LANE-BASED HARDWARE SPECIALIZATION FOR LOOP- AND FORK-JOIN-CENTRIC PARALLELIZATION AND SCHEDULING STRATEGIES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Shreesha Srinath

August 2018

LANE-BASED HARDWARE SPECIALIZATION FOR LOOP- AND FORK-JOIN-CENTRIC PARALLELIZATION AND SCHEDULING STRATEGIES

Shreesha Srinath, Ph.D.

Cornell University 2018

Serious physical design issues are breaking down traditional abstractions in computer architecture. For the past 40 years, Moore's Law and Dennard's Scaling have provided the smaller, cheaper, faster, and more power-efficient transistors that fueled innovation in computer architecture. In the mid 2000s, Dennard's Scaling broke down, and this in turn stagnated the growth in processor clock frequencies and reduced the power efficiency of transistors. More recently, there has been empirical evidence suggesting Moore's Law of transistor cost-scaling has slowed down. While transistors continue to shrink at a slower pace, technology scaling is no longer ensuring cheaper, faster, and more power-efficient transistors. In this disruptive regime, architects have a critical role in improving performance while mitigating design costs. The challenges posed by the impending end of Moore's Law and the non-existent benefits of Dennard's Scaling motivate reconsidering the traditional boundaries between hardware and software. Architects have responded by embracing parallelization and specialization across the layers of the computing stack. A key research challenge involves creating clean hardware/software abstractions that are highly programmable, yet still enable efficient execution on both traditional and specialized microarchitectures.

In this thesis, I present a lane-based hardware specialization approach to building programmable accelerators for loop- and fork-join-centric parallel programs. To mitigate the design costs and increase the applicability of hardware specialization, I make the case for lane-based behavior-specific accelerators. I propose two lane-based behavior-specific accelerators: XLOOPS and SSAs. *Explicit loop specialization (XLOOPS)* is an approach that is based on the idea of elegantly encoding inter-iteration dependence patterns in the instruction set. The XLOOPS binaries can execute on (1) traditional microarchitectures with minimal performance impact, (2) specialized microarchitectures to improve performance and/or energy efficiency, and (3) adaptive microarchitectures that can seamlessly migrate loops between traditional and specialized execution to trade-off performance vs. energy efficiency. *Smart sharing architectures (SSAs)* are a new approach to building

lane-based accelerators that can efficiently execute recursive fork-join-centric parallel programs. SSAs designs share expensive hardware resources to reduce the area costs and employ complexity-effective *smart sharing mechanisms* that exploit instruction redundancy to mitigate the loss in performance while maximizing efficiency.

BIOGRAPHICAL SKETCH

Shreesha Srinath was born to Rekha Srinath and A.K. Srinath in Bangalore, India on September 13th, 1986. Shreesha attended the Sri Aurobindo Memorial School in Bangalore from 1990–2002. During high school, in addition to his academic studies, Shreesha was interested in *Carnatic* music, and was trained in singing, playing the *Mridangam*, and playing the flute. Shreesha went on to pursue mathematics and science in the 11th and 12th grade (senior secondary school) at Sri Kumaran Children's Home in Bangalore from 2002–2004.

Determined to obtain an undergraduate degree in Electronics and Communication Engineering, Shreesha enrolled at the Visvesvaraya National Institute of Technology at Nagpur, India. Shreesha took fundamental courses in electronics, analog/digital communication systems, and signal processing, and also served as the President of the IEEE student chapter at Nagpur. Shreesha completed his B.Tech degree in 2008, and was motivated to pursue a master of science degree with a focus on wireless communication systems.

Shreesha was admitted to the MS program in the Department of Electrical and Computer Engineering at University of Wisconsin-Madison and moved to the United States in 2008. At UW-Madison, Shreesha was fortunate to have worked with Prof. Suman Banarjee on a research project that focused on the design of approximate wireless communication systems. While there he became interested in the field of reconfigurable computing and FPGAs. He was advised by Prof. Katherine (Compton) Morrow and pursued his thesis topic on automatic generation of high-performance multipliers for FPGAs. Upon graduating in 2010, Shreesha went on to work at Intel Corporation in Folsom. At Intel, he was part of the visual computing group and built functional simulators for the texture sampler units used in Intel chips. It was at Intel that Shreesha realized his interest in building prototypes of computing systems beyond just simulators.

Shreesha decided to pursue a Ph.D. degree and was offered a graduate fellowship at Cornell University in 2011. At Cornell University, Shreesha began his graduate studies under the tutelage of Prof. Christopher Batten. By working in Prof. Batten's group, he gained invaluable experience in topics related to energy-efficient computer architecture, programmability and design of specialized hardware, parallel programming models, compilers, hardware design methodology, and ASIC design.

This document is dedicated to all my teachers, to all my *gurus*.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support, encouragement, and advice from many people. I am grateful to my family, friends, and mentors who have helped me along the way both personally and professionally.

First and foremost, I would like to thank my advisor Christopher Batten who has been a true mentor, a passionate teacher, and a role model throughout my time at Cornell University. I thank Chris for believing in me and for supporting me throughout various projects I pursued in my graduate school career. Chris's passion for a vertically integrated research methodology has got me hooked and I hope to continue to build elegant computer systems in future. I would also like to thank the rest of my thesis committee, Zhiru Zhang, and Rajit Manohar. I have truly enjoyed working with Zhiru on the XLOOPS and the PolyHS projects. Thanks, Zhiru for teaching me everything I know about high-level synthesis. Thanks, Rajit for your advice and your feedback.

Thanks to all the members of Batten Research Group for working with me and for contributing to the various pieces of the research infrastructure that made this thesis possible. Derek Lockhart has been a great resource for thoughtful discussion and support. I would like to thank Derek for building the PyMTL and Pydgin frameworks, which enabled both the XLOOPS and the SSAs projects. Ji Kim is a dear friend right from my early days of graduate school. Thanks, Ji for leading the FG-SIMT and the LTA projects, it was great to work with you. Special thanks to Berkin Ilebyi, for working with me on the XLOOPS project, for being roommates at every conference I have ever attended, and for being a true friend. I have truly enjoyed my conversations with Christopher Torng. Chris, I appreciate our walks and our discussions on research methodologies, goals, and life in general. Thanks, Moyang Wang for teaching me work-stealing runtimes. Khalid Al-Hawaj, I appreciate your passion for hot sauce and will miss late-night/early-morning conversations. Shunning Jiang is a talented researcher and has been a great addition to BRG. Tuan Ta and Lin Cheng pushed me to be a better mentor. Tuan and Lin, I hope I have convinced you on RISC-V, and I look forward to your success in graduate school.

I would like to thank the members of the Zhang Research Group. Gai Liu, Steve Dai, Ritchie Zhao, and Mingxing Tan, thanks for making me feel like an extended member of ZRG. Thanks Gai and Steve, for making me coffee whenever I needed it. Nitish Srivastava is a dear friend and I am looking forward to his work on the new and improved XLOOPS.

I would like to thank all my friends at the Computer Systems Laboratory. CSL introduced me to a wonderful group of people to bounce ideas of, share knowledge, get lunch with, and explore Ithaca. Thanks to Saugata Ghose, Rob Karmazin, Jon Tse, Carlos Ortega, Ben Hill, Andrew Ferraiuolo, Daniel Lo, Yao Wang, KK Yu, Abhinandan Majumdar, Xiaodong Wang, Skand Hurkat, Mark Buckler, Helena Caminal, and Sachille Atapattu.

I owe many fellow graduate students for making the whole experience much more enjoyable. I appreciate the counsel, support, and friendship I have enjoyed from Ajay Bhat, Shivam Ghosh, Sachin Nadig, Divya Sharma, Namrata Singh, Priyanka Jagtap, and Adarsh Kowdle. Special thanks to Bret Leraul, Sanjay Dharmavaram, Janet Hendrickson, and Keiji Kunigami for helping me remain sane during the last couple years of graduate school.

To my friends from Madison, who made me feel at home when I first got to the United States: Vaishali Karanth, Diwakar Kedlaya, Chaitanya Baone, and Nidhishri Tapadia. You guys have been my family in the US.

Most importantly, I thank my mother, whose unrelenting support and encouragement provides me constant motivation and inspiration. Maa, thanks for all your hard work and unbounded love. I am especially grateful for all the values you have impressed on me and hope to be as good of a human being that you are. Thanks to my father for standing by me, always. Dad, thanks for all your sacrifices and continual support that has helped me achieve my goals. My parents have always been exceptional role models and have taught me the importance of putting others before one's self. Thanks to my younger brother, Srijith Srinath, for his love and support, and for being with my parents, when I could not. Srijith, you have cheered me up when I have needed it right from when you have been a baby to a responsible adult. I wish you good luck for your graduate school career.

This thesis was supported in part by an NSF CAREER Award #1149464, NSF XPS Award #1337240, NSF CRI Award #1512937, NSF SHF Award #1527065, DARPA Young Faculty Award #N66001-12-1-4239, AFOSR YIP Award #FA9550-15-1-0194, and equipment, tool, and/or IP donations from Intel, NVIDIA, Xilinx, Synopsys, and ARM.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

XLOOPS	explicit loop specialization
SSA	smart sharing architecture
CISC	complex instruction set computer
RISC	reduced instruction set computer
SIMD	single-instruction multiple-data
NRE	non-recurring engineering
DSL	domain-specific language
CMP	chip multiprocessor
CGRA	coarse-grained reconfigurable array
BSA	behavior-specific accelerator
DSA	domain-specific accelerator
ASA	application-specific accelerator
SIMT	single-instruction multiple-thread
GPGPU	general-purpose graphics processing unit
LUP	logical parallel unit
PE	processing element
DAG	directed-acyclic graph
TBB	(Intel) threading building blocks
GPP	general-purpose processor
LPSU	loop pattern specialization unit
AMO	atomic memory operation
MIV	mutual induction variable
CIR	cross-iteration register
LMU	lane management unit
IDQ	index queue
LLFU	long-latency functional unit
MIVT	mutual induction variable table
CIB	cross-iteration buffer
LSQ	load-store queue
APT	adaptive profiling table
LUT	look-up table
RTL	register-transfer level
VLSI	very-large-scale integration
WSRT	work-stealing runtime
SMPD	single-program multiple data
RR	round-robin
MPC	minimum-pc

CHAPTER 1
INTRODUCTION

Serious physical design issues are breaking down traditional abstractions in computer architecture. For the past 40 years, Moore's Law and Dennard's Scaling have provided the smaller, cheaper, faster, and more power-efficient transistors that fueled innovation in computer architecture. In the mid 2000s, Dennard's Scaling broke down, and this in turn stagnated the growth in processor clock frequencies and reduced the power efficiency of transistors. More recently, there has been empirical evidence suggesting Moore's Law of transistor cost-scaling has slowed down. While transistors continue to shrink at a slower pace, technology scaling is no longer ensuring cheaper, faster, and more power-efficient transistors. In this disruptive regime, architects have a critical role in improving performance while mitigating design costs. The challenges posed by the impending end of Moore's Law and the non-existent benefits of Dennard's Scaling motivate reconsidering the traditional boundaries between hardware and software. Architects have responded by embracing parallelization and specialization across the layers of the computing stack. A key research challenge involves creating clean hardware/software abstractions that are highly programmable, yet still enable efficient execution on both traditional and specialized microarchitectures.

In this thesis, I present a lane-based hardware specialization approach to building programmable accelerators for loop- and fork-join-centric parallel programs. A central theme in my thesis is to make lightweight changes to compilers, runtimes, instruction sets, and microarchitectures to improve performance, area, and energy efficiency for these loop- and fork-join-centric parallel programs. The accelerators presented in this thesis extend the capabilities of prior programmable lane-based accelerators to efficiently execute challenging loops with complex inter-iteration dependences and recursive task-parallel programs.

1.1 Trends in Technology Scaling

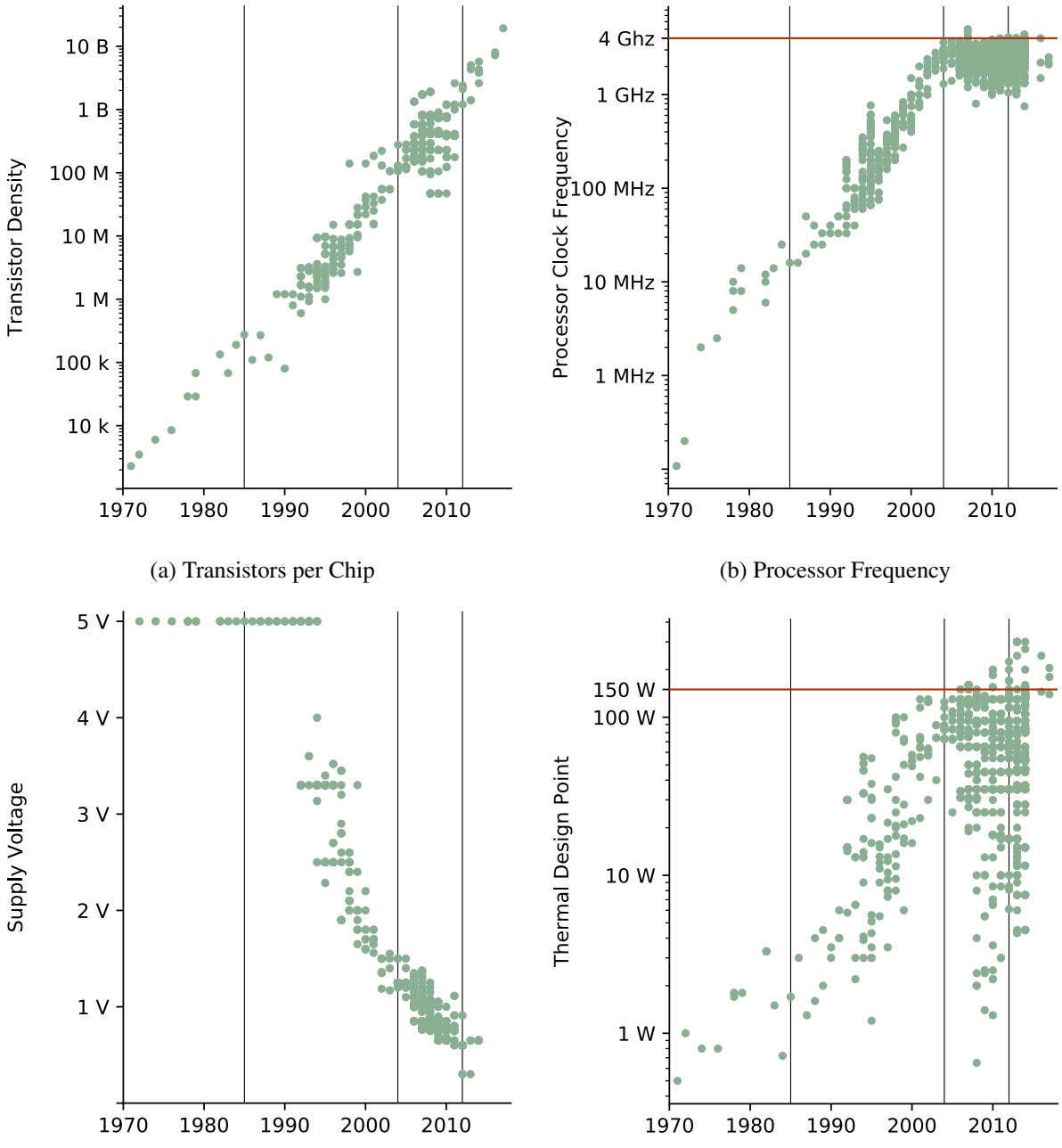
Smaller, cheaper, faster, and more power-efficient transistors have played a central role in improving the performance of computer systems. Moore's Law and Dennard's Scaling have been the driving forces for innovations in the semiconductor industry. In this section, I review these two laws and the landscape of technology scaling between 1971 to present times.

Moore's Law – In 1965, five years after the introduction of integrated circuits, Gordon Moore made an observation that the number of transistors on a single chip had doubled every year [Moo65]. Moore predicted that by 1975 the economics may dictate squeezing as many as 65,000 components on a single chip. In 1975, Moore revised his prediction to state that the number of transistors that could be economically integrated on a single chip doubles *every two years*. This simple observation set the pace for advancements in the semiconductor industry for years to come.

While the popular interpretation of the 1965 paper has focused on doubling the number of transistors, more fundamentally, Moore's Law is also about transistor cost scaling. In his original paper Moore states, *"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year."* Moore observed that there is a minimum cost for a given technology that depends on the transistor feature size and the wafer size. Smaller feature sizes increase transistor densities which amortizes the packaging costs. Larger wafer sizes provide more chips which amortizes the fabrication costs. Further, larger wafer sizes are preferred as the defects are empirically known to likely occur at the edges of a wafer. The semiconductor industry has thus focused on scaling the feature sizes and the wafer sizes to provide cheaper transistors. Moore's Law of technology scaling is the fundamental driving force that has resulted in "smaller" and "cheaper" transistors.

Dennard's Scaling – In 1974, Dennard et al. made an equally important observation that scaling voltage in proportion to the transistor dimensions results in a constant power density [DGY$^+$74]. Robert Dennard and his colleagues quantified the scaling rules of integrated circuits using a unitless scaling constant k. Dennard's Scaling states that, if the transistor dimensions are reduced by $\frac{1}{k}$ and the supply voltage is lowered by the same factor $\frac{1}{k}$, then the following results are applicable:

- the total number of the transistors increase by k^2;
- the transistor voltage V and transistor current I scale by a factor $\frac{1}{k}$;
- the transistor resistance remains constant due to $\frac{V}{I}$;
- the gate capacitance reduces by a factor $\frac{1}{k}$ as the area is decreased by $\frac{1}{k^2}$ but is cancelled by the decrease in electrode spacing by k;
- the delay time improves by a factor of k given by $\frac{VI}{C}$;
- and the power density remains constant given by $\frac{VI}{A}$.



(a) Transistors per Chip

(b) Processor Frequency

(c) Supply Voltage (Vdd_low)

(d) Typical Power

Figure 1.1: Trends in Technology Scaling from 1971 to 2017– The plots show transistor counts per chip, the operating voltage, processor frequency, and typical power for chips. The markers on the y-axis indicate the four eras of technology scaling: (i) 1971–1985: Early Era (smaller, cheaper, faster); (ii) 1985–2004: Golden Era (smaller, cheaper, faster, power-efficient); (iii) 2004–2012: Slowdown Era (smaller, cheaper); (iv) 2012–2017: Retirement Era (smaller). Plots based on the data from CPUDB [DKM+12] and additional data collected by Karl Rupp [Rup18]

The decrease in delay time means that a chip can switch faster in a new technology node while maintaining the same power density. These rules, popularly known as Dennard's Scaling, provided the guidelines for scaling that resulted in "faster" and more "power-efficient" transistors.

Figure 1.1 shows the trends for the number of transistors per chip, processor clock frequency, supply voltage, and typical power limits from 1971 to 2017. The timeline for semiconductor technology scaling can be divided into four eras based on the characteristics of transistors in each era.

Early Era (1971–1985) – In the early era, technology scaling provided for *smaller, cheaper, and faster* transistors. NMOS technology was dominant in this era as it could pack transistors more densely and cheaply compared to CMOS technology. Smaller transistors reduced the gate delay resulting in faster transistors. NMOS transistors used reverse-bias to hold the transistors in the "off" state, and this in turn constrained the voltage scaling. With increasing transistors counts, the increase in leakage current and static power consumption motivated the switch to CMOS technology. In the early era, Moore's Law resulted in increased transistor counts but the voltage remained constant at 5V.

Golden Era (1985–2004) – Around the mid-1980s, despite the density arguments, the semiconductor industry transited to CMOS technology. In the early part of the golden era voltages continued to remain at 5V, a standard that remained since the bipolar ICs. The issue of power dissipation ultimately forced the abandonment of the 5V power supply. Around the mid-1990s, with improvements in manufacturing technology, Dennard's Scaling rules were applied to improve the power efficiency of transistors. In this golden era, scaling provided *smaller, cheaper, faster, and more power-efficient* transistors. Compared to the early era, the number of transistors increased by 450×, the processor frequency increased by 195×, and the supply voltages scaled from 5V to close to 1V.

Slowdown Era (2004–2012) – Around the mid-2000s, Dennard's Scaling broke down. Lowering of supply voltages in CMOS technology was ultimately challenged by the increased leakage currents at smaller transistor dimensions. Denser chips resulted in large leakage currents that further increased the power consumption, and the threat of thermal runaway set new limits on power consumption. Processor clock frequency leveled off as the costs for cooling large chips increased due to increased dynamic power at faster frequencies. In the slowdown era, transistor scaling resulted in *smaller and cheaper* transistors that were no longer faster and power efficient.

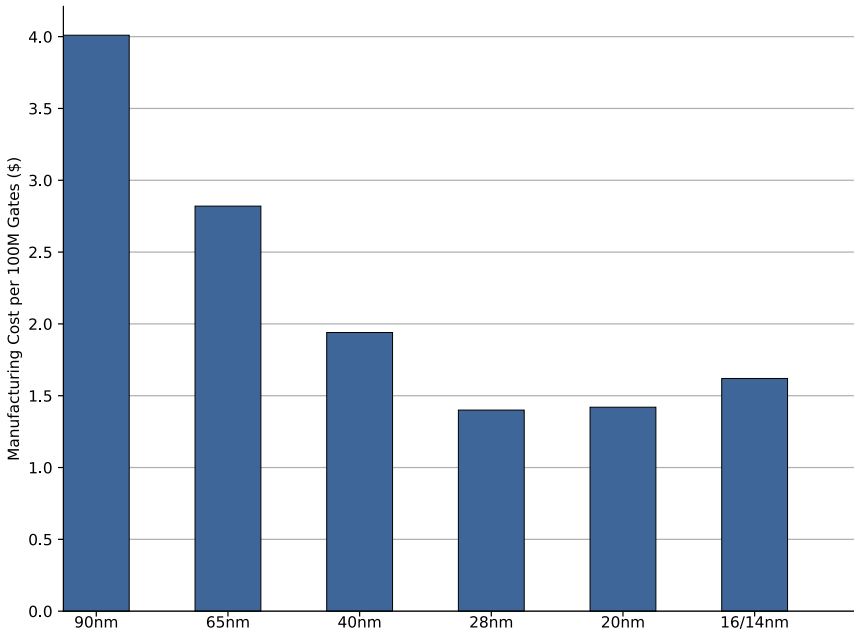
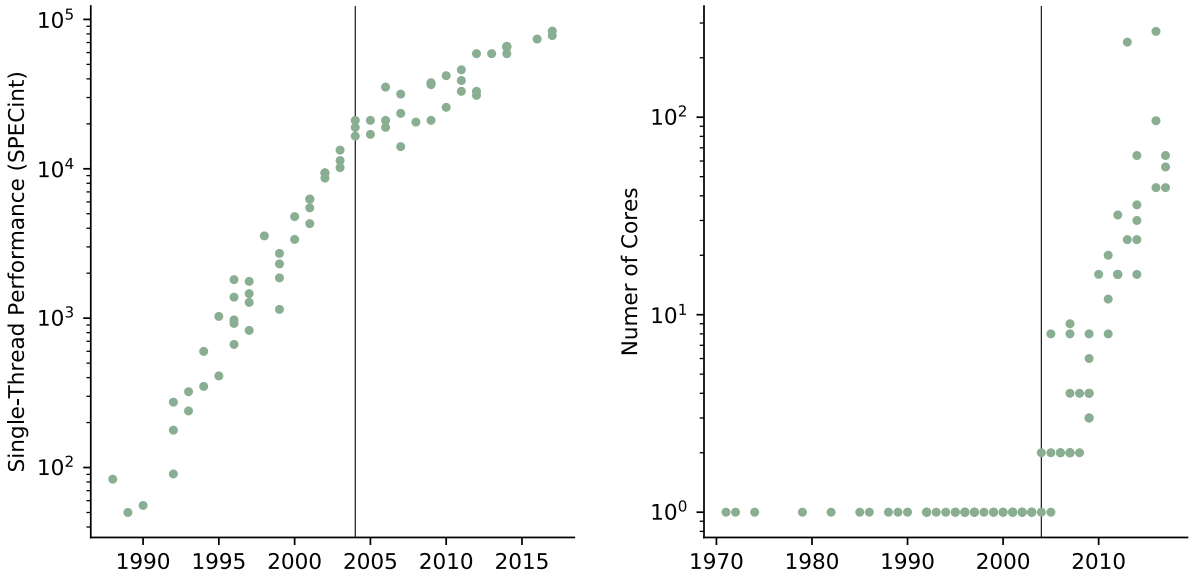


Figure 1.2: Costs per 100 Million Gates – Trend for cost reductions per 100M transistors with technology scaling based on data collected and presented in Table 1 of [Jon14]. The costs are based on the number of transistors, gate utilization, yield of transistors, and wafer costs.

Retirement Era (2012–present) – Until 2012, Moore's Law of transistor cost-scaling continued to provide smaller and cheaper transistors. Figure 1.2 shows the cost per million transistors. The figure shows that the trend of cheaper transistors stopped around 2012 when the 22/20nm technology node when into production. Fabless semiconductor companies such as NVIDIA, AMD, Qualcomm, and Broadcomm have reported that the fabrication costs are no longer declining as in previous decades [Fla17]. Additionally, the doubling of transistors *every two years* has mostly ended. Intel has acknowledged the abandonment of the famous two year "tick-tock" model with a transition to a three-year architecture and technology cadence. However, transistor scaling is projected to continue with announcements of the 5nm technology node targeted for 2021. Recent advances in technology such as FinFETs, gate all-around transistors, multi-patterning, and EUV lithography are short-term boosters for Moore's Law. In this era, scaling transistor dimensions continues albeit at a slower pace with an uncertain future. The transistors are getting smaller but are no longer cheaper. The end of Dennard's Scaling and slowing down of Moore's Law has ushered us to the retirement era where technology scaling might ultimately stop due to the diminishing returns.



(a) SPECInt Performance

(b) Number of Cores

Figure 1.3: Trends in Single-Thread SPECint Performance and the Number of Cores per Die – SPECint performance increased exponentially from 1988 until around 2004. The transition to multicore with increasing core counts has been a direct consequence of the breakdown of Dennard's Scaling. The performance continues to improve slowly with aggressive auto-vectorization and auto-parallelization. Plots based on data from CPUDB [DKM+12] and additional data collected by Karl Rupp [Rup18].

1.2 Trends in Computer Architecture

The field of computer architecture has played an instrumental role in transforming the advancements in technology scaling into innovations in embedded microcontrollers, personal computers, smart phones, warehouse-scale computers, and supercomputers. In this section, I briefly review the trends in computer architecture during each era given the technology constraints discussed in the previous section.

The introduction of integrated circuits in the early era led to the advent of early microprocessors. In 1971, Intel invented the first 4-bit microprocessor (Intel 4004) with just 2,300 transistors. With increasing transistor budgets, the 4004 was quickly followed by the 8-bit microprocessor (Intel 8008) in 1972. Architects in this era designed embedded microcontrollers that were widely used in a range of applications. Key contributions in this era were the design of early CISC instruction sets, assembly programming, and microprocessors. The density of storage devices in this era was low and accessing memory was slow. These two reasons motivated the use of CISC encoding as complex instruction sets reduced the program sizes and the required memory footprint.

During the early 1980s, CISC instruction sets were slowly being replaced by RISC instruction sets. Improvements in SRAM technologies resulted in faster caches. Advancements in compiler techniques enabled the transition to higher-level languages which replaced assembly programming. With the advancement in smaller and faster transistors architects focused on faster processor pipelines that made use of fast caches and relied on compilers to generate binaries that targeted efficient 32-bit RISC instruction sets. The abundance of transistors continued, and in the mid-1990s architects focused on exploiting *instruction-level parallelism* to continue to improve performance. Techniques such as out-of-order execution that employed advanced branch predictors, hardware memory disambiguation, large instruction windows, and large reorder buffers became popular. Instruction widths increased from 32-bits to 64-bits and several 64-bit processors were used in both consumer and embedded applications. This era also witnessed advancement in techniques for using wider ALUs to exploit *data-level parallelism* in the form of packed-SIMD units and vector units. Figure 1.3(a) shows the improvements in single-threaded performance based on the SPECint benchmark suite. The figure shows how advances in computer architecture and technology resulted in a 250× improvement in single-threaded performance during the Golden Era of technology scaling.

With the breakdown of Dennard's Scaling, computer architecture witnessed a radical shift to an era of multicore/manycore computing [ABC$^+$06]. Figure 1.3(b) shows the rise in the number of cores around 2004. Limits on power consumption and the slowdown of transistor frequency motivated new techniques that focused on exploiting *thread-level parallelism*. *Parallelization* of applications was the key to achieving improvements in performance. Figure 1.3(a) shows that the performance of single-threaded processors continued albeit at a slower pace post 2004. Auto-vectorization and auto-parallelizing compilers explain some of the performance improvements shown in the figure.

Scaling of multicore/manycore processors is ultimately limited by power constraints [EBA$^+$11]. The fast rates of transistor switching generates heat that cannot be dissipated at a rate equal to the switching. The technology constraints impose a utilization wall that results in a limited fraction of the chip that can be active leading to increasing amounts of *dark silicon*. Michael Taylor has crystallized four different approaches to address the challenges of dark silicon: shrink, dim, specialize, and technology advancements [Tay13]. Of these approaches, the *specialization* approach, has received much interest as specialization reduces the overheads of general-purpose instruction

set processing thereby improving energy efficiency. In the retirement era of technology scaling, architects can no longer rely on technology scaling to improve the performance and have to carefully budget the available transistors given the rising costs. Future computing systems will likely employ both parallelization and specialization to continue to improve performance.

1.3 Trends in Hardware Specialization

Specialized hardware performs a set of tasks with higher performance and/or better energy efficiency compared to a general-purpose processor. Specialization improves performance by exploiting the control/data-flow characteristics and the available parallelism in applications. Hardware specialization can vary from lightweight modifications to a general-purpose processor to radical new circuits tailored to an application. Fundamentally, the more specialized a unit is for a given application, the more energy efficient that unit will be when executing the application. Metrics such as performance, energy, and area efficiency, as well as generality and programmability, can be applied to evaluate a specialization technique.

One way to classify specialized hardware is along a flexibility versus specialization axis. A hardware solution is said to be *specialized* if it is more efficient for an application or a class or applications while sacrificing the broad applicability of the solution. A hardware solution is said to be *flexible* if it is broadly applicable for a variety of applications. The design of specialized hardware must navigate the tension between less efficient flexible architectures and more efficient specialized architectures. The non-recurring engineering (NRE) costs associated with the design and verification of specialized hardware must be justified by the generality of such hardware. An over-specialized solution renders the solution to be inapplicable to closely related computations (e.g., varying the arithmetic precision across solutions). Additionally, hardware specialization changes the traditional software and hardware abstractions. Hardware specialization often necessitates innovations in the software stack that span domain-specific languages (DSLs), compilers, runtimes, and instruction sets. The ease of programmability lowers the barrier and the costs for deploying a given specialized solution.

Figure 1.4 shows how several classes of hardware can be mapped along a flexibility vs. specialization axis. The various classes of specialized hardware moving from left to right are as follows: (i) chip multiprocessors (CMPs) are the most flexible solutions and can be composed of homo-

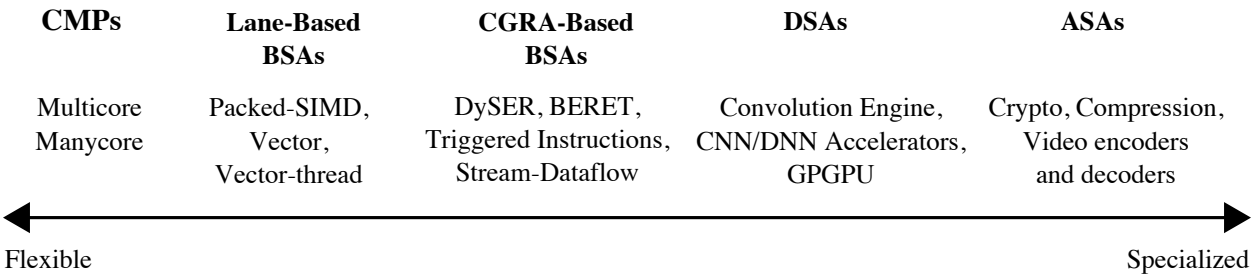


Figure 1.4: Classes of Hardware Specialization – Classes of hardware specialization arranged based on flexibility vs. specialization axis. CMPs = chip multiprocessors; BSAs = behavior-specific accelerators; DSAs = domain-specific accelerators; ASAs = application-specific accelerators.

geneous or heterogeneous processors; (ii) lane-based behavior-specific accelerators (lane-based BSAs) use instruction set programmable *lanes* to exploit parallel program behaviors; (iii) CGRA-based behavior-specific accelerators (CGRA-based BSAs) use a sea of programmable units to exploit parallel program behaviors; (iv) domain-specific accelerators (DSAs) are specialized hardware customized for a specific domain; (v) application-specific accelerators (ASAs) are circuits specialized to a given application. The classes of specialization can be further combined for a given computing platform.

CMPs – With the lack of Dennard's Scaling, CMPs have emerged as a common solution across the mobile, desktop, and server markets. The exact size and type of the processors that compose a CMP solution can vary from simple in-order processors to complex superscalar processors. Examples for homogeneous CMPs include Cavium network processors [YBC+06], TILE64-[BEA+08], and the Knights Corner platform [Bol12]. Examples of heterogeneous CMPs include ARM's big.LITTLE processing platforms [Kre11], Qualcomm's Snapdragon 810 [Gwe14a], and the Samsung Exynos platform [Gwe14b]. CMP solutions are the most flexible platforms and can execute a wide variety of applications. However, CMPs do little to directly address the problem of dark silicon as they do not reduce the overheads of general-purpose instruction set processing.

BSAs – Behavior-specific accelerators are a class of hardware specialized solutions that focus on exploiting specific parallel program behaviors. A parallel program behavior is defined by a given parallelization and scheduling strategy. Chapter 2 describes parallelization and scheduling

strategies in more detail. The goal of BSAs is to specialize hardware while balancing the sacrifices in flexibility over a broad class of applications. Typically BSAs integrate with a host general-purpose processor at either the L1 or the L2 cache boundary. I identify two classes of BSAs: lane-based BSAs and CGRA-based BSAs.

Lane-Based BSAs are composed of instruction-set-programmable *lanes*. Loosely speaking *lanes* are execution units that amortize area overheads by sharing control logic and resources for instruction and data supply. Examples of popular lane-based BSAs include packed-SIMD units [Hug15,SS00], traditional vector units [EVS98,EV96,Oya99], and vector-thread units [KBH+04, LAB+11, Lee16]. Compared to CMP solutions, lane-based BSAs provide high compute density and are efficient for certain behaviors. For example, applications with regular control-flow and data parallel loops are known to perform well on packed-SIMD and vector units. However, packed-SIMD and vector units struggle on applications containing challenging loops with inter-iteration dependences and recursive fork-join programs. Traditional lane-based BSAs, such as packed-SIMD and vector units rely on high-quality auto-vectorizing compilers.

CGRA-Based BSAs are composed of a sea of configurable processing elements that are interconnected using specialized networks and storage units. Examples of CGRA-based BSAs include Triggered Instructions [PPA+13], DySER [GHN+12], BERET [GFA+11], and Stream-dataflow [NGAS17]. CGRA-based BSAs offer high computational density for a given amount of silicon area by eliminating the circuitry for general purpose instruction set processing. Flexible interconnection networks enable mapping applications that have dependences that are otherwise ill-suited for traditional lane-based BSAs. However, CGRA-based BSAs require radical modifications to traditional software/hardware abstractions and often require high-quality compilation flows. Further, irregular applications pose a challenge to the utilization of the processing elements in a CGRA.

DSAs – Domain-specific accelerators improve energy efficiency while remaining flexible for a specific domain of applications. Examples of DSAs include general-purpose graphic processing units (GPGPUs), the convolution engine [QHS+13], the Q100 database accelerator [WLP+14], and machine-learning accelerators [CDS+14, JYP+17]. GPGPUs are examples of laned-based DSAs that are specialized for the graphics domain but are instruction set programmable compared to many other DSAs. DSAs often use functional units, memory storage elements, and interconnec-

tion networks that are specialized for the characteristics of typical applications or algorithms within a domain. However, DSAs are prone to obsoletion as the algorithms for a domain can evolve and often require the programmer to carefully map an application to the underlying DSA. DSLs and specialized compilation flows have evolved to improve the programmer productivity and increase the accessibility of DSAs.

ASAs Application-specific accelerators are orders of magnitude more efficient than general-purpose processors. ASAs use highly specialized circuits to execute a given algorithm. Examples of ASAs include video encoding/decoding accelerators [HQW+10], cryptographic accelerators, and the Sonic3D accelerator for ultrasound beam formation [SYW+13]. Compared to other classes of hardware specialization, ASAs have the best performance, area efficiency, and energy efficiency for a given application. ASAs achieve these metrics by completely giving up programmability. Given the high costs of developing ASAs the applicability of ASAs is often limited to very few cases.

Unfortunately, there is no silver bullet, and one can expect modern computing platforms to be increasingly heterogeneous with the inclusion of both parallel processors as well as accelerators along the specialization axis. Heterogeneous platforms that include both parallel processors as well as specialized hardware burden the programmer and challenge the traditional boundaries of software and hardware abstractions. With the slowdown of Moore's law, transistors are no longer cheaper and the die area is not entirely free. Programmable lane-based BSAs provide an attractive starting point as they are applicable to a broad range of applications. Embedding the principles of DSAs and CGRA-based BSAs within the template of lane-based BSAs is a promising direction. I make a case for lane-based BSAs in Chapter 2. The inclusion of the Tensor Cores in NVIDIA's Volta architecture is an example of this evolution [nvi17]. However, extending the capabilities of lane-based BSAs to handle loop-centric parallel programs with challenging inter-iteration dependences and fork-join-centric parallel programs with dynamic task parallelism remains to be addressed. As a step towards this vision, my thesis focuses on improving the broad applicability of lane-based BSAs for challenging loop- and fork-join-centric parallel programs.

1.4 Thesis Overview

This thesis presents a lane-based hardware specialization approach for loop- and fork-join centric parallelization and scheduling strategies. Chapter 2 analyzes the process of mapping applications to the underlying hardware by identifying the space of *parallelization* and *scheduling* strategies. The chapter discusses the vision for *behavior-specific accelerators* and qualitatively compares lane-based and CGRA-based BSAs. Given the need to carefully budget transistors used for specialization and the benefits of maintaining a programmable abstraction, I present a case for lane-based BSAs that execute loop- and fork-join-centric parallel programs. Chapter 3 presents a novel approach called *explicit loop specialization (XLOOPS)* that is based on the idea of elegantly encoding inter-iteration loop dependences. The XLOOPS hardware/software abstraction requires only lightweight changes to a standard compiler to generate XLOOPS binaries and enables executing these binaries on: (1) traditional microarchitectures with minimal performance impact, (2) specialized microarchitectures to improve performance and/or energy efficiency, and (3) adaptive microarchitectures that can seamlessly migrate loops between traditional and specialized execution to trade-off performance vs. energy efficiency. Chapter 4 proposes *smart sharing architectures (SSAs)*, a new approach to building lane-based BSAs which can efficiently support fork-join-centric parallel programs. SSAs employ complexity-effective *smart sharing mechanisms* to exploit *instruction redundancy* in fork-join-centric parallel programs. SSAs include a rich design space for *conjoined lanes* (lanes that share hardware resources) that are arranged as a continuum of design points between sharing no resources and sharing all resources. Chapter 5 summarizes the contributions of this thesis and discusses promising directions for future work.

The primary contributions of this thesis are:

- I make the case for single-ISA heterogeneous platforms that transparently integrate traditional general-purpose processors and lane-based BSAs to improve the performance and energy efficiency of loop- and fork-join-centric parallel programs.

- I propose an elegant new XLOOPS hardware/software abstraction that explicitly encodes inter-iteration loop dependence patterns that execute on traditional, specialized, and adaptive microarchitectures; I also propose a novel XLOOPS microarchitecture that augments a

general-purpose processor with a lane pattern specialization unit to execute the XLOOPS binaries.

- I propose smart sharing architectures, a new approach that employs complexity-effective smart sharing mechanisms to exploit instruction redundancy in fork-join-centric parallel programs to save area while maximizing efficiency and minimizing performance losses.

1.5 Collaboration, Previous Publications, and Funding

The work done in this thesis was possible thanks to contributions both small and large by many colleagues at Cornell University. My advisor Christopher Batten was integral in all aspects of both the XLOOPS and the SSA projects.

I was the lead architect for the XLOOPS project. I defined the XLOOPS instruction set extensions and was resposible for bringing up the XLOOPS LLVM compiler framework. I also ported application kernels, developed the XLOOPS gem5-PyMTL cosimulation framework, implemented the RTL models for a simple LPSU, and implemented the XLOOPS FPGA prototype as presented in Chapter 3. Berkin Ilbeyi played a key role in defining the XLOOPS instruction set and ported many application kernels. Berkin took the lead in developing the XLOOPS PyMTL cycle-level models. Mingxing Tan contributed application kernels and helped with the XLOOPS LLVM compiler. In particular, Mingxing improved the XLOOPS loop-strength-reduction pass and authored the memory alias analysis pass and the dynamic loop-bound checking pass. Gai Liu contributed by helping with the XLOOPS gem5-McPAT energy models. Pol Rosello and Paul Jackson contributed by porting a kernel each from the PBBS benchmark suite. Christopher Torng helped in bringing up the gem5 framework. Aadeetya Shreedhar helped in bringing up the initial version of traditional execution on gem5. Derek Lockhart developed the PyMTL modeling framework used in the cycle-level modeling of the LPSU. Andrew Chien and Kevin Lin implemented a preliminary version of the XLOOPS FPGA prototype that could run vector-vector add. Asha Ganesan contributed in bringing up the Zedboard framework and particularly helped in writing several key adapters and in implementing the clock-domain crossing logic on the Zedboard. Professors Christopher Batten and Zhiru Zhang advised on all the aspects of the XLOOPS project. I presented our work on the XLOOPS project at MICRO-47 held at Cambrige, UK [SIT+14].

I was responsible for defining the SSA design space and in proposing all of the smart sharing mechanisms. I proposed the idea of exploiting instruction redundancy in fork-join-centric parallel programs, developed offline and online tools to analyze the instruction redundancy, implemented a work-stealing runtime with soft-barrier hints, and ported applications to the work-stealing runtime. I extended the Pydgin instruction set simulator framework that was used to model the SSA design space. Moyang Wang authored the original work-stealing runtime that was modified and used in the SSA project. Christopher Torng and Moyang Wang ported the Cilk application kernels and several PBBS kernels used in the evaluation of the SSAs. Derek Lockhart and Berkin Ilbeyi developed the Pydgin instruction set modeling framework. Berkin Ilbeyi extended the pydgin framework to handle multicore programs which served as a starting point for the SSA evaluation framework. My advisor Christopher Batten provided valuable feedback and was involved in developing the ideas for smart sharing mechanisms.

I collaborated with Ji Kim who was the lead for the FGSIMT project that was presented at ISCA-40 [KLST13] and the LTA project that was presented at MICRO-50 [KJT+17]. I ported application kernels from the Parboil/Rodinia benchmark suites and implemented custom kernels that were used in the evaluation of the FGSIMT project. I also implemented the RTL models for the crossbars in the FGSIMT memory system and helped with the RTL models for the baseline multicore system. Working on the FGSIMT project inspired me to think of mapping challenging loops that could not be efficiently mapped onto the FGSIMT hardware, and motivated me to design decoupled lanes as in XLOOPS. I developed the gem5-PyMTL co-simulation framework that was used in the LTA project and ported many application kernels used to evaluate the LTA platform. The LTA project separated the recursive task splitting to occur on the GPPs while the LTA lanes executed the base cases for loop tasks. Working on the LTA project inspired me to propose SSAs that elegantly execute fork-join-centric parallel programs.

I collaborated with Ritchie Zhao, Prof. Zhiru Zhang, and Prof. Christopher Batten to develop the PyMTL-PolyHS high-level synthesis framework. The PyMTL-PolyHS framework was used in the DAC'16 paper on decoupled HLS data structures lead and presented by Ritchie Zhao [ZLS+16]. I also collaborated with Steve Dai on the FPGA'17 paper [DZL+17] which focused on improving hazard resolution in HLS pipelines. Working on these projects exposed me to the HLS methodology and helped me to better understand the principles and challenges in designing CGRA-based BSAs.

This thesis was supported in part by an NSF CAREER Award #1149464, NSF XPS Award #1337240, NSF CRI Award #1512937, NSF SHF Award #1527065, DARPA Young Faculty Award #N66001-12-1-4239, AFOSR YIP Award #FA9550-15-1-0194, and equipment, tool, and/or IP donations from Intel, NVIDIA, Xilinx, Synopsys, and ARM.

CHAPTER 2
A CASE FOR LANE-BASED BEHAVIOR-SPECIFIC ACCELERATORS

Exploiting parallelism is a key principle of hardware specialization. There exists a large gap between applications and mapping them to the underlying hardware. This chapter begins with the process of mapping applications to the underlying hardware by applying *parallelization strategies* and *scheduling strategies*. Parallelization and scheduling strategies provide guidelines for hardware specialization techniques. Section 2.2 makes a case for lane-based behavior-specific accelerators (BSAs). Section 2.3 presents the vision for a BSA chip and lays the roadmap for the remaining chapters of the thesis.

2.1 Parallelization and Scheduling Strategies

Modern hardware platforms are increasingly becoming more parallel. Parallelism is present in the form of threads, packed-SIMD extensions, GPGPUs, and parallel accelerators. Programmers can no longer rely on single-threaded programs to continue to improve performance given the recent hardware trends. However, there is a vast gap between mapping a given application to the underlying parallel hardware. An attractive but challenging approach is auto-parallelization of single-threaded programs. Auto-parallelization techniques commonly target loop control structures in programs and work well with regular control-flow and predictable dependences. Auto-parallelization approaches struggle with loops with complex dependences, pointers, recursion, and irregular algorithms. Fundamentally, auto-parallelization is limited by the assumptions and constraints imposed by serial programming.

Explicit parallel programming is an alternative approach where programmers expose parallelism in the form of compiler directives, function calls, or other language primitives. Explicit parallel programs can provide additional information on dependences that are otherwise not easy to capture by an auto-parallelizing compiler. In this work, I focus on explicit parallelization approaches for two reasons: (i) unlike auto-parallelization that starts with a serial program, explicit parallelism allows programmers to express algorithms that are better suited for parallel execution; (ii) explicit parallelism is the key to map challenging irregular algorithms which are difficult to analyze using auto-parallelizing compilers. To close the gap between applications and the un-

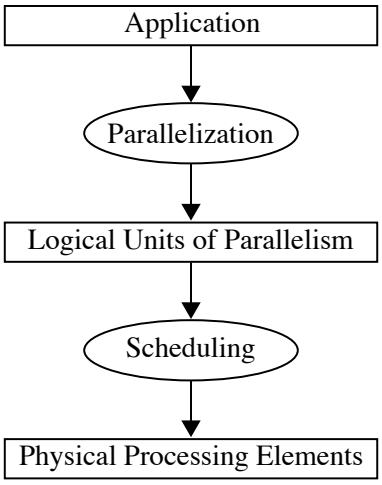


Figure 2.1: Mapping Applications to Hardware – A parallelization strategy decomposes an application into *logical units of parallelism* and a scheduling strategy maps the logical units of parallelism to *physical processing elements*.

derlying hardware, I use two abstractions: *logical units of parallelism* and *physical processing elements*. Figure 2.1 shows the process of mapping an application to the hardware using these two abstractions. A *parallelization strategy* aids in decomposing an application into logical units of parallelism (LUPs). The logical unit of parallelism is a useful abstraction as it makes it easier to reason about the description of an application and it's mapping to hardware. A *scheduling strategy* captures the mapping of LUPs onto the underlying hardware that is abstracted by the physical processing elements. A *parallel behavior* captures the available parallelism and the dependences between LUPs given a parallelization and scheduling strategy. Mismatches between the parallel behaviors and the physical processing elements increases the complexity and makes it awkward to map an application to a given hardware solution. For example, mapping loops with reductions, control-flow divergence, strided accesses, and carried dependences is challenging and ill-suited for packed-SIMD units [GNS13].

2.1.1 Parallelization Strategies

A parallelization strategy captures the parallelism present in an application by decomposing the application into logical units of parallelism. LUPs can be either fine-grained or coarse-grained. The granularity determines the overheads of creating and managing LUPs. The overheads of parallelization limit the scalability of an application and guide the selection of a suitable scheduling strategy. A parallelization strategy can capture static parallelism, where the LUPs are fixed and re-

Parallelization Strategy	Logical Units of Parallelism
Thread-Centric	Threads
Loop-Centric	Loop Iterations
Fork-Join-Centric	Tasks
Worklist-Centric	Tasks or WorkItems
Stream-Centric	Kernels
Operation-Centric	Operators

Table 2.1: Parallelization Strategies and Logical Units of Parallelism

```
1  void *func( void *thread_id ) {
2    int tid = (int)(thread_id);
3    compute( tid );
4    pthread_exit( NULL );
5  }
6
7  int main() {
8    pthread_t threads[NUM_THREADS];
9    for ( int i = 0; i < NUM_THREADS; ++i ) {
10     int rc = pthread_create( &threads[i], NULL, func, (void*) i );
11     if ( rc ) {
12       printf( "ERROR!\n" );
13       exit( -1 );
14     }
15   }
16   pthread_exit( NULL );
17 }
18
```

Figure 2.2: Thread-Centric Parallel Program using Pthreads API – The `pthread_create` function creates parallel threads for processing a function. Threads implement a part of the computation identified by the thread-id.

main constant at runtime, or can capture dynamic parallelism, where processing a LUP can create more LUPs. Further, LUPs can either be regular, i.e., each LUP captures an identical computation, or irregular, i.e., the time for processing a LUP can vary. An analysis of the nature of LUPs and the dependences between them can provide feedback for a programmer to reconsider the parallelization strategy and possibly, express an algorithm that is better suited for parallel execution. Table 2.1 shows the high-level organization of various parallelization strategies and the corresponding LUPs as considered in this work. A specialization technique is generally centered around a primary parallelization strategy but can be combined with more than one such strategy. Parallel programming frameworks embody a parallelization strategy in the form of special compiler directives, APIs, function calls, or special language constructs. Debugging and profiling tools that are associated with these frameworks help assist in analyzing the parallelization strategy.

```
#define SIZE      1000
#define CHUNKSIZE 100
int main() {
  int chunk = CHUNKSIZE;
  float src0[SIZE], src1[SIZE], dest[SIZE];
  initialize_sources( src0, src1, SIZE );
  #pragma omp for schedule(dynamic,chunk)
  for ( int i=0; i < N; ++i ) {
     c[i] = a[i] + b[i];
  }
}

```

Figure 2.3: Loop-Centric Parallel Program using OpenMP – Program parallelizes the addition of two floating-point arrays using the OpenMP parallel for directive. The `dynamic` clause groups the loop iterations into specified *chunks* and dynamically schedules the chunks onto the threads.

Thread-Centric Strategy – The thread-centric parallelization strategy is one of the prevailing approaches for parallelization. The LUP in thread-centric parallelization is a *thread*. A thread can be defined as a stream of instructions that is managed by the operating system. The behavior of thread-centric parallelization is defined by the interactions between threads in a program. Programming frameworks such as Pthreads, MPI, and OpenMP offer support for thread-centric parallelization. Figure 2.2 shows a parallel program expressed using the Pthreads API. Typically, the threads process a subset of the computation based on the thread-id. Theoretically, thread-centric parallelization gives programmers the utmost freedom to express and exploit parallelism. In practice, developing thread-centric parallel programs is challenging as the programmer is burdened with the tasks of carefully coordinating the thread interactions when accessing shared data structures, resolving deadlocks, and managing memory allocations. Further, thread-centric parallel programs are not portable and the parallelism cannot be turned-off on demand which makes it hard to debug and maintain these programs.

Loop-Centric Strategy – A loop-centric parallelization strategy exploits parallelism by executing independent loop iterations in parallel. Loop parallelism is very common in scientific computations. The LUP in a loop-centric strategy can either be fine-grained, i.e., a single iteration of a loop, or coarse-grained, i.e., a subset of parallel iterations. The behavior of loop-centric parallel programs is defined by the data-dependences amongst the loop iterations. Programming frameworks such as OpenMP, TBB, and Cilk offer high-level abstractions to express loop-level parallelism. Figure 2.3 shows the addition of two floating-point arrays that is parallelized by using

```
1  int fib( int n )
2  {
3    if (n < 2)
4      return n;
5    int x = cilk_spawn fib(n-1);
6    int y = fib(n-2);
7    cilk_sync;
8    return x + y;
9  }
10
```

Figure 2.4: Fork-Join-Centric Parallel Program using Intel Cilk Plus – The program computes a Fibonacci number using the *cilk_spawn* keyword to *fork* a child task and the *cilk_sync* keyword for the *join* synchronization.

OpenMP directives. OpenMP abstractions hide the details of the underlying hardware threads. Loops that have regular control-flow execute well when mapped onto packed-SIMD and vector units. Auto-vectorizing compilers in combination with the programming frameworks exploit parallelism across hardware threads and further within each hardware thread via packed-SIMD extensions. However, not all loops are regular with uniform control-flow. Loops with complex data-dependences have sufficient parallelism but do not map well to existing hardware platforms. Chapter 3 presents a novel architecture that exploits parallelism in the presence of challenging inter-iteration loop dependences.

Fork-Join-Centric Strategy – The fork-join-centric parallelization strategy expresses parallelism by identifying *tasks* that can potentially be executed in parallel. Tasks are a logical block of instructions that express a part of the computation and can be syntactically captured using function calls. The *fork* primitive allows a parent task to specify a child task that be executed in parallel, and the *join* primitive expresses a synchronization point for the child tasks to return to the parent tasks. Nested fork-join primitives are an elegant and productive way to write parallel programs. The behavior of a fork-join-centric parallel program can be modeled using a directed-acyclic-graph (DAG) representation where the nodes represent tasks and the edges capture the fork-join relationships. The DAG model [BL99] has been a well studied subject to reason about the available parallelism in fork-join-centric parallel programs. Examples of programming frameworks that support fork-join-centric parallelization include Intel's C++ Threading Building Blocks (TBB) [Rei07, int15], Intel's Cilk Plus [Lei09, int13], Microsoft's .NET Task Parallel Library [LSB09, CJMT10], Java's Fork/Join Framework [Lea00, jav15], and OpenMP. Figure 2.4 shows an example for a program that computes a Fibonnaci number using the Intel Cilk Plus

framework. Chapter 4 explores a design space for accelerators that reduce the area and improve the work efficiency when executing fork-join-centric parallel programs.

Worklist-Centric Strategy – In the worklist-centric parallelization strategy, an algorithm is viewed in terms of actions on shared global data structures. The actions are captured by *workitems* that are stored in an abstract *worklist* data structure. Processing the workitems out of the worklist can be *ordered* based on programmer annotations or *unordered* where any valid order is acceptable. The LUP in a worklist-centric parallel program is the workitem object. The behavior of a worklist-centric parallel program is expressed by the interactions between workitems and their actions on the global shared data structure. The worklist-centric parallelization strategy is useful to parallelize challenging irregular applications where the data-dependences between workitems are complex functions of runtime data values and in scenarios where processing a workitem may result in the addition of more workitems into the worklist. The worklist-centric parallelization strategy has been popularized by the Galois Framework [PNK+11, gal18].

Stream-Centric Strategy – In the stream-centric parallelization strategy, computation is organized as *streams* of data that flow from input sources to outputs, and the transformations on the streams are expressed as *kernels*. The LUPs are the various computational kernels that transform the incoming streams. The parallel behavior of a stream-centric parallel program is captured by high-level stream-dataflow graphs. The stream-centric parallelization strategy is attractive for applications that care about the overall throughput. Stream-centric parallel programs capture and express parallelism both within a kernel and across kernels while minimally complicating the programming abstractions. Stream-centric parallelization has been successful when restricted to a particular domains such as audio, video, and signal processing applications. StreamIt [Thi09] is an example of a recent stream-centric programming language and compiler effort that targets large streaming applications.

Operation-Centric Strategy – The operation-centric parallelization strategy decomposes an algorithm into fine-grained units of parallelism represented by *operations*. Operations can represent a single instruction or a small group of related instructions. The parallel behavior of operation-centric parallel programs can be represented by explicit dataflow graphs. A dataflow graph consists of operations represented by the nodes, and the edges represent the dependences between operations. The control-flow between operations are represented as additional edges between nodes in the dataflow graph. A specialized solution that employs an operation-centric strategy requires a

custom compilation flow to systematically transform a high-level algorithm into operations. The compiler is responsible for identifying profitable operation representations and mapping them to the underlying hardware. It is not clear if a programmer can provide additional information to aid the compiler. Needle [KSS+17], is an example of a recent LLVM-based compiler framework that leverages dynamic profile information to identify "what paths to specialize" in a program, merge and transform such paths into operations, and prepare them for acceleration.

2.1.2 Scheduling Strategies

A scheduling strategy maps the LUPs onto physical processing elements. Physical processing elements include functional units in a CGRA/spatial accelerator, packed-SIMD units, lanes in a vector processor or a GPGPU, and hardware threads in a CMP platform. It is often the case that a parallelization strategy *over decomposes* a given application, i.e., the number of LUPs exceeds the number of physical processing elements. A scheduling strategy can affect the overall performance, data locality, and may affect the total amount of computation performed by a program. The assignment of LUPs to processing elements can be done either statically by a compiler or dynamically at runtime. Dynamic scheduling for specialized hardware can be implemented in software, in hardware, or in a combination of both software and hardware.

Static Scheduling – Static scheduling can be used when the number of LUPs and the dependences between LUPs can be determined statically. Static scheduling is applicable in scenarios where the execution time for a given LUP can be estimated accurately at compile time as scheduling decisions affect the load balance of the system. Examples for static scheduling include the compile time mapping of operations onto CGRA functional units and compile time mapping of loop iterations onto packed-SIMD and vector units. Static scheduling reduces the cost of scheduling in terms of performance as the assignments of LUPs onto processing elements is made offline.

Dynamic Scheduling – Dynamic scheduling is required for scenarios when the dependences between LUPs are not known statically or when processing a unit of parallelism results in the creation of new work dynamically. Dynamic scheduling is also useful in scenarios where the amount of work is fixed and the dependences are known statically but the execution time of LUPs varies. Dynamic scheduling can be broadly classified based on whether the dependences are known statically at compilation time or the dependences are known dynamically at the runtime. If the LUPs are known to be independent at compile time then a *work sharing* strategy can be used.

If the dependences between LUPs cannot be resolved at compile time then a *work stealing* or a *speculation-based* strategy can be used.

Work Sharing is a dynamic load-balancing strategy where units of work, such as loop iterations, are stored in a central scheduling data structure. Processing elements, such as threads, interactively retrieve loop iterations out of the central data structure using locks or other forms of synchronization. For example, the OpenMP programming framework provides the *dynamic* and *guided* clauses that support work sharing of iterations in a parallel for loop. The OpenMP dynamic scheduling strategy uses an internal work queue to give out a chunk-sized block of iterations to a hardware thread. When a thread finishes the execution of a block, it retrieves the next block from the work queue. The OpenMP guided scheduling strategy is similar to the dynamic scheduling but the chunk size starts off to be large and decreases steadily as the loop nears completion. Chunking reduces the overheads of scheduling and may improve locality.

Work Stealing is a dynamic load-balancing strategy where each processing element maintains a local deque data structure to store work. Each processing element executes work retrieved from its local deque and if empty, selects a deque that belongs to a different processing element to *steal* work. Work stealing has been popularized by the Cilk programming language and is implemented in frameworks such as Intel Cilk Plus and Intel TBB. The work stealing strategy is paired with fork-join-centric parallelization. Carbon is a hardware-only approach that implements work stealing to support fine-grain parallelism [KHN07]. Each processor maintains a local task unit to store tasks and a global task unit balances the work across local task units. ADM presents a software/hardware co-designed approach to implement work stealing [SYK10]. In ADM, processors communicate work by sending direct messages using a specialized network to reduce the work stealing overheads. Chapter 4 focuses on exploiting instruction redundancy in fork-join-centric parallel programs that implement load balancing using the work stealing strategy.

Speculation-based scheduling strategies are used when the data dependences between LUPs cannot be resolved statically at compile time. Well known examples of speculation-based scheduling strategies include thread-level speculation (TLS) [SCZM00,AMW+07,SBV95] and the operator-formulation of algorithms popularized by the Galois programming framework [PNK+11]. TLS is an approach where threads operate on loop iterations speculatively by performing potentially unsafe work and temporarily storing speculative state in a buffer or cache. The speculated work is resolved at a later point in time by throwing away an incorrect computations and restoring state

back accordingly. Chapter 3 uses a similar approach to parallelize and schedule loop iterations with memory dependences.

2.2 A Case for Lane-Based BSAs

The technology trends discussed in Chapter 1 motivate the trend towards parallelization and specialization. A systematic approach for hardware specialization begins with analyzing the parallelization and scheduling strategies. Parallelization and scheduling strategies define parallel behaviors that a behavior-specific accelerator can specialize for. Given a parallel behavior there are several interesting questions that arise: What is the right software/hardware abstraction? What kind of microarchitectures are best suited for a given parallel behavior? What about the design and integration costs? To answer these questions, consider various specialization approaches based on the flexibility vs. specialization axis (discussed in Section 1.3). There are two feasible approaches that are flexible yet specialized for a range of applications: CGRA-based and lane-based behavior-specific accelerators (BSAs).

CGRA-Based BSAs primarily use operation-centric parallelization and static-scheduling strategies. CGRA accelerators are flexible as they employ reconfigurable datapaths that can be configured to best suit a given application behavior. Compared to general-purpose processors, CGRAs execute an explicit dataflow graph efficiently without incurring the overheads of instruction fetch, decode, issue, and the switching of pipeline registers. CGRAs use distributed register state, specialized memories, and specialized networks to communicate values amongst the processing elements (PEs). To increase the utilization of the PEs, CGRAs combine operation-centric parallelization with other strategies such as loop-centric [GHS11] and stream-centric [NGAS17] strategies. Design principles such as decoupled-access execute enable the CGRAs to efficiently overlap the memory access with pipelined computations. Compared to prior lane-based approaches like the packed-SIMD and vector units, CGRAs can handle the acceleration of loops with predictable inter-iteration dependences.

Lane-Based BSAs are composed of instruction set programmable lanes that allow them to flexibly use a variety of parallelization and scheduling strategies. Popular lane-based accelerators, such as the packed-SIMD and vector units, use loop-centric parallelization with static scheduling strategies. Lane-Based BSAs are efficient as they amortize the instruction and data supply over-

heads compared to GPPs. Lane-Based BSAs can easily scale with increasing parallelism in an application and offer a high computational density. Compared to CGRA-based accelerators, lane-based BSAs provide a simpler programming approach due to standard compiler and debugging tools. Additionally, there has been a considerable amount of research spent in developing elegant programming models to program lane-based BSAs. Mapping loops with dependences or exploring speculation-based scheduling strategies is ongoing research that can extend the applicability of lane-based solutions.

To qualitatively compare CGRA-based vs. lane-based BSAs, I consider three important aspects of a given specialization approach: programmability, flexibility, and design costs. *Programmability* defines the accessibility of a specialization approach. CGRA-based BSAs often require complex compilation flows compared to existing lane-based BSAs such as packed-SIMD and vector units. The programming challenge for the CGRA-based BSAs necessitate automatic compilers that handle the detection of profitable subgraphs, detecting address patterns, loop-interchange/flattening, explicit dependence insertion, and memory tiling, which adds additional complexity compared to the requirements of autovectorizing compilers. Virtualizing a given CGRA substrate for a larger dataflow graph further increases the compiler complexity. CGRA configuration bits are not portable across solutions compared to instruction set programmable abstractions used by lane-based BSAs. Further, there is a lack of tools for debugging CGRA-based programs, and it is not obvious for a programmer to restructure an algorithm to efficiently map onto CGRAs.

Flexibility of a specialization approach reduces the risk of obsoletion and increases the use of a solution across a range of applications. CGRA-based and lane-based BSAs perform well on applications with regular control-flow and large amounts of parallelism. However, CGRA-based BSAs exhibit poor performance for applications with irregular control-flow and are typically underutilized for such applications. Irregular applications have low regions of ILP which causes most of the processing elements in a CGRA to be inactive. While lane-based BSAs are also challenged by irregular applications, advancements in compiler technology and hardware predication have made packed-SIMD and vector units applicable to a subset of these applications. The lane-based BSAs proposed in this thesis extend the capabilities of prior approaches to efficiently execute irregular applications that have complex loops with inter-iteration dependences and recursive task parallelism.

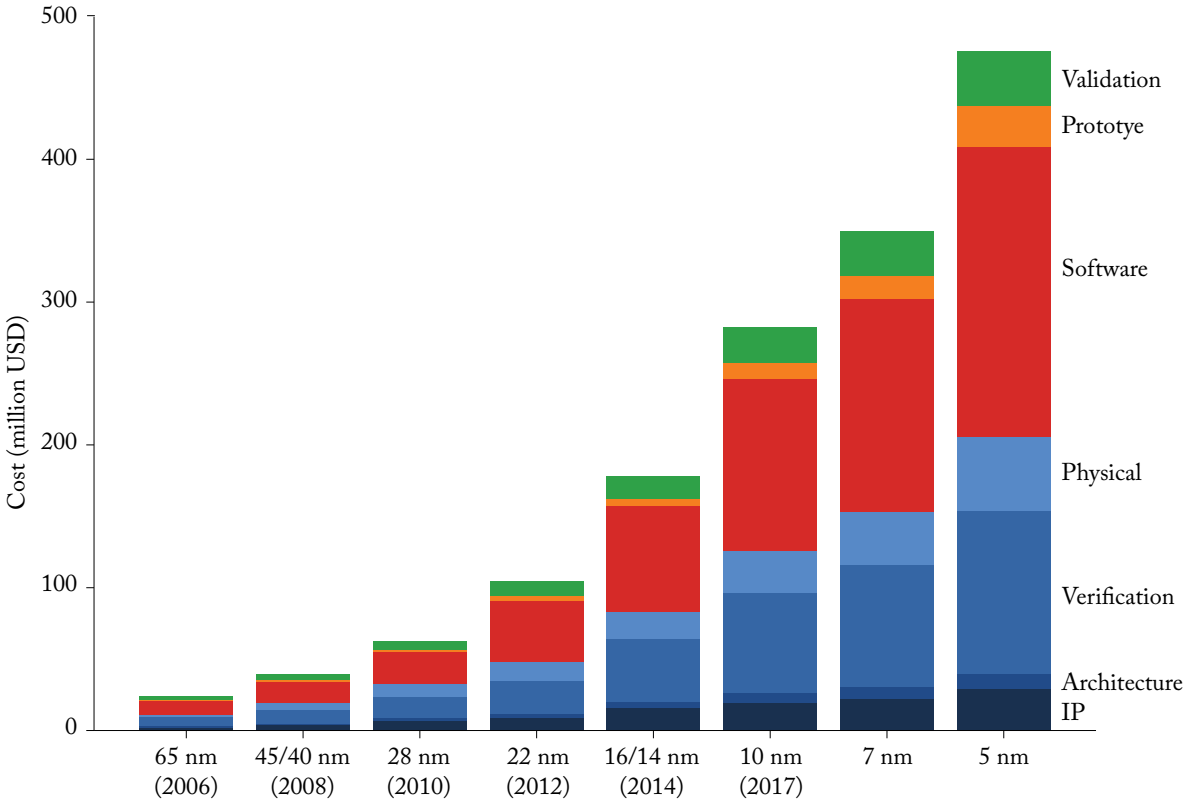


Figure 2.5: Soc Cost Breakdown – Estimated cost breakdown to build a large SoC based on data collected by International Business Strategies (IBS) [Str17] and as presented in [BPHH18]. The overall cost is increasing exponentially, and software comprises nearly half of the total cost.

Ultimately, any specialization approach has to be justified by the incurred *design costs*. Figure 2.5 shows the non-recurring engineering (NRE) costs involved in building a high-end SoC over recent technology nodes. Several components contribute to the non-recurring engineering costs for developing a specialized platform. The NRE costs include the price of developing lithography masks and the tools for design, costs of verification, and the cost of developing software to run on the platform. From an architects perspective, little can be done to mitigate the costs of developing lithography masks and manufacturing costs. From an implementation point of view, instruction set programmable lane-based BSAs are easier to build, verify, and reuse compared to CGRA-based BSAs. The figure shows that software contributes to roughly about 40% of the total costs in advanced technologies. Lane-Based BSAs are more programmable and flexible compared to CGRA-based BSAs which help reduce the software costs.

To minimize the NRE costs in building lane-based accelerators, the central theme of this thesis is to propose lightweight changes to applications, runtimes, compilers, instruction sets, and mi-

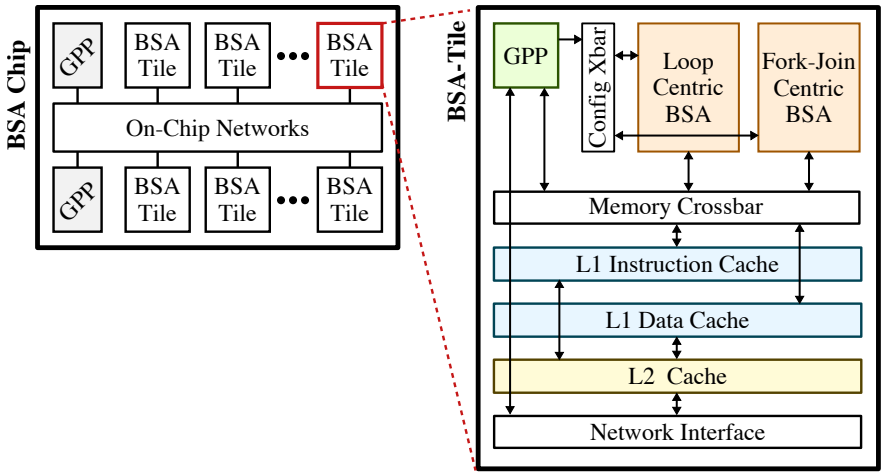


Figure 2.6: BSA Chip– A behavior-specific accelerator chip is a heterogenous CMP platform that includes general-purpose processor (GPP) tiles along with tiles that are augmented with loop- and fork-join-centric BSAs. The loop- and fork-join-centric accelerators share the L1 instruction and the data cache with the host GPP.

croarchitectures. Lightweight changes reduce the costs of design, verification, and programming. The accelerators proposed in this thesis focus on single-ISA heterogeneous architectures that transparently integrate traditional GPPs and specialized architectures. With a single-ISA and minimal changes to microarchitectures, we maintain the benefits of executing a single binary on either traditional GPPs or specialized BSAs, thereby reducing the barrier of adoption for the proposed lane-based BSAs. The lane-based-BSA approach primarily employs a parallelization strategy and provides an extensible template that can embed the principles of CGRA-based BSAs to accelerate logical units of parallelism.

2.3 Vision for Behavior-Specific Accelerators

A behavior-specific accelerator exploits the parallel behavior defined by a given parallelization and scheduling strategy. Figure 2.6 shows the high-level organization of a BSA chip. A BSA chip is a heterogeneous CMP platform that composes tiles with GPPs and tiles with GPPs that are augmented with lane-based BSAs. Lane-Based BSAs are composed of programmable lanes that are configured by a GPP to execute a parallel region. Lanes share expensive hardware resources, which include L1 instruction and data cache, the integer multiply-divide unit, and the floating-point unit. The reduced costs in area can be used to instead add more tiles, BSAs, or caches to

improve the overall system performance. BSAs are efficient as they are specialized for a given parallel behavior and can exploit similarities present in the instruction stream.

The execution model for a tile that is augmented by the BSA proceeds as follows: upon encountering a parallel region identified by a software/hardware interface, the GPP transfers the execution of the program by selecting the appropriate BSA. A BSA which is not applicable for the execution of a parallel region remains idle and is clock-gated. The GPP remains idle during the parallel execution which results in no contention for the cache ports. The BSA yields the control back to the GPP when it has finished executing a parallel region. We envision the execution to migrate efficiently between the GPP and the BSA thereby enabling an adaptive execution paradigm where online profiling maps the computation to the best suited processing element, i.e., the GPP or the BSA. Integrating the BSAs and the GPPs using the L1 cache hierarchy offers a nice trade-off in terms of the accelerator configuration and communication overheads. Integrating at the L1 cache lowers the configuration and communication overheads that enables fine-grained parallel regions.

Chapter 3 presents an elegant new XLOOPS hardware/software abstraction that explicitly encodes inter-iteration loop dependence patterns and enables performance-portable execution of loops. The chapter discusses a novel *loop pattern specialization unit (LPSU)* that augments the GPP and handles the execution of loops with complex inter-iteration dependences. XLOOPS binaries can execute on the GPPs using traditional execution, on the LPSU using specialized execution, or can use adaptive execution to choose between traditional or specialized execution to balance performance and energy efficiency. Chapter 4 presents *smart sharing architectures (SSAs)* which are a CMP-based solution where a GPP is augmented with *conjoined lanes*. Conjoined lanes are BSAs that can efficiently execute fork-join-centric parallel programs by employing smart-sharing mechanisms. Conjoined lanes execute the same runtime and the instruction set as the host GPPs and thereby can transparently integrate into existing CMP solutions. The goal of SSAs is to save area, maximize efficiency, and minimize loss in performance while sharing hardware resources.

CHAPTER 3
XLOOPS: LANE-BASED BSAS FOR LOOP-CENTRIC PARALLELIZATION AND SCHEDULING STRATEGIES

Computer architects have long realized the importance of focusing on the key loops that often dominate application performance. Hardware specialization for loop-centric parallel programs can exploit intra- and/or inter-iteration loop dependence patterns. In this chapter, we propose a new approach called explicit loop specialization (XLOOPS) based on the idea of encoding inter-iteration loop dependence patterns in the instruction set. The XLOOPS hardware/software abstraction requires only lightweight changes to a general-purpose compiler to generate XLOOPS binaries and enables executing these binaries on: (1) traditional microarchitectures with minimal performance impact, (2) specialized microarchitectures to improve performance and/or energy efficiency, and (3) adaptive microarchitectures that can seamlessly migrate loops between traditional and specialized execution to dynamically trade-off performance vs. energy efficiency.

Section 3.1 briefly discusses hardware specialization for loop-centric parallel programs and motivates the approach taken in this chapter. Section 3.2 describes the design of XLOOPS instruction sets, compilers, and microarchitectures. Our XLOOPS instruction set can encode: data-dependence patterns where the loops can appear to execute in any order both concurrently or atomically; data-dependence patterns where the loops must preserve ordering constraints expressed through either register or memory dependences; and control-dependence patterns based on fixed and dynamic bounds. Our XLOOPS compiler uses programmer annotations to automatically generate an efficient XLOOPS binary. The XLOOPS microarchitectures support a new execution paradigm based on traditional, specialized, and adaptive execution. To make the case for XLOOPS, we use a vertically integrated evaluation methodology. Section 3.3 describes the application kernels we use for evaluation and modifications to an LLVM-based compiler to support XLOOPS. Section 3.4 describes the cycle-level modeling of XLOOPS microarchitectures that support traditional, specialized, and adaptive execution. Section 3.5 describes the register-transfer-level (RTL) implementation of a simple XLOOPS microarchitecture capable of specialized execution and area, energy, and timing analysis using a commercial ASIC CAD toolflow. Section 3.6 presents results for a simple XLOOPS FPGA prototype that can execute loops that have no inter-iteration loop dependences, and Section 3.7 discusses related work.

Using specialized execution, XLOOPS is able to achieve 2.5× or higher speedup at similar or better energy efficiency for most application kernels compared to a simple single-issue in-order processor with only 40% area overhead. Compared to aggressive two- and four-way out-of-order processors, XLOOPS is able to achieve 1.5–3× improvement in energy efficiency while having speedups in the range of 1.25–2.5× on most application kernels. Adaptive execution enables applications that perform worse with specialized execution to automatically migrate to the general-purpose processor for increased performance at reduced energy efficiency.

3.1 Introduction

Hardware specialization techniques that primarily employ operation-centric parallelization and scheduling strategies exploit *intra-iteration* loop dependence patterns. These techniques usually involve custom instructions and/or small reprogrammable functional units well-suited to accelerating common sequences of operations *within* an iteration. Examples include application-specific instruction-set processors [API03, CFHZ04] and techniques for subgraph execution [CKP+04, GHS11]. Hardware specialization techniques that primarily focus on loop-centric parallelization and scheduling strategies exploit *inter-iteration* loop dependence patterns. These techniques focus at a higher level on how *different* loop iterations interact. Examples include data-parallel accelerators which exploit loops with no inter-iteration dependences [WAK+96, KP03, EVS98] and thread-level speculation which exploit loops with infrequent inter-iteration dependences [SBV95, SCZM00, KT99]. Some hardware specialization techniques such as coarse-grained reconfigurable arrays [GFA+11,GHN+12] and weakly programmable application-specific accelerators [VSG+11] target both intra- and inter-iteration loop dependence patterns.

All of these proposals must carefully navigate the tension between less efficient general architectures and more efficient specialized architectures. Some argue for exposing as much of the specialized microarchitecture as possible to enable flexible software configuration while maintaining efficiency [GNS13, DBBS+08]. Unfortunately, this comes at the expense of a clean hardware/software abstraction; highly configurable specialized architectures are often tightly coupled to a specific microarchitectural implementation. A key research challenge involves creating clean hardware/software abstractions that are highly flexible, yet still enable efficient execution on both traditional and specialized microarchitectures.

To address this challenge, we focus on architectural specialization for inter-iteration loop dependence patterns. *Inter-iteration data-dependence patterns* include loops with no inter-iteration dependences and loops with inter-iteration dependences encoded through registers and/or memory. An interesting data-dependence pattern often found in graph algorithms involves iterations that manipulate a shared data structure such that the iterations can be executed in any order as long as their updates to memory appear atomic to the other iterations. *Inter-iteration control-dependence patterns* include loops that terminate based on comparing an induction variable to a loop-invariant fixed bound, or loops that terminate based on a data-dependent-exit condition. An interesting control-dependence pattern found in more irregular worklist-based algorithms involves a loop induction variable compared to a dynamic bound that is monotonically increased during the loop execution. The inter-iteration dependence pattern for a given loop will be a combination of a specific data- and control-dependence pattern, and nested loops can be captured using the composition of multiple loop patterns.

Inter-iteration data and control dependence patterns influence the selection of a scheduling strategy. Most commonly found loops have no inter-iteration dependences and have a control-dependence with a fixed bound on the number of iterations. A scheduling strategy that achieves dynamic load-balancing by work-sharing constructs either in hardware or software is best suited for such loops. For loops with inter-iteration data-dependences encoded through registers and fixed bounds, a static scheduling strategy is employed as the data-dependences dictate the sequence of execution. For more complex loops that include inter iteration data-dependences that are encoded through memory or loops that need to appear atomic which may additionally have data-dependent control, a scheduling strategy that is speculation-based is required as the manifestation of these dependences are based on data dependent values. In a speculation-based strategy, the loop iterations are executed assuming no dependence violations and the state needs to be buffered such that the correct values can be restored upon any incorrect speculation.

The focus of this chapter is to specialize fine-grain loops that contain loop bodies on the order of one to two hundred instructions. Fine-grain loops necessitate ultra-low-overhead mechanisms to achieve significant speedups at low energy. Our approach, *explicit loop specilzation* (XLOOPS), is based on the idea of explicitly encoding inter-iteration loop dependence patterns in the instruction set to enable exploiting fine-grain loop-level parallelism. The XLOOPS hardware/software interface is lightweight and requires minimal changes to the compilers and microarchitectures that

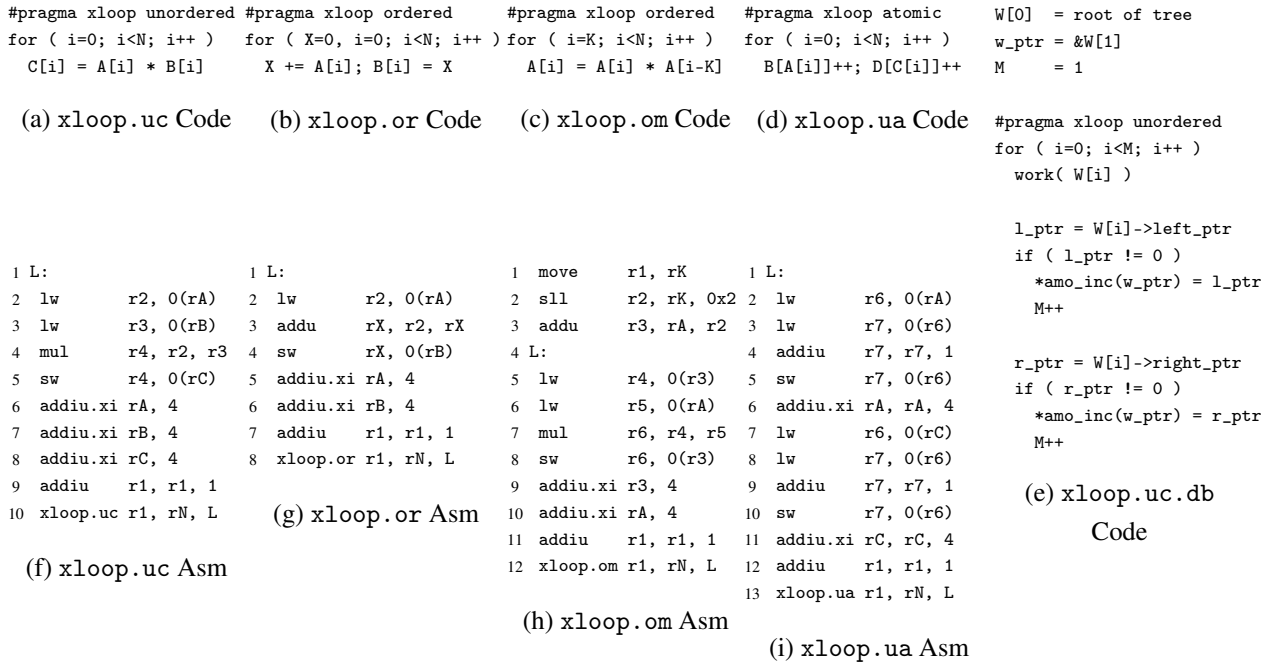


Figure 3.1: XLOOPS Instruction Set Examples – Unless otherwise specified, a fixed-bound control-dependence pattern is assumed. (a,f) `xloop.uc` encodes an unordered-concurrent data-dependence pattern, `addiu.xi` encodes a simple associative loop-carried dependence; (b,g) `xloop.or` encodes an ordered-through-registers data-dependence pattern, line 3 captures the loop-carried dependence through `rX`; (c,h) `xloop.om` encodes an ordered-through-memory data-dependence pattern, line 6 depends on an earlier instance of line 8; (d,i) `xloop.ua` encodes an unordered-atomic data-dependence pattern; (e) `xloop.uc.db` encodes an unordered-concurrent data-dependence with a dynamic-bound control-dependence pattern, `amo_inc()` uses an atomic memory operation to increment the tail pointer of the worklist by four.

execute XLOOPS binaries. The XLOOPS instructions encode the inter-iteration dependence patterns as hints which can be ignored for traditional execution. The XLOOPS interface transparently integrates general-purpose processors and lane-based BSAs that employ hardware specialization to schedule loop iterations onto the lanes given the hints as provided in XLOOPS binaries. Lastly, the XLOOPS abstraction enables adaptive execution where a loop can migrate seamlessly between traditional and specialized execution.

3.2 XLOOPS: Explicit Loop Specialization

In this section, we describe the instruction set and compiler modifications required for XLOOPS, and we propose various XLOOPS microarchitectures to enable traditional, specialized, and adaptive execution.

xloop.{d}.{c}	rI, rN, L	goto L if R[rI] $\neq$ R[rN]
addiu.xi	rX, imm	R[rX] $\leftarrow$ R[rX] + imm
addu.xi	rX, rT	R[rX] $\leftarrow$ R[rX] + R[rT]

Table 3.1: XLOOPS Instruction Set Extensions – Loop body is static sequence of instructions between L and the xloop instruction. {d} indicates data-dependence pattern: uc = unordered concurrent, or = ordered through registers, om = ordered through memory, orm = ordered through registers and memory, ua = unordered atomic. {c} indicates control-dependence pattern: no suffix implies fixed bound, db = dynamic bound.

3.2.1 XLOOPS Instruction Set

The XLOOPS instruction set is carefully designed to enable efficient execution on *both* traditional general-purpose processors (serial execution) and specialized microarchitectures (parallel execution). The XLOOPS instruction set is formed by extending a general-purpose instruction set with the instructions shown in Table 3.1. The key idea is to express inherent loop-level parallelism by encoding inter-iteration data- and control-dependence patterns using variants of the xloop instruction. All xloop instructions encode the notion of a *parallel loop body* which is defined as the static instruction sequence between a given label L and the address of the xloop instruction. It is undefined for the label L to point to an address greater than or equal to the address of the xloop instruction. Figure 3.1 uses short pseudocode and assembly examples to illustrate how these instructions are used in practice. The suffixes for the xloop instruction indicate the data- and control-dependence patterns. An xloop can contain arbitrary instructions including: arithmetic operations, memory operations, atomic memory operations (AMOs), memory fences, control flow, nested xloops, and system calls (although this is not recommended). Currently, the xloop instruction only supports fixed- and dynamic-bound control-dependence patterns; we leave exploring data-dependent-exit control-dependence patterns to future work. An xloop cannot write live-in registers and all live-out register values are undefined once the loop is finished executing, meaning an xloop must store results in memory.

xi **Instruction –** Mutual induction variables (MIVs) are variables that can be computed as a linear function of a loop induction variable. Modern compilers include a *loop-strength reduction* pass that transforms expensive MIV computations into cheap iterative operations. Naively using such optimizations can impose extra, potentially unnecessary inter-iteration dependences, but avoiding such optimizations can introduce non-trivial address computation overhead, especially when working with multi-dimensional arrays. The cross-iteration instructions (denoted with an xi

suffix) explicitly encode MIVs to allow hardware to handle MIVs either iteratively or in parallel using specialized logic. Note that the register operand R[rT] in an addu.xi instruction must be a loop-invariant value. The instructions on lines 6–8 in Figure 3.1(f) illustrate the use of the xi instruction.

xloop.uc **Instruction –** An xloop.uc encodes an unordered-concurrent data-dependence pattern. The iterations can appear to execute concurrently and in any order. Data races are possible, but atomic memory operations can provide efficient synchronization if required. Figure 3.1(a,f) illustrates using an xloop.uc for element-wise vector multiplication. The XLOOPS ISA specifies that an addiu writing the loop induction variable (e.g., line 9) does not impose an ordering constraint.

xloop.or **Instruction –** An xloop.or encodes an ordered-through-registers data-dependence pattern. We term registers that impose ordering constraints as *cross-iteration registers* (CIRs). The value in a CIR for a given iteration must be the same as if the xloop was executed serially. Any general-purpose register can act as a CIR. The CIRs must be read at least once and can be written zero or more times. As an exception to the restriction on xloop register live-outs, each CIR is guaranteed to have the same value as a serial execution when the loop is finished. As with an xloop.uc, there are no ordering constraints with respect to memory, so memory races are possible. Figure 3.1(b,g) illustrates using an xloop.or to implement parallel-prefix summation with rX as a CIR.

xloop.om **Instruction –** An xloop.om encodes an ordered-through-memory data-dependence pattern. Values read and written to memory must be the same as if the loop was executed serially. Since an xloop.om guarantees a specific order with respect to memory, there can be no race conditions. For example, if each iteration updates different portions of a shared data structure, then iterations may occasionally conflict in which case the updates are guaranteed to occur in the same order as if the loop was executed serially. Figure 3.1(c,h) illustrates using an xloop.om to implement a simple loop where the load instruction on line 6 in iteration i depends on the store instruction on line 8 in iteration i-K. An xloop.orm encodes a pattern that combines ordering through registers and memory.

xloop.ua **Instruction –** An xloop.ua encodes an unordered-atomic data-dependence pattern. The iterations can appear to execute in any order, but their memory updates must appear to execute atomically. While race conditions are not possible, the results can be non-deterministic since

the hardware is free to reorder iterations. This data-dependence pattern is often found in graph algorithms that manipulate a shared data structure where the iterations can execute in any order given that iterations update memory atomically. Figure 3.1(d,i) illustrates using an `xloop.ua` to modify two histograms with a single atomic update.

`xloop.*.db` **Instruction –** The above data-dependence patterns assume a fixed-bound control-dependence pattern. An `xloop.*.db` encodes a different inter-iteration control-dependence pattern where iterations are allowed to monotonically increase the loop bound. Figure 3.1(e) illustrates using an `xloop.uc.db` to perform work on a binary tree using a worklist-based implementation. Each iteration uses an AMO to reserve space at the tail of the worklist before adding new nodes and incrementing the loop bound. This example could also be encoded as an outer for loop with an inner `xloop.uc` to iterate over the nodes in a given level of the tree, but using an `xloop.uc.db` results in a more natural mapping and can enable more efficient specialized execution.

The XLOOPS instruction set provides precise exceptions at the instruction level within an `xloop` iteration. This means exceptions within a loop iteration are guaranteed to occur in order with respect to the other instructions *in that loop iteration*. Exceptions in *different* iterations of an `xloop.{uc,ua,or}` can occur in any order; exceptions in *different* iterations of an `xloop.{om,orm}` are guaranteed to occur in the same order as a serial execution.

The XLOOPS ISA is a clean hardware/software abstraction that provides significant freedom when designing XLOOPS compilers and XLOOPS microarchitectures. Any given loop can usually be encoded in multiple ways. For example, any valid `xloop.uc` is also a valid `xloop.or`, any valid `xloop.ua` is also a valid `xloop.om`, and any fixed-bound `xloop` is a valid `xloop.orm`. Software should choose the "least restrictive" inter-iteration dependence pattern to enable execution on simpler specialized microarchitectures and to give hardware the most freedom in choosing how to execute the `xloop`. Specialized execution of `xloop.om` is more complex than `xloop.or` which in turn is more complex than an `xloop.uc`, so an architect can choose to only support specialized execution for an `xloop.uc` and use traditional execution for the remaining patterns. Similarly, the maximum number of instructions in an `xloop` is not part of the instruction set; while software should target fine-grain loops, it is perfectly fine to generate a relatively large loop body (e.g., 200 instructions). A specific microarchitecture can always fall back to a traditional execution if the `xloop` is too large. Finally, the XLOOPS instruction set enables cleanly nesting `xloop`s. Software

can provide hints to the hardware to indicate which `xloop` might be best for specialized execution, or the hardware might adaptively explore specialized executions for different `xloop`s.

3.2.2 XLOOPS Compiler

The XLOOPS compiler currently uses programmer inserted annotations to determine which loops to encode using the XLOOPS instruction set. Figure 3.1(a–e) illustrates using `#pragma` directives and the keywords `unordered`, `ordered`, and `atomic` to convey the data-dependence patterns. Figure 3.2 illustrates annotating nested loops in the Floyd-Warshall shortest path algorithm from the Polybench Suite [pbe14], and Figure 3.3 illustrates annotating an ordered loop in the maximal matching application kernel present from the Problem-Based Benchmark Suite [SBF+12].

The XLOOPS compiler is implemented with lightweight changes to an existing general-purpose compiler. The XLOOPS approach does not interfere with existing compiler algorithms for mid-level optimization passes, and back-end algorithms for instruction scheduling, register allocation, and code generation. The XLOOPS compiler modifies the loop-strength reduction pass to directly generate appropriate `xi` instructions to encode the MIVs. Loops annotated with the `unordered` keyword are usually encoded using `xloop.uc`. Loops annotated with the `atomic` keyword are encoded using `xloop.ua`. Programmers use the `ordered` keyword to annotate loops that must preserve inter-iteration data-dependences. The programmers need not specify whether this data-dependence is through registers or memory or both. The XLOOPS compiler includes analysis passes to determine how the data-dependence is communicated and encodes the dependence patterns using `xloop.{or/om/orm}`. Register dependence testing is implemented by analyzing the use-definition chains through the PHI nodes and identifying CIRs. Memory dependence testing is implemented using well known dependence analysis techniques such as the zero-, single-, and multiple-index variable tests [GKT91]. Additionally, the XLOOPS compiler includes a pass to detect updates to the loop bound to encode such loops using `xloop.*.db`.

Although these lightweight changes to a general-purpose compiler should produce a reasonable XLOOPS compiler, there are opportunities for additional optimizations. For example, the performance of executing an `xloop.or` is limited by the *inter-iteration critical path* for each CIR. The inter-iteration critical path is the distance between the first dynamic instruction in the loop body that reads the CIR and the last dynamic instruction in that same loop body that updates the CIR. Compiler optimizations to reduce the inter-iteration critical path by modifying the instruction

```
for ( int k = 0; k < n; k++ )
  #pragma xloops ordered
  for ( int i = 0; i < n; i++ )
    #pragma xloops unordered
    for ( int j = 0; j < n; j++ )
      path[i][j] = min( path[i][j], path[i][k] + path[k][j] );
```

Figure 3.2: C Code for *war* Application Kernel – Kernel from Polybench suite implementing Floyd-Warshall shortest path algorithm. XLOOPS compiler maps inner loop to `xloop.uc` and uses dependence analysis to map outer loop to `xloop.om`.

```
#pragma xloops ordered
for ( int i = 0; i < num_edges; i++ ) {
  int v = edges[i].v; int u = edges[i].u;
  if ( vertices[v] < 0 && vertices[u] < 0 ) {
    vertices[v] = u; vertices[u] = v; out[k++] = i;
} }
```

Figure 3.3: C Code for *mm* Application Kernel – Kernel from Problem-Based Benchmark Suite implementing greedy algorithm for maximal matching on undirected graph. XLOOPS compiler uses dependence analysis to map the loop to `xloop.orm`.

scheduling within the loop body could improve the ability of XLOOPS microarchitectures to overlap independent work from different iterations. We explore the potential of such an optimization by manually scheduling instructions in Section 3.4.7.

3.2.3 XLOOPS Traditional Execution

XLOOPS binaries can be executed efficiently on a general-purpose processor (GPP) with minimal changes to the decoder logic. An `xloop` instruction is executed as a conditional branch instruction, and an `xi` instruction is executed as a simple addition. Efficient traditional execution is important for two reasons: (1) to enable gradual adoption of XLOOPS without any penalty when using XLOOPS binaries on GPPs; and (2) to enable adaptive execution to migrate an `xloop` to a GPP if it is determined that specialized execution is not resulting in any performance benefit.

3.2.4 XLOOPS Specialized Execution

Figure 3.4 shows a novel XLOOPS microarchitecture that augments a GPP with a loop-pattern specialization unit (LPSU). The GPP can either be a simple in-order or a complex out-of-order processor. The LPSU contains four decoupled lanes and a lane management unit (LMU). In our

current design, each lane in the LPSU is similar to a simple in-order processor, but it is certainly possible to use more aggressive superscalar or out-of-order lane microarchitectures to better exploit intra-iteration instruction-level parallelism. Each lane includes a loop instruction buffer to store instructions, an index queue (IDQ) to store loop indices waiting for execution, a 2r2w-port physical register file, and functional units for simple arithmetic, address generation, and control flow. The GPP and LPSU dynamically arbitrate for the data memory port and the long-latency functional unit (LLFU). The LLFU provides support for integer multiplication, integer division, and floating-point arithmetic. Specialized execution occurs in two phases: a *scan phase* initiated by the GPP and a *specialized execution phase* that occurs on the LPSU.

Scan Phase – The GPP starts the scan phase when it reaches an `xloop` instruction. In this phase, the instructions and live-in register values within the loop body are incrementally written to the instruction buffers and register files in the LPSU. To reduce the required amount of physical register storage in the LPSU, the LMU renames architectural register specifiers and updates the instruction encoding as it writes instructions into the instruction buffers. Since the registers are renamed once during the scan phase, the energy consumed for register renaming is amortized over all iterations. A complex out-of-order GPP can overlap the scan phase with the execution of instructions that are before the `xloop` body. The specialized execution phase does not start until all previous instructions are retired, all instructions in the `xloop` body have been scanned, and the `xloop` instruction reaches the head of the reorder buffer. Once the scan phase is complete, the GPP stalls until the LPSU has finished the specialized execution phase.

Specialized Execution Phase – In this phase, the LMU enqueues iteration indices into the IDQs as free IDQ entries become available. For `xloop.uc`, IDQ entries can become available in any order enabling simple dynamic load balancing, while for other inter-iteration dependence patterns, IDQ entries naturally become available in iteration order. Each lane dequeues an iteration index and executes the corresponding iteration. Since the XLOOPS ISA guarantees live-in registers are not written in the `xloop` body, there is no need to restore state before the execution of each iteration. When the execution of the entire `xloop` is finished and all memory updates are complete, the LMU notifies the GPP that the specialized execution phase has ended.

`xi` **Execution –** The LPSU uses specialized logic to execute `xi` instructions. In the scan phase, the LMU uses a mutual induction variable table (MIVT) to track the register specifier for the MIV and the loop-invariant increment value (i.e., either `imm` for `addiu.xi` or R[`rT`] for `addu.xi`). In the

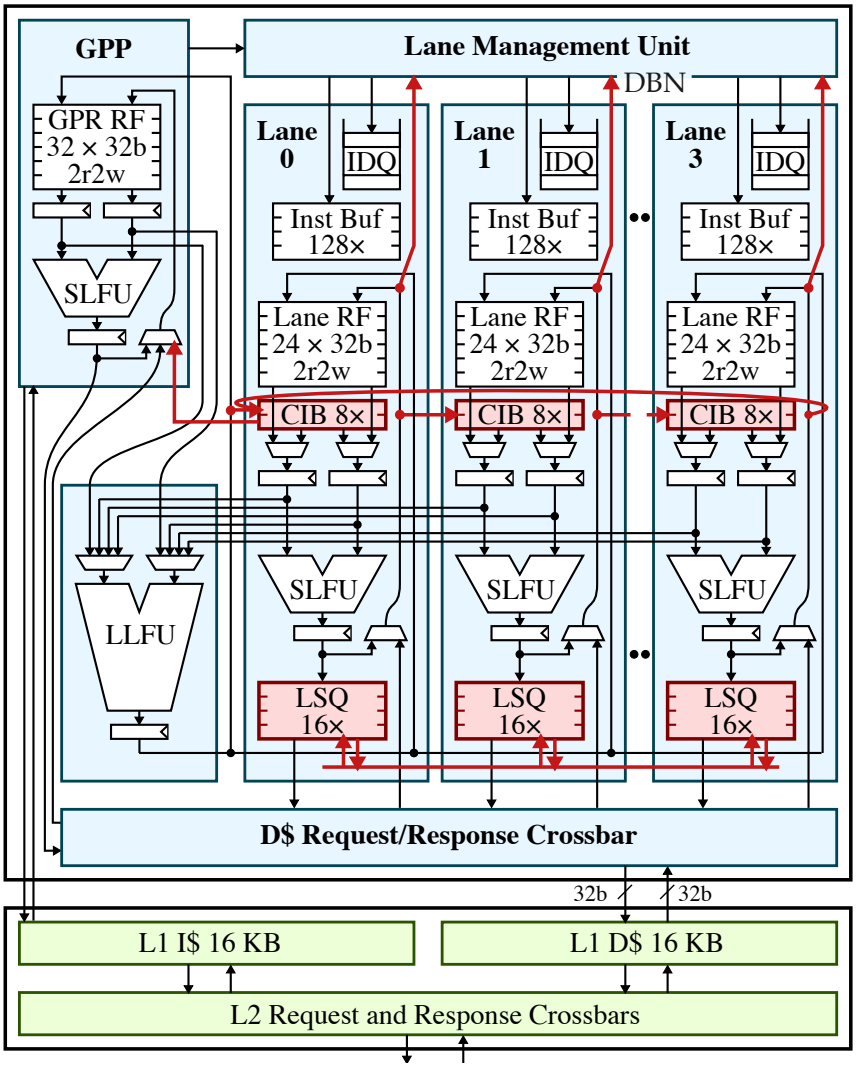


Figure 3.4: XLOOPS Microarchitecture – GPP and L1 memory system augmented with four-lane loop-pattern specialization unit (LPSU). Mechanisms required to support `xloop.{or,om,orm,ua}` beyond basic `xloop.uc` support are shown highlighted in red. GPP = general-purpose processor (either in-order or out-of-order); GPR = GPP regs; RF = regfile; IDQ = index queue; Inst Buf = instruction buffer; DBN = dynamic-bound notification; CIB = cross-iteration buffer; SLFU = short-latency functional unit; LLFU = long-latency functional unit; LSQ = load/store queue.

specialized execution phase, the lanes check the read register specifiers of a decoded instruction and compare it to the specifiers stored in the MIVT using a bit vector. If the register specifier matches an entry present in the MIVT, then the lane computes the value of the mutual induction variable using: the value present in the register file, the difference in the loop indices, and the loop-invariant increment value from the MIVT as shown:

$$\mathrm{R[rX]} \leftarrow \mathrm{R[rX]} + (\mathrm{increment} \times (1 + \mathrm{i}_{current} - \mathrm{i}_{previous}))$$

Since the difference in inter-iteration loop indices is small (usually close to the number of lanes), the lanes can use an inexpensive, narrow multiplier. When the lane executes the actual `xi` instruction, the result of the above computation is stored in the register file and used by the next iteration executed on the lane.

`xloop.uc` **Execution –** Supporting `xloop.uc` requires just the mechanisms described above. Figure 3.4 illustrates these mechanisms and highlights the additional mechanisms required to support the more sophisticated inter-iteration dependence patterns described in the rest of this section.

`xloop.or` **Execution –** The cross-iteration buffers (CIBs) between neighboring lanes are small associative buffers that are used to communicate inter-iteration register dependences when executing an `xloop.or`. The LMU needs to identify each CIR and the last instruction that updates each CIR. During the scan phase, the LMU uses two bit-vectors to track register reads and writes. Registers that are first read and then written are identified as CIRs. In scan phase, the LMU also tracks the largest PC of an instruction that updates a CIR and sets a special *last CIR write bit* in the instruction buffer for this instruction. When a lane executes an instruction, it checks if a source register is a CIR and stalls if the CIR value is not available in the CIB connected to the previous lane. If the value is available, it writes this value to the lane register file and uses this value for the execution of the instruction. The lane also checks the last CIR write bit when executing an instruction. If this bit is set, then the lane writes the instruction result to the CIB connected to the next lane. Updates to a CIR can be conditional depending on the dynamic control flow. If an instruction with the last CIR write bit set was skipped, then at the end of the iteration the lane will copy the corresponding CIR value to the CIB.

`xloop.om` **Execution –** Efficient parallel execution of `xloop.om` requires hardware memory disambiguation support to determine when speculative iterations violate the serial memory ordering constraint. Each lane includes a small 2r1w load-store queue (LSQ) to track memory accesses across iterations. Memory dependence ordering is enforced by the LMU based on the loop iteration index. A lane with the lowest iteration index is considered as non-speculative, whereas lanes with higher iteration indices are considered speculative. Loads and stores issued by a non-speculative lane are allowed to bypass the LSQ and access memory immediately. A store issued by a speculative lane is buffered in the speculative lane's LSQ and does not update memory. A load issued by a speculative lane first checks for a matching store address in the speculative lane's LSQ for store-load forwarding. If there is no match in the speculative lane's LSQ, the load is issued to the

memory system. More aggressive implementations can additionally allow a load to check the LSQs across lanes for inter-iteration store-load forwarding. To detect memory dependence violations, the address for each store issued by the non-speculative lane is broadcast to the speculative lanes when the store executes. Each speculative lane compares this broadcasted store address to the entries present in the speculative lane's LSQ. A memory dependence violation occurs if a speculative lane has already issued a load request to the same address as a store issued by a non-speculative lane. When a speculative lane detects a memory dependence violation, the lane restarts the execution of the corresponding iteration. Squashing iterations is fast since an `xloop` cannot write live-in registers; the lane simply flushes the pipeline (including the LSQ) and restarts execution from the first instruction in the `xloop` body. Speculative lanes stall execution if they fill up their corresponding LSQ. When the LMU promotes a speculative lane to be non-speculative, the lane drains its LSQ, broadcasts store requests to other lanes, and updates the memory. Supporting `xloop.orm` involves combining the mechanisms required for supporting both `xloop.or` and `xloop.om`.

`xloop.ua` **Execution –** Similar to `xloop.om`, efficient parallel execution of `xloop.ua` requires hardware memory disambiguation support. However, `xloop.ua` does not enforce sequential ordering of the loop iterations. Currently, we execute `xloop.ua` using the same mechanisms as `xloop.om`. Future work could explore microarchitectures that are less restrictive in terms of iteration index ordering and take better advantage of the `xloop.ua` data-dependence pattern.

`xloop.*.db` **Execution –** Execution of loops with a dynamic-bound control-dependence pattern is similar to loops with a fixed-bound dependence pattern with minor changes to the LMU and lane control logic. Each lane checks for instructions that update the register containing the loop bound and communicate the value of the updated loop bound to the LMU. The LMU updates the maximum bound for the loop execution and generates additional iteration indices which are enqueued in the IDQs as space becomes available. Mechanisms to execute any data-dependence pattern can be combined with the mechanism to execute the dynamic-bound control-dependence pattern, although in this work we focus on `xloop.uc.db`.

3.2.5 XLOOPS Adaptive Execution

For certain applications with significant intra-iteration instruction-level parallelism and limited inter-iteration parallelism, traditional execution on complex out-of-order GPPs can achieve better performance than specialized execution on the LPSU's simple in-order lanes. The XLOOPS

hardware/software abstraction enables microarchitectures to adaptively migrate an `xloop` between traditional execution on the GPP and specialized execution on the LPSU.

Adaptive execution adds two new phases for profiling. When the GPP first executes an `xloop` it begins a *GPP profiling phase* to determine the performance of traditional execution. After profiling for a set number of iterations or cycles, the scan phase takes place as described in Section 3.2.4. At the end of the scan phase, the GPP sends the number of profiled loop iterations and recorded cycles to the LPSU. The LPSU then begins an *LPSU profiling phase* to determine the performance of specialized execution. After the LPSU has executed the same number of iterations used in the GPP profiling phase, the LPSU compares the relative performance of traditional and specialized execution. If specialized execution is slower than traditional execution, the LPSU simply instructs the GPP to finish executing the remaining iterations. For `xloop.or`, CIR values for the last iteration executed on the LPSU are copied back to the GPP.

Migrating an `xloop` between the GPP and LPSU only occurs at loop iteration boundaries and involves transferring very little state. This makes `xloop` migration significantly more efficient compared to thread migration across cores with private caches. Since the profiling phase itself is a valid execution of the `xloop`, adaptive execution is an efficient mechanism that increases the performance of loops that struggle with specialized execution.

The GPP includes an adaptive profiling table (APT) to record the profiling progress for a small number of recently seen `xloop` instructions. The APT is indexed by the PC of the `xloop` instruction and contains an iteration count and, if profiling is complete, the decision on whether to use traditional or specialized execution for future dynamic instances of the `xloop`. When the GPP executes an `xloop` instruction, it checks the APT to see if it should continue profiling or immediately choose traditional or specialized execution. The APT enables the profiling phases to stretch across multiple dynamic instances of the `xloop` which is especially important for `xloop`s with small iteration counts. Our current implementation of adaptive execution does not reconsider the profiling results once a decision has been made, although this is an interesting direction for future work.

3.3 XLOOPS Application Kernels

We explored a diverse set of application kernels that capture multiple inter-iteration data- and control-dependence patterns for both single and nested loops. We include both numeric and non-

		Loop Characteristics			Dynamic Insns			io Speedups		ooo/2 Speedups			ooo/4 Speedups		
Name	**Suite**	**Type**	**Num Insns**	**Num Iters**	**GPI**	**XLI**	**X/G**	**T**	**S**	**T**	**S**	**A**	**T**	**S**	**A**
rgb2cmyk-uc	C	uc	32	80	209K	209K	1.00	1.00	3.13	1.00	2.24	2.18	1.00	1.22	1.21
sgemm-uc	C	uc	27	32	340K	340K	1.00	1.00	4.03	1.06	2.29	2.03	1.00	1.17	1.10
ssearch-uc	C	uc	37–58	57	2.3M	2.3M	1.00	1.00	3.93	1.07	2.65	2.56	0.99	1.52	1.51
symm-uc	Po	uc	43	32	267K	266K	1.00	1.01	3.38	1.00	1.97	1.95	1.03	1.08	1.08
viterbi-uc	C	uc	31–34	1–2K	2.5M	2.3M	0.92	1.07	2.57	1.14	2.10	2.10	1.13	1.15	1.13
war-uc	Po	uc	21	32	438K	438K	1.00	1.00	3.33	1.00	1.91	1.90	1.00	1.85	1.84
adpcm-or	M	or	52	20K	932K	992K	1.06	0.97	1.16	0.94	0.82	0.94	0.94	0.55	0.94
covar-or	Po	or	8–17	32	177K	161K	0.91	1.05	2.58	1.00	1.38	1.35	1.03	0.85	1.05
dither-or	C	or	36	256	2.3M	2.3M	1.00	1.12	1.49	1.07	0.90	1.07	0.95	0.58	0.95
kmeans-or	C	or,ua,uc	7–41	1–100	430K	428K	1.00	1.00	3.20	0.99	1.58	1.60	1.01	0.95	1.02
sha-or	M	or,uc	6–24	20–64	53K	51K	0.96	1.03	1.17	1.03	0.82	0.97	0.98	0.55	0.88
symm-or	Po	or	16	1–30	267K	268K	1.00	1.00	2.40	1.01	1.60	1.59	1.02	0.93	0.93
dynprog-om	Po	om	26	1–79	794K	795K	1.00	1.00	1.26	1.00	0.71	0.99	1.01	0.36	1.00
knn-om	P	om,uc	26–54	1–14	791K	750K	0.95	1.00	1.44	1.05	1.36	1.35	1.05	1.12	1.12
ksack-sm-om	C	om	21	99	50K	62K	1.23	0.77	2.72	0.56	1.71	1.64	0.36	1.05	1.03
ksack-lg-om	C	om	21	99	35K	39K	1.12	0.87	3.46	0.69	1.92	1.78	0.53	1.31	1.28
war-om	Po	om	21	32	438K	438K	1.00	1.00	1.09	1.00	0.63	0.99	1.00	0.60	0.99
mm-orm	P	orm,uc	7–22	256–2K	31K	31K	0.99	1.01	3.13	1.01	2.76	2.47	0.99	2.33	2.21
stencil-orm	Po	orm	20	126	639K	639K	1.00	1.00	1.02	1.00	0.66	1.00	1.00	0.66	1.00
btree-ua	C	ua,uc	11–14	1K	101K	101K	1.00	1.00	1.52	0.99	1.07	1.04	1.00	1.06	1.02
hsort-ua	C	ua	42–46	512–1K	274K	278K	1.01	0.99	1.34	0.96	0.88	1.00	1.10	0.71	1.13
huffman-ua	C	ua,uc	6–48	256–14K	290K	292K	1.01	0.96	1.57	0.97	1.09	1.18	0.99	0.74	0.96
rsort-ua	C	ua	12	1K	202K	218K	1.08	0.89	2.46	0.92	1.58	1.56	0.89	0.84	0.88
bfs-uc-db	C	uc.db	36	DYN	62K	64K	1.04	0.97	2.96	0.53	2.11	1.83	0.41	1.54	1.35
qsort-uc-db	C	uc.db	70	DYN	146K	136K	0.93	1.07	2.94	1.10	2.69	2.61	1.02	2.17	2.18

Table 3.2: XLOOPS Application Kernels and Cycle-Level Results – Suite shows the benchmark suites: Po = PolyBench; M = MiBench; P = PBBS; C = Custom. Loop characteristics shows: Type = the dependence pattern type (multiple entries means different `xloops`); Num Insns = range for static instruction counts for each `xloop` body; Num Iters = range for number of `xloop` iterations; Dynamic Insns = dynamic instruction counts for the timing critical loop; GPI = general-purpose ISA; XLI = XLOOPS ISA; X/G = normalized XLOOPS ISA dynamic instruction count compared to general-purpose ISA; *io* = in-order speedups; *ooo/2* = 2-way out-of-order speedups; *ooo/4* = 4-way out-of-order speedups; T = traditional execution; S = specialized execution; A = adaptive execution. Speedups are normalized to a standard serial implementation compiled for the general-purpose ISA and executed on the corresponding baseline GPP. For example, the **io:T** column shows the speedup of an XLOOPS binary using traditional execution on an in-order GPP relative to a serial implementation of the application kernels compiled for the general-purpose ISA executing on the same in-order GPP.

numeric kernels with regular and irregular data and control flow. Table 3.2 shows the list of application kernels and corresponding inter-iteration dependence patterns for each loop in the kernel. Our application kernels are drawn from MiBench [GRE+01], PolyBench [pbe14], PBBS [SBF+12], and our own custom kernels. The suffix for an application name indicates the inter-iteration dependence pattern that dominates the execution time. All of the kernels were parallelized by adding programmer annotations with minimal modifications to the original serial kernel. For select kernels, we also explored manual loop-transformations and hand-coded assembly implementations as described in Section 3.4.7.

We briefly describe the custom kernels. *rgb2cmyk-uc* performs color space conversion on a test image. *sgemm-uc* implements a single-precision matrix multiplication for square matrices using standard triple-nested loops. *ssearch-uc* implements the Knuth-Morris-Pratt algorithm to search a collection of byte streams for the presence of substrings. *viterbi-uc* decodes frames of convolutionally encoded data using the Viterbi algorithm. *dither-or* generates a black-and-white image from a gray-scale image using Floyd-Steinberg dithering. *kmeans-or* implements the *k*-means clustering algorithm. Assignment of objects to clusters is a dominant loop with inter-iteration register dependences. *ksack-*-om* solves the unbounded knapsack dynamic programming problem. For this problem, we have two variants, *ksack-sm-om* and *ksack-lg-om*, which have datasets of small (< 10) and large (> 10) weights respectively. *btree-ua* constructs a binary tree from a random set of integer inputs. *hsort-ua* implements the heapsort computation given a set of integer inputs. *huffman-ua* implements the Huffman entropy coding compression algorithm. *rsort-ua* performs an incremental radix sort on an array of integers. Each iteration updates histograms of digit lookups using a `xloop.ua` and computes prefix-sum updates for the next stage of sorting. *bfs-uc-db* uses a dynamically growing worklist to traverse an input graph in a breadth-first order and computes the distance given a source node to every other node. *qsort-uc-db* implements the quicksort algorithm using a dynamically growing worklist of partitions to be sorted.

We used LLVM-3.1 [llv11] for preprocessing, optimizing, and compiling, and GNU binutils for assembling and linking. We added a custom target machine backend for a 32-bit RISC ISA that does not support a branch delay-slot and uses a unified register file for integer and floating-point instructions. We implemented a preprocessing script to replace the `#pragma` annotations with external function calls to tag the parallel loops for analysis within LLVM, and modified the

`LoopRotation` and `LoopStrengthReduction` passes to include register and memory dependence analysis to compile XLOOPS kernels.

3.4 XLOOPS Cycle-Level Evaluation

In this section, we describe our cycle-level modeling methodology and results comparing XLOOPS to three baseline GPPs: a simple single-issue in-order processor, a moderate two-way out-of-order superscalar processor, and an aggressive four-way out-of-order superscalar processor.

3.4.1 Cycle-Level Methodology

For our cycle-level studies, we modified the GPP models within the gem5 simulation framework [BBB+11], and we implemented a model of the LPSU using PyMTL, a Python-based hardware modeling framework [LZB14]. Our changes to gem5 included: modifying the in-order and out-of-order GPP models to support AMOs and traditional execution; modifying the in-order and out-of-order GPP models to support co-simulation with the PyMTL-based LPSU model; and implementing mechanisms to migrate loop execution between the GPP and LPSU models to support adaptive execution.

We used McPAT-1.0 to estimate the energy of the in-order and out-of-order GPPs in a 45 nm process technology [LAS+09]. The energy for the lanes in the LPSU was modeled by adapting McPAT's models for simple in-order GPPs. We configured McPAT to model properly sized instruction buffers in each lane. We included an additional energy overhead of 5% to model the energy of the LMU, index queues, and arbiters based on the results from our detailed VLSI implementation (see Section 3.5). We conservatively accounted for the energy of `xi` instructions as 32-bit multiply operations, and accounted for the energy of inter-iteration register dependence communication with additional register-file read and write events. Lastly, we used the energy of an out-of-order load-store queue to conservatively model the energy of the LSQs in the LPSU.

Table 3.3 shows the configurations for the cycle-level models of the baseline GPPs and the LPSU lanes. We used three baseline GPPs: a single-issue in-order GPP (*io*), a two-way out-of-order superscalar GPP (*ooo/2*), and a four-way out-of-order superscalar GPP (*ooo/4*). These baseline designs enable us to quantitatively explore the performance and energy of XLOOPS compared to both simple, low-energy processors as well as complex, high-performance processors.

	io	ooo/2	ooo/4	Per lane
Issue Width	1	2	4	1
Phys Regs	32	64	128	24
Int ALU	1	2	4	1
AGU/Br Pred		2/1	2/1	
IQ Entries		16	32	
ROB Entries		48	96	
Ld/St Queue Entries		16/16	32/32	8/8
Inst Buffer Entries				128
Int Mul/Div Latency	4/10 cycles			
FP Mul/Div Latency	6/6 cycles			
FP Add/Sub Latency	4/4 cycles			
L1I$/L1D$/L2$/L3$	16KB/16KB/1MB/16MB			
Out-of-Order Features	Tournament Branch Pred Store-Set-Based Memory Dep Pred			

Table 3.3: Cycle-Level System Configuration
Cycle-Level System Configuration

We augmented each baseline GPP with an LPSU to create three XLOOPS configurations: *io+x*, *ooo/2+x*, and *ooo/4+x*. Each of these configurations supports traditional, specialized, and adaptive execution. Integrating the LPSU into all three baseline GPPs enabled understanding the subtle interactions between out-of-order and specialized execution (e.g., out-of-order scan phase, memory fences before and after specialized execution), and also enabled exploring adaptive execution in various contexts.

3.4.2 Traditional Execution

Table 3.2 shows the results for traditional execution of XLOOPS binaries. Each **T** column shows the speedup for each kernel compiled for the XLOOPS ISA using traditional execution on one of the GPPs relative to the kernel compiled for the general-purpose ISA executing on the same GPP. The goal for traditional execution is for this speedup to be as close to $1\times$ as possible. In other words, for traditional execution, we simply wish to reduce the performance overhead of using the XLOOPS ISA compared to the general-purpose ISA when executing on traditional general-purpose microarchitectures. We observe that the performance overhead of traditional execution is minimal and is within 5% of the general-purpose ISA for all processors with the exception of *ksack-*-om* and *rsort-ua*. The dynamic instruction counts suggest that compiler optimizations

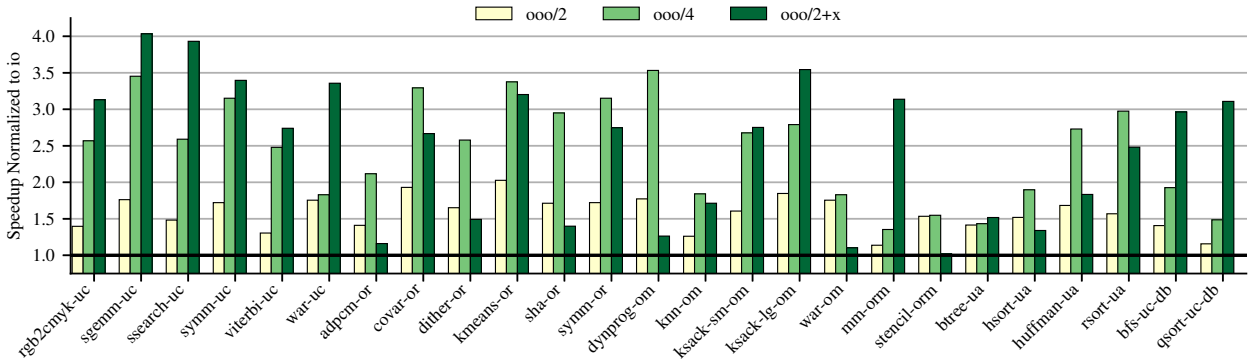


Figure 3.5: XLOOPS Cycle-Level Speedups – Each bar shows the speedup normalized to in-order (io) processor baseline kernels. ooo/2 = 2-way out-of-order processor; ooo/4 = 4-way out-of-order processor; ooo/2+x = ooo/2 augmented with LPSU.

could potentially close the gap for these kernels by reducing the number of extra instructions generated when using the XLOOPS ISA. In addition, we occasionally required additional AMOs in the XLOOPS binary compared to the general-purpose binary. Our current implementation of AMOs on the out-of-order GPPs is rather conservative, and this partly accounts for the discrepancy in traditional execution on these out-of-order GPPs (i.e., speedups <1 in **T** columns). These results are encouraging and make a case for gradual adoption of the XLOOPS abstraction in GPPs without significant overhead. In addition, efficient traditional execution will be a key enabler for adaptive execution.

3.4.3 Specialized Execution

Table 3.2 shows the results for specialized execution of XLOOPS binaries. Each **S** column shows the speedup for each kernel compiled for the XLOOPS ISA using specialized execution on a GPP+LPSU relative to the kernel compiled for the general-purpose ISA executing on the corresponding GPP. We observe that specialized execution always benefits the in-order processor. For a total of 25 application kernels, specialized execution performs better for 18 kernels compared to *ooo/2*, and performs better for 12 kernels compared to *ooo/4*.

Figure 3.5 summarizes the results comparing the baseline GPPs and the XLOOPS configurations. All speedups are normalized to each kernel compiled for the general-purpose ISA executing on *io*. The figure shows the speedup for each kernel compiled for the general-purpose ISA executing on *ooo/2* and *ooo/4*, and also shows the speedup for each kernel compiled for the XLOOPS ISA using specialized execution on *ooo/2+x*. Results for *io+x* and *ooo/4+x* are similar to *ooo/2+x*

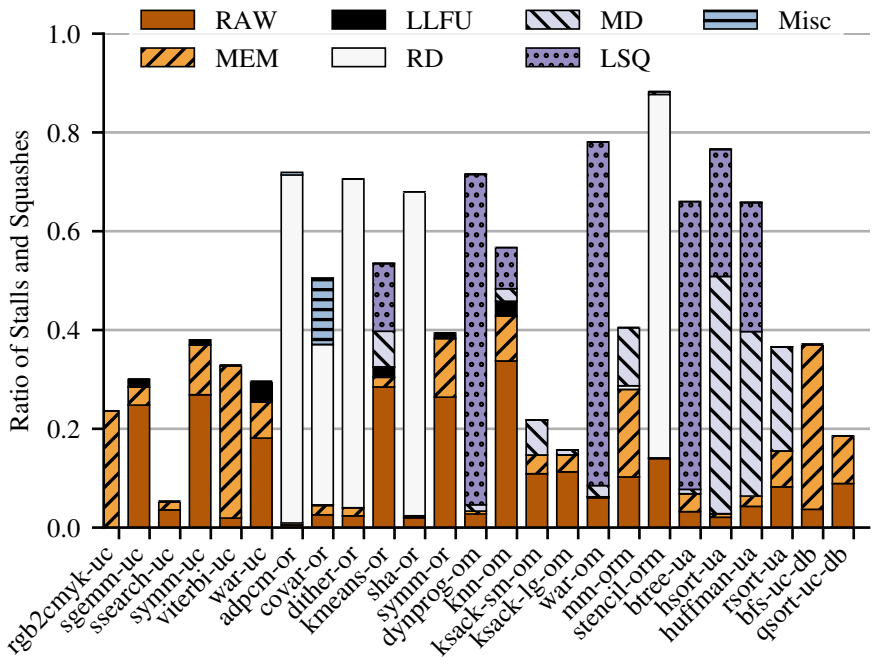


Figure 3.6: Stall and Squash Breakdown – Breakdown of average stall and squash cycles normalized to the number of cycles when the LPSU is active. RAW = read-after-write stalls; MEM = stalls due to data memory port access contention; LLFU = stalls due to LLFU access contention; RD = stalls due to inter-iteration register dependences; MD = squashes due to inter-iteration memory dependence violations; LSQ = stalls due to LSQ structural hazards; Misc = stalls due to write-after-write register-file port contention for LLFU operations and other structural hazards.

and are omitted for simplicity. Figure 3.6 shows the breakdown of stall and squash cycles for specialized execution.

Specialized execution for kernels dominated by `xloop.uc` shows speedups in the range of 2.7–4× compared to *io*. Performance of *sgemm-uc*, *war-uc*, and *symm-uc* are limited by intra-iteration RAW dependencies. *rgb2cmyk-uc* and *viterbi-uc* are constrained by stalls due to contention for the shared memory port. Figure 3.6 shows that sharing the LLFU does not significantly hurt the performance of any of the `xloop.uc` kernels. Sharing the LLFU drastically reduces the area overhead of XLOOPS (see Section 3.5). Our results show that for `xloop.uc`, specialized execution is superior to *io* and complexity-effective compared to the more complicated out-of-order GPPs.

Specialized execution for kernels dominated by `xloop.or` is usually limited by the inter-iteration critical path. For *kmeans-or* and *symm-or*, this critical path is a single instruction, resulting in improved performance compared to *ooo/2*. Most of the other `xloop.or` kernels have much longer inter-iteration register dependences. For these kernels, the out-of-order GPPs perform better than specialized execution due to their ability to exploit intra-iteration instruction-level parallelism. Future work could explore superscalar and out-of-order lane microarchitectures.

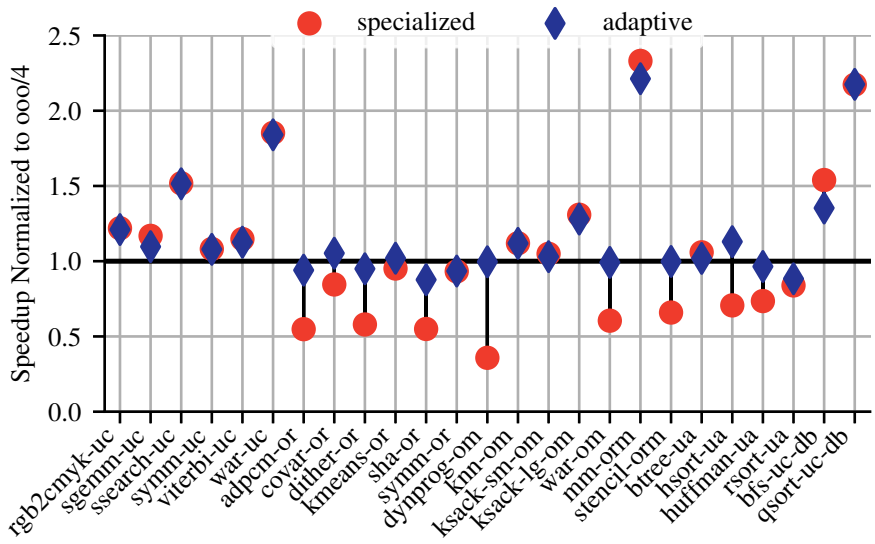


Figure 3.7: Adaptive Execution Speedups – Results for specialized execution and adaptive execution of kernels encoded with XLOOPS ISA on ooo/4+x relative to kernels encoded with general-purpose ISA on ooo/4.

Specialized execution for kernels dominated by `xloop.{om,orm,ua}` is usually limited by LSQ structural hazards and squashing speculative work due to memory dependence violations. *btree-ua*, *dynprog-om*, *war-om*, *mm-orm*, and *knn-om* are all limited by LSQ structural hazards. *hsort-ua*, *huffman-ua*, and *rsort-ua* kernels are all limited by squashing speculative work. Even with these limitations, specialized execution is still competitive with *ooo/2* on many of these kernels and even out-perform *ooo/4* on *mm-orm* and *btree-ua*. Note that the number of squashes can depend on the dataset. For example, *ksack-sm-om* has an input dataset of small weights that results in nearby iterations accessing the same memory addresses. This increases the number of memory dependence violations. *ksack-lg-om* has an input dataset of large weights that results in fewer memory dependence violations. Static compiler analysis would have difficulty predicting these data-dependent performance results.

Specialized execution for kernels dominated by `xloop.uc.db` significantly out-perform both *ooo/2* and *ooo/4*. This is because the worklist-based implementation allows the LPSU to exploit significant inter-iteration instruction- and memory-level parallelism compared to the out-of-order processors. `xloop.uc.db` kernels illustrate the potential for encoding more sophisticated inter-iteration dependence patterns in the instruction set.

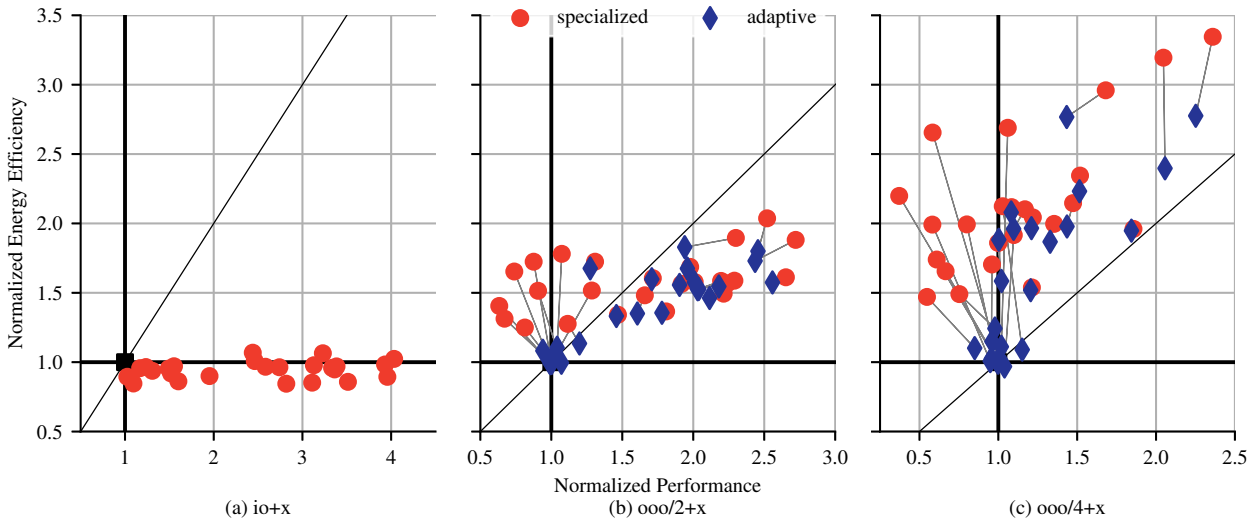


Figure 3.8: Energy Efficiency vs. Performance – Cycle-level performance and McPAT energy results of specialized and adaptive execution for (a) *io+x* normalized to *io*, (b) *ooo/2+x* normalized to *ooo/2*, (c) *ooo/4+x* normalized to *ooo/4*. Diagonal lines are iso-power contours.

3.4.4 Adaptive Execution

Adaptive execution bridges the performance gap between aggressive out-of-order GPPs and specialized execution. Figure 3.7 shows the results comparing the performance of specialized and adaptive execution on *ooo/4+x*. Based on preliminary experiments, we use 256 iterations and 2000 cycles as thresholds for the profiling phases. For kernels where traditional execution performs better than specialized execution, adaptive execution is able to automatically choose to migrate the execution from the LPSU back to the GPP. For kernels where specialized execution performs better than traditional execution, our results show that the overhead of profiling and work migration cause only minimal performance degradation. Table 3.2 also includes results for adaptive execution with *ooo/2+x*. Adaptive execution makes a compelling case for the flexibility provided by the elegant XLOOPS hardware/software abstraction.

3.4.5 Energy Efficiency vs. Performance

Figure 3.8 shows the dynamic energy efficiency and performance for specialized and adaptive execution on *io+x*, *ooo/2+x*, and *ooo/4+x*. The *io+x* results are normalized to kernels compiled for the general-purpose ISA executing on *io*, the *ooo/2+x* results are normalized to kernels compiled for the general-purpose ISA executing on *ooo/2*, and so on. The diagonal lines represent iso-power contours. Specialized execution adds minimal energy overhead and results in increased

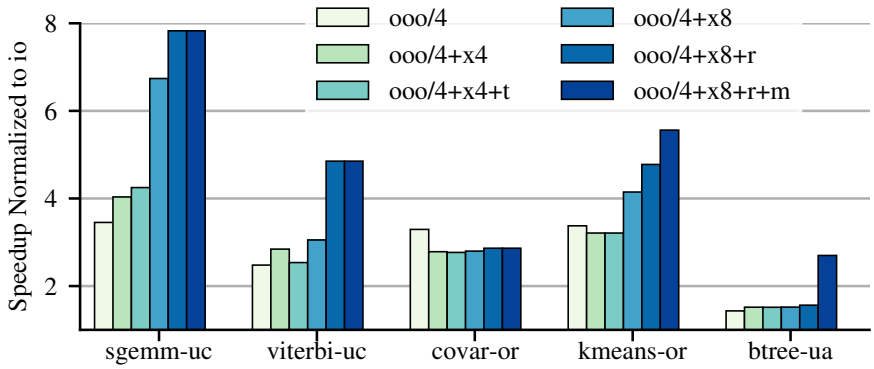


Figure 3.9: Microarchitectural Design Space Exploration Speedups – Normalized to running baseline kernel on *io*. ooo/4 = 4-way out-of-order processor; x4/x8 = LPSU design with 4 and 8 lanes, respectively; r = additional LLFUs and data memory ports; t = 2-way multithreading; m = additional LSQs.

performance compared to *io* across all applications. For *ooo/2+x* and *ooo/4+x*, specialized execution is more energy efficient across all applications. The performance trends are as explained in Sections 3.4.3 and 3.4.4. Specialized execution on *io+x* consumes more dynamic power for all applications, while specialized execution on *ooo/2+x* is power-efficient for 10 applications. Compared to *ooo/4*, specialized execution is not only more energy efficient but also consumes less power. Figure 3.8(b,c) shows that the performance benefit of adaptive execution comes at the cost of reduced energy efficiency. Overall, the results suggest that a combination of specialized and adaptive execution offers a complexity-effective design point compared to more traditional GPPs.

3.4.6 Microarchitectural Design Space Exploration

The XLOOPS hardware/software abstraction enables a rich microarchitectural design space with a variety of different potential microarchitectural optimizations. In this section, we explore some of this design space. We evaluate these features using select kernels that are representative of various inter-iteration dependence patterns.

We first consider adding limited vertical multi-threading to the lanes. Application kernels such as *sgemm-uc* that are limited by read-after-write stalls benefit from two-way multi-threading (see *ooo/4+x4+t* in Figure 3.9). However, multi-threading does not benefit all kernels, and we see reduced performance for *viterbi-uc* due to increased contention for the shared memory ports. We disable multi-threading for `xloop.{or,om,orm}` as it slows the execution of the inter-iteration critical path and/or the non-speculative lane.

Name	Loop Type	io+x	ooo/2+x	ooo/4+x
adpcm-or-opt	or	1.86	1.32	0.88
dither-or-opt	or	2.48	1.51	0.97
sha-or-opt	or	1.55	1.13	0.82
bfs-uc	uc	2.73	1.96	1.50
dither-uc	uc	2.49	1.54	1.00
kmeans-uc	uc	3.60	1.79	1.08
qsort-uc	uc	2.35	2.15	1.62
rsort-uc	uc	1.85	1.23	0.68

Table 3.4: Case Study Results – Speedups normalized to kernel compiled for general-purpose ISA.

Doubling the number of lanes to eight (*ooo/4+x8*) improves the performance for *sgemm-uc* by 68% and *kmeans-or* by 28% as neither of these applications are limited by the shared LLFU and memory port. *viterbi-uc* only sees moderate improvement as it is limited by memory port contention. Kernels limited by inter-iteration critical paths (e.g., *covar-or*) or by LSQ structural hazards (e.g., *btree-ua*) do not benefit from increased lanes.

We also consider doubling the shared LLFUs and memory ports (*ooo/4+x8+r*). This improves the performance of *viterbi-uc* by reducing memory port contention and the performance of *sgemm-uc* by reducing LLFU contention. *kmeans-or* benefits from both increased memory and LLFU resources. Finally, we explore increasing the size of the LSQs to 16+16 entries (*ooo/4+x8+r+m*) and find that the performance of *btree-ua* improves by 80%. None of the improvements in the final aggressive LPSU design reduce stalls due to inter-iteration register dependence, so we see no significant improvement in the performance of *covar-or*.

Overall, our final highly optimized LPSU design is able to significantly increase performance compared to the baseline LPSU design, but of course these optimizations also increase area and design complexity.

3.4.7 Application Case Studies

In this section we consider hand-optimized `xloop.or` kernels and manual loop transformations. Results are summarized in Table 3.4.

Hand-Optimized `xloop.or` – We observed that out-of-order GPPs perform better than the LPSU designs for several `xloop.or` kernels because of their ability to extract ILP when the LPSU

lanes stall due to inter-iteration register dependences. We compare the benefits of reducing the cross-iteration critical path for each CIR, by hand-scheduling compiler generated code, for a few select kernels. Table 3.4 shows that *adpcm-or-opt*, *dither-or-opt*, and *sha-or-opt* boost the performance of specialized execution by 50–70%. With these scheduling optimizations, specialized execution of `xloop.or` kernels is competitive with *ooo/2* and *ooo/4*. Future work can improve the XLOOPS compiler to schedule instructions more optimally.

Loop Transformations – We explored alternative loop parallelization strategies including: general parallel programming techniques such as privatize-and-reduce; using split worklists as opposed to unified worklists; and atomic data-structure updates to parallelize loops with register and memory dependences. Our results from Tables 3.2 and 3.4 suggest that transforming `xloop.or` and `xloop.om` into `xloop.uc` does not always result in improved performance. Kernels such as *bfs-uc*, *kmeans-or*, *rsort-ua*, and *qsort-uc* outperform their loop transformed counterparts. Only *dither-uc* benefits from these transformations. Because simply annotating serial versions of the kernels often performs better than code with significant transformations, XLOOPS allows ease-of-programmability without sacrificing performance.

3.5 XLOOPS VLSI Evaluation

In this section we present a register-transfer-level (RTL) model for a basic LPSU which supports `xloop.uc` instructions. We synthesize and place-and-route this implementation using a commercial ASIC CAD toolflow and present results for area, energy, and timing.

3.5.1 VLSI Methodology

Our RTL baseline design is a five-stage in-order GPP that executes 32-bit RISC instructions. The GPP uses a 16KB instruction and a 16KB data cache. We implemented a variety of detailed cycle-accurate LPSU configurations capable of supporting `xloop.uc` using parameterized Verilog RTL models to evaluate the area, energy, and timing. Note that our current RTL implementation does not support `xi` instructions. To compile the applications, we modified the `LoopStrengthReduction` pass in LLVM to disable the generation of `xi` instructions.

We target a 40 nm TSMC process using a Synopsys ASIC CAD toolflow: VCS for RTL simulation, DesignCompiler for synthesis, IC Compiler for place-and-route, and PrimeTime for power

				Percentage Area Breakdown								
Name	**CT**	**AA**	**AO**	**SP**	**I$**	**D$**	**MD**	**FP**	**IB**	**LN**	**IQ**	**MI**
scalar	1.95	0.25		8	33	37	11	10				
lpsu+i096+ln4	2.16	0.35	42	6	24	26	9	8	6	19	~0	2
lpsu+i128+ln4	2.14	0.36	44	5	23	26	9	8	6	19	~0	2
lpsu+i160+ln4	2.12	0.36	45	5	23	26	9	8	6	20	~0	2
lpsu+i192+ln4	2.20	0.37	48	5	23	25	8	8	10	18	~0	2
lpsu+i128+ln2	1.98	0.31	24	6	27	30	10	9	4	12	~0	1
lpsu+i128+ln6	2.28	0.41	65	5	20	23	8	7	8	26	1	2
lpsu+i128+ln8	2.54	0.44	77	4	19	20	8	7	10	29	1	2

Table 3.5: VLSI Area, Cycle Time Results for LPSU – CT = cycle time in nanoseconds; AA = absolute area in mm^2; AO = percent area overhead compared to scalar baseline; SP = scalar processor; I$ = instruction cache; D$ = data cache; MD = integer multiply-divide unit; FP = floating-point unit; IB = LPSU instruction buffers; LN = LPSU lanes; IQ = index queues; MI = arbiters for data-memory and LLFUs, and other miscellanous control logic; Percents rounded to nearest tens

analysis. We did not have access to a memory compiler for our target process, so we model cache tag/data SRAMs and the LPSU instruction buffer SRAM using CACTI [MBJ09]. The datasets were tailored to fit in the L1 cache.

3.5.2 VLSI Area Results

Table 3.5 presents area results based on post-place-and-route area estimates. We compare the area of the baseline GPP and the LPSU designs with four lanes by varying the capacity of instruction buffer (96–192 entries) and by varying the number of lanes (2–8) with a fixed instruction buffer size of 128 entries. Each configuration name begins with `lpsu` and a suffix with `i` to denote the instruction buffer size and `ln` to denote the number of lanes.

Total area of the primary LPSU design (lpsu+i128+ln4) is 0.36 mm^2 which is only 43% larger than the GPP (0.25 mm^2). Sharing the LLFU and memory port is a key design decision that results in minimal area overheads. Varying the instruction buffer size (96–192) with four lanes shows that area overheads range between 41–48% compared to GPP suggesting that larger instruction buffers are reasonable. Varying the number of lanes (2–8) for a fixed instruction buffer size of 128 shows that area overheads range between 24–77%. These results confirm that the area overhead of a given LPSU design increases roughly linearly with the number of lanes.

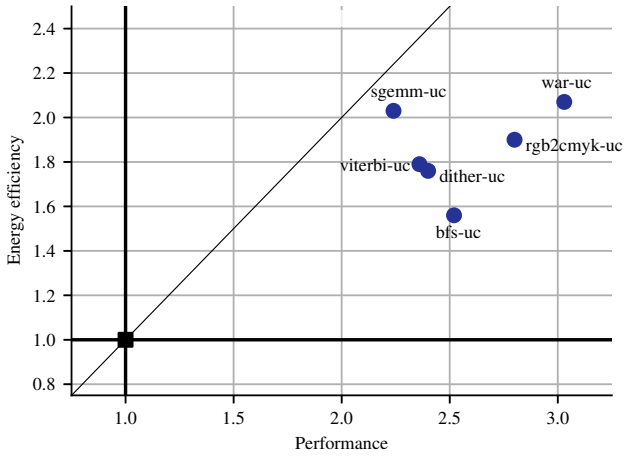


Figure 3.10: VLSI Energy Efficiency vs. Performance – Compares the in-order GPP augmented with the LPSU normalized to baseline the in-order processor. Diagonal line is the iso-power contour.

3.5.3 VLSI Energy Efficiency vs. Performance Results

Figure 3.10 compares the energy efficiency and performance of specialized execution relative to the kernels compiled for the general-purpose ISA executing on the GPP. Specialized execution improves performance by 2.4–4× (*ssearch-uc* gets a speedup of 4×; not shown in Figure 3.10). Specialized execution on the LPSU consumes more power compared to the GPP since the LPSU executes several instructions in parallel. Performance results roughly correspond to the trends seen using our cycle-level models (see Section 3.4). *sgemm-uc* has worse performance compared to the cycle-level evaluation due to an increase in dynamic instruction count caused by the lack of `xi` instructions. Our ASIC CAD toolflow reports that the energy for an access to an LPSU instruction buffer is cheaper by a factor of ten compared to an access to the instruction cache. Since most of the application time is spent in executing `xloops`, LPSU designs result in significant energy savings due to reduced instruction cache accesses. Improvements in energy efficiency are in the range of 1.6–2.1×. These results suggest that our McPAT results are relatively conservative.

3.6 XLOOPS FPGA Prototype

In this section we discuss an XLOOPS FPGA prototype for a basic LPSU which supports `xloop.uc` instructions. We present results for running simple microbenchmarks on the XLOOPS FPGA prototype.

	LUTs	Registers	BRAMs
baseline	16169	11204	95
lpsu+i256+2ln	19978	12400	95
lspu+i256+4ln	23789	13596	95
Available total	53200	106400	280

Table 3.6:
FPGA Area Results – Results for the LPSU designs implemented on the Zedboard. Based on the results above a lane requires about 1905 LUTs and 598 registers approximately. The BRAM resources remain the same as the memory system remains the same across the designs.

3.6.1 FPGA Methodology

We used the Zedboard, which includes a Zynq-7000 All Programmable SoC [Zyn18], as the platform for our XLOOPS FPGA prototype. The Zynq-7000 includes two ARM Cortex-A9 cores tightly integrated with a high-performance reconfigurable fabric. One of the ARM cores served as the host processor for the prototype meaning it ran a full Linux operating system and handled all I/O for the prototype. In particular, we used the Xillinux distribution [Xil18] which is an Ubuntu-based Linux distribution that includes a custom Xillybus FPGA development kit. The Xillybus FPGA development kit provides host drivers that expose the FPGA device as device files for the software development. The Xillybus hardware IP communicates with the host processor using the AXI protocol and provides a FIFO interface to the FPGA design. The Xillybus FPGA development kit greatly simplified the prototyping effort.

Our design includes a five-stage in-order GPP that executes 32-bit RISC instructions. The GPP uses a 16-KB two-way set-associative instruction and data cache. The instruction and data caches are backed by a 32-KB main memory. The 32-KB main memory was selected based on the size of the microbenchmarks and the datasets that were used to evaluate the prototype. The LPSU design is parameterized by the number of lanes. Each lane in the LPSU includes a 256B L0 buffer and a two-element index queue. Note that our current FPGA implementation does not support `xi` instructions. The XLOOPS FPGA design is currently programmed via assembly programming. The host processor is responsible for loading the programming into the main memory, signaling the XLOOPS GPP to begin execution, and monitoring a done signal asserted by the design to indicate the end of a program. The host program reads and writes named control registers in the design that toggle the collection of statistics.

3.6.2 FPGA Area Results

Table 3.6 presents area results for the implemented design based on the report generated by the Xilinx-ISE 14.2 design tools. We report the FPGA device utilization broken down by the total lookup-tables (LUTs), registers, and block-RAM memory usage. The table presents the device utilization for a baseline design which only has the GPP and the results for implemented the LSPU with two lanes and four lanes. Each configuration name begins with `lpsu` and a suffix with `i` to denote the instruction buffer size and `ln` to denote the number of lanes.

The design with two lanes (lpsu+i256+ln2) consumes approximately 24% overhead and the design with four lanes consumes approximately 50% overhead compared to just the general-purpose processor and L1 caches alone. This is higher than our VLSI results since the FPGA prototype does not include a floating-point unit (the FPU is part of the baseline and is shared with the LPSU in the VLSI results). The design with four lanes uses 96% of the available slices, so mapping XLOOPS accelerators with more lanes would require moving to a larger FPGA. Static timing analysis suggests the designs can run at 33 MHz, although we did our actual experiments running at 25 MHz.

3.6.3 FPGA Performance Results

We discuss the results for the primary LPSU design with four lanes (lpsu+i256+ln4) compared to the baseline design executing on the GPP only. We implemented and evaluated three microbenchmarks. *vvadd* computes element-wise vector addition of two input arrays and stores the result in a destination array. The length of the vector is 100. *binsearch* performs a binary search in a dictionary of key-value pairs. The dictionary size and the search sizes were 50 each. *mfilter* performs a masked convolution with a five-element kernel across a 50×50 image input.

For *vvadd* we observed a speedup of 3.2×, for the *binsearch* kernel we observed a speedup of 5.7×, and for the *mfilter* kernel we observed a speedup of 2.5×. Note for the *binsearch* kernel, the parallel execution is super-linear as the computation can be overlapped with cache misses. The kernel has a lot of cold misses for the scalar execution whereas the parallel execution results in hiding cold misses in the cache.

3.7 Related Work

Most of the previous work on loop-level specialization including data parallel accelerators (DPAs), speculative parallelization, hardware task scheduling, transactional memory, and accelerators, tightly couple the abstraction and microarchitecture. XLOOPS is an elegant approach that unifies many of these proposals with a novel abstraction that can be mapped on to traditional, specialized, and adaptive microarchitectures.

ASIPs integrate specialized circuits into a traditional processor pipeline which benefits a specific domain of applications and are limited in generality [CFHZ04]. Architectures such as CCA [CKP+04], DySER [GHS11], and BERET [GFA+11] provide reconfigurable datapaths to accelerate critical subgraphs of computation within a loop iteration. Our current work focuses more on inter-iteration loop dependence patterns.

`xloop.uc` – Many commercial DSPs [CWS+14, ti08] support zero-overhead loops in the form of a special loop or repeat instruction. These architectures allow the execution of loops with no ordering constraints and require no hardware support for control speculation. DPAs are examples of architectures that exploit inter-iteration data-level parallelism. Streaming SIMD extensions, Advanced Vector Extensions (AVX), and vector ISA extensions [WAK+96, KP03, EVS98] amortize the overheads of instruction processing and increase performance by executing parallel operations. These architectures suffer when executing code with intra-iteration control-flow, loop-carried register-dependences, and divergent memory accesses [GNS13]. Furthermore, they rely heavily on vectorizing compilers which is an active area of research. Mainstream GPUs [LNOM08, WKP11] and Maven [LAB+11] alleviate the problems of traditional vector processors but require more radical changes across ISA, compiler and microarchitecture compared to XLOOPS.

`xloop.ua` – Transactional memory (TM) systems [HLR10] coordinate the execution of parallel computations by committing non-conflicting memory updates. In [ZMLM08], authors modify traditional architectures to include a hardware TM system and expose transactions to software through instruction-set extensions to exploit loop-level parallelism. Our XLOOPS abstraction allows for a variety of microarchitectures that can take advantage of the `xloop.ua` data-dependence pattern.

`xloop.or` – Multiscalar [SBV95], vector-like proposals [Jes01, KBH+04], and others [KT99, ZMLM08] propose register bypass networks similar to the CIBs to handle inter-iteration register

dependences. HELIX-RC [CBK+14] proposes a ring-cache architecture to communicate register dependences. XLOOPS is potentially more elegant as it avoids requiring ISA extensions to specify the dependence communication unlike previous proposals.

`xloop.om` – Multiscalar and TLS proposals [SCZM00, KT99] are speculative parallelization techniques that provide hardware memory-dependence speculation to exploit loop-level parallelism. XLOOPS proposes per-lane LSQs and a store broadcast network to support memory-dependence speculation in hardware. Previous speculative parallelization techniques show promise but demand dramatic changes in the microarchitecture, compiler, and/or ISA. HELIX-RC [CBK+14] takes an alternative approach of decoupling memory dependence communication without employing speculation but relies on an aggressive parallelizing compiler. The XLOOPS ISA could be extended to include instructions for lane synchronization to benefit compiler optimizations as in HELIX-RC.

`xloop.*.db` – Carbon [KHN07] and Asynchronous Data Messages (ADM) [SYK10] are two proposals that exploit fine-grain loop-level parallelism through hardware-only and hybrid hardware-software work distribution queues. The XLOOPS dynamic-bound construct is similar in spirit by allowing mapping loops with dynamic work generation.

3.8 Conclusions

In this chapter, we have introduced XLOOPS, a new hardware specialization approach for exploiting inter-iteration loop dependence patterns. The XLOOPS proposal enables lane-based BSAs to execute challenging loops with complex inter-iteration dependences which cannot be mapped to traditional accelerators such as packed-SIMD and vector units. The XLOOPS instruction set provides an elegant hardware/software abstraction that serves as an effective compiler target and enables a variety of microarchitectures supporting traditional, specialized, and adaptive execution. We have used a vertically integrated evaluation methodology spanning applications, compilers, cycle-level modeling, RTL modeling, and VLSI implementation to make a compelling case for augmenting both in-order and out-of-order general-purpose processors with a loop-pattern specialization unit. We also prototyped a simple LPSU design using the Xillinx Zedboard FPGA platform.

XLOOPS makes a case for single-ISA heterogeneous architectures that transparently integrate traditional GPPs and specialized loop accelerators. Single-ISA heterogeneous architectures are not new, but past work in academia and industry has focused on composing different kinds of traditional processor microarchitectures that all implement the same instruction set, e.g., composing many simple in-order processors with a few complex out-of-order processors. XLOOPS demonstrates that it is also possible to integrate more exotic specialized loop accelerators, while still supporting a common performance-portable instruction set. Using the hardware instruction set as the portable abstraction layer (as opposed to related work on virtual instruction sets or common programming APIs) enables dynamic, fine-grain migration between different microarchitectures with minimal overhead.

The XLOOPS instruction set allows software to communicate significant compile-time information about inter-loop dependence patterns to the hardware while liberating microarchitects to explore a diverse range of accelerators. The XLOOPS instruction set provides significant freedom to microarchitects. GPPs can potentially better exploit the additional semantic information encoded in an XLOOPS binary during traditional execution. Examples include adding loop buffers managed by the `xloop` instructions, using the `xloop` instructions as hints to turn off power hungry components in out-of-order processors, or using the `xloop` instructions to guide hardware prefetching. The proposed XLOOPS microarchitecture for specialized execution is rather simple with in-order lanes, but the XLOOPS instruction set facilitates a much more diverse range of potential specialized engines including aggressive out-of-order lanes, heterogeneous LPSUs comprised of both simple and complex lanes, and clustered groups of lanes to directly exploit nested loop-level parallelism.

The XLOOPS microarchitecture supports the execution of a single binary in multiple ways: (1) traditional execution with minimal performance impact, (2) specialized execution to improve performance and/or energy efficiency, and (3) adaptive execution that can seamlessly migrate loops between traditional and specialized execution to dynamically trade-off performance vs. energy efficiency. We see this as a new execution paradigm for future single-ISA heterogeneous architectures and we hope that other researchers will adopt this execution paradigm in their work. Furthermore, the specific adaptive execution mechanism described in this chapter is relatively simplistic. There are interesting opportunities to explore a spectrum of adaptive execution mechanisms that maximize energy efficiency, distribute loops across multiple different specialized loop accelerators, use

more sophisticated performance prediction for loop migration, and dynamically determine which loop to parallelize in a loop nest.

CHAPTER 4
SSAS: LANE-BASED BSAS FOR FORK-JOIN-CENTRIC PARALLELIZATION AND SCHEDULING STRATEGIES

Loop-centric parallel programs are popular since many computations can be expressed over simple partitions of input data using loops. However, there are many algorithms that are more naturally expressed as recursive, divide-and-conquer task-parallel computations. In this chapter, I propose *smart sharing architectures* (SSAs), a new approach to building lane-based BSAs that can efficiently support fork-join-centric parallelization and scheduling strategies. Prior lane-based BSAs such as packed-SIMD and vector units as well as XLOOPs cannot elegantly handle fork-join-centric parallel programs. Currently, CMP architectures are the best-suited platforms to execute fork-join-centric parallel programs. Section 4.1 briefly discusses some of the benefits of fork-join-centric parallel programs and motivates the case for lane-based BSAs as target architectures for these programs. Section 4.2 reviews the work-stealing scheduling strategy used to execute fork-join-centric parallel programs. Section 4.3 discusses the design space for SSAs. The design of SSAs is based on two key ideas: (a) there is a considerable amount *instruction redundancy* in fork-join-centric parallel programs; and (b) exploiting instruction redundancy using complexity-effective *smart sharing mechanisms* allows SSAs to reduce costs, maximize efficiency, and improve performance of fork-join-centric parallel programs. Section 4.4 presents the evaluation methodology, application kernels used, and the results for evaluating SSA designs. Section 4.5 discusses related work, and the chapter concludes in Section 4.6.

4.1 Introduction

The MIT Cilk project [BJK+95, BL99, FLR98] popularized the use of fork-join-centric parallelization and scheduling strategies to efficiently implement recursive, divide-and-conquer task parallel computations. Cilk has inspired many task parallel programming frameworks that include Intel's C++ Threading Building Blocks (TBB) [Rei07, int15], Intel's Cilk Plus [Lei09, int13], Microsoft's .NET Task Parallel Library [LSB09, CJMT10], Java's Fork/Join Framework [Lea00, jav15], and OpenMP [ope13]. The fork-join-centric model extends the control-flow in serial programs by including *fork* and *join* primitives. The fork primitive allows a parent task to spawn a child task, which can potentially execute in parallel, and the join primitive expresses a synchroniza-

tion point for spawned child tasks upon return. Software runtime frameworks provide abstractions for fork/join primitives, and typically implement a *work-stealing* scheduling strategy. Section 4.2 discusses one such runtime inspired by the Intel TBB framework.

There are several benefits provided by fork-join-centric parallelization and scheduling strategies. The fork-join-centric model provides an important property of *serial elision*, i.e., serial execution of a fork-join-centric parallel program can be achieved by simply removing the fork and join primitives. Serial elision aids in improving the productivity of a programmer. A programmer can first focus on developing a serial program that can be easily parallelized later by adding fork and join primitives. Additionally, fork-join-centric parallel programming frameworks such as Intel TBB and Intel Cilk Plus provide data structures that can handle accesses to shared mutable objects. These frameworks also include data-race detector tools that further simplify the task of parallel programming. The task-based abstraction used in fork-join-centric parallel programming frameworks is *portable* across operating systems, compilers, and hardware platforms. The mapping of tasks to hardware threads is managed by the underlying work-stealing scheduling runtime thus, making it easy to maintain fork-join-centric parallel programs. Lastly, fork-join-centric parallel programs naturally express divide-and-conquer algorithms with *good cache behavior*. The recursive division of a large problem into smaller sub-problems using fork primitives reduces the size of the working-set of a leaf task to fit in the cache. Work-stealing runtimes use locality-aware heuristics to efficiently combine the results of each sub-problem using join primitives. The field of cache-oblivious algorithms [FLPR12] focuses on formulating divide-and-conquer algorithms expressed as fork-join-centric parallel programs to exploit good cache behavior.

Currently, CMP architectures are the most flexible and best-suited hardware platforms to execute fork-join-centric parallel programs. Tasks in fork-join-centric parallel programs fundamentally are based on more that loop iterations which makes it difficult to use packed-SIMD and vector units. Auto-vectorization of a work-stealing runtime implementation is not possible as each thread maintains its own stack and local queues, and each thread also requires complex synchronization mechanisms to manage task distribution for efficient load-balancing. The XLOOPS proposal can encode dynamic work using the `xloop.*.db` instruction but requires a programmer to express an algorithm using loop constructs only. Hence, fork-join-centric parallel programs have an affinity towards multi-core platforms. With the growing popularity of task-based parallel programming frameworks, architects have proposed a variety of specialized CMP-based platforms.

			Percentage Area Breakdown				
Name	**Tech node.**	**AA**	**SP**	**I$**	**D$**	**MD**	**FP**
RV64G	16nm	0.10	10	30	36		22
XLOOPS	40nm	0.25	8	33	37	11	10

Table 4.1: Area Breakdown for RV64G and XLOOPS scalar Core – The area numbers show that the scalar in-order core for both the 64-bit RV64G processor and the 32-bit RISC-based XLOOPS processor are dominated by instruction and data caches as well as as the long-latency funcitonal units. Note, the RV64G core uses an iterative integer multiply-divide unit which is part of the scalar pipeline. AA = absolute area in mm^2; SP = scalar processor; I$ = instruction cache; D$ = data cache; MD = integer multiply-divide unit; FP = floating-point unit;

In Carbon [KHN07], the authors propose to implement a work-stealing scheduler using a set of hardware task queues and use specialized communication networks to efficiently distribute tasks amongst the cores. ADM [SYK10] uses a software/hardware co-design approach, where cores communicate tasks by sending direct messages over a specialized network while representing task queues in software. Minnow [ZMTC18] is a recent technique that augments each CMP core with a Minnow engine, a programmable accelerator that offloads worklist scheduling and performs worklist-directed prefetching. Carbon, ADM, and Minnow seek to scale CMP platforms for fine-grained task parallelism by focusing on improving the overheads of work-stealing runtimes.

Scaling cores in a CMP platform is fundamentally challenged by the power constraints [EBA$^+$11] and area costs. CMP architectures that target large amounts of fine-grained task parallelism typically use simple in-order cores to scale core counts to match the available parallelism. For example, in ADM the authors envision a hardware platform with 64–128 cores and a more recent proposal, SWARM [JSY$^+$15], envisions 128–256 cores. We make an important observation that the area costs of a simple in-order core are dominated by expensive resources such as instruction and data caches, integer multiply-divide units, and floating-point units. Table 4.1 shows the component-wise area breakdown of an in-order 64-bit RISC-V core (RV64G) based on the open-source Berkeley Rocket-Chip SoC generator and a simpler 32-bit RISC-based in-order processor as implemented in the XLOOPS project. The RV64G core is implemented using a 16nm TSMC process, and is capable of booting Linux. Note, the RV64G processor uses a simple iterative integer multiply-divide unit that is included as part of the scalar in-order core. The XLOOPS core is implemented using a 40nm TSMC process and is a simpler core that executes in bare-metal mode. The VLSI results show that the area of the scalar in-order core is dominated by other hardware resources. Given

a simple scalar in-order processor, we observe that resources such as instruction and data caches, as well as the long latency functional units, are often under-utilized. Sharing these resources is an attractive solution to improve efficiency, drive down hardware costs, and reduce leakage. However, sharing resources fundamentally trades off performance for a reduction in area costs.

Fork-join-centric parallel programs execute parallel tasks that are part of the *same* program. Executing recursive task-parallel programs often involves threads which execute instructions that match exactly or map to the same cache line at any given time. These instructions are fetched redundantly from the instruction cache and contribute to energy overheads. We term the property of threads executing the same instruction or instructions that map to the same cache-line as *instruction redundancy*. Exploiting instruction redundancy in fork-join-centric parallel programs thus far remains an unexplored opportunity. I propose *smart sharing architectures* (SSAs), a new approach to building lane-based BSAs that exploit instruction redundancy to improve efficiency and performance of fork-join-centric parallel programs. SSAs is a CMP-based approach where the GPPs are augmented with a lane-based BSA that is composed of simple *conjoined-lanes*. Conjoined-lanes are lightweight in-order pipelines that execute user-threads and are managed by the GPP. The execution model for SSAs is discussed in Section 4.2. The key research question for SSAs is to explore complexity-effective sharing mechanisms that maximize efficiency and mitigate performance loss. Figure 4.1 shows an example SSA microarchitecture. As in XLOOPS, the GPP can be either be a simple in-order or complex out-of-order core. GPP and the lanes share expensive resources, which include the instruction and data caches as well as the long latency functional units (LLFUs). Accessed to the shared resources is governed by *smart sharing mechanisms*.

4.2 SSA Runtimes

Work-stealing is a dynamic load-balancing scheduling strategy for fork-join-centric parallel programs. A work-stealing runtime consists of multiple *worker* threads. Each worker maintains a *task queue* data structure that stores forked tasks. A *worker-loop* is a program executed by each worker thread. When a worker finishes the execution of a task, it selects the next available task from its task queue and executes the task. If the task queue is empty, the worker *steals* a task from the task queue of another worker. The stealing worker is called the *thief*, and the worker whose task is stolen is a *victim*. Typically, a worker selects a task from its local task queue in

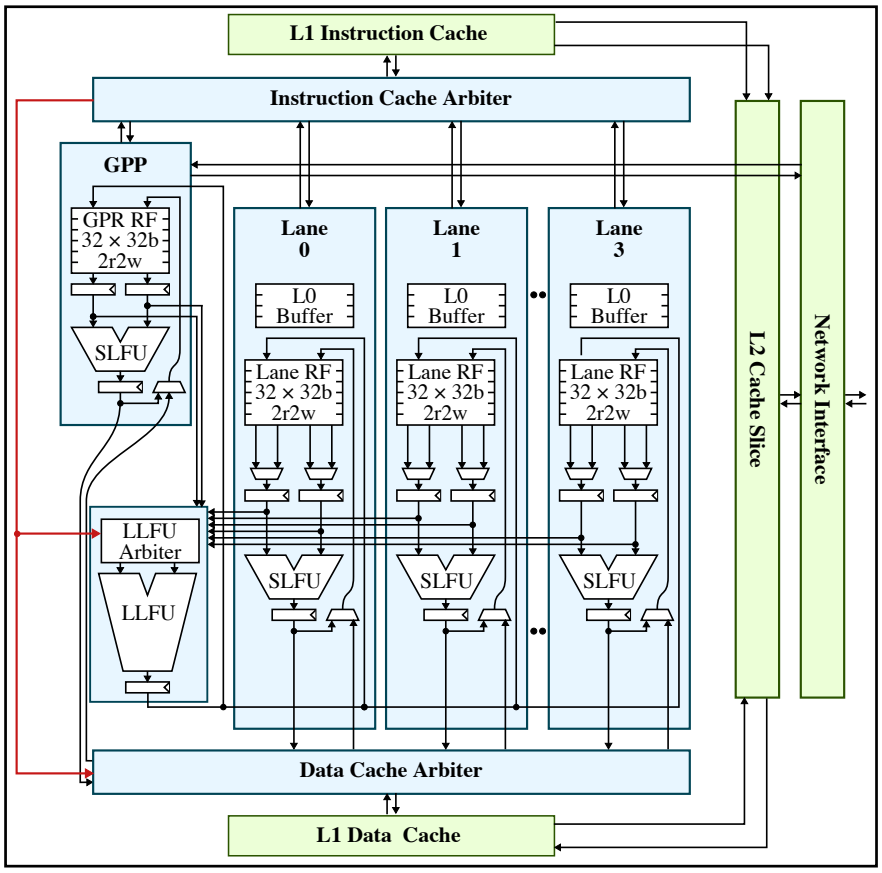


Figure 4.1: SSA Microarchitecture Example – An example SSA tile organization augments the GPP with four conjoined-lanes. GPP and the conjoined-lanes use arbiters to share the L1 instruction- and data-cache as well as the LLFUs. Each lane includes an L0 instruction line buffer that eases the pressure on instruction cache bandwidth. The L1 instruction arbiter communicates with the data and LLFU arbiters to coordinate the use of resources based on smart-sharing policies, highlighted in red. GPP = general-purpose processor (either in-order or out-of-order); GPR = GPP regs; RF = regfile; SLFU = short-latency functional unit; LLFU = long-latency functional unit implements the integer multiply-divide and floating point arithmetic.

LIFO order and steals a task from a victim in FIFO order to preserve locality [FLR98]. The policies for task-selection and victim-selection can vary across different implementations of work-stealing. Figure 4.2 shows the pseudo-code for a worker-loop implementation. The workers enter the worker-loop when the runtime program is initialized. The workers exit the worker-loop when the thread that executed the root task sets a global done flag indicating that all of the forked tasks have finished execution.

```
1  void worker_loop() {
2    tid = get_thread_id();
3    while( !done ) {
4      task = task_queues[tid].pop();
5      if ( task != NULL ) {
6        execute_task( task );
7      } else {
8        victim_id = select_victim();
9        task = task_queues[victim_id].steal();
10       if ( task != NULL ) {
11         execute_task( task );
12       }
13     }
14   }
15 }
16
```

Figure 4.2: Worker Loop Pseudo-code Implementation – Each worker executes the worker loop until a global *done* flag is set by the worker which executed the root task. The policy for task-selection is encapsulated by the `TaskQueue.pop()` and the `TaskQueue.steal()` functions. The `select_victim()` function can be random or occupancy based.

There are two key design choices for implementing fork-join parallelism based on what happens at a fork point and what happens at a join point [Rob14]. In *child stealing*, on encountering a fork point a parent task pushes the child task into its local task queue and continues execution. In *continuation stealing*, the parent task pushes the continuation into its local task queue and executes the newly forked child task first. TBB and PPL are examples of runtimes that use child stealing and Cilk is an example of a runtime that uses continuation stealing. The choice of child- or continuation stealing impacts the asymptotic space bounds of task queue storage. The advantage of continuation stealing is that the space requirement is guaranteed to grow within a constant factor of the number of processors [BJK+95]. Whereas in child stealing, if not careful, the space requirements could be unbounded. However, continuation stealing, as implemented in Cilk, requires language extensions and extensive compiler support. Upon encountering a join point there are two choices available: *stalling strategy* and *greedy strategy*. In the stalling strategy, a thread executing the parent task stalls for the completion of child tasks whereas, in the greedy strategy, no thread is idle waiting for other tasks. In the greedy strategy, a thread that executes the last child task to reach the join point must execute the computation after the join point. TBB and PPL are examples of runtimes that implement the stalling strategy, and Cilk implements the greedy strategy. The greedy strategy provides an asymptotic bound on performance known as "Brent's Lemma" which

is useful in the analysis of parallel programs. Stalling schedulers prevent the application of this theory. However, stalling schedulers simplify assumptions of thread identity. In greedy schedulers that use continuation stealing, a function can return on a different thread which could complicate programs that use thread local storage and mutexes.

The work-stealing runtime system implemented in this thesis is inspired by the TBB framework. The work-stealing runtime (WSRT) employs child stealing with Chase-Lev task queues [CL05] and uses an occupancy-based victim selection [CM08]. We also implement a baseline loop-centric runtime that is based on the single-program-multiple-data (SPMD) programming model. The baseline SPMD runtime uses a static-scheduling strategy that partitions an input data set such that each thread is roughly distributed an equal number of loop iterations. The SPMD runtime makes a good baseline, since most related work exploits instruction redundancy within such runtimes (see Section 4.5).

In the SSA execution model, the GPP is in charge of coordinating the execution of a fork-join-centric parallel program. The GPP executes the main program and the conjoined-lanes execute the *worker loop*. The GPP spawns a root task onto a lane by writing to a lane's local task queue. The parallel execution of the fork-join-centric parallel program terminates when the lane which executed the root task sets the global done flag. The GPP can remain idle during the parallel execution or participate by executing the worker loop. We consider the GPP to remain idle in this work to investigate the performance of executing fork-join-centric parallel programs on conjoined-lanes. Conjoined-lanes are assumed to be clock-gated in serial sections of the code when the GPP is executing.

The worker loop in Figure 4.2 explains why popular loop-centric accelerators are awkward targets for fork-join-centric parallel programs. Most loop-centric accelerators focus on executing simple, dense data-parallel loops. It is not apparent that the worker-loop is such a loop. The presence of task queues and complex control-flows challenge an auto-vectorizing compiler and hence, limit the applicability of packed-SIMD and vector units. GPGPUs are more flexible compared to packed-SIMD and vector units. Modern GPGPUs support complex unstructured control flows and do not rely on an auto-vectorizing compiler. However, the memory system of GPGPUs is not optimized for synchronization of threads within a warp. Mapping a fork-join-centric parallel program either requires cbit hanges to the algorithm or motivates changes to the GPGPU hardware [GSO12, KB14].

Carefully inspecting the worker-loop provides clues to sources of instruction redundancy in the WSRT. Conjoined-lanes either execute the scheduling logic, which involves inspecting task queues, or execute parallel tasks. Tasks in recursive divide-and-conquer parallel programs are similar as the premise of this style of programming is to divide a large problem into smaller independent subproblems that are identical but work on different input data set partitions. The key to exploiting the latent instruction redundancy in fork-join-centric parallel programs is to "line-up" the executions across independent lanes. SSAs employ smart sharing mechanisms to exploit instruction redundancy dynamically during the course of execution, thereby, circumventing the limitations of loop-centric accelerators.

4.3 SSA Microarchitectures

SSA microarchitectures include a rich design space of conjoined-lanes based on the degree of shared hardware resources. Sharing no resources and sharing all resources are extreme points in the design space. It is important to fundamentally understand the design space between these extreme points as it is unlikely for either one of the designs to be optimal across all workloads. The implications of cost reductions and complexity of designs cannot be identified without understanding the impact of *smart sharing mechanisms*.

Figure 4.3 shows the design space we consider for conjoined-lane architectures using four lanes. Lanes in SSAs are simple in-order pipelines that share resources. We abstract resources that compose a lane pipeline into the following: (i) instruction cache port (I), shown in blue; (ii) architectural state that represents the user-thread context (P), shown in red; (iii) the front-end logic (F) that includes the fetch, decode, and issue logic, shown in green; (iv) the data cache port and the LLFUs (L), shown in grey; and (v) the back-end logic (B) which includes the ALU and writeback logic, shown in yellow. The hardware resources that can be shared include the instruction cache port (I), the front-end logic (F), and the data cache port/LLFUs (L). The topmost design point allocates exclusive resources to each lane, and we refer to this design point as the *no sharing* design. The *no sharing* design is used as the baseline, since we expect this design to perform the best due to the lack of any resource contention. The bottom design point shares all resources by multiplexing in time, and we refer to this design point as the *sharing all* design. The *sharing all* design point greatly reduces area costs while sacrificing performance. Moving from top to bottom

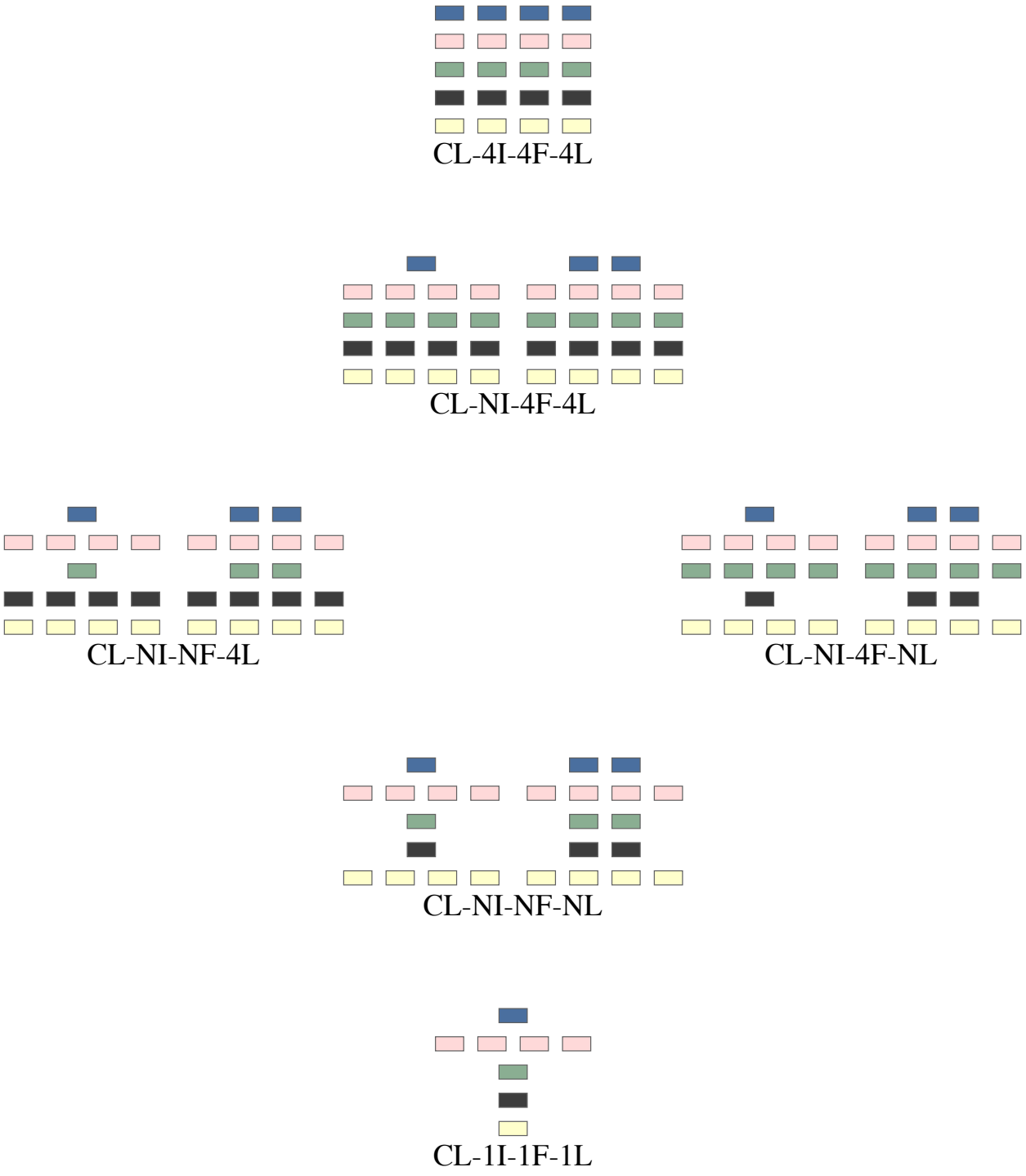


Figure 4.3: SSA Design Space – Abstract microarchitecture models showing various organizations for four conjoined-lanes. The topmost represents no sharing in space and the bottom most shows sharing all resources in time. Area costs reduce moving from top to bottom. The resources that can be shared include instruction cache ports (I) shown in blue, front-end logic (F) shown in green, and the data cache port/LLFU (L) shown in grey. The thread contexts are shown in red and the back-end logic is shown in yellow. Notation *CL-NI-NF-NL* represents a static organization of resources. The value *N* for a statically shared resource can be 1 or 2.

in Figure 4.3 reduces area costs and increases degree of hardware sharing. We use the notation *CL-NI-NF-NL* to specify a static organization of resources. For example, *CL-4I-4F-4L* represents the *no sharing* design point whereas the *CL-1I-1F-1L* represents the *sharing all* design point. The value for a shared static resource denoted by *N* can be 1 or 2.

The key to sharing resources in SSAs is to employ complexity-effective *smart sharing mechanisms* that mitigate the loss in performance while maximizing efficiency. The smart sharing mechanisms we explore include: (i) instruction coalescing; (ii) soft-barrier hints; (iii) prioritized thread-selection; and (iv) lockstep sharing. The following sections describe the static microarchitectures considered for SSAs and introduce the sharing mechanisms as appropriate.

4.3.1 Sharing the Instruction Port Only

The lanes in the *CL-NI-4F-4L* design share the instruction memory port using round-robin arbitration. We call this design point as a *sharing imem only* design. Instruction caches in simple in-order cores can occupy around 30% of the tile area (see Table 4.1). Sharing the instruction cache is an attractive solution to reduce area costs. Since, the lanes execute the worker loop and user tasks that are derived from the same program, sharing the instruction cache would not result in thrashing. This design choice is scalable as the lanes have exclusive resources otherwise. The potential drawback to sharing the instruction port includes reduced capacity, increased instruction access latency due to wide multiplexors and wires in the instruction port arbiters, and loss in performance due to reduced bandwidth. An obvious solution is to simply duplicate the instruction port (*N=2*). SSAs implement additional smart sharing mechanisms to address the bandwidth contention while minimally trading off area compared to duplicating an instruction port.

Adding an L0 line buffer to each lane is the first strategy. The L0 line buffers are cheap and store a single cache line worth of instructions to feed the simple single-issue in-order lane pipeline. The design rationale for L0 buffers is based on the observation that the instruction stream for executing the worker loop involves operations on the task queue data structure which is mostly straight line code and that most parallel programs have some spatial locality in the instruction stream. The L0 line buffers aid in additional efficiency improvements as accessing an instruction out of an L0 line buffer is cheaper in energy compared to accessing from an instruction cache.

Instruction coalescing is a smart sharing mechanism which is based on the observation that tasks in a recursive fork-join-centric parallel program are similar and that lanes mostly execute

instructions that can be coalesced into a single cache-line access. The idea is to combine requests from two or more lanes if they are fetching a word in the same cache line. Coalescing is a common mechanism employed in GPGPUs to improve memory bandwidth. The probability of instruction coalescing is high when the control-flow is regular. Instruction coalescing not only improves the performance but can also help in reducing the energy by avoiding expensive tag checks that are redundant.

Soft-barrier hint instructions are one way to indicate opportunities for control-flow convergence in work-stealing runtimes. Lanes can execute tasks out of local task queues or can execute tasks by stealing from victims. Synchronizing just before executing a task improves the potential for instruction coalescing. Lines 6 and 11 in Figure 4.2 are potential points in the worker loop where soft-barrier hints can be added. We propose soft-barrier hint instructions that have the semantics of a *NOP* instruction. A microarchitectural implementation for soft-barrier hints would be to pause a lane execution by holding back the instruction response in the arbiters for some number of cycles. An arbiter will eventually return a response to a paused lane if a lane reaches a maximum time-out limit or if the arbiter observes that all lanes have executed a soft-barrier hint instruction. When a lane reaches a maximum time-out, the arbiter can pair up any other lane waiting to find a "matching" peer. The soft-barrier hints minimally instrument the runtime with no changes to the application and are not required for correctness.

4.3.2 Sharing the Instruction Port and the Front-end Only

The lanes in the *CL-NI-NF-4L* design share the instruction memory port and front-end resources. We call this design point as a *sharing imem+fe only* design. In these microarchitectures, a single instruction that is fetched and decoded can be issued to the back-ends of multiple lanes as long as thread contexts for each lane are executing the same instruction. Sharing the front-end in addition to the instruction port further improves efficiency as long as execution of lanes is converged.

Statically sharing the front-end is inefficient in scenarios where lanes execute different control-flows. In the extreme case where each lane is independently executing a unique instruction stream, the front-end logic is essentially time multiplexed which results in lower utilization of lanes. As in the case of *sharing imem only*, a simple policy for fairness under divergence would be to employ a round-robin arbitration mechanism. Soft-barrier hints aid in synchronizing threads just as before

but we observe that if the control-flow within the task execution diverges then the lanes remain diverged. The hints inserted in the runtime are essentially agnostic of the control-flow within the user-defined tasks.

We propose a **prioritized thread-selection** mechanism based on the minimum-pc heuristic to improve probabilities of converged execution. The intuition behind prioritizing the selection of a thread with minimum-PC is based on the observation that compilers layout basic blocks in memory in an order that preserves *dominance relationships*. A basic block B post-dominates a basic block A if all paths to the exit node of a control-flow graph starting at basic block A must go through basic block B. Reconvergence points are usually located at the immediate post dominator frontiers which are laid out at high-address locations by the compiler. Hence, prioritizing a thread with the minimum-PC allows the control-flow to reach a potential reconvergence point. The use of the minimum-PC heuristic is inspired by SIMT architectures [Col11]. However, a key difference in the mechanism we employ in SSAs is to use a *hybrid minimum-PC and round-robin arbitration* heuristic. The minimum-PC heuristic aids reconvergence and the round-robin heuristic guarantees forward-progress. Using only the minimum-PC heuristic can lead to deadlocks in SSAs as control-flows can be completely unstructured given the recursive calls into the worker loop.

4.3.3 Sharing the Instruction Port and the LLFUs Only

The lanes in the *CL-4I-NF-4L* design share the instruction memory port and the LLFU resources. We call this design point as a *sharing imem+llfu only* design. The LLFU resources in SSAs include the data memory port, integer multiply-divide units, and floating-point units. The integer multiply-divide and floating-point units can account for approximately 20% of area and the data cache accounts for approximately 36% area of an in-order tile. Sharing these resources can significantly reduce area costs. We employ the same strategies of L0 line buffers, instruction coalescing, and soft-barrier hints to attack issues of reduced instruction fetch bandwidth as discussed in Section 4.3.1. To reduce contention in accessing the data memory ports, we employ **data coalescing** mechanisms where we combine load requests that are accessing words from the same cache line. We do not support coalescing for store requests and atomic memory operations as the hardware logic in detecting such synergistic sharing is more complex.

The design rationale behind sharing the LLFUs is based on the observation that many fork-join-centric parallel programs under utilize these resources (see Section 4.4.2). We make an important

observation that instructions executed by the lanes are affected by the sharing policies both in the front-end *and* back-end of the pipeline. Sharing the LLFU resources can lead to divergence amongst the lanes that are executing the same instruction. We propose a novel **lockstep sharing** mechanism based on this observation. Lockstep sharing is a simple idea wherein two or more lanes executing the same instruction need to wait for each other while sharing the LLFUs. One way to implement this mechanism is to provide additional information of PCs to the arbiters that manage access to the LLFU resources. The arbiters can utilize this information to advance lanes executing the same instruction in a lockstep fashion. Sharing the LLFUs in such lockstep fashion aids in exploiting instruction redundancy. However, lockstep sharing trades a loss in performance for better efficiency. If programs sparingly use LLFU resources, lockstep execution can greatly help in efficiency gains with minimal loss in performance.

4.3.4 Sharing the Instruction Port, Front-end, and LLFUs

The lanes in the *CL-NI-NF-NL* design share the instruction memory port, front-end, data memory port, and LLFUs. We call this design point as a *sharing imem+fe+llfu* design. Unlike the *sharing all* design point, the ALU resources remain exclusive to each lane. The *sharing imem+fe+llfu* design employs L0 buffers, instruction coalescing, and soft-barrier hint instructions for reducing instruction fetch bandwidth contention and improving instruction fetch efficiency. Additionally, we include the *prioritized thread-selection* mechanism and the *lockstep sharing* mechanism to increases the chances of lanes executing similar instructions. One can expect this design point to combine benefits of reduced instruction and data accesses and additionally, benefit from the amortization of front-end logic.

4.4 SSA Evaluation Methodology and Results

This section presents the evaluation methodology and application kernels. We focus on evaluating the performance of single-tile SSA architectures which use conjoined-lanes to execute fork-join-centric parallel programs. We evaluate the smart sharing mechanisms for each design point in the SSA design space. For comparison, we include baseline SPMD runtime results for algorithms that can also be expressed as loop-centric parallel programs.

4.4.1 SSA Simulation Models

To evaluate the design space for the conjoined-lane organizations in SSAs, we extended the Pydgin instruction-set simulator framework [LIB15] to execute multi-threaded programs. The Pydgin framework was additionally modified to model each design point in the SSA design-space by modeling the impact of resource conflicts. The model accepts the number of instruction and data ports, the cache line sizes, the number of L0 line buffers, and the number of LLFU resources as inputs. A set of static resources configure the arbiters to simulate a design point of interest. In addition to the resource constraints, the inputs to the model set the options for instruction coalescing, soft-barrier hints, prioritized thread-selection, and lockstep sharing. The model reports performance in steps taken for program completion. In each step, the model evaluates the resource demands of instructions currently executing on each lane and advances the instructions based on the availability of the resources. A lane stalls execution if a resource was not allocated based on the outcome of various arbiters. We assume a perfect memory model and do not account for any memory-access-related timing overheads. We measure efficiency for instruction accesses as a ratio of total number of accesses that are coalesced or hit in the L0 buffers divided by the total number of instruction accesses. The efficiency for data accesses is measured as a ratio of the number of accesses that get coalesced divided by the total number of data accesses.

The evaluation methodology for SSAs differs compared to the detailed cycle-level and register-transfer-level modeling used in XLOOPS. In XLOOPS, we narrowed the design space to a primary configuration of four lanes with fixed sharing mechanisms. In SSAs, we are interested in fundamentally understanding the interactions of application characteristics, resource constraints, and smart sharing mechanisms over a rich and vast desig space. We collected the results for 10 static SSA configurations while exhaustively sweeping the sharing mechanisms which resulted in a total of approximately 6000 simulations. Our approach lets us understand these trade-offs and enables us to draw conclusions that are not tied to a specific micoarchitectural instantiation of the design. The extended Pydgin models are open-sourced and available in the `tpa-analysis` branch of the Pydgin repository (`https://github.com/cornell-brg/pydgin/tree/tpa-analysis`).

Name	Suite	Input	PM	DInsts	Integer	Load	Store	AMO	MDU	FPU	Task	WLoop
bilateral	custom	256×256 image	p	26.57	34.48%	13.25%	1.02%	0.00%	0.73%	50.51%	99.21%	0.79%
dct8x8m	custom	782 8x8 blocks	p	54.73	5.87%	17.10%	12.72%	0.00%	0.00%	64.31%	99.95%	0.05%
mriq	custom	100-space, 256 points	p	8.48	54.56%	17.35%	6.20%	0.01%	0.00%	21.89%	98.25%	1.75%
rgb2cmyk	custom	1380×1080 image	p	42.93	61.72%	13.95%	24.33%	0.00%	0.00%	0.00%	99.52%	0.48%
strsearch	custom	210 strings, 210 docs	p	20.88	80.98%	18.67%	0.35%	0.00%	0.00%	0.00%	98.15%	1.85%
uts	custom	-t 1 -a 2 -d 3 -b 6 -r 502	np	14.83	76.85%	12.25%	9.97%	0.13%	0.75%	0.06%	92.12%	7.88%
bfs-d	pbbs	randLocalGraph_J_5_150K	p	36.92	73.32%	20.37%	5.63%	0.67%	0.00%	0.00%	96.12%	3.88%
bfs-nd	pbbs	randLocalGraph_J_5_150K	p	59.02	74.01%	18.58%	7.14%	0.28%	0.00%	0.00%	97.49%	2.51%
dict	pbbs	exptSeq_1M_int	p	45.15	80.13%	16.39%	3.15%	0.34%	0.00%	0.00%	99.75%	0.25%
mis	pbbs	randLocalGraph_J_5_50000	p	29.62	73.19%	21.33%	4.12%	1.35%	0.00%	0.00%	99.66%	0.34%
rdups	pbbs	trigramSeq_300K_pair_int	p	50.58	78.28%	17.24%	4.35%	0.14%	0.00%	0.00%	99.78%	0.22%
sarray	pbbs	trigramString_120K	p	79.94	69.81%	17.95%	12.22%	0.02%	0.00%	0.00%	87.75%	12.25%
qsort	pbbs	exptSeq_10K_double	rss	27.92	73.95%	14.91%	11.13%	0.01%	0.00%	0.00%	67.28%	32.72%
qsort-1	pbbs	almostSortedSeq_10K_double	rss	25.45	73.20%	15.08%	11.71%	0.01%	0.00%	0.00%	71.24%	28.76%
qsort-2	pbbs	trigramSeq_50K	rss	24.19	73.57%	22.19%	4.20%	0.04%	0.00%	0.00%	82.20%	17.80%
sampsort	pbbs	exptSeq_10K_double	np	40.52	64.03%	18.73%	17.03%	0.19%	0.02%	0.00%	87.30%	12.70%
sampsort-1	pbbs	almostSortedSeq_10K_double	np	29.51	62.10%	19.71%	17.89%	0.28%	0.02%	0.00%	81.66%	18.34%
sampsort-2	pbbs	trigramSeq_50K	np	64.83	59.74%	25.06%	14.64%	0.52%	0.04%	0.00%	70.11%	29.89%
hull	pbbs	2Dkuzmin_100000	rss	14.70	61.83%	14.74%	5.02%	0.04%	0.00%	18.37%	95.56%	4.44%
cilksort	cilk	-n 300000	rss	47.39	75.67%	14.02%	10.29%	0.02%	0.00%	0.00%	98.62%	1.38%
heat	cilk	-g 1 -nx 256 -ny 64 -nt 1	rss	5.10	70.68%	13.11%	10.83%	0.03%	2.54%	2.81%	97.62%	2.38%
ksack	cilk	knapsack-small-1.input	rss	35.15	54.48%	22.04%	22.74%	0.60%	0.15%	0.00%	63.86%	36.14%
matmul	cilk	200	rss	68.51	51.41%	24.04%	0.77%	0.01%	0.00%	23.76%	99.25%	0.75%

Table 4.2: Application Kernel Characteristics for WSRT – Suite = benchmark suite; Input = input dataset & options; PM = parallelization methods: p = `parallel_for`, np = nested `parallel_for`, rss = recursive spawn-and-sync; DInsts = total dynamic instruction counts in millions collected on the *ideal MIMD* model; Integer = percent of integer instructions; Load = percent of load instructions; Store = percent of store instructions; AMO = percent of atomic memory instructions; MDU = percent of multiply-divide instructions; FPU = percent of floating-point instructions; Task = percent of instructions in tasks; WLoop = percent of instructions in worker loop.

Name	Suite	Input	PM	DInsts	Integer	Load	Store	AMO	MDU	FPU
bilateral	custom	256×256 image	p	26.41	34.01%	13.47%	0.98%	0.00%	0.74%	50.80%
dct8x8m	custom	782 8x8 blocks	p	54.70	5.85%	17.09%	12.71%	0.00%	0.00%	64.35%
mriq	custom	100-space, 256 points	p	8.32	54.36%	17.29%	6.03%	0.00%	0.00%	22.32%
rgb2cmyk	custom	1380×1080 image	p	42.72	61.62%	13.96%	24.42%	0.00%	0.00%	0.00%
strsearch	custom	210 strings, 210 docs	p	20.41	81.26%	18.52%	0.22%	0.00%	0.00%	0.00%
uts	custom	-t 1 -a 2 -d 3 -b 6 -r 502	p	15.35	78.25%	11.61%	9.24%	0.11%	0.73%	0.06%
bfs-d	pbbs	randLocalGraph_J_5_150K	p	35.26	74.12%	20.26%	4.96%	0.66%	0.00%	0.00%
bfs-nd	pbbs	randLocalGraph_J_5_150K	p	57.33	74.60%	18.42%	6.71%	0.26%	0.00%	0.00%
dict	pbbs	exptSeq_1M_int	p	45.08	80.28%	16.33%	3.07%	0.33%	0.00%	0.00%
mis	pbbs	randLocalGraph_J_5_50000	p	29.53	73.24%	21.34%	4.06%	1.36%	0.00%	0.00%
rdups	pbbs	trigramSeq_300K_pair_int	p	50.44	78.36%	17.21%	4.29%	0.14%	0.00%	0.00%
sarray	pbbs	trigramString_120K	p	63.48	69.72%	18.13%	12.15%	0.00%	0.00%	0.00%
hull	pbbs	2Dkuzmin_100000	p	13.96	61.97%	14.37%	4.31%	0.00%	0.00%	19.35%

Table 4.3: Application Kernel Characteristics for SPMD – Suite = benchmark suite; Input = input dataset & options; PM = parallelization methods: p = `parallel_for`; DInsts = total dynamic instruction counts in millions collected on the *ideal MIMD* model; Integer = percent of integer instructions; Load = percent of load instructions; Store = percent of store instructions; AMO = percent of atomic memory instructions; MDU = percent of multiply-divide instructions; FPU = percent of floating-point instructions.

4.4.2 SSA Application Kernels

We have ported a total of 19 application kernels to a 32-bit RISC-based instruction-set (same as XLOOPS) using the work-stealing runtime (WSRT) framework (developed in-house). Prior work (see [TWB16] and [KJT+17]) presents results our WSRT runtime implementation as compared with the state-of-the-art Intel Cilk++ and Intel TBB runtimes. The application kernels are ported from the problem-based benchmark suite (PBBS v.0.1) [SBF+12], Cilk benchmarks (Cilk v. 5.4.6) [FLR98], and an in-house benchmark suite. The kernels include simple for loops expressed using a *parallel_for*, nested parallel loops, and recursive spawn-and-sync parallelism. In addition to the WSRT implementations, we ported a subset of 13 algorithms to a baseline loop-centric (SPMD) runtime implementation. We include both the WSRT and SPMD implementations for the following reasons: (i) to compare related work that has extensively investigated exploiting redundancies for algorithms expressed as loop-centric parallel programs; (ii) to highlight the differences in implementations for a given algorithm expressed using different parallelization and scheduling strategies; and (iii) to show that the instruction overheads in employing a dynamic scheduling strategy are minimal compared to a static scheduling strategy. All kernels were compiled using a cross-compiler toolchain that uses GCC-4.4.1, Newlib-1.17.0, and the GNU standard C++ library.

Application kernels are selected from a diverse range of application domains including those with regular task parallelism like image processing (e.g., bilateral filter, color space conversion, discrete cosine transform) and scientific computations (e.g., heat diffusion simulation, MRI griding, and dense matrix multiplication) as well as challenging irregular graph algorithms (e.g., breadth-first search, maximal matching, maximal independent set), text processing (suffix array), and non-numeric search/sort/optimization problems (e.g., radix sort, substring matching, dictionary, knapsack, convex hull). Table 4.2 shows the application characteristics for the WSRT implementations and Table 4.3 shows the application characteristics for the baseline SPMD implementations. Detailed descriptions for PBBS kernels and Cilk kernels can be found in [SBF+12, FLR98].

We briefly describe the custom in-house kernels. The *bilateral* kernel performs a bilateral image filter with a lookup table for the distance function and an optimized Taylor-series expansion for calculating the intensity weight. Computation is parallelized across output pixels. *dct8×8m* calculates the 8×8 discrete cosine transform on an image. Computation is parallelized across 8×8 blocks. *mriq* is an image reconstruction algorithm for MRI scanning inspired by the Parboil benchmark suite [SRS+12]. Computation is parallelized across the output magnetic field gradient vector.

rgb2cmyk performs color space conversion on an image and computation is parallelized across the rows. *strsearch* implements the Knuth-Morris-Pratt algorithm with a deterministic finite automata to search a collection of byte streams for a set of substrings. Computation is parallelized across different streams. *uts* is inspired by the implementation in [OHL+06]. The unbalanced tree search (UTS) benchmark is a synthetic kernel specifically designed to evaluate the performance and ease of programming for parallel applications requiring dynamic load-balancing. The SPMD implementation parallelizes the kernel across the frontiers using double buffering whereas the WSRT implementation uses nested parallelism which is more straightforward to express.

Table 4.2 and Table 4.3 show that the overheads of the WSRT runtime in terms of total dynamic instruction counts are not significantly higher than the SPMD implementations. Additionally, most of the applications spend all of the time executing tasks in the parallel regions with a few exceptions for kernels with limited amounts of parallelism which include *qsort, sampsort,* and *ksack*. For these kernels one can expect low redundancies in the instruction streams. The application characteristics also include a detailed breakdown of the instruction mix. The instruction mix reveals that for most challenging non-numeric workloads the LLFUs are indeed used sparingly. Table A.1 shows the speedup of the *no sharing* design that uses four lanes compared to executing the kernels on a single-thread. The results we present use the *no sharing* design as a baseline.

4.4.3 Evaluating the Potential for SSA Designs

This section presents the results for the SSA design-space exploration. To understand the potential benefits and trade-offs involved for conjoined-lane organizations in the SSAs, we answer the following key questions:

- Q1. Is there sufficient redundancy in SPMD and WSRT applications?
- Q2. What mechanisms are key to sharing the instruction port only?
- Q3. What mechanisms are key to sharing the instruction port and the front-end only?
- Q4. What mechanisms are key to sharing the instruction port and the LLFUs only?
- Q5. What mechanisms are key to sharing the instruction port, front-end, and the LLFUs?
- Q6. How do the SSA design points compare?

Q1. Is there sufficient redundancy in SPMD and WSRT applications?

As noted earlier, exploiting instruction redundancy is a key principle in SSAs to improve performance and efficiency. The benefits of area reductions by sharing expensive resources only makes sense if there is a sufficient amount of instruction redundancy present in SPMD and WSRT applications. To answer the question of what is the amount of redundancy present in a given application, we consider the *no sharing* design. Instruction redundancy is defined as the number of instructions that are identical and could potentially be fetched only once and broadcast to the lane back-ends. To measure the instruction redundancy we use the equation as shown in 4.1. For a conjoined-lane architecture with four lanes, the maximum value for instruction redundancy would be 75% which means that in the best case one unique instruction fetched can be shared by the four lanes.

$$Instruction\ Redundancy(\%) = \frac{\sum Total\ Instructions - \sum Unique\ Instructions}{\sum Total\ Instructions} \times 100 \qquad (4.1)$$

Figure 4.4(a) shows the instruction redundancy in SPMD applications and Figure 4.4(b) shows the instruction redundancy in WSRT applications. The grey bars in the plots show the measurements for instruction redundancy executing on the *no sharing* design. The results suggest that there is no instruction redundancy in both the SPMD and WSRT applications. However, if we modify the SPMD and WSRT runtime implementations to include soft-barrier hint instructions the probability of lanes executing similar instructions improves. The soft-barrier hints indicate a synchronization point and delaying the lane executions at these synchronization points increases instruction redundancy. The green bars in both plots show the instruction redundancy measurements when executing the applications with soft-barrier hints.

We can group applications based on the instruction redundancy into three categories. Applications with *high redundancy* (50%–75%) which include *dct8x8m, matmul, bilateral, rgb2cymk,* and *hull*. Applications with *medium redundancy* (25%–50%) which include *bfs-nd, rdups,* and *sarray*. Applications with *low redundancy* (up to 25%) which include *bfs-d, dict, mis, strsearch, cilksort, mriq,* and *uts*. The *uts* kernel shows how instruction redundancy can vary based on the parallelization and scheduling strategy. The SPMD implementation of *uts* shows that there is barely any redundancy whereas the WSRT implementation shows that there is 17% instruction redundancy. The *qsort* and *samport* kernels in WSRT workloads show that instruction redundancy can vary

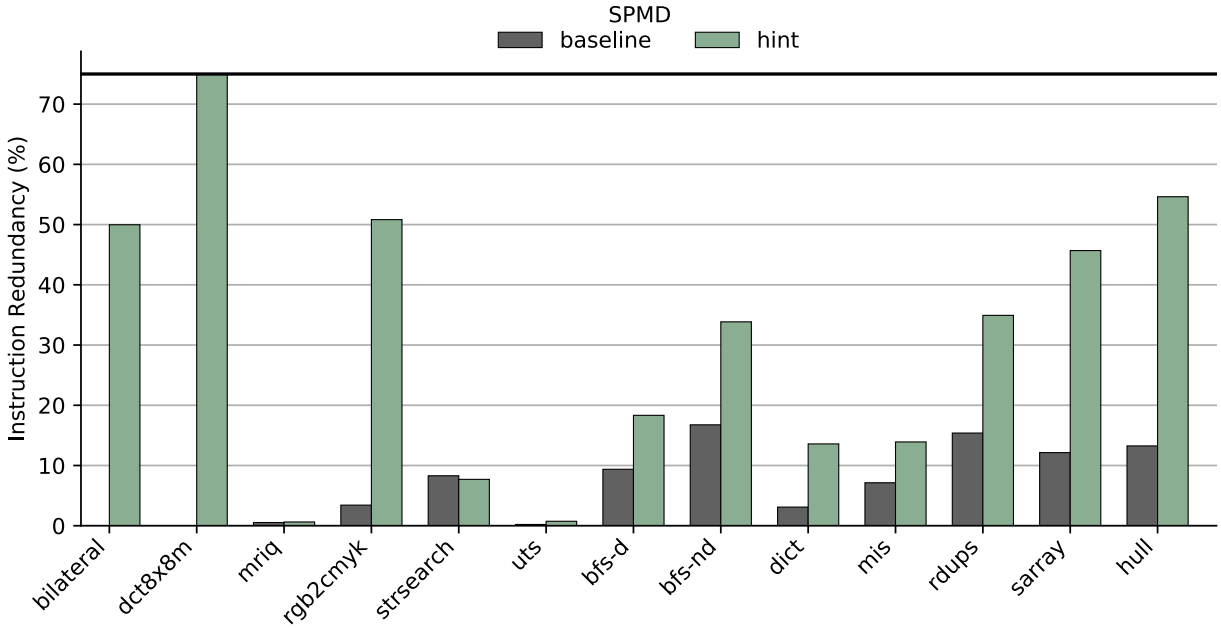
based on the input dataset for a given kernel. Our results show that there is inherent redundancy in SPMD and WSRT applications that can be exploited by using *soft-barrier hints*.

Recall soft-barrier hints in SSAs are implemented by pausing the lane execution for a maximum number of steps. The idea is to delay the lane execution at potential synchronization points in the runtimes in order to increase the probability for converged program execution. For the results shown we use a maximum delay count of 1000 steps. We choose 1000 steps as the limit based on the number of instructions it takes to push a task to a task queue (100 instructions), pop a task from a task queue (40 instructions), and the stealing overheads involved in victim selection and checking for valid work (50 instructions). A task can spawn two or more child tasks and the maximum wait timeout limit gives sufficient opportunities for threads to sync up.

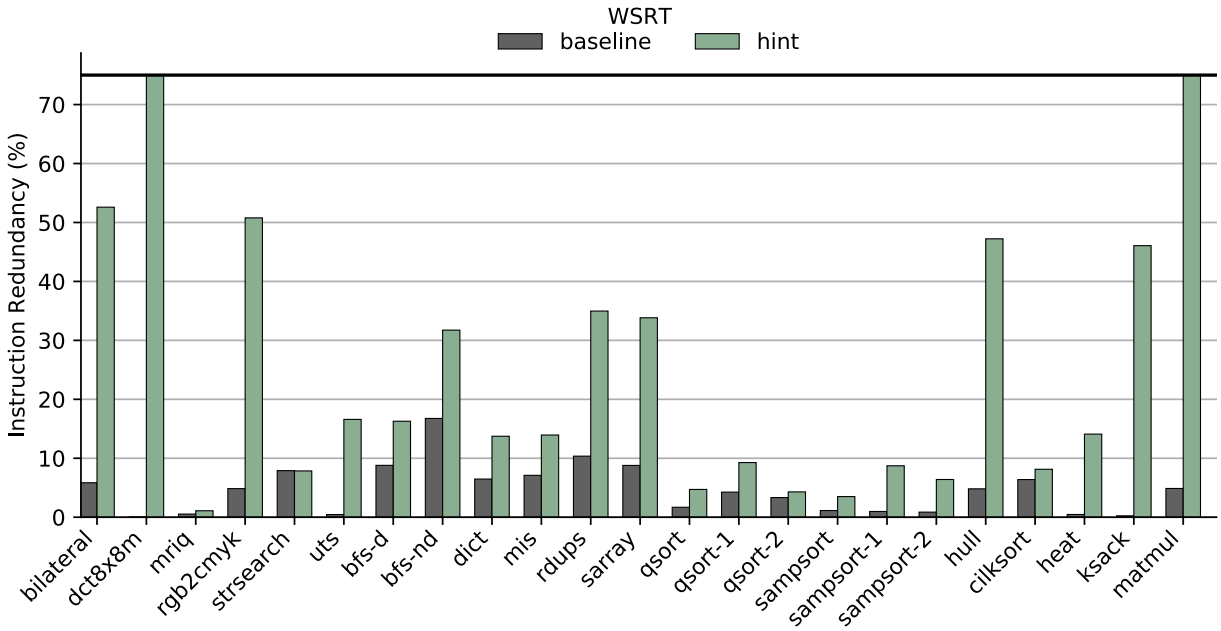
For SPMD applications, the impact of delaying lane execution is minimal as the hint instructions are used to pause lanes at the start of a loop execution. The time spent in setting up the parallel region is minimal. For WSRT applications, the hint instructions are included in the worker loop. For applications with sufficient parallelism, the impact of hint instructions are minimal as lanes are mostly converged in execution of the tasks or scheduling logic (worker loop). However, for applications with low parallel regions hint instructions can hurt the performance. The application kernels for which we observe a loss in performance (increase in delay shown in percent) include: *uts (27%)*, *qsort-2 (11%)*, *sampsort (37%), sampsort-1 (46%), sampsort-2 (72%)*, and *ksack (57%)*. For *uts* and *ksack* kernels, the loss in performance can be traded for an increase in efficiency. Future work can explore an adaptive mechanism that adjusts the delays in soft-barrier hints to mitigate these performance losses. Detailed performance numbers are in the Table A.2.

Result 1 – *There is sufficient instruction redundancy in WSRT and SPMD runtimes that can be exploited by using soft-barrier hints.*

Our results for SPMD applications are in accord with previous research in the context of SPMD runtimes [MCM+14]. To the best of our knowledge, we are the first to explore instruction redundancy in the context of WSRT runtimes. The potential to save instruction accesses can vary from 7% to 75%.



(a) Instruction redundancy in SPMD applications



(b) Instruction redundancy in WSRT applications

Figure 4.4: Instruction Redundancy in SPMD and WSRT Applications – Instruction redundancy exists as different threads work on different sets of data but execute the same instructions in the form of parallel loops in SPMD runtimes and parallel tasks in WSRT runtimes. Results using the *no sharing* design show that soft-barrier hints are required to expose the inherent instruction redundancy.

Q2. What mechanisms are key to sharing the instruction port only?

As discussed in Section 4.3.1, we explore the following strategies to address reduced instruction bandwidth for the *sharing imem only* designs: L0 buffers, instruction coalescing, and soft-barrier hints. We present the results for delay and efficiency by normalizing to the *no sharing* design point.

Figures 4.5 and 4.6 show results of sharing a single instruction port amongst conjoined-lanes for SPMD and WSRT kernels respectively. The bars represent the following design configurations. *base* represents a naive configuration of the *CL-1I-4F-4L* design point which uses round-robin arbitration to share the instruction port. *coalesce* represents inclusion of the instruction coalescing smart sharing mechanism in the *base* configuration. *coalesce+hints* combines the instruction coalescing and soft-barrier hint mechanisms. The *l0* configuration adds a single L0 line buffer to the *base* configuration. The *l0+coalesce* adds instruction coalescing to a design with L0 line buffers and lastly, the *l0+coalesce+hint* configuration combines all mechanisms.

Result 2 – *Adding L0 buffers is a complexity-effective mechanism to improve the normalized delay and reduce the number of instruction accesses.*

Compared to adding instruction coalescing only (*coalesce*) and combining instruction coalescing with soft-barrier hints (*coalesce+hints*) adding a single L0 line buffer improves the performance and efficiency across all the application kernels for both the SPMD and WSRT runtimes. L0 buffers are cheap and easy to implement. The L0 buffers improve the performance by reducing the delay from 4× to a range between 1×–1.7×for SPMD applications and 1×–1.6×for WSRT applications. The instruction access efficiency improves in the range of 26%–38% for SPMD applications and in the range of 27%–40% for WSRT applications. The L0 buffers do not exploit redundancy across the lanes compared to the smart sharing mechanisms but mostly exploit the spatial locality within an instruction stream executing on each lane. The L0 buffers improve efficiency as well as the performance by reducing the pressure on the instruction port.

Result 3 – *Instruction coalescing and soft-barrier hints along with L0 buffers are the key mechanisms for improving the performance and efficiency when only sharing the instruction port.*

Instruction coalescing works by combining requests that miss in the L0 buffer across lanes. On an average, instruction coalescing further improves the performance by about 18% in SPMD and 15% in WSRT applications whereas the efficiency (reduction in instruction accesses) improves on an average by 11% in SPMD and 6% in WSRT runtimes. The soft-barrier hints seem to not make much of a difference to SPMD kernels whereas the WSRT kernels see an improvement in

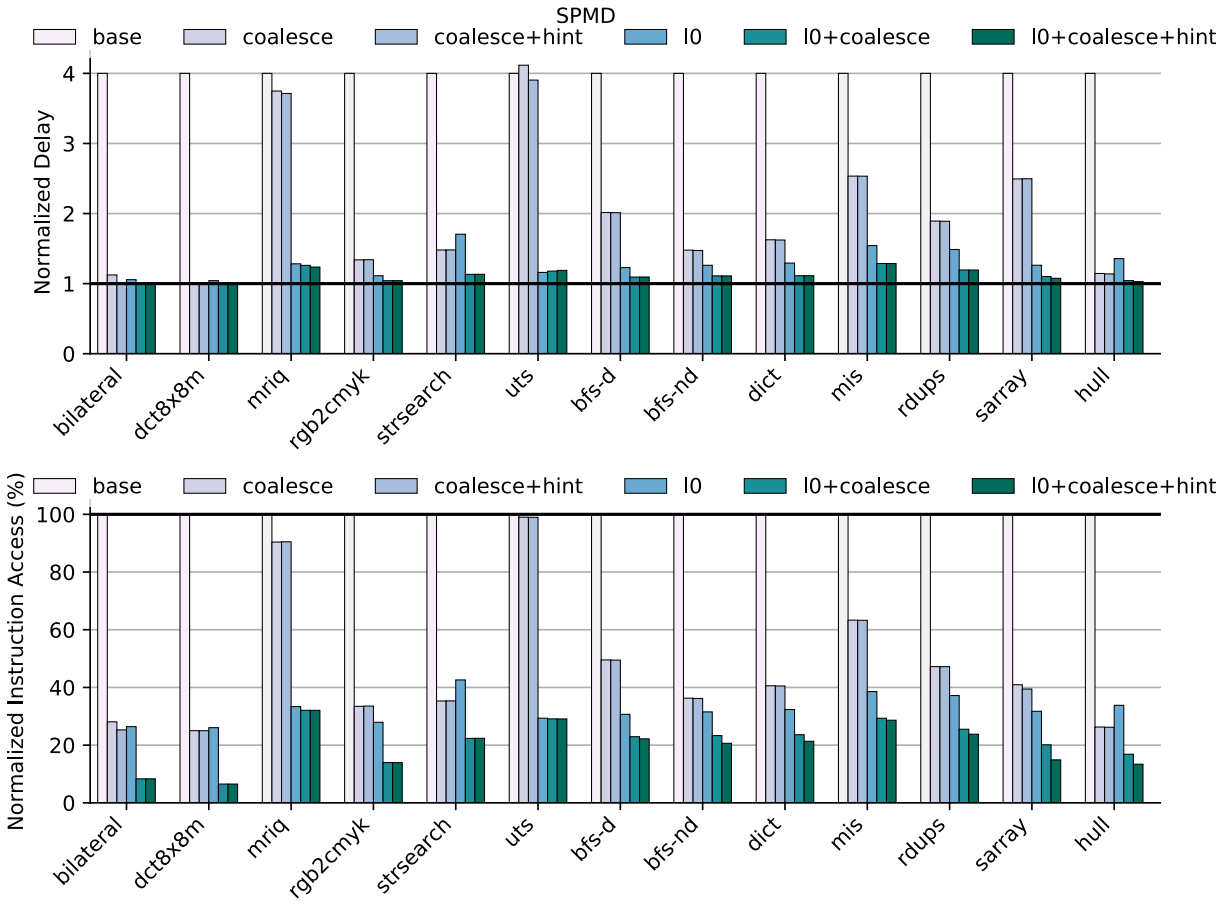


Figure 4.5: SPMD Results for Sharing One Instruction Port – The results show that the combination of L0 buffers, instruction coalescing, and soft-barrier hints significantly reduces the instuction access and improves the delay for SPMD applications. The figure on the top shows the normalized delay and the bottom shows the normalized instruction access compared to the *no sharing* design point. Each bar represents the configurations as explained: (i) *base* baseline with round-robin arbitration; (ii) *coalesce* baseline with instruction coalescing enabled; (iii) *coalescing+hint* combining coalescing with soft-barrier hints; (iv) *l0* adding a L0 buffer to each lane; (v) *l0+coalesce* combining L0 buffers with instruction coalescing; (vi) *l0+coalesce+hint* combining L0 buffer, instruction coalescing, and soft-barrier hints.

efficiency with the aid of hints. In particular, regular kernels such as *dct8x8m* and *matmul* greatly benefit from hints and are able to reach the same efficiency as the SPMD kernels (25% to 6.7%). Soft-barrier hints hurt the performance of irregular WSRT kernels with low regions of parallelism as expected based on the instruction redundancy results. However, in the case of *ksack* the efficiency improves by 15% when combining coalescing with hints but the benefit comes at a loss of performance by about 37%. WSRT runtimes benefit slightly from hints because of the unstructured control-flow in the worker loop. The combination of L0 buffers, instruction coalescing, and soft-barrier hints improve the efficiency of SPMD and WSRT kernels by reducing the instruction

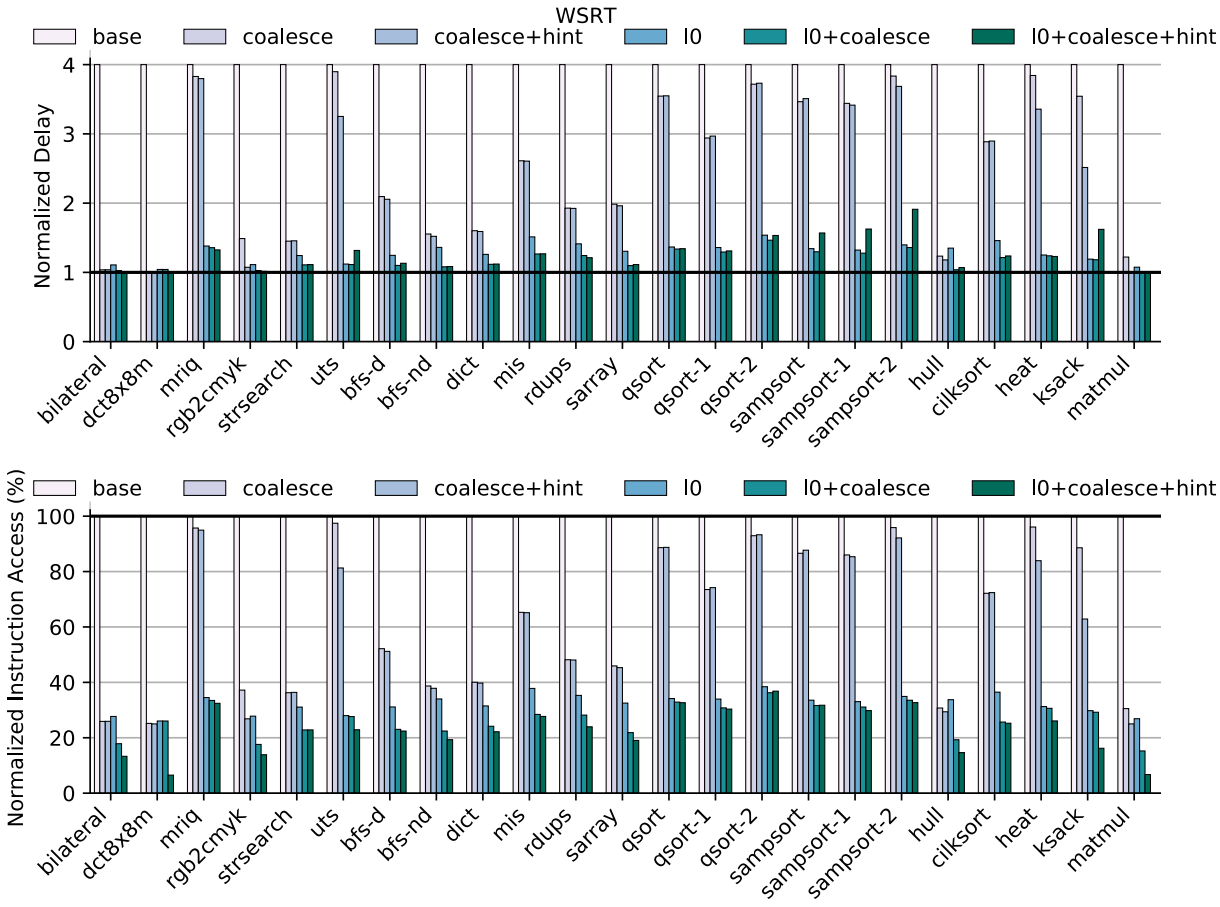
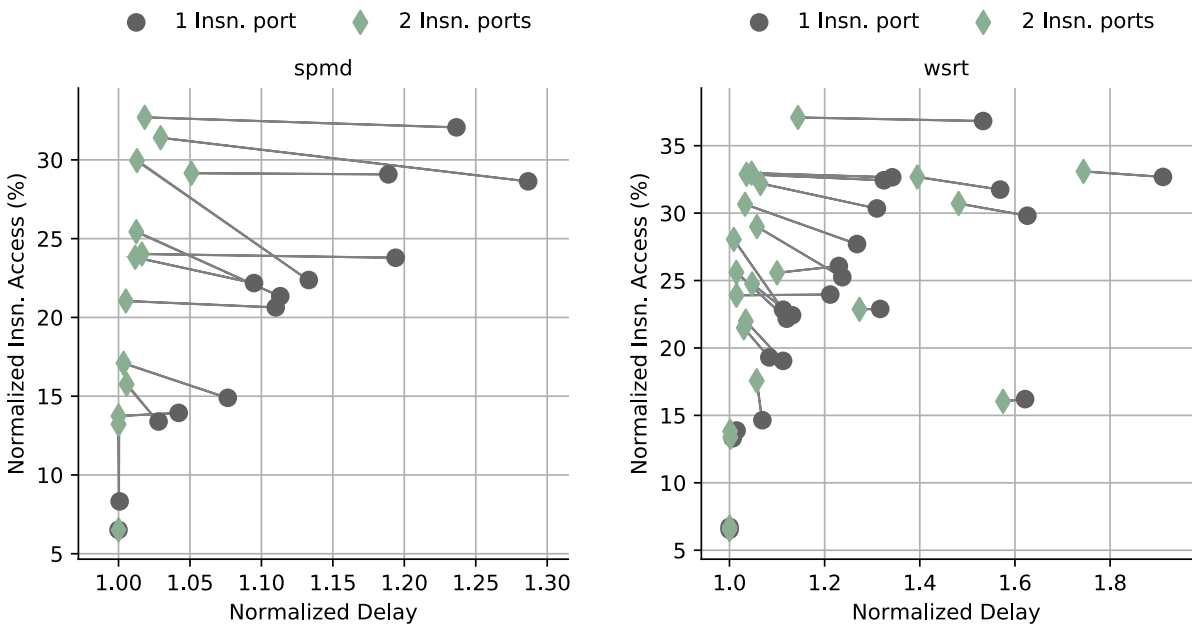


Figure 4.6: WSRT Results for Sharing One Instruction Port – The results show that the combination of L0 buffers, instruction coalescing, and soft-barrier hints significantly reduces the instuction access and improves the delay for WSRT applications. Adding soft-barrier hints reduces instruction access at the cost of increase in delay for applications with low parallel regions: *uts,sampsort,ksack*. The figure on the top shows the normalized delay and the bottom shows the normalized instruction access compared to the *no sharing* desing point. Each bar represents the configurations as explained: (i) *base* baseline with round-robin arbitration; (ii) *coalesce* baseline with instruction coalescing enabled; (iii) *coalescing+hint* combining coalescing with soft-barrier hints; (iv) *l0* adding a L0 buffer to each lane; (v) *l0+coalesce* combining L0 buffers with instruction coalescing; (vi) *l0+coalesce+hint* combining L0 buffer, instruction coalescing, and soft-barrier hints.

accesses from 100% to a range of 6.5%–36% which is even better than the expected results from the instruction redundancy results. With a single instruction port and the smart sharing mechanisms the performance loss can be minimized to just 15% for SPMD and 26% for WSRT kernels.

Comparing results for increasing instruction ports

Figure 4.7 compares the performance of SPMD and WSRT kernels when sharing one port to two ports on designs that use L0 buffers combined with instruction coalescing and soft-barrier



(a) SPMD applications

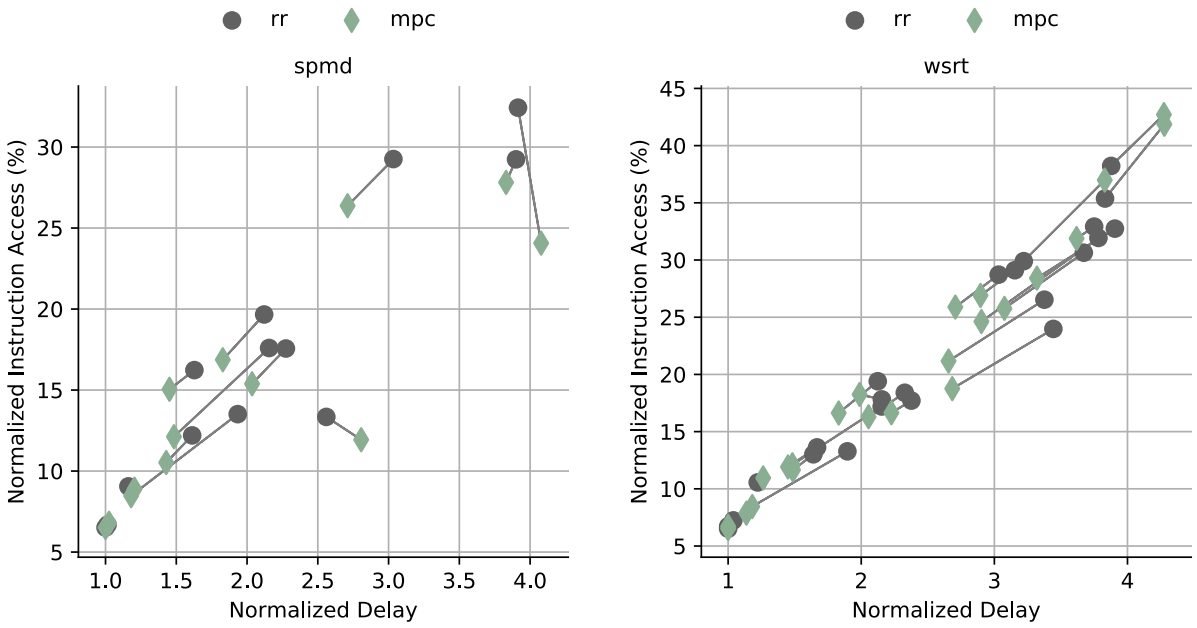
(b) WSRT applications

Figure 4.7: SPMD and WSRT Results Sharing One vs. Two Instruction Ports – Sharing two instruction ports improves delay for most of the SPMD and WSRT applications as indicated by the flat slopes of the lines that connect the points for sharing one instruction port (grey circles) and the points for sharing two instruction ports (green diamonds).

hints. The results show that 9 out of 13 SPMD kernels do not benefit more than 10% improvement in performance for almost similar trends in efficiency by using two ports. SPMD kernels that benefit the most include *mriq, uts, mis, and rdups*. With the exception of *mriq, mis, rdups, qsort, and sampsort* most of the WSRT kernels also show no significant benefits in performance by using an additional instruction port. Increasing the instruction ports improves performance for both SPMD and WSRT kernels while increasing the area costs. Detailed results for the design point sharing two instruction ports are present in Tables A.3 and A.4.

Q3. What mechanisms are key to sharing the instruction port and the front-end only?

We now consider the *sharing imem+fe only* design point. Based on the results from *sharing imem only*, we enable a naive SSA design to include L0 buffers and use soft-barrier hints. Instruction coalescing does not apply to this design as there is a single front-end. The naive SSA design uses the round-robin thread-selection mechanism under divergence. It is important to recall that the thread-selection mechanism when sharing the front-end needs to guarantee forward progress



(a) Min-pc/RR thread-selection helps all SPMD applications

(b) Min-pc/RR thread-selection helps all WSRT applications except *qsort*

Figure 4.8: Comparison of Thread-selection Mechanism for Sharing One Instruction and Front-end – The baseline configuration statically has the L0 buffer and soft-barrier hints enabled and statically shares one instruction port and front-end. The results show that the hybrid minimum-pc/round-robin thread-selection mechanism improves the delay and reduces the instruction access for most of the SPMD and WSRT application kernels.

for unstructured control-flows and the baseline round-robin thread-selection is a reasonable design choice.

Result 4 – *The hybrid minimum-pc/round-robin thread-selection mechanism is the key to improve performance and efficiency to sharing the instruction and the front-end only.*

Figure 4.8 shows the results that compare the round-robin (RR) vs. hybrid minimum-pc/round-robin (MPC) thread-selection mechanisms for both the SPMD and WSRT kernels. The scatter plot shows that the MPC mechanism improves both the performance and efficiency for both SPMD and WSRT kernels. *rgb2cmyk* sees a 40% improvement in performance and efficiency for both SPMD and WSRT implementations. The kernel is mostly regular but has some data-dependent control-flow which causes divergence. *dict* is an example of an irregular kernel that sees a 30% improvement in performance and efficiency for both SPMD and WSRT implementations. It is interesting to note that *qsort* observes a loss in performance when using MPC mechanism. The base-case of *qsort* is highly irregular as it performs a comparison-sort using the quick-sort algo-

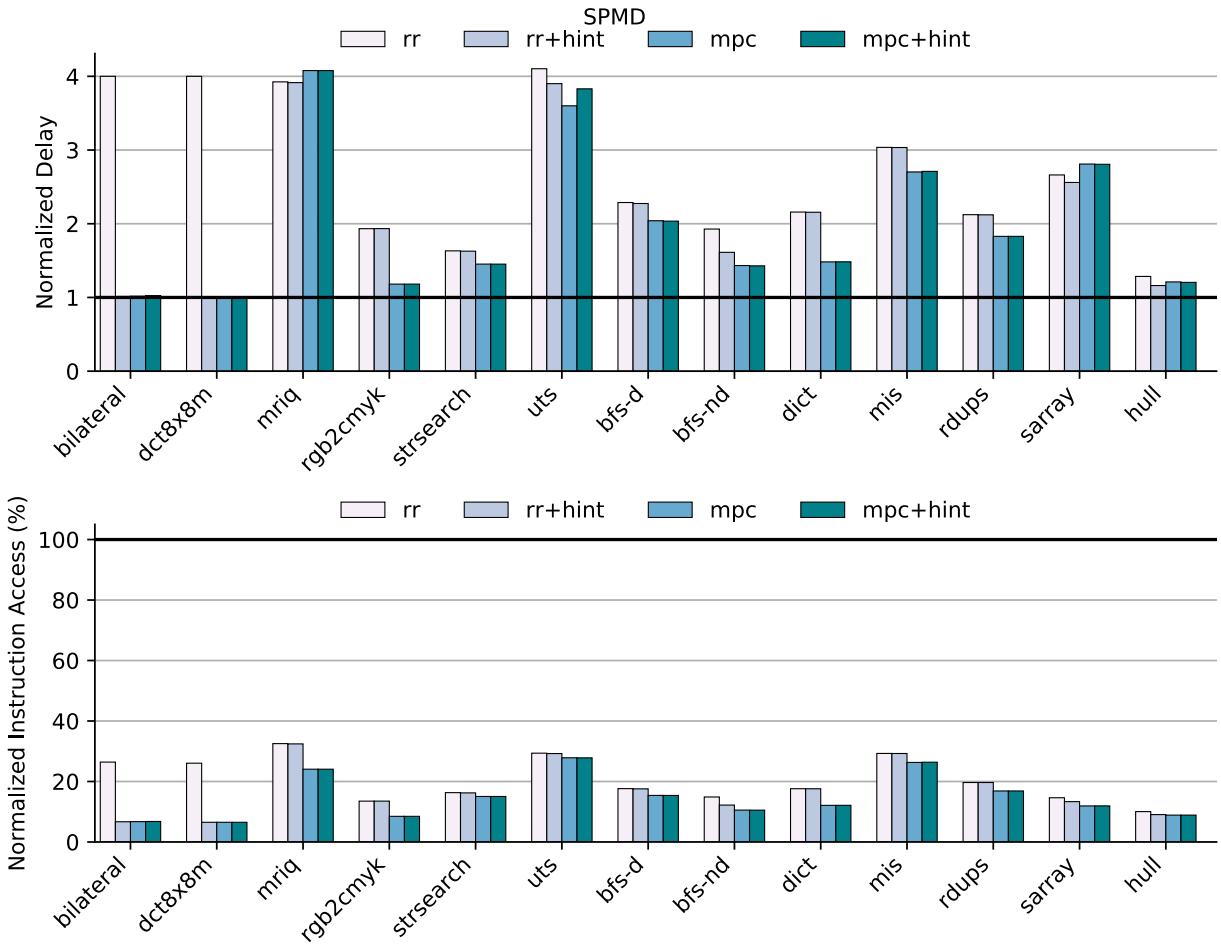


Figure 4.9: SPMD Results for Sharing One Instruction and Front-end – The results show that for a baseline static configuration with L0 buffers and a single instruction-port and frontend, the hybrid minimum-pc/round-robin thread-selection mechanism is the key for pareto-optimality. The figure on the top shows the normalized delay and the bottom shows the normalized instruction access compared to the ideal MIMD model (lower the better). Each bar represents the configurations as explained: (i) *rr* baseline with round-robin arbitration; (ii) *rr+hint* baseline with soft-barrier hints enabled; (iii) *mpc* baseline with the hybrid minimum-pc/round-robin thread selection mechanism; (iv) *mpc+hint mpc* configuration with soft-barrier hints.

rithm. The RR mechanism works well for *qsort* as under high divergence each thread gets a fair use of the front-end to make forward progress.

Are soft-barrier hints important when sharing the front-end?

Figure 4.9 and Figure 4.10 show the detailed results for comparing the performance and efficiency of SPMD and WSRT using RR and MPC mechanisms combined with the soft-barrier hints. For the SPMD kernels the soft-barrier hints does not make a difference when using either the RR

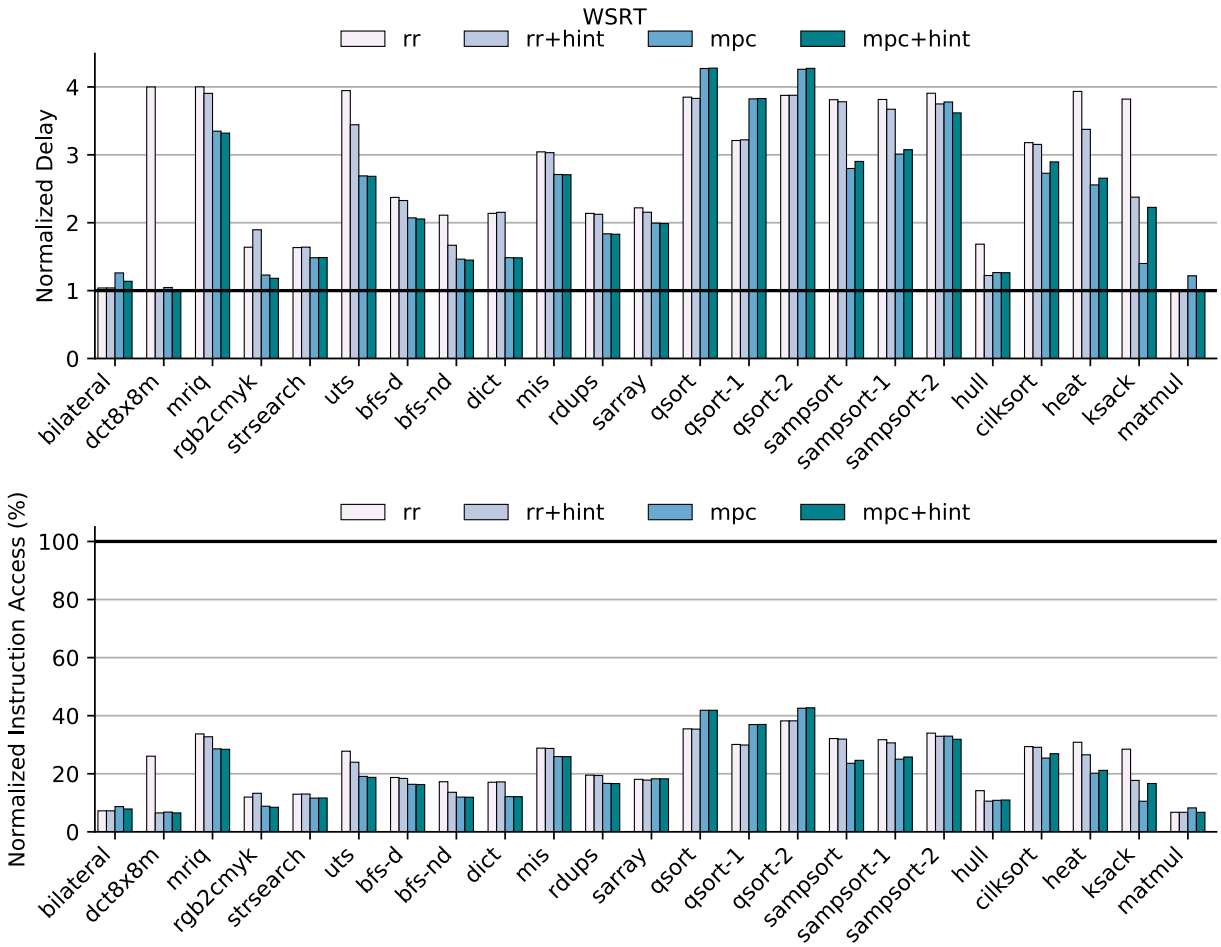
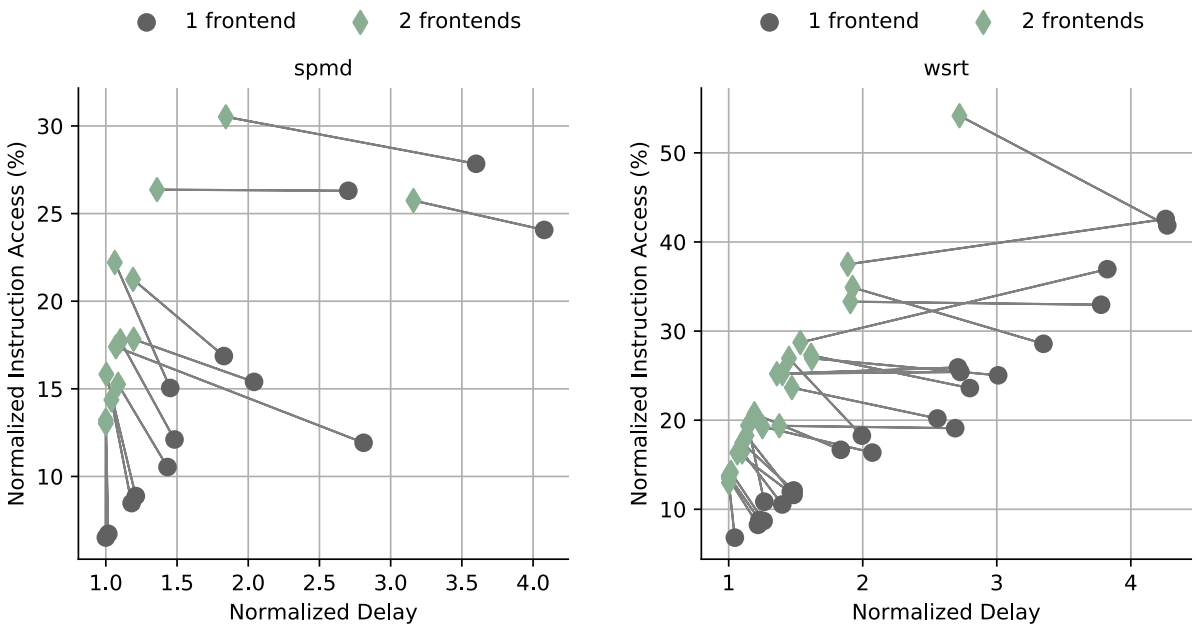


Figure 4.10: WSRT Results for Sharing One Instruction and Front-end – The results show that for a baseline static configuration with L0 buffers and a single instruction-port and frontend, the hybrid minimum-pc/round-robin thread-selection mechanism is the key for pareto-optimality. The *mpc* thread-selection hurts the *qsort* kernel due to it's highly irregular control-flow. The figure on the top shows the normalized delay and the bottom shows the normalized instruction access compared to the ideal MIMD model (lower the better). Each bar represents the configurations as explained: (i) *rr* baseline with round-robin arbitration; (ii) *rr+hint* baseline with soft-barrier hints enabled; (iii) *mpc* baseline with the hybrid minimum-pc/round-robin thread selection mechanism; (iv) *mpc+hint mpc* configuration with soft-barrier hints.

or MPC thread-selection. For WSRT kernels, some kernels such as *dct8x8m, uts, heat, ksack* see improvements in performance and efficiency with the addition of hints to the RR thread-selection. Combining MPC and the hints does not impact the performance and efficiency results with the exception of the *ksack* kernel where the performance with hints is worsened as the kernel has low amounts of parallelism. Overall performance and efficiency improvements by using just the MPC thread-selection seem to be the key mechanism. Sharing the front-end and using the MPC thread-



(a) SPMD applications

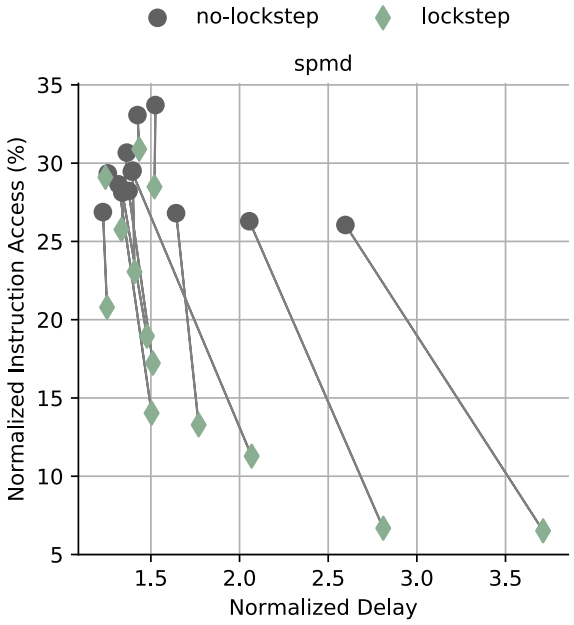
(b) WSRT applications

Figure 4.11: SPMD and WSRT Results Sharing One vs. Two Instruction Ports and Front-ends – Two front-ends improve the execution delay for irregular application kernels and trade-off the instruction access. The improvements in execution delay suggest that two front-ends are a compelling design point as the increase in instruction access is minimal as indicated by flat slopes for the lines that connect the points for sharing one instruction port (grey circles) and the points for sharing two instruction ports (green diamonds).

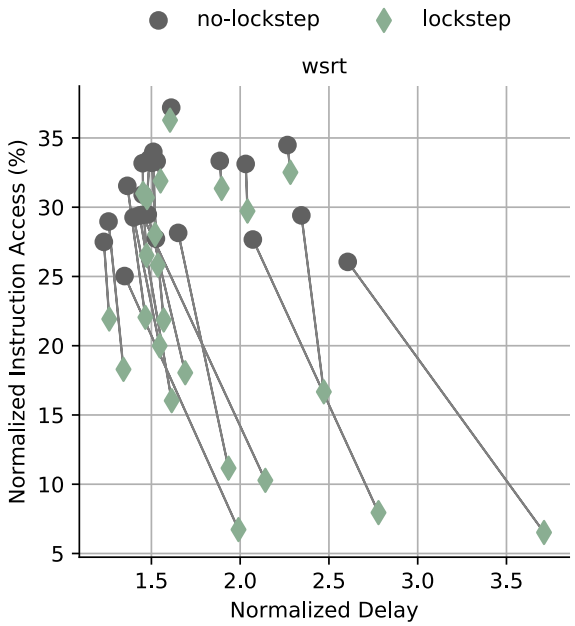
selection policy achieves the goal of convergent execution which increases instruction redundancy. *Soft-barrier hints are not critical to designs which use a single front-end.*

Comparing results for increasing the front-end resources

Figure 4.11 shows the results of comparing designs that share one vs. two front-ends. Both the designs include the L0 buffers and MPC thread-selection. Using two front-ends improves the performance by 30% for a 5% increase in instruction accesses for SPMD kernels. For WSRT kernels, using two front-ends improves the performance on average by 40% for an increase in instruction accesses by 5%. Using two front-ends trades of an increase in area and efficiency for attractive improvements in performance. Compared to just using the MPC thread-selection, adding an extra front-end improves the performance of both the SPMD and WSRT kernels under divergence as the utilization of the back-end resources increase with an additional front-end. Detailed results for sharing two instruction ports and front-ends are present in Tables A.5 and A.6.



(a) Lockstep sharing for SPMD applications



(b) Lockstep sharing hurts performance of *bilateral,dct8x8m,matmul* the most for WSRT applications

Figure 4.12: Comparing Lockstep Mechanism for Sharing One Instruction and LLFU – The baseline configuration statically has the L0 buffer and soft-barrier hints enabled and statically shares one instruction port and the LLFU using round-robin thread-selection. The results show that lockstep sharing mechanism trades-off reduction in instruction accesses for increased execution delay for most SPMD and WSRT application kernels.

Q4. What mechanisms are key to sharing the instruction port and the LLFUs only?

Tables 4.2 and 4.3 show that there quite a few kernels that use the LLFUs sparingly. The PBBS kernels have a low usage of the LLFUs as these kernels are designed to compare algorithmic approaches, parallel programming language styles, and machine architectures across a broad set of problems. The *sharing imem+llfu* design point is an attractive solution for such kernels as it reduces the area costs of LLFUs in addition to sharing the instruction port. The mechanisms that are applicable to this design include L0 buffers, instruction coalescing, soft-barrier hints, and lockstep sharing.

Result 5 – *Lockstep sharing is the key mechanism for improving efficiency at a minimal loss in performance when sharing instruction port and LLFU only.*

Figure 4.12 shows the results that compare a baseline design that includes L0 buffers, instruction coalescing, and round-robin arbitration without lockstep sharing to the same design with lock-

step sharing for SPMD and WSRT kernels. The results show that the lockstep sharing mechanism trades reduction in instruction accesses for a loss in performance for most kernels. Figures 4.13 and 4.14 shows detailed results for sharing a single instruction port and LLFU for SPMD and WSRT kernels. For SPMD, most of the PBBS kernels observe an improvement in efficiency in the range of 6%–50% while observing a maximum loss of 10% in performance. Kernels with high instruction redundancy like *dct8x8m, bilateral, and hull* see an improvement in efficiency in the range of 60%–75% for a higher loss in performance (36%–48%). The trends for the WSRT kernels are similar. In particular, kernels with low redundancy such as *mriq, uts, mis, qsort, sampsort, and ksack* see an improvement in efficiency in the range of 2.5% to 8% for almost no difference in performance (<5%). Lockstep sharing is trades off performance for an improvement in efficiency and is effective for kernels that sparingly use these resources.

Are soft-barrier hints important when sharing the instruction port and LLFUs only?

Soft-barrier hints seem to make no difference to SPMD kernels. Adding hints minimally improves the efficiency of WSRT kernels and can sometimes hurt the performance for application kernels with low parallel regions (*uts, qsort, sampsort, ksack*). Sharing the instruction port in addition to the LLFUs constrains the execution on the lanes. *With more sharing, the impact of explicit hints for convergence seems to be less important.*

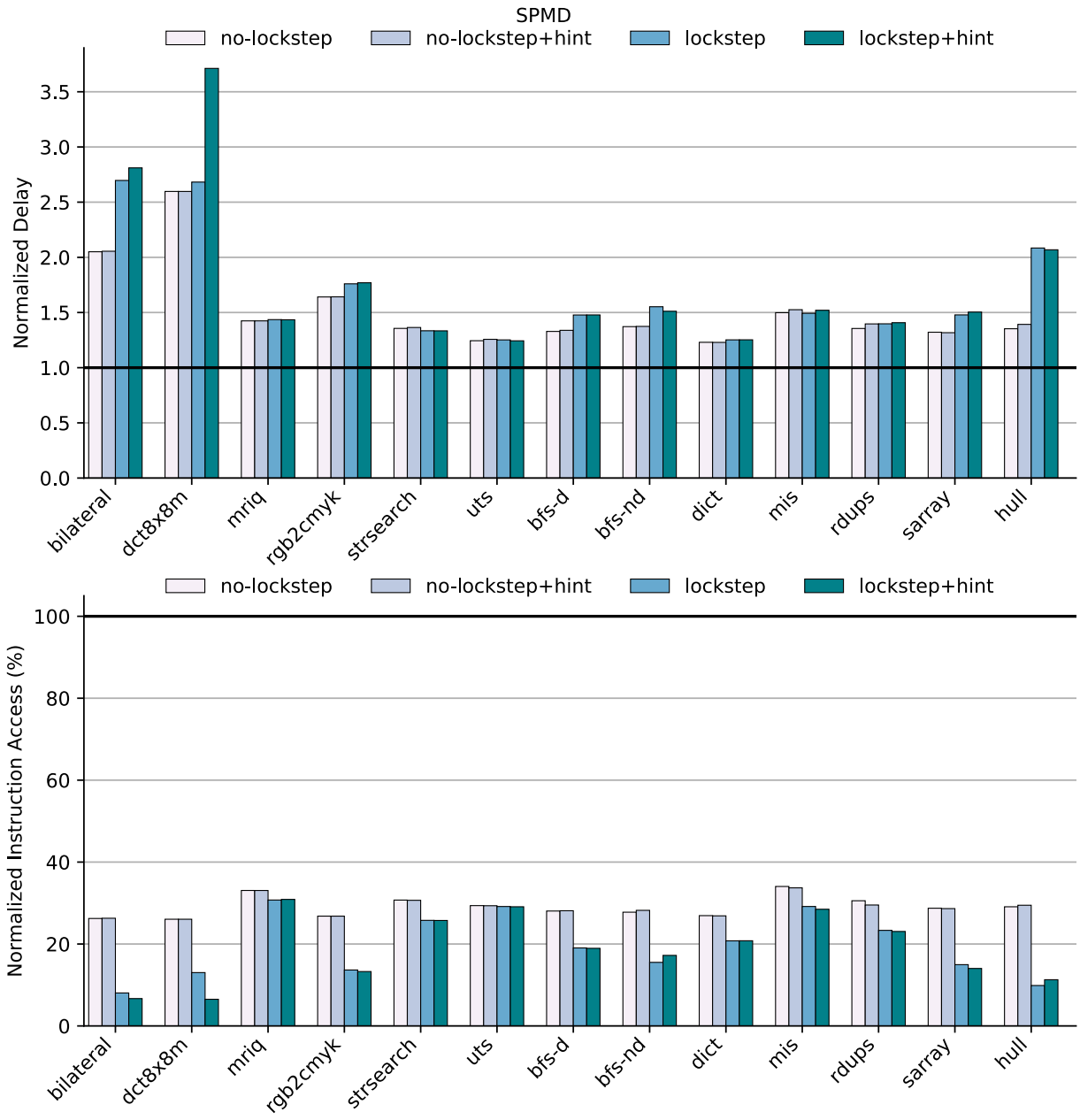


Figure 4.13: SPMD Results for Sharing One Instruction Port and LLFU – The baseline static configuration is provisioned with one L0 buffer and a single instruction-port, data-port, and LLFU with round-robin thread-selection. The *lockstep* execution mechanism trades-off an increase in execution delay for a considerable reduction in instruction accesses. Adding hints is not as important and hurts *dct8x8m* which is the most sensitive to a single LLFU resource. The figure on the top shows the normalized delay and the bottom shows the normalized instruction access compared to the ideal MIMD model (lower the better). Each bar represents the configurations as explained: (i) *no-lockstep* baseline with round-robin arbitration; (ii) *no-lockstep+hint* baseline with soft-barrier hints enabled; (iii) *lockstep* enables the lockstep sharing; (iv) *lockstep+hint* combines lockstep execution and soft-barrier hints.

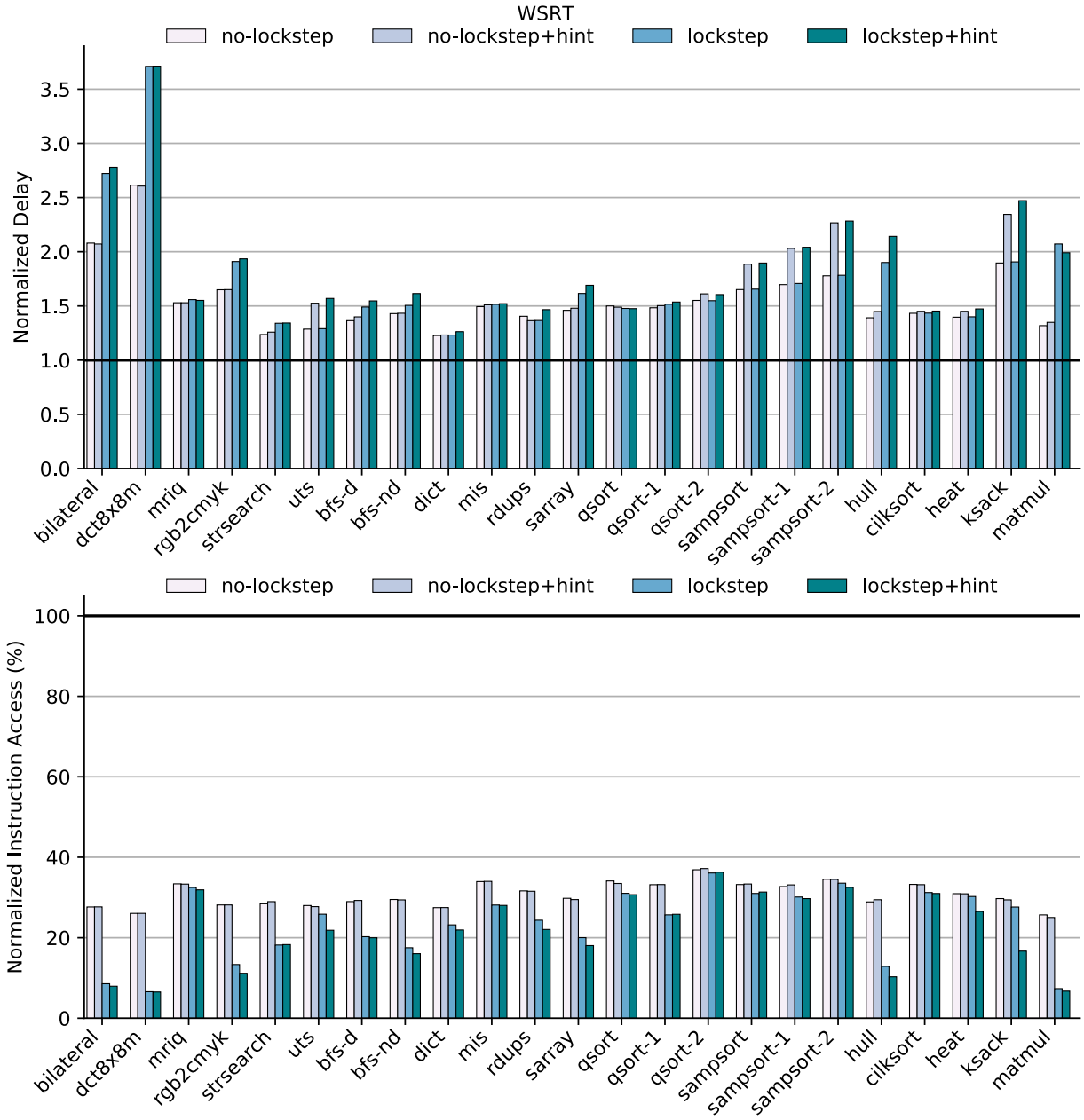
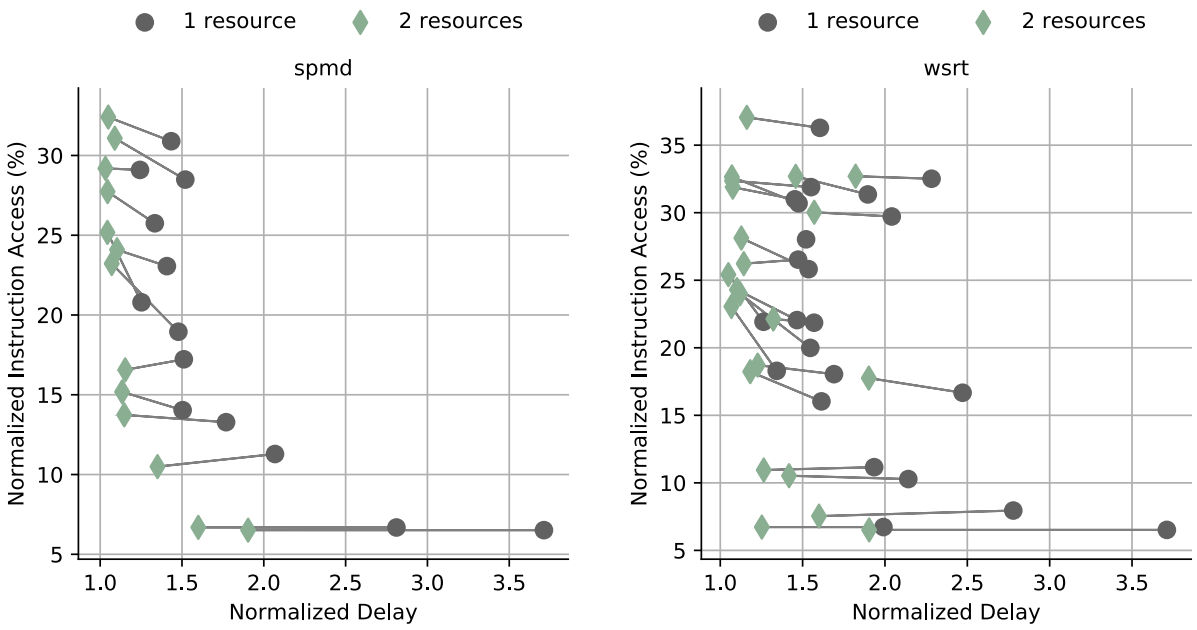


Figure 4.14: WSRT Results for Sharing One Instruction Port and LLFU – The baseline static configuration is provisioned with L0 buffers and a single instruction-port, data-port, and LLFU with round-robin thread-selection. The *lockstep* execution mechanism trades-off an increase in execution delay for a considerable reduction in instruction accesses. Kernels such as *bilateral, dct8x8m, matmul* are extreme examples for pareto-optimality. Adding hints is not as important and hurts *uts, sampsort, ksack* which have regions with low parallelism. The figure on the top shows the normalized delay and the bottom shows the normalized instruction access compared to the ideal MIMD model (lower the better). Each bar represents the configurations as explained: (i) *no-lockstep* baseline with round-robin arbitration; (ii) *no-lockstep+hint* baseline with soft-barrier hints enabled; (iii) *lockstep* enables the lockstep sharing; (iv) *lockstep+hint* combines lockstep execution and soft-barrier hints.



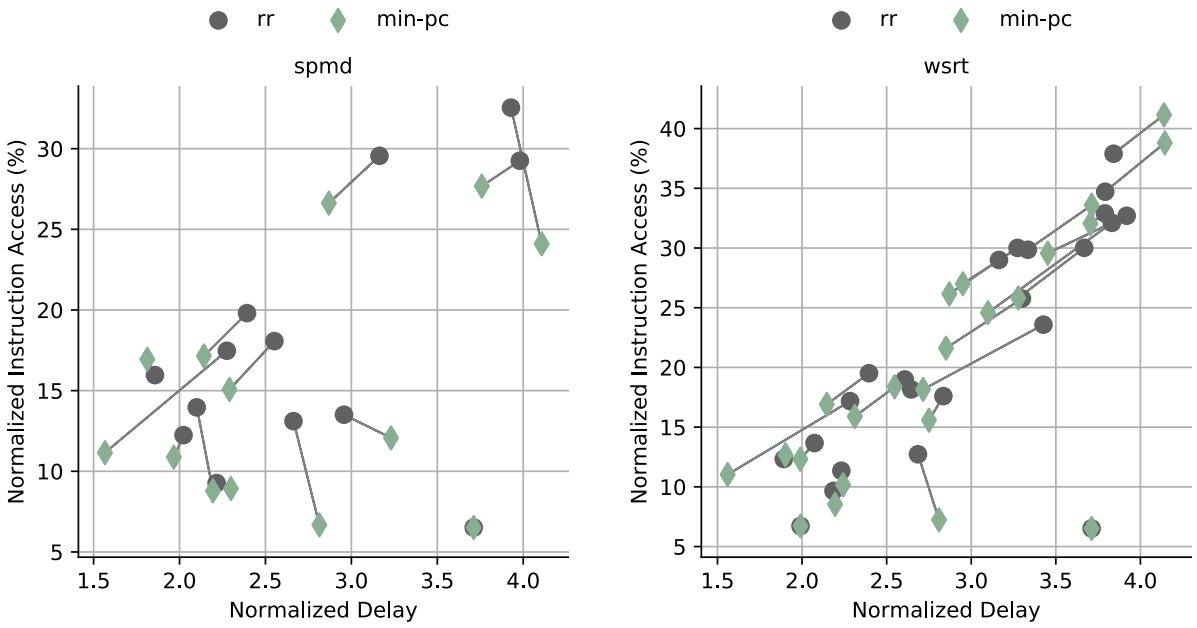
(a) SPMD applications

(b) WSRT applications

Figure 4.15: SPMD and WSRT Results Sharing One vs. Two Instruction Ports and LLFUs – Two instruction ports and LLFUs improve the execution delay for irregular applications and LLFUs for both the SPMD and WSRT applications. The improvements in execution delay for minimal increase in instruction access is as indicated by flat slopes for the lines that connect the points for sharing one instruction port and LLFUs (grey circles) and the points for sharing two instruction ports and LLFUs (green diamonds).

Comparing results for increasing the LLFUs

Figure 4.15 shows the results for increasing the number of instruction and LLFUs. Additional LLFU units in the form of integer multiply-and-divide and floating-point units imply a 22% area overhead. For most SPMD kernels, an additional increase in LLFUs provides an improvement in performance in the range of ≈17% to 34%. *biateral, dct8x8m, and hull* benefit the most in terms of performance which is evident given the application characteristics. Adding an extra LLFU can increase the divergence which results in a slight increase in instruction access as in the case of *bfs-nd, and dict* by about 20%. Regular WSRT kernels such as *bilateral, dct8x8m, hull, and matmul* benefit from an additional instruction and LLFU resource. Most of the PBBS kernels for the WSRT runtime show performance improvements of 20%–35% for an additional increase in area by 22%. These results suggest that for irregular applications a marginal increase in area by increasing LLFU resources makes sense for improvements in performance. Detailed results for increasing the instruction ports and LLFUs are presented in Tables A.7, A.10, A.7, and A.10.



(a) SPMD applications

(b) WSRT applications

Figure 4.16: Comparing Thread-selection for Sharing One Instruction, Front-end and LLFU – The results show that the hybrid minimum-pc/round-robin thread-selection mechanism improves the delay and reduces the instruction access for most of the SPMD and WSRT applications.

Q5. What mechanisms are key to sharing the instruction port, front-end, and the LLFUs?

The design rationale to share the instruction port, front-end, and the LLFUs is based on the observation that sharing the front-end improves the efficiency beyond just reducing the instruction accesses and the fact that the SSA kernels sparingly use the LLFUs. For this design point, the mechanisms applicable are thread-selection and soft-barrier hints.

Result 6 – *The minimum-pc/round-robin thread-selection mechanism is the key to improve performance and efficiency for a design that shares the instruction, front-end, and LLFU.*

Figure 4.16 shows the results for comparing RR vs. MPC thread-selection mechanisms. The diagonal slopes for most of the SPMD and WSRT kernels show that the MPC mechanism is the key to improve efficiency and performance. The SPMD kernels see an improvement in efficiency by 16% for a minimal loss of just 3% on an average. Most of the WSRT kernels see an improvement in efficiency and performance that ranges from 3% to 20% respectively.

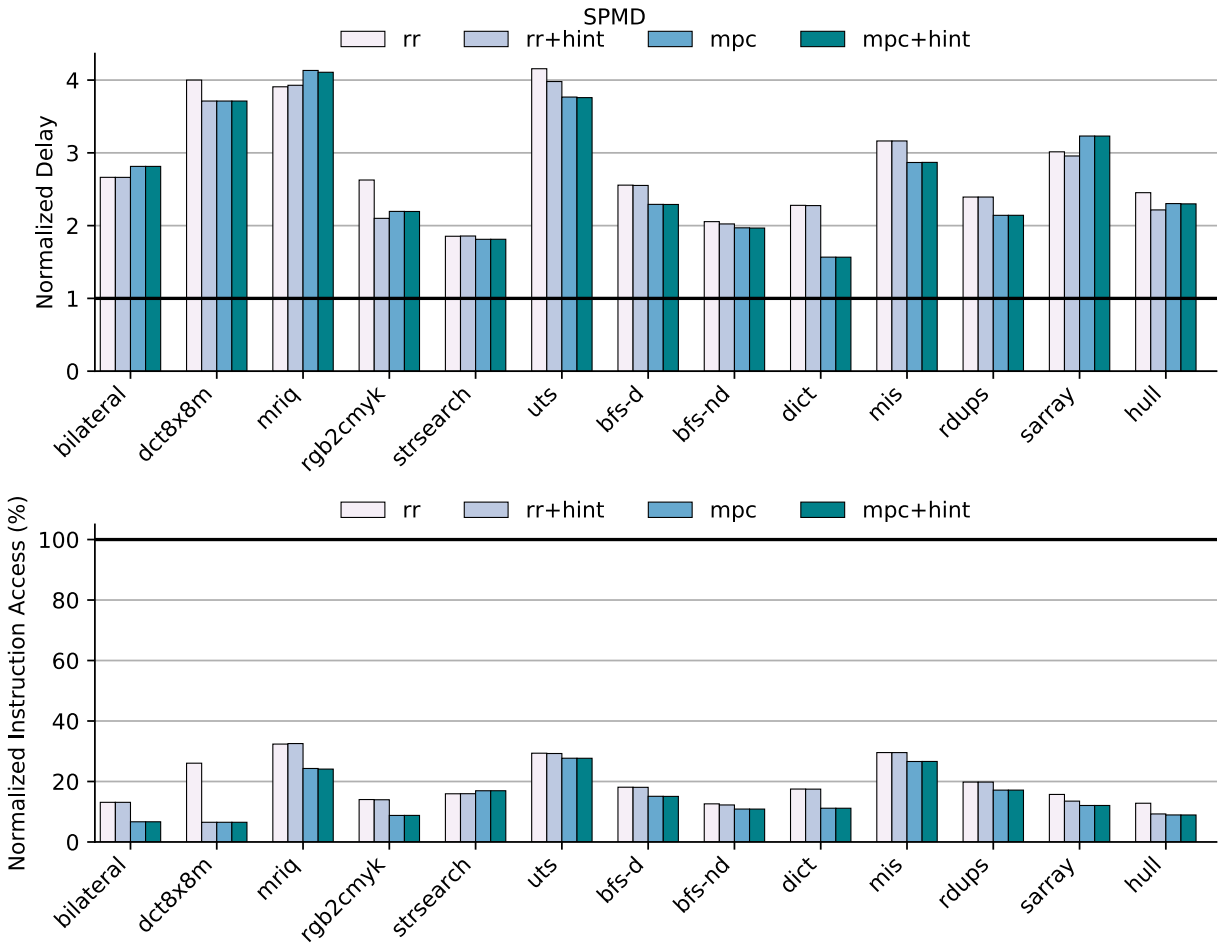


Figure 4.17: SPMD Results for Sharing One Instruction Port, Front-end, and LLFU – The baseline static configuration is provisioned with L0 buffers and a single instruction port, frontend, and the LLFU, the hybrid minimum-pc/round-robin thread-selection mechanism is the key for pareto-optimality. The figure on the top shows the normalized delay and the bottom shows the normalized instruction access compared to the ideal MIMD model (lower the better). Each bar represents the configurations as explained: (i) *rr* baseline with round-robin arbitration; (ii) *rr+hint* baseline with soft-barrier hints enabled; (iii) *mpc* baseline with the hybrid minimum-pc/round-robin thread selection mechanism; (iv) *mpc+hint mpc* configuration with soft-barrier hints.

Are soft-barrier hints important when sharing the instruction port, front-end, and the LLFUs?

Figures 4.17 and 4.18 show that with an increased amount of sharing the hints do not impact the performance and efficiency. Reduced resources affect the dynamic schedule of the instructions executing on the lanes. A unified front-end and the MPC thread-selection mechanism are the only required mechanisms for this design point.

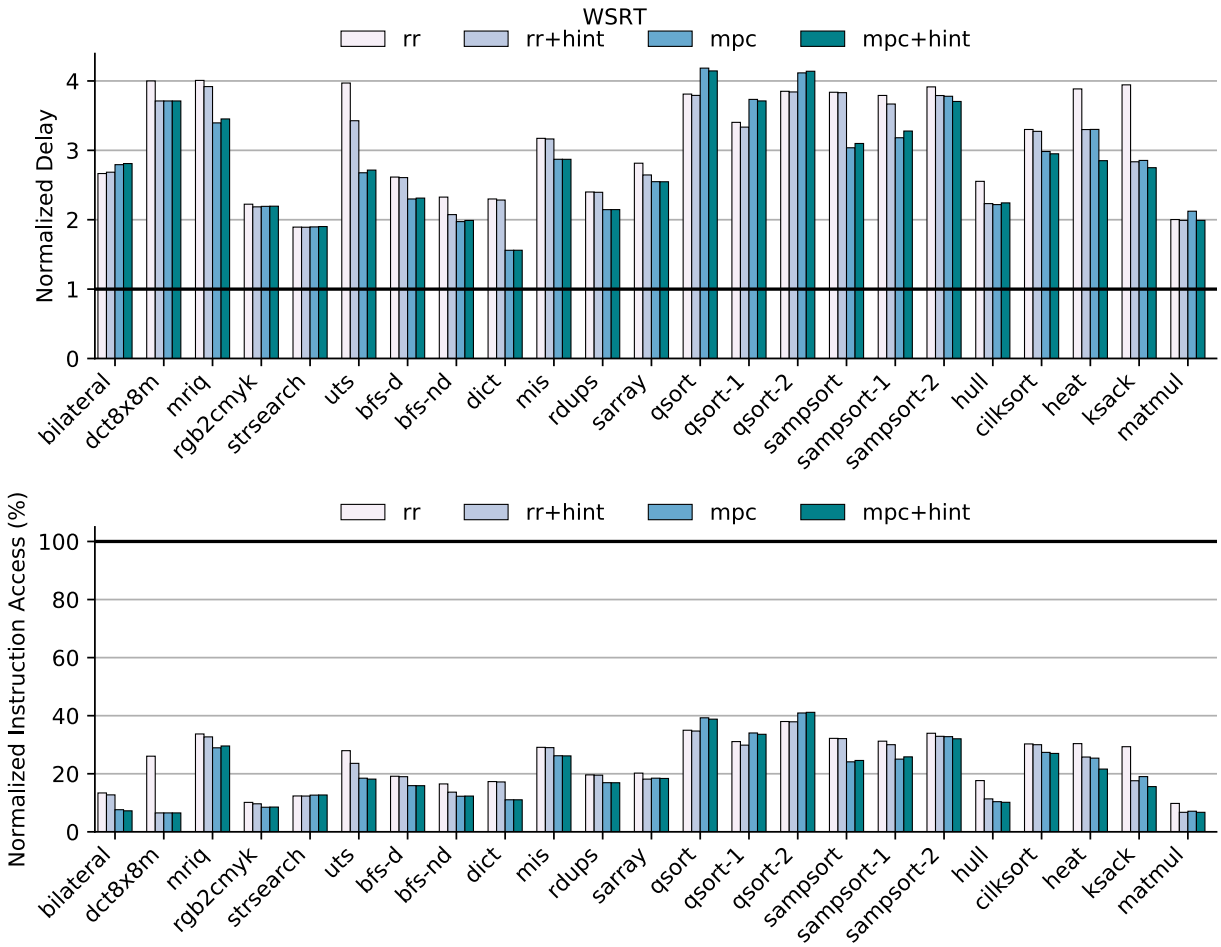
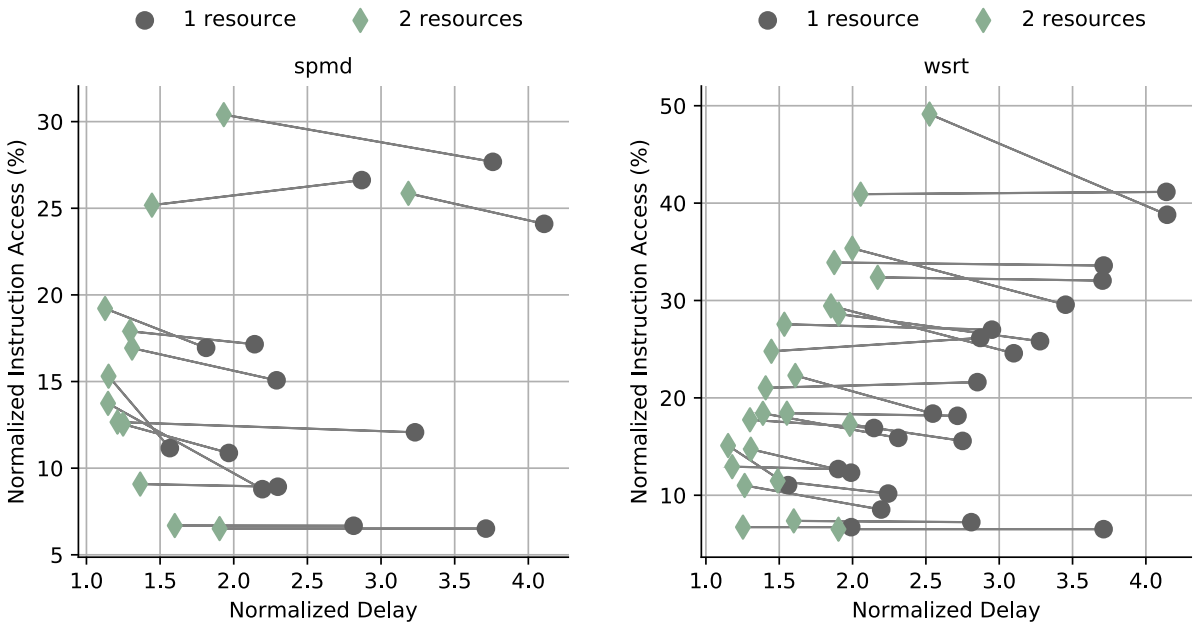


Figure 4.18: WSRT Results for Sharing One Instruction Port, Front-end, and LLFU – The results show that for a baseline static configuration with L0 buffers and a single instruction port, frontend, and the LLFU, the hybrid minimum-pc/round-robin thread-selection mechanism is the key for pareto-optimality. The figure on the top shows the normalized delay and the bottom shows the normalized instruction access compared to the ideal MIMD model (lower the better). Each bar represents the configurations as explained: (i) *rr* baseline with round-robin arbitration; (ii) *rr+hint* baseline with soft-barrier hints enabled; (iii) *mpc* baseline with the hybrid minimum-pc/round-robin thread selection mechanism; (iv) *mpc+hint mpc* configuration with soft-barrier hints.

Comparing results for increasing the front-end and LLFUs

Figure 4.19 shows the results of increasing the resources for the *sharing imem+fe+llfu* design point. Both the SPMD and the WSRT kernels observe an improvement in performance by 30% on an average for a minimal decrease in efficiency of 6%. Increasing the front-end and the LLFU resources is complexity-effective when compared to the sharing a single front-end and a single LLFU.



(a) SPMD applications

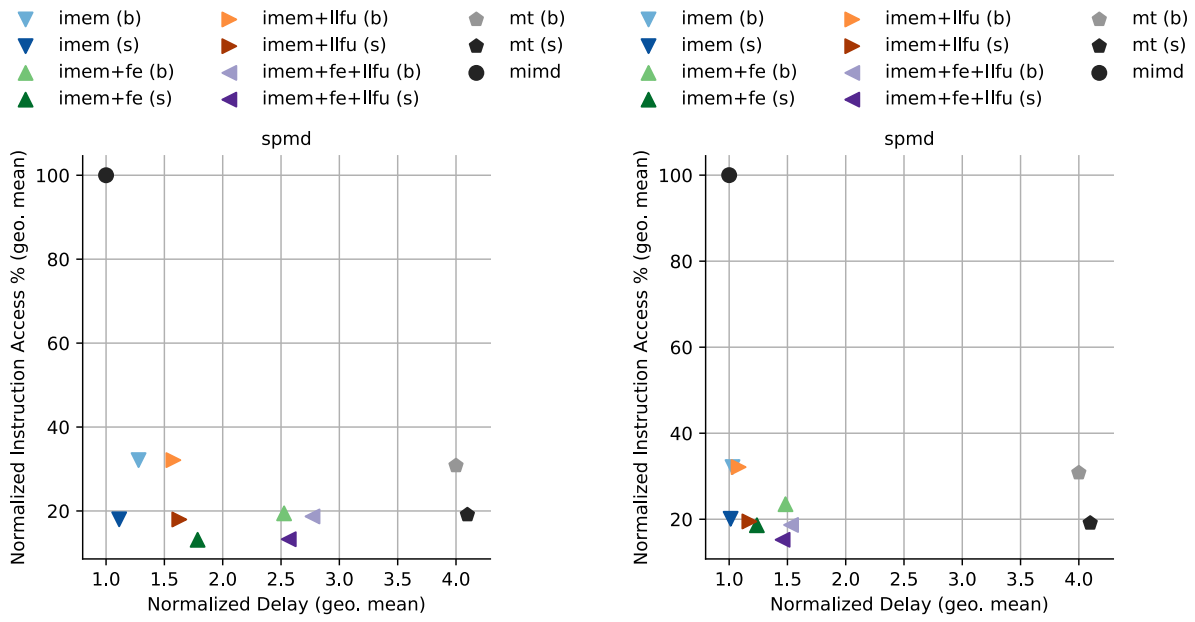
(b) WSRT applications

Figure 4.19: SPMD and WSRT Results Sharing One vs. Two Instruction Ports, Front-ends and LLFUs – Two resources improve the execution delay for minimal changes in instruction accesses as indicated by the flat slopes for the lines that connect the points for sharing one instruction resource (grey circles) and the points for sharing two instruction resources (green diamonds).

Q6. How do the SSA design points compare?

We motivated the SSA design space with the premise that there is a continuum of design points between sharing no resources and sharing all resources. Moving from the left to right in Figure 4.3 saves area by sharing more resources but trades off performance and efficiency. In this section, we present results for comparing the SSA designs.

Figures 4.20 and 4.21 compare the SSA design points for SPMD and WSRT kernels. In these plots, we compare the baseline designs which naively share hardware resources to designs that employ smart sharing mechanisms that yield the best results. All the designs use an L0 buffer for instruction bandwidth amplification. We include the *mimd* design point that represents no sharing as well as the *mt* design point that represents sharing all the resources. The baseline naive sharing, indicated by *b*, as well as the alternative designs that employ smart sharing mechanisms, indicated by *s*, as below.

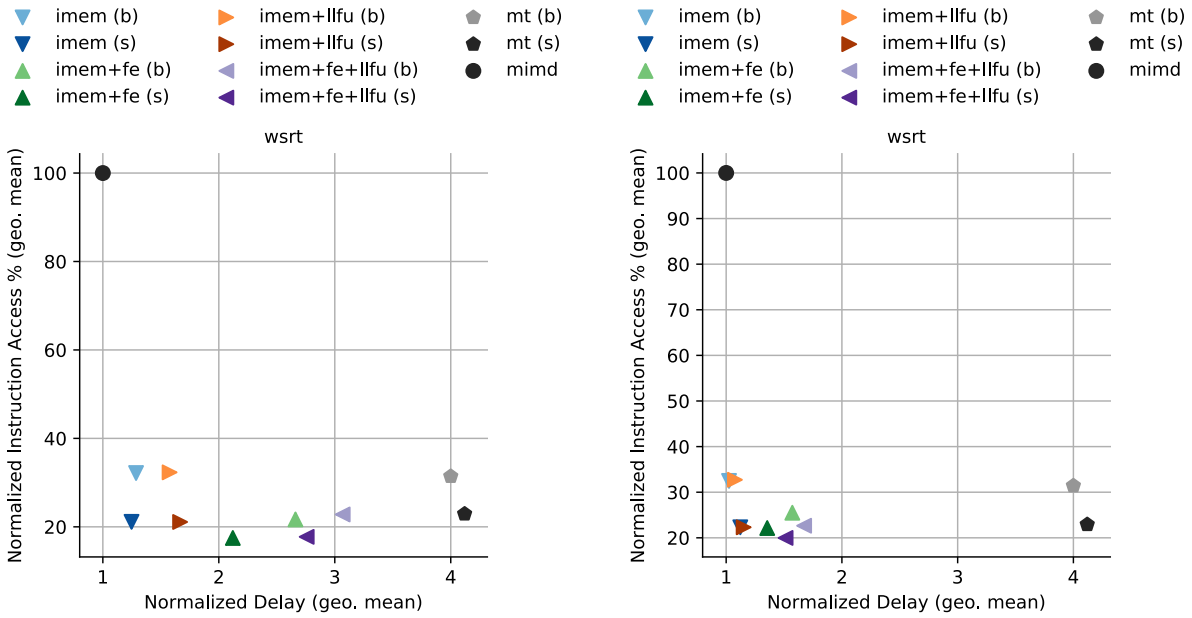


(a) SPMD Kernels using 1 Resource

(b) SPMD Kernels using 2 Resources

Figure 4.20: Comparing SSA Designs for SPMD Kernels – Smart sharing mechanisms are the key to improve efficiency and performance in SSA designs. The lighter colors represent the baseline designs (b) for each design point and the darker colors respresent the smart-sharing designs (s). The plot on the left shows the results with one resource and results for two resources are shown to the right.

- *imem (b)* baseline design represents the *sharing imem only* design which employs the round-robin arbitration to share a static number of instruction ports. The *imem (s)* shows the same design using instruction coalescing and soft-barrier hints smart sharing mechanisms.
- *imem+fe (b)* baseline design represents the *sharing imem+fe only* design which employs the round-robin thread-selection mechanism to share the instruction port and the front-end. The *imem+fe (s)* shows the same design that uses minimum-pc/round-robin thread-selection mechanism.
- *imem+llfu (b)* baseline design represents the *sharing imem+llfu only* design which employs the round-robin arbitration to share a static number of instruction and LLFU resources. The *imem+llfu (s)* shows the design that uses instruction coalescing and the lockstep sharing mechanisms.
- *imem+fe+llfu (b)* baseline design represents the *sharing imem+fe+llfu only* designs which employs the round-robin thread-selection mechanism to share the instruction port, front-



(a) WSRT Kernels using 1 Resource

(b) WSRT Kernels using 2 Resources

Figure 4.21: Comparing SSA Designs for WSRT Kernels – Smart sharing mechanisms are the key to improve efficiency and performance in SSA designs. The lighter colors represent the baseline designs (b) for each design point and the darker colors respresent the smart-sharing designs (s). The plot on the left shows the results with one resource and results for two resources are shown to the right.

end, and the LLFUs. The *imem+fe+llfu (s)* shows the same design that uses the minimum-pc/round-robin thread-selection mechanism.

- *mt (b)* represents the design where all resources are shared using round-robin based thread-selection mechanism. The *mt (s)* design represents the design that uses the minimum-pc/round-robin thread-selection mechanism.

To compare the designs, we compute the geometric mean for the normalized delay and normalized instruction accesses across all the applications on each design point. The results are normalized to the *mimd* design where no resources are shared. We include results that compare the designs sharing a single resource as well as two resources.

Figures 4.20(a) and 4.21(a) show results for SPMD and WSRT kernels with a single resource being shared. The results show the following:

- Sharing only the instruction port (*imem*) is 20% worse in performance and saves nearly 79% of the instruction accesses compared to the ideal *mimd* design. Sharing an instruction cache

results in significant area savings compared to CMP designs with modest loss in performance for significant gains in efficiency.

- The *imem+llfu* design performs better than *imem+fe* with very similar efficiencies. For fork-join-centric parallel programs that are irregular, exclusive front-end designs utilize the back-end resources better than having a fixed front-end. The LLFUs are not used heavily in kernels that are in the domain of graph processing, text processing, and search/optimization. Sharing the LLFUs is a viable design point. Lockstep sharing results in a minimal 5% loss in performance with close to 40% improvements in efficiency when compared to the baseline design (*imem+llfu (b)*).
- The *imem+fe+llfu* design performs better than *mt* designs with nearly same efficiencies. This implies that the area costs of adding additional front-ends and short-latency units are justified when compared to simple time-multiplexing of all the resources. The additional cost of decoder/issue logic and ALUs result in a complexity-effective solution.
- The minimum-pc/round-robin thread-selection mechanism helps the *mt* design to further save 30% of the instruction accesses when compared to simple round-robin based multiplexing.
- SSA designs indeed line-up on a continuum of pareto-optimality offering a great degree of freedom to the designer in saving area costs while balancing the impact on performance and improving efficiency.

Figures 4.20(b) and 4.21(b) show results for SPMD and WSRT kernels with two resources being shared. The results show the following:

- Designs that share only the instruction port (*imem*) observe only an additional 10% improvement in performance for a considerable increase in area. A single port is sufficient when using L0 buffers, instruction coalescing, and soft-barrier hints.
- Designs that share a single front-end *imem+fe* gain an additional 50% improvement in performance for nearly same efficiency. Fork-join-centric parallel programs that are irregular adding an additional front-end helps improve the utilization of the resources but the performance is still slightly worse than designs that do not share the front-end.
- Designs that share the instruction ports and the LLFUs gain an additional 30% improvement in performance for nearly same efficiency. The performance of *imem+llfu* design is very

similar to *imem* which means that reducing the number of LLFUs is an area and performance-efficient solution.

- Adding additional resources to *imem+fe+llfu* designs further improves the performance by about 45% for nearly same efficiency. It is interesting to note that the gap in performance and efficiency between naive sharing and smart sharing reduces considerably when sharing two resources.

The results presented above discusses pareto-optimality of the SSA designs considering the normalized instruction accesses and the normalized delay dimensions. Future work should explore the pareto-optimality of the SSA designs by considering an addition dimension of the area costs of the designs. Factoring in the area would better compare the SSA designs and help in evaluating the complexity-effectiveness of smart sharing mechanisms.

4.5 Related Work

In this section, we discuss related work for SSAs. Most of the previous work on exploiting instruction redundancy has focused on SPMD application kernels executing on modified SMT architectures. To the best of our knowledge, we are the first to consider exploiting instruction redundancy in WSRT runtimes.

Kumar et al. proposed the idea of sharing resources in conjoined-core chip multiprocessing [KJT04]. Conjoined-cores proposes the idea of sharing the L1 caches, FPU, and cross-bar ports for two adjacent cores. The lanes in SSAs are lightweight in-order cores compared to "full-fledged" cores as in conjoined-cores. The SSA design space systematically varies each resource as compared to conjoined-cores. The idea of *fetch combining* in conjoined-cores is similar to instruction coalescing.

Thread Fusion [GCC$^+$08] proposes to *fuse* two instances of the same instruction in a 2-way SMT processor into a single instruction which results in reducing the resource usage of the front-end pipeline stage by half. The fused instruction is treated as two separate instructions in the back-end. Thread fusion assumes SPMD applications with parallel loops and relies on compiler inserted synchronization hints. The idea of soft-barrier hints is similar to the compiler generated hints although in SSAs the applications are unmodified with minimal changes only to the runtime.

Minimal Multithreading (MMT) [LFB+10] improves the techniques proposed in Thread Fusion by removing the compiler generated hints. In MMT, the authors measure redundancy and motivate two opportunities by identifying *fetch-identical* SMT threads which fetch instructions from the same address and *execute-identical* threads which further have the same operands for the fetched instructions. Fetch-identical threads eliminate inefficiencies in the front-end and the execute-identical threads eliminate back-end inefficiencies. The synchronization between threads is achieved by using a *fetch history buffer* which records branches for of the executing threads. If a thread finds an instruction in a different thread's history buffer for a branch, then the executing thread is given higher priority in *CATCHUP* mode. The fetch history buffer and the associated logic is very expensive compared to the simple minimum-pc/round-robin based thread-selection mechanism as implemented in SSAs. Execute-identical redundancy can be exploited in SSA designs which have a shared front-end but we believe that the possibilities to exploit such opportunities in other designs with exclusive front-ends is quite difficult to implement across the lanes.

Multithreaded-Instruction Sharing (MIS) [DFR10] is another proposal that exploits execute-identical threads or *value similarity*. In MIS, the instructions that are execute-identical are retired without execution on a 2-way SMT processor. MIS uses a *match table* to identify execute-identical threads by recording the source operands and the results of previous instruction executions.

In Execution drafting [MBW14], the authors propose modifications to an SMT processor to execute identical instructions from different programs or threads, such that they flow down the pipe consecutively or draft. The goal is to reduce energy by reducing switching in the pipeline. A redundant instruction can be either partial or fully redundant. The partial duplicate or redundant instruction is one which has the same opcode but has different machine code. The authors use a hybrid thread synchronization method which uses minimum-pc and random thread-selection schemes. Execution drafting mainly targets data center workloads unlike the multithreaded workloads we consider in SSAs.

In [MCM+14], the authors compare a variety of thread-scheduling heuristics with the goals of reconvergence in SPMD programs. The authors compare the minimum-pc thread-selection, the heuristic proposed in [LFB+10], and the mechanism used in [LAB+12] and propose a minimum-sp/pc heuristic that prioritizes threads with a lower stack pointer. The minimum-sp/pc heuristic works well for loops with iterations that have nested function calls. The methods surveyed in this

paper are categorized as implicit reconvergence techniques. Similar heuristics can be applied to SSAs with a key requirement of guaranteed forward progress.

DITVA [KCSS16] is a recent proposal that presents the idea of "dynamic vectorization" of SPMD programs on a SIMD-enabled SMT processor. DITVA utilizes SIMD units to execute instructions that are identical in a SPMD program. DITVA uses a hybrid minimum-sp/pc round-robin heuristic inspired by [MCM+14]. We implemented the same heuristic in SSAs, and we find that while some SPMD kernels show improvements in efficiency, the WSRT kernels do not observe any improvements in efficiency over the minimum-pc/round-robin thread-selection scheme. For WSRT kernels, the worker loop can be deeply nested and prioritizing thread using the minimum stack pointer heuristic hurts the goals of dynamic load-balancing. SSAs do not assume a SIMD-enabled processor and are much simpler. We believe SSAs are complexity-effective compared to the modifications to a complex SIMD-enabled SMT processor as required in DITVA.

LTA [KJT+17] is a recent proposal that proposes to execute the parallel for loops expressed using a fork-join runtime. The key idea is to use the host GPP processor for the recursive division of a loop range object, and the base case for the loop which executes the user tasks are offloaded to an efficient *loop task accelerator* using a lightweight hint instruction. LTA can be configured at design time for different amounts of spatial/temporal decoupling to efficiently execute both regular and irregular loop tasks. SSA uses spatially decoupled lanes to execute generalized fork-join programs. Unlike LTA, the base case loop is executed serially on a single lane that grabs the base case. Exploring the temporal dimension for coupling on SSAs inspired by LTAs is a promising future direction.

4.6 Conclusions

This chapter presented a novel design space for smart sharing architectures. Sharing hardware resources is an attractive solution to build lane-based BSAs that execute challenging fork-join programs that otherwise have to execute on CMPs. SSAs employ complexity-effective smart sharing mechanisms to improve the efficiency and mitigate performance loss when sharing resources. We presented four smart sharing mechanisms that exploit instruction redundancies which include *instruction coalescing, soft-barrier hints, prioritized thread-selection, and lockstep sharing*. Contrary to conventional wisdom, we show that there is sufficient instruction redundancy in

work-stealing runtimes. We compared the results for SSAs executing application kernels that are drawn from the PBBS, Cilk, and an in-house benchmark suite. While we focused primarily on the instruction fetch redundancy, there are further opportunities to exploit redundancy in the operand values and data accesses. SSAs present a continuum of design points for a designer to choose from while balancing the area costs, efficiency, and performance.

CHAPTER 5
CONCLUSION

Technology constraints have driven computer architects to embrace parallelization and hardware specialization across the layers of computing stack. Providing clean hardware/software abstractions that are highly programmable yet still enable efficient execution on both traditional and specialized platforms is a key research challenge. The choice of hardware specialization not only affects the software stack but also fundamentally affects the cost of design and verification. This thesis presents a lane-based hardware specialization approach to build programmable accelerators for loop- and fork-join-centric parallel programs. The techniques presented in this thesis require lightweight changes to applications, compilers, instruction sets, and microarchitectures which reduces the barrier of adoption.

5.1 Thesis Summary and Contributions

This thesis began by discussing the trends in technology scaling, computer architecture, and hardware specialization. Hardware specialization can range from fixed-function hardware to programmable chip multiprocessors. I presented a taxonomy for hardware specialization and presented a systematic approach to building lane-based behavior specialized accelerators. A parallelization strategy decomposes an application into logical units of parallelism, and a scheduling strategy maps the logical units of parallelism onto the underlying hardware resources. Mismatches between parallel behaviors and the underlying hardware increase the complexity of hardware specialization. CGRA-based and lane-based BSAs are two attractive options for hardware specialization solutions. Given the benefits of programmability, flexibility, and design costs, I presented a case for lane-based BSAs. The vision for the accelerator platform in this thesis includes a heterogeneous CMP platform that composes tiles with GPPs and tiles with GPPs that are augmented with lane-based BSAs. The remainder of this thesis discussed two novel lane-based BSAs that focus on executing loop- and fork-join-centric parallel programs.

The XLOOPS proposal is a new hardware specialization approach for exploiting inter-iteration loop dependence patterns. The XLOOPS instruction set provides an elegant hardware/software abstraction that serves as an effective compiler target and enables a variety of microarchitectures supporting traditional, specialized, and adaptive execution. We have used a vertically integrated

evaluation methodology spanning applications, compilers, cycle-level modeling, RTL modeling, and VLSI implementation to make a compelling case for augmenting both in-order and out-of-order general-purpose processors with a loop-pattern specialization unit. We also implemented an initial FPGA prototype that adds credibility to the XLOOPS proposal.

The SSAs proposal is a new approach to building lane-based BSAs which can efficiently support fork-join-centric parallel programs. Executing fork-join-centric parallel programs on lane-based accelerators is challenging and is relatively less explored. In the SSAs proposal, I presented a rich design space for lane-based BSAs that can share hardware resources to varying degrees and thereby reduce the area and static power consumption. I presented four complexity-effective smart sharing mechanisms that include: instruction coalescing, soft-barrier hints, prioritized thread-selection, and lockstep resource sharing. Smart sharing mechanisms exploit instruction redundancy to maximize efficiency, and improve performance of fork-join-centric parallel programs. I presented a novel evaluation methodology that fundamentally explores the interactions between application characteristics and shared resource constraints.

The primary contributions of this thesis are reiterated as below:

- I make the case for single-ISA heterogeneous platforms that transparently integrate traditional general-purpose processors and lane-based BSAs to improve the performance and energy efficiency of loop- and fork-join-centric parallel programs.
- I propose an elegant new XLOOPS hardware/software abstraction that explicitly encodes inter-iteration loop dependence patterns that execute on traditional, specialized, and adaptive microarchitectures; I also propose a novel XLOOPS microarchitecture that augments a general-purpose processor with a lane pattern specialization unit to execute the XLOOPS binaries.
- I propose smart sharing architectures, a new approach that employs complexity-effective smart sharing mechanisms to exploit instruction redundancy in fork-join-centric parallel programs to save area while maximizing efficiency and minimizing performance losses.

5.2 Future Work

The vision of lane-based BSA platforms, which include XLOOPS and SSAs, are a first step towards *generalized* hardware specialization. This section discusses opportunities to extend the ideas presented in the thesis.

Exploring stream-centric parallelization and scheduling strategies – Stream-centric parallelization decomposes an application into data streams and computational kernels. The data stream abstraction provides guidelines to systematically approach hardware specialization for memory access patterns and the kernels can be mapped to lane-based BSAs. The operations on data streams for a domain or commonly accessed patterns can use specialized hardware that can prefetch and manage memory regions which can greatly reduce the energy overheads and improve the performance. The stream-centric parallelization and scheduling strategies have been explored in the past but utilize simple in-order cores in a CMP platform to execute the computation kernels. Employing lanes as opposed to cores can reduce the area costs of these solutions.

Unified lane-based accelerators – The XLOOPS and SSAs proposals use simple in-order lanes that are augmented with additional hardware to manage inter-iteration dependences, as in the case of XLOOPS, or manage shared resources, as in the case of SSAs. Future work can explore implementing a programmable lane-based BSA that can handle either loop- or fork-join-centric parallel programs. Such an accelerator can trade the area costs of implementing multiple lane-based BSAs and also reduce the costs of design.

Handling nested loops and data-dependent exits – The LPSU in XLOOPS can currently be configured to execute a single loop in a loop nest with multiple loops. Future work can explore an LPSU design that can elegantly parallelize execution of loop nest with multiple loops. Executing multiple loops requires the loop bodies to be loaded into the loop buffers and tagged with the nested level. The mechanisms that include the cross-iteration buffers and speculative stores/loads need to be virtualized across the loop nests. Extending the LPSU to handle loops that include a data-dependent exit condition, as found in while loops, further increases the applicability of the XLOOPS accelerator. Handling data-dependent exit condition needs control speculation as in the case of dynamic-bound for loops.

Parallelizing base-cases for fork-join-centric parallel programs – Loop-based programs that are parallelized by using a work-stealing runtime transform the *parallel for* under the hood

into recursive task-based programs. The loop is transformed to a recursive task-based program where the tasks that execute an inductive-case simply divide the loop-range, and tasks that execute the base-case execute a serial for-loop for a reduced subset of the loop-range. Currently, each lane serially executes the entire loop of the base-case. Future work can explore parallelizing the base-case by using a unified lane-based BSA. The fork-join based division of the loop-range is scalable across tiles, and further improvements in performance and efficiency can be achieved by using the lanes for the execution of the base-case.

Exploiting data and value redundancy – The SSAs proposal mainly focuses on improving the efficiency of a fork-join-centric parallel program by exploiting the instruction redundancy to reduce the number of instruction accesses. While we enable coalescing for data accesses across the lanes, we observed minimal benefits from data coalescing. Initial exploration suggests that a co-designed software runtime that is aware of a lane-based BSA can transform the loop-ranges and schedule tasks onto the hardware such that the potential benefits from data coalescing increase. Such a runtime would likely use different policies to manage the tasks in the task queue for stealing and local dequeues based on locality of working sets. Value redundancy occurs when identical operations across the lanes have the same value for the operands. In such cases, fetching and executing an instruction once and broadcasting results to all the lanes can further improve performance and efficiency. Value redundancy is fundamentally present in applications due to book-keeping overheads and in domains such as image and audio processing which quantize the range of input values.

Exploring value memoization – Memoization is an optimization technique that improves performance and efficiency by storing the results of expensive functions calls. When a *memoized* function is executed with the same inputs, a simple lookup for the stored result can replace expensive computation. In kernels such as *mriq*, the computational loop is very regular and has no data-dependent control-flow but for the use of transcendental function calls. The implementation of most of these functions includes irregular loops with data-dependent exit conditions. Memoization in hardware for such function calls can greatly improve the performance and energy efficiency for not just the function calls but for the code around these calls which is otherwise regular.

Exploring multi-tile SSA designs – The evaluation of SSAs focused on the single-tile SSAs primarily to understand the interaction of application characteristics and smart sharing mechanisms. Future work can explore the scalability of the software work-stealing runtime on multi-tile

SSA designs. The conjoined-lanes increase the parallelism within a tile while incurring minimal area costs when compared to scaling GPP tiles. Comparing the performance of baseline CMP designs and multi-tile SSA designs for a given amount of area would be interesting. Conjoined-lanes incur low communication and synchronization overheads as lanes within a tile share the data cache. Co-designed software runtimes that are aware of conjoined-lanes can better exploit the memory access patterns both within a tile and across a tile, potentially reducing coherency traffic.

Integrating DSAs and BSAs – A BSA platform provides a programmable template to integrate both lane-based accelerators and other domain-specific accelerators. Inclusion of the tensor cores in NVIDIA Volta architecture is an example of such a platform. The emergence of important workloads such as machine-learning and graph processing justifies specialized hardware for these domains. Integrating such DSAs into a programmable BSA-based platform simplifies the software challenges and can potentially generalize the solution. Efficient execution on the lanes combined with domain-specialized functions while maintaining clean hardware/software abstractions is a promising future direction.

APPENDIX A
DETAILED SSA RESULTS

Name	SPMD	WSRT
bilateral	3.90	3.91
dct8x8m	4.00	4.00
mriq	3.82	3.81
rgb2cmyk	3.98	3.98
strsearch	3.88	3.88
uts	2.62	4.02
bfs-d	2.52	2.44
bfs-nd	1.56	1.53
dict	3.50	3.49
mis	1.86	1.86
rdups	2.75	2.74
sarray	2.67	3.14
hull	3.27	3.19
qsort		2.62
qsort-1		2.14
qsort-2		3.05
sampsort		1.77
sampsort-1		1.80
sampsort-2		1.10
cilksort		3.94
heat		3.88
ksack		2.34
matmul		3.96

Table A.1: Speedups for *no sharing* Design – Speedup for the *no sharing* design that executes application kernels using four lanes compared to the execution with a single thread.

Name	SPMD	WSRT
bilateral	1.00	1.00
dct8x8m	1.00	1.00
mriq	1.00	1.00
rgb2cmyk	1.00	1.00
strsearch	1.00	1.00
uts	1.00	1.27
bfs-d	1.00	1.04
bfs-nd	1.00	1.02
dict	1.00	1.00
mis	1.00	1.00
rdups	1.00	1.00
sarray	1.00	1.02
hull	1.00	1.05
qsort		1.01
qsort-1		1.03
qsort-2		1.11
sampsort		1.37
sampsort-1		1.46
sampsort-2		1.72
cilksort		1.03
heat		1.08
ksack		1.57
matmul		1.00

Table A.2: Instruction Redundancy Performance Overheads – Normalized performance for the *no sharing* design point that executes application kernels instrumented with soft-barrier hints compared to the same design when executing without any soft-barrier hints. *no sharing* design with soft-barrier hints enables the measurements for instruction redundancy.

	base		coalesce		coalesce hints		l0		l0 coalesce		l0 coalesce+hint	
N=1	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	4.00	100.00	1.12	28.08	1.01	25.29	1.06	26.41	1.00	8.32	1.00	8.32
dct8x8m	4.00	100.00	1.00	25.00	1.00	25.00	1.04	26.05	1.00	6.51	1.00	6.51
mriq	4.00	100.00	3.75	90.40	3.71	90.49	1.28	33.38	1.26	32.07	1.24	32.06
rgb2cmyk	4.00	100.00	1.34	33.45	1.34	33.55	1.11	27.93	1.04	13.95	1.04	13.94
strsearch	4.00	100.00	1.48	35.31	1.48	35.34	1.71	42.60	1.13	22.34	1.13	22.37
uts	4.00	100.00	4.12	99.01	3.90	98.96	1.16	29.34	1.18	29.11	1.19	29.08
bfs-d	4.00	100.00	2.01	49.53	2.01	49.48	1.23	30.68	1.09	22.92	1.09	22.18
bfs-nd	4.00	100.00	1.48	36.28	1.47	36.18	1.26	31.53	1.11	23.31	1.11	20.64
dict	4.00	100.00	1.63	40.57	1.62	40.50	1.29	32.34	1.11	23.61	1.11	21.35
mis	4.00	100.00	2.53	63.32	2.53	63.27	1.54	38.57	1.29	29.32	1.29	28.64
rdups	4.00	100.00	1.89	47.22	1.89	47.21	1.49	37.18	1.19	25.51	1.19	23.79
sarray	4.00	100.00	2.49	40.93	2.50	39.45	1.26	31.74	1.10	20.11	1.08	14.90
hull	4.00	100.00	1.14	26.27	1.14	26.20	1.36	33.81	1.04	16.85	1.03	13.39
N=2	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	2.00	100.00	1.00	50.10	1.00	48.20	1.01	26.41	1.00	12.64	1.00	13.22
dct8x8m	2.00	100.00	1.00	36.63	1.00	25.00	1.00	26.05	1.00	13.03	1.00	6.51
mriq	2.00	100.00	1.88	91.73	1.87	91.29	1.02	33.38	1.02	32.70	1.02	32.70
rgb2cmyk	2.00	100.00	1.06	47.52	1.00	46.73	1.01	27.93	1.00	14.69	1.00	13.74
strsearch	2.00	100.00	1.08	47.76	1.08	47.73	1.08	42.60	1.01	29.97	1.01	29.96
uts	2.00	100.00	1.84	99.20	1.80	98.96	1.03	29.45	1.00	29.31	1.05	29.16
bfs-d	2.00	100.00	1.23	58.50	1.23	56.88	1.02	30.68	1.01	25.89	1.01	23.81
bfs-nd	2.00	100.00	1.10	49.78	1.10	45.22	1.01	31.53	1.00	24.82	1.01	21.04
dict	2.00	100.00	1.15	55.69	1.14	52.06	1.02	32.34	1.01	27.76	1.01	25.44
mis	2.00	100.00	1.39	67.35	1.39	66.07	1.06	38.57	1.03	33.42	1.03	31.41
rdups	2.00	100.00	1.23	57.85	1.23	51.83	1.05	37.18	1.02	27.41	1.02	24.02
sarray	2.00	100.00	1.26	50.89	1.10	43.24	1.05	31.74	1.00	24.84	1.00	17.09
hull	2.00	100.00	1.02	40.18	1.02	34.34	1.02	33.81	1.00	25.91	1.01	15.75

Table A.3: SPMD Results for Sharing the Instruction Port Only – SPMD results for *sharing imem only* (CL-NI-4F-4L) design point. *base* = baseline with round-robin arbitration; *coalesce* = baseline with instruction coalescing enabled; *coalescing+hint* = combining coalescing with soft-barrier hints; *l0* = adding a L0 buffer to each lane; *l0+coalesce* = combining L0 buffers with instruction coalescing; *l0+coalesce+hint* = combining L0 buffer, instruction coalescing, and soft-barrier hints; *P* = Normalized delay; *IA* = Normalized instruction access

	base		coalesce		coalesce hints		l0		l0 coalesce		l0 coalesce+hint	
N=1	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	4.00	100.00	1.04	25.91	1.04	25.94	1.11	27.70	1.03	17.85	1.01	13.31
dct8x8m	4.00	100.00	1.01	25.19	1.00	25.01	1.04	26.05	1.04	26.05	1.00	6.52
mriq	4.00	100.00	3.83	95.72	3.80	94.95	1.38	34.52	1.36	33.48	1.32	32.44
rgb2cmyk	4.00	100.00	1.49	37.20	1.07	26.85	1.11	27.80	1.03	17.60	1.01	13.88
strsearch	4.00	100.00	1.45	36.27	1.46	36.39	1.24	31.08	1.11	22.82	1.11	22.84
uts	4.00	100.00	3.90	97.44	3.25	81.30	1.12	28.00	1.11	27.67	1.32	22.89
bfs-d	4.00	100.00	2.09	52.16	2.06	51.19	1.25	31.12	1.10	23.04	1.13	22.43
bfs-nd	4.00	100.00	1.55	38.69	1.52	37.86	1.36	34.01	1.08	22.44	1.08	19.30
dict	4.00	100.00	1.60	40.04	1.59	39.75	1.26	31.49	1.12	24.15	1.12	22.17
mis	4.00	100.00	2.61	65.29	2.61	65.16	1.51	37.81	1.27	28.43	1.27	27.71
rdups	4.00	100.00	1.93	48.17	1.92	48.08	1.41	35.30	1.24	28.19	1.21	23.96
sarray	4.00	100.00	1.98	45.94	1.96	45.32	1.30	32.51	1.10	21.86	1.11	19.04
qsort	4.00	100.00	3.55	88.63	3.55	88.74	1.37	34.16	1.34	32.88	1.34	32.66
qsort-1	4.00	100.00	2.94	73.52	2.97	74.20	1.36	33.96	1.29	30.79	1.31	30.35
qsort-2	4.00	100.00	3.72	92.94	3.73	93.29	1.54	38.43	1.46	36.25	1.53	36.83
sampsort	4.00	100.00	3.46	86.60	3.51	87.72	1.34	33.57	1.30	31.65	1.57	31.74
sampsort-1	4.00	100.00	3.44	86.00	3.41	85.36	1.32	33.05	1.28	31.09	1.63	29.81
sampsort-2	4.00	100.00	3.83	95.87	3.69	92.15	1.40	34.92	1.36	33.55	1.91	32.69
hull	4.00	100.00	1.23	30.74	1.18	29.38	1.35	33.75	1.04	19.29	1.07	14.64
cilksort	4.00	100.00	2.89	72.16	2.90	72.41	1.46	36.47	1.21	25.68	1.24	25.25
heat	4.00	100.00	3.84	96.08	3.36	83.93	1.25	31.27	1.24	30.63	1.23	26.08
ksack	4.00	100.00	3.54	88.59	2.51	62.87	1.19	29.78	1.18	29.21	1.62	16.21
matmul	4.00	100.00	1.22	30.53	1.00	25.01	1.08	26.88	1.01	15.25	1.00	6.73
N=2	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	2.00	100.00	1.00	49.75	1.00	44.56	1.00	27.83	1.00	24.79	1.00	13.38
dct8x8m	2.00	100.00	1.00	50.01	1.00	25.01	1.00	26.05	1.00	23.98	1.00	6.52
mriq	2.00	100.00	1.89	94.61	1.90	94.73	1.03	33.43	1.04	33.54	1.04	32.88
rgb2cmyk	2.00	100.00	1.02	46.80	1.00	46.74	1.03	27.81	1.00	21.06	1.00	13.82
strsearch	2.00	100.00	1.09	49.36	1.09	49.51	1.01	31.32	1.00	28.11	1.01	28.05
uts	2.00	100.00	1.97	98.34	1.74	83.21	1.01	28.14	1.01	27.87	1.27	22.86
bfs-d	2.00	100.00	1.26	60.27	1.27	58.70	1.02	31.18	1.01	26.34	1.05	24.75
bfs-nd	2.00	100.00	1.12	48.84	1.13	47.51	1.02	34.05	1.00	26.39	1.03	21.49
dict	2.00	100.00	1.13	54.01	1.13	51.44	1.02	31.50	1.01	28.54	1.01	25.61
mis	2.00	100.00	1.42	69.30	1.42	67.35	1.05	37.82	1.03	32.93	1.03	30.66
rdups	2.00	100.00	1.25	58.07	1.25	52.65	1.02	35.30	1.01	29.94	1.01	23.91
sarray	2.00	100.00	1.23	56.74	1.22	54.07	1.02	33.82	1.01	27.79	1.03	21.99
qsort	2.00	100.00	1.80	90.09	1.69	84.49	1.05	37.15	1.03	33.38	1.05	32.97
qsort-1	2.00	100.00	1.41	70.12	1.44	71.11	1.05	36.74	1.03	32.48	1.07	32.20
qsort-2	2.00	100.00	1.78	88.89	1.79	89.34	1.05	39.59	1.04	36.96	1.14	37.09
sampsort	2.00	100.00	1.77	88.40	1.88	90.85	1.03	33.64	1.03	32.69	1.39	32.68
sampsort-1	2.00	100.00	1.75	87.49	1.89	87.22	1.03	33.07	1.03	32.18	1.48	30.71
sampsort-2	2.00	100.00	1.93	96.19	2.15	92.00	1.03	34.86	1.03	34.14	1.74	33.09
hull	2.00	100.00	1.04	44.84	1.08	38.17	1.02	33.93	1.00	27.07	1.06	17.57
cilksort	2.00	100.00	1.45	69.68	1.46	70.48	1.04	36.49	1.02	29.43	1.06	29.00
heat	2.00	100.00	1.93	96.27	1.70	84.62	1.02	31.22	1.02	30.90	1.10	25.57
ksack	2.00	100.00	1.87	93.44	1.64	54.31	1.02	29.78	1.02	28.95	1.57	16.04
matmul	2.00	100.00	1.02	42.31	1.00	25.01	1.00	26.88	1.00	22.73	1.00	6.73

Table A.4: WSRT Results for Sharing the Instruction Port Only – WSRT results for *sharing imem only* (CL-NI-4F-4L) design point. *base* = baseline with round-robin arbitration; *coalesce* = baseline with instruction coalescing enabled; *coalescing+hint* = combining coalescing with soft-barrier hints; *l0* = adding a L0 buffer to each lane; *l0+coalesce* = combining L0 buffers with instruction coalescing; *l0+coalesce+hint* = combining L0 buffer, instruction coalescing, and soft-barrier hints; *P* = Normalized delay; *IA* = Normalized instruction access

	rr		rr hint		mpc		mpc hint	
N=1	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	4.00	26.41	1.01	6.68	1.02	6.73	1.03	6.78
dct8x8m	4.00	26.05	1.00	6.51	1.00	6.51	1.00	6.51
mriq	3.92	32.52	3.91	32.43	4.08	24.07	4.08	24.07
rgb2cmyk	1.93	13.50	1.93	13.51	1.18	8.48	1.18	8.48
strsearch	1.63	16.31	1.63	16.23	1.45	15.05	1.45	15.05
uts	4.10	29.36	3.90	29.24	3.60	27.84	3.83	27.82
bfs-d	2.29	17.64	2.27	17.56	2.04	15.40	2.04	15.39
bfs-nd	1.93	14.88	1.61	12.20	1.43	10.54	1.43	10.53
dict	2.16	17.61	2.16	17.60	1.48	12.11	1.48	12.12
mis	3.04	29.28	3.03	29.26	2.70	26.30	2.71	26.38
rdups	2.12	19.66	2.12	19.66	1.83	16.87	1.83	16.87
sarray	2.66	14.63	2.56	13.34	2.81	11.92	2.81	11.93
hull	1.29	10.04	1.16	9.05	1.21	8.88	1.20	8.88
N=2	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	1.98	26.31	1.00	13.21	1.00	13.21	1.00	13.21
dct8x8m	2.00	26.05	1.00	6.51	1.00	13.03	1.00	6.51
mriq	1.97	32.64	1.96	32.46	3.16	25.74	3.16	25.74
rgb2cmyk	1.39	16.93	1.00	13.74	1.04	14.37	1.00	13.74
strsearch	1.17	23.96	1.17	23.97	1.06	22.21	1.06	22.20
uts	2.00	29.39	2.01	29.26	1.84	30.52	1.86	30.48
bfs-d	1.38	21.35	1.36	19.83	1.20	17.83	1.20	17.11
bfs-nd	1.16	17.95	1.15	15.92	1.09	15.26	1.09	14.67
dict	1.48	24.02	1.48	22.85	1.10	17.74	1.10	16.56
mis	1.65	31.81	1.64	30.37	1.36	26.37	1.36	25.04
rdups	1.34	24.86	1.34	21.47	1.19	21.24	1.19	18.10
sarray	1.20	19.72	1.15	15.37	1.07	17.40	1.07	13.84
hull	1.05	17.44	1.02	13.57	1.00	15.83	1.01	11.18

Table A.5: SPMD Results for Sharing the Instruction Port and Frontend Only – SPMD results for *sharing imem+fe only* (CL-NI-NF-4L) design point. *rr* = baseline with round-robin arbitration; *rr+hint* = baseline with soft-barrier hints enabled; *mpc* = baseline with the hybrid minimum-pc/round-robin thread selection mechanism; *mpc+hint* = *mpc* configuration with soft-barrier hints. *P* = Normalized delay; *IA* = Normalized instruction access

	rr		rr hint		mpc		mpc hint	
N=1	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	1.04	7.24	1.04	7.24	1.26	8.69	1.14	7.87
dct8x8m	4.00	26.05	1.00	6.52	1.05	6.82	1.00	6.52
mriq	4.00	33.72	3.91	32.74	3.35	28.57	3.32	28.42
rgb2cmyk	1.64	11.97	1.90	13.28	1.23	8.82	1.18	8.45
strsearch	1.63	12.96	1.64	13.02	1.48	11.61	1.49	11.65
uts	3.95	27.77	3.44	23.98	2.69	19.10	2.68	18.76
bfs-d	2.37	18.72	2.33	18.40	2.07	16.37	2.05	16.30
bfs-nd	2.11	17.26	1.67	13.62	1.46	11.96	1.45	11.91
dict	2.14	17.08	2.15	17.20	1.49	12.11	1.48	12.10
mis	3.04	28.83	3.03	28.72	2.71	25.92	2.71	25.89
rdups	2.14	19.53	2.12	19.40	1.84	16.68	1.83	16.63
sarray	2.22	18.09	2.15	17.82	1.99	18.26	1.99	18.24
qsort	3.85	35.47	3.83	35.38	4.27	41.86	4.28	41.87
qsort-1	3.21	30.10	3.22	29.90	3.82	36.95	3.83	36.98
qsort-2	3.88	38.21	3.88	38.23	4.26	42.58	4.27	42.71
sampsort	3.81	32.12	3.78	31.93	2.80	23.60	2.90	24.62
sampsort-1	3.82	31.73	3.67	30.64	3.01	25.04	3.08	25.77
sampsort-2	3.91	33.99	3.75	32.92	3.78	32.95	3.62	31.88
hull	1.68	14.18	1.22	10.56	1.27	10.85	1.26	10.95
cilksort	3.18	29.35	3.15	29.11	2.73	25.41	2.90	26.91
heat	3.93	30.84	3.38	26.53	2.56	20.19	2.66	21.17
ksack	3.82	28.46	2.38	17.71	1.40	10.54	2.23	16.65
matmul	1.00	6.73	1.00	6.72	1.22	8.25	1.00	6.72
N=2	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	1.00	13.93	1.00	12.47	1.02	14.14	1.00	12.41
dct8x8m	2.00	26.05	1.00	6.52	1.00	13.00	1.00	6.52
mriq	1.96	32.81	1.96	32.86	1.92	34.90	1.71	29.78
rgb2cmyk	1.21	17.97	1.00	13.77	1.00	13.76	1.00	13.77
strsearch	1.17	18.37	1.18	18.54	1.06	16.35	1.07	16.41
uts	1.98	27.93	1.71	22.51	1.38	19.37	1.51	18.90
bfs-d	1.41	22.12	1.39	20.93	1.25	19.27	1.27	18.69
bfs-nd	1.18	18.13	1.18	17.39	1.13	18.27	1.15	16.43
dict	1.48	23.33	1.48	22.17	1.10	17.45	1.10	16.29
mis	1.65	31.20	1.65	29.85	1.36	25.21	1.36	24.62
rdups	1.53	27.20	1.34	21.23	1.19	20.71	1.19	17.84
sarray	1.36	23.83	1.29	21.02	1.45	26.93	1.46	25.92
qsort	1.79	32.79	1.91	35.49	2.72	54.17	2.70	53.57
qsort-1	1.73	32.11	1.68	31.23	1.53	28.72	1.88	36.12
qsort-2	1.82	35.94	1.94	38.61	1.89	37.49	2.07	41.38
sampsort	1.92	32.37	1.98	32.26	1.61	27.28	1.79	29.26
sampsort-1	1.91	31.60	1.92	29.73	1.62	26.94	1.85	28.45
sampsort-2	1.96	34.05	2.17	32.92	1.91	33.30	2.14	32.24
hull	1.28	21.12	1.08	15.30	1.14	19.40	1.15	16.69
cilksort	1.67	30.57	1.66	30.25	1.40	25.23	1.47	26.23
heat	1.97	30.91	1.69	26.49	1.47	23.65	1.44	23.15
ksack	1.96	29.22	1.64	16.25	1.10	16.44	1.82	17.38
matmul	1.04	14.09	1.00	6.73	1.00	13.45	1.00	6.73

Table A.6: WSRT Results for Sharing the Instruction Port and Frontend Only – WSRT results for *sharing imem+fe only* (CL-NI-NF-4L) design point. *rr* = baseline with round-robin arbitration; *rr+hint* = baseline with soft-barrier hints enabled; *mpc* = baseline with the hybrid minimum-pc/round-robin thread selection mechanism; *mpc+hint* = *mpc* configuration with soft-barrier hints. *P* = Normalized delay; *IA* = Normalized instruction access

	0Lo-0Co-0Hi		0Lo-0Co-1Hi		0Lo-1Co-0Hi		0Lo-1Co-1Hi		1Lo-0Co-0Hi		1Lo-0Co-1Hi		1Lo-1Co-0Hi		1Lo-1Co-1Hi	
N=1	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	2.05	26.41	2.05	26.41	2.05	26.24	2.06	26.29	2.09	26.41	2.09	26.41	2.70	8.06	2.81	6.68
dct8x8m	2.61	26.05	2.61	26.05	2.60	26.05	2.60	26.05	2.63	26.05	2.64	26.05	2.68	13.03	3.71	6.51
mriq	1.45	33.38	1.45	33.38	1.42	33.07	1.42	33.07	1.46	33.38	1.46	33.38	1.44	30.74	1.43	30.89
rgb2cmyk	1.69	27.93	1.77	27.93	1.64	26.80	1.64	26.80	1.84	27.93	1.84	27.93	1.76	13.66	1.77	13.28
strsearch	1.72	42.60	1.72	42.60	1.36	30.72	1.36	30.67	1.74	42.60	1.74	42.60	1.33	25.78	1.33	25.75
uts	1.27	29.44	1.30	29.43	1.24	29.38	1.26	29.35	1.23	29.45	1.29	29.43	1.25	29.16	1.24	29.09
bfs-d	1.35	30.68	1.35	30.68	1.33	28.06	1.34	28.12	1.39	30.68	1.39	30.68	1.48	19.04	1.48	18.95
bfs-nd	1.44	31.53	1.43	31.53	1.37	27.77	1.37	28.22	1.47	31.53	1.48	31.53	1.55	15.52	1.51	17.22
dict	1.33	32.34	1.33	32.34	1.23	26.93	1.23	26.88	1.33	32.34	1.33	32.34	1.25	20.77	1.25	20.80
mis	1.59	38.58	1.59	38.58	1.50	34.05	1.53	33.71	1.61	38.58	1.61	38.58	1.49	29.16	1.52	28.49
rdups	1.52	37.18	1.52	37.18	1.36	30.56	1.40	29.52	1.52	37.18	1.52	37.18	1.40	23.33	1.41	23.06
sarray	1.41	31.74	1.42	31.74	1.32	28.74	1.32	28.65	1.45	31.74	1.45	31.74	1.48	14.96	1.50	14.03
hull	1.47	33.81	1.51	33.81	1.35	29.09	1.39	29.47	1.51	33.81	1.70	33.81	2.08	9.87	2.07	11.29
N=2	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	1.17	26.41	1.17	26.41	1.17	26.09	1.17	26.12	1.23	26.41	1.23	26.41	1.60	6.69	1.60	6.69
dct8x8m	1.34	26.05	1.34	26.05	1.33	25.80	1.34	25.82	1.73	26.05	1.36	26.05	1.90	6.51	1.90	6.51
mriq	1.06	33.38	1.06	33.38	1.05	33.14	1.05	33.14	1.05	33.38	1.05	33.38	1.05	32.34	1.05	32.41
rgb2cmyk	1.13	27.93	1.13	27.93	1.12	27.22	1.12	27.22	1.17	27.93	1.18	27.93	1.15	13.74	1.15	13.74
strsearch	1.08	42.60	1.08	42.60	1.02	32.30	1.02	32.29	1.09	42.60	1.09	42.60	1.05	27.76	1.05	27.73
uts	1.03	29.50	1.06	29.44	1.06	29.42	1.04	29.39	1.07	29.49	1.04	29.44	1.05	29.23	1.03	29.19
bfs-d	1.03	30.68	1.03	30.68	1.02	27.20	1.02	27.00	1.03	30.68	1.04	30.68	1.04	24.85	1.07	23.22
bfs-nd	1.04	31.53	1.04	31.53	1.03	27.69	1.03	24.25	1.05	31.53	1.05	31.53	1.09	21.57	1.15	16.56
dict	1.03	32.34	1.03	32.34	1.02	28.63	1.02	27.51	1.03	32.34	1.03	32.34	1.02	28.67	1.04	25.18
mis	1.07	38.57	1.09	38.58	1.05	33.82	1.05	33.82	1.08	38.58	1.09	38.58	1.08	32.44	1.09	31.08
rdups	1.05	37.19	1.09	37.19	1.03	31.71	1.03	30.16	1.07	37.19	1.09	37.19	1.08	25.66	1.10	24.09
sarray	1.05	31.74	1.05	31.74	1.03	28.60	1.02	26.37	1.08	31.74	1.08	31.74	1.12	20.51	1.13	15.18
hull	1.04	33.81	1.10	33.81	1.04	29.67	1.09	25.48	1.05	33.81	1.14	33.81	1.32	14.23	1.35	10.49

Table A.7: SPMD Results for Sharing the Instruction Port and LLFUs Only with Round-Robin Thread Selection – SPMD results for *sharing imem+llfu only* (CL-NI-4F-NL) design point with round-robin thread selection mechanism; The columns represent the values smart sharing mechanisms using the notation of (BLo-BCo-BHi) where Co = instruction coalescing; Lo = lockstep execution; Hi = soft-barrier hints; B = boolean choice of true or false which is used a prefix to indicate the applicability of a mechanism; *P* = Normalized delay; *IA* = Normalized instruction access.

	0Lo-0Co-0Hi		0Lo-0Co-1Hi		0Lo-1Co-0Hi		0Lo-1Co-1Hi		1Lo-0Co-0Hi		1Lo-0Co-1Hi		1Lo-1Co-0Hi		1Lo-1Co-1Hi	
N=1	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	2.06	27.81	2.07	27.83	2.08	27.65	2.07	27.67	2.10	27.82	2.10	27.83	2.72	8.57	2.78	7.95
dct8x8m	2.61	26.05	2.62	26.06	2.61	26.05	2.61	26.05	2.63	26.05	2.65	26.08	3.71	6.58	3.71	6.52
mriq	1.53	33.57	1.56	34.08	1.53	33.39	1.53	33.32	1.54	33.61	1.54	33.69	1.56	32.48	1.55	31.89
rgb2cmyk	1.66	27.98	1.68	28.05	1.65	28.14	1.65	28.15	1.89	27.84	1.89	27.81	1.91	13.34	1.93	11.16
strsearch	1.27	31.03	1.27	31.15	1.24	28.44	1.26	28.97	1.28	31.03	1.29	31.14	1.34	18.19	1.34	18.29
uts	1.29	27.99	1.52	28.01	1.29	28.00	1.53	27.73	1.29	27.98	1.54	27.92	1.29	25.85	1.57	21.86
bfs-d	1.39	31.52	1.43	31.82	1.36	28.98	1.40	29.27	1.43	31.60	1.47	31.91	1.49	20.25	1.55	19.99
bfs-nd	1.55	34.46	1.59	34.60	1.43	29.50	1.43	29.40	1.59	34.72	1.61	34.83	1.51	17.51	1.61	16.03
dict	1.31	31.73	1.31	31.74	1.23	27.48	1.23	27.50	1.31	31.75	1.31	31.75	1.23	23.19	1.26	21.93
mis	1.58	38.17	1.58	38.06	1.49	33.95	1.51	34.00	1.59	38.13	1.60	38.13	1.51	28.12	1.52	28.03
rdups	1.48	35.79	1.49	35.80	1.41	31.64	1.36	31.54	1.54	35.81	1.54	35.84	1.37	24.35	1.47	22.05
sarray	1.55	33.12	1.56	33.01	1.46	29.80	1.48	29.48	1.58	33.11	1.61	33.01	1.62	20.02	1.69	18.05
qsort	1.51	34.84	1.51	34.59	1.50	34.11	1.49	33.48	1.52	35.05	1.51	34.59	1.48	31.04	1.48	30.70
qsort-1	1.52	34.55	1.53	34.50	1.48	33.17	1.50	33.21	1.53	34.54	1.55	34.49	1.52	25.68	1.54	25.82
qsort-2	1.60	38.73	1.65	38.67	1.55	36.89	1.61	37.19	1.61	38.87	1.66	38.98	1.55	36.08	1.60	36.28
sampsort	1.66	33.68	1.90	33.80	1.65	33.21	1.88	33.34	1.67	33.57	1.90	33.79	1.66	31.00	1.90	31.35
sampsort-1	1.71	33.20	2.03	33.28	1.70	32.71	2.03	33.12	1.71	33.02	2.05	33.52	1.71	30.11	2.04	29.72
sampsort-2	1.79	35.00	2.28	34.95	1.78	34.52	2.27	34.49	1.79	35.02	2.29	34.94	1.78	33.55	2.28	32.52
hull	1.52	33.73	1.55	33.76	1.39	28.90	1.45	29.44	1.58	33.75	1.60	33.80	1.90	12.87	2.14	10.28
cilksort	1.52	36.81	1.53	36.42	1.43	33.24	1.45	33.18	1.52	36.51	1.54	36.45	1.43	31.23	1.45	30.99
heat	1.40	31.22	1.46	31.19	1.40	30.95	1.45	30.91	1.40	31.24	1.45	31.16	1.40	30.24	1.47	26.53
ksack	1.90	29.78	2.40	29.79	1.90	29.72	2.34	29.41	1.91	29.78	2.44	29.79	1.91	27.63	2.47	16.67
matmul	1.32	26.92	1.33	26.89	1.32	25.68	1.35	25.03	1.52	26.96	1.48	26.88	2.07	7.35	1.99	6.73
N=2	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	1.18	27.87	1.17	27.83	1.18	27.51	1.17	27.46	1.23	27.86	1.25	27.87	1.57	8.01	1.60	7.54
dct8x8m	1.37	26.81	1.37	26.78	1.37	26.45	1.37	26.59	1.79	26.05	1.78	26.06	1.90	6.56	1.90	6.52
mriq	1.07	34.14	1.07	33.60	1.06	33.17	1.06	33.28	1.06	33.64	1.08	34.23	1.08	33.06	1.07	32.37
rgb2cmyk	1.11	29.79	1.12	28.89	1.11	28.87	1.13	21.63	1.17	28.77	1.19	28.92	1.26	10.94	1.26	10.95
strsearch	1.01	31.23	1.01	31.26	1.01	29.51	1.01	29.55	1.01	31.22	1.02	31.26	1.05	22.81	1.07	23.05
uts	1.02	28.20	1.29	27.96	1.02	27.91	1.29	27.74	1.02	27.96	1.30	28.06	1.03	27.04	1.32	22.13
bfs-d	1.04	31.45	1.08	31.63	1.03	28.12	1.08	28.24	1.05	31.52	1.09	31.81	1.07	24.40	1.12	24.00
bfs-nd	1.05	34.61	1.10	34.78	1.04	29.40	1.07	27.56	1.06	34.58	1.11	34.76	1.15	18.18	1.18	18.23
dict	1.02	31.62	1.03	31.63	1.02	29.35	1.02	27.61	1.02	31.61	1.03	31.62	1.04	25.71	1.05	25.41
mis	1.07	38.08	1.07	38.10	1.06	34.81	1.06	33.28	1.08	38.07	1.07	38.04	1.08	31.39	nan	nan
rdups	1.03	35.62	1.06	35.63	1.03	31.19	1.03	28.13	1.03	35.64	1.06	35.62	1.06	26.99	1.10	24.30
sarray	1.05	33.81	1.08	33.95	1.04	29.97	1.06	28.77	1.09	34.00	1.12	33.96	1.15	21.58	1.23	18.72
qsort	1.07	37.36	1.07	36.44	1.06	34.87	1.07	34.44	1.07	37.09	1.08	36.75	1.06	32.89	1.07	32.65
qsort-1	1.07	36.75	1.10	36.76	1.06	34.04	1.09	34.03	1.08	36.82	1.11	36.72	1.10	27.87	1.13	28.13
qsort-2	1.07	39.51	1.17	39.51	1.06	37.56	1.16	37.69	1.07	39.54	1.17	39.55	1.06	36.79	1.16	37.06
sampsort	1.09	33.83	1.44	33.79	1.09	33.40	1.43	33.38	1.09	33.69	1.44	33.78	1.09	31.86	1.46	32.69
sampsort-1	1.09	33.12	1.55	33.43	1.09	32.81	1.56	32.76	1.10	33.25	1.57	33.49	1.11	30.77	1.57	30.03
sampsort-2	1.11	34.99	1.80	34.92	1.11	34.63	1.81	34.44	1.12	35.03	1.82	34.94	1.12	33.77	1.82	32.70
hull	1.04	33.90	1.13	34.25	1.03	29.97	1.11	27.09	1.08	33.91	1.14	34.20	1.28	13.54	1.42	10.53
cilksort	1.05	36.51	1.09	36.44	1.04	32.98	1.07	32.36	1.06	36.57	1.09	36.44	1.04	31.78	1.08	31.89
heat	1.04	31.24	1.12	31.23	1.04	31.06	1.12	31.19	1.04	31.25	1.12	31.35	1.04	30.76	1.14	26.23
ksack	1.14	29.78	1.86	29.79	1.14	29.74	1.85	28.47	1.15	29.79	1.88	29.80	1.16	27.23	1.90	17.76
matmul	1.00	26.92	1.06	26.88	1.00	26.84	1.01	13.47	1.02	26.93	1.13	26.88	1.26	10.55	1.25	6.73

Table A.8: WSRT Results for Sharing the Instruction Port and LLFUs Only with Round-Robin Thread Selection – WSRT results for *sharing imem+llfu only* (CL-NI-4F-NL) design point with round-robin thread selection mechanism; The columns represent the values smart sharing mechanisms using the notation of (BLo-BCo-BHi) where Co = instruction coalescing; Lo = lockstep execution; Hi = soft-barrier hints; B = boolean choice of true or false which is used a prefix to indicate the applicability of a mechanism; *P* = Normalized delay; *IA* = Normalized instruction access.

	0Lo-0Co-0Hi		0Lo-0Co-1Hi		0Lo-1Co-0Hi		0Lo-1Co-1Hi		1Lo-0Co-0Hi		1Lo-0Co-1Hi		1Lo-1Co-0Hi		1Lo-1Co-1Hi	
N=1	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	2.05	26.41	2.06	26.41	2.05	26.28	2.06	26.22	2.09	26.41	2.09	26.41	2.81	6.68	2.81	6.68
dct8x8m	2.63	26.05	2.63	26.05	2.62	26.02	2.61	26.04	2.63	26.05	2.59	26.05	2.66	13.03	3.71	6.51
mriq	1.60	33.38	1.60	33.38	1.48	31.12	1.48	31.12	1.60	33.38	1.60	33.38	1.48	27.69	1.47	27.97
rgb2cmyk	1.72	27.93	1.72	27.93	1.70	26.93	1.70	26.92	1.74	27.93	1.74	27.93	1.85	12.76	1.85	12.76
strsearch	1.68	42.60	1.68	42.60	1.35	20.87	1.35	20.88	1.68	42.60	1.68	42.60	1.41	18.12	1.41	18.12
uts	1.49	29.79	1.49	29.78	1.43	29.58	1.40	29.63	1.47	29.78	1.39	29.76	1.40	29.25	1.36	29.24
bfs-d	1.40	30.67	1.40	30.67	1.33	27.86	1.34	27.92	1.44	30.67	1.44	30.67	1.52	17.55	1.53	17.21
bfs-nd	1.49	31.53	1.49	31.53	1.38	27.26	1.37	27.32	1.53	31.53	1.52	31.53	1.52	15.46	1.55	14.59
dict	1.37	32.34	1.37	32.34	1.21	24.95	1.21	24.94	1.37	32.34	1.37	32.34	1.25	16.73	1.25	16.71
mis	1.62	38.59	1.62	38.59	1.51	33.72	1.51	33.72	1.63	38.59	1.63	38.59	1.53	27.70	1.53	27.69
rdups	1.60	37.18	1.60	37.18	1.35	31.14	1.35	30.63	1.61	37.19	1.61	37.19	1.42	21.58	1.47	20.01
sarray	1.54	31.74	1.53	31.74	1.30	28.87	1.31	28.87	1.54	31.74	1.53	31.74	1.49	14.37	1.49	14.30
hull	1.56	33.81	1.60	33.81	1.42	29.03	1.39	29.36	1.57	33.81	1.59	33.81	2.11	8.99	2.11	8.96
N=2	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	1.17	26.41	1.18	26.41	1.17	26.08	1.19	25.85	1.20	26.41	1.20	26.41	1.60	6.69	1.60	6.69
dct8x8m	1.38	26.05	1.38	26.05	1.38	25.98	1.38	25.93	1.60	26.05	1.50	26.05	1.90	6.55	1.90	6.51
mriq	1.05	33.38	1.05	33.38	1.05	32.96	1.05	32.97	1.05	33.38	1.05	33.38	1.06	31.80	1.06	31.80
rgb2cmyk	1.13	27.93	1.13	27.93	1.13	27.28	1.13	27.28	1.15	27.93	1.15	27.93	1.23	17.37	1.15	13.74
strsearch	1.08	42.60	1.08	42.60	1.02	32.33	1.02	32.35	1.08	42.60	1.08	42.60	1.05	27.20	1.05	27.23
uts	1.07	29.66	1.06	29.61	1.01	29.51	1.05	29.46	1.07	29.64	1.07	29.61	1.01	29.37	1.09	29.27
bfs-d	1.03	30.68	1.04	30.68	1.02	27.86	1.02	26.95	1.04	30.68	1.04	30.68	1.04	25.34	1.07	22.98
bfs-nd	1.04	31.53	1.04	31.53	1.03	27.21	1.03	24.04	1.05	31.53	1.05	31.53	1.09	21.51	1.15	16.57
dict	1.03	32.34	1.03	32.34	1.02	29.95	1.02	27.63	1.03	32.34	1.03	32.34	1.02	27.74	1.04	25.39
mis	1.06	38.57	1.08	38.57	1.06	35.20	1.06	33.84	1.07	38.57	1.08	38.57	1.07	33.66	1.09	30.95
rdups	1.04	37.19	1.06	37.19	1.03	31.97	1.03	30.25	1.04	37.19	1.06	37.19	1.04	28.83	1.10	24.16
sarray	1.04	31.74	1.04	31.74	1.02	27.71	1.02	25.14	1.05	31.74	1.06	31.74	1.11	18.86	1.13	15.30
hull	1.07	33.81	1.10	33.81	1.03	30.78	1.09	25.75	1.07	33.81	1.11	33.81	1.29	14.04	1.35	10.32

Table A.9: SPMD Results for Sharing the Instruction Port and LLFUs Only with Hybrid Minimum-PC Thread Selection – SPMD results for *sharing imem+llfu only* (CL-NI-4F-NL) design point with hybrid minimum-pc/round-robin thread selection mechanism; The columns represent the values smart sharing mechanisms using the notation of (BLo-BCo-BHi) where Co = instruction coalescing; Lo = lockstep execution; Hi = soft-barrier hints; B = boolean choice of true or false which is used a prefix to indicate the applicability of a mechanism; *P* = Normalized delay; *IA* = Normalized instruction access.

	0Lo-0Co-0Hi		0Lo-0Co-1Hi		0Lo-1Co-0Hi		0Lo-1Co-1Hi		1Lo-0Co-0Hi		1Lo-0Co-1Hi		1Lo-1Co-0Hi		1Lo-1Co-1Hi	
N=1	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	2.07	27.81	2.09	27.80	2.07	27.58	2.09	27.60	2.10	28.48	2.10	28.49	2.44	12.13	2.80	7.70
dct8x8m	2.61	26.07	2.64	26.06	2.61	26.04	2.61	26.04	2.62	26.12	2.62	26.12	3.71	6.58	3.71	6.52
mriq	1.75	37.30	1.75	37.06	1.70	35.07	1.70	34.86	1.74	37.10	1.75	37.03	1.60	28.91	1.60	28.88
rgb2cmyk	1.64	28.96	1.64	28.94	1.64	27.84	1.64	27.85	1.65	29.08	1.65	29.10	1.87	13.92	1.82	12.73
strsearch	1.27	30.83	1.28	30.85	1.25	27.97	1.25	27.88	1.29	30.86	1.31	31.20	1.41	15.17	1.41	15.09
uts	1.29	28.00	1.54	27.95	1.28	27.71	1.54	27.20	1.29	27.99	1.54	28.00	1.31	23.80	1.60	19.58
bfs-d	1.42	31.93	1.45	32.17	1.37	28.89	1.39	29.21	1.44	31.77	1.47	32.08	1.55	18.42	1.59	18.70
bfs-nd	1.55	35.02	1.56	35.14	1.41	28.61	1.43	28.85	1.56	34.88	1.58	35.11	1.56	16.56	1.62	15.29
dict	1.32	31.98	1.32	31.91	1.21	25.19	1.22	25.22	1.32	31.96	1.32	31.94	1.23	18.64	1.26	17.62
mis	1.59	38.28	1.59	38.25	1.51	33.59	1.51	33.55	1.60	38.28	1.60	38.29	1.53	27.00	1.53	26.97
rdups	1.47	36.45	1.47	36.43	1.34	29.46	1.34	29.45	1.49	36.48	1.49	36.47	1.44	20.91	1.47	19.94
sarray	1.63	35.92	1.65	35.98	1.54	32.48	1.54	32.02	1.62	35.47	1.63	35.43	1.72	19.76	1.77	18.41
qsort	2.00	47.01	2.01	46.99	1.83	41.15	1.82	40.75	2.01	47.21	2.03	47.33	1.70	34.40	1.66	33.06
qsort-1	1.97	45.95	1.99	45.84	1.77	39.02	1.78	39.00	1.97	45.79	1.99	45.76	1.67	27.14	1.70	27.52
qsort-2	1.86	45.21	1.91	45.29	1.70	40.19	1.76	40.58	1.86	45.02	1.90	44.91	1.65	37.86	1.71	38.17
sampsort	1.70	34.31	1.92	34.41	1.69	33.22	1.91	33.68	1.70	34.37	1.93	34.59	1.68	29.17	1.93	29.99
sampsort-1	1.74	33.93	2.06	34.05	1.72	32.70	2.04	33.15	1.74	33.79	2.07	34.13	1.73	28.10	2.07	29.05
sampsort-2	1.79	35.12	2.28	35.04	1.78	34.58	2.29	34.61	1.79	35.13	2.29	34.98	1.79	33.58	2.29	32.47
hull	1.52	34.25	1.56	34.11	1.43	29.24	1.48	29.55	1.52	34.31	1.56	34.42	2.09	11.21	2.15	10.34
cilksort	1.52	36.50	1.54	36.43	1.46	34.03	1.48	34.04	1.53	36.56	1.55	36.44	1.47	31.83	1.49	31.79
heat	1.42	31.32	1.48	31.25	1.41	30.38	1.48	30.05	1.42	31.38	1.48	31.36	1.41	28.67	1.52	24.16
ksack	1.89	29.79	2.41	29.79	1.89	29.59	2.44	29.15	1.89	29.78	2.39	29.79	1.91	26.33	2.47	16.21
matmul	1.30	27.00	1.32	26.96	1.31	25.80	1.28	23.96	1.54	27.01	1.54	27.02	2.03	8.94	1.99	6.73
N=2	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	1.18	27.86	1.18	27.81	1.18	27.45	1.18	27.45	1.20	27.87	1.21	27.84	1.57	8.02	1.60	7.53
dct8x8m	1.38	26.97	1.38	26.98	1.38	26.71	1.38	26.68	1.45	26.86	1.50	26.06	1.90	6.58	1.90	6.52
mriq	1.09	34.57	1.09	34.50	1.08	34.16	1.08	33.59	1.09	34.56	1.09	34.48	1.09	32.99	1.09	32.53
rgb2cmyk	1.12	29.90	1.13	29.52	1.12	28.92	1.13	21.64	1.14	30.01	1.14	28.45	1.26	11.33	1.26	10.95
strsearch	1.01	31.19	1.01	31.22	1.01	29.35	1.01	29.34	1.01	31.18	1.02	31.36	1.06	22.61	1.07	22.70
uts	1.02	28.02	1.29	28.06	1.02	27.99	1.29	27.79	1.02	28.15	1.29	27.96	1.02	26.57	1.32	22.30
bfs-d	1.04	31.49	1.09	31.90	1.03	28.42	1.07	27.86	1.05	31.64	1.10	31.94	1.08	24.31	1.12	23.97
bfs-nd	1.08	34.71	1.13	34.82	1.04	28.93	1.07	26.80	1.09	34.77	1.13	34.88	1.14	18.66	1.18	17.97
dict	1.03	31.62	1.03	31.63	1.02	28.09	1.03	27.23	1.03	31.62	1.03	31.62	1.04	25.17	1.05	24.88
mis	1.08	38.16	1.08	38.03	1.06	34.26	1.06	33.18	1.09	38.17	1.08	38.04	1.10	30.28	1.10	30.33
rdups	1.04	35.72	1.06	35.71	1.03	31.70	1.03	28.02	1.05	35.73	1.08	35.69	1.09	24.75	1.10	24.09
sarray	1.06	34.45	1.10	34.69	1.06	31.14	1.08	28.62	1.06	34.24	1.10	34.45	1.15	22.23	1.23	19.07
qsort	1.12	39.28	1.11	38.12	1.10	37.07	1.12	37.05	1.12	39.46	1.14	39.07	1.10	36.49	1.08	30.99
qsort-1	1.11	38.42	1.14	38.52	1.08	35.38	1.13	37.54	1.12	38.85	1.15	39.07	1.10	25.30	1.13	25.21
qsort-2	1.09	40.64	1.19	40.65	1.08	39.38	1.19	39.83	1.09	40.34	1.21	41.24	1.08	38.85	1.16	35.93
sampsort	1.09	33.76	1.44	33.93	1.08	33.35	1.44	33.53	1.10	33.99	1.44	33.95	1.10	31.66	1.45	31.84
sampsort-1	1.10	33.27	1.56	33.36	1.10	33.01	1.56	32.91	1.10	33.25	1.57	33.22	1.12	30.68	1.59	30.60
sampsort-2	1.12	35.18	1.79	34.93	1.11	34.79	1.82	34.80	1.12	35.13	1.82	34.89	1.12	33.82	1.83	32.77
hull	1.05	34.00	1.14	34.30	1.02	29.91	1.11	27.87	1.08	33.97	1.16	34.27	1.22	14.94	1.41	10.60
cilksort	1.05	36.50	1.09	36.45	1.04	33.77	1.08	33.57	1.05	36.47	1.09	36.43	1.04	33.26	1.08	33.08
heat	1.04	31.24	1.13	31.44	1.04	31.04	1.12	31.02	1.04	31.23	1.13	31.43	1.04	30.53	1.14	27.17
ksack	1.14	29.78	1.86	29.80	1.14	29.73	1.87	28.63	1.14	29.79	1.87	29.80	1.17	26.75	1.89	17.04
matmul	1.00	26.91	1.01	26.88	1.00	25.74	1.01	13.47	1.01	26.96	1.01	26.88	1.25	6.73	1.25	6.73

Table A.10: WSRT Results for Sharing the Instruction Port and LLFUs Only with Hybrid Minimum-PC Thread Selection – WSRT results for *sharing imem+llfu only* (CL-NI-4F-NL) design point with hybrid minimum-pc/round-robin thread selection mechanism; The columns represent the values smart sharing mechanisms using the notation of (BLo-BCo-BHi) where Co = instruction coalescing; Lo = lockstep execution; Hi = soft-barrier hints; B = boolean choice of true or false which is used a prefix to indicate the applicability of a mechanism; *P* = Normalized delay; *IA* = Normalized instruction access.

	rr		rr hint		mpc		mpc hint	
N=1	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	2.66	13.11	2.66	13.11	2.81	6.68	2.81	6.68
dct8x8m	4.00	26.05	3.71	6.51	3.71	6.51	3.71	6.51
mriq	3.91	32.37	3.93	32.54	4.13	24.30	4.11	24.10
rgb2cmyk	2.63	14.03	2.10	13.96	2.20	8.78	2.19	8.79
strsearch	1.85	15.94	1.86	15.96	1.81	16.94	1.81	16.95
uts	4.16	29.35	3.98	29.25	3.77	27.69	3.76	27.68
bfs-d	2.56	18.11	2.55	18.07	2.29	15.11	2.29	15.07
bfs-nd	2.05	12.61	2.02	12.24	1.97	10.89	1.97	10.88
dict	2.28	17.50	2.28	17.47	1.57	11.16	1.57	11.16
mis	3.16	29.56	3.16	29.56	2.87	26.61	2.87	26.63
rdups	2.39	19.80	2.39	19.80	2.14	17.15	2.14	17.15
sarray	3.01	15.72	2.96	13.51	3.23	12.07	3.23	12.07
hull	2.45	12.81	2.22	9.27	2.30	8.94	2.30	8.93
N=2	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	1.61	7.32	1.60	6.69	1.58	7.12	1.60	6.69
dct8x8m	2.00	26.05	1.90	6.51	1.90	6.51	1.90	6.51
mriq	1.97	32.65	1.97	32.56	3.19	25.86	3.19	25.86
rgb2cmyk	1.29	13.29	1.15	13.74	1.15	13.63	1.15	13.74
strsearch	1.25	20.45	1.25	20.57	1.13	19.22	1.13	19.22
uts	1.96	29.28	2.01	29.22	2.00	30.39	1.93	30.40
bfs-d	1.48	20.28	1.45	19.30	1.30	17.49	1.31	16.93
bfs-nd	1.33	14.05	1.32	13.78	1.21	14.13	1.25	12.55
dict	1.52	23.80	1.52	22.59	1.15	15.32	1.15	15.31
mis	1.70	30.51	1.70	30.56	1.45	25.19	1.44	25.18
rdups	1.43	22.67	1.43	21.09	1.29	18.16	1.30	17.90
sarray	1.25	15.87	1.28	14.17	1.19	13.53	1.21	12.66
hull	1.40	9.09	1.38	8.92	1.35	10.56	1.37	9.08

Table A.11: SPMD Results for Sharing the Instruction Port, Frontend and LLFUs – SPMD results for *sharing imem+fe+llfu* (CL-NI-NF-NL) design point. *rr* = baseline with round-robin arbitration; *rr+hint* = baseline with soft-barrier hints enabled; *mpc* = baseline with the hybrid minimum-pc/round-robin thread selection mechanism; *mpc+hint* = *mpc* configuration with soft-barrier hints. *P* = Normalized delay; *IA* = Normalized instruction access

	rr		rr hint		mpc		mpc hint	
N=1	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	2.67	13.39	2.68	12.73	2.79	7.62	2.81	7.24
dct8x8m	4.00	26.05	3.71	6.52	3.71	6.52	3.71	6.52
mriq	4.01	33.70	3.92	32.70	3.40	28.94	3.45	29.57
rgb2cmyk	2.22	10.14	2.19	9.66	2.19	8.47	2.20	8.53
strsearch	1.89	12.36	1.89	12.33	1.90	12.64	1.90	12.69
uts	3.97	27.95	3.43	23.58	2.68	18.47	2.72	18.16
bfs-d	2.61	19.14	2.61	19.00	2.30	15.91	2.31	15.89
bfs-nd	2.33	16.50	2.07	13.67	1.97	12.26	1.99	12.32
dict	2.30	17.31	2.28	17.18	1.56	11.04	1.56	11.03
mis	3.17	29.10	3.16	29.00	2.87	26.20	2.87	26.15
rdups	2.40	19.62	2.40	19.51	2.15	16.93	2.15	16.91
sarray	2.81	20.22	2.65	18.16	2.55	18.48	2.55	18.38
qsort	3.81	34.99	3.79	34.71	4.18	39.27	4.14	38.81
qsort-1	3.40	31.08	3.33	29.86	3.73	34.05	3.71	33.59
qsort-2	3.85	37.99	3.84	37.90	4.12	40.93	4.14	41.15
sampsort	3.84	32.19	3.83	32.11	3.04	24.11	3.10	24.58
sampsort-1	3.79	31.22	3.67	30.01	3.18	25.04	3.28	25.82
sampsort-2	3.91	33.95	3.79	32.90	3.78	32.78	3.70	32.04
hull	2.55	17.65	2.23	11.35	2.22	10.39	2.24	10.17
cilksort	3.30	30.27	3.27	30.01	2.98	27.36	2.95	27.00
heat	3.89	30.38	3.30	25.76	3.30	25.38	2.85	21.62
ksack	3.94	29.32	2.84	17.60	2.85	19.03	2.75	15.58
matmul	2.00	9.78	1.99	6.72	2.12	7.10	1.99	6.72
N=2	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	1.60	7.31	1.60	7.34	1.56	8.07	1.60	7.37
dct8x8m	2.00	26.05	1.90	6.52	1.41	13.03	1.90	6.52
mriq	2.00	33.62	1.96	32.69	2.11	37.94	2.00	35.37
rgb2cmyk	1.26	11.01	1.26	10.97	1.31	11.50	1.26	11.01
strsearch	1.26	14.75	1.27	14.98	1.18	12.88	1.18	12.93
uts	1.98	27.78	1.73	22.29	1.43	19.52	1.55	18.43
bfs-d	1.48	20.35	1.49	20.26	1.36	18.21	1.39	18.41
bfs-nd	1.35	16.02	1.34	15.52	1.26	15.17	1.31	14.73
dict	1.62	25.06	1.52	22.10	1.15	15.88	1.15	15.10
mis	1.74	30.69	1.70	29.91	1.44	25.24	1.45	24.78
rdups	1.43	21.48	1.43	20.86	1.29	18.28	1.30	17.75
sarray	1.54	21.06	1.46	18.08	1.58	23.71	1.61	22.30
qsort	1.81	32.96	1.80	32.60	2.25	43.08	2.52	49.14
qsort-1	1.78	31.80	1.54	26.38	1.47	25.12	1.87	33.90
qsort-2	1.83	35.90	1.83	35.98	2.04	40.70	2.05	40.92
sampsort	1.93	32.32	1.99	32.27	1.65	26.99	1.85	29.44
sampsort-1	1.92	31.54	1.98	30.20	1.66	26.54	1.90	28.61
sampsort-2	1.97	34.06	2.19	32.87	1.90	32.91	2.17	32.38
hull	1.58	18.37	1.44	10.34	1.46	13.65	1.49	11.48
cilksort	1.71	30.81	1.68	30.29	1.53	27.69	1.53	27.56
heat	1.97	30.80	1.66	25.51	1.51	23.21	1.41	21.03
ksack	1.96	28.98	1.87	16.39	1.40	17.68	1.98	17.28
matmul	1.26	6.78	1.25	6.73	1.39	9.16	1.25	6.73

Table A.12: WSRT Results for Sharing the Instruction Port, Frontend and LLFUs – WSRT results for *sharing imem+fe+llfu* (CL-NI-NF-NL) design point. *rr* = baseline with round-robin arbitration; *rr+hint* = baseline with soft-barrier hints enabled; *mpc* = baseline with the hybrid minimum-pc/round-robin thread selection mechanism; *mpc+hint* = *mpc* configuration with soft-barrier hints. *P* = Normalized delay; *IA* = Normalized instruction access

	rr		rr hint		mpc		mpc hint	
SPMD	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	4.00	26.41	4.00	13.22	3.99	9.19	3.99	9.19
dct8x8m	4.00	26.05	4.00	6.51	4.00	12.07	4.00	7.77
mriq	4.00	33.26	4.00	33.24	4.82	28.82	4.82	28.82
rgb2cmyk	4.00	26.98	4.00	27.01	3.98	12.63	3.98	11.97
strsearch	4.00	40.13	4.00	40.30	3.93	27.32	3.93	27.32
uts	4.00	29.49	4.09	29.25	4.09	29.86	4.15	29.92
bfs-d	4.00	28.70	4.00	25.27	4.02	22.24	4.02	22.24
bfs-nd	4.00	28.07	4.00	20.45	4.03	17.46	4.03	17.48
dict	4.00	31.67	4.00	28.17	4.00	16.90	4.00	16.90
mis	4.00	36.29	4.00	33.58	4.00	32.93	4.00	32.91
rdups	4.00	36.15	4.00	25.03	4.00	24.77	4.00	24.77
sarray	4.00	30.03	4.00	18.09	4.43	18.20	4.43	18.19
hull	4.00	30.59	4.00	13.97	4.06	14.11	4.06	14.12
WSRT	**P**	**IA**	**P**	**IA**	**P**	**IA**	**P**	**IA**
bilateral	4.00	26.41	4.00	13.22	3.99	9.19	3.99	9.19
bilateral	4.00	27.85	4.00	14.74	3.99	9.91	3.99	9.73
dct8x8m	4.00	26.05	4.00	6.52	4.00	7.78	4.00	7.77
mriq	4.00	33.33	4.05	33.78	4.36	34.28	4.35	34.01
rgb2cmyk	4.00	27.05	4.00	13.53	3.98	10.69	3.98	10.37
strsearch	4.00	29.70	4.00	29.70	3.97	23.39	3.97	23.40
uts	4.00	28.18	3.98	24.94	3.97	22.70	3.97	22.01
bfs-d	4.00	29.16	4.01	26.18	4.02	23.25	4.02	23.05
bfs-nd	4.00	28.87	4.01	23.44	4.02	19.43	4.04	19.47
dict	4.00	30.89	4.00	27.33	4.00	16.64	4.00	16.61
mis	4.00	34.22	4.00	32.83	4.00	32.63	4.00	32.52
rdups	4.00	32.88	4.00	24.75	4.01	24.61	4.01	24.57
sarray	4.00	32.58	3.99	23.57	3.99	23.58	4.07	23.93
qsort	4.00	36.93	4.00	35.28	5.07	41.14	5.07	40.89
qsort-1	4.00	35.38	4.00	36.35	4.85	35.61	4.85	36.96
qsort-2	4.00	38.83	3.99	38.01	4.49	42.22	4.52	42.61
sampsort	4.00	33.41	4.01	32.91	4.07	28.82	4.09	29.01
sampsort-1	4.00	32.93	4.00	31.21	4.11	29.24	4.08	29.31
sampsort-2	4.00	34.70	4.01	33.56	4.02	34.22	4.03	33.77
hull	4.00	32.31	3.98	17.72	4.05	15.73	3.98	14.91
cilksort	4.00	34.97	3.99	34.63	3.99	33.83	3.99	33.92
heat	4.00	31.28	4.00	30.74	4.00	29.11	4.02	26.63
ksack	4.00	29.75	4.00	16.01	4.00	23.96	4.00	20.51
matmul	4.00	25.66	4.00	6.72	4.00	13.61	4.00	13.52

Table A.13: SPMD and WSRT Results for Sharing All Resources – SPMD and WSRT results for *sharing all* (CL-1I-1F-1L) design point. *rr* = baseline with round-robin arbitration; *rr+hint* = baseline with soft-barrier hints enabled; *mpc* = baseline with the hybrid minimum-pc/round-robin thread selection mechanism; *mpc+hint* = *mpc* configuration with soft-barrier hints. *P* = Normalized delay; *IA* = Normalized instruction access

BIBLIOGRAPHY

[ABC+06] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, EECS Department, University of California, Berkeley, Dec 2006.

[AMW+07] M. Agarwal, K. Malik, K. M. Woley, S. S. Stone, and M. I. Frank. Exploiting Postdominance for Speculative Parallelization. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2007.

[API03] K. Atasu, L. Pozzi, and P. Ienne. Automatic Application-Specific Instruction-Set Extensions Under Microarchitectural Constraints. *Design Automation Conf. (DAC)*, Jun 2003.

[BBB+11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, May 2011.

[BEA+08] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 Processor: A 64-Core SoC with Mesh Interconnect. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2008.

[BJK+95] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Jul 1995.

[BL99] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM*, 46(5):720–748, Sep 1999.

[Bol12] J. Bolaria. Xeon Phi Targets Supercomputers. *Microprocessor Report*, Sep 2012.

[BPHH18] S. Bell, J. Pu, J. Hegarty, and M. Horowitz. Compiling Algorithms for Heterogeneous Systems. *Synthesis Lectures on Computer Architecture*, 13(1):1–107, 2018.

[CBK+14] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks. HELIX-RC: An Architecture-Compiler Co-Design for Automatic Parallelization of Irregular Programs. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2014.

[CDS+14] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Feb 2014.

[CFHZ04] J. Cong, Y. Fan, G. Han, and Z. Zhang. Application-Specific Instruction Generation for Configurable Processor Architectures. *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, Feb 2004.

[CJMT10] C. Campbell, R. Johnson, A. Miller, and S. Toub. *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures (Patterns & Practices)*. Microsoft Press, 2010.

[CKP+04] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner. Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2004.

[CL05] D. Chase and Y. Lev. Dynamic Circular Work-stealing Deque. *Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun 2005.

[CM08] G. Contreras and M. Martonosi. Characterizing and Improving the Performance of Intel Threading Building Blocks. *Int'l Symp. on Workload Characterization (IISWC)*, Sep 2008.

[Col11] S. Collange. Stack-less SIMT Reconvergence at Low Cost. Technical Report HAL-00622654, ARENAIRE, Sep 2011.

[CWS+14] L. Codrescu, A. Willie, S.Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule. Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications. *IEEE Micro*, 34(2):34–43, Mar/Apr 2014.

[DBBS+08] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient Embedded Computing. *IEEE Computer*, 47(7):27–32, Jul 2008.

[DFR10] M. Dechene, E. Forbes, and E. Rotenberg. Multithreaded Instruction Sharing. Technical report, ECE Department, North Carolina State University, Dec 2010.

[DGY+74] R. H. Dennard, F. H. Gaensslen, H.-N. 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 (JSSC)*, 9(5):256–268, Oct 1974.

[DKM+12] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. CPU DB: Recording Microprocessor History. *Communications of the ACM*, 55(4):55–63, Apr 2012.

[DZL+17] S. Dai, R. Zhao, G. Liu, S. Srinath, U. Gupta, C. Batten, and Z. Zhang. Dynamic Hazard Resolution for Pipelining Irregular Loops in High-Level Synthesis. *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, Feb 2017.

[EBA+11] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2011.

[EV96] R. Espasa and M. Valero. Decoupled Vector Architectures. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 1996.

[EVS98] R. Espasa, M. Valero, and J. E. Smith. Vector Architectures: Past, Present, and Future. *Int'l Symp. on Supercomputing (ICS)*, Jul 1998.

[Fla17] K. Flamm. Measuring MooreâĂŹs Law:Evidence from Price, Cost, and Quality Indexes. Technical report, University of Texas, Austin, Nov 2017.

[FLPR12] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. *ACM Transactions on Algorithms*, 8(1):4:1–4:22, 2012.

[FLR98] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 1998.

[gal18] Galois System. Online Webpage, 2018 (accessed March, 2018). `http://iss.ices.utexas.edu/?p=projects/galois`.

[GCC+08] J. Gonzalez, Q. Cai, P. Chaparro, G. Magklis, R. N. Rakvic, and A. Gonzalez. Thread fusion. *Int'l Symp. on Low-Power Electronics and Design (ISLPED)*, Aug 2008.

[GFA+11] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. Bundled Execution of Recurring Traces for Energy-efficient General Purpose Processing. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2011.

[GHN+12] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro*, 32(5):38–51, Sep/Oct 2012.

[GHS11] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically Specialized Datapaths for Energy-Efficient Computing. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2011.

[GKT91] G. Goff, K. Kennedey, and C.-W. Tseng. Practical Dependence Testing. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 1991.

[GNS13] V. Govindaraju, T. Nowatzki, and K. Sankaralingam. Breaking SIMD Shackles with an Exposed Flexible Microarchitecture and the Access Execute PDG. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2013.

[GRE+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. *IEEE Annual Workshop on Workload Characterization*, Dec 2001.

[GSO12] K. Gupta, J. A. Stuart, and J. D. Owens. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. *Innovative Parallel Computing (InPar)*, May 2012.

[Gwe14a] L. Gwennap. Qualcomm Tips Cortex-A57 Plans: Snapdragon 810 Combines Eight 64-Bit CPUs, LTE Baseband. *Microprocessor Report*, Apr 2014.

[Gwe14b] L. Gwennap. Samsung First with 20 nm Processor. *Microprocessor Report*, Sep 2014.

[HLR10] T. Harris, J. Larus, and R. Rajwar. Transactional Memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[HQW+10] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding Sources of Inefficiency in General-Purpose Chips. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2010.

[Hug15] C. J. Hughes. Single-Instruction Multiple-Data Execution. *Synthesis Lectures on Computer Architecture*, 2015.

[int13] Intel Cilk Plus Language Extension Specification, Version 1.2. Intel Reference Manual, Sep 2013. `https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm`.

[int15] Intel Threading Building Blocks. Online Webpage, 2015 (accessed Aug 2015). `https://software.intel.com/en-us/intel-tbb`.

[jav15] Java API: ForkJoinPool. Online API Documentation, 2015 (accessed Aug 2015). `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html`.

[Jes01] C. Jesshope. Implementing an Efficient Vector Instruction Set in a Chip Multiprocessor Using Micro-Threaded Pipelines. *Australia Computer Science Communications*, 23(4):80–88, 2001.

[Jon14] H. Jones. Why Migration to 20nm Bulk CMOS and 16nm/14nm FinFETs is Not Best Approach for Semiconductor Industry. IBS Whitepaper, 2014.

[JSY+15] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emery, and D. Sanchez. A Scalable Architecture for Ordered Parallelism. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2015.

[JYP+17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt,

J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[KB14] J. Kim and C. Batten. Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[KBH+04] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović. The Vector-Thread Architecture. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[KCSS16] S. Kalathingal, S. Collange, B. N. Swamy, and A. Seznec. Execution Drafting: Energy Efficiency Through Computation Deduplication. *Intl'l Symp. on Computer Architecture and High-Performance Computing (SBAC-PAD)*, Oct 2016.

[KHN07] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2007.

[KJT04] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-Core Chip Multiprocessing. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2004.

[KJT+17] J. Kim, S. Jiang, C. Torng, M. Wang, S. Srinath, B. Ilbeyi, K. Al-Hawaj, and C. Batten. Using Intra-core Loop-task Accelerators to Improve the Productivity and Performance of Task-based Parallel Programs. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2017.

[KLST13] J. Kim, D. Lockhart, S. Srinath, and C. Torng. Microarchitectural Mechanisms to Exploit Value Structure in Fine-Grain SIMT Architectures. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2013.

[KP03] C. Kozyrakis and D. Patterson. Scalable Vector Processors for Embedded Systems. *IEEE Micro*, 23(6):36–45, Nov 2003.

[Kre11] K. Krewell. ARM Pairs Cortex-A7 With A15: Big.Little Combines A5-Like Efficiency With A15 Capability. *Microprocessor Report*, Nov 2011.

[KSS+17] S. Kumar, N. Sumner, V. Srinivasan, S. Margerm, and A. Shriraman. Needle: Leveraging Program Analysis to Analyze and Extract Accelerators from Whole Programs. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2017.

[KT99] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Computer*, 48(9):866–880, Sep 1999.

[LAB+11] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerator Cores. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2011.

[LAB+12] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerator Cores. *ACM Trans. on Computer Systems (TOCS)*, 31(3):6, Aug 2012.

[LAS+09] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2009.

[Lea00] D. Lea. A Java Fork/Join Framework. *Java Grade Conference*, Jun 2000.

[Lee16] Y. Lee. *Decoupled Vector-Fetch Architecture with a Scalarizing Compiler*. Ph.D. Thesis, UC Berkeley, 2016.

[Lei09] C. E. Leiserson. The Cilk++ Concurrency Platform. *Design Automation Conf. (DAC)*, Jul 2009.

[LFB+10] G. Long, D. Franklin, S. Biswas, P. Oritz, J. Oberg, D. Fan, and F. T. Chong. Minimal Multi-threading: Finding and Removing Redundant Instructions in Multi-threaded Processors. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2010.

[LIB15] D. Lockhart, B. Ilbeyi, and C. Batten. Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar 2015.

[llv11] The LLVM Compiler Infrastructure Project. Online Webpage, 2011 (accessed February, 2011). `http://www.llvm.org`.

[LNOM08] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computer Architecture. *IEEE Micro*, 28(2):39–55, Mar/Apr 2008.

[LSB09] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. *Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, Oct 2009.

[LZB14] D. Lockhart, G. Zibrat, and C. Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[MBJ09] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches, 2009.

[MBW14] M. Mckeown, J. Balkind, and D. Wentzlaff. Execution Drafting: Energy Efficiency Through Computation Deduplication. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[MCM+14] T. Milanez, S. Collange, F. Magno, Q. Pereira, W. Meira, and R. Ferreira. Thread Scheduling and Memory Coalescing for dynamic Vectorization of SPMD Workloads. *Journal of Parallel Computing*, 40(9), 2014.

[Moo65] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics Magazine*, 1965.

[NGAS17] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam. Stream-Dataflow Acceleration. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[nvi17] NVIDIA TESLA V100 GPU Architecture. NVIDIA White Paper, 2017. `http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`.

[OHL+06] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An Unbalanced Tree Search Benchmark. *Int'l Workshop on Lanaguages and Compilers for Parallel Computing (LCPC)*, Nov 2006.

[ope13] OpenMP Application Program Interface, Version 4.0. OpenMP Architecture Review Board, Jul 2013. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[Oya99] Y. Oyanagi. Development of Supercomputers in Japan: Hardware and Software. *Parallel Computing*, 25(13–14):1545–1567, Dec 1999.

[pbe14] Polyhedral Benchmark Suite. Online Webpage, 2014 (accessed May, 2014). `http://www.cse.ohio-state.edu/~pouchet/software/polybench`.

[PNK+11] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. Hassaan, R. Kaleem, T. Lee, A. Lenharth, R. Manevich, M. Mendez-Lojo, D. Prountzos, and X. Sui. The Tao of Parallelism in Algorithms. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2011.

[PPA+13] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lusting, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer. Triggered Instructions: A Control Paradigm for Spatially-programmed Architectures. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2013.

[QHS+13] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. Horowitz. Convolution engine: balancing efficiency and flexibility in specialized computing. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2013.

[Rei07] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[Rob14] A. D. Robison. A Primer on Scheduling Fork-Join Parallelism with Work Stealing. Technical report, Intel Corporation, Jan 2014.

[Rup18] K. Rupp. Microprocessor Trend Data. Github Page, 2018 (accessed March 18, 2018. `https://github.com/karlrupp/microprocessor-trend-data`.

[SBF+12] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, and H. V. Simhadri. Brief Announcement: The Problem Based Benchmark Suite. *Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun 2012.

[SBV95] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 1995.

[SCZM00] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. *Int'l Symp. on Computer Architecture (ISCA)*, May 2000.

[SIT+14] S. Srinath, B. Ilbeyi, M. Tan, G. Liu, Z. Zhang, and C. Batten. Architectural Specialization for Inter-Iteration Loop Dependence Patterns. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[SRS+12] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical report, UIUC, IMPACT-12-01, Mar 2012.

[SS00] N. Slingerland and A. J. Smith. Multimedia Instruction Sets for General Purpose Microprocessors: A Survey. Technical report, EECS Department, University of California, Berkeley, Dec 2000.

[Str17] I. B. Strategies. IBS July 2017 monthly report: Design Activities and Strategic Implications. Technical report, International Business Strategies, Jul 2017.

[SYK10] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2010.

[SYW+13] R. Sampson, M. Yangt, S. Weit, C. Chakrabarti, and T. F. Wenisch. Sonic Millip3De: A Massively Parallel 3D-Stacked Accelerator for 3D Ultrasound. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2013.

[Tay13] M. B. Taylor. A Landscape of the New Dark Silicon Design Regime. *IEEE Micro*, 33(5):8–19, Sep/Oct 2013.

[Thi09] W. Thies. *Language and Compiler Support for Stream Programs*. Ph.D. Thesis, MIT, 2009.

[ti08] TMS320C28x Floating Point Unit and Instruction Set. Reference Guide, 2008. http://www.ti.com/lit/ug/sprueo2a/sprueo2a.pdf.

[TWB16] C. Torng, M. Wang, and C. Batten. Asymmetry-Aware Work-Stealing Runtimes. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2016.

[VSG+11] G. Venkatesh, J. Sampson, N. Goulding, S. K. Venkata, M. B. Taylor, and S. Swanson. QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores. *Int'l Symp. on Microarchitecture (MICRO)*, 2011.

[WAK+96] J. Wawrzynek, K. Asanović, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan. Spert-II: A Vector Microprocessor System. *IEEE Computer*, 29(3):79–86, Mar 1996.

[WKP11] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU Architecture. *IEEE Micro*, 31(2):50–59, Mar/Apr 2011.

[WLP+14] L. Wu, A. Lottarini, T. K. Paine, M. Kim, and K. A. Ross. Q100: the architecture and design of a database processing unit. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Feb 2014.

[Xil18] Xillinux Linux Distribution. Online Webpage, 2018 (accessed March, 2018). http://xillybus.com/xillinux.

[YBC+06] V. Yalala, D. Brasili, D. Carlson, A. Hughes, A. Jain, T. Kiszely, K. Kodandapani, A. Varadharajan, and T. Xanthopoulos. A 16-Core RISC Microprocessor with Network Extensions. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2006.

[ZLS+16] R. Zhao, G. Liu, S. Srinath, C. Batten, and Z. Zhang. Improving High-Level Synthesis with Decoupled Data Structure Optimization. *Design Automation Conf. (DAC)*, Jun 2016.

[ZMLM08] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering Hidden Loop Level Parallelism in Sequential Applications. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2008.

[ZMTC18] D. Zhang, X. Ma, M. Thomson, and D. Chiou. Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2018.

[Zyn18] Zynq-7000 Programmable SoC. Online Webpage, 2018 (accessed March, 2018). https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.