EFFICIENT FINE-GRAIN COOPERATIVE EXECUTION OF DYNAMIC TASK PARALLELISM ON HETEROGENEOUS MULTI/MANYCORE SYSTEMS

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
Moyang Wang
May 2021

EFFICIENT FINE-GRAIN COOPERATIVE EXECUTION OF
DYNAMIC TASK PARALLELISM ON
HETEROGENEOUS MULTI/MANYCORE SYSTEMS

Moyang Wang, Ph.D.
Cornell University 2021

Since the end of Dennard's scaling, computer architects have fully embraced parallelism to continue improving the performance and energy efficiency of general-purpose processors. Multicore processors with a few to tens of high performance processor cores have been the centerpiece of many computing platforms ranging from mobile devices to data centers. Manycore processors with hundreds or thousands of simple processing elements have demonstrated their ability to achieve even higher throughput and energy efficiency when abundant explicit parallelism exists in the workloads. However, large-scale manycore processors often lack hardware-based cache coherence. There is a growing trend towards a tighter integration between multicore and manycore processors, forming heterogeneous multi/manycore systems. These systems use heterogeneous cache coherence (HCC) with hardware-based cache coherence within the multicore and software-centric cache coherence with in the manycore. Unfortunately, programming heterogeneous multi-/manycore systems to enable collaborative execution is challenging, especially when considering dynamic task parallelism. This thesis uses a combination of light-weight software and hardware techniques to elegantly address this problem. It provides a detailed description of how to implement a work-stealing runtime to enable dynamic task parallelism on heterogeneous cache-coherent systems with a unified task-based programming model. This thesis also proposes direct task stealing (DTS), a new technique based on user-level interrupts to bypass the memory system and thus improve the performance and energy efficiency of work stealing. The cycle-level results in this thesis demonstrate that executing dynamic task-parallel applications on a 64-core system (4 big, 60 tiny) with complexity-effective HCC and DTS can achieve: 7$\times$ speedup over a single big core; 1.4$\times$ speedup over an area-equivalent eight big-core system with hardware-based cache coherence; and 21% better performance and similar energy efficiency compared to a 64-core system (4 big, 60 tiny) with full-system hardware-based cache coherence. This thesis also describes a realistic hardware implementation of heterogeneous multi/manycore systems based on an open-source hardware prototyping framework, OpenPiton. Using a VLSI methodology, this thesis shows that the heterogeneous multi/manycore approach achieves 3$\times$ hardware parallelism with the same area compared to a traditional homogeneous manycore.

BIOGRAPHICAL SKETCH

Moyang Wang was born in Beijing, China on January 27th, 1992. He is the only child of Fang Zhong and Jianmin Wang. Thanks to his father, Moyang had access to computers at the age of three. In his early education, Moyang enjoyed learning mathematics, physics, and basic computer programming. He also liked playing violins in his free time. Moyang attended the Second High School Attached to Beijing Normal University, where he met his favorite high school physics teacher, Menghua Peng, who inspired him to participate Physics Olympiad.

Moyang was accepted to Tsinghua University in Beijing, China as an undergraduate student major in electrical engineering. At Tsinghua, he received comprehensive education on how computer systems work, from the underlying principles of quantum physics, to application-level programming. Moyang completed his bachelor degrees in 2014, and was motivated to pursue further research in computer engineering with a focus on computer architecture.

Moyang was admitted to the PhD program in the School of Electrical and Computer Engineering at Cornell University in 2014. Throughout the next six years, he was fortunate to be advised by Professor Christopher Batten. He learned tremendous amount of knowledge and experience about computer architecture, VLSI, and software. More importantly, he learned from Professor Batten how to do good research. Moyang had the opportunity to be involved in multiple projects, spanning from software frameworks to silicon prototypes, at Batten Research Group.

Moyang is grateful for his time at Cornell. Despite of a lot of struggles, his PhD career was a time for indispensable personal growth with the guidance of a great advisor, and with the help of many friends and colleagues.

This document is dedicated to my parents and all my teachers.

ACKNOWLEDGEMENTS

My Ph.D journey has been a roller-coaster ride. I would like to thank many individuals who supported me during my time in graduate school. This dissertation would not have been possible without them.

I cannot say enough how wonderful my advisor Professor Christopher Batten is. Throughout my Ph.D career, he was always available for me, providing support when I need it. I have disappointed him many, many times: I have failed to pass the candidacy exam; I have run out of research ideas; and there have been times when I simply lost all motivations to work. Despite of all this, Chris has always been supportive and encouraging: he would ask me to take a walk with him to brainstorm new ideas; he would constantly check in to make sure I am on the right track; he would also remind me which area I need to work on (such as communication skills). Chris knows deeply about computer architecture, but what I learned from him the most is far beyond the technical subjects. Chris taught me to maintain the highest standard of research integrity; Chris taught me that details matter, even for things as small as Oxford commas or code indentations; Chris taught me to be proud of the work I do. Without Chris, my graduate school journey would not have been worthwhile.

I am also deeply thankful to my other two Ph.D committee members. Professor Zhiru Zhang always reminds me to look at the bigger picture, and think about "what are the killer apps for your research". Professor José Martínez has never failed challenge me with incisive questions (e.g., "why is your topic important?"), and forces me to defend what I am doing. Without their feedback, my thesis would not have been as strong and rigorous.

I would like to thank fellow PhD students at the Batten Research Group (BRG). Our conversations during and after work hours have provided me endless amount of inspirations and fun. I thank Derek Lockhart and Ji Kim for being senior role models in BRG, showing me how to be a good researcher. I am grateful for Shreesha Srinath, who gave me infinite amount of ideas to try on. I thank Christopher Torng for being my first collaborator and mentor when I first joined the group. I thank Berkin Ilbeyi for his impressive research on JIT magic which makes everything faster. I am also thankful to younger BRG members: Shunning Jiang, I admire your courage to talk to new people during conferences; Khalid Al-Hawaj, I enjoyed so much your jokes; Tuan Ta, your expertise on simulators truly impressed me, and I believe you will do some amazing research using it; Lin Cheng, you taught me a lot about consistency and coherence; Yanghui Ou and Peitian Pan, you have drastically made our team stronger by bringing in your VLSI and type-theory knowledge. I was fortunate to meet and work with two great postdoc researchers in BRG: Shady Agwa, your knowledge on chip tapeout helped me immensely; Cheng Tan, your advice on how to make papers get accepted was so helpful. I enjoyed working with many undergraduate students as well. I would like to specifically thank Eric Tang and Xiaoyu Yan. It was a pleasure to work with you on the CIFER tapeout project.

Within the CSL community I would like to thank all the fellow students who has chatted with me, shared an office with me, or grabbed a lunch with me. It is impossible to get through graduate school without the friendship from peers. Especially I want to give a shout-out to Ritchie Zhao, Yi Jiang, Charles Jeon. You made my life at CSL much more enjoyable.

I would like to express my gratitude to researchers outside of Cornell I have interacted with. I would like to especially thank Professor I-Ting Angelina Lee from Washington University in St. Louis. She is a wonderful researcher and an expert on work-stealing algorithms. She was so generous to me with her time that she agreed to have a weekly meeting with me to explore ideas on accelerating work stealing using hardware. She provided me so many insightful ideas to try on. While I did not do a good job to bring them into fruition at that time, the discussion I had with her is crucial to this thesis. My ISCA submission and this thesis would not have been possible without her insights on the interaction between work stealing and cache coherence. I would also like to thank Professor Sarita V. Adve from the University of Illinois at Urbana-Champaign, Professor Matt Sinclair from the University of Wisconsin-Madison, and Dr. Johnathan Alsop from AMD Research. They are all expert on cache coherence. In fact, they are from the research group which started the DeNovo project. I have truly benefited from our discussion after the ISCA conference. Their constructive feedback is critical to this thesis.

Lastly, I want to thank my family. I was with my mother Fang Zhong from primary school until high school. I thank her for raising me up and always believing in me. I want to thank my father, Jianmin Wang, for introduce me to engineering and computer, and for teaching me the importance of work ethics. I look forward to coming back home to see my parents, despite being on the other side of the world.

This thesis was supported in part by DARPA POSH Award #FA8650-18-2-7852, DARPA SDH Award #FA8650-18-2-7863, AFOSR YIP Award #FA9550-15-1-0194, NSF XPS Award #1337240, NSF CRI Award #1512937, NSF CAREER Award #1149464, a research gift from Xilinx, Inc., the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, and equipment, tool, and/or physical IP donations from Intel, Xilinx, Synopsys, Cadence, and ARM. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

MPI	message passing interface
MIC	many integrated core
GPGPU	general-purpose graphics processing unit
PGAS	partitioned global address space
SIMD	single instruction, multiple data
SIMT	single instruction, multiple threads
SoC	system on chip
APU	Accelerated Processing Unit
PCIe	Peripheral Component Interconnect Express
LLC	last-level cache
RISC	reduced instruction set computer
TBB	(Intel) Threading Building Blocks
CUDA	(NVIDIA) Compute Unified Device Architecture
API	application programming interface
SPMD	single program, multiple data
ISA	instruction set architecture
RTL	register-transfer level
VLSI	very-large-scale integration
ASIC	application-specific integrated circuit
SRAM	static random access memory
DRAM	dynamic random access memory
I$	instruction cache
D$	data cache

CHAPTER 1
INTRODUCTION

Since the end of Dennard's scaling, computer architects have fully embraced parallelism to continue improving the performance and energy efficiency of general-purpose processors. *Multicore processors* with a few to tens of high performance processor cores have been the centerpiece of many computing platforms ranging from mobile devices to data centers. *Manycore processors* with hundreds or thousands of simple processing elements have demonstrated their ability to achieve even higher throughput and energy efficiency when abundant explicit parallelism exists in the workloads. General-purpose graphics processing unit (GPGPU), a manycore architecture specialized for regular control flow and memory access patterns, has achieved tremendous success in domains like deep learning. However, how to efficiently support cooperative execution between multicore and manycore processors remains an open research question because of two major challenges. First, unlike multicore processors, manycore processors often lack hardware-based cache coherence, which forces programmers to use programming models different from the traditional shared-memory model. Second, emerging applications, such as graph processing, exhibit fine-grained parallelism where synchronization is required at task (as small as a few thousands of instructions) and word granularity, which makes traditional coarse-grained heterogeneous computing techniques infeasible.

In this thesis, I propose a hardware and software approach to efficiently support cooperative execution of fine-grained dynamic task-parallel applications on *heterogeneous multi/manycore systems* (systems consisting of both multicore and manycore processors). I present a state-of-the-art C++ task-based parallel programming framework with a work-stealing runtime to provide a unified programming model for heterogeneous manycore systems, enabling seamless cooperative execution of dynamic task-parallel applications. I discuss and evaluate direct task stealing, a hardware and software technique to improve performance and energy efficiency of fine-grained task parallelism on heterogeneous cache coherence. Lastly, I use an open-source hardware prototyping platforms to evaluate the area and energy implications of the heterogeneous multi/manycore systems proposed in this thesis.

1.1 A Survey of Manycore Processors

Increasing processor core count has been the primary approach used by computer architects to turn transistors into performance since the breakdown of Dennard's scaling. This trend is exemplified by manycore processors, which prioritize multi-thread performance rather than single-thread performance. Figure 1.1 illustrates the trend of increasing processor core count in the past two decades. As shown in the figure, there have been manycore processors with a hundred to a thousand cores recently. However, depending on their design goals, these manycore processors differ in their microarchitectures, interconnects, memory systems, and supported programming models. In this section, I briefly review some of the manycore processors designed and manufactured over the past two decades. The die photos of manycore processors mentioned in this chapter are shown in Figure 1.2.

Manycore processors date back to the early 2000s, when a few research prototypes were made to demonstrate the potential of the manycore approach in executing thread-parallel workloads. These prototypes used a multiple instruction, multiple data (MIMD) architecture. The MIT RAW processor [TKM+03, TLM+04] had 16 simple in-order cores connected by a $4{\times}4$ 2D mesh on-chip network. The Intel Teraflops research chip [HVS+07, VHR+07] contained 80 tiles arranged as a $10{\times}8$ 2D mesh of floating-point cores and routers. The Intel Single-Chip Cloud Computer

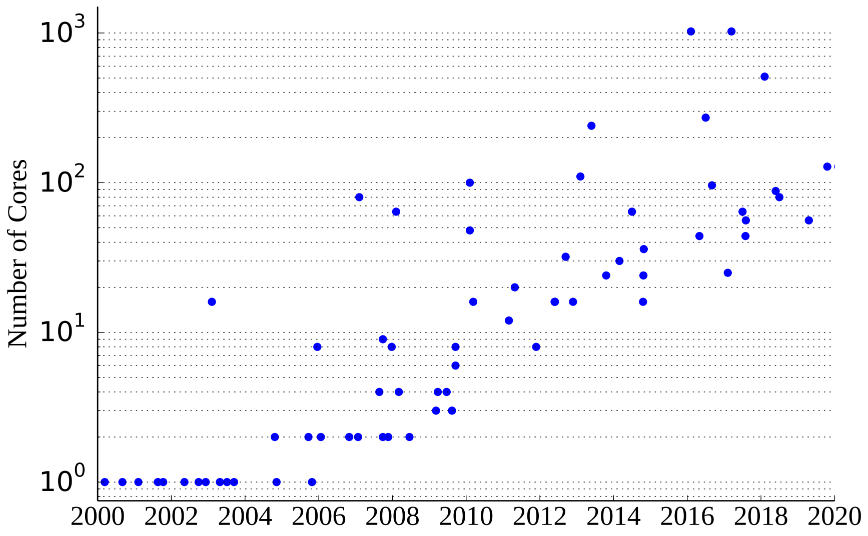
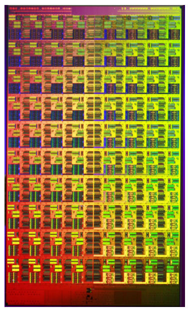
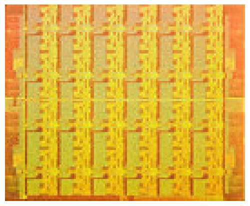
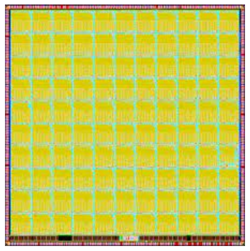
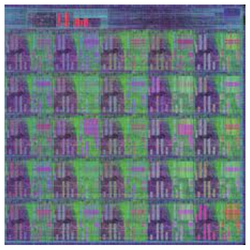


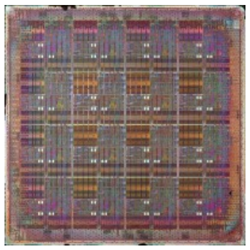
Figure 1.1: Trend of Processor Core Count – This figure shows the number of cores in selected processors from 2000 to 2020. The data is collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten, and K. Rupp [Rup21].

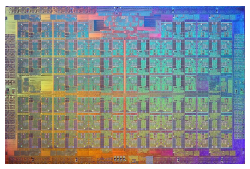

(a) Intel Teraflops Research Chip [HVS+07]

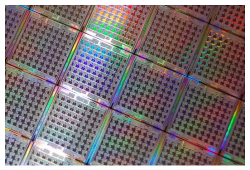

(b) Intel Single-Chip Cloud Computer [How10]

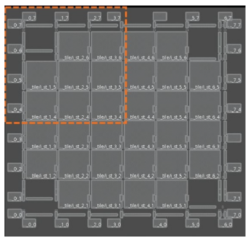

(c) The 110-core Execution Machine (EMP) [LSC+13]

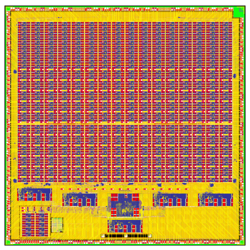

(d) The 25-core Piton [MFN+17]

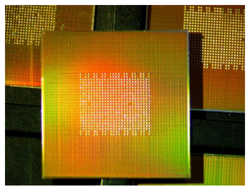

(e) MIT RAW Processor [TKM+03]

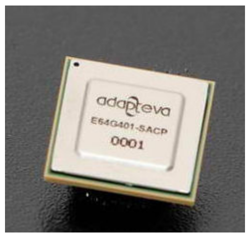

(f) Intel Knights Landing [SGC+16]


(g) TILE-GX100 [Ram11]


(h) Ampere Altra [Whe20]


(i) The 511-core Celerity Research Chip [DXT+18]


(j) The 1024-core KiloCore [BSP+17]


(k) The 1024-core Adapteva Epiphany-V [Olo16]

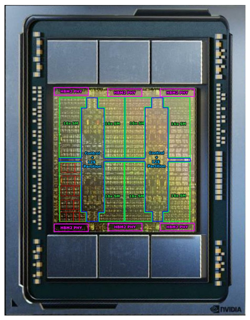

(l) NVIDIA A100 [nvi20]

Figure 1.2: Examples of Manycore Processors – Chip plots or die photos of selected manycore processors mentioned in this thesis. Pictures are adapted from their respective cited publications.

(SCC) [How10] was a manycore processor with 48 Pentium cores connected with a 4×6 2D-mesh. The 110-core Execution Migration Machine (EMP) [LSC+13] is a directory-less shared-memory manycore based on hardware-level thread migration. These early prototypes, except for EMP, have private instruction and data memory in each of the tiles, and rely on software to manage data transfer between tiles. RAW and Teraflops required custom programming models while SCC supported the message passing interface (MPI) used in high-performance computing. EMP provides a shared memory programming model, but requires applications to migrate between threads to exploit locality.

Commercial MIMD manycore processors started coming to the market around 2010. The Tilera Tile64 [WGH+07, BEA+08] is a 64-core manycore processor targeting high-performance embedded applications such as networking and multimedia. Its successor, TILE-GX100 [Ram11], expanded the core count to 100 cores. Intel has produced three generations of its many integrated core (MIC) architecture for high-performance computing applications: Knights Corner [Bol12], Knights Landing [SGC+16], and Knights Mill [BCC+17], offering up to 72 x86-64 cores in single socket. There are also academic research prototypes targeting the same domain, such as the 25-core Piton processor [MFN+17]. More recently, responding to the rise of scale-out applications in cloud data centers, vendors have provided ARM-based server-class manycore processors. Marvell ThunderX3 [Hal20] contains 96 cores and 384 threads per socket. Ampere has disclosed its 80-core Altra [Gwe20a] and 128-core Altra Max [Whe20]. These manycore processors have sophisticated mesh interconnects and hardware-based cache coherence, allowing programmers to use standard shared-memory multicore programming models.

Recent research prototypes have pushed the MIMD manycore approach further by scaling core counts by another order-of-magnitude. Celerity [DXT+18] incorporates 511 RISC-V cores into a single chip. KiloCore [BSP+17] contains 1000 cores. Adapteva Epiphany-V contains 1024 RISC-V 64-bit cores per chip [Olo16]. While these prototypes use a tile-based design with 2D mesh interconnects, they need to further simplify the hardware in order to scale up to hundreds of cores. They forgo hardware-based cache coherence, and require programmers to explicitly maintain coherency among cores. The programming models supported by these prototypes are different from cache-coherent manycore as well, such as remote store [DXT+18], partitioned global address space (PGAS) [Olo16], and message passing [Olo16].

GPGPUs are the most widely adopted type of manycore processor. Unlike MIMD manycore processors, GPGPUs have a single instruction multiple data (SIMD) architecture. They play a dominant role in many computing domains such as machine learning, bioinformatics, data science, and many more. A modern GPGPU contains thousands of scalar pipelines, known as CUDA cores (for NVIDIA GPGPU) or stream processors (SP, for AMD GPGPU), organized into tens of stream multiprocessors (SM, for NVIDIA GPGPU) or compute units (CU, for AMD GPGPU). For example, The NVIDIA A100 GPGPU has 6912 CUDA cores in 54 SMs [nvi20]; and the AMD MI100 has 7680 SPs in 120 CUs [amd20]. A GPGPU's memory system uses a light-weight, streaming-oriented cache coherence protocol which writes through dirty data and requires software-initiated self-invalidations at synchronization points for private caches in each SM or CU. GPGPUs support single-instruction multiple-thread (SIMT) programming models, such as CUDA and OpenCL.

Overall, there exists a fundamental tradeoff in manycore processor design. Simpler hardware leads to better scalability, but increases the programming model complexity. Complex hardware features, such as hardware-based cache coherence, facilitates programming on manycore processors, but may limit hardware scalability.

1.2 A Trend Towards Heterogeneous Manycore Systems

Although manycore processors have throughput and energy efficiency advantages over multicore processors when executing massively parallel workloads, it is unlikely for manycore processors to replace multicore processors due to the need for single-thread performance, support for legacy code, and support for operating systems. Instead, modern computing platforms often feature both multicore and manycore processors, constituting *heterogeneous multi/manycore systems*, to leverage both of their strengths. A tighter integration between multicore and manycore processors offers programmability and communication efficiency benefits.

Recent mobile and desktop system on chips (SoC) have multicore processors and GPGPUs integrated into a single chip. This approach offers better inter-device bandwidth and latency than the standard PCI Express (PCIe) interface used between hosts and discrete co-processors. AMD's accelerated processing unit (APU) [BFS12] offers shared access to off-chip memory for both multicore processors and GPGPUs. Intel SoCs, from Sandy Bridge [YKM$^+$11] to recent Tiger Lake [Gwe20b], use a shared last-level cache (LLC) for both their multicore processors and integrated GPGPUs.

A unified coherent memory space can improve programmability. There are many recent solutions to provide coherent memory access to host memory for manycore co-processors and other accelerators. IBM OpenCAPI [SSI$^+$18] uses a coherent MESI-based cache with a snoopy MESI LLC fabric to enable cache coherence between the host and other devices. CCIX [cci20] offers full inter-device cache coherency built on the PCIe physical layer and transport protocol. Heterogeneous system coherence (HSC) [PBG$^+$13], Fusion [KSV15], and AMD APUs use a hierarchical MESI-based cache structure to integrate the multicore processors and the manycore co-processors.

These solutions prioritize intra-device communication, and inter-device communication incurs additional latency and energy overhead. Spandex [ASA18] is a recent research proposal that efficiently and flexibly integrates various devices with diverse coherence properties. It uses a LLC that interfaces with requests from multiple coherence protocols. It has been shown that Spandex performs well across a wide range of data sharing patterns, including frequent and fine-grained inter-device communication.

1.3 Heterogeneous Programming Models

Heterogeneous manycore systems present a programmability challenge, primarily because of the disparity in programming models supported by multicore and manycore processors. In this section, I first briefly review programming models used by multicore and manycore processors, then discuss recent progress in programming models for heterogeneous manycore systems.

Multicore processors are mostly equipped with hardware-based cache coherence to provide a unified coherent memory space to all cores. This enables shared-memory programming models. The most common low-level programming model used in multicore processors is POSIX threads (pthreads). Task-based parallel programming models are becoming popular as they support a wide range of parallel patterns, provide automatic load balancing, and improve portability for legacy code [MRR12]. They can realize *dynamic parallelism* where parallel tasks are generated and mapped to hardware dynamically at run time. Examples of task-based parallel programming include Intel Cilk Plus [int13], Intel Threading Building Blocks (TBB) [int15], and OpenMP [ACD$^+$09, ope13].

While some manycore processors with modest core counts retain hardware-based cache coherence (e.g., Intel MIC), most manycore processors with massive core counts, including GPGPUs, do not provide hardware-based cache coherence or do not have caches at all. As a result, these manycore processors must use different programming model than the shared-memory model. X10 [CGS$^+$05], Chapel [cha20], and UPC [upc13] are examples of PGAS programming models, in which the global memory space is logically partitioned among processing elements. PGAS relies on software to transfer data between partitions of memory space. Some manycore processors use MPI [mpi13], a message passing framework where data is shared between cores only through explicit messages. GPGPUs support SIMT programming models, such as CUDA [cud13] and

OpenCL [ope11]. Both CUDA and OpenCL have the notion of multiple memory spaces (e.g., SM/CU-private and globally shared), and require explicit transfer between two memory spaces.

Programmers have to overcome the gap between different programming models in order to efficiently utilize heterogeneous manycore systems. This involves carefully decomposing the workload into partitions, parallelizing each partition with a programming model supported by the hardware, and balancing the execution between multicore and manycore processors. The most common application of this technique is in CPU-GPU heterogenous computing [MV15], where the multicore processors compute a portion of the workload by shared-memory programming models and offload the rest of the workload to the manycore GPGPU by launching CUDA or OpenCL kernels. Unfortunately, the current heterogeneous computing techniques still require programmers to use more than one programming model, limiting the overall programmability. Moreover, they partition the workload at a kernel granularity, limiting the opportunity to exploit fine-grained dynamic task parallelism with cooperative execution between multicore and manycore processors.

1.4 Thesis Overview

This thesis explores software and hardware techniques to efficiently support cooperative execution of fine-grained dynamic task-parallel applications in heterogeneous multi/manycore systems with a unified programming model. I will limit the discussions in this thesis to systems with MIMD manycore processors, because MIMD architectures handle control and memory access irregularity more efficiently.

Chapter 2 discusses a new lightweight and modular C++ framework for dynamic task-parallel programming called `appl`. `appl` contains a work-stealing runtime which provides support for dynamic task parallelism with a set of application programming interfaces (APIs) similar to Intel Cilk Plus and Intel Threading Building Blocks (TBB). In this chapter, I provide a detailed discussion of the implementation of `appl`, since an in-depth understanding of the work-stealing runtime in `appl` lays the foundations for later chapters.

Chapter 3 presents a software and hardware approach to support efficient fine-grained dynamic task parallelism in heterogeneous manycore systems. In this chapter, I demonstrate how to extend the work-stealing runtime in `appl` to heterogeneous manycore systems without full-system hardware-based cache coherence by leveraging structures of task-parallel applications and properties of work-stealing runtimes. The extended `appl` offers a unified programming model for both multicore and manycore processors in heterogeneous multi/manycore systems. I propose direct task stealing (DTS), a lightweight software and hardware technique that addresses inherent synchronization overheads in heterogeneous manycore systems due to the lack of full-system hardware-based cache coherence. DTS can effectively improve `appl`'s performance and energy efficiency in heterogeneous manycore systems.

Chapter 4 is a real-world case study of heterogeneous manycore systems. In this chapter, I provide a detailed description of how to implement heterogeneous multi/manycore systems at the register-transfer level (RTL), based on an open-source hardware prototyping platform, OpenPiton. I discuss area and timing of my RTL implementations based on VLSI results. I use these result to demonstrate the potentials of heterogeneous multi/manycore systems in future hardware.

Chapter 5 summarizes the contributions of this thesis and discusses directions for future work. The primary contributions of this thesis are:

- A new lightweight and modular C++ framework for dynamic task-parallel programming called `appl`, which serves as an extensible foundation for architecture and parallel programming research.
- To the best of my knowledge, the first detailed description of how to extend work-stealing runtimes to enable cooperative execution of fine-grained dynamic task-parallel applications on heterogeneous multi/manycore systems.
- A novel software and hardware technique which uses *direct task stealing* to improve the performance and energy efficiency of fine-grained dynamic task-parallel applications on heterogeneous multi/manycore systems.
- A realistic implementation of a heterogeneous multi/manycore system based on an open-source hardware prototyping platform which enables a detailed VLSI area and timing evaluation.

1.5 Collaboration, Previous Publications, and Funding

This thesis would not have been possible without the support from my advisor, Christopher Batten, and contributions from my colleagues at Cornell University and outside collaborators. My advisor Christopher Batten was a primary source of inspiration and guidance, and he was integral in all aspects of my research projects.

I developed the `appl` programming framework presented in Chapter 2 from scratch. I ported several task-parallel application kernels from Cilk-5 [FLR98], PBBS [SBF+12], PARSEC [BKSL08], and Ligra [SB13] benchmark suites to `appl`. Both `appl` and these application kernels were used in multiple projects led by my colleagues. I collaborated with Christopher Torng who led the asymmetric-aware work-stealing runtimes (AAWS) project. I instrumented an early version of `appl` to provide the underlying hardware hints for dynamic voltage and frequency scaling. The AAWS project was presented at International Symposium on Computer Architecture (ISCA) in 2016 [TWB16]. It played an important role in Chris Torng's Ph.D. thesis [Tor19] as well. I also collaborated with Ji Kim, who led the loop-task accelerator (LTA) project. I implemented a version of `appl` that can utilize LTA to accelerate loop tasks. The LTA project was published at the International Symposium on Microarchitecture (MICRO) in 2017 [KJT+17] and is an integral part of Ji Kim's Ph.D. thesis [Kim17]. Working on AAWS and LTA inspired me to propose direct task stealing that enables efficient execution of task parallelism on large-scale systems.

I led the efficient cooperative execution of task parallelism on heterogeneous multi/manycore systems work presented in Chapter 3, but the project would not have been possible without significant contributions from my colleagues Tuan Ta and Lin Cheng. Tuan Ta led the implementation of the on-chip network model for direct task stealing (DTS). We worked together on implementing three heterogeneous cache coherence protocols in the gem5+ruby cycle-level simulation framework. Tuan also played an important role in developing ideas for task-based cache coherency mechanisms. Lin Cheng was integral to the success of this project. He offered immense help on verifying and debugging the cache coherence protocols. Both Tuan Ta and Lin Cheng helped performing cycle-level simulations for this project. The DTS project was presented at the International Symposium on Computer Architecture (ISCA) in 2020 [WTCB20].

I led the project of implementing heterogeneous multi/manycore systems in the OpenPiton hardware prototyping framework presented in Chapter 4, but a strong team of colleagues worked

with me throughout the project. Xiaoyu Yan and Eric Tang authored a blocking cache in RTL that supports a software-centric cache coherence protocol. We then worked together to integrate the cache into a multi/manycore system. Yanghui Ou led the VLSI result. He contributed an ASIC flow for the heterogeneous multi/manycore system. Jonathan Balkind, Fei Gao, and Professor David Wentzlaff from Princeton University generously provided guidance to me on understanding the OpenPiton framework. They offered tremendous help with fixing RTL implementation issues of the heterogeneous cache coherence protocol.

This thesis was supported in part by DARPA POSH Award #FA8650-18-2-7852, DARPA SDH Award #FA8650-18-2-7863, AFOSR YIP Award #FA9550-15-1-0194, NSF XPS Award #1337240, NSF CRI Award #1512937, NSF CAREER Award #1149464, a research gift from Xilinx, Inc., the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, and equipment, tool, and/or physical IP donations from Intel, Xilinx, Synopsys, Cadence, and ARM. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

CHAPTER 2
APPL — A C++11 TASK-PARALLEL PROGRAMMING LIBRARY

Task-based parallel programming models have become popular for shared-memory multicore processors. They offer improved scalability by dynamically balancing work across available hardware threads. They also provide programmability benefits by allowing programmers to only spend their efforts on expressing *logical parallelism*, rather than on managing the actual software threads. Current state-of-the-art task-based programming frameworks provide programmability and scalability benefits, but there are two challenges when extending them to heterogenous multi/manycore systems. First, their code size is substantial. For example, Intel Threading Building Blocks has more than 15,000 lines of code; Intel Cilk Plus runtime has more than 19,000 lines of code and requires a special compiler. Second, they are not designed for extensibility. The components of these framework are highly interdependent. It requires significant amount of effort to modify them or port them to a new architecture.

To address these challenges, I developed `appl` (stands for application parallel programming library), a lightweight and modular C++11 task-based parallel programming library. It has only 3,000 lines of code and uses a modular design. It not only lays a foundation for a unified programming model for heterogenous multi/manycore systems, but also can be extended to other research projects. Section 2.1 gives an overview of the `appl` library. Section 2.2 describes the task model used in `appl`. Section 2.3 discusses the work-stealing scheduler in `appl`. Section 2.4 presents high-level parallel patterns supported by `appl`. Section 2.5 provides an evaluation of `appl` on real machines against state-of-the-art commercial parallel programming frameworks, Intel TBB and Intel Cilk Plus. Lastly, Section 2.6 presents two case studies on extending `appl` to support two architecture research projects.

2.1 Overview

`appl` is structured in a four-layer design, as shown in Figure 2.1. Each layer uses a set of application programming interfaces (APIs) provided by the layer below. The thread pool layer interacts with threading libraries provided by the underlying operating system (e.g., POSIX threads). Programmers are expected to use high-level APIs to express logical parallelism in their programs

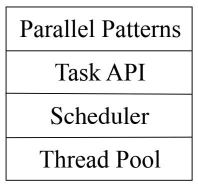


Figure 2.1: The Overall Structure of `appl`

in the parallel patterns layer, but they can also choose to interact with the scheduler using the low-level APIs in the task API layer.

Thread Pool – The thread pool layer manages a set of software threads to execute parallel programs. At the beginning of a parallel program, a `ThreadPool` object creates some number of threads, typically equals to the number of hardware threads in the system, using system APIs (e.g., `pthread_create` in POSIX) and these threads run a *runtime loop* provided by the scheduler discussed below. The threads that the `ThreadPool` manages are called *worker threads*. Worker threads usually are not terminated until the end of parallel programs, when the `ThreadPool` signals each worker thread to finish and waits for them to exit (e.g., using `pthread_join`).

Scheduler – The scheduler is the core of the `appl` runtime system. It schedules *tasks*, units of parallel execution, to worker threads. The scheduler implements the runtime loop that each worker thread runs. `appl`'s scheduler uses the work-stealing scheduling algorithm to provide load balancing. The details of the scheduler will be discussed in Chapter 2.3.

Task API – The task API layer consists of low-level APIs that work directly with the scheduler. It allows programmers to express potential parallelism using the *fork-join* parallelization strategy. The fork-join task model will be discussed in Chapter 2.2. However, programmers are expected to use the more intuitive high-level parallel patterns instead of low-level task APIs for common parallel patterns, as described below.

Parallel Patterns – The parallel patterns are set of high-level APIs provided to programmers. Currently, `appl` offers parallel for loop, reduce, fork-join, and while loop patterns. These `appl` APIs use meta-programming techniques to completely hide details of tasks from programmers, so that programmers can solely focus on expressing logical parallelism in their programs. They offer programmers productivity benefits by leading to fewer lines of code and more similarity to serial programs. These parallel patterns will be discussed in Chapter 2.4.

2.2 Task Model

In a task-based parallel programming framework, a *task* is a unit of computation that can be executed in parallel with other tasks. In appl, each task is represented by a C++ object derived from a base class, Task, shown in Figure 2.2. A task has a user-defined execute function, which defines the computation for the task. The execute function is called by the scheduler when a task is executed. Each task has a minimum of two member variables. m_ready_count counts the number of outstanding *child tasks*. m_successor_ptr is a pointer to the *successor* task of this task. The meaning of these two variables will be discussed later.

appl supports *dynamic task parallelism*, where tasks and dependencies among tasks are generated at runtime. In a parallel application, task objects derived from Task are constructed at runtime and submitted to the scheduler for execution. The computation model for dynamic task parallelism in appl is the *fork-join* model, which was initially used by MIT Cilk [BJK$^+$95] and was later popularized by many modern programming frameworks, including Intel Cilk Plus [Lei09, int13], Intel TBB [Rei07, int15], and others [CGS$^+$05, SML17]. In a fork-join execution model, the program's control flow diverges into two or more flows that can be executed in parallel (but not necessarily).

```
class Task {
public:
    Task(int ready_count, Task* succ_p)
        : m_ready_count(ready_count), m_successor_ptr(succ_p)
    { }

    virtual Task* execute() = 0;

    Task* get_successor() const {
        return m_successor_ptr;
    }

    int get_ready_count() const {
        return m_ready_count;
    }

protected:
    int    m_ready_count;
    Task*  m_successor_ptr;
};
```

Figure 2.2: appl **Task –** the definition of the base class of tasks in appl

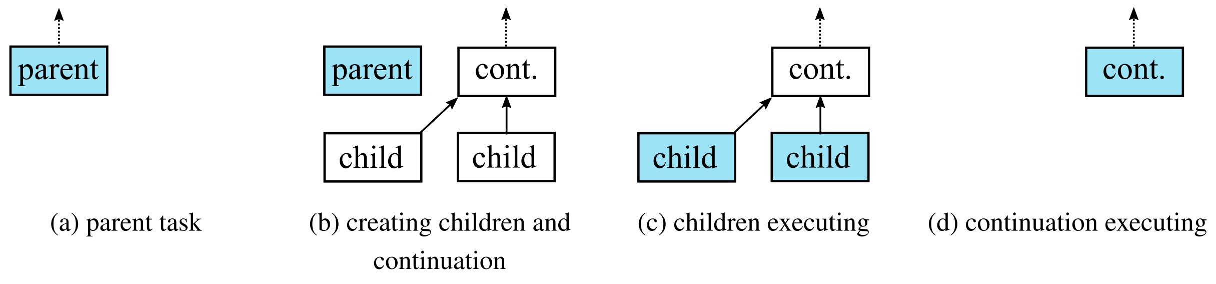


Figure 2.3: Continuation-Passing Form of Fork-Join Model in `appl` – An example showing a parent task creates two children and a continuation (cont.) task. Shaded tasks are the ones being executed.

The divergence point is called a *fork*. Two control flows converge into a single flow after they are finished, which is called a *join*.

A task can fork by creating two or more parallel tasks, which are referred to as *spawning* tasks. The created tasks are called the *child* tasks (or simply *children*); the spawning task becomes the *parent* task of its children. The parent task suspends its execution until its children complete. After the children are finished, the parent task resumes. In other words, the children join the parent. `appl` supports two forms of the fork-join model: continuation-passing and blocking.

2.2.1 Continuation-Passing Form

In the *continuation-passing form*, the parent task creates child tasks and specifies a *continuation* task to be executed when the children complete. The continuation inherits the parent's parent. The parent task can exit after its computation is done without blocking on its children. The children subsequently run, and after they (or their continuations) finish, the continuation task starts running. Figure 2.3 demonstrates an example of a parent task creating two children and a continuation task, and passing the execution to the continuation task when the children finish.

In `appl`, tasks can be explicitly constructed to use the continuation-passing form. Figure 2.4 presents an example of recursive Fibonacci number calculation using `appl`'s continuation-passing form. Line 1–11 define the continuation task. In line 23–25, a continuation task is constructed. The `m_successor_ptr` variable of a task indicates the next task to execute when this task is done. The continuation task shares the same `m_successor_ptr` pointer with the current task. The `m_ready_count` is set to 2, which equals to the number of children. Two children tasks are constructed in line 27–30, whose `m_successor_ptr` points to the continuation task, not the current task. In line 32–33, the two children are submitted to the scheduler for execution. The details of the

```
1   struct FibContinuation : public Task {
2       int* sum_p;
3       int  x, y;
4
5       FibContinuation(int* sp) : sum_p(sp) { }
6
7       Task* execute() override {
8           *sum_p = x + y;
9           return nullptr;
10      }
11  };
12
13  struct FibTask : public Task {
14      int   n;
15      int*  sum_p;
16
17      FibTask(int n_, int* sp, int g_) : n(n_), sum_p(sp) { }
18
19      Task* execute() override {
20          if ( n < 2 ) {
21              *sum_p = n;
22          } else {
23              FibContinuation* cont_task_p = new FibContinuation( sum_p );
24              cont_task_p->m_successor_ptr = this->m_successor_ptr;
25              cont_task_p->m_ready_count = 2;
26
27              FibTask* a_p = new FibTask(n-2, &cont_task_p->y);
28              FibTask* b_p = new FibTask(n-1, &cont_task_p->x);
29              a_p->m_successor_ptr = cont_task_p;
30              b_p->m_successor_ptr = cont_task_p;
31
32              spawn( a_p );
33              spawn( b_p );
34          }
35          return nullptr;
36      }
37  };
```

Figure 2.4: `appl` **Continuation-Passing Form –** The program computes a Fibonacci number in parallel using the continuation-passing form of fork-join model in `appl`.

scheduler will be discussed in Chapter 2.3. In the continuation-passing form, both the continuation and children outlive the current scope, they need to be allocated on the heap (line 23, 27, and 28). The scheduler is in charge of deallocating them when they are finished.

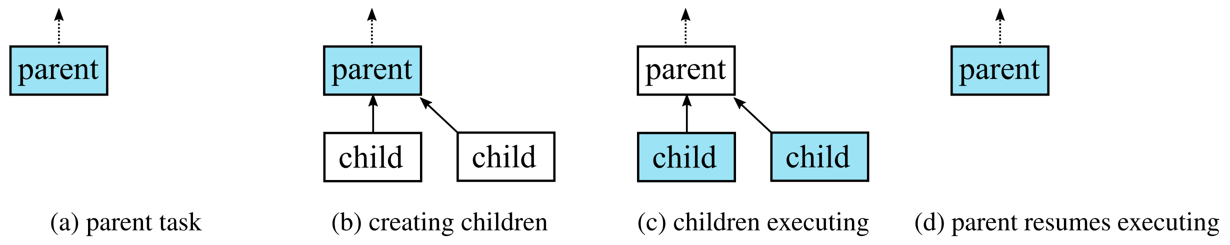


Figure 2.5: Blocking Form of Fork-Join Model in `appl` – An example showing a parent task creates two children. Shaded tasks are the ones being executed.

2.2.2 Blocking Form

While the continuation-passing form is more efficient in theory, it is more difficult to program without language and compiler support. The other fork-join form is the *blocking form*, which is less efficient in theory than the continuation-passing form, but more convenient to program. In the blocking form, the parent task blocks until its children complete, as shown in the Figure 2.5. The parent task resumes its execution after its children finish. In the blocking form, the continuation of the parent task is implicit in itself.

Figure 2.6 presents a concrete example of the blocking form of fork-join using the low-level APIs in `appl`. It is the same recursive Fibonacci calculation. However, in contrast to the continuation-passing form, there is no explicit continuation task. In line 15–16, the successor of the two children now points to the parent, instead of a distinct continuation task. Line 19–20 submit two children to the scheduler. The key difference in the blocking form is at line 21, where a scheduler function `wait` is called. The `wait` function enters a `runtime loop` defined by the scheduler until the children are finished (i.e., the `m_ready_count` of the parent goes down to zero). Afterwards, the parent resumes the execution after its children are finished, as shown in line 23.

In both forms of the fork-join model, the way to interact the scheduler is through `spawn` and `wait` functions. Next, I will discuss how the scheduler distributes tasks to threads and performs load balancing using the work-stealing algorithm.

2.3 Work-Stealing Scheduler

`appl`'s scheduler is a runtime system using the *work-stealing* algorithm [BL99] to schedule tasks to threads. In work-stealing schedulers, each thread is associated with a *task queue* data

```
struct FibTask : public Task {
    int  n;
    int* sum_p;

    FibTask(int n_, int* sp) : n(n_), sum_p(sp) { }

    Task* execute() override {
        if ( n < 2 ) {
            *sum_p = n;
        } else {
            int x, y;

            FibTask* a_p = new FibTask(n-2, &y);
            FibTask* b_p = new FibTask(n-1, &x);
            a_p->m_successor_ptr = this;
            b_p->m_successor_ptr = this;
            this->m_ready_count = 2;

            spawn(a_p);
            spawn(b_p);
            wait(this);

            *sum_p = x + y;
        }
        return nullptr;
    }
};
```

Figure 2.6: `appl` Blocking Form – The program computes a Fibonacci number in parallel using the blocking form of fork-join model in `appl`.

structure to store tasks that are available for execution. The task queue is a double-ended queue (*deque*). When a task spawns a child task, it *enqueues* the child on to the task queue of the executing thread. When a thread is available, it first attempts to *dequeue* a task from its own deque in last-in-first-out (LIFO) order from one end. If the thread's own deque is empty, it tries to *steal* a task from another thread's deque in first-in-first-out (FIFO) order from the other end. The thread that steals becomes a *thief*, and the thread whose tasks are stolen becomes a *victim*. When a parent task is waiting for its children to join, the thread executing the parent can steal from other threads. This algorithm thus automatically balances the workload across threads. It leads to better locality and helps establish time and space bounds when using the continuation-passing form [BL99, FLR98].

As I discussed in Chapter 2.2, the scheduler has two main functions, `spawn` and `wait`. Figure 2.7 presents the code of the `spawn` function, which simply enqueue a given task to the task

```
void spawn( Task* task_p ) {
    size_t tid = get_thread_id();
    TaskQueue& my_queue = m_queues[tid];
    my_queue.data.push_back( task_p );
}
```

Figure 2.7: The `spawn` Function of `appl`'s Scheduler – The `spawn function enqueue a task to the task queue of the current worker thread.`

```
void Scheduler::wait(Task* parent_task_p) {
    work_stealing_loop([&]() -> bool {
        return wait_task_p->get_ready_count() == 0;
    });
}
```

Figure 2.8: The `wait` Function of `appl`'s Scheduler – The `wait` function enters the work-stealing runtime loop until the given parent's `m_ready_count` goes down to zero.

queue associated with the current worker thread. The `wait` function enters the work-stealing runtime loop until the given parent task's `m_ready_count` is zero.

The work-stealing runtime loop is the core of the scheduler. It performs work stealing to balance the workload across all worker threads and achieves automatic load balancing. Figure 2.9 shows the code for the work-stealing loop. The loop continues until a given condition is met, as in line 7. Inside the work-stealing loop, a thread first checks if there is any task in its own deque, as in line 8. If so, it dequeues a task in LIFO order from its local deque to execute (line 11). If there is no task left on its local deque, the current thread becomes a thief and attempts to steal tasks from a randomly-selected worker thread (victim), as shown in line 19–28. The thief accesses a victim's deque in FIFO order. When a task is executed, its parent's reference count is atomically decremented, in line 24.

The work-stealing loop is not only entered during a `wait` call. All worker threads except for the main thread in the thread pool enter the work-stealing loop until the main thread terminates them at the end of the execution.

2.4 Parallel Patterns

The low-level task APIs provided by the scheduler and `Task` can be used to express a wide range of parallel algorithms. However, dealing with them directly can be tedious and error-prone.

```cpp
template <typename Func>
void Scheduler::work_stealing_loop(Func&& cond) {
    size_t      my_id    = get_thread_id();
    TaskQueue&  my_queue = m_queues[my_id];

    // wait until cond() == true
    while ( !cond() ) {
        Task* task_p = my_queue.data.pop_back();

        if (task_p) {
            execute_task(task_p);
        } else {
            size_t victim_id = rand() % m_nthreads; // select victim randomly
            // steal from the victim
            TaskQueue& victim_queue = m_queues[victim_id];
            Task* task_p = victim_queue.data.pop_front();

            if (task_p) {
                // steal attempt is successful
                Task* successor = task_p->get_successor();
                // execute the stolen task
                execute_task(task_p);
                if (successor) {
                    __atomic_fetch_sub(&successor->m_ready_count, 1);
                }
            }
        }
    }
}
```

Figure 2.9: The Work-Stealing Runtime Loop of `appl`'s Scheduler– The loop that performs work-stealing algorithm for task distribution and load balancing.

For example, using task APIs to parallelize a recursive calculation of Fibonacci, as shown in Figure 2.4 and Figure 2.6, causes the code size to increase by many times compared to a serial implementation. Fortunately, by leveraging C++ meta-programming techniques, a set of parallel patterns provided by the framework can simplify efficient parallel programming. A parallel pattern is a common control structure seen in parallel algorithms. It can be seen as an "algorithm skeleton" [MSM05]. Parallel patterns are defined by logical parallelism, and they hide implementation details from programmers. In this section, I describe a subset of parallel patterns provided by `appl`. Users can also easily add more patterns by using the low-level task APIs and/or by composing existing patterns.

```
void vvadd(int dest[], int src0[], int src1[], int size) {
    appl::parallel_for(0, size, [&] (int i) {
        dest[i] = src0[i] + src1[i];
    });
}
```

Figure 2.10: Example of `parallel_for` **–** An example of a vector-vector add function parallelized using `parallel_for`

2.4.1 `parallel_for`: Loop Pattern

The loop pattern is one of the most common parallel patterns. It describes a single function being applied to different data in parallel. It resembles the single instruction, multiple data (SIMD) model if the function's control flow does not depend on the data, or the single program, multiple data (SPMD) model if the function's control flow does depend on the data. The closest C structure is a fixed-bound `for` loop with no inter-iteration dependency.

`appl` has `parallel_for` template to express the loop pattern. Figure 2.10 shows an example of vector-vector add parallelized using the loop pattern. The `parallel_for` template takes a range, [0, size), and a functor (in this example, a lambda expression) that will be applied to each index within the range.

The `parallel_for` template constructs tasks based on the range and the functor. The granularity of the task (i.e., how many indices a task process) is critical to performance. If the granularity is too coarse, i.e., there are only a few big tasks, there is not enough parallelism. On the other hand, if the granularity is too fine, i.e., there are too many small tasks, the overhead of the runtime may overwhelm the useful work. How the range is divided among tasks is also important for the work-stealing scheduler. Since a steal has more overhead than an access to the local task queue, it is desirable to distribute the range evenly to worker threads with the least amount of steals possible.

`appl`'s `parallel_for` uses two heuristics to divide the range among tasks. The task constructed by `parallel_for` is shown in Figure 2.11. First, the granularity is set so that the total number of tasks is a few times as many as the number of worker threads. This scheme creates sufficient parallelism as well as opportunities for load-balancing without too much overhead. Second, the whole range is recursively divided by half until a set of granularity is reached. The division also happens lazily when the task is executed. Since steals happen in FIFO order, a single steal can distribute a large chunk of indices.

```
template <typename RangeT, typename BodyT>
class ParallelForTask : public Task {
public:
    ParallelForTask(RangeT range, BodyT body)
        : m_range(std::move(range)), m_body(std::move(body))
    { }

    Task* execute() {
        if (m_range.divisible()) {
            RangeT new_range = m_range.split();
            Task   join_point(2);

            ParallelForTask<RangeT, BodyT> right_half(new_range, m_body);
            right_half.set_successor(&join_point);

            spawn(&right_half);

            execute(); // execute the left half directly

            wait(&join_point);
        } else {
            m_body(m_range);
        }
        return nullptr;
  }

private:
    RangeT m_range;
    BodyT  m_body;
};
```

Figure 2.11: Implementation of `parallel_for` **–** The task defined by `parallel_for`.

2.4.2 `parallel_invoke`: Fork-Join Pattern

The fork-join pattern is another commonly used parallel structures. It describes a control flow that forks into multiple flows that join later. The forked flows can be executed in parallel. This pattern is identical to the underlying task execution model in `appl` described in Section 2.2.

The `parallel_invoke` template provides an easier way to implement the fork-join model. It takes two or more functors that can be executed in parallel. Figure 2.12 shows the same example of Figure 2.6, but using `parallel_invoke` instead of the low-level task APIs. It shows that the `parallel_invoke` template can significantly simplify the application code.

Because the fork-join pattern matches the task execution model, the implementation of `parallel_invoke` is straightforward. In Figure 2.13, an implementation of `parallel_invoke` with three functors is shown. Versions of `parallel_invoke` that take more functors are implemented similarly.

```
int fib(int n) {
    if (n < 2)
        return n;

    int x, y;

    appl::parallel_invoke(
        [&] { x = fib(n-1); },
        [&] { y = fib(n-2); }
    );

    return x + y;
}
```

Figure 2.12: Example of `parallel_invoke` – An example of a recursive Fibonacci parallelized using `parallel_invoke`

```
template <typename Func0, typename Func1, typename Func2>
void parallel_invoke(Func0&& func0, Func1&& func1, Func2&& func2 ) {
    Task join_point(3);

    auto t2 = Task(std::forward<Func2>(func2), &join_point);
    auto t1 = Task(std::forward<Func1>(func1), &join_point);

    spawn( &t2 );
    spawn( &t1 );

    // call func0 directly
    func0();

    wait( &join_point );
}
```

Figure 2.13: Implementation of `parallel_invoke`

2.4.3 `parallel_reduce`: Reduction Pattern

The reduction pattern is similar to the loop pattern, in which a function is applied to multiple data. However, in reduction pattern, the results are combined (reduced) to a single value.

`appl`'s `parallel_reduce` provides a way to express the reduction pattern. In addition to the arguments required by `parallel_for`, `parallel_reduce` takes another functor, which defines how two results are combined. Figure 2.14 shows an example of calculating the sum of a vector parallelized using `parallel_reduce`.

The implementation of `parallel_reduce` is similar to `parallel_for`. The given range is recursively divided into halves until the set granularity is reached. At the joining point of the two halves, the reduction functor is invoked to produce a combined result.

```cpp
double vector_sum(std::vector<double> vec) {
    double result = parallel_reduce(
        // range
        0, vec.size(),
        // left identity of the reduction (0 is the identity for summation)
        0.0,
        // reduction over a range, from an initial value
        [] (int start, int end, double init_value ) -> double {
            double value = init_value;
            for (int i = start; i < end; i++)
                value += vec[i];
            return value;
        },
        // reduction from two values
        [] ( double x, double y ) -> double { return x + y; }
    );
    return result;
}
```

Figure 2.14: Example of `parallel_reduce` – An example of calculating the sum of a vector parallelized using `parallel_reduce`.

2.5 Evaluation

To evaluate `appl`'s performance relative to the state-of-the-art task-based parallel programming frameworks, I have compiled `appl` for x86-64 architectures and benchmarked it using five application kernels from PBBS [SBF+12] benchmark suite. The speedup of each benchmark is calculated by dividing the execution time of the *optimized serial implementation* with the execution time of the parallel implementation. The input size for each benchmark is selected so that it takes ≈ 30 seconds to run the serial implementation. As a comparison, the same set of benchmark is also compiled with Intel Cilk Plus and Intel TBB. The speedup of Intel TBB and Intel Cilk Plus implementations of each benchmark is measured on the same machine.

The speedup results are shown in Table 2.1. The results show that `appl` has similar performance to Intel TBB and Intel Cilk Plus and is sometimes slightly faster due to the fact that `appl` is lighter weight and does not include advanced features like C++ exceptions or cancellations from within tasks.

	Cilk+	**TBB**	`appl`	`appl` **vs. TBB**
dict	4.02	5.02	5.53	+10%
radix	7.05	4.87	5.58	+14%
rdups	3.96	4.36	4.54	+4%
mis	2.75	2.42	2.40	-1%
nbody	7.37	7.10	6.95	-3%

Table 2.1: Performance of `appl` vs. Intel Cilk Plus and Intel TBB on Real System – Numbers are speedups vs. scalar implementation. Cilk+ = original Cilk implementation of PBBS apps. TBB = ported PBBS apps using `parallel_for` with Intel TBB 4.4 build 20150928. Baseline = ported PBBS apps using `parallel_for` of `appl`. All configurations are compiled with Intel C++ Compiler 14.0.2 (with Intel Cilk Plus language support). Each configuration uses eight threads running on an unloaded Linux server with two Intel Xeon E5620 processors.

2.6 Case Studies: Extending `appl` for Architecture Research

Since `appl` is light weight and modular, it can be easily modified and extended to facilitate architecture research. In this section, I use two research projects I have contributed to as two case studies to demonstrate the extensibility of `appl`.

2.6.1 Asymmetry-Aware Work-Stealing Runtimes

The focus of the asymmetry-aware work-stealing runtimes (AAWS) [TWB16, Tor19] is to exploit both the static and dynamic asymmetry in multicore systems for improving performance and energy efficiency of task-parallel applications. Static asymmetry arises from differences in core microarchitectures (e.g., ARM's big.LITTLE [Gre11]). Dynamic asymmetry arises from applying dynamic voltage and frequency scaling (DVFS). AAWS proposes three techniques to improve performance and energy efficiency of asymmetric multicore systems: work-pacing, work-sprinting, and work-mugging. Work-pacing uses a marginal-utility-based approach to maximize throughput in the high-parallel (HP) region by increasing the voltage of little in-order cores and decreasing the voltage of big out-of-order cores, while keeping the total power consumption constant. Work-sprinting combines the power slack generated from resting waiting cores in the low-parallel (LP) region with a marginal-utility-based approach to again maximize throughput. Work-mugging preemptively migrates work from little cores to big cores during the low-parallel region.

All three techniques require the software runtime (i.e., the work-stealing scheduler) to inform the underlying hardware of the activity status of each hardware thread. In this project, `appl`'s runtime loop (described in Section 2.3) was instrumented with hint instructions. Figure 2.15 shows

```
1      template <typename Func>
2      void Scheduler::work_stealing_loop(Func&& cond) {
3          size_t      my_id    = get_thread_id();
4          TaskQueue&  my_queue = m_queues[my_id];
5
6          // enter the runtime loop, begins resting
7          hint_rest();
8          while ( !cond() ) {
9              Task* task_p = my_queue.data.pop_back();
10
11             if (task_p) {
12                 // leave the runtime loop, start actively working
13                 hint_work();
14                 execute_task(task_p);
15                 // return to the runtime loop, begins resting
16                 hint_rest();
17             } else {
18                 size_t victim_id = rand() % m_nthreads;
19                 TaskQueue& victim_queue = m_queues[victim_id];
20                 Task* task_p = victim_queue.data.pop_front();
21
22                 if (task_p) {
23                     Task* successor = task_p->get_successor();
24                     execute_task(task_p);
25                     if (successor) {
26                         __atomic_fetch_sub(&successor->m_ready_count, 1);
27                     }
28                 }
29             }
30         }
31         // leave the runtime loop, start actively working
32         hint_work();
33     }
```

Figure 2.15: Asymmetry-Aware Runtime Loop of `appl`'s Scheduler

a simplified version of instrumented `appl` runtime loop. When a worker thread enters or exits the runtime loop (in line 7, line 13, and line 32), the worker thread executes a hint instruction that toggles an activity bit indicating the status of each core. Each core then adjust its DVFS status based on a marginal-utility-based approach.

Figure 2.16 presents the evaluation result of an application, `radix-2` from the PBBS benchmark suite running on a four big core, four little core (4B4L) system. The instrumented `appl` scheduler provide the DVFS controller run-time information about activities of each core, so that the DVFS controller adjust the voltages and frequencies to perform work-pacing in HP regions and work-sprinting in LP regions. The hints in the `appl` also inform the runtime to perform

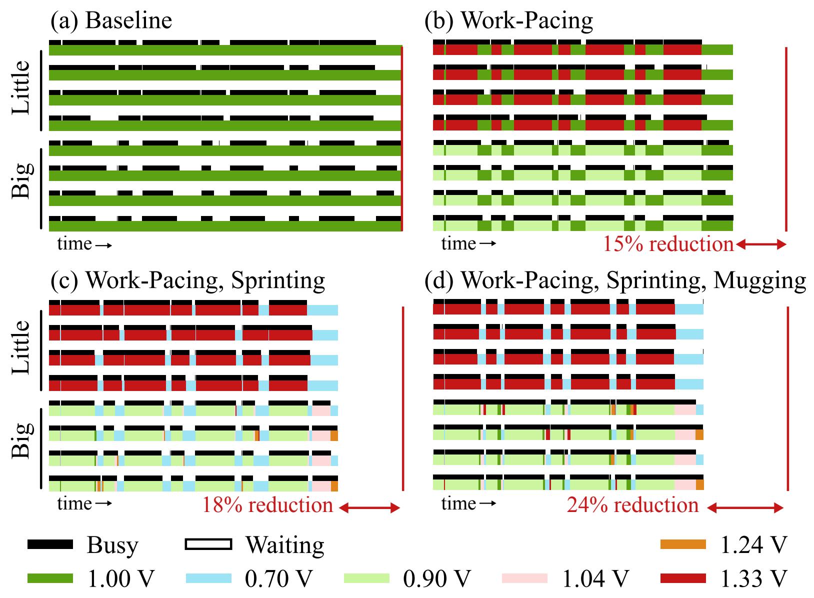


Figure 2.16: Activity Profiles for `radix-2` on 4B4L – Execution times of (b), (c), and (d) normalized to (a). Each row corresponds to a core's activity (black strip) and DVFS operating mode (colored strip) over time. (a) baseline 4B4L system; (b) applying work-pacing reduces HP region; (c) combining work-pacing and -sprinting reduces both HP and LP regions; (d) the complete AAWS version of `appl` runtime with work-pacing, sprinting, and mugging. The figure is adapted from [TWB16].

work-mugging in LP region if tasks are scheduled to little cores but not to big cores. Overall, the combined effect of work-pacing, work-sprinting, and work-mugging guided by modified `appl` scheduler is a 24% reduction in execution time compared to the `appl` runtime.

2.6.2 Loop-Task Accelerator

The loop-task accelerator (LTA) [KJT$^+$17, Kim17] project addresses two fundamental challenges when using packed-SIMD units with task-parallel programs: (1) the intra-core parallel abstraction gap; and (2) inefficient execution of irregular tasks. LTA focuses the loop pattern (Section 2.4.1), one of the most common forms of task parallelism. LTA has of three components. First, LTA provides a work-stealing runtime for distributing tasks across cores in software. Second, LTA proposes a new `xpfor` instruction that explicitly encodes loop-tasks as a function applied over a range of loop iterations. This instruction can be executed on either traditional general-purpose processors (GPPs) or on LTA engines that are designed to accelerate loop-task execution. Third, LTA proposes augmenting multicore processors with an intra-core accelerator that can be configured at

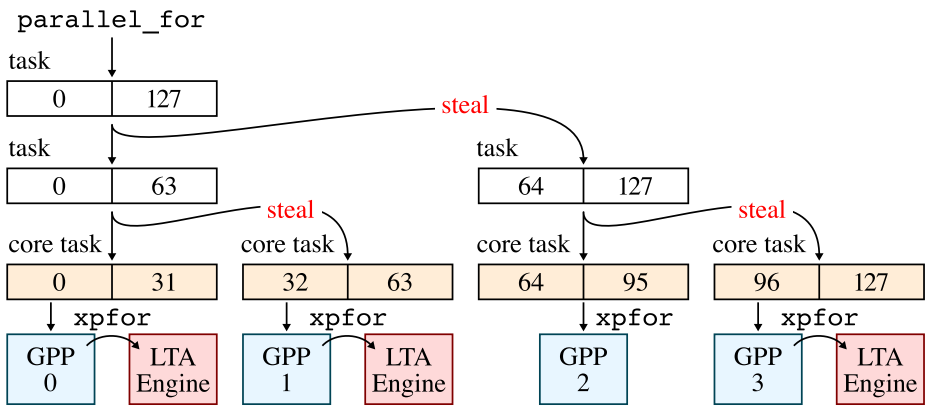


Figure 2.17: Example of LTA `appl` Runtime Task Partitioning – LTA `appl` runtime partitions tasks into *core tasks* which are distributed across cores to exploit loop-task parallelism. Core tasks are executed using the `xpfor` instruction that acts as an indirect function call to a loop-task function. If available, an LTA engine can further exploit loop-task parallelism within a core. The figure is adapted from [Kim17].

design time for different amounts of spatial/temporal decoupling to efficiently execute both regular and irregular loop tasks encoded by `xpfor` instructions.

The LTA runtime is a specialized version of `appl` that utilizes the `xpfor` instruction when executing loop tasks. In the LTA version of `appl`, tasks that are generated by the `parallel_for` template are called *LTA core tasks*. `appl` partitions and distributes LTA core tasks as described earlier. However, when the range of a core task is within a pre-set threshold, `appl` uses the `xpfor` instruction to perform accelerated execution on the core task when the core is augmented with an LTA engine. Figure 2.17 is an example of how the LTA-version of `appl` distributes core tasks and leverage LTA engines for specialized execution.

Figure 2.18 presents the performance results of the LTA-version `appl` across a wide range of loop-task-parallel application kernels on 4-core systems where each core is augmented with an LTA engine. The `appl` work-stealing scheduler distributes loop tasks across cores and uses the `xpfor` instruction within a core to utilize LTA engines. `appl` helps to achieve multiplicative speedup by combining inter-core parallelism and intra-core parallelism within the LTA engine.

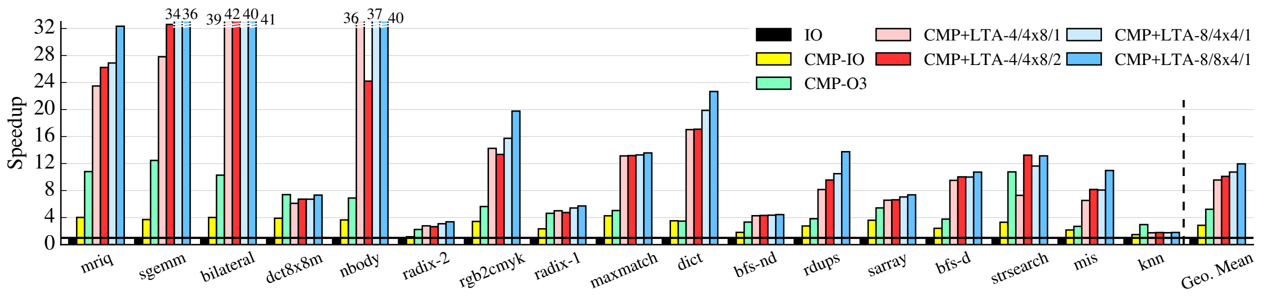


Figure 2.18: Performance of Multi-Core LTA Systems – Speedups of the most promising 4-lane and 8-lane 32-μthread LTA engines in a 4-core system normalized against a single in-order core for each application kernel. The speedups of the in-order and out-of-order baseline cores in a 4-core system are also shown for reference. The figure is adapted from [KJT$^+$17].

CHAPTER 3
EFFICIENTLY SUPPORTING DYNAMIC TASK PARALLELISM ON HETEROGENEOUS MULTI/MANYCORE SYSTEMS

Manycore processors, with tens to hundreds of tiny cores but no hardware-based cache coherence, can offer tremendous peak throughput on highly parallel programs while being complexity and energy efficient. Heterogeneous multi/manycore systems consist of a few high-performance big cores (for executing operating systems, legacy code, and serial regions) and tens to hundreds of simple tiny cores. These systems use heterogeneous cache coherence (HCC) with hardware-based cache coherence between big cores and software-centric cache coherence between tiny cores. Unfortunately, programming these heterogeneous cache-coherent systems to enable collaborative execution is challenging, especially when considering fine-grained dynamic task parallelism.

In this chapter, I seek to address this challenge using a combination of light-weight software and hardware techniques. Section 1 briefly discusses the programming challenge of HCC that motivates the approach taken in this chapter. Section 3.2 provide a brief background on HCC and work-stealing runtimes. Section 3.3 provides a detailed description of how to extend `appl`'s work-stealing runtime (described in Chapter 2) to enable dynamic task parallelism on heterogeneous cache-coherent systems. Section 3.4 describes the direct task stealing (DTS) technique to address inherent overheads in HCC. Section 3.4 also explains how DTS enables some important optimizations in the work-stealing runtime to improve performance and energy efficiency of multi-/manycore systems with HCC. Section 3.5 describes the cycle-level evaluation methodology used in this chapter. Section 3.6 presents the cycle-level evaluation results. Section 3.7 discusses related work of this chapter.

3.1 Introduction

Parallelism and specialization are currently the two major techniques used to turn the increasing number of transistors provided by Moore's law into performance. While hardware specialization has demonstrated its strength in certain domains (e.g., GPUs for data parallelism and accelerators for deep learning), general multi-threaded applications are still better suited for multi-core processors. Hardware architects have relied on parallelism to improve the performance of processors for several decades, and the trend of increasing processor core count is likely to continue. There has been a growing interest in using a *manycore* approach which integrates tens or hundreds of relatively simple cores into a system to further increase hardware parallelism. Examples of manycore processors include the 72-core Intel Knights Landing [SGC+16], 64-core Tilera TILE64 [BEA+08], and 25-core Piton [MFN+17]. The manycore approach has demonstrated its potential in achieving high throughput and energy efficiency per unit area for multi-threaded workloads.

Hardware designers have realized that an unoptimized hardware-based cache coherence protocol (e.g., directory-based MESI and its variants) is difficult to scale due to directory state storage overhead, network latency overhead, as well as design and verification complexity. Designing a performance-, complexity-, and area-scalable hardware-based cache coherence protocol remains an active area of research [ZSD10, CLS05, ZSM07, Mos05, MHS12, FLKBF11, BS13, FW15]. Another approach to continue increasing the number of cores in manycore systems is to sidestep hardware-based cache coherence and adopt software-centric cache coherence [KJJ+09, CKS+11, SKA13, SA15, RK12], software-managed scratchpad memory [BSL+02, KSA+15], and/or message passing without shared memory [KMPW11]. Manycore processors without hardware cache coherence have been fabricated both in industry (e.g., 1024-core Adapteva Epiphany-V [Olo16]) and academia (e.g., 511-core Celerity [DXT+18], 1000-core KiloCore [BSP+17]). By moving away from hardware-based cache coherence, these processors achieve exceptional theoretical throughput and energy efficiency (due to their massive core count), with relatively simple hardware. However, they have not been widely adopted because of challenges posed to software developers.

Programmers expect to use familiar CPU programming models, especially ones that support dynamic task-based parallelism, such as Intel Cilk Plus [int13], Intel Threading Building Blocks (TBB) [int15], and OpenMP [ACD+09, ope13]. These programming models allow parallel tasks to be generated and mapped to hardware dynamically through a software runtime. They can express a wide range of parallel patterns, provide automatic load balancing, and improve portability for legacy code [MRR12]. Unfortunately, manycore processors without hardware-based cache coherence require programmers to explicitly manage data coherence among private caches/memories and adopt a more restricted programming model, such as explicit task partitioning [KJJ+09], message passing [Olo16], or remote store programming [DXT+18]. The difficult programming model is arguably the primary reason why manycore processors without hardware-based cache coherence have not yet been widely accepted.

Furthermore, existing manycore processors without hardware cache coherence are generally used as a discrete co-processor, living in a separate memory space from the main general-purpose processor (i.e., host processor). Enabling efficient collaborative execution between the host and the manycore processor requires significant effort to bridge the gap between their disparate programming models and hide the data offloading latency [SFR$^+$14]. Recent work in *heterogeneous cache coherence (HCC)* has demonstrated that hardware-based cache coherence protocols can be seamlessly and efficiently integrated with software-centric cache coherence protocols on chip in a unified memory address space [ASA18]. While HCC solves the issue of data offloading by tightly integrating the host and the manycore, the programming model challenge remains to be addressed.

The focus of this chapter is to improve the performance and energy efficiency of work-stealing runtimes on heterogeneous cache-coherent multi/manycore systems. The Work-stealing runtimes discussed in this chapter allows dynamic task-parallel applications written for popular programming frameworks, such as Intel Cilk Plus and Intel TBB, to work collaboratively on both the big cores and the tiny cores without fundamental changes. However, as we will discuss in this chapter, heterogeneous cache coherent entails significant overheads when using work-stealing runtimes. Our approach, *direct-task stealing* (DTS), is based on the idea of using light-weight inter-processor user-level interrupts to bypass shared memory when performing work stealing. DTS enables some important optimizations in the work-stealing runtime to reduce synchronizations, therefore improving performance and energy efficiency of dynamic task

The contributions of this chapter are: (1) we provide, to the best of our knowledge, the first detailed description on how to implement work-stealing runtimes for HCC; (2) we propose a direct task stealing technique to improve performance and energy efficiency of dynamic task-parallel applications on manycore processors with HCC; and (3) we provide a detailed cycle-level evaluation on our technique.

3.2 Background

This section provides a background of several cache coherence protocols, HCC, and dynamic task parallelism. We first characterize four representative hardware-based and software-centric

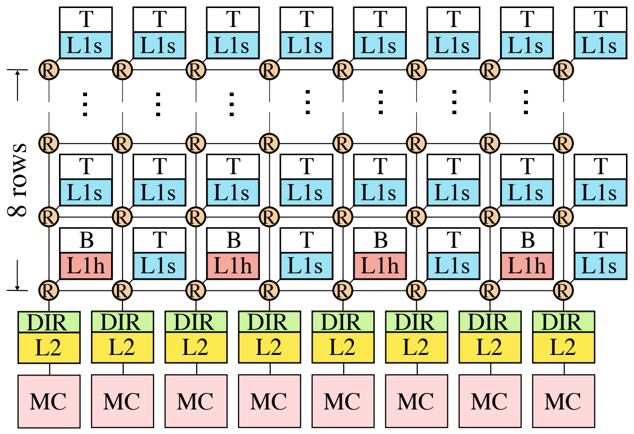


Figure 3.1: Block Diagram of a big.TINY Manycore System with HCC – T = in-order high efficiency tiny core; B = out-of-order high performance (big) core; R = on-chip interconnect router; L1s = private L1 cache with software-centric cache coherence; L1h = L1 cache with hardware-based cache coherence; DIR = directory; L2 = shared L2 cache bank; MC = main memory controller.

coherence protocols. We then describe some existing work on HCC systems. Lastly, we provide a brief overview of dynamic task parallelism exemplified by TBB/Cilk-like programming models.

3.2.1 Hardware-Based and Software-Centric Coherence

There are two types of cache coherence protocols: *hardware-based* and *software-centric*. In hardware-based protocols, data coherence among private caches is handled completely by hardware and transparent to software. General-purpose processors usually support hardware-based cache coherence protocols since they are easier to program. In contrast, in software-centric protocols, software is in charge of enforcing the coherence of shared data among private caches. Software-centric cache coherence protocols are relatively simple to implement in hardware but require more effort from software programmers.

We use a similar taxonomy described in previous work [ASA18] to categorize four representative coherence protocols: MESI, DeNovo [SA15], GPU-WT, and GPU-WB. A cache coherence protocol can be described using three properties: stale invalidation, dirty propagation, and write granularity. The *stale invalidation property* defines how and when stale data in a private cache is invalidated so that a read of the data returns its most up-to-date version. There are two ways to initiate an invalidation of stale data: writer-initiated and reader-initiated. In the first approach, a writer invalidates existing copies of the target data in all private caches prior to writing the data. This approach is used by hardware-based coherence protocols. The other approach is called reader-

Protocol	**Who initiates invalidation?**	**How is dirty data propagated?**	**Write Granularity**
MESI	Writer	Owner, Write-Back	Line
DeNovo	Reader	Owner, Write-Back	Word/Line
GPU-WT	Reader	No-Owner, Write-Through	Word
GPU-WB	Reader	No-Owner, Write-Back	Word

Table 3.1: Classification of Cache Coherence Protocols

initiated: a reader invalidates potentially stale data in its private cache before it reads the data. The *dirty propagation property* defines how and when dirty data becomes visible to other private caches. Some coherence protocols track which private cache owns dirty data, so that the owner can propagate the data to readers. This strategy is called ownership dirty propagation. Both MESI and DeNovo take this strategy to propagate dirty data. In contrast, GPU-WB and GPU-WT do not track the ownership of dirty data. Instead, they rely on the writer to write back dirty data. Writes can either be immediately sent to the shared cache (i.e., write-through), or the dirty data can be written back later using a special *flush* instruction. Lastly, the *write granularity property* defines the unit size of data at which writes are performed and the ownership of cache lines is managed.

We summarize the four coherence protocols, MESI, DeNovo, GPU-WT, and GPU-WB, in Table 3.1. These differences represent some fundamental design trade-offs between hardware-based and software-centric coherence protocols. In hardware-based protocols, the single-writer multiple-reader (SWMR) invariance [SHW11] is enforced. With the SWMR property, hardware-based coherence protocols are transparent to software, i.e., software can assume there is no cache at all. However, it requires additional hardware complexity, including communication and storage overheads (e.g., extra invalidation traffic, transient coherence states, and directory storage). On the other hand, software-centric protocols (such as DeNovo, GPU-WT, and GPU-WB) suffer less from these overheads: by using the reader-initiated stale invalidation strategy, these protocols do not need to track all readers of a particular cache line, saving both communication traffic in the interconnection network and directory storage. Neither GPU-WT nor GPU-WB requires tracking ownership for writes. DeNovo is a design point between MESI and GPU-WB/GPU-WT: it uses ownership dirty propagation (like MESI) to potentially improve performance of writes and atomic memory operations (AMO); and it uses reader-initiated stale invalidation (like GPU-WB/GPU-

WT) to reduce the invalidation overhead. However, software-centric coherence protocols push the complexity into software. Software needs to issue invalidation and/or flush requests at appropriate times to ensure coherence. GPU-WT and GPU-WB may also suffer from slower AMO performance. AMOs have to be handled in a globally shared cache, since private caches do not have ownership.

3.2.2 Heterogeneous Cache Coherence

Previous studies have explored different ways to integrate multiple coherence protocols into a heterogeneous cache-coherent system to serve a diversity of coherence requirements. IBM Coherent Accelerator Processor Interface (CAPI) provides a coherent MESI-based proxy interface for accelerators (e.g., FPGAs) to communicate with general-purpose processors through a shared last-level cache and directory [SBJS15]. Power et al. proposed a directory-based approach to maintain data coherence between a CPU and GPU at a coarse granularity (i.e., group of cache lines) [PBG+13]. Spandex [ASA18] is a recent proposal on providing a general coherence interface implemented in a shared last-level cache to coordinate different coherence protocols, where different protocols interact with the interface through their own translation units.

3.2.3 Programming Models for Dynamic Task Parallelism

Task parallelism is a style of parallel programming where the workload is divided into *tasks*, units of computation, that can be executed in parallel. In dynamic task parallelism, tasks and dependencies among tasks are generated at runtime. Tasks are dynamically assigned to available threads. The most common computation model for dynamic task parallelism is the *fork-join* model, which was initially used by MIT Cilk [BJK+95] and was later popularized by many modern programming frameworks, including Intel Cilk Plus [Lei09, int13], Intel TBB [Rei07, int15], and others [CGS+05, SML17]. Fork-join parallelism refers to a way of specifying parallel execution of a program: the program's control flow diverges (*forks*) into two or more flows that can be executed in parallel (but not necessarily); and then these control flows *join* into a single flow after they are finished. A task can fork by creating two or more parallel tasks, which is referred to as *spawning* tasks. The created tasks are called the *child* tasks (or simply *children*); the spawning task becomes the *parent* tasks of its children. The parent task waits until its children completes so that the children can join the parent. This model can serve as a basis to express many complex parallel patterns, including divide-and-conquer, parallel loop, reduction, and nesting [Rei07].

In programming frameworks with the fork-join model, parallel execution is realized by a runtime system using the *work-stealing* algorithm [BL99]. In work-stealing runtimes, each thread is associated with a *task queue* data structure to store tasks that are available for execution. The task queue is a double-ended queue (*deque*). When a task spawns a child task, it *enqueues* the child on to the task queue of the executing thread. When a thread is available, it first attempts to *dequeue* a task from its own deque in last-in-first-out (LIFO) order from one end. If the thread's own deque is empty, it tries to *steal* a task from another thread's deque in first-in-first-out (FIFO) order from the other end. The thread that steals becomes a *thief*, and the thread whose tasks are stolen becomes a *victim*. When a parent task is waiting for its children to join, the thread executing the parent can steal from other threads. This algorithm thus automatically balances the workload across threads. It leads to better locality and helps establish time and space bounds [BL99, FLR98].

3.3 Implementing Work-Stealing Runtimes on Heterogeneous Cache Coherence

This section gives a detailed description of our approach to implement a work-stealing runtime for HCC. We first show a baseline runtime for hardware-based cache coherence with a TBB/Cilk-like programming model. We then describe our implementation for heterogeneous cache coherence. We conclude this section with a qualitative analysis of the implications of HCC on work-stealing runtime systems.

3.3.1 Programming Example

We use the application programming interface (API) of Intel TBB to demonstrate our programming model (see Figure 3.2). Tasks are described by C++ classes derived from a base class, `task`, which has a virtual `execute()` method. Programmers override the `execute()` method to define the execution body of the task. Scheduling a new task is done by calling the `spawn(task* t)` function. Tasks are synchronized using the `wait()` function. The *reference count* tracks the number of unfinished children. When a parent task executes `wait()`, the execution of the parent task stalls until all of its children are finished. In addition to low-level APIs (e.g., `spawn`,

```
class FibTask : public task {
public:
    long* sum;
    int n;

    FibTask( int _n, long* _sum ) : n(_n), sum(_sum) {}

    void execute() {
      if ( n < 2 ) {
          *sum = n;
          return;
      }
      long x, y;
      FibTask a( n - 1, &x );
      FibTask b( n - 2, &y );
      this->reference_count = 2;
      task::spawn( &a );
      task::spawn( &b );
      task::wait( this );
      *sum = x + y;
  }
}
```

(a) `fib` using `spawn()` and `wait()`

```
long fib( int n ) {
    if ( n < 2 ) return n;
    long x, y;
    parallel_invoke(
      [&] { x = fib( n - 1 ); },
      [&] { y = fib( n - 2 ); }
    );
    return (x + y);
}
```

(b) `fib` using `parallel_invoke`

```
void vvadd( int a[], int b[], int dst[], int n ) {
    parallel_for( 0, n, [&]( int i ) {
        dst[i] = a[i] + b[i];
    });
}
```

(c) `vector-vector` add using `parallel_for`

Figure 3.2: A simple example for calculating the Fibonancci number using two different APIs – (a) a low-level API with explicit calls to `spawn` and `wait`; and (b) a high-level API with a generic templated `parallel_invoke` pattern. (c) shows an alternative generic templated `parallel_for` pattern.

```
void task::spawn(task* t) {
    tq[tid].lock_aq();
    tq[tid].enq(t);
    tq[tid].lock_rl();
}

void task::wait(task* p) {
    while ( p->rc > 0 ) {
        tq[tid].lock_aq();
        task* t = tq[tid].deq();
        tq[tid].lock_rl();

        if (t) {
           t->execute();
           amo_sub(t->p->rc, 1);
        }

        else {
            int vid = choose_victim();

            tq[vid].lock_aq();
            t = tq[vid].steal();
            tq[vid].lock_rl();

            if (t) {
                t->execute();
                amo_sub(t->p->rc, 1);
            }
        }
    }
}
```

Figure 3.3: A Work-Stealing Runtime Implementation for Hardware-Based Cache Coherence – p = parent task pointer; t = current task pointer; rc = reference count; tid = worker thread id; lock_aq = acquire lock; lock_rl = release lock; tq = array of task queues with one per worker thread; enq = enqueue on tail of task queue; deq = dequeue from tail of task queue, returns 0 if empty; choose_victim = random victim selection; vid = victim id; steal = dequeue from head of task queue; amo_or = atomic fetch-and-or; amo_sub = atomic fetch-and-sub.

wait), programmers can use higher-level templated functions that support various parallel patterns. For example, programmers can use parallel_for for parallel loops and parallel_invoke for divide-and-conquer.

3.3.2 Baseline Work-Stealing Runtime

Figure 3.3 shows an implementation, similar to Intel TBB, of the spawn and wait functions for hardware-based cache coherence. spawn pushes a task pointer onto the current thread's task deque, and `wait` causes the current thread to enter a *scheduling loop*. Inside the scheduling loop, a thread first checks if there is any task in its own deque. If so, it dequeues a task, in LIFO order, from its local deque to execute (lines 9–16). If there is none left in the local deque, the current thread becomes a thief and attempts to steal tasks from another thread in FIFO order (lines 19–28). When a task is executed, its parent's reference count is atomically decremented (line 27). When the reference count reaches zero, the parent task exits the scheduling loop and returns from `wait()`. A thread also exits the scheduling loop when the whole program is finished, and the main thread terminates all other threads.

3.3.3 Supporting Shared Task Queues on HCC

A task queue is a data structure shared by all threads. On processors with hardware-based cache coherence protocols, a per-deque lock implemented using atomic read-modify-write operations is sufficient for implementing proper synchronization (see lines 2, 4, 9, 11, 21, 23 of Figure 3.3). On processors with heterogeneous cache coherence, where some private caches use software-centric cache coherence protocols, additional coherence operations are required. Before a thread can access a task queue, in addition to acquiring the lock for mutual exclusion, all clean data in the private cache of the executing thread needs to be invalidated to prevent reading stale data. After a thread finishes accessing a task queue, in addition to releasing the lock, all dirty data needs to be written back (flushed) to the shared cache so that the data becomes visible to other threads. Figure 3.4 shows an implementation of `spawn` and `wait` for HCC protocols. We add an `invalidate` instruction following the lock acquire (e.g., lines 3 and 13), and a `flush` instruction before the lock release (e.g., lines 15 and 29). Note that not all protocols require `invalidate` and `flush`. On private caches with DeNovo, `flush` is not required because it uses an ownership stale-invalidation strategy (see Table 3.1). With DeNovo, `flush` can be treated as no-op. MESI protocol requires neither `flush` or `invalidate`, so both are treated as no-ops.

3.3.4 Supporting Task-Stealing on HCC

Adding proper `invalidate` and/or `flush` instructions along with per-deque locks ensures all task queues deques are coherent on HCC systems. However, user-level data needs to be coherent as well. Fortunately, the TBB programming model we use is structured by the DAG consistency

```
void task::spawn(task* t) {
    tq[tid].lock_aq();
    cache_invalidate();
    tq[tid].enq(t)
    cache_flush();
    tq[tid].lock_rl();
}

void task::wait(task* p) {
    while (amo_or(p->rc, 0) > 0) {

        tq[tid].lock_aq();
        cache_invalidate();
        task* t = tq[tid].deq();
        cache_flush();
        tq[tid].lock_rl();

        if (t) {
            t->execute();
            amo_sub(t->p->rc, 1);
        }

        else {
            int vid = choose_victim();

            tq[vid].lock_aq();
            cache_invalidate();
            t = tq[vid].steal()
            cache_flush();
            tq[vid].lock_rl();

            if (t) {
                cache_invalidate();
                t->execute();
                cache_flush();
                amo_sub(t->p->rc, 1);
            }
        }
    }
    cache_invalidate();
}
```

Figure 3.4: A Work-Stealing Runtime Implementation for Heterogeneous Cache Coherence – `cache_flush` = flush all dirty data in cache (no-op on MESI, DeNovo, and GPU-WT); `cache_invalidate` = invalidate all clean data in cache (no-op on MESI).

model [BFJ+96]. Informally speaking, the DAG consistency model means data sharing only exists between a parent task and its child task(s). To observe the DAG consistency, there are two requirements that the runtime system must fulfill: (1) child tasks need to see the latest values produced by their parent; and (2) a parent task needs to see all values produced by its children after the `wait()`. There is no synchronization between *sibling* tasks required because sibling tasks from the same parent must be data-race-free with regard to each other. This property allows us to correctly implement work-stealing runtimes on HCC without requiring changes in the user code. When a parent task spawns child tasks, the `flush` after the enqueue (see Figure 3.4, line 5) ensures data written by the parent task is visible to its child tasks, even if the children are stolen and executed by another thread. For requirement (2), the parent task needs to `invalidate` data in its own cache, in case any children are stolen and the parent may have stale data (line 40). Moreover, a thread needs to `invalidate` and `flush` before and after executing a stolen task respectively (lines 33 and 35), because the parent task is executed on a different thread (the victim thread).

3.3.5 HCC Performance Impacts

In this section, we qualitatively discuss the performance impacts of HCC on work-stealing runtimes. We defer the quantitative characterization to Section 3.6.

Reader-initiated invalidation strategy degrades performance in all three software-centric cache coherence protocols (i.e., DeNovo, GPU-WT, and GPU-WB) due to more cache misses. Every `spawn` and `wait` causes the executing thread to invalidate all data in the private cache, causing later reads to experience more cache misses.

Atomic operations may be slower. DeNovo uses the ownership dirty-propagation strategy, so the AMOs can be performed in private caches in the same way as MESI. However, GPU-WT and GPU-WB require AMOs to be performed at the shared cache, increasing the latency per operation.

Flushing is inefficient. DeNovo and GPU-WT do not need `flush` operations. GPU-WB, on the other hand, requires an explicit `flush` at every `spawn`, as well as after executing every stolen task. Writing back an entire private cache to the shared cache may require a significant amount of time and memory traffic, depending on the amount of data being written back.

Fine-grained tasks may exacerbate all of the above problems. Performance impacts discussed above are directly related to the task granularity. Finer-grained applications have many small tasks, and thus require a large number of AMOs, invalidations, and flushes. Therefore, HCC is expected to have more severe performance impacts on applications with fine-grained tasks.

3.4 Direct Task Stealing

In this section, we propose a hardware technique called direct task stealing (DTS) to improve the performance of work-stealing runtimes on HCC. DTS leverages the following properties of work-stealing runtimes: **(1) when parallelism is sufficient, the number of steals is relatively small compared to the number of local task enqueues and dequeues [BL99, FLR98]; and (2) for a steal, synchronization is only required between a victim thread and a thief thread, not among all threads**. As we have mentioned in the previous section, work-stealing may incur significant overhead in terms of performance and memory traffic on HCC. DTS addresses this issue by allowing tasks to be stolen *directly* using user-level interrupts (described below), rather than *indirectly* through shared memory.

3.4.1 Implementing Direct Task Stealing

A user-level interrupt (ULI) is a short, light-weight inter-processor interrupt. ULI is included in modern instruction-set architectures (ISA) such as RISC-V [ris19, CS12]. Like regular interrupts, user-level interrupts are handled asynchronously, and can be enabled and disabled on the receiving core by software. When a core has ULI enabled, and receives an ULI, its current executing thread is suspended, and the core jumps to a software-specified ULI handler. The only difference between ULI and regular interrupts is that ULIs are handled completely in user mode. The cost of handling an ULI is similar to regular context switching, minus the overheads associated with privilege mode changing.

Figure 3.5 shows an implementation of a work-stealing runtime with DTS. DTS uses ULI to perform work-stealing: when a thread attempts to steal a task, it sends a ULI to the victim thread (line 26). If the victim has ULI enabled, execution of the victim is redirected to a handler, shown in Figure 3.6. In the handler, the victim accesses its *own* task deque, retrieves a task, and sends the task to the thief through shared memory. The victim also sends an ACK message to the thief as a ULI response. With DTS, the victim steals tasks on behalf of the thief. If the victim has ULI

```
void task::spawn(task* t) {
    uli_disable();
    tq[tid].enq(t);
    uli_enable();
}

void task::wait(task* p) {
    int rc = p->rc;
    while (rc > 0) {

        uli_disable();
        task* t = tq[tid].deq();
        uli_enable();

        if (t) {
            t->execute();
            if (t->p->has_stolen_child);
                amo_sub(t->p->rc, 1);
            else
                t->p->rc -= 1;
        }

        else {
            int vid = choose_victim();

            uli_send_req(vid);
            t = read_stolen_task(tid);

            if (t) {
                cache_invalidate();
                t->execute();
                cache_flush();
                amo_sub(t->p->rc, 1);
            }
        }

        if (p->has_stolen_child)
            rc = amo_or(p->rc, 0);
        else
            rc = p->rc;
    }

    if (p->has_stolen_child)
        cache_invalidate();
}
```

Figure 3.5: A Work-Stealing Runtime Implementation with Direct Task Stealing – `uli_disable` = disable servicing ULI; `uli_enable` = enable servicing ULI; `read_stolen_task` = read stolen task from per-thread mailbox; `write_stolen_task` = store the stolen task into mailbox.

disabled, a NACK message is replied to the thief. In the rest of this section, we discuss why DTS can help reduce the overheads of work-stealing runtimes on HCC.

3.4.2 Optimizations to Reduce Task Queue Synchronization

DTS reduces the synchronization cost associated with task stealing on HCC. With DTS, task queues are no longer shared data structures. A task queue is only accessed by its owner, either for local accesses, or for steals through ULI. DTS therefore eliminates the need of synchronization (i.e., locks) on task queues. Accesses to task queues are kept mutually exclusive, by requiring a thread to disable ULI when it operates on its task deque (line 11). Work-stealing runtimes on HCC without DTS require every deque access, including ones made to a thread's own task deque, to have a pair of `invalidate` and `flush` (associated with lock acquire and release respectively, as in line 27 and 29 of Figure 3.4). In work-stealing runtimes with DTS, accessing task queues no longer incurs cache invalidations or flushs since task queues are private. DTS reduces cache misses on HCC caused by cache invalidation and flush, and thus improves performance and reduces memory traffic.

3.4.3 Optimizations to Reduce Parent-Child Synchronization

DTS offers an opportunity to further reduce synchronization cost for work stealing on HCC using software optimizations. In work-stealing runtimes without DTS, it is difficult to asses whether a task is actually stolen, since work stealing happens implicitly through shared memory. Without this information, a runtime must conservatively assume all tasks can potentially be stolen, and the runtime must always ensure proper synchronization between parent and child tasks. For example, the reference count in each task always needs to be updated using AMOs (Figure 3.4, line 20), and a parent task always invalidates its cache before returning from `wait` function in case of any of its children has been stolen (Figure 3.4, line 40).

DTS enables the runtime to track whether a task has been stolen and avoid synchronization entirely when a task is not stolen. This is particularly important to cache coherence protocols where flush (e.g., GPU-WB) or AMOs (e.g., GPU-WT and GPU-WB) are expensive. To track whether a task has been stolen, we add an auxiliary variable (`has_stolen_child`) to each task, indicating whether at least one of its child tasks has been stolen (Figure 3.5, line 37). Before

```
void uli_handler(int thief_id) {
    task* t = tq[tid].deq();
    if (t)
        t->p->has_stolen_child = 1;
    write_stolen_task(thief_id, t);
    cache_flush();
    uli_send_resp(thief_id);
}
```

Figure 3.6: ULI Handler for Direct Task Stealing – `uli_send_req` = send ULI request message and wait for response, calls `uli_handler` on receiver; `uli_send_resp` = send ULI response message.

sending the stolen task to the thief in the ULI handler, the victim sets `has_stolen_child` to true (line 50). The `has_stolen_child` variable is only accessed by the thread running the parent task, and thus can be modified with a regular load and store instead of an AMO.

According to the DAG-consistency model, if a child task is not stolen (i.e., `has_stolen_child` is false), it is not necessary to make its writes visible to other threads. Because its parent, the only task that needs to read these writes, is executed on the same thread as the child. A `flush` is only required when a task is actually stolen (Figure 3.5, line 51), instead of after each enqueue operation (Figure 3.4, line 5). If there are considerably more local enqueues and dequeues than steals, which is the common case when there is sufficient parallelism, DTS can significantly reduce the number of `flush`. Furthermore, if no child task has been stolen, it is unnecessary to perform an `invalidation` at the end of `wait()`; the parent task cannot possibly read stale data, since all of its child tasks are executed on the same thread as itself (Figure 3.5, line 43). Finally, if no child of a parent task is stolen, it is not necessary to update the parent's reference count using AMOs (line 20 and 40). Instead, the reference count can be treated as a local variable, because updates to it so far has been performed by the same thread.

In summary, DTS enables software optimizations to leverage properties of the DAG-consistency model in work-stealing runtimes to further reduce the overheads (i.e., invalidation, flush, and AMO) of work-stealing runtimes on HCC.

3.5 Evaluation Methodology

In this section, we describe our cycle-level performance modeling methodology used to quantitatively evaluate our proposal.

3.5.1 Simulated Hardware

We model manycore systems with the gem5 cycle-level simulator [BBB$^+$11]. We implement HCC protocols described in Section 3.2 using the Ruby memory modeling infrastructure. We use Garnet 2.0 [AKPJ09] to model on-chip interconnection networks (OCN). Our modeled systems have per-core private L1 caches and a shared banked L2 cache. All caches have a 64B cache line size.

We use the shared L2 cache to integrate different cache coherence protocols, similar to Spandex [ASA18]. The L2 cache supports different coherence request types required by the four cache coherence protocols we study in this work. The L2 cache in the baseline MESI protocol is inclusive of L1 caches. The L2 cache in HCC protocols is inclusive of MESI private L1 caches only. The directory is embedded in the L2 cache and has a precise sharer list for all MESI private L1 caches. There is no additional directory latency.

We implement the inter-processor ULI in our simulated manycore systems as specified by the RISC-V ISA [ris19]. We model the ULI network as a mesh network with two virtual channels (one for request and one for response to prevent deadlock) using Ruby `SimpleNetwork`. We assume the ULI network has a 1-cycle channel latency and a 1-cycle router latency. We also model the buffering effect in the ULI network. Each ULI message is single-word long. Only one ULI is allowed to be received by a core at any given time. We enhance each core with a simple hardware unit for sending and receiving ULI. A ULI is sent by writing to a dedicated control register. The hardware unit on each core has one buffer for requests and one buffer for responses. When the buffer is full, the receiver sends a NACK to senders.

We configure a 64-core big.TINY system with four out-of-order high-performance big cores, and 60 in-order tiny cores. A tiny core has a private L1 cache capacity of 4KB (i.e., 1/16 the capacity of a big core's L1 cache). The big and tiny cores are connected by an 8×8 on chip mesh network. Each column of the mesh is connected to a L2 cache bank and a DRAM controller. Figure 3.1 shows the schematic diagram of our simulated systems. Table 3.2 summarizes its key parameters.

We study the following big.TINY configurations: *big.TINY/MESI* is a system where both big and tiny cores are equipped with MESI hardware-based cache coherence; *big.TINY/HCC-dnv* is a big.TINY system with HCC, where big cores use MESI and tiny cores use DeNovo (DeNovoSync [SA15] variant); similarly, *big.TINY/HCC-gwt* and *big.TINY/HCC-gwb* are HCC config-

Tiny Core	RISC-V ISA (RV64GC), single-issue, in-order, single-cycle execute for non-memory inst. L1 cache: 1-cycle, 2-way, 4KB L1I and 4KB L1D, software-centric coherence;
Big Core	RISC-V ISA (RV64GC), 4-way out-of-order, 16-entry LSQ, 128 Physical Reg. 128-entry ROB. L1 cache: 1-cycle, 2-way, 64KB L1I and 64KB L1D, hardware-based coherence;
L2 Cache	Shared, 8-way, 8 banks, 512KB per bank, one bank per mesh column, support heterogeneous cache coherence;
OCN	8×8 mesh topology, XY routing, 16B per flit, 1-cycle channel latency, 1-cycle router latency;
Main Memory	8 DRAM controllers per chip, one per mesh column. 16GB/s total bandwidth.

Table 3.2: Simulator Configuration

urations with GPU-WT and GPU-WB on the tiny cores, respectively. In addition, we implement DTS proposed in Section 3.4 on each HCC configuration. We refer to those configurations with DTS as *big.TINY/HCC-DTS-dnv*, *big.TINY/HCC-DTS-gwt*, and *big.TINY/HCC-DTS-gwb*.

As a comparison to our big.TINY configurations, we also study a traditional multicore architecture with only big cores. *O3×1*, *O3×4*, and *O3×8* are configurations with one, four, and eight big cores, respectively. We use CACTI [MBJ09] to model the area of L1 caches. Our results show that a big core's 64KB L1 cache is 14.9× as large as a tiny core's 4KB L1 cache. Based on total L1 capacity and the CACTI results, we estimate that *O3×8* has similar area to our 64-core big.TINY configurations (4 big cores and 60 tiny cores).

Future manycore processors are likely to have hundreds to thousands of cores. To overcome the challenge of simulation speed with large systems, we use a *weak scaling* approach to only simulate a piece of the envisioned large-scale manycore systems. We scale down the core count and the memory bandwidth to the 64-core system described in Table 3.2. We also choose moderate input dataset sizes with moderate parallelism (see Table 3.3). We attempt to make our results representative of future manycore systems with more cores, more memory bandwidth, and running proportionally larger inputs. To validate our proposal in large systems, we also select a subset of the application kernels and evaluate them using larger datasets on a bigger 256-core big.TINY system.

					Cilkview			
Name	**Input**	**GS**	**PM**	**DInst**	**Work**	**Span**	**Para**	**IPT**
cilk5-cs	3000000	4096	ss	456M	524M	0.9M	612.1	31.9K
cilk5-lu	128	1	ss	155M	170M	4.8M	35.5	6.5K
cilk5-mm	256	32	ss	124M	184M	0.4M	449.3	8.7K
cilk5-mt	8000	256	ss	322M	416M	0.5M	829.3	135K
cilk5-nq	10	3	pf	100M	180M	0.7M	274.9	0.4K
ligra-bc	rMat_100K	32	pf	80M	129M	1.1M	117.9	0.4K
ligra-bf	rMat_200K	32	pf	151M	252M	1.2M	203.3	0.4K
ligra-bfs	rMat_800K	32	pf	236M	351M	0.9M	402.6	0.5K
ligra-bfsbv	rMat_500K	32	pf	152M	201M	0.7M	277.5	0.5K
ligra-cc	rMat_500K	32	pf	226M	278M	0.7M	383.5	0.6K
ligra-mis	rMat_100K	32	pf	183M	243M	1.3M	177.7	0.5K
ligra-radii	rMat_200K	32	pf	364M	437M	1.4M	311.4	0.7K
ligra-tc	rMat_200K	32	pf	286M	342M	1.0M	334.9	3.5K

Table 3.3: Simulated Application Kernels – Input = input dataset; GS = task granularity; PM = parallelization methods: pf = `parallel_for` and ss = recursive spawn-and-sync; DInsts = dynamic instruction count in millions; Work = total number of x86 instructions; Span = number of x86 instructions on the critical path; Para = logical parallelism, defined as work divided by span; IPT = average number of instructions per task; Work, span, and IPT are analyzed by Cilkview.

3.5.2 Work-Stealing Runtime Systems

We implement three variations of a C++ library-based work-stealing runtime described in Section 3.3. We have compared the performance of our baseline runtime system for hardware-based cache coherence (Figure 3.3 to Intel Cilk Plus and Intel TBB using applications we study in this paper running natively on an 18-core Xeon E7-8867 v4 processor, with dataset sizes appropriate for native execution. Our results show that our baseline work-stealing runtime has similar performance to Intel TBB and Intel Cilk Plus.

3.5.3 Benchmarks

We port 13 dynamic task-parallel applications from Cilk v5.4.6 [FLR98] and Ligra [SB13] to use our work-stealing runtime systems (see Table 3.3). We select applications with varied parallelization methods: applications from Cilk mainly use recursive spawn-and-sync parallelization (i.e., `parallel_invoke`); applications from Ligra mainly use loop-level parallelization (i.e., `parallel_for`). Ligra applications also exhibit non-determinism, as they typically use fine-grained synchronization such as compare-and-swap in their code. *cilk5-cs* performs parallel merge-sort algorithm; `lu` calculates LU matrix decomposition; *cilk5-mm* is blocked matrix multiplication; *cilk5-mt* is matrix transpose; *cilk5-nq* uses backtracking to solve the N-queen problem; *ligra-bc* calculates betweeness centrality of a graph; *ligra-bf* uses Bellman-Ford algorithm to calculate the single-source shortest path in a graph; *ligra-bfs* performs bread-first search on graphs; *ligra-bfsbv* is a bit-vector optimized version of bread-first search; *ligra-cc* computes connected components in graphs; *ligra-mis* solves the maximum independent set problem; *ligra-radii* computes the radius of a given graph; *ligra-tc* counts the number of triangles in a graph. A more detailed description for these benchmarks can be found in previous work [FLR98, SB13].

3.5.4 Task Granularity

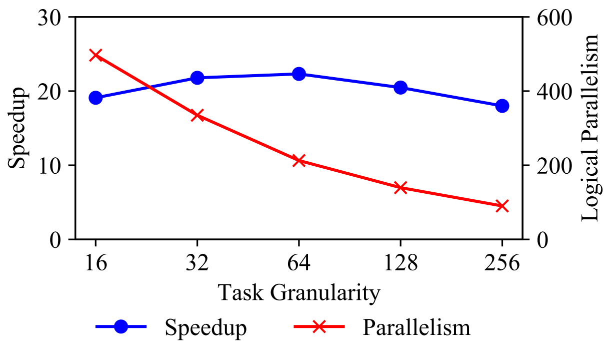


Figure 3.7: Speedup and Logical Parallelism of *ligra-tc* Running on a 64-core System – Task Granularity = the number of triangles processed by each task.

Task granularity (i.e., the size of the smallest task) is an important property of task-parallel applications. Programmers can control task granularity by dividing the work into more (fine-grained) or less (coarse-grained) tasks. Task granularity presents a fundamental trade-off: fine-grained tasks increase logical parallelism, but incur higher runtime overheads than coarse-grained tasks. We use a hybrid simulation-native approach to choose the task granularity for each application. We sweep the granularity and use Cilkview [HLL10] to analyze the logical parallelism. We evaluate the speedup over serial code for each granularity on a simulated manycore processor with 64 tiny cores. We select suitable granularity for each application to make sure it achieves the best or close to the best speedup over serial execution (see instruction-per-task (IPT) in Table 3.3). As an example, Figure 3.7 shows the speed up and the logical parallelism of *ligra-tc* with different granularity.

		Speedup over Serial IO				Speedup over b.T/MESI					
					b.T/	b.T/HCC			b.T/HCC-DTS		
Name	**Input**	**O3×1**	**O3×4**	**O3×8**	**MESI**	**dnv**	**gwt**	**gwb**	**dnv**	**gwt**	**gwb**
cilk5-cs	3000000	1.65	4.92	9.78	18.70	1.01	1.01	1.02	1.01	0.99	1.01
cilk5-lu	128	2.48	9.46	17.24	23.93	0.91	0.37	1.00	0.85	0.34	1.06
cilk5-mm	256	11.38	11.76	22.04	41.23	1.00	0.89	0.94	0.98	0.93	1.11
cilk5-mt	8000	5.71	19.70	39.94	57.43	0.71	1.05	0.69	0.72	1.04	0.70
cilk5-nq	10	1.57	3.87	7.03	2.93	1.01	1.18	1.09	0.56	1.52	1.76
ligra-bc	rMat_100K	2.05	6.29	13.06	11.48	0.96	0.96	1.01	1.01	1.25	1.60
ligra-bf	rMat_200K	1.80	5.36	11.25	12.80	0.97	0.95	0.98	0.89	1.10	1.32
ligra-bfs	rMat_800K	2.23	6.23	12.70	15.63	1.02	1.05	1.10	1.08	1.23	1.52
ligra-bfsbv	rMat_500K	1.91	6.17	12.25	14.42	1.01	0.98	0.98	1.00	1.06	1.18
ligra-cc	rMat_500K	3.00	9.11	20.66	24.12	0.82	0.94	0.99	0.91	1.08	1.24
ligra-mis	rMat_100K	2.43	7.70	15.61	19.01	0.88	0.89	0.92	0.97	1.07	1.35
ligra-radii	rMat_200K	2.80	8.17	17.89	25.94	0.83	0.81	0.85	1.00	1.03	1.17
ligra-tc	rMat_200K	1.49	4.99	10.89	23.21	1.01	0.86	0.98	1.07	0.92	1.05
geomean		2.56	7.26	14.70	16.94	0.93	0.89	0.96	0.91	1.00	1.21

Table 3.4: Speedup of Simulated Application Kernels – Input = input dataset; b.T = big.TINY; HCC = heterogeneous cache coherence; dnv = DeNovo; gwt = GPU-WT; gwb = GPU-WB.

It demonstrates that both a too big and a too small granularity lead to sub-optimal performance: the former due to lack of parallelism, and the latter due to runtime overheads. A smaller granularity penalizes HCC configurations more heavily, and the benefits of DTS technique would be more pronounced. Our choice of task granularity aims to optimize the performance of our baseline, not the relative benefits of our proposed DTS technique.

3.6 Results

Table 3.4 summarizes the speedup of the simulated configurations. Figure 3.8 illustrates the speedup of each big.TINY HCC configuration relative to *big.TINY/MESI*. Figure 3.9 shows the hit rate of L1 data caches. Figure 3.11 presents the execution time breakdown of the tiny cores. Figure 3.10 shows the total memory traffic (in bytes) on the on-chip network.

3.6.1 Baseline Runtime on big.TINY/MESI

On 11 out of 13 applications, *big.TINY/MESI* has better performance than *O3*$\times$*8*. The baseline work-stealing runtime enables collaborative execution and load balancing between the big and tiny cores in *big.TINY/MESI*. *cilk5-nq* performs worse on *big.TINY/MESI* than *O3*$\times$*8* because the runtime overheads outweigh the parallel speedup (as discussed in Section 3.5.4). Overall, our *big.TINY/MESI* vs. *O3*$\times$*8* results demonstrate the effectiveness of unlocking more parallelism using a big.TINY system compared to an area-equivalent traditional multi-core configuration *O3*$\times$*8*.

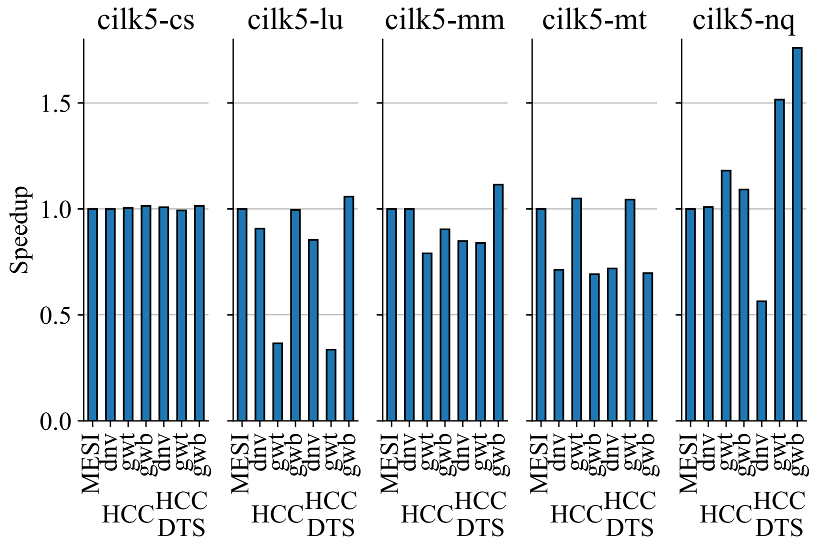
3.6.2 Work-Stealing Runtime on HCC

We now discuss our work-stealing runtime on HCC (shown in Figure 3.4 by analyzing the performance and energy of *big.TINY/HCC-dnv*, *big.TINY/HCC-gwt*, and *big.TINY/HCC-gwb*.

Compared with *big.TINY/MESI*, *big.TINY/HCC-dnv* has decreased L1 hit rate due to its reader-initiated invalidation strategy, as shown in Figure 3.9. This decrease in L1 hit rate causes a slight increase in memory traffic, as shown in the *cpu_req* and *data_resp* categories in Figure 3.10. The impact of these negative effects on performance is modest on most of the applications, except for *cilk5-mt*. *cilk5-mt* has a significant performance degradation due to additional write misses caused by invalidation. This effect can be seen in the increased data store latency and write-back traffic (see Figure 3.10).

big.TINY/HCC-gwt is a write-through and no write-allocate protocol. In GPU-WT, a write miss does not refill the cache. Therefore, *big.TINY/HCC-gwt* is unable to exploit temporal locality in writes, resulting in significantly lower L1 hit rate compared to both *big.TINY/MESI* and *big.TINY/HCC-dnv*. The network traffic of *big.TINY/HCC-gwt* is also significantly higher than others, especially in the *wb_req* category. The reason is every write (regardless of hit or miss) updates the shared cache (write-through). The latency for AMOs and network traffic are also increased (shown in Figure 3.11 and Figure 3.10 respectively). *big.TINY/HCC-gwt* has slightly worse performance and significantly more network traffic compared to *big.TINY/MESI* and *big.TINY/HCC-dnv* in all applications except *cilk5-lu*, where it performs significantly worse.

big.TINY/HCC-gwb has similar performance to *big.TINY/HCC-gwt* when dealing with AMOs. However, the write-back policy allows *big.TINY/HCC-gwb* to better exploit temporal locality. On all applications except *cilk5-mt*, *big.TINY/HCC-gwb* has less memory traffic, higher L1 hit rate,



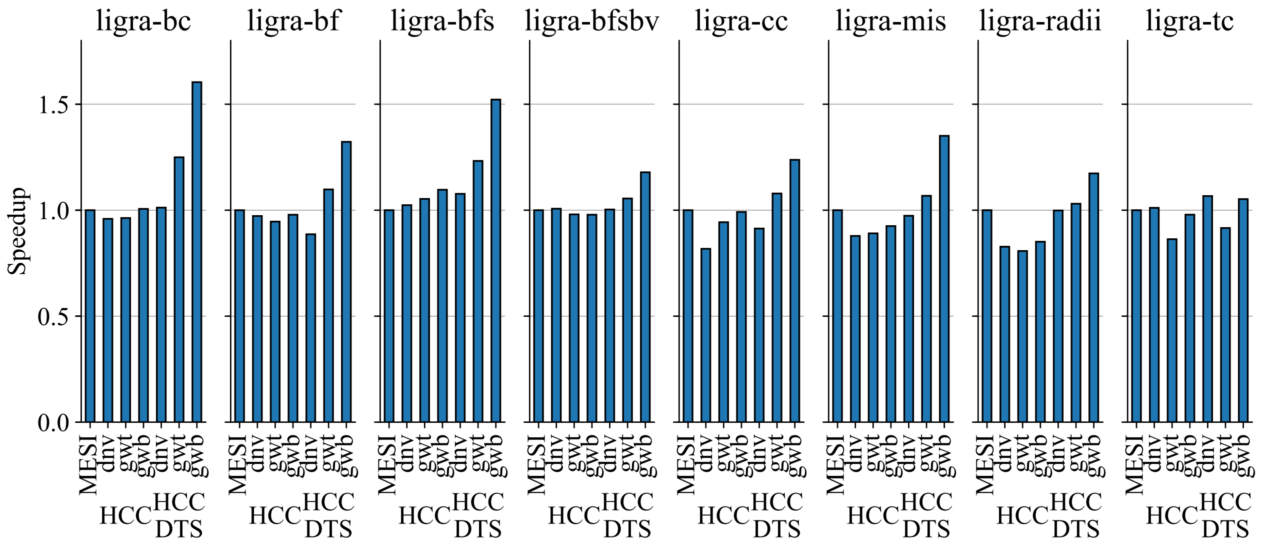


Figure 3.8: Speedup Over big.TINY/MESI – MESI = big.TINY/MESI; HCC = configurations with heterogeneous cache coherence; DTS = direct task stealing; dnv = tiny cores use DeNovo protocol; gwt = tiny cores use GPU-WT protocol; gwb = tiny cores use GPU-WB protocol.

and better performance than *big.TINY/HCC-gwt*. *big.TINY/HCC-gwb* is less efficient in memory traffic compared *big.TINY/HCC-dnv* due to its lack of ownership tracking: every private cache needs to propagate dirty data through the shared cache.

In summary, our baseline work-stealing runtime on HCC has moderately worse performance than the *big.TINY/MESI* configuration on almost all applications. These results demonstrate that HCC can effectively reduce hardware complexity with a small performance and energy penalty.

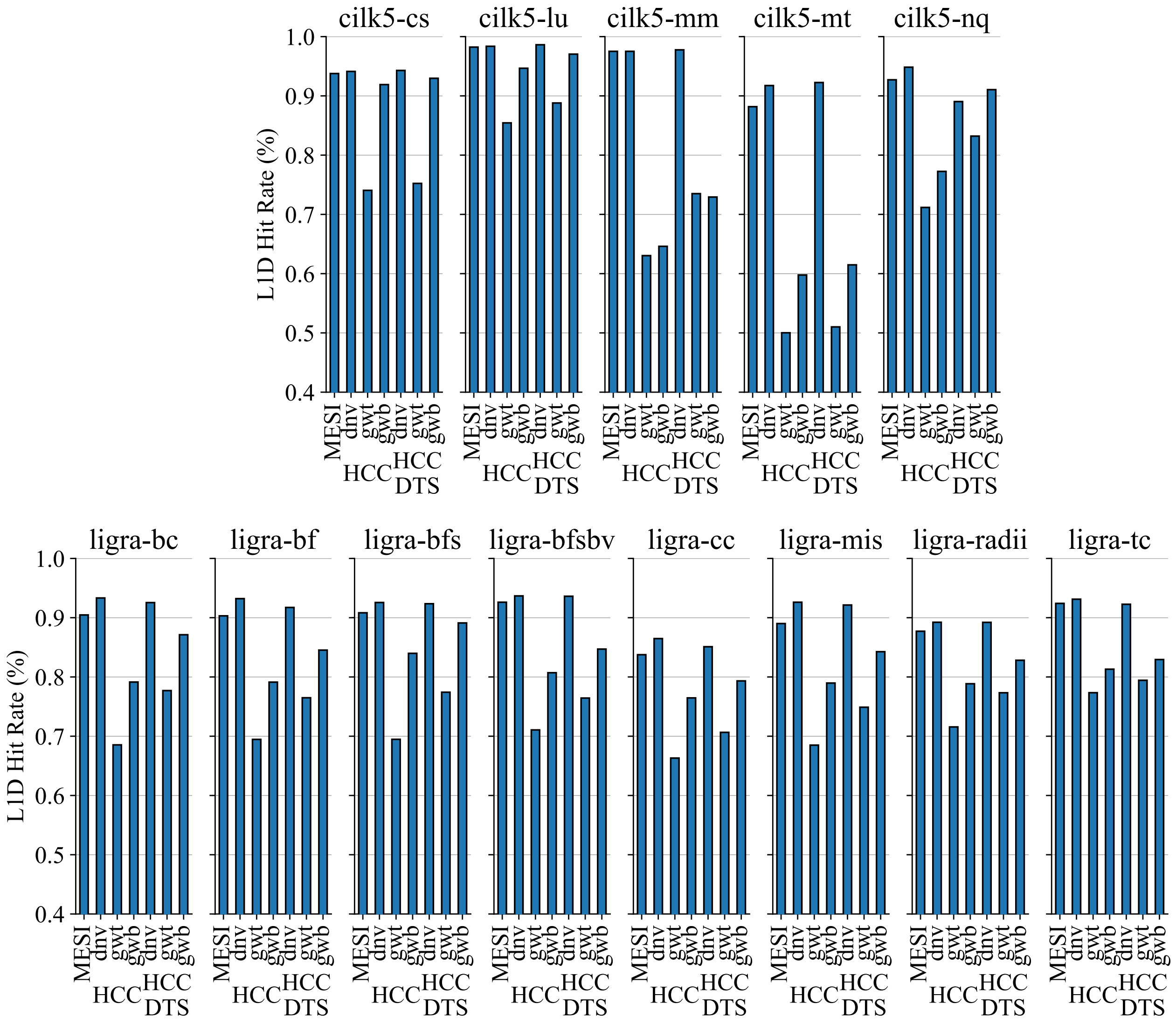


Figure 3.9: L1 Data (L1D) Cache Hit Rate – MESI = big.TINY/MESI; HCC = configurations with heterogeneous cache coherence; DTS = direct task stealing; dnv = tiny cores use DeNovo protocol; gwt = tiny cores use GPU-WT protocol; gwb = tiny cores use GPU-WB protocol.

3.6.3 Evaluation of HCC with DTS

In Section 3.4, we motivate DTS by observing that synchronization is only needed when a task is stolen. DTS avoids using cache invalidations and/or flushes unless a steal actually happens. We compare the results of HCC configurations without DTS (*big.TINY/HCC-**) with those with DTS (*big.TINY/HCC-DTS-**). We profile the number of invalidated and flushed cache lines for each configuration. We summarize the reduction in the number of invalidations and flushes in Table 3.5. We also calculate the increase in L1 hit rate of *big.TINY/HCC-DTS-** configurations compared with corresponding *big.TINY/HCC-** (HCC without DTS) configurations.

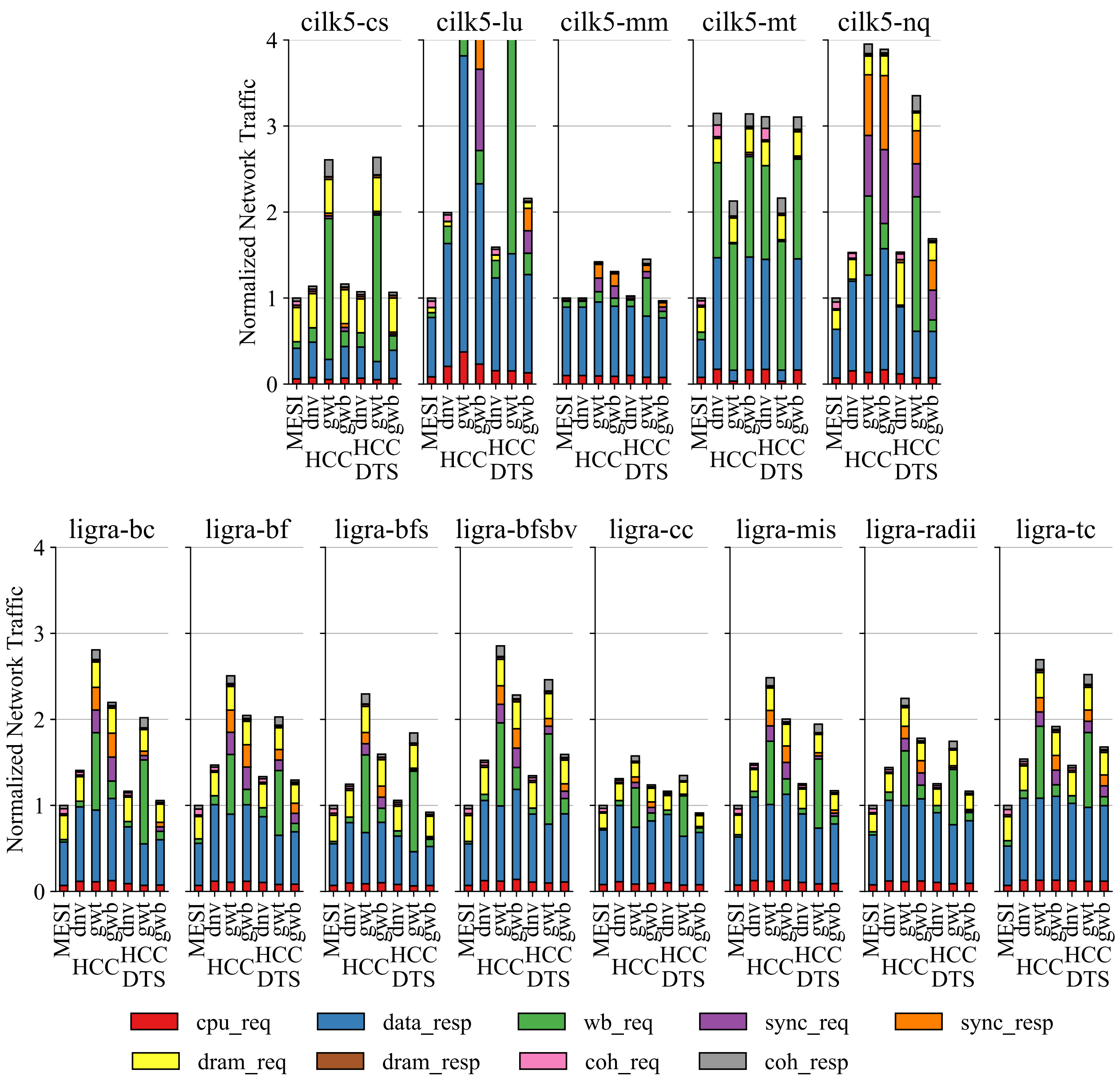


Figure 3.10: Total On-Chip Network Traffic (in Bytes) Normalized to big.TINY/MESI – *cpu_req* = requests from L1 to L2; *data_resp* = response from L2 to L1; *wb_req* = request for write-back data; *sync_req* = synchronization request; *sync_resp* = synchronization response; *dram_req* = request from L2 to DRAM; *dram_resp* = response from DRAM to L2; *coh_req* = coherence request; *coh_resp* = coherence response; MESI = big.TINY/MESI; HCC = configurations with heterogeneous cache coherence; DTS = direct task stealing; dnv = tiny cores use DeNovo protocol; gwt = tiny cores use GPU-WT protocol; gwb = tiny cores use GPU-WB protocol.

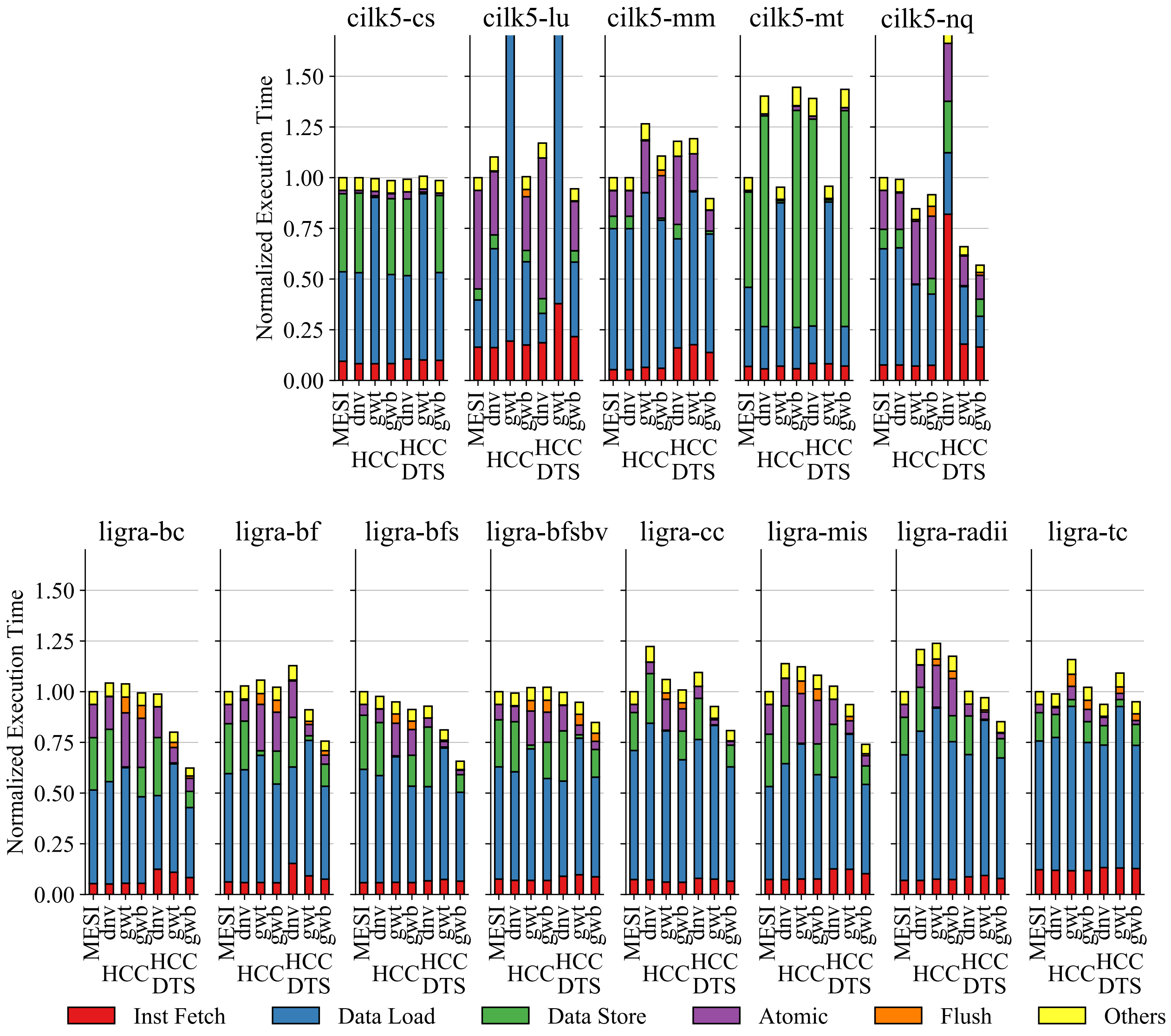


Figure 3.11: Aggregated Tiny Core Execution Time Breakdown Normalized to big.TINY/MESI – MESI = big.TINY/MESI; HCC = configurations with heterogeneous cache coherence; DTS = direct task stealing; dnv = tiny cores use DeNovo protocol; gwt = tiny cores use GPU-WT protocol; gwb = tiny cores use GPU-WB protocol.

In all three HCC protocols across all benchmarks, *big.TINY/HCC-DTS-** have significantly lower number of cache invalidations. *ligra-bf* and *ligra-bfsbv* show a reduction of 30–50%. *ligra-tc* has a reduction of 10–20%. The rest of the benchmarks each has more than 90% reduction. The number of flushes is also reduced on *big.TINY/HCC-DTS-gwb*. 10 of 13 benchmarks have a reduction of more than 90%. In *ligra-tc*, *ligra-bfsbf*, and *ligra-bf*, DTS achieves less flush reduction due to the relatively higher number of steals.

Table 3.5 shows the reduction in invalidations translates to higher L1 hit rate. The effect of increasing L1 hit rate is less significant on *big.TINY/HCC-DTS-dnv* because it has higher L1 hit

	InvDec (%)			FlsDec (%)	HitRateInc (%)		
App	**dnv**	**gwt**	**gwb**	**gwb**	**dnv**	**gwt**	**gwb**
cilk5-cs	99.42	99.28	99.50	98.86	1.80	2.45	1.30
cilk5-lu	98.83	99.78	99.53	98.40	1.12	7.12	2.94
cilk5-mm	99.22	99.67	99.62	99.12	30.03	42.19	36.80
cilk5-mt	99.88	99.73	99.93	99.82	12.45	2.70	6.56
cilk5-nq	97.74	97.88	98.32	95.84	16.84	28.87	27.04
ligra-bc	94.89	97.04	97.33	93.80	7.64	21.43	14.99
ligra-bf	29.02	38.14	40.24	21.63	7.22	17.14	11.17
ligra-bfs	94.18	95.85	95.90	91.23	3.48	15.76	8.00
ligra-bfsbv	39.31	47.36	50.74	29.46	3.10	12.65	7.56
ligra-cc	98.03	98.17	98.16	95.89	3.11	11.11	6.17
ligra-mis	97.35	98.28	98.36	96.16	5.62	16.29	11.10
ligra-radii	95.97	98.17	98.19	95.75	3.62	11.00	7.03
ligra-tc	10.83	15.99	17.02	7.52	1.59	3.55	3.02

Table 3.5: Cache Invalidation, Flush, and Hit Rate – Comparisons of big.TINY/HCC-DTS with big.TINY/HCC. InvDec = % decrease in cache line invalidations. FlsDec = % decrease in cache line flushes. HitRateInc = % increase in L1 D$ hit rate.

rate to begin with (due to its ownership-based dirty propagation). The increase in L1 hit rates also leads to reduced network traffic. Figure 3.10 shows *big.TINY/HCC-DTS-** have reduced network traffic in *cpu_req* and *data_resp*. In *big.TINY/HCC-DTS-gwb*, *wb_req* traffic is also significantly reduced, due to the reduction in flushes. However, DTS cannot help reduce *wb_req* traffic in *big.TINY/HCC-DTS-gwt* since each write still causes a write-through to the shared cache.

As we discuss in Section 3.4, DTS also enables runtime optimizations to avoid using AMOs for the reference count unless a child is stolen. *big.TINY/HCC-DTS-gwt* and *big.TINY/HCC-DTS-gwb* benefit from this optimization since the AMO latency is high in these two protocols. In Figure 3.10, we can see the reduction of traffic due to less AMO requests. Figure 3.11 shows the reduced execution time in the `Atomic` category.

Out of the three HCC protocols, *big.TINY/HCC-DTS-gwb* benefits the most from DTS. It can leverage reduction in invalidations, flushes, and AMOs. *big.TINY/HCC-gwt* and *big.TINY/HCC-dnv* benefit from DTS less because they do not need flushes. *big.TINY/HCC-gwt* benefit from reduction in AMOs, but its *wb_req* traffic is unaffected by DTS.

We measure the overhead of DTS in our simulations. In all applications and configurations, we observe that the ULI network has less than 5% network utilization rate, indicating DTS is

			Speedup		
			vs O3×1	vs b.T/MESI	
Name	**Input**	**DInst**	**b.T/MESI**	**HCC-gwb**	**HCC-DTS-gwb**
cilk5-cs	6000000	2.22×	27.7	0.94	1.07
ligra-bc	rMat_1M	8.65×	14.3	0.96	1.61
ligra-bfs	rMat_3M	2.90×	13.5	1.04	1.78
ligra-cc	rMat_1M	1.99×	27.7	0.92	1.26
ligra-tc	rMat_1M	6.05×	18.5	0.69	0.76

Table 3.6: Results for A 256-core big.TINY System – Results of using bigger datasets on a 256-core big.TINY manycore processor. The processor consists of four big cores and 252 tiny cores, configured in a 8-row, 32-column mesh. It has 32 L2 banks (16MB total L2 capacity) and 32 DRAM controllers (see Figure 3.1). Big cores and tiny cores use parameters in Table 3.2. Input = input datasets; DInsts = dynamic instruction count relative to the smaller datasets in Table 3.3; b.T = big.TINY; HCC = heterogeneous cache coherence; DTS = direct task stealing; gwb = GPU-WB.

infrequent. The average latency of the ULI network is around 20 hops (50 cycles). DTS does not incur cache invalidations or flushes on the big cores, where hardware-based coherence is available. It takes a few cycles to interrupt the tiny core, and 10 to 50 cycles to interrupt the big core, since an interrupt needs to wait until all instructions in-flight in the processor pipeline are committed before jumping to the handler. The total number of cycles spent on DTS is less than 1% of the total execution time, and thus DTS has minimal performance overhead.

3.6.4 Results on Larger-Scale Systems

To evaluate our techniques on a larger-scale system, we select five application kernels with larger input datasets, and execute these kernels on a 256-core big.TINY system. Compared to the 64-core big.TINY system we use for the rest of the paper, the 256-core big.TINY system has four big cores, 252 tiny cores, 4× the memory bandwidth, and 4× the number of L2 banks. We scale up the input sizes to increase the dynamic instruction count and amount of parallelism, in order to approximately achieve *weak scaling*. For HCC configurations, we select the best performing HCC protocol, GPU-WB. The results are presented in Table 3.6. A big core has 16× the L1 cache capacity as a tiny core (64KB vs. 4KB), and therefore the total L1 cache capacity of the 256-core big.TINY system is equivalent to 20 big cores. The results demonstrate that, even in a larger-scale big.TINY system, HCC with our work-stealing runtime allows dynamic task-parallel applications to achieve similar performance with simpler hardware compared to big.TINY/MESI. Our DTS technique significantly improves performance of our work-stealing runtime on HCC. The relative benefit of DTS is *more* pronounced in the 256-core big.TINY system than in the 64-core system, because stealing overheads without DTS are higher in systems with more cores.

3.6.5 Summary

Our overall results show that on big.TINY systems, combining HCC and work-stealing runtimes allows dynamic task-parallel applications with TBB/Cilk-like programming models to work with simpler hardware compared to hardware-based coherence, at the cost of slightly worse performance. The relative performance and energy efficiency of three HCC protocols depend on the characteristics of the application. DTS reduces the number of invalidations on all three HCC protocols. It also reduces the number of flushes on *big.TINY/HCC-gwb*. DTS is able to close the gap and enables HCC systems to match or even exceed the performance of fully hardware-based cache coherent systems. Regarding the geometric mean of all apps, the best performing HCC protocol combined with DTS achieves 21% improvement in performance, and similar amount of network traffic compared to hardware-based cache coherence.

3.7 Related Work

There is a large body of work on optimizing hardware-based cache coherence protocols. Coarse-grained sharer list approaches reduce storage overhead for tracking sharers of each cache line, at the expense of unnecessary invalidations and broadcasts [GdWM90, ZSD10, CLS05, ZSM07, Mos05]. Hierarchical directories attacks the same overhead by adding additional levels of indirection in the directories [MHS12]. Techniques for increasing directory utilization allow for smaller directories, but increase hardware complexity [SK12, FLKBF11]. Prior work like SWEL propose removing the sharer list altogether [CRG+11, PSNB10]. However, those techniques perform well only if most of the cache lines are not shared. Other optimization techniques, such as dynamic self-invalidation [LW95], token coherence [MHW03, RBM08], and destination-set prediction [MHS+03], have also been proposed for systems with sequential consistency (SC). There has been work focusing on optimizing banked shared caches, particularly those with non-uniform cache access (NUCA), using static and/or dynamic techniques. Jigsaw [BS13] addresses the scalability and interference problem in shared caches by allowing software to control data placement in collections of bank partitions. Coherence domain restriction [FW15] improves scalability of shared caches by restricting data sharing. Whirlpool [MBS16] leverages both static and dynamic information to reduce data movement between shared caches. HCC used in our work primarily addresses scalability of private caches and can be complementary to these NUCA techniques in shared caches.

Techniques to improve hardware-based cache coherence protocols by leveraging relaxed consistency models (e.g., release consistency (RC) [KCZ92, KK10, RK12, DWB+91, GLL+90], entry consistency [BZS93], and scope consistency [ISL96]) have been proposed in the literature as well. An important observation of this prior work is, unlike in the case of SC, cache coherence only needs to be enforced at synchronization points for relaxed consistency models. Some prior work proposed to use self-invalidation at acquires in RC to remove the need of tracking sharers [KCZ92, KK10, RK12]. SARC [KK10] and VIPS [RK12] further eliminate the directory all together by leveraging a write-through policy. TSO-CC is another self-invalidation based protocol designed for the total store ordering (TSO) consistency model [EN14]. To maintain the stronger TSO consistency, however, TSO-CC needs additional hardware logic such as timestamps and epoch tables. Our work-stealing runtime system leverages the DAG-consistency [BFJ+96] model for heterogeneous coherence.

There have been several software-centric coherence protocols in prior work. DeNovo [CKS+11] uses self-invalidation to eliminate sharer-tracking. In addition, by requiring the software to be data-race-free (DRF), DeNovo eliminates transient states. While DeNovo greatly simplifies the hardware, the DRF requirement limits its software scope. To address this problem, follow-up work on DeNovo (i.e., DeNovo-ND [SKA13] and DeNovoSync [SA15]) added support for lock-based non-determinism and arbitrary synchronization. We used DeNovoSync in our studies. GPU coherence protocols combine write-through and self-invalidation, and achieve the simplest hardware suitable for GPU workloads [SSF+13, SAA17]. Our GPU-WT protocol is similar to the ones described in the literature. GPU-WB differs from GPU-WT in deferring the write-through until barriers. GPU-WB is studied in manycore systems with a single globally shared task queue [KJJ+09].

Prior work has proposed efficient heterogeneous coherence protocols for integrated GPU-CPU systems [PBG+13, HCH+14], and accelerators [KSV15]. Spandex explores an efficient interface for various hardware-based and software-centric cache coherence protocols [ASA18]. Similar to this prior work, the heterogeneous coherence protocol we described in this paper uses a MESI-based LLC to integrate different cache coherence protocols in private caches.

Past work has looked at improving task-based programming models in various ways such as improving the scheduling algorithms [BL99,FLR98,WCD+10], improving the efficiency of software-based task queues [ACR13,CL05], and reducing the overhead of memory fences [LMLV11]. Carbon [KHN07] improves the performance of fine-grained tasks. ADM [SYK10] uses active messages [vECGS92] to improve task scheduling. While both Carbon and ADM provide a task-based model, their programming frameworks are drastically different from widely-used ones like Cilk or TBB. Prior work has also explored improving work-stealing runtime systems for heterogeneous system with different core architectures [TWB16]. We implemented our proposed direct task-stealing (DTS) mechanism with user-level interrupt [CS12], which is similar to but much simpler than active messages used by ADM.

3.8 Conclusions

This chapter has demonstrated how careful hardware/software co-design can enable efficiently exploiting dynamic task parallelism on heterogeneous multi/manycore systems. This work provides a small yet important step towards ushering in an era of complexity-effective big.TINY architectures that combine a few big out-of-order high-performance cores with many tiny energy-efficient cores, while at the same time supporting highly productive programming frameworks. Our vision is of a future where programmers use traditional dynamic task parallel programming frameworks to improve their performance across a few big cores, and then seamlessly without effort these applications can see significant performance improvements (our results suggest up to 2–3× comparing *big.TINY/HCC-DTS-gwb* to *O3×4*) by simply allowing collaborative execution across big and tiny cores using work stealing.

CHAPTER 4 **OPENPITON-HCC: EVALUATING THE AREA AND TIMING OF HETEROGENEOUS MULTI/MANYCORE SYSTEMS**

Chapter 3 presented a combination of hardware and software techniques to enable efficient co-operative execution of fine-grained task parallelism on heterogeneous multi/manycore systems. It presented a set of promising performance and energy results using a cycle-level evaluation methodology. This chapter discusses in detail how to realistically build hardware prototypes of heterogeneous multi/manycore systems at the register-transfer level (RTL). This chapter is built on an existing open-source hardware prototyping platform, OpenPiton [BMF+16, BMF+19]. OpenPiton offers a complete infrastructure to build cache-coherent multicore prototypes. In order to implement heterogeneous multi/manycore systems, we extended OpenPiton to support heterogeneous cache coherence. The new platform is called *OpenPiton-HCC*. OpenPiton-HCC enables prototypes of *little.TINY* multi/manycore systems, which consists of Linux-capable hardware-cache-coherent in-order multicore processors (little) and smaller software-cache-coherent manycore processors (tiny). Using OpenPiton-HCC, I present a quantitative evaluation of the area and timing of several little.TINY heterogeneous multi/manycore systems. I then build an area model based on the VLSI results which shows the potential of heterogeneous multi/manycore approach in future systems.

Section 4.1 presents a brief background on the OpenPiton hardware prototyping framework. Section 4.2 discusses the details of implementing heterogenous cache coherence in OpenPiton. Section 4.3 presents tile-level VLSI area and timing results. Section 4.4 discusses chip-level area results of multiple configurations of little.TINY systems. Section 4.5 presents an area model built upon VLSI results that shows the heterogeneous multi/manycore approach can increase parallelism given the same area.

4.1 Background

This section presents a background of the OpenPiton hardware prototyping platform. OpenPiton is the first open-source research platform that facilitates cache-coherent multicore prototypes. It offers an open-source Linux-capable generous-purpose processor, OpenSPARC T1 [ora20], a scalable network-on-chip (NoC), a MESI hardware-based cache coherent protocol, and other un-

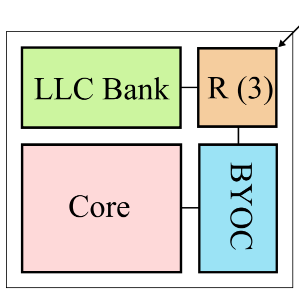


Figure 4.1: Architecture of An OpenPiton Tile – An OpenPiton tile has one or more cores, a BYOC cache, a bank of shared LLC, and three NOC routers.

core modules (e.g., interrupt controller, IO bridges, etc.). It also comes with an extensive set of verification infrastructure. OpenPiton also provides a "bring your own core" (BYOC) [BLS$^+$20] interface that allows new processor cores with different ISAs and memory interfaces to be added to the OpenPiton framework. OpenPiton is therefore a solid foundation for building prototypes of heterogeneous multi/manycore systems.

4.1.1 OpenPiton Overview

OpenPiton has a tiled architecture. An tile consists of one or multiple cores, a BYOC private cache, a bank of distributed shared last-level cache (LLC), and NoC routers. Tiles are connected via three NoCs in a 2D mesh topology. The three NoCs are used to support a directory-based MESI cache coherence protocol. Figure 4.1 presents a schematic diagram of a tile in OpenPiton.

The tiled architecture enables OpenPiton to easily scale up both within a chip and across multiple chips connected by provided chipset modules. Currently, the NoCs in OpenPiton have an address space supporting up to 65,536 tiles (256$\times$256 2D mesh) within a chip. In the following sections, I will discuss the components of OpenPiton that are relevant to the exploration of heterogeneous multi/manycore systems.

4.1.2 Network-On-Chip

The NoCs in OpenPiton provide communication between the tiles for cache coherence, I/O, memory accesses, and inter-core interrupts. They also support packets destined for off-chip to the chip bridge. The NoCs maintain point-to-point ordering between a single source and destination, a feature required by many cache coherence protocols. Each NoC is a single physical network without virtual channels, and consists of two 64-bit uni-directional links, one in each direction.

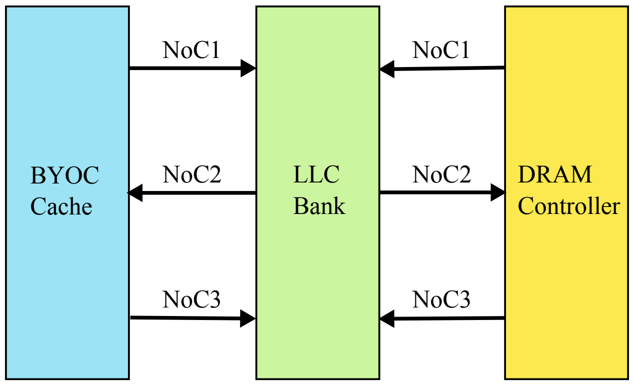


Figure 4.2: OpenPiton's Memory Hierarchy – Components of OpenPiton's memory hierarchy are connected using three NoCs: NoC1, NoC2, and NoC3.

The links use credit-based flow control. Packets are routed using dimension-ordered wormhole routing to ensure a deadlock-free network. The packet used by the OpenPiton NoC has 29 bits in total (8 bits for X- and Y-dimension within a chip and 13 bits for inter-chip address) of router addressability, making it scalable up to 500 million cores.

4.1.3 Cache Coherent Protocol

The OpenPiton platform uses a directory-based MESI cache coherence protocol that operates among the BYOC private caches, the distributed (banked) shared LLC, and main memory controllers. It uses three NoCs with point-to-point ordering to prevent deadlock. The LLC is inclusive and the directory information is embedded in each LLC cache line. It uses a four-hop design, meaning all communication between BYOC caches is through the LLC, and there is no direct communication between private BYOC caches. Figure 4.2 illustrates how the BYOC caches, distributed LLC, and memory controllers are connected using three NoCs.

Table 4.1 summarizes coherence operations in OpenPiton's memory hierarchy. The priority order of three NoCs from high to low is NoC3, NoC2, and NoC1. To prevent deadlock, a request on a NoC of lower priority cannot block a request on a NoC of higher priority. The highest-priority NoC, NoC3, is never blocked by the other two NoCs. Cache lines can have four states: modified (M), exclusive (E), shared (S), and invalid (I), as in the classical MESI protocol [SHW11].

A BYOC cache line transitions from I state to S state when issuing a read request, `ReqRd`. If the directory in the LLC is already in S state, it directly returns the requested data using `AckDt`. This results in a two-hop transaction. Otherwise, the directory needs to send a downgrade request to the

	NoC Name	Contains Data	Operations
`ReqRd`	NoC1	No	Request cache line for shared state
`ReqEx`	NoC1	No	Request cache line for exclusive state
`ReqAMO`	NoC1	Yes	Atomic memory operation request
`WBGuard`	NoC1	No	Prevent read from overtaking writeback
`FwdRd`	NoC2	No	Fowarded read reequest
`FwdEx`	NoC2	No	Fowarded exclusive reequest
`Inv`	NoC2	No	Invalidation of a cache line
`LdMem`	NoC2	No	Load cache line reqeust to memory controller
`StMem`	NoC2	Yes	Store cache line reqeust to memory controller
`Ack`	NoC2	No	Acknowledges BYOC request without data
`AckDt`	NoC2	Yes	Acknowledges BYOC request with data
`FwdRdAck`	NoC3	Yes	Acknowledges `FwdRd` from LLC
`FwdExAck`	NoC3	Yes	Acknowledges `FwdEx` from LLC
`LdMemAck`	NoC3	Yes	Memory acknowledges `LdMem` from LLC
`StMemAck`	NoC3	No	Memory acknowledges `StMem` from LLC
`WB`	NoC3	Yes	BYOC cache evicts/writes back dirty data

Table 4.1: Coherence Operations of OpenPiton – NoC1 messages are initiated by the private cache (BYOC) to the shared LLC. NoC2 messages are initiated by LLC to private BYOC caches, or the memory controllers. NoC3 messages are responses from BYOC or memory controllers to the shared LLC.

owner (a remote BYOC cache) first using `FwdRd` which results in a 4-hop transaction. Notice that if the directory is in E state, the owner could be either in E or M state because of the possible silent transition from E to M state.

A BYOC cache line transitions from I state to E state when issuing a read request `ReqRd` and the cache line is in I state in the directory. If the directory is in I state but the cache line exists in LLC, it directly returns the requested data. Otherwise, if the cache line is not in the LLC, the LLC has to load the data for the memory first. Another case is when the directory is in E state and after a downgrade, the directory detects the owner has already invalidated itself in silence, so the BYOC cache line still ends up in E state.

A BYOC cache line transitions from I state to M state when issuing an exclusive request (for stores). If the directory is in I state, it directly returns the requested data (may need to fetch it from the memory first if the data does not exist in the LLC). Otherwise the directory downgrades the owner (in E or M state) using `FwdEx` or invalidates the sharers (in S state) using `Inv`.

The BYOC cache evicts a cache line through the following steps. If the line is in E or S state, the eviction is silent without notifying the directory. Otherwise if the line is in M state, the dirty data needs to be written back using `WB` request. `WB` requests are never acknowledged. In order to avoid race conditions, the `WB` request is sent through NoC3 instead of NoC1. Since all NoCs maintain point-to-point ordering, this guarantees that if another BYOC cache requests the same cache line at the same time and the request arrives at the directory before the `WB` request, the downgrade will not bypass the write-back request. Another race condition occurs when the same BYOC cache receives a load/store and sends out a `ReqRd` or `ReqEx` to NoC1 after sending `WB` to NoC3. To avoid the later `ReqRd` or `ReqEx` request arriving at the directory first, a `WBGuard` message is inserted into NoC1 upon sending out a `WB` request to NoC3.

When a cache line in the LLC is evicted, since the LLC is inclusive and the directory information is embedded with each LLC cache line, all copies in private caches and the directory information are invalidated as well. The directory sends out `Inv` if the cache line is in S state or downgrade (`FwdEx`) if the cache line is in E or M state to invalidate sharers or the owner and transition to I state in the end. The evicted cache line is subsequently written back to memory if it is dirty.

In OpenPiton, atomic memory operations (AMOs) are handled in the LLC, rather than in the private BYOC caches. A BYOC cache sends an AMO request with data using `ReqAMO` on NoC1. If the cache line in the LLC is owned by other BYOC caches (in E or M state) or shared (in S) state, the LLC needs to invalidate or downgrade the cache line from other BYOC caches first before performing the AMO. The result of the AMO is returned to the BYOC cache requestor using `AckDt` on NoC2.

4.1.4 Last-Level Cache

The last-level cache (LLC) is a distributed (banked) write-back cache shared by all cores. The default cache size is 64KB per core. It is 4-way set associative and the block size is 64 bytes. A directory array is also integrated with the L2 cache with 64 bits per entry. Therefore, the directory is able to keep track of up to 64 sharers. The L2 cache is inclusive of private BYOC caches so every private cache line has a copy in the L2 cache. Each distributed L2 cache bank receives input requests from NoC1 and NoC3 and sends output responses to NoC2.

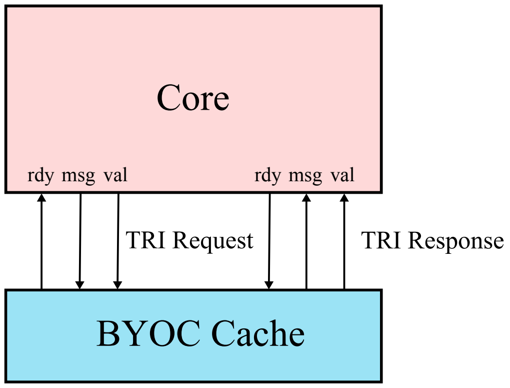


Figure 4.3: TRI between BYOC Cache and Core – val = valid signal; rdy = ready signal; msg = message.

4.1.5 BYOC Cache

The BYOC cache is a private write-back cache. It has a default size of 8KB and it is 4-way set associative. However, the main purpose of the BYOC cache is to provide an easy-to-plug-in interface for cores with a variety of ISAs and memory interfaces. As discussed earlier, the BYOC caches connect to the rest of the memory hierarchy through three NoCs. Each NoC connects to the BYOC cache with a credit-based interface. The BYOC cache uses a simple transaction-response interface (TRI) to connect to the core. Therefore, the BYOC cache also serves as a TRI-to-NoC adapter (transducer).

Figure 4.3 shows how a core is connected to a BYOC cache using TRI. TRI consists of two parts: one from the core to the BYOC cache, known as TRI request; and one from the BYOC cache to the core, known as TRI response. Each direction uses the valid-then-ready handshake micro-protocol [Tay18]. A TRI request contains a message type, a cache-line-aligned physical address, a request size, and data (if any). A TRI response contains a message type and data.

A core can be directly connected to a BYOC cache using TRI, or it can have L1 caches. If a core contains L1 caches, they must be write-through, since the directory in the LLC needs to track all modified cache lines. Each BYOC cache line tracks in the L1 cache way if this line is also cached in an L1. This allows a BYOC cache to invalidate an L1 cache line if the BYOC cache line is evicted or invalidated by LLC.

4.2 Heterogeneous Cache Coherence in OpenPiton

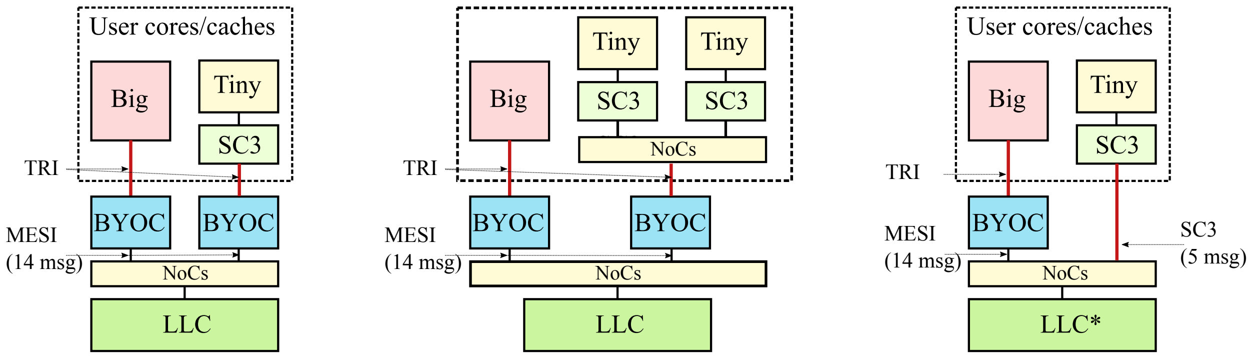
In this section, I provide a detailed description of how to implement heterogeneous multi/-manycore systems based on the OpenPiton platform. Chapter 3 demonstrated that heterogeneous cache coherence is a promising approach to scale-out multi/manycore systems by adding many software-centric cache-coherent tiny cores. However, the existing OpenPiton framework assumes MESI hardware-based cache coherence for all cores. This section discusses how to extend the OpenPiton's memory hierarchy to support software-centric cache coherent protocols. The extended OpenPiton platform with heterogenous cache coherence support is called OpenPiton-HCC. OpenPiton-HCC can serve as a template to explore future heterogeneous multi/manycore systems.

4.2.1 The Challenges of Using BYOC Approach

While the BYOC cache makes it straightforward to plug in a new core, efficiently adding tiny cores with software-centric coherence (as described in Chapter 3) is challenging with this approach. To add a tiny core with software-coherent L1 caches, the most obvious choice is to connect each core-L1 composition to a BYOC cache via TRI, and to use the BYOC cache as a private L2. This approach is illustrated in Figure 4.4 (a). However, this method carries all the communication and storage overhead of MESI, entirely defeating the purpose of using software-centric cache coherence protocols.

Another intuitive design is to connect multiple tiny cores (each with its own L1 cache) to a single BYOC cache, i.e., to use the BYOC cache as a shared L2 cache. Figure 4.4 (b) illustrates an example of a multi/manycore using this approach. This configuration also suffers several overheads. First, the BYOC cache is designed as a private cache and supports limited number of requests at the same time due to the size of miss status holding registers (MSHR). When sharing a BYOC with multiple cores, the bandwidth to BYOC quickly becomes a bottleneck when the core count increases. Second, the shared BYOC needs to be sufficiently large to reduce MESI overhead. The area of the BYOC cache can easily out-weight the area savings of the tiny cores.

These challenges in using the BYOC approach motivated me to implement OpenPiton-HCC, an extension to OpenPiton with heterogeneous cache coherence support. In OpenPiton-HCC, a private cache with software-centric cache coherence (SC3) can be directly integrated with the rest of the OpenPiton components, without using the TRI or the MESI-based BYOC cache. An example of



(a) Connect both types of cores using BYOC caches provided by the framework

(b) Connect multiple SC3 caches to a BYOC cache using intra-tile NoCs

(c) Connect SC3 caches directly to NoCs with HCC support

Figure 4.4: Three Options to Build HCC systems – The red lines indicate the interface points between the user cores/caches and the framework. Big = multicore processor (big core) with hardware-based cache coherence; Tiny = tiny core; LLC = last-level cache; LLC* = modified last-level cache with support of HCC; NoC = on-chip network; msg = types of messages (requests and responses) at the interface.

an SC3 cache connecting to the NoCs in OpenPiton-HCC is shown in Figure 4.4 (c). An important insight is that compared to MESI hardware-based cache coherence protocol, an SC3 protocol is significantly simpler. For example, the BYOC cache uses 14 message types (request and response types combined) to communicate with the rest of the memory system (see Table 4.1). In contrast, an SC3 cache that supports the GPU-WB protocol only uses five message types (`ReqAMO`, `Ack`, `AckDt`, and the two in Table 4.2). Therefore, connecting SC3 caches directly to the NoCs does not significantly increase hardware complexity compared to the BYOC+TRI approach.

4.2.2 Adding Support of GPU Coherence Protocols

Compared to the baseline OpenPiton, OpenPiton-HCC's cache coherence protocol added two new coherence operations so that private caches with software-centric cache coherence protocols (e.g., GPU coherence as described in Section 3.2) can directly communicate with the LLC. Since OpenPiton-HCC supports a superset of coherence operations, BYOC caches with MESI protocol function in the same way as in the baseline OpenPiton. More importantly, OpenPiton-HCC allows BYOC caches with MESI, private caches with software-centric cache coherence protocol, and the shared LLC to seamlessly work together. BYOC caches and private caches with GPU coherence protocols can share data at cache line granularity. OpenPiton-HCC implements an efficient heterogeneous cache coherence similar to the cycle-level model used in Chapter 3 and prior work

	NoC Name	Contains Data	Operations
`ReqV`	NoC1	No	Request cache line data only
`WB-GPU`	NoC1	Yes	GPU cache evicts/writes back dirty data

Table 4.2: Additional Coherence Operations of OpenPiton-HCC – These are additional coherence operations supported by OpenPiton-HCC. OpenPiton-HCC also supports all the operations in Table 4.1.

Spandex [ASA18]. Although OpenPiton-HCC currently only supports GPU coherence protocols, other software-centric protocols (e.g., DeNovo [CKS+11, SKA13, SA15]) can be added in similar ways.

As I discussed in Section 3.2, the key difference between GPU coherence protocols and MESI is that in GPU coherence protocols, clean cache lines are reader-invalidated, and the private cache does not need to obtain ownership before modifying a cache line. Therefore, if a private cache with GPU coherence has a load miss, *it only needs to fetch the cache line data from the LLC, but does not to be registered as a sharer*. When a private cache with GPU coherence writes back a cache line, *the cache line is not owned by the private cache*. To accommodate these differences, we added two coherence request types, listed in Table 4.2.

The `ReqV` request is issued by a private cache with the GPU coherence protocol to refill a cache line when the cache has a load or store miss. `ReqV` is similar to `ReqRd` but *it does not register the requestor as a sharer*. However, if the cache line requested is owned by another MESI private cache, i.e., the cache line in LLC is in E or M state, the LLC needs to send `FwdEx` to the owner. The owner is downgraded and sends the cache line data to the LLC and the cache line in LLC transitions to S state. Then the LLC can supply the requestor of `ReqV` with the up-to-date cache line data. Although in this scenario the cache line in LLC ends up in S state, the requestor is not registered as a sharer.

The `WB-GPU` request is used to write back dirty cache lines from private caches in GPU coherence protocols. Because GPU coherence protocols track dirty data at word granularity, each `WB-GPU` request contains a *word mask*, a bit-vector indicating which words in the cache line are dirty. The LLC will only update words whose corresponding bit in the word mask is set. A key difference between `WB-GPU` and `WB` is that `WB-GPU` is issued by a non-owner private cache, but `WB` is always issued by an owner whose original cache line state is M state (eviction of S and E is silent). The target cache line state of a `WB-GPU` request in the LLC can be any of the four states (M, E, S, and I). If the LLC cache line state is I, the LLC may need to refill the cache line from the backing store, and update the words of the cache line based on the word mask. If the LLC cache line state is S, the LLC sends `Inv` to all sharers of this cache line to invalidate, and updates the cache line based on the word mask. If the cache line state is E or M, the LLC sends downgrade request `FwdRd` to the owner of the cache line. After the previously up-to-date cache line data is transferred from the owner private cache to the LLC, `WB-GPU` can be performed in the LLC. GPU coherence protocols only require two NoCs instead of three, so the `WB-GPU` request travels on NoC1 instead of NoC3 in OpenPiton-HCC.

To reduce the storage overhead, the GPU-WB protocol described above only provides coherence at word granularity using the per-line word masks. In the infrequent cases of data sharing at byte granularity (i.e., multiple threads have conflicting accesses to different bytes in the same word), the GPU-WB protocol requires additional compiler and/or runtime support to ensure coherence [CKS+11]. This thesis does not consider byte-level data sharing.

4.2.3 Implementing an L1 Cache with GPU-WB Protocol

Compared to the MESI protocol, the GPU-WB protocol (see Section 3.2) can be implemented in private caches much more easily. In fact, the difference between a cache supporting the GPU-WB coherence protocol and a single-core cache is minimal. To demonstrate this, I collaborated with my colleagues to implement a GPU-WB private cache in RTL. Figure 4.5 presents the pipeline diagram of this cache. Both the tag arrays and data arrays are implemented using SRAM and have one cycle read latency.

The cache has three pipeline stages. In stage 0, requests from the core or refill response from the LLC are received, and the tag arrays are accessed or updated. In stage 1, the tag array content is read and is used to compute the indices for the data arrays. In stage 2, responses or refill request are generated and sent out. The cache has a blocking design, and uses a single-entry miss status holding register (MSHR) to keep track of the not-yet-refilled miss request. In addition to the normal per-line valid bit used in single-core caches, this GPU-WB cache has a per-line word mask stored in the tag array.

The cache supports two special request types from the core, in addition to load, store, and AMO. An `invalidate` request from the core invalidates all clean cache lines. A `flush` request writes back all dirty cache lines to the LLC. The `invalidate` request lets the cache iterate through all cache lines and clear the valid bit in the tag array at a rate of one cache line per cycle. When

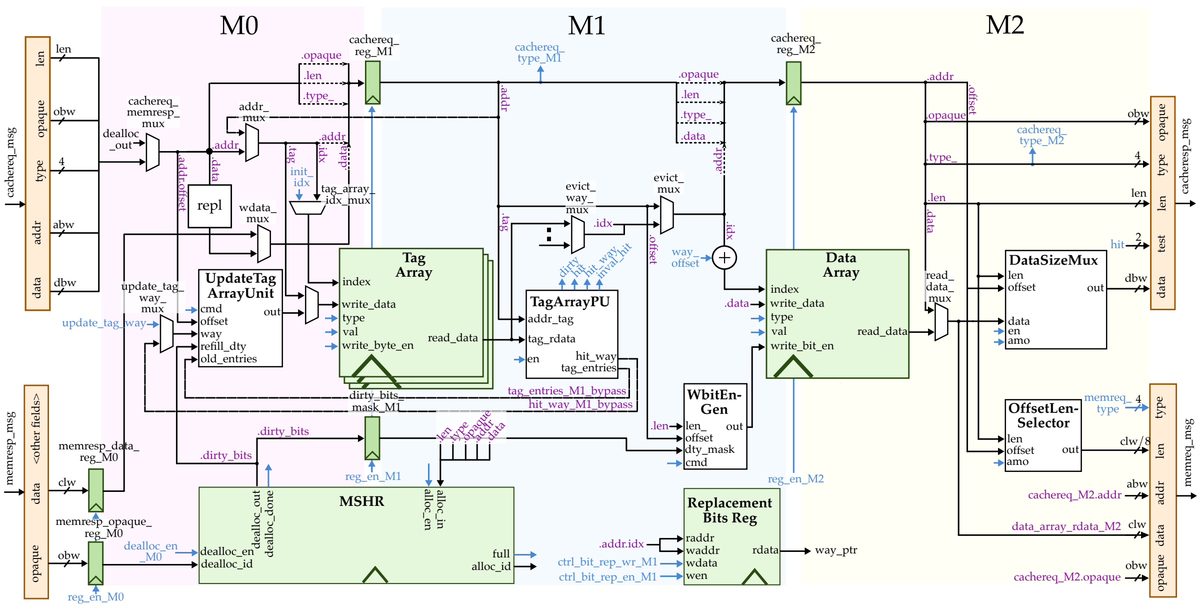
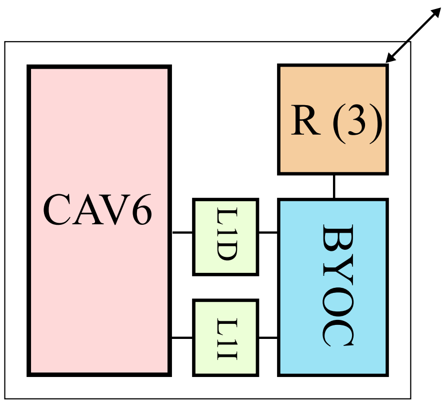


Figure 4.5: Data Path Diagram of a GPU-WB Private Cache – M0, M1, and M2 are three pipeline stages. Control signals are in blue. Status signals are in purple. The figure is authored by Xiaoyu Yan and Eric Tang.

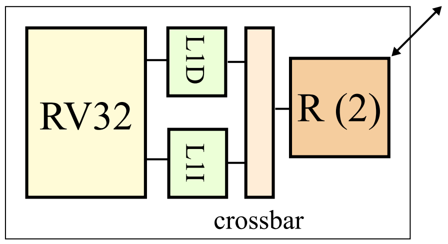
receiving a `flush` request, the cache iterates through all cache lines. If a cache line's word mask is not all-zero, it sends a `WB-GPU` request to the LLC along with the cache line's word mask to write back the dirty data. For cache line refill, the cache uses `ReqV` request to the LLC. The cache requires two NoCs: NoC1 is used for sending requests (`WB-GPU`, `ReqV`, and `ReqAMO`) to the LLC, and NoC2 is used for receiving responses from the LLC. The cache's interfaces originally used a valid-then-ready micro-protocol [Tay18], but they can be easily adapted to the credit-based interface used by the OpenPiton NoCs.

4.3 Tile-Level Evaluation

OpenPiton-HCC enables heterogeneous multi/manycore tiles to be seamlessly integrated using fine-grained heterogenous cache coherence. In this section, I present a tile-level study of heterogeneous multi/manycore systems. Since the OpenPiton framework lacks an open-source out-of-order (big) core, I focus on a type of heterogeneous multi/manycore systems called little.TINY (see Figure 4.6). A little tile consists of an Linux-capable 64-bit RISC-V RV64GC [ris17] in-order



(a) Little Tile



(b) Tiny Tile

Figure 4.6: Little and Tiny Tile in OpenPiton-HCC – The diagrams show the composition of little and tiny tiles in a little.TINY heterogeneous multi/manycore system. The figure is not to scale.

processor called CVA6 (formerly known as Ariane) [ZB19], a BYOC private cache, and three NoC routers. The CVA6 core contains a 16kB read-only L1 instruction (L1I) cache and an 8kB write-through L1 data (L1D) cache. The two L1 caches are connected to the BYOC cache using TRI. The little tiles have hardware-based cache coherence and are intended for building the multicore portion of the system. A tiny tile consists of a tiny RISC-V 32-bit RV32IMAF processor, two 4kB GPU-WB caches described in Section 4.2.3, and two NoC routers. The tiny tiles are used to build the manycore portion of the system. Unlike the original OpenPiton system, where each tile contains a shared LLC bank, we removed LLC banks from both little and tiny tiles. LLC banks are placed separately in their own tiles. Table 4.3 summarizes the parameters of little and tiny tiles. Overall, the tiny tile has a smaller cache size, has a smaller and less capable core, and uses one less NoC than the little core.

4.3.1 GPU-WB vs. MESI Area Evaluation

This subsection uses VLSI area results to answer the question: *how much smaller is a cache, if it uses a simpler software-centric coherence protocol rather than MESI hardware-based cache coherence?* We configured an 8kB GPU-WB cache and compared it with an 8kB BYOC cache. Both caches have a 16B cache line size. Both have the same associativity and hit latency. They are synthesized using a commercial ASIC tool flow and a 14nm technology. They use SRAM cells generated from the same SRAM compiler.

	Little Tile	Tiny Tile
ISA	RV64GC	RV32IMAF
OS-Capable	Yes	No
Pipeline Stages	6	5
TLB	Yes	No
L1I	16kB, read-only	4kB, read-only
L1D	8kB, write-through	4kB, GPU-WB
L1 cache line	16B	16B
L2	8kB, MESI (BYOC cache)	No
L2 cache line	16B	N/A
NoCs	NoC1, NoC2, NoC3	NoC1, NoC2

Table 4.3: Parameters of Little and Tiny Tile

	BYOC (μm^2)	GPU-WB (μm^2)	Area Saved (%)
Data Array	9551	7661	19.7
Tag Array	3871	2826	27.0
NoC Buffers	1422	390	72.6
Coherence Logic	1337	0	100
MSHR	530	81	84.7
Way Map	1928	0	100
Other Logic	3476	1512	56.5
Total	21585	12471	42.9

Table 4.4: Post-Synthesis Area of 8kB Caches – An area comparison between an 8kB BYOC MESI and an 8kB GPU-WB cache. The way map in BYOC is used to invalidate the L1 when the BYOC is used as a private L2.

Table 4.4 shows the area breakdown from post-synthesis results under a 1 GHz target frequency. Both caches met the timing constraint. The VLSI area results show that a cache with GPU-WB protocol is significantly smaller than a BYOC cache with the same capacity, associativity, and hit latency. The most notable area saving of the GPU-WB cache is from the coherence logic, the NoC buffers, and the tag arrays. Compared to MESI, the GPU-WB protocol requires less states need to be stored in the tag arrays. The coherence logic is also drastically simpler than MESI. GPU-WB only needs to use two NoCs instead of three NoCs, therefore it also saves area in NoC buffers.

Another benefit of software-centric cache coherence protocols is the design complexity. To quantify this, I measured the total lines of code of the two implementations. The 8kB BYOC cache has a total of 12,207 lines of Verilog code (not counting blank lines and comments). The GPU-WB cache is implemented using a Python-based RTL design language, PyMTL [LZB14, JPOB20]. It

	Little Tile (μm^2)	Tiny Tile (μm^2)
Core	42.86k	14.79k
L1I	33.74k	7.88k
L1D	22.83k	7.92k
Core + L1	99.43k	30.60k
L2	21.24k	0
NoC routers	4.42k	2.96k
Other Logic	1.30k	1.98k
Total	126.38k	35.53k

Table 4.5: Post-Synthesis Area Breakdown of Little Tile and Tiny Tile

has 2,130 lines of Python code. The PyMTL design generates 3,492 lines of Verilog code, which is less than one third of the BYOC cache's lines of code.

Overall, this preliminary area breakdown shows the complexity effectiveness of software-centric cache coherence protocols and further motivates the heterogeneous multi/manycore systems.

4.3.2 Little vs. Tiny Tile Area Evaluation

In this subsection, I discuss VLSI area results of the little and the tiny tile shown in Figure 4.6. To obtain the area results, an instance of the little tile and an instance of tiny tile were synthesized using a commercial logic synthesis tool, under the timing constraint of 1 GHz. We used a 14nm standard cell library and a commercial SRAM compiler. An ASIC layout of each tile was then generated from the synthesized design using an industry-standard place-and-route tool.

Table 4.5 presents the post-synthesis area breakdown of both tiles and Table 4.6 presents the post-place-and-route area breakdown. The tiny tile has significantly less area compared to the little tile. The 32-bit tiny bare-metal core is one third of the area of the CVA6 64-bit Linux-capable core. The GPU-WB L1 caches in the tiny tile are less than one third of the write-through L1 caches in the little tile, mainly due to the difference in capacity. The little tile contains a BYOC cache as its L2, whereas the tiny tile does not have an L2. All things combined, a tiny tile's area is 28% of a little tile's area (35.53k vs. 126.38k in Table 4.5). The conclusion of these results is that *the area of one little tile can fit three tiny tiles*. Adding *many* tiles is an area-efficient way to increase the core count in heterogeneous multi/manycore systems.

	Little Tile (µm^2)	Tiny Tile (µm^2)
Core	45.89k	15.11k
L1I	34.03k	7.99k
L1D	23.31k	8.07k
Core + L1	103.22k	31.17k
L2	21.32k	0
NoC routers	4.04k	2.63k
Other Logic	1.38k	1.90k
Total	129.96k	35.71k

Table 4.6: Post-Place-and-Route Area Breakdown of Little Tile and Tiny Tile

4.4 Chip-Level Evaluation

In this section, I present the chip layouts of several heterogeneous multi/manycore systems implemented using the OpenPiton-HCC framework. These results demonstrate that the heterogeneous multi/manycore approach can realistically increase hardware parallelism for a given area or save area for given a core count. These heterogeneous multi/manycore chips can leverage the software and hardware techniques described in previous chapters to provide programmers a unified task-based parallel programming framework and can efficiently and cooperatively execute fine-grained dynamic task parallelism in both the multicore portion (a few little tiles) and the manycore portion (many tiny tiles).

We used a hierarchical ASIC design methodology to implement layouts for heterogeneous multi/manycore systems. Each tile is first synthesized, then the place-and-route tool uses the synthesized design to generate a *hard macro*. Multiple hard macros of both types are composed (placed) in the place-and-route tool to build the chip-level layout. Then the tool performs chip-level routing to obtain the final layout.

Figure 4.7 shows the layout of a little-tile hard macro. It measures 350 µm in width and 500 µm in height. The macro includes a CVA6 core, two write-through L1 caches, a BYOC cache (as an L2 cache) with MESI hardware-based cache coherence, and three NoC routers. Multiple little tile macros can be used to build a traditional multicore with full-system hardware-based cache coherence, by simply arranging them in a 2D mesh topology and connecting them with wires. When building multi/manycore systems, the little tile macros are used to build the hardware-coherent multicore portion, and compose with many more tiny tiles.

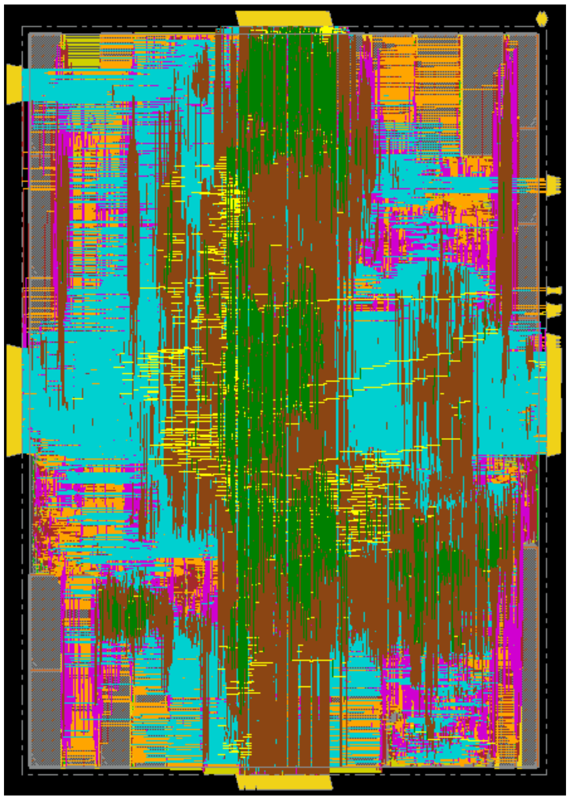


Figure 4.7: Hard Macro of Little Tile – This is a plot of the little tile hard macro. The macro measures 350 µm in width and 500 µm in height.

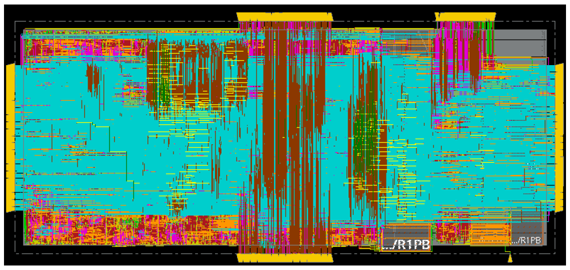


Figure 4.8: Hard Macro of Tiny Tile – This is a plot of the tiny tile hard macro. The macro measures 350 µm in width and 150 µm in height.

Figure 4.8 shows the layout of a tiny-tile hard macro. It measures 350 µm in width and 150 µm in height. The tiny tile is significantly smaller than the little tile. In a multi/manycore system, tiny tile macros are used to scale-out the number of cores to increase hardware parallelism in a complexity-effective way.

Figure 4.9 shows the layout for a tile of 16KB shared LLC bank. It measures 350 µm in width, and 500 µm in height. The LLC banks can be placed anywhere in the chip, but they are usually placed at the edge of the chip.

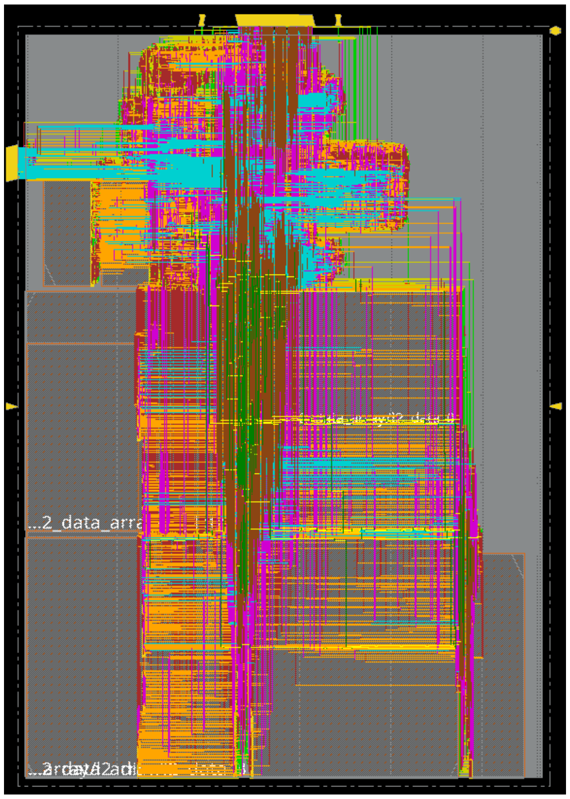
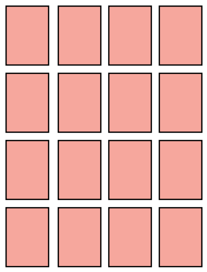


Figure 4.9: Hard Macro of LLC Bank Tile – This is a plot of the 16KB LLC Bank hard macro. The macro measures 350 µm in width and 500 µm in height.

Figure 4.10(a) is a to-scale layout for a 16-core multicore systems. LLC banks are not showed in this picture. It consists of 16 little tiles in a 4×4 2D-mesh. The pitch between tiles is 100 µm in both X and Y directions. The layout was successfully placed in the place-and-route tool and can have no timing or routing issue. The dimension of this chip layout is 1595 µm in width and 2195 µm in height. For the same area, core count can be scaled up by replacing some of the little cores with tiny cores to form a heterogeneous multi/manycore system. Figure 4.10(b) shows a 4×11 multi/manycore with four little cores and 40 tiny cores. The layout is of the same height and width as Figure 4.10(a).

Figure 4.11(a) shows a larger-scale example of 8×8 little tile multicore array. The whole chip measures 3255 µm in width and 4455 µm in height. By replacing all little tiles except for the top and bottom row in the 8×8 mesh with tiny tiles, the heterogeneous multi/manycore approach enables the core count to grow to 176 (16 little tiles and 160 tiny tiles), as shown in Figure 4.11(b).


(a) 16 Little Tiles

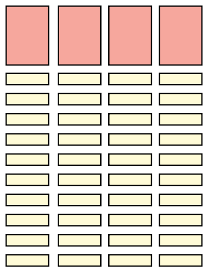
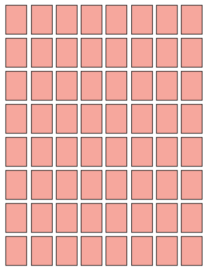

(b) 4 Little Tiles and 40 Tiny Tiles

Figure 4.10: To-Scale Layout of a 1595 µm **in Width by** 2195 µm **in Height Chip –** This to-scale layout shows the same area can be used to implement either: (a) a 16-core multicore with all little tiles; or (b) a 44-core heterogeneous multi/manycore with 4 little cores and 40 tiny cores. LLC banks are not shown.


(a) 64 Little Tiles

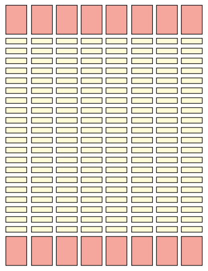

(b) 16 Little Tiles and 160 Tiny Tiles

Figure 4.11: To-Scale Layout of a 3255 µm **in Width by** 4455 µm **in Height Chip –** This to-scale layout shows the same area can be used to implement either: (a) a 64-core multicore with all little tiles; or (b) a 176-core heterogeneous multi/manycore with 16 little cores and 160 tiny cores. LLC banks are not shown.

4.5 Area Model of Heterogeneous Multi/Manycore Systems

In addition to the concrete instances of multi/manycore chips presented in previous sections, this section discusses an area model based on the post-place-and-route VLSI results. The area model projects the total chip area given a target core count and demonstrates the potential of the heterogeneous multi/manycore approach in future systems.

The model assumes a baseline of eight little tiles, each with a CAV6 core, a 16kB L1I cache, an 8kB L1D cache, an 8kB BYOC L2 cache, and three NoC routers (parameters are the same as in Table 4.3). The area of the little tile in this model is from Table 4.6. The total LLC capacity is fixed at 16MB. The model also assumes a full map implementation of the LLC directory, i.e., the number of bits used to track sharers in each directory entry equals to the total number of sharers. The area of the directory in the LLC is extrapolated from the 64B-per-entry directory's SRAM area assuming linear scaling. To increase core count, the model considers three approaches:

Multicore approach – The multicore approach is to add identical little cores to the system. Because each little tile contains an 8kB MESI-based L2 cache (BYOC cache), the directory entries (i.e., the sharer lists) in the LLC grow with the total core count.

Multi/manycore approach with heterogeneous cache coherence (multi/many HCC) – This approach increases the total core count by adding tiny tiles. A tiny tile contains an RV32 core, a pair of 4kB GPU-WB L1I and L1D caches, and two NoC routers, as described in Table 4.3. The area of the tiny tile is also from post-place-and-route VLSI results presented in Table 4.6. Since GPU-WB caches never becomes sharer, the size of sharer list is fixed to be 8 bits per entry, which equals to the number of the little cores.

Multi/manycore approach with full-system hardware-based cache coherence (multi/many HW) – This approach also increases the total core count by adding tiny tiles. However, the tiny tile's L1I and L1D are replaced by a single BYOC MESI cache. In this configuration full-system hardware-based cache coherence (i.e., MESI) is provided to both the little cores and the tiny cores. The directory entries (i.e., the sharer lists) in the LLC grow with the total core count.

Figure 4.12 shows the whole-chip area vs. the total number of cores using this area model. As shown in this figure, due to the larger size of the little tile compared to the tiny tile, as well as the directory overhead, the multicore approach consumes the most amount of area for a given core count. The area of the multi/many HW approach also scales significantly faster than multi/many HCC, because of the directory storage overhead. *For an area of 67* mm^2*, the multi/many HCC can fit 1024 cores (8 little, 1016 tiny); the multi/many HW can fit about 220 cores (8 little, 212 tiny); and the multicore approach can fit 180 little cores*. Looking at it in the other way, *to scale to a 256-core systems, the multi/many HCC uses 39* mm^2*; the multi/many HW approach uses 63* mm^2*; and the multicore approach uses 84* mm^2. This comparison shows significant area benefits for the multi/manycore approach with heterogeneous cache coherence.

Realistic MESI implementations often use techniques like limited map or coarse map [SHW11] to trade performance for area. I modified the model to limit each sharer list entry to be at most 64 bits. The revised model results are shown in Figure 4.13. Limiting the sharer list size significantly reduces the area required for both multi/many HW and multicore approaches. The figure shows that *for an area of 67* mm^2*, the multi/many HCC can fit 1024 cores (8 little, 1016 tiny); the multi/many HW can fit about 750 cores (8 little, 742 tiny); and the multicore approach can fit 220 tiny cores. to scale to a 256-core systems, the multi/many HCC uses 39* mm^2*; the multi/many*

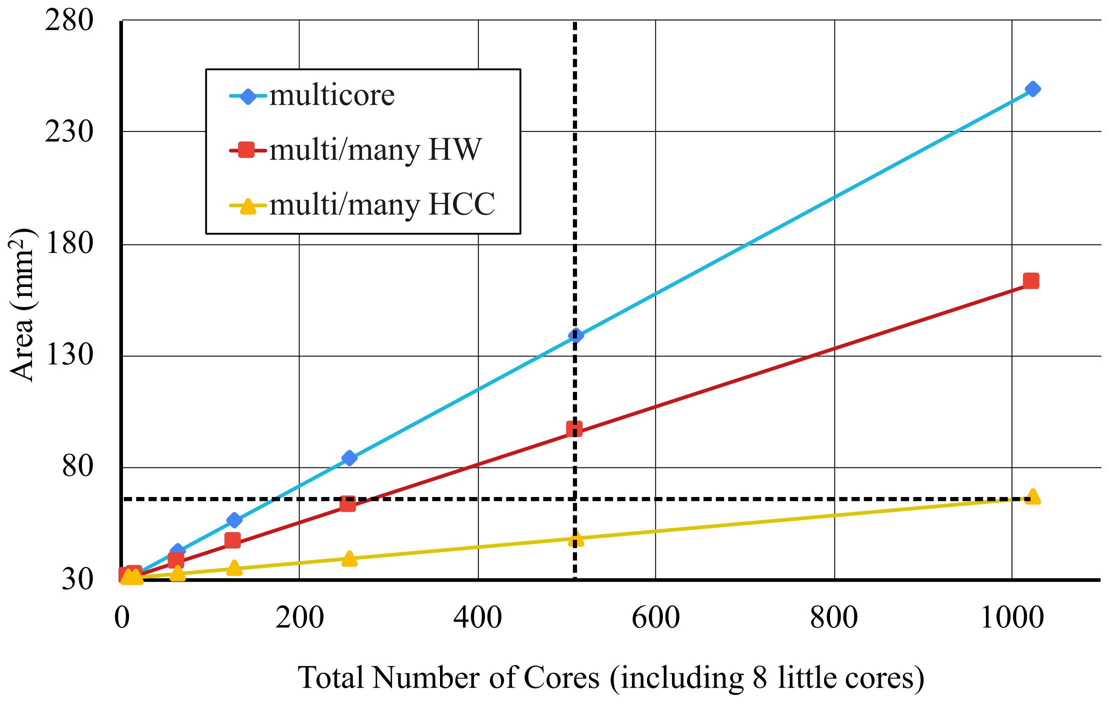


Figure 4.12: Area vs. Number of Cores using the Multi/Manycore Area Model – The model assumes 64B cache line, 16MB total LLC capacity. The number of cores includes eight little cores. multicore = scaling up using the multicore approach by adding little tiles with hardware-based cache coherence; multi/many HW = scaling up the core count by adding tiny tiles but each tiny tile contains a BYOC cache to achieve full-system hardware-based cache coherence; multi/many HCC = scaling up the core count by adding tiny tiles with GPU-WB caches.

HW approach uses 46 mm^2*; and the multicore approach uses 67* mm^2. Even after considering a limited directory, the multi/manycore approach with heterogeneous cache coherence still holds area benefits over multi/manycore with full-system hardware-based cache coherence (1024 cores vs. 750 cores) and multicore (1024 cores vs. 220 cores).

4.6 Conclusion

This chapter presents OpenPiton-HCC, an extension to OpenPiton which supports heterogenous cache coherence. OpenPiton-HCC enables VLSI-based area and timing evaluations of heterogenous multi/manycore systems. VLSI results show the heterogenous multi/manycore approach has advantage in area for scaling up core counts and does not sacrifice timing. Based on the VLSI results, I build an area model to project area implications of the multi/manycore approach for future large-scale systems. The model shows, for the same amount of area, the heterogenous multi/many-

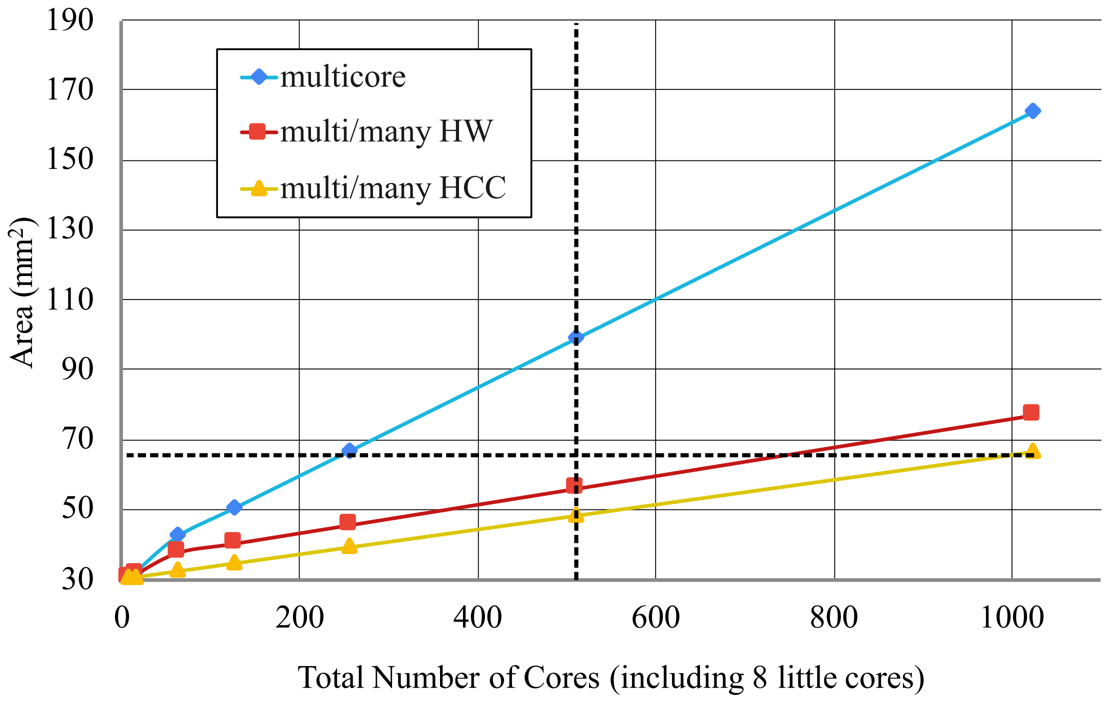


Figure 4.13: Area vs. Number of Cores using the Multi/Manycore Area Model with Limited Directory– The model assumes 64B cache line, 16MB total LLC capacity. The number of cores includes eight little cores. Each directory entry is limited to at most 64 bits. multicore = scaling up using the multicore approach by adding little tiles with hardware-based cache coherence; multi/many HW = scaling up the core count by adding tiny tiles but each tiny tile contains a BYOC cache to achieve full-system hardware-based cache coherence; multi/many HCC = scaling up the core count by adding tiny tiles with GPU-WB caches.

core approach can increase the core count by more than 2× compared with a multi/manycore systems with full-system hardware-based cache coherence, and increase the core count by more than 4× compared to the traditional multicore approach.

CHAPTER 5
CONCLUSION

Technology constraints continue to drive computer architects to increase parallelism. Manycore processors have been increasingly popular in modern computing platforms. However, a key research challenge remains: how to efficiently utilize manycore processors to cooperatively execute fine-grained dynamic task-parallel applications across multicore and manycore processors. This thesis proposed software and hardware approaches to enable efficient co-operative execution of fine-grained dynamic task parallelism on heterogeneous multi/manycore systems. On the software side, this thesis proposed a work-stealing runtime designed for heterogeneous cache coherence to provide a unified programming model for multi/manycore systems. On the hardware side, this thesis explored using the multi/manycore approach with heterogeneous cache coherence to increase parallelism efficiently in terms of performance and area. This thesis also proposed direct task stealing, a lightweight software/hardware technique to increase the performance and energy efficiency of work-stealing runtimes on heterogeneous multi/manycore systems. In addition, this thesis presented a realistic VLSI evaluation to demonstrate the feasibility of the proposed techniques. The rest of this chapter summarizes primary contributions of this thesis and discusses future research directions.

5.1 Thesis Summary and Contributions

This thesis began by discussing the hardware and the software trends in manycore processors. I presented a survey of manycore processors with MIMD and SIMD architectures that came into existence over the past two decades. The survey showed a fundamental tradeoff between hardware scalability and software programmability. MIMD and SIMD manycore processors with massive core counts often lack hardware-based cache coherence, preventing programmers from using traditional multicore shared-memory programming models. I discussed the hardware trend of tighter integration between manycore processors and multicore processors in recent heterogeneous computing platforms. This trend opens up opportunities for cooperative execution between multicore and manycore processors. I then discussed current heterogeneous programming models and showed the challenges of using them to achieve efficient cooperative execution, especially when considering fine-grained dynamic parallelism in emerging applications. This chapter motivated a holistic software and hardware solution to address the challenge of cooperative execution of fine-grained task parallelism on heterogeneous multi/manycore systems.

The thesis then discussed a new lightweight and modular C++ framework for dynamic task-parallel programming called `appl`. The purpose of developing `appl` was to address the challenges of modifying and extending commercial task-based parallel programming frameworks. While `appl` is a fraction of the size of commercial parallel programming frameworks, it supports a most-commonly used subset of APIs of popular task-based frameworks, which allows programmers to express a wide range of dynamic task-parallel patterns. An evaluation of `appl` on real machines showed that `appl` can achieve similar performance to the state-of-the-art task-based parallel programming frameworks. Two case studies were provided to demonstrate the extensibility of `appl` for architecture and parallel programming research. `appl` laid a software foundation for the explorations in the rest of the thesis and was used to provide a uniform programming model to heterogeneous multi/manycore systems proposed in this thesis.

The core of the thesis detailed how to map work-stealing runtimes, such as the one used by `appl`, to heterogeneous cache coherence. To the best of my knowledge, this was the first detailed description of how to extend work-stealing runtimes to enable cooperative execution of fine-grained dynamic task-parallel applications on heterogeneous multi/manycore systems. A cycle-level evaluation methodology showed the work-stealing runtime on heterogeneous cache coherence achieved similar performance to full-system hardware-based cache coherence while using simpler hardware. The thesis also provided insights on the source of overheads when using heterogeneous cache coherence for fine-grained task parallelism. These insights helped motivate using software and hardware techniques to improve performance and energy efficiency of work-stealing runtimes on systems with heterogeneous cache coherence.

The direct task stealing (DTS) technique proposed by this thesis was an answer to the overheads of heterogeneous cache coherence. It was based on the observation that steals are relatively rare, and cache coherence only needs to be maintained when steals happens. DTS uses a light-weight hardware feature called user-level interrupts to bypass the heterogeneous cache-coherent memory system for work stealing. DTS therefore can minimize the frequency of coherence operations when steals do not happen. The cycle-level evaluation results presented in this thesis showed that DTS effectively improved performance and energy efficiency of work-stealing runtimes on heterogeneous multi/manycore systems across multiple dynamic task-parallel applications. The overall results showed that executing dynamic task-parallel applications on a 64-core complexity-effective heterogeneous multi/manycore system (4 big, 60 tiny) with DTS achieved: 7× speedup over a single big core; 1.4× speedup over an area-equivalent eight big-core system with hardware-based cache coherence; and 21% better performance and similar energy efficiency compared to a 64-core system (4 big, 60 tiny) with full-system hardware-based cache coherence. The thesis also presented the results of selected applications running on a 256-core (4 big, 252 tiny) heterogeneous multi/manycore system with proportionally larger input datasets. These results showed DTS had similar efficacy on the larger 256-core system as the 64-core system.

In addition to the cycle-level evaluation, this thesis presented how to realistically implement the proposed heterogeneous multi/manycore systems in real hardware. The thesis discussed an extension of an existing hardware prototyping platform, OpenPiton, to support heterogeneous cache coherence. The thesis detailed an RTL implementation of a heterogeneous cache coherence protocol in OpenPiton-HCC, as well as an RTL implementation of a cache supporting the GPU-WB cache coherence protocol. This thesis presented VLSI area and timing results of a multicore tile with hardware-based cache coherence, and a manycore tiny tile with software-centric cache coherence. The thesis explored a few realistic whole-chip layouts generated by ASIC tools, which showed using the multi/manycore approach can realistically increase hardware parallelism with the same amount of area. The thesis also presented an analytical area model based on the post-place-and-route VLSI area results. The analytical area model showed the heterogeneous multi/manycore approach is significantly more area efficient than the full-system hardware cache coherence approach in future thousand-core-scale systems.

The primary contributions of this thesis are reiterated below:

- A new lightweight and modular C++ framework for dynamic task-parallel programming called `appl`, which serves as an extensible foundation for architecture and parallel programming research.

- To the best of my knowledge, the first detailed description of how to extend work-stealing runtimes to enable cooperative execution of fine-grained dynamic task-parallel applications on heterogeneous multi/manycore systems.

- A novel software and hardware technique which uses *direct task stealing* to improve the performance and energy efficiency of fine-grained dynamic task-parallel applications on heterogeneous multi/manycore systems.
- A realistic implementation of a heterogeneous multi/manycore system based on an open-source hardware prototyping platform which enables a detailed VLSI area and timing evaluation.

5.2 Future Work

The techniques presented in this thesis are a first step towards energy efficient, high-performance, and highly programmable future general-purpose systems. There are many opportunities to build on the ideas presented in this thesis. This section discusses some promising research directions for future work.

Exploring new task models and scheduling strategies – The thesis focused on task-parallel applications using the fork-join task model and work-stealing scheduling strategies. While this combination can efficiently express a wide range of parallel patterns, there are other task models and scheduling strategies that are better-suited to certain emerging applications. For example, the Galois [NLP13] graph processing framework has a *worklist* model where tasks are processed in an algorithm-specific order. OpenMP's dynamic and guided clause uses a *work sharing* scheduling strategy where tasks are stored in a centralized task queue. Dynamic pipeline parallelism [LLS+13] is prevalent in modern applications but it requires additional scheduling strategies. Each task model and scheduling strategy has its unique coherence requirement. Future work can explore techniques to improve performance and energy efficiency of heterogeneous multi/manycore systems when running task-parallel applications with other task models and scheduling strategies. While DTS may not be directly applied, the insights presented in this thesis can still be useful. The insight is to identify coherence requirement of the task model, and only maintain cache coherence when *true data sharing* occurs.

Application-specific cache coherence stratgies – DTS leverages the fork-join task model and work-stealing scheduling to minimize coherence operation while still observes the DAG consistency. However, in certain cases, this strategy can be over-conservative. For example, the fork-join model specifies the control dependency between tasks, but tasks that are control-dependent may not be data-dependent. In addition, certain data, such as read-only data, does not need to be coherent at all. Application-specific knowledge about true data dependency can further guide the runtime as well as the hardware to avoid coherence operations if there is no true data dependency. Some programming models, such as Deterministic Parallel Java [JVA+09], allow programmers to annotate a data type with a region, and only provide coherence among data that is within the same region. While this approach can be effective in some cases, it reduces programmability by requiring programmer to identify and annotate regions. An interesting research question is how to use hardware and software techniques to infer true data dependency without programmers' intervention. The software runtime systems and the underlying hardware may both be involved to achieve this goal.

Work-stealing for specialized hardware (accelerators) – In addition to parallelism, specialization is another important technique embraced by computer architects to continue improving performance and energy efficiency given the technology constraints. It is commonplace that a modern computing platform includes several or tens of different types of application-specific hardware accelerators. Enabling fine-grained cooperative execution between general-purpose cores and accelerators is an open research challenge because one needs to design an abstraction for accelerators such that a task can be either executed by a big core, a tiny core, or an accelerator. Future work may look at how to extend the programming model to include accelerators.

Combining intra- and inter-core parallelism – This thesis proposed heterogeneous multi/-manycore systems that mainly exploit thread parallelism between cores. However, modern processor cores can leverage intra-core parallelism, such as subword-SIMD (e.g., Intel AVX instruction set extension). Ideally, we want to combine inter- and intra-core parallelism and achieve multiplicative speedup. Recent work on LTA [KJT+17] showed promising results exploiting both inter- and intra-core parallelism for loop tasks. An interesting direction is increasing parallelism within a core, instead of inter-core parallelism. This can be beneficial to the performance and energy efficiency of heterogeneous cache coherence, since tasks executed on the same core do not trigger coherence operations. One idea is to combine multiple tiny cores into a core with multiple "lanes". Each lane executes a worker thread and can steal tasks from other lanes within the core or from another core. This approach obviously requires a more sophisticated scheduling strategy as well.

VLSI energy evaluation – OpenPiton-HCC presented in this thesis provided a way to build hardware prototypes of heterogeneous multi/manycore systems. This thesis offered area and timing analysis based on VLSI results. Future work may further evaluate the performance of heterogeneous multi/manycore systems at RTL. The RTL implementations also offers opportunities to quantitatively study energy implications of the techniques proposed by this thesis. While RTL simulation is time-consuming, FPGA or ASIC prototypes of OpenPiton-HCC can be used to evaluate real-world energy and performance on larger input data sets.

BIBLIOGRAPHY

[ACD$^+$09] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 20(3):404–418, Mar 2009.

[ACR13] U. A. Acar, A. Chargeéraud, and M. Rainey. Scheduling Parallel Programs by Work Stealing with Private Deques. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Feb 2013.

[AKPJ09] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2009.

[amd20] AMD CDNA Architecture. AMD White Paper, 2020. `https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf`.

[ASA18] J. Alsop, M. Sinclair, and S. V. Adve. Spandex: A Flexible Interface for Efficient Heterogeneous Coherence. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2018.

[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 Computer Architecture News (CAN)*, 39(2):1–7, Aug 2011.

[BCC$^+$17] D. Bradford, S. Chinthamani, J. Corbal, A. Hassan, K. Janik, and N. Ali. Knights Mill: New Intel Processor for Machine Learning. *Symp. on High Performance Chips (Hot Chips)*, Aug 2017.

[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.

[BFJ$^+$96] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms. *Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun 1996.

[BFS12] A. Branover, D. Foley, and M. Steinman. AMD Fusion APU: Llano. *IEEE Micro*, 32(2):28–37, Mar/Apr 2012.

[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)*, Aug 1995.

[BKSL08] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct 2008.

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

[BLS$^+$20] J. Balkind, K. Lim, M. Schaffner, F. Gao, G. Chirkov, A. Li, A. Lavrov, T. M. Nguyen, Y. Fu, F. Zaruba, K. Gulati, L. Benini, and D. Wentzlaff. BYOC: A "Bring Your Own Core" Framework for Heterogeneous-ISA Research. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2020.

[BMF$^+$16] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff. OpenPiton: an Open Source Hardware Platform for Your Research. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2016.

[BMF$^+$19] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff. OpenPiton: an Open Source Hardware Platform for Your Researc. *Communications of the ACM*, 62(12):79–87, Dec 2019.

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

[BS13] N. Beckmann and D. Sanchez. Jigsaw: Scalable Software-Defined Caches. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2013.

[BSL$^+$02] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: Design Alternative for Cache On-Chip Memory in Embedded Systems. *Intl'l Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, May 2002.

[BSP$^+$17] B. Bohnenstiehl, A. Stillmaker, J. J. Pimentel, T. Andreas, B. Liu, A. T. Tran, E. Adeagbo, and B. M. Baas. KiloCore: A 32-nm 1000-Processor Computational Array. *IEEE Journal of Solid-State Circuits (JSSC)*, 52(4):891–902, Apr 2017.

[BZS93] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. *The Midway Distributed Shared Memory System.* Digest of Papers. Compcon Spring, 1993.

[cci20] The CCIX Consortium. Online Webpage, 2020 (accessed Nov 27, 2020). `https://www.ccixconsortium.com`.

[CGS$^+$05] P. Charles, C. Grothoff, V. Sarkar, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. *Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, Oct 2005.

[cha20] Chapel: Productive Parallel Programming. Online Webpage, 2020 (accessed Nov 17, 2020). `https://chapel-lang.org`.

[CKS$^+$11] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct 2011.

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

[CLS05] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2005.

[CRG$^+$11] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato. Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2011.

[CS12] J. Chung and K. Strauss. User-Level Interrupt Mechanism for Multi-Core Architectures. US Patent 8255603, Aug 2012.

[cud13] CUDA. Online Webpage, 2013 (accessed Nov 17, 2013). `http://www.nvidia.com/object/cuda_home_new.html`.

[DWB$^+$91] M. Dubois, J. C. Wang, L. A. Barroso, K. Lee, and Y.-S. Chen. Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*, Aug 1991.

[DXT$^+$18] S. Davidson, S. Xie, C. Torng, K. Al-Hawaj, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. G. Dreslinski, C. Batten, and M. B. Taylor. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro*, 38(2):30–41, Mar/Apr 2018.

[EN14] M. Elver and V. Nagarajan. TSO-CC: Consistency Directed Cache Coherence for TSO. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2014.

[FLKBF11] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo Directory: A Scalable Directory for Many-Core Systems. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2011.

[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)*, May 1998.

[FW15] Y. Fu and D. Wentzlaff. Coherence Domain Restriction on Large Scale Systems. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2015.

[GdWM90] A. Gupta, W. dietrich Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. *Int'l Conf. on Parallel Processing (ICPP)*, Aug 1990.

[GLL$^+$90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *Int'l Symp. on Computer Architecture (ISCA)*, May 1990.

[Gre11] P. Greenhalgh. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *EE Times*, Oct 2011. `http://www.eetimes.com/document.asp?doc_id=1279167`.

[Gwe20a] L. Gwennap. Ampere Pushes Neoverse to 80 Cores. *Microprocessor Report, The Linley Group*, Apr 2020.

[Gwe20b] L. Gwennap. Tiger Lake Debuts 10nm SuperFin. *Microprocessor Report, The Linley Group*, Sep 2020.

[Hal20] T. R. Halfhill. ThunderX3's Cloudburst of Threads. *Microprocessor Report, The Linley Group*, Apr 2020.

[HCH$^+$14] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. QuickRelease: A Throughput-Oriented Approach to Release Consistency on GPUs. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2014.

[HLL10] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview Scalability Analyzer. *Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun 2010.

[How10] J. Howard. A 48-core IA-32 Processor with On-Die Message-Passing and DVFS in 45nm CMOS. *Asian Solid-State Circuits Conf. (ASSCC)*, pages 1–4, 2010.

[HVS$^+$07] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar. A 5-GHz Mesh Interconnect for a Teraflops Processor. *IEEE Micro*, 27(5):51–61, Sep/Oct 2007.

[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`.

[ISL96] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. *Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun 1996.

[JPOB20] S. Jiang, P. Pan, Y. Ou, and C. Batten. PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification. *IEEE Micro*, 40(4):58–66, Jul/Aug 2020.

[JVA^{+}09] R. L. B. Jr, M. Vakilian, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, and H. Sung. A Type and Effect System for Deterministic Parallel Java. *Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, Oct 2009.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. *Int'l Symp. on Computer Architecture (ISCA)*, May 1992.

[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.

[Kim17] J. Kim. *Software/Hardware Co-Design to Improve Productivity, Portability, and Performance of Loop-Task Parallel Applications*. Ph.D. Thesis, Cornell University, 2017.

[KJJ^{+}09] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2009.

[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.

[KK10] S. Kaxiras and G. Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro*, 30(5):54–65, Sep 2010.

[KMPW11] R. Kumar, T. G. Mattson, G. Pokam, and R. V. D. Wijngaart. The Case for Message Passing on Many-Core Chips. *Multiprocessor System-on-Chip*, pages 115–123, Dec 2011.

[KSA^{+}15] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve. Stash: Have Your Scratchpad and Cache It Too. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2015.

[KSV15] S. Kumar, A. Shriraman, and N. Vedula. Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2015.

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

[LLS$^+$13] I.-T. A. Lee, C. E. Leiserson, T. B. Schardl, J. Sukha, and Z. Zhang. On-the-Fly Pipeline Parallelism. *Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun 2013.

[LMLV11] E. Ladan-Mozes, I.-T. A. Lee, and D. Vyukov. Location-Based Memory Fences. *Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun 2011.

[LSC$^+$13] M. Lis, K. S. Shim, M. H. Cho, I. Lebedev, and S. Devadas. Hardware-Level Thread Migration in a 110-Core Shared-Memory Multiprocessor. Technical Report 512, MIT CSAIL CSG, Nov 2013.

[LW95] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. *Int'l Symp. on Computer Architecture (ISCA)*, Jul 1995.

[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.

[MBS16] A. Mukkara, N. Beckmann, and D. Sanchez. Whirlpool: Improving Dynamic Cache Management with Static Data Classification. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2016.

[MFN$^+$17] M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, J. Balkind, A. Lavrov, M. Shahrad, S. Payne, and D. Wentzlaff. Piton: A Manycore Processor for Multitenant Clouds. *IEEE Micro*, 37(2):70–80, Mar/Apr 2017.

[MHS$^+$03] M. M. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-memory Multiprocessors. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2003.

[MHS12] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why On-chip Cache Coherence is Here to Stay. *Communications of the ACM*, Jul 2012.

[MHW03] M. M. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2003.

[Mos05] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2005.

[mpi13] Message Passing Interface (MPI) Standard. Online Webpage, 2013 (accessed Nov 17, 2013). `http://www.mcs.anl.gov/research/projects/mpi/standard.html`.

[MRR12] M. McCool, A. D. Robinson, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012.

[MSM05] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2005.

[MV15] S. Mittal and J. S. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys*, Jul 2015.

[NLP13] D. Nguyen, A. Lenharth, and K. Pingali. A Lightweight Infrastructure for Graph Analytics. *Symp. on Operating Systems Principles (SOSP)*, Nov 2013.

[nvi20] NVIDIA A100 Tensor Core GPU Architecture. NVIDIA White Paper, 2020. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf`.

[Olo16] A. Olofsson. Epiphany-V: A 1024-processor 64-bit RISC System-On-Chip. *CoRR arXiv:1610.01832*, Aug 2016.

[ope11] OpenCL Specification, v1.2. Khronos Working Group, 2011. `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`.

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

[ora20] OpenSPARC T1. Online Webpage, 2020 (accessed Dec 10, 2020). `https://www.oracle.com/servers/technologies/opensparc-t1-page.html`.

[PBG$^+$13] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous System Coherence for Integrated CPU-GPU Systems. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2013.

[PSNB10] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian. SWEL: Hardware Cache Coherence Protocols to Map Shared Data onto Shared Caches. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2010.

[Ram11] C. Ramey. TILE-Gx100 ManyCore Processor: Acceleration Interfaces and Architecture. *Symp. on High Performance Chips (Hot Chips)*, Aug 2011.

[RBM08] A. Raghavan, C. Blundell, and M. M. Martin. Token Tenure: PATCHing Token Counting Using Directory-Based Cache Coherence. *Int'l Symp. on Microarchitecture (MICRO)*, Nov 2008.

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

[ris17] The RISC-V Instruction Set Manual Volume I: User-Level ISA. Online Webpage, 2017 (accessed Aug 15, 2017). `https://riscv.org/specifications`.

[ris19] The RISC-V Instruction Set Manual Volume II: Privileged Architecture. Online Webpage, 2019 (accessed Jun 8, 2019). `https://riscv.org/specifications/privileged-isa/`.

[RK12] A. Ros and S. Kaxiras. Complexity-Effective Multicore Coherence. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2012.

[Rup21] 42 Years of Microprocessor Trend Data. Online Webpage, 2018 (accessed Apr 21, 2021). `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data`.

[SA15] H. Sung and S. V. Adve. DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2015.

[SAA17] M. D. Sinclair, J. Alsop, and S. V. Adve. Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[SB13] J. Shun and G. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Feb 2013.

[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.

[SBJS15] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. CAPI: A Coherent Accelerator Processor Interface. *IBM Journal of Research and Development*, 59(1):7:1–7:7, Jan/Feb 2015.

[SFR$^+$14] L. Song, M. Feng, N. Ravi, Y. Yang, and S. Chakradhar. COMP: Compiler Optimizations for Manycore Processors. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[SGC+16] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46, Mar/Apr 2016.

[SHW11] D. J. Sorin, M. D. Hill, and D. A. Wood. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 2011.

[SK12] D. Sanchez and C. Kozyrakis. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2012.

[SKA13] H. Sung, R. Komuravelli, and S. V. Adve. DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr 2013.

[SML17] T. B. Schardl, W. S. Moses, and C. E. Leiserson. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Jan 2017.

[SSF+13] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. Cache Coherence for GPU Architectures. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2013.

[SSI+18] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, and B. Allison. IBM POWER9 Opens Up a New Era of Acceleration Enablement: OpenCAPI. *IBM Journal of Research and Development*, 62(4/5):8:1–8:8, Jul/Aug 2018.

[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.

[Tay18] M. B. Taylor. BaseJump STL: SystemVerilog Needs a Standard Template Library for Hardware Design. *Design Automation Conf. (DAC)*, Jun 2018.

[TKM+03] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, W. Lee, A. Saraf, N. Shnidman, V. Strumpen, S. Amarasinghe, and A. Agarwal. A 16-Issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2003.

[TLM+04] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[Tor19] C. Torng. *Software, Architecture, and VLSI Co-Design for Fine-Grain Voltage and Frequency Scaling*. Ph.D. Thesis, Cornell University, 2019.

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

[upc13] Unified Parallel C. Online Webpage, 2013 (accessed Nov 17, 2013). `http://upc.lbl.gov/`.

[vECGS92] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. *Int'l Symp. on Computer Architecture (ISCA)*, May 1992.

[VHR+07] S. Vangali, J. Howard, G. Ruhi, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyerl, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskotel, and N. Borkarl. 80-Tile 1.28 TFlops Network-on-Chip in 65 nm CMOS. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2007.

[WCD+10] L. Wang, H. Cui, Y. Duan, F. Lu, X. Feng, and P.-C. Yew. An Adaptive Task Creation Strategy for Work-Stealing Scheduling. *Int'l Symp. on Code Generation and Optimization (CGO)*, Apr 2010.

[WGH+07] D. Wentzlaff, P. Griffin, H. Hoffman, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27:15–31, Sep/Oct 2007.

[Whe20] B. Wheeler. Ampere Maxes Out at 128 Cores. *Microprocessor Report, The Linley Group*, Jul 2020.

[WTCB20] M. Wang, T. Ta, L. Cheng, and C. Batten. Efficiently Supporting Dynamic Task Parallelism on Heterogeneous Cache-Coherent Systems. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2020.

[YKM+11] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A Fully Integrated Multi-CPU, GPU, and Memory Controller 32 nm Processor. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2011.

[ZB19] F. Zaruba and L. Benini. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FD-SOI Technology. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, 27(11):2629–2640, Nov 2019.

[ZSD10] H. Zhao, A. Shriraman, and S. Dwarkadas. SPACE: Sharing Pattern-Based Directory Coherence for Multicore Scalability. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2010.

[ZSM07] J. Zebchuk, E. Safi, and A. Moshovos. A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2007.