boolean generation*). We use building blocks to assemble an accelerator that speeds up the processing of complex SQL queries. The accelerator first filters out the selected rows, then adds two columns together, and lastly aggregates results by the key. We explore the design space by exploiting request-level parallelism and data-level parallelism. We exploit request-level parallelism by adding more modules at the front end to filter multiple requests concurrently, and we exploit data-level parallelism by vectorizing the ALU.

We hand-built 12 different design configurations with various memory bandwidths and degrees of vectorization to find the optimal design balance between performance and area. To take advantage of increased memory bandwidth, developers must place more modules on the system to

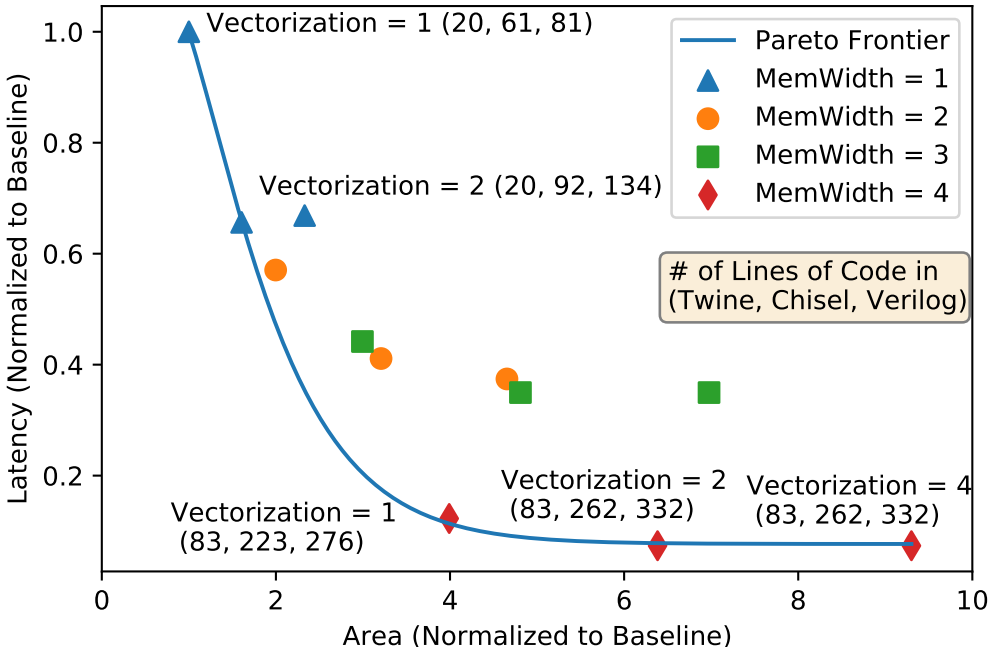


Figure 3.7: Pareto chart of a data processing accelerator design space. MemWidth represents the number of requests that the accelerator can read from or write to memory each cycle. Performance is measured as the average latency of processing a batch of 80 requests.

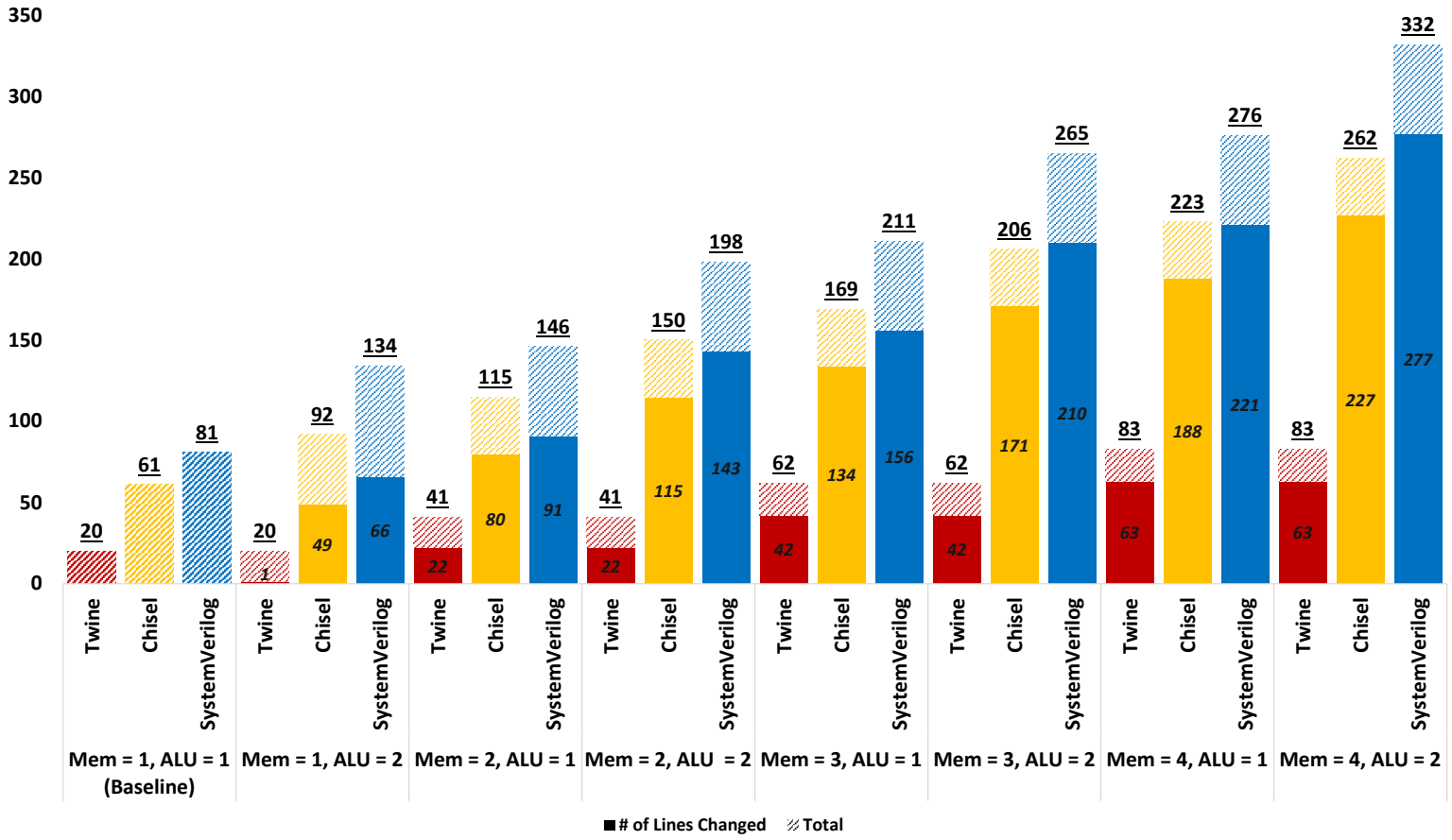


Figure 3.8: The number of lines needed for different configurations when implemented in Twine, Chisel, and SystemVerilog. The solid portion represents the number of changed lines compared to the baseline, with the memory bandwidth of 1 row of data and a single ALU that processes 1 row at a time.

Table 3.3: Area and frequency comparison of RISCV-MINI in Chisel and Twine.

	RISCV-MINI in Chisel	RISCV-MINI in Twine
Area	727004.94 μm^2	725937.9μm^2 (-0.14%)
Clock Period	0.85 ns	0.82 ns (-3.5%)

exploit data parallelism, while also considering area and power budgets. The degree of vectorization will cause the module interface to change. Thus, developers need to adapt and buffer the data between the vectorized modules and non-vectorized modules. These two tunable parameters will lead to a process in which the developers must add, remove, change, and reconnect hardware modules, which is common in the design space exploration phase of heterogeneous design.

We first simulate the accelerators in Verilator to estimate the average number of cycles each design takes to process 80 requests. Then, we synthesize each configuration to determine the clock period and estimate the area. Since we focus on evaluating design methodology rather than the design itself, we use performance and area as examples in this experiment. The developer can measure different metrics based on their targets. The performance is measured as the average latency required to process 80 requests. Figure 3.7 shows the 12 different configurations we explored with regard to performance/area. Through the figure, we can identify and label five Pareto optimal configurations.

We then compared the handwritten implementations in three different design languages: Twine, Chisel, and SystemVerilog. Figure 3.8 shows the number of modules and the total number of lines required to assemble the eight most representative configurations in different languages. We count the number of line changes for each language to specify every target configuration from the baseline design, where the accelerator can read one request per cycle and has only one ALU. Thanks to high-level specifications and automation, the number of changes in Twine is primarily affected by the number of new modules and new producer-consumer relations, while implementations in Chisel and SystemVerilog also need to consider timing behaviors and data formats. In Chisel and SystemVerilog, we need to manually coordinate signals for each new module and modify the control signals for existing ones that are indirectly affected by the new configuration. In Twine, such tasks are automated during design elaboration. . Our results show that designers write and modify significantly less code in Twine to assemble a new scalable system with reusable modules, which results in improved productivity and facilitates design space exploration.

3.5.2 Design Quality

To verify that Twine achieves comparable design quality to Chisel, we ported an in-order RISC-V core [87] from Chisel to Twine. In the original design, the datapath, all stage registers, and the

control logic are specified inside one monolithic module. In the Twine design, all function units and stages are encapsulated into nine separate modules. The top-level module only specifies the data flow between modules and the communication between the datapath and the cache. Table 3.3 compares the area and frequency between the two designs synthesized with IBM 45nm SOI12S0 CMOS process. Twine can achieve approximately the same performance as the Chisel design. The marginal variance between the two designs is due to slight differences in the interface.

3.5.3 Early Adopter Experience

To assess the impact of Twine on new users, we assembled a team of seven graduate students with diverse backgrounds but no prior experience with Twine. Out of the seven students, one had used Chisel before, two had experience with hardware design but had not used Chisel, and the rest had no experience with hardware design. The early adopters were given a library of Twine modules with a sample design and a description of what the design does. The sample design has 10 modules and functions as a basic data filter and aggregation engine. They were then asked to insert new modules into the pipeline to add another layer of data filtering and accommodate an additional stream of data input based on a new description. There are twice as many modules in the new design as compared to the sample design. All of them learned the language and modified the design correctly within 35 minutes, even though many of them had no hardware design background. On average, each of them changed 12 lines of code, or 22% of the original sample codes, to accommodate a system that was twice as complex. All of them responded positively when asked about their experience with Twine.

3.5.4 Limitations

During our evaluation, we identified a few cases where the highly structured nature of Twine had a negative impact on design quality. First, the standard interfaces in Twine are *not sufficiently flexible for designs to change processing granularity dynamically*. A module may dynamically decide to batch requests together for higher throughput or proceed with them immediately for lower latency. Since the conversion logic is finalized during generation, the module cannot change it during execution. To overcome this limitation, developers can fall back to Chisel to specify control and conversion logic manually or integrate the functionality inside the module. Meanwhile, dynamic scheduling can be easily achieved through software. Second, there may be *missed cross-module optimization opportunities*. Since Twine imposes high modularity requirements, developers may not be able to easily identify cross-module optimization opportunities (*e.g.*, early forwarding). However, such opportunities are usually hard to find and exploit in scalable heterogeneous systems.

3.6 Chapter Summary

In this chapter, we discuss the emerging challenges hardware designers face in the new age of heterogeneous designs. We propose Twine, a Chisel extension that supports module-level abstraction to improve accessibility and productivity. Twine standardizes module interface, provides high-level semantics, and automates system-level control coordination with inter-module data formats adaptation. Twine is evaluated against Chisel and SystemVerilog for developer productivity by performing the same design space explorations. Twine is further assessed by its design quality (design area and timing) by replicating a three-stage RISC-V CPU, which is initially written in Chisel. Our results show that Twine is easy to learn, easy to use, and considerably improves productivity for designing heterogeneous hardware while retaining the same high design quality.

CHAPTER 4

Tolerating Communication Overheads in Heterogeneous Designs

4.1 Introduction

Heterogeneous systems with domain-specific accelerators have been widely adopted in practice [146]. In such systems, the host offloads a heavy compute kernel to the target accelerator and retrieves the result asynchronously when it is ready. Under such a design paradigm, the host and the accelerator sustain the least communication possible: once to initiate the request and once to receive a result. As a result, the communication latency overhead between the host and the accelerator is often overlooked because it is trivial compared to the accelerator execution time.

However, recent developments in security [23] and computing theory [44, 128] necessitate the acceleration of instruction-level kernels.

A short instruction-level kernel is a simple, functional accelerator command that takes multiple input operands to produce a result. These instruction-level kernels have an edge over large kernels because they provide programmers with general-purpose operators and extra features to support arbitrary algorithms, while adding minimal cost to the overall system.

This chapter will use two instruction-level kernels in the production environment as case studies. The first one is a customized secure enclave [23] that supports security operations on sensitive ciphertext, where the key resides only in a separate, standalone hardware device. These security operations are RISC-like and Turing-complete but cannot be executed on a regular CPU due to restricted security requirements. The second one is the hardware support for Posit32 [44, 128]. Posit32 is a new number format proposed to replace the IEEE 754 floating-point format. Since Posit32 supports general scientific calculations, the kernels are RISC-like arithmetic instructions.

[1]The work presented in this chapter has been published in *Shibo Chen, Hailun Zhang, and Todd Austin. 2025. Zipper: Latency-Tolerant Optimizations for High-Performance Buses. In Proceedings of the 30th Asia and South Pacific Design Automation Conference (ASPDAC '25). Association for Computing Machinery, New York, NY, USA, 567–574, doi: 10.1145/3658617.3697546*

Other examples include Microchip Divide and Square Root Accelerator [151] and extended Floating Point Coprocessor [147], to name a few.

Such light kernels exhibit two distinct characteristics compared to large kernels. First, executing an instruction-level kernel only takes tens or hundreds of cycles. This is a tiny fraction of the time compared to computing a large kernel. Second, since instruction-level kernels are more general-purpose, they do not harden control logic into hardware circuits. Instead, they rely on the host to control and schedule decisions, leading to frequent host-accelerator communications. For these reasons, applications that depend on these instruction-level kernels are typically susceptible to communication latency.

Such kernels are best served with tight integration into the CPU pipeline. However, before major CPU vendors add them as Instruction Set Architecture (ISA) extensions to their designs, developers must support them through separate IPs, acceleration cards, FPGA fabric, or Chiplets. Recently, Intel introduced its Chiplet-based FPGA development board as part of the prototyping and evaluation loop.[95, 75] Developers can trial and error with FPGA before mass-producing their customized designs. Such advancement opens more doors to accelerators of various sizes, assuming communication latency issues can be mitigated.

While state-of-the-art high-performance data buses have increased memory bandwidth over the years, access latency has not scaled in proportion because link traversing does not scale well as the technology node shrinks [168]. A recent study [100] shows that the round-trip latency through the popular PCIe Gen 3.0 [6] or Intel UPI [73] is at the microsecond scale. Modern computer systems cannot efficiently tolerate microsecond-level latencies. Thus, most of these latencies are fully exposed [20]. Inefficiency in latency tolerance hinders the broad deployment of instruction-level accelerations in production environments. New communication protocols, like CXL [64], also do not fully solve the latency challenge, as they cannot overcome the fundamental physical design limitations.

Although communication latency is hard to reduce through improved physical designs, we observed two significant opportunities for latency-tolerant optimizations. First, there is parallelism at both the request and the device level. We can enhance host and accelerator utilization by enabling out-of-order and parallel execution; second, instruction-level kernels exhibit significant temporal locality: the results of previous requests often become inputs for subsequent requests. We can exploit this locality to reduce data movement. However, there are multiple challenges to overcome before it is possible to capitalize on these opportunities in a real production system:

- **Performance projection:** Since latency-hiding optimizations on heterogeneous systems usually comprise multiple moving parts, it is desirable to project whether applying these optimizations is beneficial before devoting the engineering efforts to building them. However, this is a complex task as the performance gain relies on multiple hardware parameters and

intrinsic features of the workload.

- **Complex parallelism model:** data dependency can exist between a host instruction and an accelerator instruction or within accelerator instructions. Harvesting parallelism requires a sophisticated scheduling algorithm that understands the semantics of two sets of ISA and handles data communication across device boundaries.

- **Data tracking:** Since data is continually moved between host and accelerator, the system needs to precisely track the location of the data to ensure functional correctness.

- **Design Complexity:** Considering the sheer size of possible accelerator designs and hardware platforms, customizing compilers would be a heavy technology burden for ordinary system developers. To make things worse, major vendors only provide function-level APIs [163, 104], which makes adding compiler support even more difficult. A portable and extensible solution is necessary to make instruction-level accelerations accessible and scalable.

To address the issues above and make instruction-level acceleration accessible, we first propose a qualitative analytic model. This model helps system designers quickly identify opportunities in their instruction-level acceleration workloads for optimizations and projects how much benefit latency-tolerant optimizations can provide through a collection of hardware/software constraints: the degree of temporal locality, the length of data dependency chain between accelerator instruction and the host instruction, the memory access latency, and the degree of parallelism.

To exploit the opportunities identified in the analytical model, we further propose *Zipper*. Working on top of existing data buses, Zipper is a protocol optimization layer that optimizes latency-sensitive applications deployed on a heterogeneous system where two devices are connected through a high-performance data bus. It dynamically analyzes data dependency, tracks data movement across devices, and exploits temporal locality and parallelism as a program proceeds. Zipper uses a software-defined request scheduling approach that requires no modification to application logic, compilers, or data buses. On the software side, Zipper provides a runtime library that identifies temporal locality and parallel execution opportunities and schedules optimized accelerator requests to enable resource-constraint-aware parallelism and data reuse. The implementation details of this runtime library are hidden from software developers, and the developers can use encapsulated data types as if they were host-native. On the hardware side, Zipper offers a small request buffer to cache data and enables out-of-order execution of requests with data reuse.

We conducted experiments on two benchmark suites with two distinct instruction-level kernels to demonstrate the effectiveness of Zipper. Our experiments show that Zipper provides end-to-end speedup from 1.5x to as much as 8x.

4.1.1 Chapter Organization

The remainder of the chapter is organized as follows: §4.2 explores optimization opportunities in host-accelerator communication and quantifies the costs of instruction-level acceleration. §4.3 describes the Zipper architecture, including its communication protocol, hardware structure, and latency-tolerant enhancements. §4.4 details the experimental setup and presents performance evaluation results. §4.5 discusses design trade-offs, accelerator memory access patterns, and the impact of different optimization techniques. Limitations are presented in §4.6, followed by the chapter summary in §4.7.

4.1.2 Chapter Contributions

We summarize the contributions of this chapter as follows:

- systematically studying optimization opportunities in instruction-level acceleration on a heterogeneous system.
- proposing a qualitative analytic model to estimate the benefits of various optimization techniques for instruction-level acceleration on a heterogeneous system.
- proposing a general protocol optimization layer, Zipper, to tolerate communication latency over high-performance buses.
- thoroughly evaluating Zipper's performance improvements and area overhead with two case studies.
- providing insights into Zipper's performance improvements and breakdown of the contributions of each Zipper feature.
- discussing the scalability and compatibility of Zipper to various designs and platforms.

4.2 Discovering Optimization Opportunities

This section will discuss three key opportunities to optimize instruction-level acceleration and propose a new model to help developers understand the true performance implications of instruction-level acceleration.

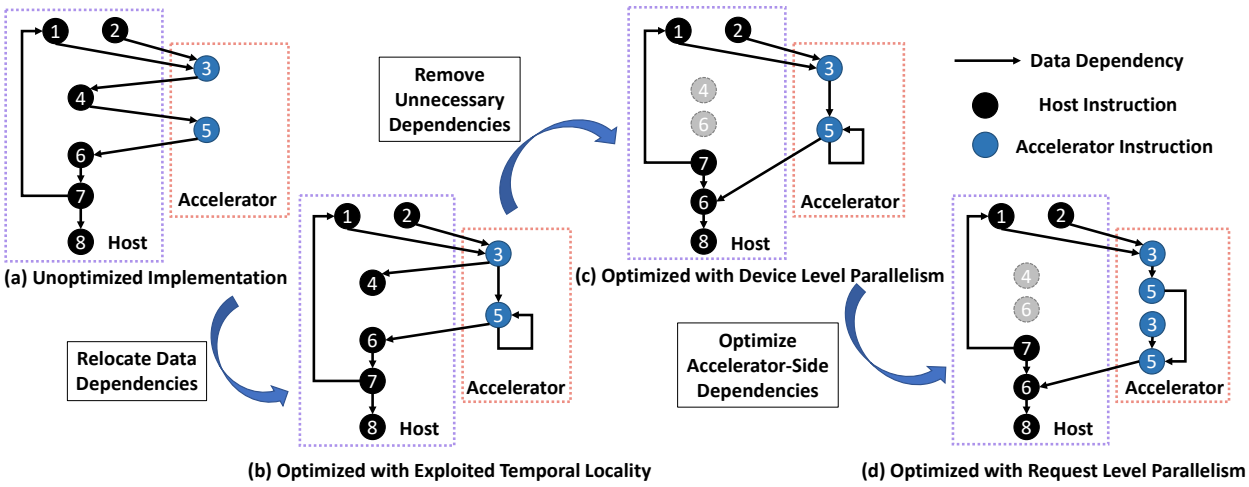


Figure 4.1: A step-by-step optimization for the instruction sequence shown in Algorithm 4.1

Data: *An array of* $2n$ *elements* : $arr[2n]$, *A write* $-$ *back address* $addr_{wr}$
Result: $y = \prod_{i=0}^{2n-1} a_i$ over special operator $\otimes$
$i \leftarrow 0$; $result \leftarrow 1$
while $i < (2n-1)$ **do**
  $1 : a \leftarrow load(Mem[arr+i])$
  $2 : b \leftarrow load(Mem[arr+i+1])$
  $3 : c \leftarrow a \otimes b$
  $4 : fetch\ c$
  $5 : result \leftarrow c \otimes result$
  $6 : fetch\ result$
  $7 : i \leftarrow i+2;$
end
$8 : Mem[addr\_\{wr\}] \leftarrow result$

Program 4.1: A reduction algorithm with operator $\otimes$.

4.2.1 Optimization Opportunities

Despite the inefficiencies of instruction-level acceleration, many exploitable opportunities exist to compensate for the communication overhead. In this section, we use an example reduction algorithm over an abstract hardware-accelerated operator $\otimes$, as shown in Algorithm 4.1, to demonstrate existing latency-tolerant opportunities. In this algorithm, we want to calculate the product of the $2n$ inputs over a hardware-accelerated operator $\otimes$ and store the result in the write-back address. The developer chooses to provide only the operator acceleration for its generality and small area footprint, and executes the rest of the instructions on the host. The accelerator is attached to the host system, which runs the algorithm by high-performance data buses, *i.e.*, UPI, PCIe, etc.

Figure 4.1 shows the data-dependence graph of Algorithm 4.1 and the optimizations to eliminate dependencies and enable parallelism. Starting with the unoptimized implementation in (a), the developer partitions the instructions based on the devices' capabilities: execute instruction `1`, `2`, `4`, and `6-8` on the host, and offload instruction `3`, `5` to the accelerator. An off-the-shelf compiler cannot analyze and optimize cross-device dependencies; therefore, the system must execute instructions sequentially in program order. Since the shared memory used for communication is small, the host must fetch the results back to its own memory space to free up space for future transactions. In the unoptimized design scheme, all accelerator results need to be fetched back before the system can continue execution. The system pays the full round-trip overhead for each accelerator instruction in this unoptimized implementation.

4.2.1.1 Exploitable Temporal Locality

We first notice that not every dependency is created equal. A cross-device data dependency is much more costly than a local one due to the communication overhead. Based on this observation, we eliminate cross-device dependencies and replace them with local ones whenever possible. That is, instruction `5`'s two operands result from instruction `3` and its result from the last iteration. Therefore, instruction `5` does not need to get its inputs from the host, and we can move this cross-device dependency to a dependency that is local to the accelerator, as shown in (b). By relocating cross-device dependencies, we can avoid much inter-device communication and thus reduce communication overhead.

We observe that many applications exhibit temporal locality on the accelerator side: ***the result of an instruction is likely to be used by its subsequent instructions as input operands***, which presents many opportunities to relocate cross-device dependencies. §4.5.2 presents a detailed analysis of the temporal locality of applications.

4.2.1.2 Device-level Parallelism

More optimization opportunities appear once cross-device dependencies have been relocated to the accelerator. Instructions `4` and `6` block instructions that fetch results from the shared memory. After the dependencies have been relocated, instructions `4` and `6` can now be pushed off and away from the critical path. This means the system does not need to block the program to fetch results, as shown in (c). The host can continue execution while the accelerator is working on the received requests. Being non-blocking, the host can run ahead to fetch new data for future accelerator requests. As long as there is no data dependency across devices, the two devices can run in parallel and do not need to synchronize.

4.2.1.3 Request-level Parallelism

After relocating the data dependencies and unleashing the device-level parallelism, we can completely offload a sequence of requests to the accelerator. We can also extract request-level parallelism locally on the accelerator to maximize the performance gain. In our example, since instruction `3` is independent of previous accelerator instructions, shown in (d), it can bypass previous requests or interleave with other requests. The only limitations would be the number of requests the accelerator can handle simultaneously and the accelerator's compute throughput.

The observed optimization opportunities above are innate to the program. Optimizing for one aspect does not necessarily affect another. However, when the resource is limited, there could be coupling effects between them, *i.e.*, reordering the requests may affect both locality and parallelism.

4.2.2 Assessing the Cost of Instruction-level Acceleration

Knowing these opportunities exist, *how do we estimate the performance of instruction-level acceleration?* To answer this question, we propose a qualitative model that helps developers assess the performance of instruction-level acceleration and identify optimization opportunities in their software or hardware implementation. The model aims to provide an intuition of observed optimization opportunities and a reference for developers to optimize their programs.

Figure 4.2 shows our approach to evaluating the optimization opportunities of instruction-level acceleration. The y-axis is the average latency for an accelerator request. The x-axis is the accelerator's local data reuse rate. A higher local reuse rate suggests more opportunities to relocate cross-device dependencies to local dependencies. Parameters and symbols are annotated in the figure and thus omitted in the text.

The most straightforward implementation has no optimization. The average latency $L_{original}$ is the summation of the cost of all the sequential operations starting from the host initiating the

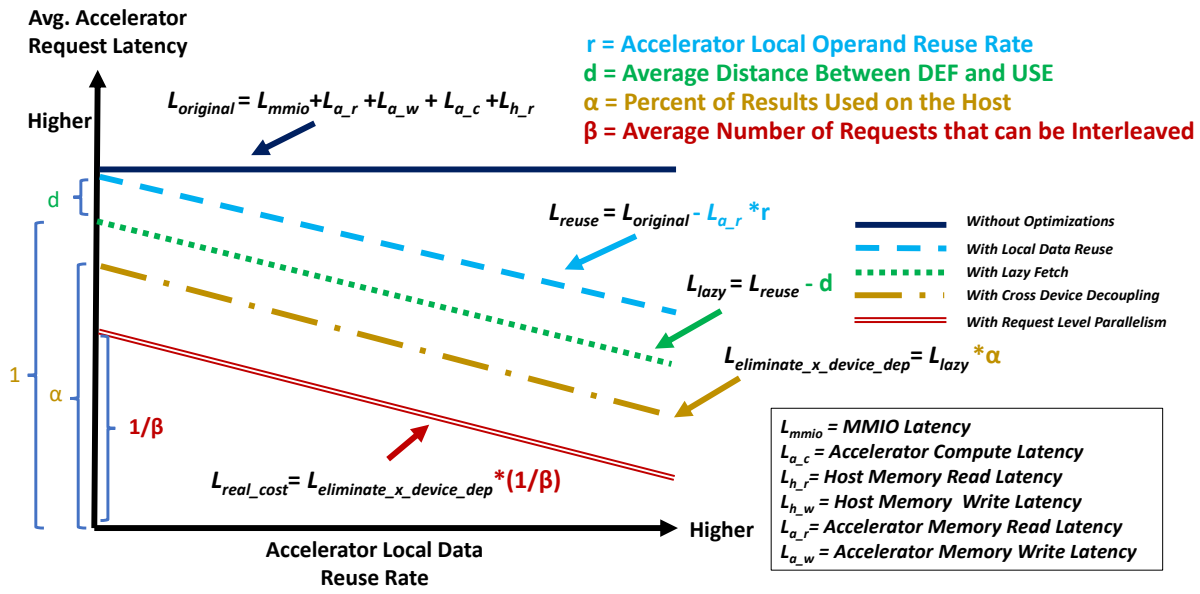


Figure 4.2: Zipper model factors in multiple parameters to identify optimization opportunities.

request until it receives the result, which includes a MMIO `Write Request` from the host to the accelerator, a pair of `Mem Read` and `Mem Write` from the accelerator, the computation latency on the accelerator, and a `Mem Read` request for the host to fetch the result back.

Then, we start from the first optimization opportunity: *exploiting temporal locality*. We assume the accelerator data reuse rate is r. Reusable data is a result that will be used as input operands for subsequent requests. We do not need to access the shared memory to get those operands, and thus save $L_{a_r} * r$ from $L_{original}$.

After relocating the dependencies, we can start to enable device-level parallelism. We classify accelerator requests into two categories: results used on the host or results used solely by subsequent accelerator instructions. We assume α percentage of requests fall into the first category and the rest fall into the second category. For the first category, the host needs to block execution to fetch the results from shared memory temporarily. However, if the host does not need the result immediately after issuing, the host can continue execution and only block when the result is required. Both the host and the accelerator can proceed in parallel during this period. Assume the average time difference between the host issuing the request and the host requiring the result is d; we can deduct this d from our estimate because the system is not affected during this period. Their cost can be optimized entirely for the second category of requests because the host does not need to block those requests under any circumstances.

The last step is to enable request-level parallelism. Assume the level of parallelism is β, where β is the average number of interleaving requests the host can send to the accelerator simultane-

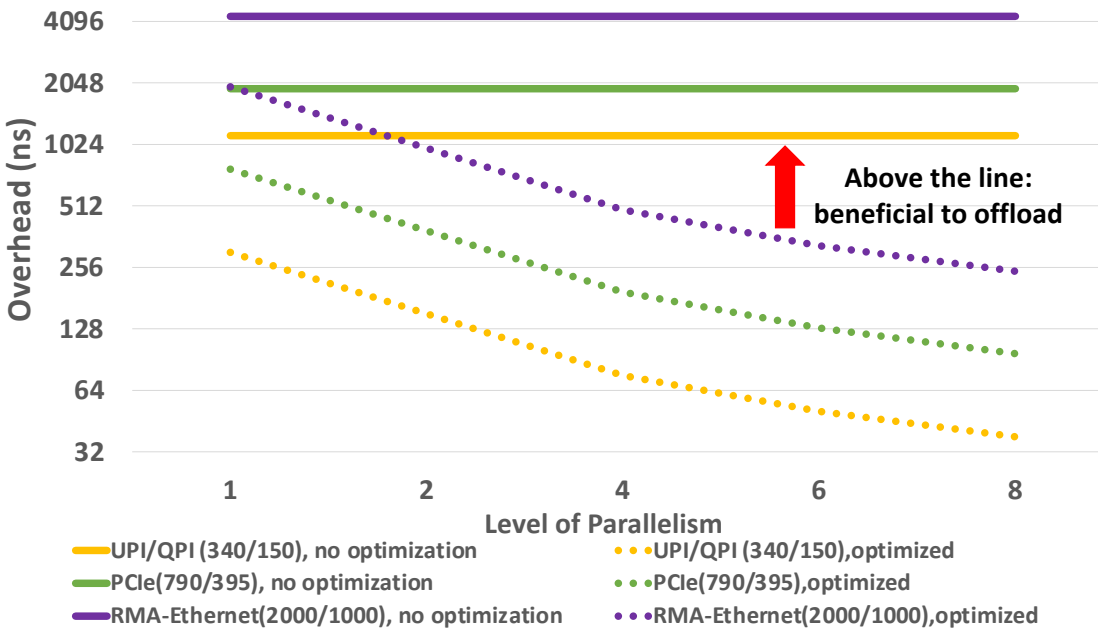


Figure 4.3: Comparison of the overhead of offloading among different bus interfaces, level of parallelism, and the presence of optimizations. Numbers in parentheses represent the estimated read/write latency in nanoseconds [100, 150].

ously; the overall cost the system needs to pay for these β requests is approximately the same as one request, assuming there is enough computing resource.

Taking all these factors into account, we arrive at the equation for our average latency of an accelerator request:

$$L_{real\_cost} = \frac{(L_{mmio}+L_{a\_r}*(1-r)+L_{a\_w}+L_{a\_c}+L_{h\_r}-d)*\alpha}{\beta} \tag{4.1}$$

4.2.3 Pushing the Boundary of Kernel Offloading

We use the model to estimate the potential overhead reduction with an optimal scheduling and offloading strategy. Figure 4.3 shows the overhead of offloading across three types of popular bus interfaces for accelerators—UPI/QuickPath Interconnect (QPI), PCIe, Ethernet—and a comparison between the unoptimized and optimized implementations over different levels of request-level parallelism. The solid lines represent overhead without optimization, and the dotted lines represent the optimized versions. In the study, we assume the operand reuse rate is 50% and the host-side result utilization rate is 23%, which will be further discussed in §5.4.

As a rule of thumb, it is beneficial to offload the kernel when the time difference between emulating on CPUs and computing on the accelerator is greater than the communication overhead.

Without any optimization, the offloaded kernel has to compensate for microsecond-level communication overhead for the optional offloading to be beneficial. Moreover, the performance cost would be rather formidable if the software had to offload specific operations to a remote entity, as in many security applications. In comparison, when optimally optimized, the amortized penalty per request can be reduced to tens of nanoseconds. This would open up more design space for fine-grained offloading, and the overall system would be more friendly to security applications.

4.3 Architecting Zipper Optimizations

While the analytic model helps developers evaluate the optimization potential of an application and a system setup, extracting values from those opportunities is not easy. To enable those optimizations, the system will need to:

- **Support device-level and request-level parallelism:** To enable device-level parallelism, the interface must be non-blocking on the host side, correctly honor data dependencies, and fetch accelerator request results. The interface must also handle interleaving requests to provide exploitable request-level parallelism opportunities for the accelerator. This optimization hides communication latency by decoupling and overlapping instructions.
- **Enable accelerator-side caching to eliminate temporally adjacent data transfers:** to relocate cross-device dependencies and eliminate temporally adjacent data transfers, the host needs to analyze data dependencies, track data movements, and manage coherence between the two devices. The accelerator requires an additional structure to cache recent results and eliminate memory transactions through the data bus.
- **Exploit memory coalescing opportunities:** For many data buses, each read access will fetch a chunk of continuous memory data, usually one complete cache line or 64 bytes. By coalescing multiple operands, we can improve the efficiency of data bus bandwidth.

To bring these optimizations to instruction-level acceleration, we propose *Zipper*. **Zipper is a protocol layer that resides between the physical bus and the application logic, and thus it does not require any changes to the compiler, compute kernel, or the underlying data bus.**

4.3.1 Overview of Zipper

Zipper uses a set of communication semantics that capture the locality and dependency information to connect the host and accelerator. Zipper adds a request buffer table to the accelerator that tracks the status of operands and caches recent request results. On the host side, Zipper uses a

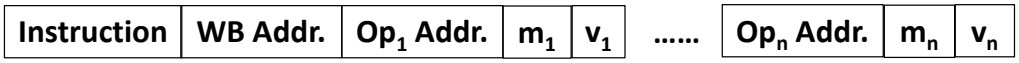


Figure 4.4: Zipper request layout template. WB = write back, m = mode, v = version.

runtime library to analyze data dependency and catch the data reuse opportunities by observing and tracking accelerator requests. The runtime library also manages communication between the host and the accelerator, hiding tedious implementation details from software developers. The rest of this section will first describe the communication protocol between Zipper host and accelerator, Zipper's hardware structure, and then explain Zipper's approach for enabling device and request-level parallelism, accelerator-side caching, and memory coalescing.

4.3.2 Zipper Host-Accelerator Communication Protocol

In Zipper, the host sends requests to the accelerator through MMIO and communicates input operands and results with the accelerator through shared memory.

Figure 4.4 shows the template of a Zipper request. The exact number of bits in each field and the number of fields can vary depending on the use case. As a rule of thumb, each request should include `Instruction`, `Write-back Address`, and `Operand Information`. Each request can have multiple operands. Due to a recent request, each operand can reside in shared memory or within the Zipper hardware structure. The request includes three pieces of information for each operand to help hardware fetch the correct operands:

- **mode:** The `mode` bit tells whether the operand is in the shared memory or the hardware buffer table.
- **address:** This is the operand's location in the memory or the buffer, depending on where it resides.
- **version:** Zipper uses the `version` bit to distinguish whether the value in the memory is input to the most recent request or expired input for the past requests. §4.3.6 will provide a detailed explanation.

The shared memory is the communication channel between the host and accelerator for input operands and results. We partition the shared memory into the operand partition and the result partition. The input operands are continuously placed in the operand partition and wrapped over to reuse the old memory when it reaches capacity. The result partition maintains the bijection with the accelerator-side buffer table entries.

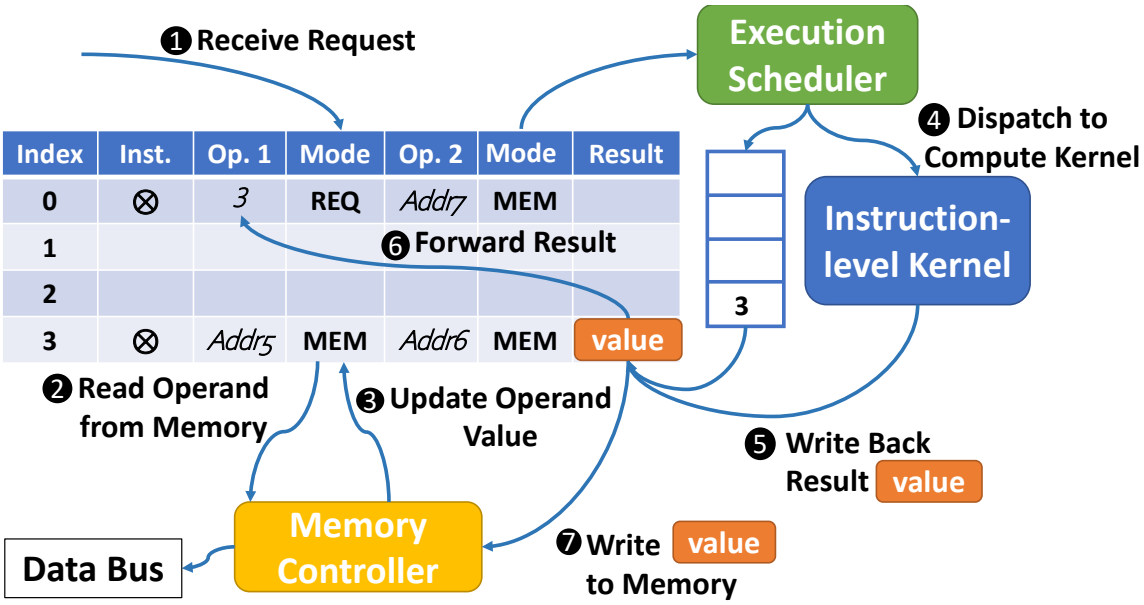


Figure 4.5: Zipper hardware structure and life cycle of an accelerator request. Some design details are omitted due to space limitations.

4.3.3 Zipper Hardware Structure

Zipper hardware resides on the accelerator side and handles requests it receives from the host. It consists of four parts, as shown in Figure 4.5: a request buffer table, an execution scheduler, a memory controller, and the accelerator. The memory controller is platform-specific, and the instruction-level kernel is user-specific. Zipper does not need to make intrusive modifications to these two components to work.

Zipper uses the request buffer table and the execution scheduler to enable request-level parallelism. In ❶, when Zipper hardware receives a request from its software counterpart, typically through MMIO, it will first store the request information in the request buffer table. The request buffer table stores and tracks all details of pending and recently completed requests, including the instruction, the status of each operand, and the write-back address, among others. The request buffer table can be of different sizes. We will discuss the impact of buffer size in §4.5.2. The execution scheduler decides which request is ready for execution and can dispatch instructions out-of-order. The scheduling logic prioritizes older requests when multiple requests are ready to be dispatched.

The request buffer table also caches recent results until new requests take the entries. Zipper hardware then reconstructs the data dependency chain based on the information embedded in the requests. For each accelerator instruction, if its operand values have not been fully resolved, Zipper will fetch their values based on the information provided in the request. ❷ If the operand is in

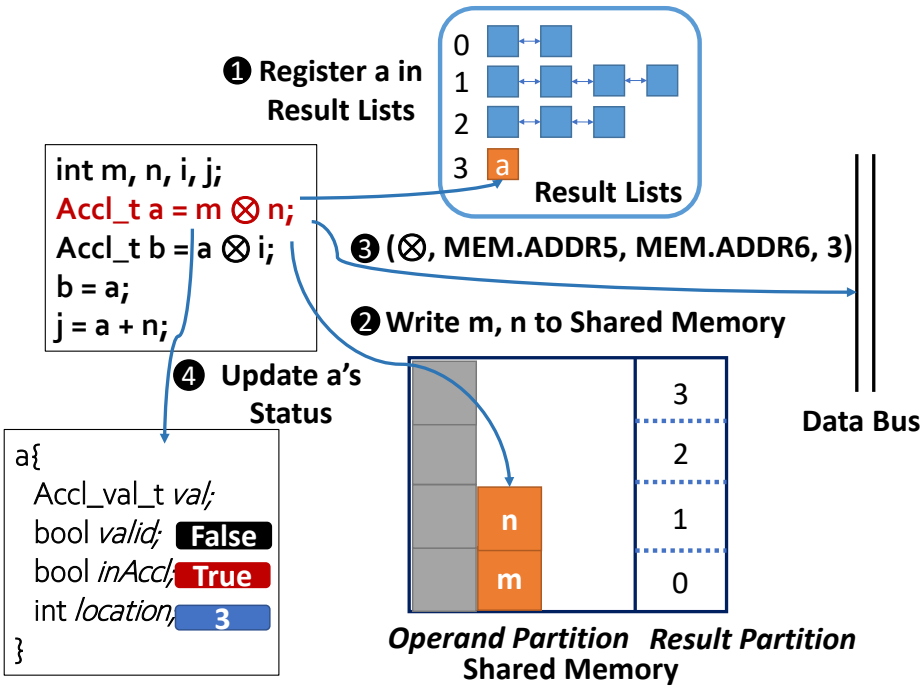


Figure 4.6: Issuing new request to accelerator

the memory, Zipper will issue a read request to the memory controller and mark it as "in fetch" to avoid duplicated read requests. If the value of the operand comes from a prior request, Zipper will either fetch the value if it is ready in the buffer table or wait until the prior request has been completed. ❹ Once all operands have been resolved, we will mark this request as ready to be dispatched. ❺ When the computation is done, Zipper will store the results back in the buffer table and write the results into their corresponding write-back address in the memory, shown in ❼. ❻ If there are pending requests whose inputs are dependent on the newly completed request, Zipper will directly forward the value when the result has been written into the buffer table.

Due to memory coalescing, updated and stale data can be packed into one cache line. Zipper hardware checks the freshness of each operand in the cache line and opportunistically fetches them into the request buffer table based on the version number. §4.3.6 will provide more details.

4.3.4 Latency-Tolerant Accelerator Interface

In Zipper, providing a non-blocking host-side interface that tolerates multiple pending requests within the accelerator's resource limitation is essential.

On the host side, Zipper conducts dependency analysis, request scheduling, and result fetching in software with a runtime library. Zipper provides well-packaged data classes to host applications as if they were host-native types. These data classes encapsulate overridden functions and neces-

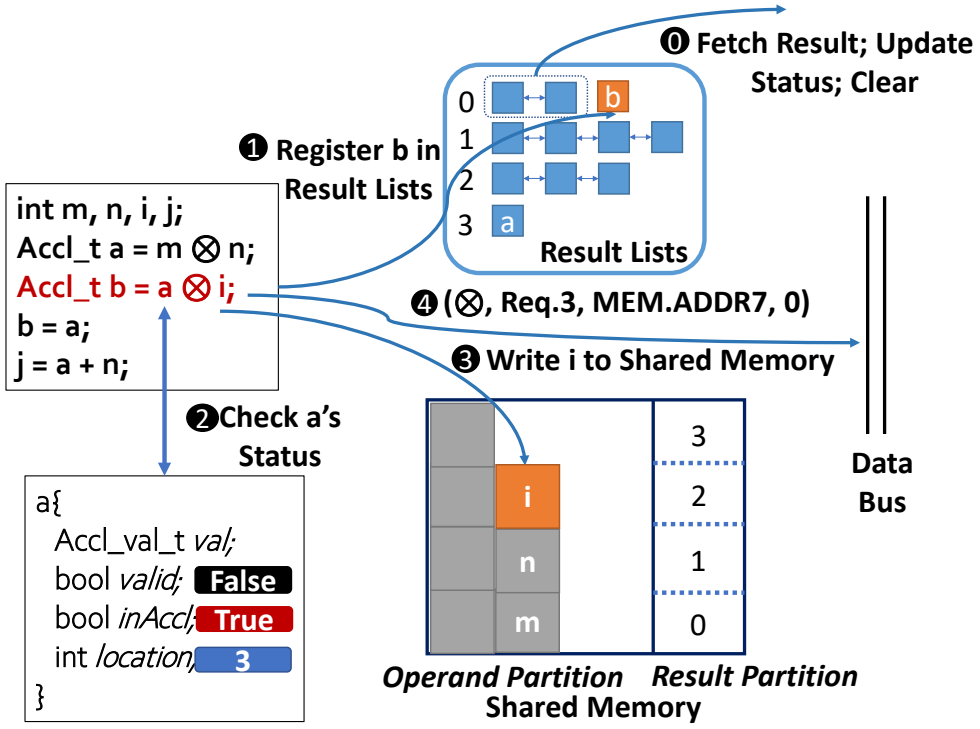


Figure 4.7: Issuing new request with operand reuse

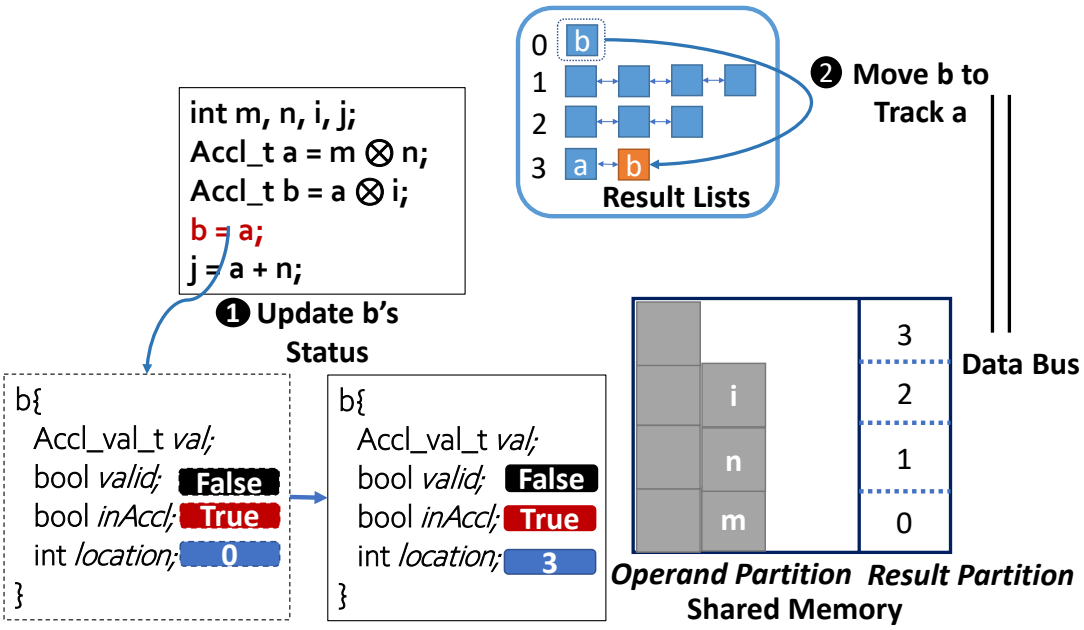


Figure 4.8: Object reassignment

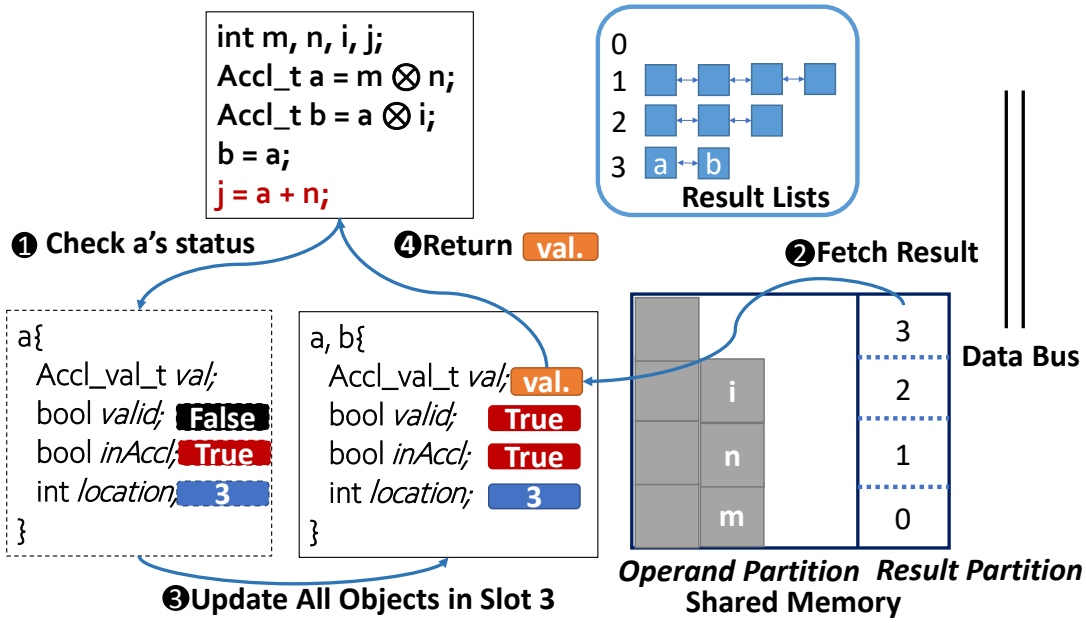


Figure 4.9: Fetching results from accelerator

sary metadata. This approach also provides flexibility and dynamic scheduling capabilities without compiler modification. Figures 4.6, 4.7, 4.8, and 4.9 show Zipper's software data structures and corresponding updates when performing different functions.

The Zipper runtime library maintains two data structures to track data objects and communicate with the accelerator: class objects and result lists. To track the status of accelerator results, Zipper captures the following information into data objects:

- **value:** The value of the result if it has been fetched from the shared memory to the host memory.
- **valid:** A `Boolean` type marks whether the value is ready to use or if we need to access shared memory for results.
- **inAccl:** A `Boolean` type indicates whether this result is present in the Zipper buffer table. This is useful when the runtime library explores opportunities for reuse.
- **location:** If the result is present in the buffer table or shared memory, Zipper needs to track its location precisely.

The result lists track all the software data objects associated with each hardware-side buffer table entry. Whenever Zipper fetches results from the accelerator or clears a table entry for a new

request, we iterate through the list and update all relevant data objects. In this way, Zipper can track multiple in-flight requests.

We will use a snippet of code shown in the figure to demonstrate the operation of the Zipper runtime library and its non-blocking nature. In this code snippet, we first calculate an accelerator request and its result, *a*, using inputs from the host, and then calculate another variable, *b*, reusing *a*'s value. After these two accelerator requests, we reassign *b* to *a*. Lastly, we fetch *a*'s value from the accelerator and return it to the host.

4.3.4.1 Issuing New Request

Figure 4.6 shows the procedures when Zipper issues a new accelerator request to the accelerator. ❶ Within the context shown in the figure, Zipper checks the result lists and finds slot number 3 is available, so Zipper registers object *a* into slot number 3 as object *a*'s value is mapped to the result at this location. ❷ Since input operands *m* and *n* are not present in the accelerator buffer, Zipper will store them in the shared memory and send their relative location in the shared memory as part of the request to the accelerator. In the last step ❹, Zipper updates *a*'s validity as `false` and marks it to be inside the accelerator at location 3. After this step, Zipper does not wait for *a*'s value to be ready; instead, it continues to the next host instruction.

4.3.4.2 Object Reassignment

When reassigning an object to track another object, as in Figure 4.8, Zipper changes the data structure to reflect this reassignment. In the figure, we reassign *b* to variable *a*. ❶ Zipper copies *a*'s metadata to *b* and moves *b* away from its original slot in the result lists to the same slot as *a*. Similarly, Zipper removes the object from the result list when the object is deleted. The host can proceed without stalling since Zipper does not fetch *a*'s value.

4.3.4.3 Lazy Fetch

Zipper never proactively makes memory access to fetch results until they are needed, and this strategy is called *Lazy Fetch*. As we continue the execution of the code, the host eventually asks for the value of *a* to proceed, shown in Figure 4.9. In this case, Zipper fetches *a*'s result from its tracking location 3. If the result is not ready, the host will stall because it cannot continue execution without it. Once Zipper fetches the value from the shared memory, it will update *a*'s value and its metadata to set *valid* bit as true. Zipper will also update all objects that track location 3. However, *a*'s value is still in the accelerator buffer as no new request evicts *a* yet, so we keep *inAccl* as `true` and keep tracking *a*'s location in the hardware buffer table.

4.3.4.4 Overloading Host Operators

In Zipper, host operators that involve accelerator-computed variables are overloaded in the runtime library. When the host calls overloaded operators, dependency and status checks are invoked, and the host will fetch the result from shared memory if it has not already been retrieved. Algorithm 4.3 shows an example of a host-native addition with an accelerator-computed result as one of its input operands.

4.3.5 Enable Accelerator-Side Caching

While the Zipper hardware caches recent request results upon compute completion, it lacks the execution context to utilize the data efficiently. Data dependency information has to be shared by the host. We use the following example to show that Zipper detects temporal locality and issues requests instructing the accelerator to reuse data in the buffer.

In Figure 4.7, Zipper needs to make another accelerator request. However, there are two major differences from the last request issued: 1. There is no empty table entry available in the hardware. 2. One of its input operands, a, is the result of a prior request. To make space for this new request, Zipper first clears the oldest entry as shown in Step ⓿. Zipper forces each object mapped to slot 1 to fetch its value to host memory if it has not already, and updates it as it is no longer in the accelerator's buffer. During the analysis stage, Zipper detects that variable a is at location 3 of the accelerator buffer, so Zipper will not write a to shared memory and does not need to fetch a's value back. Instead, Zipper constructs the request to inform the hardware that it should get a's value directly from buffer table slot 3. In this way, Zipper detects the relocation opportunities on the host and utilizes the hardware buffer to exploit them. We then append b to the result lists and update its metadata similar to what we did to a in the last request. Algorithm 4.2 summarizes the procedures to send a new request to the accelerator.

4.3.6 Exploiting Memory Coalescing

For typical instruction-level workloads, input operands are usually smaller than 64 bytes. This leaves another optimization opportunity. To reduce the number of memory accesses, Zipper packs multiple operands for different requests into a single cache line. When the accelerator reads a cache line from memory, it may contain multiple operands. These operands may be inputs for accelerator requests that Zipper receives after the memory read request is issued. Since a race condition exists between the host write and the accelerator read, and the ordering between the two events is uncertain, Zipper attaches a version bit to each operand. The Zipper runtime library flips the version bit every time it recycles the same operand slot. The Zipper accelerator verifies

```
Data: A list of n input objects Ops and Instruction inst
Result: An Object r that tracks the result
r.valid ← false
 r.inAccl ← true
 r.location ← NextSlot++
 if ResultLists[NextSlot].Occupied then
    for obj ← ResultLists[NextSlot] do
       obj.FetchResult
    end
end
Request req
 req.inst ← inst
 for i ← (0 → n−1) do
    if Ops[i].inAccl then
       req.Ops[i].mode ← BUF_TBL
        req.Ops[i].location ← Ops[i].location
    end
    else
       req.Ops[i].mode ← MEM
        req.Ops[i].location ← NextMemOperandSlot
        NextMemOperandSlot ← (NextMemOperandSlot + 1)%MaxNumOfSlots
    end
end
send req
 return r
```

Program 4.2: Procedures to issue new requests in Zipper runtime library.

```
Data: A host-native variable a and an accelerator-involved variable b
Result: The host-native addition of the input operands
if !b.valid then
    for obj ← ResultLists[b.location] do
       obj.FetchResult
    end
end
return a + b.val
```

Program 4.3: Overloading of host-native addition for accelerator-computed variables.

Table 4.1: System Configuration

Host CPU	Intel Xeon CPUs (E5-2699v4)
Host Frequency	2.2 GHz
FPGA Type	Arria10 GX1150
Interconnection	QPI
Bus Interface	CCI-P

the freshness of the operand by matching the version bit with the version bit it receives from the runtime library. It only fetches fresh values; otherwise, it issues a new read request to the shared memory to fetch the input operands.

4.4 Evaluation

To evaluate the performance of Zipper, we evaluated two applications, each of which is a representative example of the use cases discussed in §2.4.

In the first application, we replaced the floating point representation in the C++-ported The NAS Parallel Benchmarks (NPB) [35] with a Posit32 number representation. Posit is a number format that achieves better precision than floating points but currently lacks native hardware support. All the Posit number computations are computed with a hardware compute kernel. Each operand and result is 32 bits in size.

In the second application, we implemented hardware isolation support for the integer subset of VIP-Bench [24]. VIP-Bench is a set of algorithms implemented in a data-oblivious manner where only the SE hardware enclave can see the plaintext values of the secrets. We prototyped the SE enclave on an FPGA, and all privacy-enhanced operators are offloaded to the SE enclave. In this application, every value is encrypted under a 128-bit Advanced Encryption Standard (AES) key; thus, all operands and results are 128 bits in size. VIPBench and NPB represent interesting applications in privacy and High-Performance Computing (HPC) domains, which are gaining traction and benefit greatly from fine-grained offloading.

4.4.1 Experiment Setup

We conducted our experiments on Intel HARPv2 [41] with an in-package FPGA. The detailed configuration is shown in Table 4.1. CPU core and FPGA are connected with QPI [72] and CCI-P [74] with a 64K FPGA-side coherent cache. The software code in the Zipper runtime library runs on the CPU, and the Zipper-enabled compute kernel is synthesized and runs on the Arria10 FPGA.

We used an 8-entry buffer table design for the NPB Posit32 accelerator and a 2-entry design

Table 4.2: Logic overhead of Zipper over the baseline accelerator design.

	NPB w/ Posit (8-entry)	VIP-Bench w/ SE (2-entry)
Adaptive Logic Modules (ALM)	80,957	87,784
ALM Overhead	4.3%	0.9%
Registers	88,140	88,140
Registers Overhead	1.2%	0.3%
Block RAM (BRAM)	277	275
BRAM Overhead	0%	0%
DSP Blocks	6	6
DSP Blocks Overhead	0%	0%

for the VIP-Bench benchmarks running on the SE enclave design. We will elaborate on the buffer entry choice in §4.5.2. The baseline implementation is an unoptimized design where each request is issued and executed sequentially.

4.4.2 Performance Speedup and Logic Overhead

Figure 4.10 shows the relative performance of Zipper over the baseline design. On average, Zipper provides 8x speedup for NPB with Posit32 and 1.5x speedup for VIP-Bench with the SE enclave.

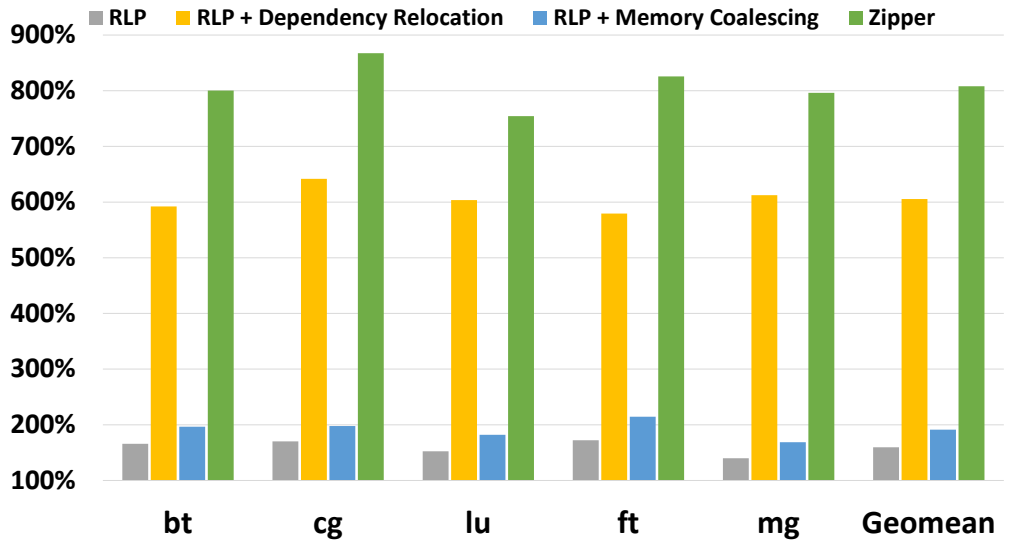
We synthesized our design using Intel Quartus Pro Version 16.0.0.211 onto the targeted FPGA platform. Table 4.2 summarizes the number of ALM and Registers used by Zipper compared to the baseline design. The logic overhead of Zipper optimizations is only 4.3% for the 8-entry Zipper Posit32 design over the baseline Posit32 design and 0.9% for the 2-entry Zipper SE enclave design over the baseline SE enclave design.

4.5 Discussion

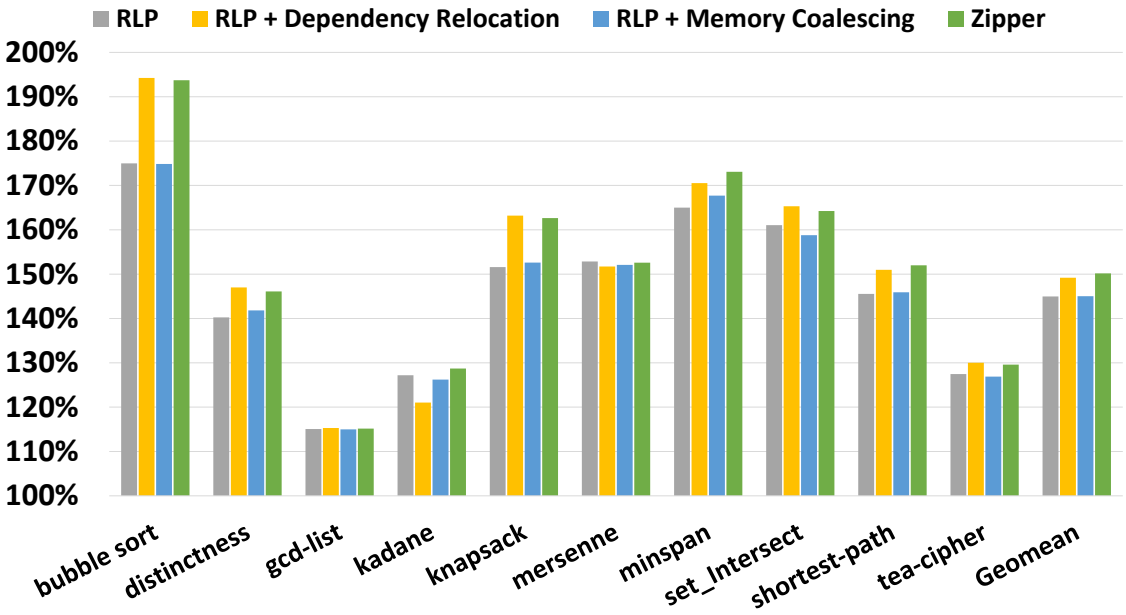
This section provides an in-depth discussion of the impact of various optimizations, request buffer table sizes, workload profiles, and other relevant design aspects.

4.5.1 Accelerator Memory Access

Zipper's performance benefits greatly from reducing accelerator memory demands by exploiting temporal locality and memory coalescing. Figure 4.11 shows the percentage of the bus transactions Zipper and other de-featured design options make over the baseline design.

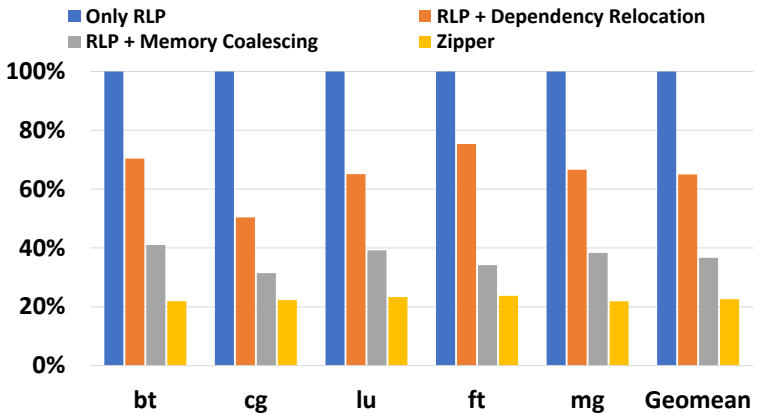


(a) NPB with Posit

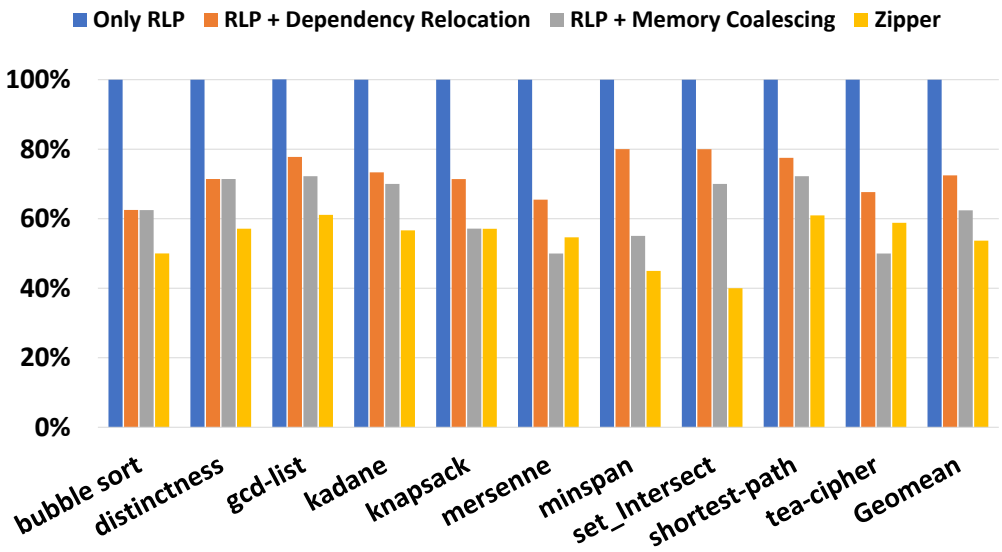


(b) VIP-Bench with SE Enclave

Figure 4.10: Relative performance of Zipper and various de-featured Zipper over the baseline. Note that RLP does not exploit locality.



(a) NPB with Posit32



(b) VIP-Bench with SE Enclave

Figure 4.11: Comparison of the number of bus transactions by accelerator between Zipper, de-featured Zipper, and the baseline.

	1	2	3	4	5	6	7-19	20/+	
bt	27%	3%	2%	1%	2%	1%	4%	36%	NPB with Posit
cg	26%	24%	0%	0%	0%	0%	0%	48%	
lu	30%	5%	2%	2%	1%	1%	7%	31%	
ft	25%	0%	8%	0%	0%	0%	0%	50%	
mg	33%	1%	3%	0%	0%	0%	3%	47%	
Average	28%	7%	3%	1%	1%	0%	3%	42%	
bubble-sort	25%	25%	12%	0%	0%	0%	0%	37%	VIP-Bench with SE Enclave
distinctness	29%	0%	14%	0%	0%	0%	0%	0%	
gcd-list	22%	11%	11%	11%	0%	0%	0%	0%	
kadane	27%	13%	7%	13%	0%	13%	0%	0%	
knapsack	29%	0%	14%	0%	0%	0%	0%	14%	
mersenne	35%	6%	6%	3%	3%	0%	0%	9%	
minspan	20%	10%	0%	5%	0%	5%	0%	30%	
shortest-path	22%	11%	0%	0%	0%	0%	2%	64%	
set-intersect	20%	20%	0%	0%	0%	0%	0%	0%	
tea-cipher	32%	3%	12%	6%	9%	0%	11%	0%	
Average	26%	10%	8%	4%	1%	2%	1%	15%	

Figure 4.12: Distribution of distance between accelerator request result and operand reuse.

# of Buffer Table Entries	2	4	8	
bt	1.88	3.52	6.54	NPB with Posit
cg	2.00	4.00	6.13	
ft	1.99	3.99	7.67	
lu	1.77	3.02	5.21	
mg	1.81	3.14	5.31	
Average	1.89	3.53	6.17	
bubble_sort	1.67	1.00	1.00	VIP-Bench with SE Enclave
distinctness	1.67	2.33	3.67	
gcd-list	2.00	3.00	4.69	
kadane	1.83	2.33	1.83	
knapsack	2.00	3.66	6.32	
mersenne	1.81	2.86	4.48	
minspan	2.00	3.32	4.65	
shortest-path	1.99	2.95	4.85	
set-intersect	2.00	3.00	5.00	
tea-cipher	1.70	2.41	2.89	
Average	1.87	2.69	3.94	

Figure 4.13: Request-level parallelism with 2/4/8 buffer entries.

# of Buffer Table Entries	0	2	4	8	
bt	100%	47%	38%	31%	NPB with Posit
cg	100%	50%	3%	2%	
ft	100%	50%	33%	33%	
lu	100%	43%	30%	21%	
mg	100%	34%	27%	21%	
Average	100%	45%	26%	22%	
bubble_sort	100%	100%	33%	33%	VIP-Bench with SE Enclave
distinctness	100%	33%	0%	0%	
gcd-list	100%	50%	33%	0%	
kadane	100%	83%	50%	0%	
knapsack	100%	33%	0%	0%	
mersenne	100%	44%	19%	6%	
minspan	100%	67%	22%	22%	
shortest-path	100%	50%	25%	25%	
set-intersect	100%	50%	0%	0%	
tea-cipher	100%	41%	29%	17%	
Average	100%	55%	21%	10%	

0% represents a non-zero but negligible number.

Figure 4.14: Percentage of request results to fetch back to the host.

# of Buffer Table Entries	0	2	4	8	
bt	0.001	157.46	204.83	218.16	NPB with Posit
cg	3.731	7.20	134.91	157.22	
ft	1181.5	2315.8	3547.5	3347.3	
lu	0.001	28.79	42.31	54.72	
mg	74.078	208.72	274.06	328.12	
Average	251.86	543.59	840.71	821.11	
bubble_sort	42.11	42.42	128.11	133.17	VIP-Bench with SE Enclave
distinctness	0.04	0.08	0.02	0.02	
gcd-list	0.03	0.07	0.10	8.99	
kadane	0.08	0.11	0.17	10.58	
knapsack	2.74	8.16	2195	2197.8	
mersenne	562.10	1235.4	2926	8775	
minspan	4.18	6.00	18.45	19.30	
shortest-path	33.31	66.79	135.46	130.89	
set-intersect	5.27	10.74	11166	11161	
tea-cipher	0.10	0.07	0.14	0.36	
Average	65.00	136.98	1656.9	2243.7	

Figure 4.15: Timing distance (microseconds) between host request issue and fetch.

For NPB with Posit32, Zipper reduces the accelerator's bus transactions by 77% from the baseline design. RLP enables out-of-order execution but does not reduce any memory access. Since each operand is 32-bit in size and, with metadata, we can pack eight operands into one cache line. As a result, memory coalescing reduces 63% of bus transactions over the baseline design. Dependency relocation exploits temporal locality and data reuse, reducing 34% of bus transactions over the baseline Posit32 design.

Since operands are much larger in the VIP-Bench with SE enclave design, we cannot pack the operands as tightly as with Posit32 numbers. Therefore, it is more likely that the operands for the same request span two cache lines, which leads to more memory access, and there are fewer opportunities for memory coalescing in this workload. Zipper reduces 46% of the bus transactions while memory coalescing and dependency relocation reduce 37% and 27% of the bus transactions over the baseline SE enclave design, respectively.

It is worth noting that memory coalescing and dependency relocation are not necessarily compatible, especially when only few operands can be packed into a single cache line. Dependency relocation can impact data alignment, leading to increased memory access, as we observed in *Mersenne* and *tea-cipher*.

4.5.2 Size of the Reuse Window and Performance Impact

To study how many entries we need to provide in Zipper for different workloads, we analyzed the distance of the data dependency chain in Zipper requests and the number of entries required for efficient dependency relocation. Figure 4.12 shows the percentage distribution of reuse distance over the number of all request operands. We classify distances of 19 requests or fewer as exploitable temporal locality. Our experiment results show that 91% and 92% of the temporal locality can be captured with only four buffer entries for two workloads, respectively. We further profile more aspects of the applications under different numbers of buffer entries as discussed in §4.2. Figure 4.13 shows the number of requests that can be processed in parallel within the different sizes of instruction windows, assuming any dependency on the result before the start of the window under study has been resolved. As we increase the number of buffer entries, Zipper can exploit more parallelism, but faces diminishing returns. Figure 4.14 shows the percentage of results required to be fetched back into the host memory. Figure 4.15 shows the average time distance (*in microseconds*) between the host issuing a request and the hosting using the request result. As Zipper harvests more operand reuse with larger buffers, the percentage of results that need to be fetched decreases as more request dependencies get relocated. The average time distance also increases as fewer results are fetched back to the host, giving the host more time to execute host-side codes in parallel. Note that this analysis assumes the system has perfect knowledge of instruction dependencies

during runtime. In practice, Zipper always fetches results back when the buffer entry gets recycled to ensure functional correctness. A larger buffer gives the host more time to continue execution until it needs to recycle a buffer entry. However, the results shown in Figures 4.14 and 4.15 are still significant, as they suggest potential acceleration opportunities.

We then analyzed the performance and logic overhead of various sizes. We construct our experiments around the buffer size of 4. The results are shown in Figure 4.16. The logic overhead increases exponentially as the buffer size increases because we need more logic for scheduling and more space to store results and operands.

Interestingly, Zipper for NPB with Posit32 and VIP-Bench with SE Enclave manifest distinct performance characteristics. For NPB with Posit32, the speedup increases logarithmically as we put more entries in the buffer table. However, VIP-Bench with SE Enclave's performance only increases slightly with more buffer entries. The difference is attributed to the latency of each compute kernel. The Posit32 kernel takes two cycles to complete an instruction while SE Enclave kernel takes 24 cycles for each instruction. The SE Enclave is more compute-bound on the kernel itself.

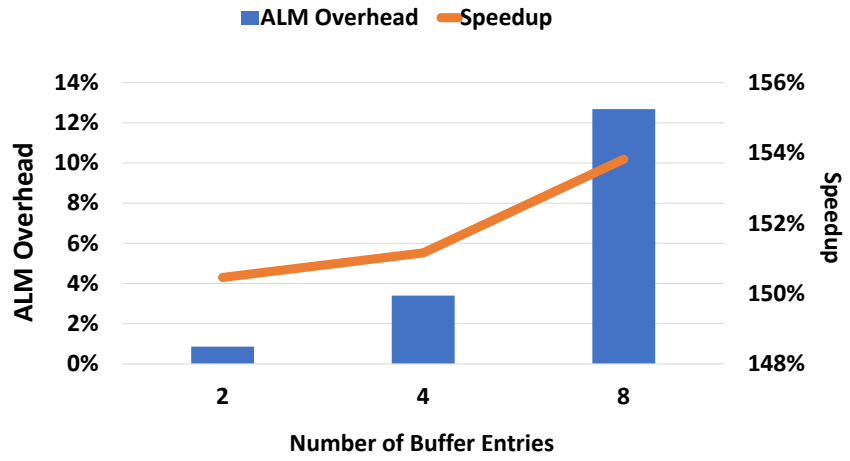
4.5.3 Impact of Different Optimizations

With insights into memory access and the impact of buffer size, we can better understand how different optimizations affect the overall performance of other applications. Since each optimization requires engineering effort to implement, a breakdown can help developers further understand the benefits of each design decision and make more cost-effective choices.

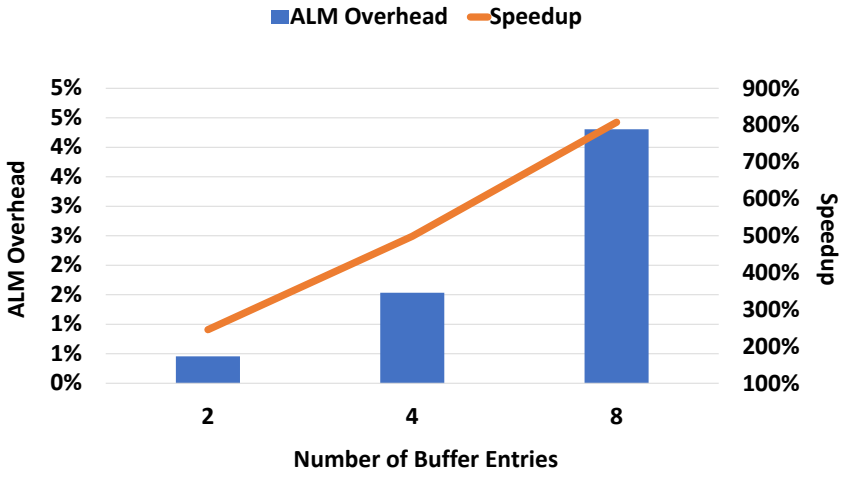
Request-level parallelism does not affect the memory access and offers 1.4x-1.6x speedup by having the host issue multiple requests to the accelerator in parallel. Meanwhile, dependency relocation provides significant performance improvement through exploiting temporal locality to reduce the number of memory accesses. Small kernels would benefit more from these two optimizations. The benefit diminishes for larger kernels, where each instruction takes more time to compute. This is because the workload becomes compute-bound and blocks execution, leading to request serialization.

Memory coalescing works well for compute kernels with smaller input operands, as it allows us to pack multiple operands into a single cache line. It contributes to the compound effect with request-level parallelism and dependency relocation, as Zipper can serve more requests with a single memory access. For larger operands, alignment will play a role in determining the benefit that Zipper can eventually extract.

Although Zipper exploits the program's innate features, in practice, optimizations are limited to a single scheduling window. As a result, moving the request locations would affect device-level



(a) NPB with Posit32



(b) VIP-Bench with SE Enclave

Figure 4.16: Impact of different numbers of buffer table entries on performance and area for Zipper.

parallelism but not request-level parallelism, memory coalescing, or locality. Switching request orders would affect all aspects of the system.

4.5.4 Support for Multi-tenant Time Sharing

In an actual deployment, an accelerator may be shared by multiple hosts in a time-shared manner. In Zipper, there could be false dependencies if the switch between different tenants is not handled correctly. In the very likely case that tenants do not share the same memory space in communication with the accelerator, there is no danger of data dropping. However, the accelerator needs to complete all the requests from the previous tenant before changing to the new memory space. When the new tenant initiates requests, it must assume all data in the accelerator buffer is stale and send input operands through the memory. Suppose two tenants share the same memory space. In that case, Zipper fetches results back on behalf of the previous tenant before relinquishing the buffer entry to the new tenant and factors in tenant information when calculating value reusability.

4.5.5 Comparison to the Model Projected Performance

While Zipper optimizes away significant communication overhead, Zipper still has to fetch results back to the host to ensure correctness. This overhead can only be optimized if the program has no runtime branches and jumps. After factoring in all factors, the model projects that Zipper will speed up NPB with Posit and VIPBench with SE by 12.1x and 4.3x, respectively. We can achieve 8x and 1.5x in the actual deployment for these two applications. Besides the fact that SE is bounded by the compute, as discussed above, we identified that software overhead and API inefficiency are two other significant factors that prevent us from achieving the full potential. For SE, data movement also takes more time because it uses larger data types, which is harder to overcome without a wider bus to compensate. Access to lower-level APIs and hardware optimizations would help Zipper further improve the speedup.

4.5.6 Comparison to Address-Indexed Cache and Integrated Registers

Other solutions to exploit temporal locality include *address-indexed caching*, such as in [139, 160, 138], and *integrated register files* in the accelerator, such as in [120, 62, 133].

Address-indexed caching deploys coherent storage for the shared memory space, providing fast access to recently used data. Note that the platform used for the experiment baseline already includes a 64KB coherent cache maintained by CCI-P on the FPGA side [3]. However, the cache fails to capture locality due to a lack of support for properly tracking request completion, controlling data movement, and managing value dependencies. Even with sufficient tracking and scheduling,

caching can only optimize memory access latency, while Zipper removes some memory access entirely.

Integrated register files provide fast temporary storage within the accelerator pipeline and rely on explicit `Load` and `Store` instructions to manage values in the registers.

Both approaches induce significant design complexity. Integrated register files even require intrusive modifications to user designs. More importantly, existing systems lack software and compiler support on the host side to manage data dependencies between two separate compute domains automatically. Moreover, many vendors limit access to their accelerators and FPGA state by forcing all interactions through a high-level API. This software interface approach creates significant hurdles for integrating compiler support.

In comparison, Zipper can exploit request and device-level parallelism that existing solutions cannot due to its reduced demands on system software and the compiler. Furthermore, Zipper has the following advantages in system deployments:

- **Platform-Agnostic:** Zipper works for any platform regardless of underlying communication protocols and other architectural supports. The acceleration opportunities Zipper exploits are self-contained within the workloads.
- **Portability:** Zipper software is implemented as runtime libraries that can be easily reconfigured, recompiled, and linked to different applications and machines, which provides excellent portability.
- **Extensibility:** Zipper can be easily extended to support more devices, optimized scheduling algorithms, and instruction sets by simply updating the library file. This is more approachable than extending compiler support.

4.6 Limitations

While Zipper demonstrates tremendous performance improvement over the baseline, there are additional improvements we can explore as the continuation of this line of work.

Request Reordering: Zipper leverages the optimization opportunities that applications present. However, there would be more temporal locality by reordering the requests and exploiting operator commutativity and associativity. To achieve this, Zipper can create and maintain a request buffer on the software end and issue requests in batches after optimizing requests within a scheduling window.

More Accurate Analytic Model: The Zipper model helps identify optimization opportunities when assuming program statistics are even out for the whole program. A dynamic and epoch-based

model would be more insightful for developers to balance trade-offs across different system setups and application behaviors. We must quantitatively formulate correlations between workload features, platform parameters, and performance gain to project the performance benefits. More powerful frameworks would be necessary to profile workloads, hardware platforms, and the accelerator design in more detail.

Simulation Framework: While an analytic model can be helpful, simulation can provide more accurate performance predictions. Complete profiling and simulation tools are essential to study other use cases further.

Multi-Agent Cooperation: We considered the scenario with only one accelerator in this work. Multiple accelerators can cooperate to complete the computation in a more complex system. Zipper poses well to enable such extensions that the developer can use optimized scheduling algorithms to dispatch requests to different accelerators.

4.7 Chapter Summary

In conclusion, this chapter identifies opportunities for optimization to tolerate latency in high-performance data buses. By relocating data dependencies, heterogeneous systems can exploit parallelism and temporal locality to improve performance. We introduced an analytical model to evaluate the performance of applications that rely on instruction-level acceleration. We further proposed *Zipper*, a protocol optimization layer that extracts the performance from the system by capitalizing on temporal locality and parallelism. Zipper uses runtime library support for dynamic scheduling and an additional hardware structure to execute requests from the host. Deployed on Intel's HARPv2 platform, Zipper achieves a 1.5x-8x speedup with low logic overhead for the two case studies included in this dissertation.

CHAPTER 5

Efficiently Allocating Communication Resources in Heterogeneous Systems

5.1 Introduction

As Moore's Law [112] and Dennard Scaling [46] have been withering over the past decade, system designers have increasingly turned to heterogeneous designs to achieve an optimal balance of cost, power, and performance. Unlike the homogeneous systems of the past, heterogeneous systems integrate diverse components that work together to maximize system-level efficiency. ARM's big.LITTLE architecture [16] exemplifies this design philosophy, with similar approaches subsequently adopted by AMD and Intel [38, 76].

While these heterogeneous components execute their respective workloads independently, they typically share the same network and memory subsystems—*i.e.*, interconnects, and memory devices. Each component's bandwidth and memory demands fluctuate dynamically with workload characteristics, making shared resource pools more cost-effective than dedicating separate memory devices to each component. However, efficiently allocating these shared resources to maximize system performance remains a significant challenge. Poor allocation can lead to considerable memory access delays and degraded performance, as shown in prior work [58, 49, 29, 43].

With the emergence of chiplet technology [122] and CXL [64], system designers now have the flexibility to assemble systems from heterogeneous components sourced from multiple vendors, tailoring configurations to specific workloads. This chapter focuses on optimizing the allocation and regulation of interconnect and memory access, arguably the most critical shared resources, for heterogeneous systems assembled through chiplet and/or CXL technology. These systems introduce unique challenges: the system topology remains undefined during hardware design and manufacturing, and component performance characteristics can vary significantly from one configuration to another. Further complicating matters, individual components may be agnostic to the system's topology and unaware of other coexisting elements, thus lacking the global perspective necessary for system-wide optimization.

Existing solutions fall short when applied to such highly reconfigurable heterogeneous systems. While modern interconnect designs support static bandwidth allocation or traffic prioritization, they fail to account for broader system-wide implications. Software-based solutions can dynamically adjust bandwidth but rely on centralized control nodes that must be reprogrammed to support new topologies and participating agents, which introduces engineering overhead and limits scalability. Hybrid solutions like Intel Memory Bandwidth Allocation (MBA) [40] and solutions built on top of it, *i.e.*, EMBA [161], rely on features supported by a specific architecture from a particular vendor and are neither extensible to heterogeneous components nor scalable to support large-scale clusters. Resource partitioning works, like [31, 32, 167, 30], leverage online learning and/or machine learning models to perform space search for optimal partitions of memory bandwidth. Still, all of them fall back on Intel MBA as the sole actuator and enforcer of partition policies and thus do not cater to heterogeneous hardware systems.

To address these limitations, we propose **Overpass**, a flexible interconnect architecture designed to optimize performance in heterogeneous systems. An Overpass Interconnect comprises multiple Overpass router nodes that can be connected into arbitrary topologies. Each node autonomously profiles runtime performance without prior knowledge and then adjusts upstream bandwidth allocation based on optimal performance combinations over recent time intervals. Overpass decentralizes bandwidth allocation and memory arbitration decisions, distributing them across router nodes. Each node communicates only with its neighbors and uses this localized information to solve a bounded knapsack problem [25], determining an optimal bandwidth allocation strategy. Bandwidth allocations are then enforced through a novel debt-opportunistic credit-based arbitration mechanism that regulates access priorities for connected agents and neighboring nodes. Collectively, the router nodes form a scalable interconnect system that manages traffic and memory access at each hop without relying on a centralized controller. Additionally, Overpass can dynamically throttle agent-side prefetchers based on observed traffic patterns, preventing bandwidth over-consumption.

Traditionally, system-level performance monitoring and bandwidth regulation have been handled by a centralized ”supervisor” node. However, in the emerging paradigm enabled by chiplets and CXL, the interconnect becomes the only hardware component with visibility into all system components and the contextual knowledge needed for fine-grained bandwidth control. **Overpass is the first interconnect architecture designed specifically for this paradigm shift.**

We evaluate Overpass in a system of 8 compute agents interconnected via Universal Chiplet Interconnect Express (UCIe) and sharing a single memory device through CXL. Each agent continuously performs General Matrix Multiply (GEMM) using Single CPU Core Matrix Multiplication Benchmarks (MMPerf) [110]. Our experimental environment is built using The Structural Simulation Toolkit (SST) [131] and DRAMsim3 [97], both of which are extended with CXL capabilities.

We measure the number of completed GEMM operations over 1.6 milliseconds following a 400-microsecond warm-up, totaling 2 milliseconds of execution. Results show that Overpass delivers a 35% average system-level performance improvement over the best baseline arbitration mechanism. Each Overpass router node introduces less than 3% area overhead relative to a baseline NoC router design [14], which is negligible given that NoC typically occupies only around 8% of the total system area in an 8-core design [92], with routers constituting just a part of that.

5.1.1 Chapter Organization

The remainder of the chapter is organized as follows: **§5.2** describes the Overpass architecture and its mechanisms for bandwidth allocation, traffic arbitration, and prefetcher throttling. **§5.3** details the experimental setup and presents evaluation results. Design optimizations and their performance implications are discussed in **§5.4**. Limitations and future directions are presented in **§5.5**, followed by the chapter summary in **§5.6**.

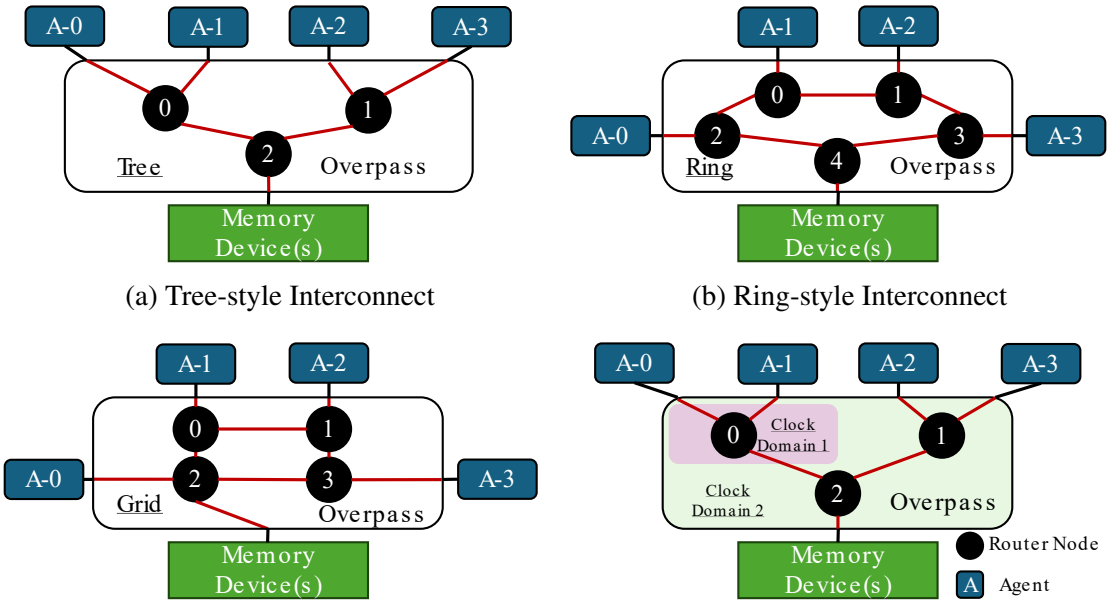
5.1.2 Chapter Contributions

This chapter makes the following contributions:

- studying current limitations of the existing works and the challenges to build a flexible interconnect for high-performance heterogeneous systems.
- proposing a composable interconnect design that optimizes memory traffic for heterogeneous systems. To the best of our knowledge, this is the first interconnect traffic management mechanism based on a modular design that can form any topology without requiring software intervention or significant hardware changes.
- presenting a novel debt-opportunistic credit-based arbitration mechanism that balances crossbar utilization and bandwidth enforcement.
- evaluating Overpass's performance benefits and area overheads with an eight-agent system.
- analyzing and comparing various Overpass setups and management strategies of Overpass.

5.2 Overpass Interconnect

Overpass Interconnect is a modular interconnect design that optimizes the overall system performance by allocating and prioritizing communication bandwidth. Each router passes information to neighboring nodes and implements local bandwidth allocation policies. When interconnected,



(a) Tree-style Interconnect

(b) Ring-style Interconnect

(c) Grid-style Interconnect

(d) Interconnect with two clock domains

Figure 5.1: Examples of Overpass Interconnects forming various topologies composed of Overpass routers.

Overpass routers form a more extensive interconnect system that is flexible enough to create any topology and regulate bandwidth for the entire system.

Overpass supports the following features:

1. **Performance monitoring**: Overpass does not require any a priori knowledge of the workload. Overpass collects the performance profile of each agent over time and updates it periodically as agents proceed with the execution.

2. **Bandwidth tracking and dynamic allocation**: Overpass tracks the bandwidth usage at the router level and adjusts the bandwidth allocated to each traffic direction accordingly. Overpass's dynamic bandwidth adjustment naturally adapts to any topology without any node knowing the complete picture of the topology.

3. **An efficient dynamic adaptive arbiter:** To realize the bandwidth allocation policy set by each router, Overpass employs a dynamic adaptive arbitration mechanism that combines LRU tie-breaking with a credit-based scheme to decide traffic priorities. The arbitration logic of Overpass is free of starvation and deadlock. The design is similar to SuDO [66] but with improvements that address SuDO's limitations, which will be detailed in §5.2.4.

4. **Prefetch throttling:** Overpass monitors the prefetching behaviors of each agent and node. If

Table 5.1: Overpass commands and their usage across interfaces.

Command	Description	Data Fields	Interface
REP_PSCORE	Report performance score	perf_score: uint_16	Agent-Overpass, Internal
ALLOC_BW	Allocate bandwidth	new_bw: uint_8	Internal
THRTL_PF	Throttle prefetcher	timeout: uint_16	Agent-Overpass, Internal
RST_PROFILE	Reset profile	N/A	Agent-Overpass, Internal

the interconnect system is stressed, the router under stress can ask upstream nodes to throttle the prefetching requests.

Each Overpass system is comprised of interconnected Overpass routers. Every Overpass router acts independently based on the information the upstream and downstream nodes provide. This autonomous and homogeneous design enables Overpass to be self-composing. Hence, it functions optimally for any topology without re-programming the bandwidth management algorithm every time the topology changes.

Figure 5.1 shows Overpass configured into four common topologies in the production environment. We will use them as examples as we describe each Overpass feature in detail in the rest of this section.

Assumptions: We assume all compute agents in the system are in good faith and will report their performance proactively and accurately. We assume Overpass Interconnect and the routers in the network can collect device information and build routing tables during the device discovery and setup phase with dynamic routing [107]. Furthermore, we assume application performance monotonically increases with more bandwidth and access priority, a phenomenon observed in prior work [145].

5.2.1 Overpass Interfaces

Overpass defines a small set of extensible commands to facilitate performance reporting, bandwidth allocation, and prefetch throttling. Table 5.1 summarizes the commands and their use across the Agent-Overpass and internal router interfaces.

Agent-Overpass Interface: The communication between agents and Overpass includes the following:

- REP_PSCORE: Agents self-report a 16-bit unsigned integer performance score. This score reflects the agent's performance during the last session window. The metric can be system-defined—*e.g.*, instructions per second, throughput, latency violations, or a composite score of multiple metrics.

- `THRTL_PF`: When a router experiences congestion, it can issue this throttling signal upstream. The message includes a timeout (in microseconds) indicating how long the prefetcher should remain throttled.
- `RST_PROFILE`: When workloads or devices are swapped, the agent can ask Overpass to reset its performance and bandwidth history using this command.

Internal Router Interface: Routers communicate among themselves using these commands:

- `REP_PSCORE`: Routers propagate aggregated performance scores downstream to facilitate bandwidth decisions.
- `THRTL_PF`: Routers send or forward throttling requests to the upstream until they reach the affected agents.
- `ALLOC_BW`: Routers allocate bandwidth to in-flow directions based on the bandwidth decisions. The decisions are sent to the upstream routers so they can use the allocated bandwidth from the downstream routers as the new optimization target. The allocation is expressed as 8-bit multiples of a base unit defined by the system designer.
- `RST_PROFILE`: Routers forward profile reset requests downstream to clear the history for affected nodes. All nodes on the path from the resetting agent to the destination should have their profiles reset, as outdated records may no longer reflect the agent's updated behavior.

5.2.2 Overpass Router

As shown in Figure 5.2, an Overpass router extends a generic router design with three components:

1. **Bandwidth Controller:** Performs performance tracking and bandwidth adjustment.
2. **Prefetch Throttling Manager:** Detects and regulates prefetch traffic based on congestion signals.
3. **Dynamic Adaptive Arbiter:** Regulates traffic priority using credit-based and LRU mechanisms.

The bandwidth controller observes in-flow traffic to collect real-time usage and performance metrics, bypassing standard buffers for control traffic. It adjusts bandwidth allocations accordingly and informs upstream nodes. It also interacts with switch allocators to enforce updated priorities until the next allocation round.

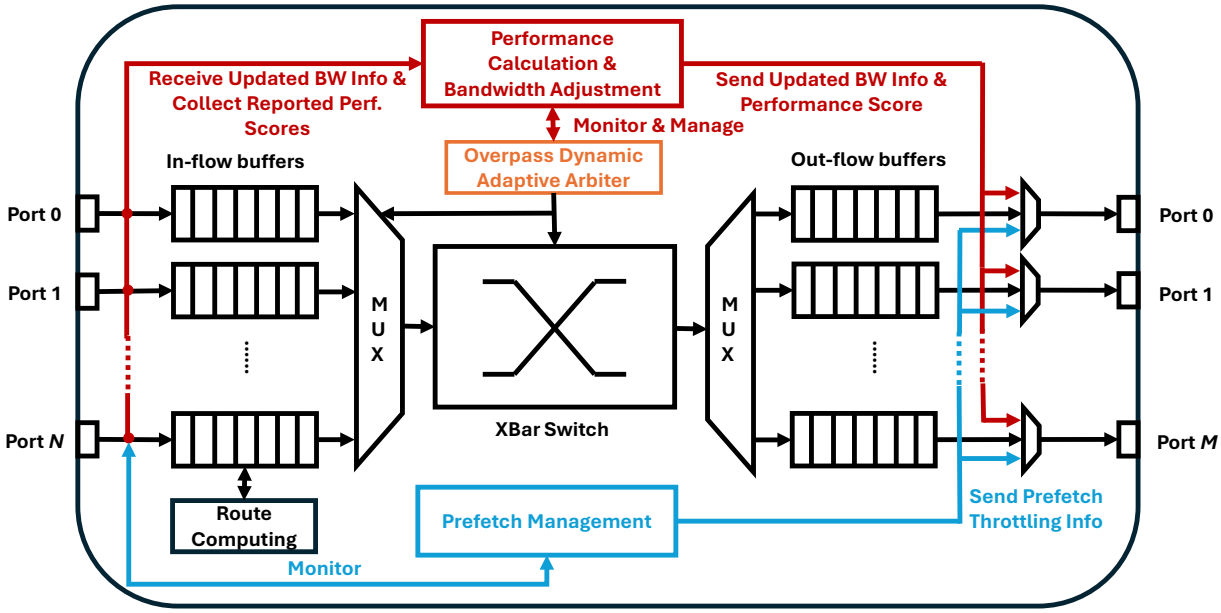


Figure 5.2: High-level diagram of an Overpass router. Credit return paths are omitted. **VC** = Virtual Channel, **XBar** = Crossbar.

The prefetch throttling manager filters `THRTL_PF` messages from incoming ports and decides which upstream nodes should receive the throttling signals. If the observed bandwidth exceeds a configurable threshold, it proactively issues throttling signals upstream.

The dynamic adaptive arbitration logic implemented in Overpass routers will be detailed in Section 5.2.4.

5.2.3 Bandwidth and Performance Information Management

Each Overpass router aggregates the performance scores from upstream nodes and forwards the aggregate downstream. Based on this data, downstream routers then allocate bandwidth among their upstream counterparts and propagate updated allocations back upstream.

Example: In the grid topology (Figure 5.1c), router ❷, closest to memory, first partitions bandwidth among Agent A-0, node ❶, and node ❸. Nodes ❶ and ❸ further subdivide this allocation to their respective upstream nodes. For instance, ❶ allocates to A-1 and ❹, and ❹ allocates entirely to A-2.

5.2.3.1 Cyclic Path and Double Counting Prevention

Overpass maintains a strict directional flow based on routing policies (**e.g.**, in this case, X-Y or Y-X). Performance scores are propagated along the static and preset routing path to the shared

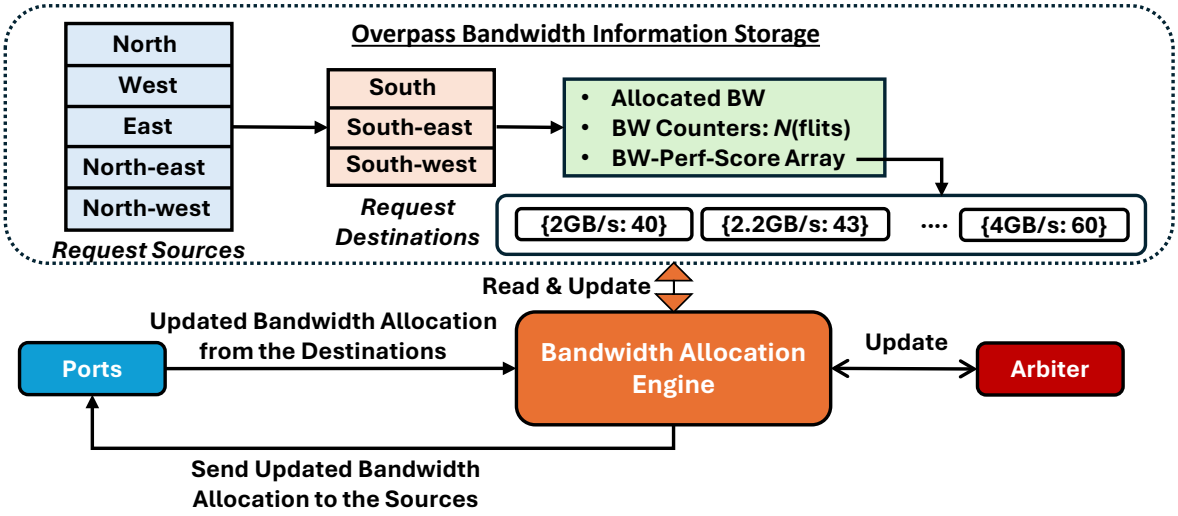


Figure 5.3: Overpass bandwidth allocation engine and information store. Bandwidth-performance profiles are maintained per source-destination direction.

resource, preventing double-counting and ensuring acyclic dependency graphs. A router closer to the shared resources, based on the routing algorithm, always regulates nodes further away.

5.2.3.2 Data Structure

As shown in Figure 5.3, each router maintains a table indexed by source-destination direction pairs. Each entry stores:

- A flit counter that tracks the traffic in flits from one direction to another. This information will be used to calculate bandwidth in the next bandwidth allocation update.
- A bandwidth-performance score array (BW-Perf Score Array) that tracks the recorded bandwidth and its corresponding performance score over time. This information will be used to optimize the system's performance. The number of entries depends on the maximum physical bandwidth and the finest adjustment granularity supported by the router. Specifically, the number of entries equals the maximum bandwidth divided by the adjustment granularity.
- A register that records the currently allocated bandwidth.

5.2.3.3 Performance Score Update

Routers receive performance reports from upstream nodes and associate them with observed bandwidth usage in the most recent session. If multiple reports are received in a session, they are averaged.

5.2.3.4 Bandwidth Allocation Update

Overpass uses the performance score profiled as the proxy for system performance. Overpass models bandwidth allocation problems as a bounded Knapsack problem with constraints, which can be solved with integer linear programming. We set the overall capacity target as the maximum physical bandwidth or the allocated bandwidth from the downstream node, whichever is lower. We then constrain the bandwidth selection of each in-flow direction to the bandwidth currently assigned BW_c or one of the neighboring bandwidths (one basic unit more $BW_c + BW_b$ or one basic unit less $BW_c - BW_b$. BW_b is the smallest unit that the router can adjust). The bandwidth allocated to each direction should be at least BW_b and no more than the physical bandwidth limit BW_{max}. This constraint is implemented to reduce compute overhead and make more gradual adjustments without drastic system disruption. The goal is to maximize the aggregated performance of all upstream nodes and agents. We can formalize the problem as such:

Assume there are N in-flow directions and M out-flow directions. The downstream nodes predetermine the bandwidth of all out-flow directions as

$$BW^{j}_{dest},\ j \in \{0,1,..,M-1\}$$

Similarly, the bandwidth of an in-flow direction to a particular out-flow direction can be annotated as

$$BW^{ij},\ i \in \{0,1,...,N-1\},\ j \in \{0,1,..,M-1\}$$

A lookup function F exists that translates a direction pair and its corresponding bandwidth into an integer performance score. We annotate the performance score returned by the look-up as

$$PS^{ij}_{BW_{ij}} = F(i,\ j,\ BW_{ij})$$

The knapsack problem with constraints for an out-flow direction j can be written as

$$\max \sum_{i=0}^{N-1} PS^{ij}_{BW_{ij}}$$

constrained by

$$\sum_{i=0}^{N-1} BW^{ij} \leq BW^{j}_{dest}$$

$$BW^{ij} \in \{BW^{ij}_c, min\{BW^{ij}_c + BW_b, BW_{max}\},$$
$$max\{BW^{ij}_c - BW_b, BW_b\}\}$$

For bandwidth allocation updates, each direction starts with evenly allocated bandwidth. The update is run every preconfigured time T or after Overpass receives a new bandwidth allocation request from downstream nodes. If a neighboring bandwidth's performance score is unknown, we will add a slight bonus to the current performance score to encourage the system to explore new bandwidth options. The bonus is a configurable parameter of Overpass and can be tuned to larger for a more aggressive adjustment or smaller for a more conservative adjustment. As a rule of thumb, if the observed bandwidth is less than the allocated bandwidth, Overpass prefers allocating less bandwidth during the update by adding a larger bonus to the smaller bandwidth option; if the observed bandwidth is close to or above the allocated bandwidth, Overpass prefers allocating more bandwidth.

5.2.4 Arbitration in Overpass Router

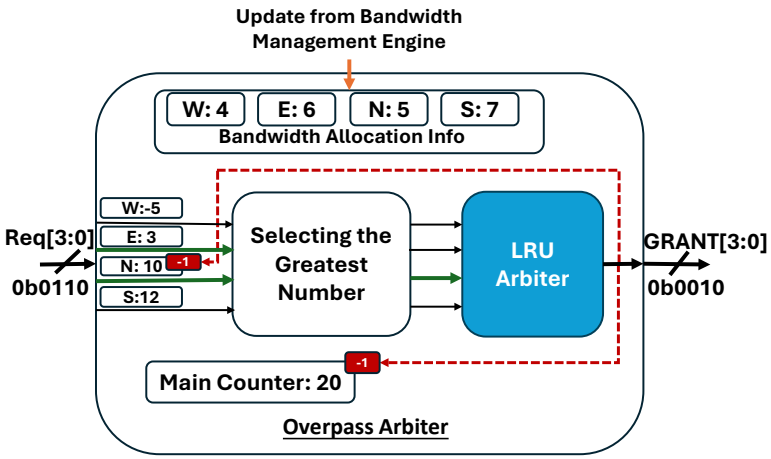
Overpass arbitration enforces dynamic bandwidth allocations while ensuring fairness and avoiding starvation. It combines a credit-based scheme with LRU tie-breaking. Overpass arbiter is debt-opportunistic because it allows the requesters to overdraw credits to maximize crossbar utilization, contrary to the common practice of stalling access if the credit runs out.

Each requester receives credits proportional to its bandwidth allocation. Credits are tracked in per-port counters and a shared main counter.

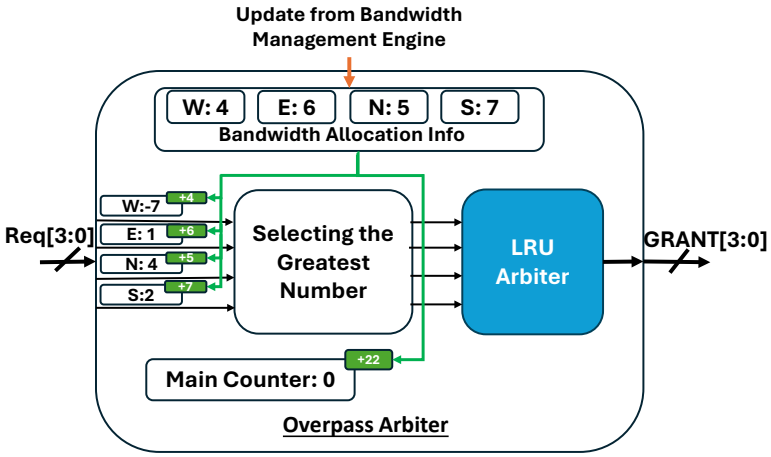
Arbitration: When multiple requests are active, the arbiter selects the one with the highest credit (Figure 5.4a). If tied, LRU is used. Credits are decremented on flit transmission. Requesters can overdraw credits (resulting in negative values) if they are the only active requester.

Refilling: When the main counter is exhausted, all counters are refilled based on the latest bandwidth policy (Figure 5.4b). Refilling occurs every time the credits in the main counter run out. The credits only reset when the current session ends and the bandwidth allocation is updated. This allows flexible overdraw and deferred fairness enforcement: the requesters that start with fewer credits will eventually get selected after the ones with more use up their credits.

Comparison to SuDO [66]: SuDO is an arbiter that uses a similar debt-opportunistic approach. Overpass arbiter differs from SuDO in two ways: First, Overpass uses a main counter to control the refill interval, whereas SuDO only refills when all per-port credits are used up. This helps Overpass maintain relative priority when one or a few requesters hold off the refill and others cut into debt. Second, Overpass arbiter uses LRU as the tie-breaker instead of RR, which avoids starvation issues.



(a) Arbitrating crossbar access



(b) Refilling credit counters

Figure 5.4: Illustration of Overpass arbiter in action.

Table 5.2: Specification of the simulated hardware system.

Component Name	Description
CPU Core	Ariel Core, max 4 Issue per cycle
Cache	512KB, 8-way associative
Prefetcher	Stride prefetcher, detect range of 4
Die-to-die Latency	10ns
Off-chip Latency	50ns
Interconnect Frequency	1GHz
Memory Device	8 channel DDR4@3200MT/s

5.2.5 Prefetch Throttling

When bandwidth usage exceeds a configurable threshold, a router issues `THRTL_PF` signals upstream with a timeout value. These signals propagate hop by hop until they reach agents, which are instructed to reduce prefetch activity temporarily.

Though conceptually simple, this mechanism is highly effective under network stress (see §5.3). Since the compute agent lacks global visibility, its reactive throttling may be suboptimal. Overpass offers a proactive alternative by issuing throttling signals from points of congestion.

Note that not all agents support adjustable prefetchers. However, even without agent-side prefetch throttling, Overpass still delivers significant performance gains through its adaptive bandwidth management alone.

5.3 Evaluation

We evaluated Overpass using SST [131], a widely adopted environment for high-performance system simulation, and DRAMsim3 [97] for modeling memory device behavior. We extended SST with a CXL host engine, which translates memory requests into CXL messages, and a CXL Device Coherency Agent (DCOH) to support shared memory across devices. The Overpass Router was implemented atop SST's high-radix generic router module.

5.3.1 Experiment Setup and Configurations

- **Base system:** Our simulated system consists of 8 CPU cores grouped into four compute chiplets. These chiplets connect to a CXL-enabled DDR4 memory backend via a central IO die. Each core-cache complex includes a CXL host engine, which translates memory requests into CXL messages before they hit the interconnect system. ECore subsystems operate independently, unaware of the broader system. Each core-cache complex has a stride

Table 5.3: Experiment setup configurations.

<table>
<tr><th colspan="2" rowspan="2">Setup Name
(Letter Symbols)</th><th colspan="4">Hardware Configurations(freq.)</th><th colspan="4">Workload Tiling Size</th><th rowspan="2">Network
Maximum Bandwidth</th></tr>
<tr><th>Die0</th><th>Die1</th><th>Die2</th><th>Die3</th><th>Die0</th><th>Die1</th><th>Die2</th><th>Die3</th></tr>
<tr><td rowspan="2">Hardware Cascade
(HC-4/8)</td><td>Core0</td><td>1GHz</td><td>2GHz</td><td>2GHz</td><td>4GHz</td><td colspan="4" rowspan="4">16</td><td rowspan="10">4GB/s or 8GB/s</td></tr>
<tr><td>Core1</td><td>1GHz</td><td>2GHz</td><td>2GHz</td><td>4GHz</td></tr>
<tr><td rowspan="2">Hardware High-Low
(HHL-4/8)</td><td>Core0</td><td>1GHz</td><td>1GHz</td><td>4GHz</td><td>4GHz</td></tr>
<tr><td>Core1</td><td>1GHz</td><td>1GHz</td><td>4GHz</td><td>4GHz</td></tr>
<tr><td rowspan="2">Hardware Alternating
(HA-4/8)</td><td>Core0</td><td>1GHz</td><td>1GHz</td><td>1GHz</td><td>1GHz</td><td colspan="4" rowspan="2">16</td></tr>
<tr><td>Core1</td><td>4GHz</td><td>4GHz</td><td>4GHz</td><td>4GHz</td></tr>
<tr><td rowspan="2">Workload Cascade
(WC-4/8)</td><td>Core0</td><td colspan="4" rowspan="6">2 GHz</td><td>4</td><td>8</td><td>8</td><td>16</td></tr>
<tr><td>Core1</td><td>4</td><td>8</td><td>8</td><td>16</td></tr>
<tr><td rowspan="2">Workload High-Low
(WHL-4/8)</td><td>Core0</td><td>4</td><td>4</td><td>16</td><td>16</td></tr>
<tr><td>Core1</td><td>4</td><td>4</td><td>16</td><td>16</td></tr>
<tr><td rowspan="2">Workload Alternating
(WA-4/8)</td><td>Core0</td><td>4</td><td>4</td><td>4</td><td>4</td></tr>
<tr><td>Core1</td><td>16</td><td>16</td><td>16</td><td>16</td></tr>
</table>

prefetcher, initialized to detect a range of 4, meaning a stride is identified if at least 4 blocks are accessing the same stride. The detailed specification of the hardware system is listed in Table 5.2.

- **Workload:** Each compute agent performs element-wise matrix multiplications similar to MMPerf [110] consecutively. Each matrix multiplication is performed as $\mathbf{C}_{784\times256} = \mathbf{A}_{784\times512} \times \mathbf{B}_{512\times256}$ with each element of type `float`, a typical dimension used in ResNet [59]. The same computation with different tiling strategies [153] results in different data reuse rates in cache and instruction counts per multiplication. We use 4, 8, and 16 tiling sizes in our evaluation.

- **System configurations:** We evaluated Overpass in 12 different system configurations. Three different hardware setups run the same workload but mix-and-match slow (1GHz), medium (2GHz), and fast cores (4GHz), and three different workload setups that all cores run at 2GHz but with different tiling strategies. These six setups are given two network conditions: constrained (4GB/s) and sufficient (8GB/s). The configurations are detailed in Table 5.3.

- **Overpass configuration:** The session interval was set to 100μs, with bandwidth adjustment granularity at 100MB/s. We evaluated three variants of Overpass: defeatured Overpass with bandwidth adjustment only, defeatured Overpass with prefetch throttling only, and full-featured Overpass. When prefetch throttling is enabled, Overpass issues the throttling signal with a 100μs timeout when the network is stressed. When the agents receive throttling signals, they increase the prefetcher detect range to 8, resulting in fewer prefetch requests. The detection range returns to 4 after the timeout.

- **Baseline and Performance Measurements:** Performance is measured as GEMM operations per second across the 8-core system. Each run simulates 2ms of system time, with a 400μs warm-up phase. The results are compared against four baseline arbiters: RR, LRU, age-based, and random. In this system, the age-based arbiter is effectively equivalent to the LRU arbiter.

5.3.2 Performance Improvements

Figure 5.6 summarizes the performance gains of Overpass. Across 12 setups, baseline arbiter effectiveness varies: round-robin performs best under bandwidth constraints, while LRU(AGE-based) excels with ample bandwidth.

Overpass is compared to the best baseline design in each setup. The full-featured Overpass provides 35% performance improvement over the best baseline. Overpass can improve performance

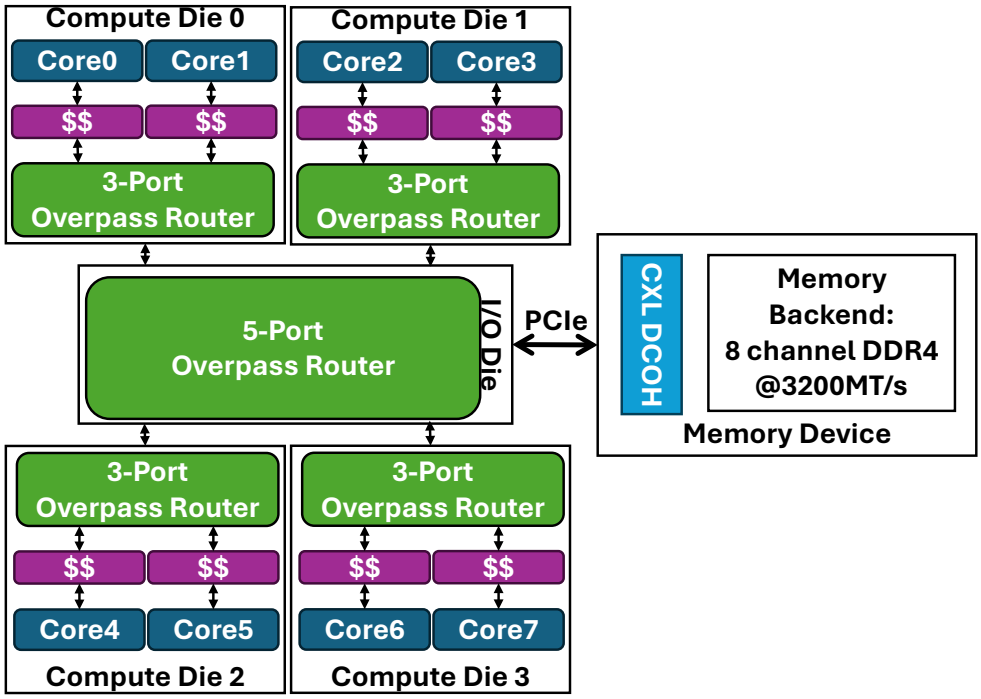


Figure 5.5: The experiment setup where four compute dies are connected to one I/O die in the center for CXL memory access through PCIe.

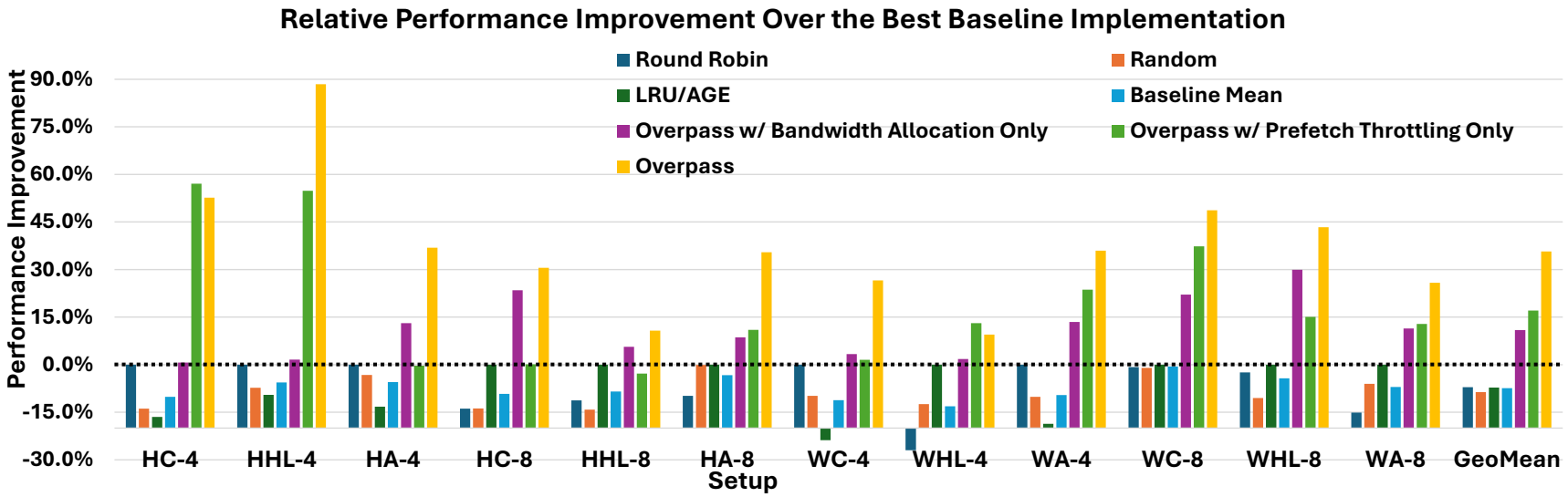


Figure 5.6: Performance improvements of Overpass and its variants over the best-performing baseline. The dotted line denotes the best baseline.

Table 5.4: Area comparison between baseline and Overpass router designs. The unit for area is μm^2.

	5 ports		
Category	**Baseline**	**Overpass**	**Overhead**
Combinational	313,886.8	317,521.6	1.2%
Buf/Inv	19,912.9	20,865.2	4.8%
Non-combinational	10,579.7	12,572.2	18.8%
Total	324,466.5	330,093.8	1.7%
	3 ports		
	Baseline	**Overpass**	**Overhead**
Combinational	117,964.6	120,040.4	1.8%
Buf/Inv	9,424.5	9,920.3	5.3%
Non-combinational	8,679.8	9,788.7	12.8%
Total	126,644.4	129,829.1	2.5%

by 11% with only dynamic bandwidth allocation and 17% with only prefetch throttling. Dynamic bandwidth allocation will have a more considerable impact on the system performance in high system bandwidth scenarios, providing 16.5% performance improvement for setups with a bandwidth of 8GB/s compared to 5.5% improvement when the system has 4GB/s. Conversely, the prefetch throttling impacts systems with constrained bandwidth more, bringing a geometric mean of 22.9% performance improvement to 4GB/s setups and 11.5% improvement to systems with 8GB/s total bandwidth.

5.3.3 Area Overhead

We implemented 3-port and 5-port Overpass routers in SystemVerilog, based on a baseline network-on-chip design [14], with 8-entry buffers per port. Both Overpass and baseline designs were synthesized using IBM's 45nm standard cell library. Table 5.4 shows the Overpass router overhead in each category. Overall, the area overhead is 1.7% for 5-port routers and 2.5% for 3-port routers. The most significant increase in area is for non-combinational logic, which Overpass uses to calculate bandwidth allocation, record credit, and update performance status.

5.4 Discussion

This section analyzes Overpass's runtime behavior and highlights how different features contribute to system performance. We explore the varying effectiveness of Overpass across configurations and provide insights into its decision-making process.

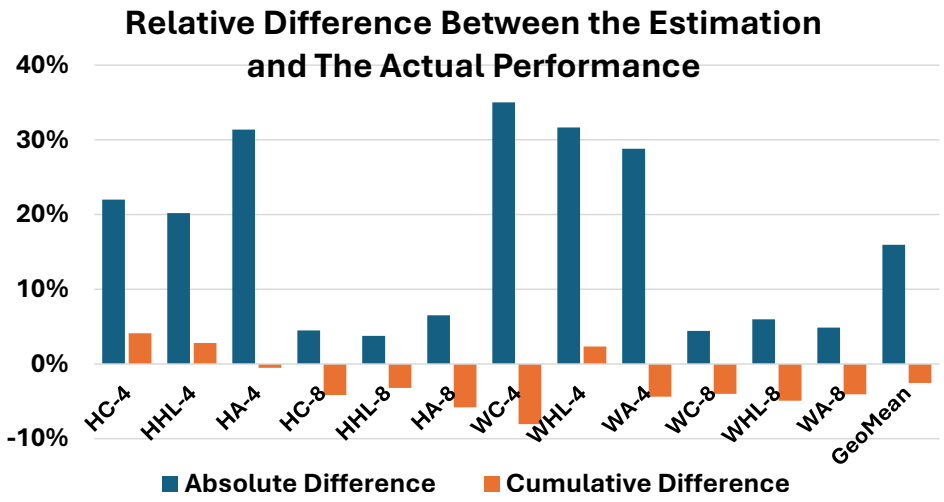


Figure 5.7: Relative difference between the estimated performance after bandwidth allocation and the actual performance.

5.4.1 Predicting the Performance Score

Accurate performance prediction is key to effective bandwidth allocation. Figure 5.7 shows the relative error between the estimated performance (after bandwidth adjustment) and the actual performance observed in the subsequent session. Across all setups, the geometric mean of error is 16%. When calculated cumulatively, where overestimations and underestimations cancel out, Overpass exhibits a 3% net overestimation. This indicates that while system variations and noise make short-term predictions difficult, Overpass's prediction of long-term performance is reasonably accurate.

Bandwidth availability significantly impacts prediction accuracy. With limited bandwidth (4GB/s), the average error rises to 28%, compared to just 5% when bandwidth is ample (8GB/s). This explains why dynamic bandwidth allocation is more effective in high-bandwidth scenarios and highlights the challenge of performance prediction in constrained environments.

Despite these inaccuracies, Overpass relies on relative—not absolute—performance rankings to guide decisions. This allows Overpass to optimize system-wide performance even when prediction precision is imperfect.

5.4.2 The Effectiveness of Bandwidth Allocation

To illustrate Overpass's dynamic bandwidth allocation in action, we examine setup HC-8 with only dynamic bandwidth allocation (without prefetch throttling). Figure 5.8 shows the bandwidth taken by each die on the central I/O for every $100\mu s$ interval. In this setup, Overpass (without prefetching throttling) improves the overall system performance by 23%. For this setup, die0 has

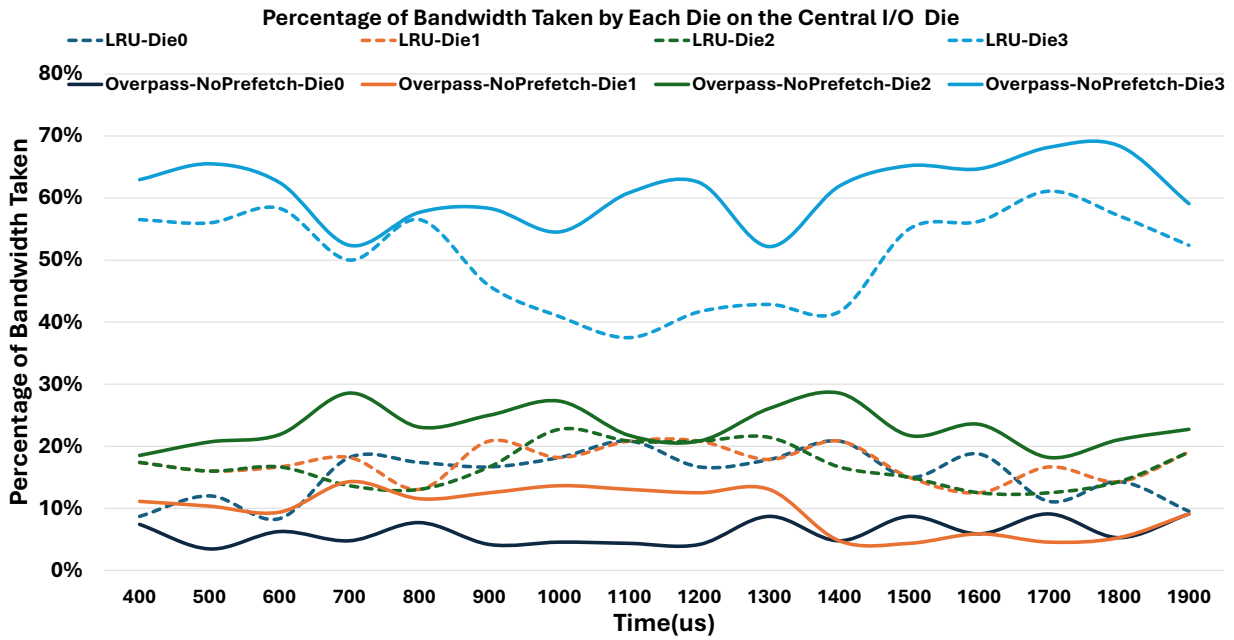


Figure 5.8: Relative bandwidth taken by each die on the central I/O for setup HC-8. The dashed lines indicate LRU (best baseline); solid lines represent Overpass without prefetch throttling. The same die is marked with the same color.

two cores of 1GHz. die1 and die2 each has two cores of 2GHz, and die3 has two cores of 4GHz. Ideally, we would like to serve the faster cores(die3) first so it can output more work and slightly prefer die1 and die2's traffic over die0 because die0 is the slowest die.

Table 5.5 shows the relative bandwidth allocation and performance change of Overpass compared to the baseline (LRU) implementation. While the baseline prioritizes Die3, it lacks finer-grained classification among the other dies. On the other hand, Overpass gives die3 and die2 more bandwidth resources while depriving die1 and die0's bandwidth usage. Even though die1 and die2 have the same configuration, Overpass would prefer one over the other, a behavior that will be further discussed in §5.4.4.

5.4.3 Prefetch Throttling Reduces Memory Bandwidth Pressure and Memory Access Latency

Prefetch throttling proves especially effective in bandwidth-constrained systems where interconnect congestion is a limiting factor. We analyze the setup HHL-4 to understand how reducing prefetch activity leads to performance improvements.

As shown in Table 5.6, enabling prefetch throttling reduces I/O die usage by up to 56%. This leads to a substantial drop in memory requests and a 39% reduction in average memory access

Table 5.5: Relative bandwidth allocation and performance change of Overpass with only bandwidth allocation over the baseline (LRU) implementation.

	B/W Percentage Change	Performance Change	Absolute B/W Usage
Die0(1GHz)	-9.1%	-19%	-43%
Die1(2GHz)	-7.6%	-22%	-60%
Die2(2GHz)	+6.2%	+47%	+43%
Die3(3GHz)	+10.5%	+47%	+30%

latency. These improvements reflect a more efficient memory subsystem and reduced stalling in agent pipelines.

5.4.4 Exploiting Imbalance for Better Performance

As noted in §5.4.2, Overpass improves system throughput by favoring higher-performing agents, often at the cost of fairness. Figure 5.9 illustrates this in WA-4 and WA-8 setups, comparing per-core performance under Overpass (with and without throttling) against the best baseline.

Overpass accelerates some agents while throttling others. This imbalance maximizes system-level throughput, as the gains of the prioritized agents outweigh the losses of the deprioritized ones. This pattern recurs across multiple configurations, confirming that Overpass favors aggregate performance over equal resource distribution.

Although Overpass may not be ideal for situations where balanced allocation is required across the board, it is highly effective for performance-centric applications, such as AI inference and HPC tasks, where maximizing throughput is the primary objective.

5.5 Limitations

While Overpass provides significant performance improvements with low area overhead, it comes with limitations that await further studies.

- **Parameter Selection:** Overpass has many reconfigurable features, *i.e.* the bandwidth adjustment delta, the session length. Currently, it relies on the developers to decide the best parameters that work for the system, and the optimal configuration often depends on the topology and application performance profiles. More detailed studies are needed to uncover the relationship between the parameters and the performance improvements of different setups.

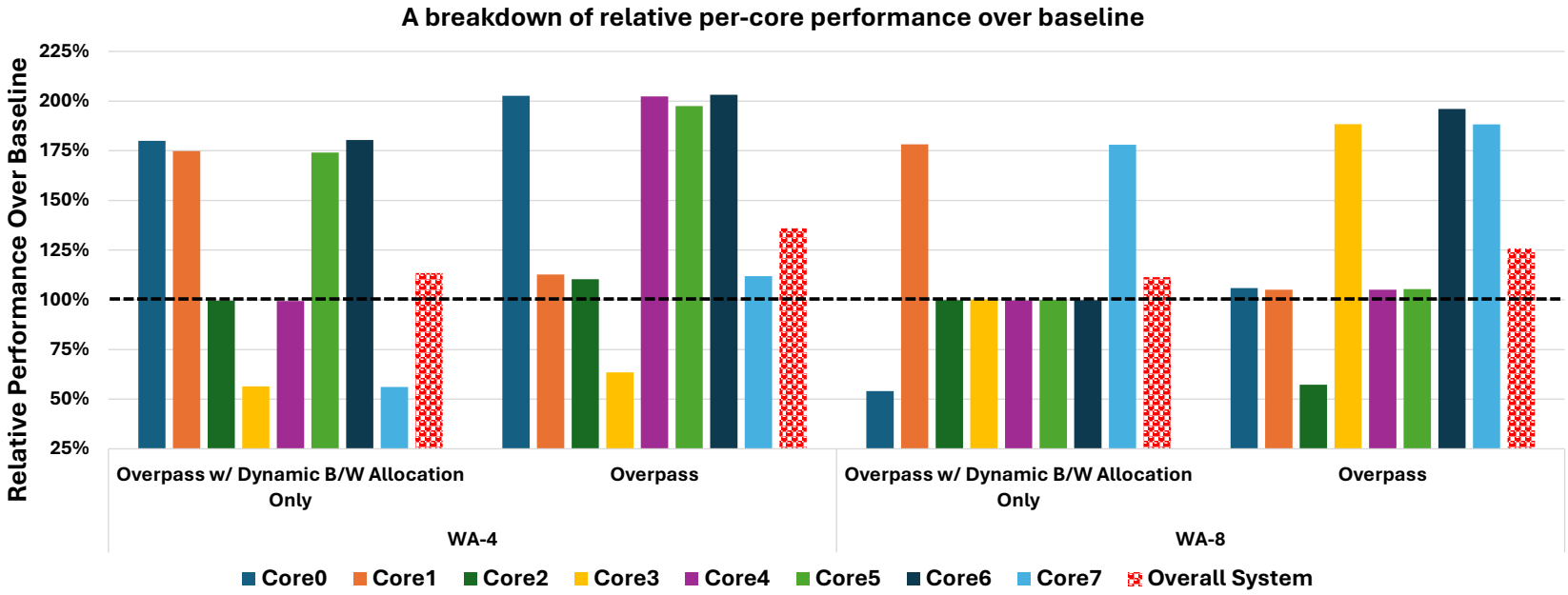


Figure 5.9: Relative per-core performance of Overpass over the baseline with WA-4 and WA-8 setups. The dotted line indicates the best baseline.

Table 5.6: Memory usage comparison of baseline and Overpass configurations. Percentage changes are shown in parentheses.

	Baseline	**Overpass W/ Only Prefetch Throttling (Change Over Baseline)**	**Overpass (Change Over Baseline)**
B/W Usage on I/O die	97%	50% (-49%)	42% (-56%)
Mem. Read Requests	102741	58909(-43%)	62027 (-40%)
Mem. Time Per Cache Access (ns)	4.338	2.739 (-37%)	2.629 (-39%)

- **Handling program phase change and avoiding local optima:** Overpass uses the past performance-bandwidth observations to guide the future bandwidth allocations. This approach can be problematic for long-running applications with shifting performance phases, as the historical data may become outdated. One option is to deploy an expiration time for each data point. However, this presents a trade-off: if the data point expires too quickly, Overpass lacks information to make good allocation decisions; if it is too late, Overpass uses stale information to make decisions that might lead to Overpass being trapped in local optima. The best expiration policy needs further study.

5.6 Chapter Summary

This chapter presents Overpass, a flexible interconnect design with distributed bandwidth allocation capabilities for heterogeneous systems. Overpass profiles the system performance dynamically and allocates target bandwidth to different upstream nodes to achieve the best system-wide performance. Overpass router enforces the bandwidth allocation with a dynamic adaptive arbiter, prioritizing traffic and leading to improved work output. Overpass can also send signals to throttle prefetching so the system can perform better. We evaluated Overpass with an eight-agent system interconnected with five Overpass routers across 12 configurations that differed in hardware performance, workload characteristics, or network bandwidth. The system performance is evaluated as the operation throughput of ResNet-level matrix multiplications. Compared to the best-performing unmoderated interconnect system, Overpass improves system performance by 35% while incurring less than 3% area overhead. Overpass exploits imbalanced bandwidth distribution to maximize system yield and throttles prefetch behaviors to prioritize on-demand memory access, improving network efficiency. The results demonstrate that a composable interconnect system employing distributed resource allocation significantly enhances performance in scalable heterogeneous systems, with minimal area and engineering overhead.

CHAPTER 6

Related Works

As heterogeneous hardware systems gain traction, a wide range of research has emerged to address the design, communication, and resource allocation challenges associated with them. This chapter surveys key related works across these domains and critically compares them against the solutions proposed in Chapters 3, 4, and 5. Specifically, we examine existing hardware design methodologies, host-accelerator communication optimizations, and shared resource management strategies, highlighting where prior efforts succeed, where they fall short, and how this dissertation's contributions—Twine, Zipper, and Overpass—push the boundary of what is achievable in streamlining heterogeneous system design.

In recent years, many hardware design languages have been proposed to improve design productivity. Table 6.1 compares Twine with other popular hardware design languages on features that help developers design heterogeneous systems.

Chisel [19] enables polymorphism in the hardware design workflow and provides the flexibility to dynamically generate hardware designs. Chisel provides an optional pre-defined interface, *e.g.*, `Valid` and `Decoupled`. However, the semantics of the pre-defined ports are not specified. Thus, automating control coordination is a challenge. It also lacks the functionality to help developers match the data format and queue the data in the presence of backpressure or out-of-order execution.

V++ [108] proposes using a compiler-generated communication channel with a multiple-writer-single-reader model as a homogeneous communication mechanism. However, such an interface is overly expensive for light modules, which leads to high and unnecessary performance, power, and area overhead.

SpinalHDL [121] provides a stream module interface, which is similar to `DecoupledIO`. However, the communication behaviors are only well-defined between two modules and thus do not support system-level elaboration. SpinalHDL does not provide queueing and data format adapting capability for the stream module interface.

BaseJump Standard Template Library (STL) [149] provides a standard hardware module library with templates but lacks system-level automation between modules.

Table 6.1: Twine compared to other popular hardware design languages on features that help with heterogeneous design.
✓= fully supported; ✗= not supported; ✓✗= partially supported.

	Chisel w/ Twine	SystemVerilog	VHDL	Chisel HDL	Bluespec Verilog	SpinalHDL	V++	PyMTL	MyHDL
Reusable Standard Interface	✓	✗	✗	✓✗	✓✗	✓✗	✓	✗	✗
Design Elaboration	✓	✓✗	✓✗	✓	✓✗	✓	✓✗	✓	✓
Component-level Semantics	✓	✗	✗	✗	✗	✓✗	✓	✗	✗
Built-in Queueing	✓	✗	✗	✗	✗	✗	✓	✗	✗
Built-in Control Coordination	✓	✗	✗	✗	✗	✓✗	✓✗	✗	✗
Built-in Data Formatting	✓	✗	✗	✗	✗	✗	✗	✗	✗
Built-in Serialization	✓	✗	✗	✗	✗	✗	✗	✗	✗
Easy Parameterization	✓	✓✗	✓✗	✓	✗	✓	✗	✓	✓

Other hardware design languages are not designed to address the challenges that developers are facing in heterogeneous design. **PyMTL [102]** and **MyHDL [39]** are two Python-based HDLs, aiming to make hardware design more accessible by providing a Python frontend. However, they offer comparable gate-level semantics to existing HDLs and lack crucial features for easy meta-programming, which is fundamental for accessible heterogeneous design. Wire sorts [37] is designed to verify the correctness of module interconnections.

In comparison, Twine provides multiple universal interfaces to meet design needs and offers system-level solutions for large-scale, heterogeneous systems, rather than partial automation that only works locally. Twine provides automation capabilities that are essential for quickly exploring the design space.

While design automation addresses the productivity challenges at the early stages of heterogeneous system development, communication latency overheads between host and accelerators become the next major bottleneck in deployment. We now shift focus to related efforts aimed at mitigating communication latency and inefficiency, the context within which Zipper was developed.

Two main research directions specifically address the latency challenge: latency-reduction techniques, which directly decrease communication costs, and latency-tolerant techniques that hide communication latency to reduce overall system overhead.

For direct latency reduction techniques, improvements come from optimized bus transistor and microarchitecture designs [103, 47, 7, 94, 158, 156, 157], protocol setups [64, 21], or data compression and approximation schemes [51, 144, 125, 18, 99, 134].

Improved bus transistor and microarchitecture designs introduce new materials, arbiters, and links into the system to reduce the point-to-point transmission delay of signals directly.

New bus protocols reduce communication delay for specific domains and use cases by reducing or specializing packet headers and communication handshake packets.

Data compression and approximation schemes minimize communication overhead by letting the system send fewer data over the data bus. These techniques are orthogonal to Zipper as Zipper does not modify the data bus. Zipper exploits opportunities within the application's semantics and characteristics and would benefit from more efficient bus designs to further improve hardware resource utilization.

As for latency-tolerant techniques, there are four major approaches: prefetching [13, 141, 27, 113, 123, 81, 109, 5, 9, 79], caching [142, 55, 123, 89, 127, 36, 139], multithreading [8, 155, 48, 143, 154, 132], and relocating [54, 86, 140, 101, 136, 70, 130].

Prefetching predicts the memory access pattern and issues memory access before the data is used. This technique does not apply to the challenge tackled in this chapter because accelerator requests often rely on host-side data-based control flow, making it hard to issue in advance.

Caching keeps data closer to the compute by exploiting spatial and temporal locality. Comparisons between Zipper and common caching techniques have been extensively discussed in §4.5.6. In summary, Zipper is more flexible and area-efficient than cache-based designs.

Multithreading hides access latency by allocating the hardware resources to another thread while waiting for the long-latency operation to complete. However, its benefits diminish when the operation is at or below the microsecond level due to context switch overhead. Moreover, multithreading relies on having enough threads to schedule and focuses on the throughput. In comparison, Zipper does not rely on switching to other work to occupy the host and significantly speeds up the end-to-end latency.

Relocating (*i.e.*, in-memory/near-memory computing) is a design philosophy that moves compute closer to the data. For instruction-level acceleration, even if placed near the memory, the system still needs to tolerate the latency between the host and the accelerator. Such a challenge is not directly addressed by relocation as a result.

Zipper is complementary to works like **UCNN**[60], which exploit data locality and access patterns of certain classes of algorithms. In comparison, Zipper is agnostic to algorithms because it calculates data reuse, removes false dependencies, and exploits opportunities for parallelism during runtime. Zipper can be used as a complementary tool to speed up solutions like UCNN further.

GPU Command Processor (GCP) [10] reads instructions from memory, schedules those instructions onto functional units, and maintains thread synchronization. Zipper's hardware does shoulder similar responsibilities as GCP, but Zipper's novelty lies in its holistic approach to exploiting parallelism across multiple devices and takes advantage of data locality.

Overall, Zipper is flexible enough to support any algorithm and deploys a more systematic approach to exploit optimization opportunities.

Although optimizing communication between hosts and accelerators can significantly boost the performance of individual applications, heterogeneous systems must also efficiently manage shared resources at the system level to avoid contention and interference. We now turn our attention to related work in bandwidth allocation and interconnect design, positioning Overpass against these solutions.

Existing interconnect management faces significant challenges in creating truly flexible and interconnected heterogeneous systems. We compared the features of different network and memory arbitration and allocation strategies in Table 6.2.

The most common arbiters used in the interconnect and memory allocators are the **static arbiters**, *i.e.*, strict priority, RR [56], LRU [82], and age-based (AGE) [1]. While simple to implement, static arbiters ignore performance variances across different agents, thus becoming insufficient for system and workload heterogeneity and gradually marginalized in real-world designs.

Multiple **dynamic adaptive arbiters** [166, 96, 65], have been introduced to provide priority and

Table 6.2: Comparison between Overpass and other existing solutions or proposed works. ✓✗= Yes, but it takes considerable engineering efforts.

	Static Arbiters	**Adaptive Arbiters**	**QoS Class**	**Software Controllers**	**Overpass**
Optimize system-level performance	✗	✓	✗	✓	✓
Fine-grained dynamic adjustment	✗	✗	✗	✓	✓
No extra software	✓	✗	✓	✗	✓
No extra profiling process	N/A	✗	✗	✓	✓
Distributed (Scalable)	N/A	✗	✓	✗	✓
Arbitrarily composable	✓	✗	✓	✓✗	✓

bandwidth control to shared resources while ensuring equality. However, these schemes only try to impose a predetermined bandwidth allocation policy without considering the impact of allocation on overall system performance, as pointed out in Ibarra-Delgado et al. [65]. Instead, the bandwidth allocation decisions are deferred to other agents in the system.

QoS based solutions [69] categorize communication and requests into different classes and serve them accordingly. QoS-based solutions rely on compute agents and sometimes applications themselves to generate QoS priority in their requests. Under this approach, compute agents are unaware of the system-level traffic and the performance status of their peers and thus cannot adjust accordingly. In a bandwidth-constrained situation, agents may increase their QoS priority to secure arbitration, leading to excessive resource contention and monopolization of network and memory bandwidth, as observed in Jeong et al. [80].

Multiple **software-based solutions** [169, 170, 115, 4] have been proposed using system-level algorithms to monitor traffic status from performance registers and throttle overused resources through the operating system's kernel layer. Like adaptive arbiters, software-based solutions rely on prior knowledge of workload characteristics or predefined performance indicators for a particular workload. Such approaches require a master node and a system-level operating system responsible for monitoring and regulating all other agents. While this might hold for SoC designs, it becomes unrealistic for CXL-connected systems where each agent is opaque to any other agents in the system. Furthermore, software solutions must be able to expect and adapt to different topologies to generate optimal allocation policies, which is a considerable hurdle for deployment.

Intel's **MBA** [40] provides developers with mechanisms to throttle application bandwidth usage. EMBA [161] leverages this capability to optimize system-wide performance. However, MBA-based techniques regulate the request rate to the last-level cache—a feature unique to Intel CPU core architectures, which inherently limits their applicability in heterogeneous systems. Additionally, EMBA's centralized control mechanism incurs a latency of at least 17 milliseconds and up to 800 milliseconds to adjust for the next scheduling interval on an eight-core system. In contrast, Overpass does not rely on any agent-architecture-specific features and can respond within tens of microseconds or even nanoseconds when required.

Several resource partitioning efforts of colocated workloads have been proposed in recent years. [30, 31] formalize the problem as contextual multi-armed bandit problems and iteratively tune system partition policies, while [32, 167] leverage machine learning models to estimate the results of actions without actually interacting with the system and reduce the search space. However, all of them remain algorithmic and lack the infrastructure and design to enforce the partition policy. All of these solutions rely solely on Intel's MBA capabilities for memory bandwidth enforcement and thus do not address the challenges in the context of heterogeneous hardware. However, methodologies proposed in these works could be integrated with Overpass as the bandwidth allocation

algorithm.

In summary, while existing works have made significant strides in design generation, latency mitigation, and shared resource management for heterogeneous systems, most require substantial engineering trade-offs, lack flexibility, or introduce prohibitive complexity. Twine, Zipper, and Overpass offer minimally invasive yet highly effective alternatives, addressing critical pain points with practical, scalable solutions. Collectively, they represent a meaningful advancement toward democratizing heterogeneous hardware design, making it more accessible, efficient, and sustainable for future systems.

CHAPTER 7

Future Directions in Heterogeneous System Design

While this dissertation addresses key challenges in the heterogeneous design process and demonstrates significant performance improvements, the landscape of heterogeneous system design is vast, with many open challenges remaining. This chapter outlines potential future research directions, beginning with a discussion of this dissertation's limitations in § 7.1. Unlike the limitations discussed in Chapters 3–5, which are specific to the respective solutions, this section considers broader limitations within the heterogeneous systems research domain.

Sections 7.2–7.4 present future opportunities from multiple perspectives: §7.2 highlights key system properties—such as security and reliability—that merit further investigation; §7.3 explores emerging design methodologies driven by Artificial Intelligence (AI), particularly Large Language Model (LLM)s; and §7.4 discusses new technologies and design targets not covered in this dissertation.

7.1 Limitations

Given the rapidly evolving landscape of computing technologies and the breadth of heterogeneous design, this dissertation has several limitations:

- **Security Concerns:** This work does not address the security implications of heterogeneous integration. As components may originate from different vendors, establishing mutual trust and ensuring overall system correctness become significantly more complex.
- **Design Space Assumptions:** The dissertation assumes a fixed set of components and interconnect topology, focusing on integration challenges. However, the wide variety of component choices and interconnect topologies inherent to heterogeneous design can yield drastically different trade-offs. This broader design space is not explored here.
- **Emerging Technologies:** Although there have been promising advances in wireless communication and advanced packaging techniques, these technologies were not mature or

widespread at the time of writing. As such, their potential impact on heterogeneous system design is not deeply explored.

- **Memory Coherence Models:** This work either enforces coherence through software or assumes non-overlapping data access. The challenge of integrating components that support different memory coherence and consistency models remains an open problem.

7.2 Multifaceted Properties of Heterogeneous Systems

7.2.1 Security

In homogeneous systems, the entire chip is often treated as a single trusted domain. As a result, on-chip data does not typically need encryption between components. In contrast, heterogeneous systems involve components from different vendors and potentially varying levels of trust, making it infeasible to assume a secure, unified trust boundary.

Techniques like Sequestered Encryption [23] and its verification framework [148] aim to minimize the trust footprint, allowing only a subset of hardware to handle sensitive data securely. However, these approaches can introduce performance overheads. Striking a balance between security guarantees and system performance will be critical for the viability of secure heterogeneous systems.

7.2.2 Reliability

Heterogeneous designs often incorporate components fabricated using different process nodes and materials, connected via advanced packaging and interconnect technologies. This diversity increases the risk of manufacturing defects and operational variability. A deeper understanding of failure modes and the development of cost-effective reliability mitigation techniques will be crucial to enabling widespread adoption.

7.3 AI-Assisted Design Methodologies

AI has started to affect our daily lives profoundly. A recent study [28] shows that individuals can improve their productivity when paired with LLMs. Although AI has not yet become pervasive in hardware design life cycles, it has the potential to disrupt and improve our approach to heterogeneous design.

7.3.1 AI as Design Tools

LLMs have shown significant potential in automating software development, but hardware design presents unique challenges. Much of it is proprietary, hardware-centric, and constrained by geometric and timing requirements—domains not well-represented in public datasets used to train current models.

A rigorous evaluation of AI tools in the context of heterogeneous hardware design could help bridge this gap. Moreover, new approaches are needed to translate between the human-readable inputs that LLMs process and the structural, geometric nature of hardware designs.

7.3.2 AI as Evaluation Methods

Fast design iteration is vital in heterogeneous systems, but accurately evaluating it through synthesis and simulation remains time-consuming. Recent works [164, 50, 165] have explored using deep learning models to directly predict performance, power, and area characteristics from design descriptions.

While promising, these techniques remain imprecise, sometimes with errors as high as 70% [165]. Future research should focus on improving the accuracy and generalizability of these prediction models to support more reliable and efficient design space exploration.

Another aspect of heterogeneous designs that is hard to evaluate is the ease of integration into the ecosystem and the programmability. Usually, it would take years of trial and error in the field to find out and improve these qualitative properties of a design. A rapid iteration of this process with the help of AI would significantly reduce the friction between development and deployment.

7.4 Emerging Design Targets and Technologies

7.4.1 Chiplet-Based Architectures

Chiplet-based architectures offer a promising alternative to monolithic designs, particularly as reticle limits constrain chip sizes. These architectures require efficient inter-chiplet communication to achieve high performance. However, communication overheads can significantly impact latency and power consumption.

Automated tools for exploring chiplet topologies, interconnect strategies, and data routing policies could help mitigate these challenges and unlock the full potential of chiplet-based systems.

7.4.2 Unified Memory Access Across Coherent and Incoherent Agents

As heterogeneous systems increasingly integrate CPUs and accelerators on a shared memory substrate, new challenges arise in maintaining consistent and efficient memory access. CPUs typically rely on strong, hardware-enforced coherence, while accelerators may use software-managed or relaxed coherence models to optimize performance.

Designing memory systems that can support both models efficiently——and ensuring correctness and performance under this hybrid paradigm——remains an important area for future work.

CHAPTER 8

Conclusion

This dissertation addresses significant challenges associated with designing high-performance heterogeneous hardware systems. The central contribution lies in streamlining the complexities of heterogeneous system design, focusing specifically on automating the generation of modular hardware, reducing communication overhead, and optimizing resource allocation within heterogeneous architectures. This dissertation compiles three complementary efforts that help developers overcome the challenges at various stages of developing a heterogeneous architecture from scratch, as illustrated in Figure 8.1. Specifically, each of the three efforts addresses one of the burning problems in designing heterogeneous hardware: the tremendous complexity of generating heterogeneous designs when reconfiguring and reusing modules to construct IPs, high communication overhead between components when connecting IPs to form dies, and ineffecient sharing of resources between compute dies on a single chip. Collectively, these efforts enhance the efficiency, feasibility, and flexibility of heterogeneous systems, marking substantial progress in both the theoretical and practical aspects of computer architecture research.

The first part of this dissertation introduces Twine, a language extension designed to automate and simplify modular hardware design, thereby significantly reducing the complexity associated with manually integrating heterogeneous components. Twine standardizes control interfaces, thereby increasing the reusability of hardware modules, accelerating iterative design, and improving productivity. The evaluation results clearly indicated Twine's efficacy in facilitating rapid design generation, reducing code complexity, and improving design productivity. These outcomes offer system designers a robust methodology to explore design spaces efficiently, reducing engineering overhead while improving hardware design quality.

In the second part, we identified and addressed the critical problem of high communication overhead between host processors and accelerators——a bottleneck significantly impeding heterogeneous system performance. To tackle this, Zipper was developed, incorporating novel latency-tolerant bus optimization techniques that exploit the locality and parallelism inherent in many applications. Zipper's approach allows heterogeneous systems to maintain high performance without extensive redesign or complex compiler interventions. Evaluation through comprehensive case

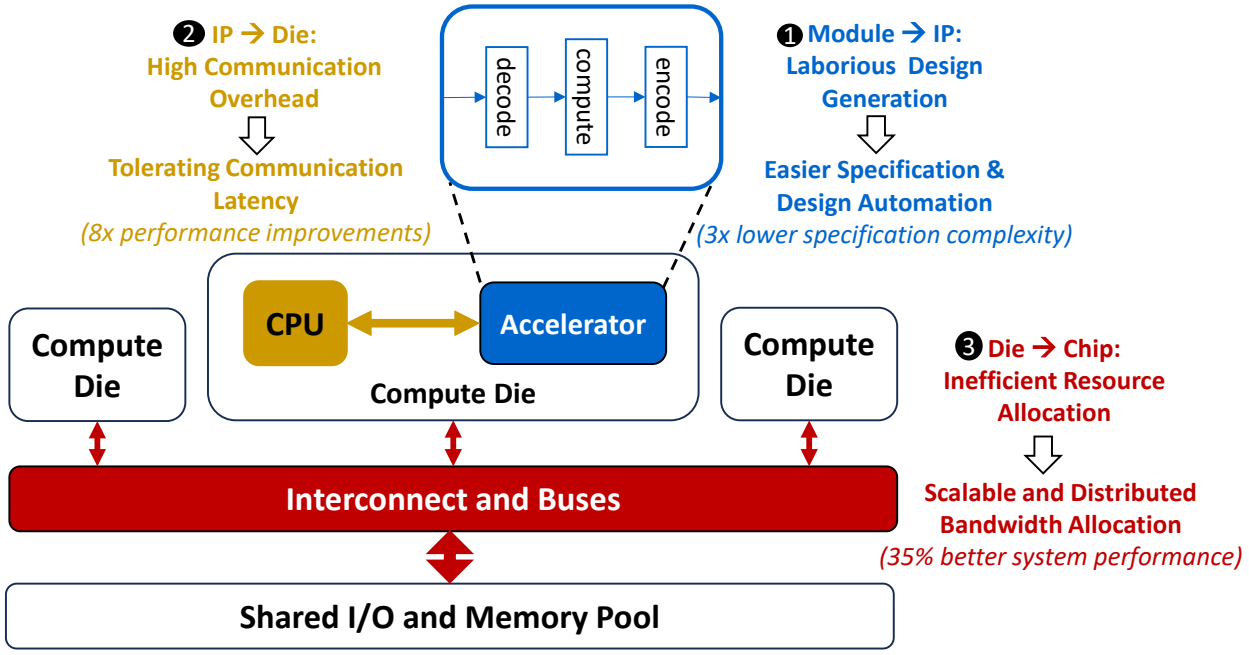


Figure 8.1: A high-level summary of the contributions of this dissertation matched with the corresponding challenges addressed by the solutions proposed in this dissertation.

studies on real FPGA-based systems demonstrated up to 8x performance improvements, with less than 5% hardware overhead. These results validate Zipper's potential to profoundly enhance system efficiency, enabling previously infeasible system configurations and reinforcing the practicality of heterogeneous system deployments.

The third major contribution of this dissertation, Overpass, targets inefficiencies in communication resource allocation within heterogeneous systems. Overpass introduces a flexible interconnect architecture featuring dynamic bandwidth management and distributed resource allocation strategies, effectively minimizing resource contention and interference among heterogeneous components. Empirical evaluations indicated that Overpass significantly enhances overall system throughput, achieving up to 35% performance improvement, costing less than 0.3% area overhead. This approach provides system designers with a powerful mechanism to optimize interconnect resources adaptively, improving both system responsiveness and scalability.

The limitations of this dissertation are summarized at the end of this dissertation. Built on the findings and the limitations, this dissertation discusses future directions for optimizing and implementing high-performance heterogeneous designs to bridge the gap between ideas and realities. The principle that unites these ideas is effectively sharing design and performance information across the heterogeneous components during design time and runtime. Such constructive discussions extend the value of this dissertation and benefit the computer architecture community.

Across the three distinct yet complementary contributions presented in this dissertation, and

through the exploration of future research directions, a unifying theme emerges: the pursuit of making hardware design more accessible, scalable, and flexible. By reducing the friction that traditionally burdens the creation of complex systems, the solutions proposed here——Twine, Zipper, and Overpass——empower developers to fully exploit the potential of heterogeneous architectures. Yet the reach of these methodologies is not confined to heterogeneity alone. The foundational principles of modular composition, latency tolerance, and distributed resource management address systemic challenges inherent to modern hardware design. As systems continue to scale in complexity, even within homogeneous environments, these approaches will become indispensable tools for sustaining innovation. Thus, the ideas set forth in this dissertation not only advance the state of the art in heterogeneous system design but also lay critical groundwork for a new era of flexible, high-performance, and architecturally diverse computing.

BIBLIOGRAPHY

[1] Dennis Abts and Deborah Weisser. Age-based Packet Arbitration in Large-radix K-ary N-cubes. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–11, 2007.

[2] Accellera. SystemC.

[3] Michale Adler. Intel CCI: Core Cache Interface. Sept 2017.

[4] Homa Aghilinasab, Waqar Ali, Heechul Yun, and Rodolfo Pellizzoni. Dynamic Memory Bandwidth Allocation for Real-Time GPU-Based SoC Platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3348–3360, 2020.

[5] Sam Ainsworth and Timothy M Jones. Software prefetching for indirect memory accesses. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 305–317. IEEE, 2017.

[6] Jasmin Ajanovic. Pci express 3.0 overview. In *Hot Chips Symposium*, pages 1–61, 2009.

[7] M Nishat Akhtar and Othman Sidek. An intelligent adaptive arbiter for maximum cpu utilization, fair bandwidth allocation and low latency. *IETE Journal of Research*, 59(1):48–54, 2013.

[8] Haitham Akkary and Michael A Driscoll. A dynamic multithreading processor. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–236. IEEE, 1998.

[9] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857, 2020.

[10] AMD. Amd gpu hardware basics. 2010.

[11] AMD. AMD Infinity Architecture: The Foundation of the Modern Datacenter. Aug 2019.

[12] AMD Inc. together we advance data centers, 2022.

[13] DW Anderson, FJ Sparacio, and Robert M Tomasulo. The ibm system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967.

[14] Filippo Testa Andrea Galimberti and Alberto Zeni. NoCRouter - RTL Router Design in SystemVerilog, 2017.

[15] Brandon Hill Andrew E. Freedman. Apple's A18 and A18 Pro processors powers the iPhone 16 and 16 Pro — and Apple Intelligence. *tom's HARDWARE*, 2024.

[16] ARM. big.LITTLE Technology: The Future of Mobile. 2013.

[17] ARM. AMBA AXI and ACE Protocol Specification. Version H.c. Jan 2021.

[18] Todd M Austin and Gurindar S Sohi. Zero-cycle loads: Microarchitecture support for reducing load latency. In *Proceedings of the 28th annual International Symposium on Microarchitecture*, pages 82–92. IEEE, 1995.

[19] 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 *DAC Design Automation Conference 2012*, pages 1212–1221, 2012.

[20] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, mar 2017.

[21] Edith Beigné, Fabien Clermidy, Pascal Vivet, Alain Clouard, and Marc Renaudin. An asynchronous noc architecture providing low latency service and its multi-level design framework. In *11th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 54–63. IEEE, 2005.

[22] L. Benini and G. De Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, 2002.

[23] Lauren Biernacki, Meron Zerihun Demissie, Kidus Birkayehu Workneh, Fitsum Assamnew Andargie, and Todd Austin. Sequestered encryption: A hardware technique for comprehensive data privacy. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 73–84, 2022.

[24] Lauren Biernacki, Meron Zerihun Demissie, Kidus Birkayehu Workneh, Galane Basha Namomsa, Plato Gebremedhin, Fitsum Assamnew Andargie, Brandon Reagen, and Todd Austin. Vip-bench: A benchmark suite for evaluating privacy-enhanced computation frameworks. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 139–149. IEEE, 2021.

[25] J Randall Brown. Bounded Knapsack Sharing. *Mathematical Programming*, 67:343–382, 1994.

[26] Neil Burgess, Jelena Milanovic, Nigel Stephens, Konstantinos Monachopoulos, and David Mansell. Bfloat16 Processing for Neural Networks. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 88–91, 2019.

[27] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. *ACM SIGARCH Computer Architecture News*, 19(2):40–52, 1991.

[28] Alexia Cambon, Brent Hecht, Ben Edelman, Donald Ngwe, Sonia Jaffe, Amy Heger, Mihaela Vorvoreanu, Sida Peng, Jake Hofman, Alex Farach, et al. Early LLM-Based Tools for Enterprise Information Workers Likely Provide Meaningful Boosts to Productivity. *Microsoft.*, 2023.

[29] Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. A Holistic Memory Contention Analysis for Parallel Real-Time Tasks under Partitioned Scheduling. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 239–252, 2020.

[30] Ruobing Chen, Wangqi Peng, Yusen Li, Xiaoguang Liu, and Gang Wang. Orchid: An Online Learning Based Resource Partitioning Framework for Job Colocation With Multiple Objectives. *IEEE Transactions on Computers*, 72(12):3443–3457, 2023.

[31] Ruobing Chen, Haosen Shi, Yusen Li, Xiaoguang Liu, and Gang Wang. OLPart: Online Learning based Resource Partitioning for Colocating Multiple Latency-Critical Jobs on Commodity Computers. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 347–364, New York, NY, USA, 2023. Association for Computing Machinery.

[32] Ruobing Chen, Jinping Wu, Haosen Shi, Yusen Li, Xiaoguang Liu, and Gang Wang. DRLPart: A Deep Reinforcement Learning Framework for Optimally Efficient and Robust Resource Partitioning on Commodity Servers. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '21, page 175–188, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Shibo Chen, Yonathan Fisseha, Jean-Baptiste Jeannin, and Todd Austin. Twine: a chisel extension for component-level heterogeneous design. In *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe*, DATE '22, page 466–471, Leuven, BEL, 2022. European Design and Automation Association.

[34] Shibo Chen, Hailun Zhang, and Todd Austin. Zipper: Latency-Tolerant Optimizations for High-Performance Buses. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, ASPDAC '25, page 567–574, New York, NY, USA, 2025. Association for Computing Machinery.

[35] Steven W. D. Chien, Ivy B. Peng, and Stefano Markidis. Posit npb: Assessing the precision improvement in hpc scientific applications. In Roman Wyrzykowski, Ewa Deelman, Jack Dongarra, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics*, pages 301–310, Cham, 2020. Springer International Publishing.

[36] Jongsok Choi, Kevin Nam, Andrew Canis, Jason Anderson, Stephen Brown, and Tomasz Czajkowski. Impact of cache architecture and interface on performance and area of fpga-based processor/parallel-accelerator systems. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 17–24. IEEE, 2012.

[37] Michael Christensen, Timothy Sherwood, Jonathan Balkind, and Ben Hardekopf. Wire sorts: A language abstraction for safe hardware composition. In *Proceedings of the 42nd*

ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021, page 175–189, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Brad Cohen, Mahesh Subramony, and Mike Clark. Next Generation "Zen 5" Core. In *2024 IEEE Hot Chips 36 Symposium (HCS)*, pages 1–27. IEEE, 2024.

[39] MyHDL Community. MyHDL.

[40] David Cornu, Marcel, Andrew J Herdrich, and Khawar Munir Abbasi. Introduction to Memory Bandwidth Allocation, Mar 2019.

[41] Ian Cutress. Intel shows xeon scalable gold 6138p with integrated fpga, shipping to vendors. 2018.

[42] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Elsevier, 2004.

[43] Dakshina Dasari, Vincent Nelis, and Benny Akesson. A Framework for Memory Contention Analysis in Multi-Core Platforms. *Real-Time Syst.*, 52(3):272–322, May 2016.

[44] Florent de Dinechin, Luc Forget, Jean-Michel Muller, and Yohann Uguen. Posits: The good, the bad and the ugly. In *Proceedings of the Conference for Next Generation Arithmetic 2019*, CoNGA'19, New York, NY, USA, 2019. Association for Computing Machinery.

[45] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[46] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[47] Masoumeh Ebrahimi, Masoud Daneshtalab, Pasi Liljeberg, Juha Plosila, and Hannu Tenhunen. Cluster-based topologies for 3d networks-on-chip using advanced inter-layer bus architecture. *Journal of Computer and System Sciences*, 79(4):475–491, 2013. JCSS CADS 2010.

[48] Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, Rebecca L Stamm, and Dean M Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE MICRO*, 17(5):12–19, 1997.

[49] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Bandwidth Bandit: Quantitative Characterization of Memory Contention. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 457–458, 2012.

[50] Wenji Fang, Yao Lu, Shang Liu, Qijun Zhang, Ceyu Xu, Lisa Wu Wills, Hongce Zhang, and Zhiyao Xie. MasterRTL: A Pre-Synthesis PPA Estimation Framework for Any RTL Design. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023.

[51] Matthew Farrens and Arvin Park. Dynamic base register caching: A technique for reducing address bus width. *ACM SIGARCH Computer Architecture News*, 19(3):128–137, 1991.

[52] Yinxiao Feng and Kaisheng Ma. Chiplet actuary: a quantitative cost model and multi-chiplet architecture exploration. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, page 121–126, New York, NY, USA, 2022. Association for Computing Machinery.

[53] Charlie Giattino, Edouard Mathieu, Veronika Samborska, and Max Roser. Data Page: Computation Used to Train Notable Artificial Intelligence Systems, by domain. https://ourworldindata.org/grapher/artificial-intelligence-training-computation, 2023. Part of the publication: *Artificial Intelligence*. Data adapted from Epoch. Online resource from Our World in Data.

[54] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The terasys massively parallel pim array. *Computer*, 28(4):23–31, 1995.

[55] James R Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th annual international symposium on Computer architecture*, pages 124–131, 1983.

[56] Ellen Louise Hahne. *Round Robin Scheduling for Fair Flow Control in Data Communication Networks*. PhD thesis, Massachusetts Institute of Technology, 1986.

[57] Peter Hallam. What Do Programmers Really Do Anyway. *Microsoft Developer Network (MSDN)—C# Compiler*, 2006.

[58] Mohamed Hassan and Rodolfo Pellizzoni. Analysis of Memory-Contention in Heterogeneous Cots MPSOCs. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, pages 23–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[59] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[60] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W. Fletcher. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 674–687. IEEE Press, 2018.

[61] John L. Hennessy and David A. Patterson. A New Golden Age for Computer Architecture. *Commun. ACM*, 62(2):48–60, January 2019.

[62] C.N. Hinds. An enhanced floating point coprocessor for embedded signal processing and graphics applications. In *Conference Record of the Thirty-Third Asilomar Conference on Signals, Systems, and Computers (Cat. No.CH37020)*, volume 1, pages 147–151 vol.1, 1999.

[63] Joe Hindy. Musk's xAI Launches Grok 3: Here's What You Need to Know. Accessed: March 21, 2025.

[64] Compute Express Link. Compute Express Link, 2025.

[65] Salvador Ibarra-Delgado, Remberto Sandoval-Arechiga, José Ricardo Gómez-Rodríguez, Manuel Ortíz-López, and María Brox. A Bandwidth Control Arbitration for SoC Interconnections Performing Applications with Task Dependencies. *Micromachines*, 11(12), 2020.

[66] Salvador Ibarra-Delgado, Remberto Sandoval-Arechiga, José Ricardo Gómez-Rodríguez, Manuel Ortíz-López, and María Brox. A bandwidth control arbitration for soc interconnections performing applications with task dependencies. *Micromachines*, 11(12), 2020.

[67] IEEE. VHDL IEEE 1076-2019. 2011.

[68] IEEE. SystemVerilog Standard. 2019.

[69] E.800 : Definitions of Terms Related to Quality of Service, August 1994.

[70] Daniele Ielmini and H-S Philip Wong. In-memory computing with resistive switching devices. *Nature electronics*, 1(6):333–343, 2018.

[71] Qualcomm inc. Snapdragon 8 Gen 2 Mobile Platform.

[72] Intel. Intel quickpath interconnect: Maximizing multicore processor performance.

[73] Intel. Intel® Xeon® Processor Scalable Family Technical Overview. Oct 2019.

[74] Intel. Intel Acceleration Stack for Intel® Xeon® CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual. Nov 2019.

[75] Intel. Intel Emulation and Prototyping. *Intel*, 2023.

[76] Intel. Unveiling Intel's 2024 Xeon Architecture, 2024.

[77] Intel Inc. Agile 7 M-Series FPGA Network-on-Chip (NoC) User Guide - Quality of Service (QoS) Support, 2025.

[78] Vikram Jain, Sebastian Giraldo, Jaro De Roose, Linyan Mei, Bert Boons, and Marian Verhelst. TinyVers: A Tiny Versatile System-on-Chip With State-Retentive eMRAM for ML Inference at the Extreme Edge. *IEEE Journal of Solid-State Circuits*, 58(8):2360–2371, August 2023.

[79] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 747–764, 2022.

[80] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 850–855, New York, NY, USA, 2012. Association for Computing Machinery.

[81] Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. Orchestrated scheduling and prefetching for gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 332–343, 2013.

[82] Theodore Johnson, Dennis Shasha, et al. 2Q: a Low Overhead High Performance Bus Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450. Citeseer, 1994.

[83] ES Jung. Creating the future with silicon. *Advanced Materials Technologies*, 8(20):2200867, 2023.

[84] William Kahan. Ieee standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.

[85] Michael Kanellos. Moore's Law to Roll On for Another Decade, 2023.

[86] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S Lee, et al. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 790–803. IEEE, 2020.

[87] Donggyu Kim. RISCV-Mini. 2021.

[88] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[89] PAH Knoben. Software caching for tree-based algorithms on accelerator cards. Master's thesis, University of Twente, 2021.

[90] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[91] Yinan Kong and Md Selim Hossain. Fpga implementation of modular multiplier in residue number system. In *2018 IEEE International Conference on Internet of Things and Intelligence System (IOTAIS)*, pages 137–140, 2018.

[92] R. Kumar, V. Zyuban, and D.M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 408–419, 2005.

[93] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen. Are We There Yet? A Study on the State of High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):898–911, 2019.

[94] Kook-Pyo Lee, Yang-Hee Joung, and Seong-Jun Kang. Bus arbitration considering waiting cycle. *Journal of the Korea Institute of Information and Communication Engineering*, 18(11):2703–2708, 2014.

[95] Steven Leibson. Intel's Chiplet Strategy Accelerates FPGA Development. *Forbes*, 2023.

[96] Haishan Li, Ming Zhang, Wei Zheng, and Dongxiao Li. An Adaptive Arbitration Algorithm for SoC Bus. In *2007 International Conference on Networking, Architecture, and Storage (NAS 2007)*, pages 245–246, 2007.

[97] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020.

[98] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[99] Jiangjiang Liu, Krishnan Sundaresan, and Nihar R Mahapatra. Fast, performance-optimized partial match address compression for low-latency on-chip address buses. In *2006 International Conference on Computer Design*, pages 17–24. IEEE, 2006.

[100] Yanqiang Liu, Jiacheng Ma, Zhengjun Zhang, Linsheng Li, Zhengwei Qi, and Haibing Guan. Megatron: Software-managed device tlb for shared-memory fpga virtualization. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1213–1218, 2021.

[101] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. Concurrent data structures for near-memory computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–245, 2017.

[102] Derek Lockhart, Gary Zibrat, and Christopher Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 280–292. IEEE, 2014.

[103] Ruibing Lu, Aiqun Cao, and Cheng-Kok Koh. Samba-bus: A high performance bus architecture for system-on-chips. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(1):69–79, 2007.

[104] Enno Luebbers, Song Liu, and Michael Chu. Simplify Software Integration for FPGA Accelerators with OPAE.

[105] Jan Macheta, Agnieszka Dabrowska-Boruch, Pawel Russek, and Kazimierz Wiatr. Arpalib: A big number arithmetic library for hardware and software implementations. a case study for the miller-rabin primality test. In Stephan Wong, Antonio Carlos Beck, Koen Bertels,

and Luigi Carro, editors, *Applied Reconfigurable Computing*, pages 323–330, Cham, 2017. Springer International Publishing.

[106] Saurav Maji, Utsav Banerjee, Samuel H. Fuller, Mohamed R. Abdelhamid, Phillip M. Nadeau, Rabia Tugce Yazicigil, and Anantha P. Chandrakasan. A Low-Power Dual-Factor Authentication Unit for Secure Implantable Devices. In *2020 IEEE Custom Integrated Circuits Conference (CICC)*, page 1–4. IEEE, March 2020.

[107] Terrence Mak, Peter Y. K. Cheung, Kai-Pui Lam, and Wayne Luk. Adaptive routing in network-on-chips using a dynamic-programming network. *IEEE Transactions on Industrial Electronics*, 58(8):3701–3716, 2011.

[108] Patrick C McGeer, Szu-Tsung Cheng, Michael J Meyer, and Patrick Scaglia. Hardware Design Language for the Design of Integrated Circuits, July 16 2002. US Patent 6,421,808.

[109] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)*, 49(2):1–35, 2016.

[110] https://github.com/mmperf/mmperfSingle CPU Core Matrix Multiplication Benchmarks, 2022.

[111] Gordon E. Moore. Cramming More Components onto Integrated Circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[112] Gordon E Moore et al. Cramming More Components onto Integrated Circuits, 1965.

[113] Todd C Mowry, Monica S Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. *ACM Sigplan Notices*, 27(9):62–73, 1992.

[114] Pascal Nasahl, Robert Schilling, Mario Werner, and Stefan Mangard. Hector-v: A heterogeneous cpu architecture for a secure risc-v execution environment. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 187–199, New York, NY, USA, 2021. Association for Computing Machinery.

[115] Vincent Nollet, Théodore Marescaux, Diederik Verkest, Jean-Yves Mignolet, and Serge Vernalde. Operating-system Controlled Network on Chip. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, page 256–259, New York, NY, USA, 2004. Association for Computing Machinery.

[116] Nvidia. NVIDIA GH200 Grace Hopper Superchip.

[117] Nvidia. NVIDIA Blackwell Architecture.

[118] Andreas Olofsson. Silicon Compilers-Version 2.0. *keynote, Proc. ISPD*, 2018.

[119] OpenTitan.

[120] John F Palmer. The intel® 8087 numeric data processor. In *Proceedings of the May 19-22, 1980, national computer conference*, pages 887–893, 1980.

[121] Charles Papon. SpinalHDL.

[122] D.A. Patterson. RAMP: Research Accelerator for Multiple Processors - a Community Vision for a Shared Experimental Parallel HW/SW Platform. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 1–, 2006.

[123] R Hugo Patterson, Garth A Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 79–95, 1995.

[124] Vasilis F. Pavlidis, Ioannis Savidis, and Eby G. Friedman. Chapter 2 - manufacturing of three-dimensional packaged systems. In Vasilis F. Pavlidis, Ioannis Savidis, and Eby G. Friedman, editors, *Three-Dimensional Integrated Circuit Design (Second Edition)*, pages 15–35. Morgan Kaufmann, Boston, second edition edition, 2017.

[125] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 377–388, 2012.

[126] Peter Walker. Cross Bar Systems.

[127] Christian Pinto, Yiannis Gkoufas, Andrea Reale, Seetharami Seelam, and Steven Eliuk. Hoard: A distributed data caching system to accelerate deep learning training on the cloud. *arXiv preprint arXiv:1812.00669*, 2018.

[128] Artur Podobas and Satoshi Matsuoka. Hardware implementation of posits and their application in fpgas. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 138–145, 2018.

[129] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 519–530, New York, NY, USA, 2013. Association for Computing Machinery.

[130] Carlos Ríos, Nathan Youngblood, Zengguang Cheng, Manuel Le Gallo, Wolfram HP Pernice, C David Wright, Abu Sebastian, and Harish Bhaskaran. In-memory computing on a photonic platform. *Science Advances*, 5(2):eaau5759, 2019.

[131] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42, mar 2011.

[132] Amir Roth and Gurindar S Sohi. Speculative data-driven multithreading. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 37–48. IEEE, 2001.

[133] C. Rowen, M. Johnson, and P. Ries. The mips r3010 floating-point coprocessor. *IEEE Micro*, 8(3):53–62, 1988.

[134] Eyal Rozenberg and Peter Boncz. Faster across the pcie bus: a gpu library for lightweight decompression: including support for patched compression schemes. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, pages 1–5, 2017.

[135] Karl Rupp. Microprocessor Trend Data.

[136] Fabian Schuiki, Michael Schaffner, Frank K Gürkaynak, and Luca Benini. A scalable near-memory architecture for training deep neural networks on large in-memory datasets. *IEEE Transactions on Computers*, 68(4):484–497, 2018.

[137] Scotten Jones. TSMC 2nm Process Disclosure – How Does it Measure Up? *TechInsights*, 2025.

[138] Nimish Shah, Paragkumar Chaudhari, and Kuruvilla Varghese. Runtime programmable and memory bandwidth optimized fpga-based coprocessor for deep convolutional neural network. *IEEE Transactions on Neural Networks and Learning Systems*, 29(12):5922–5934, 2018.

[139] Yakun Sophia Shao, Sam Xi, Viji Srinivasan, Gu-Yeon Wei, and David Brooks. Toward cache-friendly hardware accelerators. In *HPCA Sensors and Cloud Architectures Workshop (SCAW)*, pages 1–6, 2015.

[140] Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, Roel Jordans, Henk Corporaal, and Albert-Jan Boonstra. A review of near-memory computing architectures: Opportunities and challenges. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 608–617. IEEE, 2018.

[141] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, 1978.

[142] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[143] Lawrence Spracklen and Santosh G Abraham. Chip multithreading: Opportunities and challenges. In *11th International Symposium on High-Performance Computer Architecture*, pages 248–252. IEEE, 2005.

[144] Jacob R Stevens, Ashish Ranjan, and Anand Raghunathan. Axba: An approximate bus architecture framework. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. ACM, 2018.

[145] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 639–650, 2013.

[146] Karthik Swaminathan and Augusto Vega. Hardware specialization: From cell to heterogeneous microprocessors everywhere. *IEEE Micro*, 41(6):112–120, 2021.

[147] K. Takeda, F. Ishino, Y. Ito, and T. Nakashima. A Single-Bhip 80-Bit Floating Point Processor. *IEEE Journal of Solid-State Circuits*, 20(5):986–992, 1985.

[148] Qinhan Tan, Yonathan Fisseha, Shibo Chen, Lauren Biernacki, Jean-Baptiste Jeannin, Sharad Malik, and Todd Austin. Security Verification of Low-Trust Architectures. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 945–959, New York, NY, USA, 2023. Association for Computing Machinery.

[149] Michael Bedford Taylor. Invited: Basejump stl: Systemverilog needs a standard template library for hardware design. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[150] Mellanox Technologies. Roce vs. iwarp competitive analysis. 2017.

[151] Microchip Technology. DIVAS(Divide and Square Root Accelerator) accelerate computations. *Microchip Technology*, 2020.

[152] Tenstorrent. Tensix Neo.

[153] Walkthrough: Matrix Multiplication, 2023.

[154] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[155] Theo Ungerer, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Computing Surveys (CSUR)*, 35(1):29–63, 2003.

[156] Zhehui Wang, Zhifei Wang, Jiang Xu, Yi-Shing Chang, Jun Feng, Xuanqi Chen, Shixi Chen, and Jiaxu Zhang. Camon: Low-cost silicon photonic chiplet for manycore processors. *IEEE transactions on computer-aided design of integrated circuits and systems*, 39(9):1820–1833, 2019.

[157] Sebastian Werner, Pouya Fotouhi, Roberto Proietti, and S. J. Ben Yoo. Awgr-based optical processor-to-memory communication for low-latency, low-energy vault accesses. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '18, page 269–278, New York, NY, USA, 2018. Association for Computing Machinery.

[158] Sebastian Werner, Javier Navaridas, and Mikel Luján. Designing low-pwer, low-latency networks-on-chip by optimally combining electrical and optical links. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 265–276. IEEE, 2017.

[159] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The architecture and design of a database processing unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[160] Ning Wu, Tao Jiang, Lei Zhang, Fang Zhou, and Fen Ge. A reconfigurable convolutional neural network-accelerated coprocessor based on risc-v instruction set. *Electronics*, 9(6), 2020.

[161] Yaocheng Xiang, Chencheng Ye, Xiaolin Wang, Yingwei Luo, and Zhenlin Wang. EMBA: Efficient Memory Bandwidth Allocation to Improve Performance on Intel Commodity Processor. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, New York, NY, USA, 2019. Association for Computing Machinery.

[162] Xilinx. Vivado HLS.

[163] Xilinx. Xilinx Runtime Library (XRT).

[164] Ceyu Xu, Chris Kjellqvist, and Lisa Wu Wills. SNS's Not A Synthesizer: A Deep-Learning-Based Synthesis Predictor. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 847–859, New York, NY, USA, 2022. Association for Computing Machinery.

[165] Ceyu Xu, Pragya Sharma, Tianshu Wang, and Lisa Wu Wills. Fast, Robust and Transferable Prediction for Hardware Logic Synthesis. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 167–179, New York, NY, USA, 2023. Association for Computing Machinery.

[166] booktitle=2006 8th International Conference on Solid-State and Integrated Circuit Technology Proceedings Xu, Yi and Li, Li and Gao, Ming-lun and Zhang, Bing and Jiang, Zhao-yu and Du, Gao-ming and Zhang, Wei. An Adaptive Dynamic Arbiter for Multi-Processor SoC. pages 1993–1996, 2006.

[167] Tiannuo Yang, Ruobing Chen, Yusen Li, Xiaoguang Liu, and Gang Wang. CoTuner: A Hierarchical Learning Framework for Coordinately Optimizing Resource Partitioning and Parameter Tuning. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 317–326, New York, NY, USA, 2023. Association for Computing Machinery.

[168] Greg Yeric. Moore's law at 50: Are we planning for retirement? In *2015 IEEE International Electron Devices Meeting (IEDM)*, pages 1.1.1–1.1.8, 2015.

[169] Heechul Yun, Waqar Ali, Santosh Gondi, and Siddhartha Biswas. Bwlock: A dynamic memory access control framework for soft real-time applications on multicore platforms. *IEEE Transactions on Computers*, 66(7):1247–1252, 2017.

[170] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-Core

Platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

[171] Mark Zhao, Mingyu Gao, and Christos Kozyrakis. Shef: Shielded enclaves for cloud fpgas. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1070–1085, 2022.