University of California
Los Angeles

Memory System Optimizations for Customized Computing – From Single-Chip to Datacenter

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

by

Yu-Ting Chen

2016

Abstract of the Dissertation

Memory System Optimizations for Customized Computing – From Single-Chip to Datacenter

by

Yu-Ting Chen

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2016

Professor Jingsheng Jason Cong, Chair

This dissertation investigates memory system optimizations for customized computing from the single-chip level to the datacenter level. The efficiency of a memory system determines the chip performance and energy efficiency. Our goal is to design a high-performance and energy-efficient memory system that supports customized computing in both general-purpose processors and accelerator-rich architectures (ARAs). Furthermore, we would like to explore the optimizations and customized support at the datacenter level since some of the application domains require computing power beyond a single chip.

This dissertation is composed of the following three topics. The first two topics investigate the memory system design and optimizations of the single-chip level. The third topic discusses the optimization and customization support in a datacenter.

First, we focus on performance modeling and energy reduction for on-chip multilevel caches, which are widely used in general-purpose systems. We propose a fast co-simulation framework to simulate a memory system with both cache and scratch-

pad memories for efficient design space exploration in customized computing. We further propose a hybrid cache architecture which is composed of both SRAM and non-volatile memories (NVM) to reduce leakage. Our proposed optimization techniques can hide the drawbacks of NVM and significantly reduce power.

Moreover, we believe that an ARA is an attractive alternative to general-purpose processors due to its high performance and energy efficiency. Therefore, we need a methodology to fast evaluate the performance of an ARA. Furthermore, accelerators usually demand higher on-chip and off-chip memory bandwidth. We provide an optimized two-level on-chip interconnect synthesis algorithm for designing the ARA memory system.

Third, we propose customized optimizations at the datacenter level to improve the efficiency of using in-memory computation infrastructure. The current single-chip and single-server solutions cannot meet performance target of the DNA sequencing pipelines, which is an important domain we are looking into. In addition to using only general-purpose processors for computation, we further provide customized support of using FPGA accelerators in the datacenter.

At the single-chip level, we cover the following two topics.

Simulation and Optimizations for Multilevel Caches: Performing simulations on the application(s) is the most popular way to find the optimal memory configuration. However, the simulation-based approach suffers from long simulation time due to the need to exhaustively simulate all configurations. In this dissertation we propose a novel simulator, HC-Sim, which adopts elaborate data structures, a centralized hash table, and a novel miss counter structure to effectively reduce the simulation time. Furthermore, HC-Sim can simulate L1 caches and a scratchpad

memory (SPM) simultaneously. SPMs are usually used for customized accelerators. HC-Sim helps designers to explore the design space considering both L1 cache configurations and the SPM sizes.

Energy-efficiency is an important goal for on-chip multilevel caches, especially for L2 caches or last-level caches (LLCs). We first propose a novel reconfigurable hybrid cache architecture (RHC), in which NVM is incorporated in the last-level cache together with traditional SRAM cells. RHC can be reconfigured by powering on/off SRAM/NVM arrays in a way-based manner. In addition, we provide hardware-based mechanisms to dynamically reconfigure RHC on-the-fly based on the cache demand. Second, a combined static and dynamic scheme is proposed to optimize the block placement for energy-efficiency and endurance in a hybrid cache. We use the compiler to provide static hints to guide initial data placement, and use the hardware to correct the hints based on the run-time cache behavior. With the proposed scheme, the write energy on NVM is significantly reduced while performance is maintained. Moreover, NVM endurance is maximized.

Memory System Optimization for Accelerator-Rich Architectures: An accelerator-rich architecture (ARA) is an attractive solution to provide high performance and energy efficiency for replacing current state-of-the-art general-purpose processors in the age of dark silicon. It is important to evaluate the performance and energy of an ARA design. Instead of running extremely time-consuming full-system simulation, we develop the ARAPrototyper, a highly automated and highly parameterized prototyping flow with runtime system and user APIs for fast prototyping and evaluation. Furthermore, we observe that the memory system efficiency is key to enabling a high-performance ARA. We provided an optimized two-level interconnect synthesized algorithm for synthesizing the memory system for ARAs. Our prototype

achieves 7.44X energy-efficiency gain over state-of-the-art processors.

Datacenter-Level Optimizations in an In-Memory Cluster Computing Framework: At the datacenter level, many big data applications need computation power that is beyond a single server. In order to provide acceleration on such applications, we adopt the in-memory cluster computing infrastructure, Spark, to provide a scalable speedup. We are interested in accelerating the DNA sequencing pipeline, which inherits both big data and compute-intensive characteristics.

In this dissertation we accelerate the first and one of the most time-consuming steps, read alignment, in the DNA sequencing pipeline by using in-memory cluster computing together with FPGA acceleration. We first design cloud-scale BWAMEM (CS-BWAMEM), a cloud-scale read aligner built on top of Spark. We provide customized optimization strategies in the in-memory cluster, such as broadcast avoidance of the shared large human reference genome, data caching, and pipelining I/O with computation, to improve system performance. CS-BWAMEM can outperform the state-of-the-art aligners by more than 18x and can finish the whole-genome alignment in 32 minutes. The proposed optimizations are not only limited to the DNA sequencing application but can be applied to the other large-scale data processing applications.

Furthermore, we provide customized support to deploy the customized FPGA accelerators in Spark to further accelerate the compute-intensive kernel, such as Smith-Waterman algorithm. We provide runtime system support and algorithm-level optimization to reduce the data transfer overhead between Spark workers and FPGA accelerators. Our proposed batch processing mechanism can groups small tasks in a batch and thus accelerators can process data in a batched way to reduce communication overhead.

The dissertation of Yu-Ting Chen is approved.

Giovanni Coppola

Tyson Condie

Glenn D. Reinman

Jingsheng Jason Cong, Committee Chair

University of California, Los Angeles

2016

To my family and my fiancé

for their love, concern, and constant support.

To all the people who supported and helped me all these years.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my sincerest gratitude to my advisor, Professor Jason Cong, for his constant support, professional advice, and insightful guidance during my Ph.D. studies at UCLA. Professor Cong is a Chancellor's Professor in the Computer Science Department and Electrical Engineering Department at UCLA, Director of the Center for Customizable Domain-Specific Computing, and Director of the UCLA VAST Lab. I would not have reached this far in my academic career without our weekly group discussions, email exchanges, and other offline discussions which inspired my research ideas. From Professor Cong I not only learned the principles of conducting research, but I also learned many things beyond research. More importantly, he encouraged me to set high standards and goals for my research. This conviction to pursue excellence is one of most valuable things I learned during my research career at UCLA. It is my great fortune and honor to have Jason Cong as my advisor.

My gratitude further goes to Professor Glenn Reinman for his dedicated guidance in the cache simulation and optimizations and accelerator-rich architecture research in the dissertation. His insightful suggestions and rich experience in computer architecture research really brought further improvements to the dissertation.

I am also very grateful to my committee members, Professor Tyson Condie and Professor Giovanni Coppola. Professor Condie gave me valuable suggestions in the cluster computing field so that we can apply hardware accelerators in the datacenter. Professor Coppola introduced me to a large amount of background knowledge in the biological field and helped me bridge the gap between the computer science and the biological fields. Through discussions, I and my fellow researchers came to know the

demands and goals of biological applications. Professor Coppola's feedback helped us further improve our software stack for processing the DNA sequencing data.

I would also like to thank Janice Wheeler for her help in editing the language of my papers and this dissertation. She always tried her best to edit my papers, even if I sent her those papers just a couple of days before submission deadlines. Without Janice's help, I wouldn't have been able get these important papers edited—including attention to accurate word selection—before submission deadlines.

Furthermore, I would like to thank all my fellow researchers who contributed in some way to this dissertation. In Chapters 3 and 4, Chunyue Liu had many discussions with me so that I could refine my ideas. He also shared his experience with me in the design of the full-system simulation framework for architectural explorations. In Chapter 4, Hui Huang and Raghu Prabhakar helped design the compiler framework to generate codes for simulation.

In Chapter 5, the dissertation presents a prototyping framework for evaluating performance of accelerator-rich architectures. This is a multi-year research effort with a large amount of collaborative effort from our group. Bingjun Xiao, Peipei Zhou, Zhenman Fang, Muhuan Huang, Mohammad Ali Ghodrat, Chunyue Liu, and Yi Zou were all involved in the multi-year development process.

In Chapter 7, Peng Wei, Sen Li, and Peipei Zhou contributed to the implementation of the cloud-level aligner, CS-BWAMEM. In Chapter 8, Peng Wei helped design and implement the batch processing algorithm. Professor Jie Lei contributed to the detailed implementation of the Smith-Waterman FPGA acceleration engine.

I am also thankful to my fellow researchers in the Center for Domain-Specific Computing (CDSC) and the VAST Lab who made my graduate study colorful and

wonderful. I would like to thank Hao Yu and Po-Tsang Huang for their collaboration on a recent research paper. I want to thank Karthik Gururaj, Sen Li and Di Wu for their efforts toward maintaining a stable platform so that I could conduct my research smoothly. Moreover, I would like to thank all the faculty and stuff in the Computer Science Department at UCLA for helping me successfully complete my Ph.D. program.

I thank all my collaborators in the biological field, including Professor Paul Spellman (OHSU), Professor Giovanni Coppola (UCLA), Professor Guoping Fan (UCLA), Myron Peto (OHSU), Jason Chen (UCLA), Kevin Huang (UCLA), and Yu-Chih Chen (University of Michigan). Through our interdisciplinary collaborations, these researchers helped to broaden my view beyond the computer science field. Thanks to them, I gained abundant knowledge in both biological and genomic fields. I am also thankful to all our Intel friends, including George Powley, Mishali Naik, Karthik Gururaj, and Ganapati Srinivasa, for their support on Intel HARP platform, the knowledge sharing in the genomics domain, and their kind donation of equipment to our lab.

My love and thanks to my parents and my fiancé for their constant love, concern, support, patience and strength all these years.

The work in this dissertation is partially supported by the NSF Expeditions in Computing Award CCF-0926127; and CFAR (Center for Future Architecture Research), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA), and NSF Innovation Transition (InTrans) Program (NSF Award CCF-1436827) with Intel's support (Intel Award 20134321), NSF Award CCF-1436827, and SRC Contract 2009-TJ-1984.

Vita

2001–2005	B.S., Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan
2001–2005	B.A., Department of Economics, National Tsing-Hua University, Hsinchu, Taiwan
2005–2007	M.S., Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan
2009–2016	Research Assistant, Department of Computer Science, University of California, Los Angeles, U.S.A.

Publications

[Book] Yu-Ting Chen, Jason Cong, Michael Gill, Glenn Reinman, and Bingjun Xiao, "Customizable Computing," *Synthesis Lectures on Computer Architecture*, vol. 10, issue 3, pp. 1-118, Morgan & Claypool Publishers, July 2015.

Mau-Chung Frank Chang, Yu-Ting Chen, Jason Cong, Po-Tsang Huang, Chun-Liang Kuo and Cody Hao Yu, "The SMEM Seeding Acceleration for DNA Sequence Alignment," *International Symposium on Field-Programmable Custom Computing Machines* (*FCCM*), 2016.

Yu-Ting Chen, Jason Cong, Zhenman Fang, and Peipei Zhou, "ARAPrototyper: Enabling Rapid Prototyping and Evaluation for Accelerator-Rich Architecture," *International Symposium on Field-Programmable Gate Arrays* (*FPGA*), 2016.

Yu-Ting Chen, Jason Cong, Jie Lei, Sen Li, Myron Peto, Paul Spellman, Peng Wei, and Peipei Zhou, "CS-BWAMEM: A fast and scalable read aligner at the cloud

scale for whole genome sequencing," *High Throughput Sequencing, Algorithms and Applications* (*HiTSeq*), 2015. **(Best Poster Award)**

Yu-Ting Chen and Jason Cong. "Interconnect Synthesis of Heterogeneous Accelerators in a Shared Memory Architecture," *International Symposium on Low Power Electronics and Design* (*ISLPED*), 2015, pp. 359–364.

Yu-Ting Chen, Jason Cong, Jie Lei, and Peng Wei. "A Novel High-Throughput Acceleration Engine for Read Alignment," *International Symposium on Field-Programmable Custom Computing Machines* (*FCCM*), 2015, pp. 199–202.

Yu-Ting Chen, Jason Cong, and Bingjun Xiao. "ARACompiler: A Prototyping Flow and Evaluation Framework for Accelerator-Rich Architectures," *International Symposium on Performance Analysis of Systems and Software* (*ISPASS*), 2015.

Yu-Ting Chen, Jason Cong, Mohammad Ali Ghodrat, Muhuan Huang, Chunyue Liu, Bingjun Xiao and Yi Zou, "Accelerator-Rich CMPs: From Concept to Real Hardware," *International Conference on Computer Design* (*ICCD*), 2013.

Yu-Ting Chen, Jason Cong, Hui Huang, Chunyue Liu and Glenn Reinman, "Static and Dynamic Co-Optimizations for Blocks Mapping in Hybrid Caches," *International Symposium on Low Power Electronics and Design* (*ISLPED*), 2012, pp. 237–242.

Yu-Ting Chen, Jason Cong, Hui Huang, Chunyue Liu and Glenn Reinman, "Dynamically Reconfigurable Hybrid Cache: An Energy-Efficient Last-Level Cache Design," *Design, Automation and Test in Europe* (*DATE*), 2012, pp. 45–50.

Yu-Ting Chen, Jason Cong and Glenn Reinman, "HC-Sim: A Fast and Exact L1 Cache Simulator with Scratchpad Memory Co-Simulation Support," *International Conference on Hardware/Software Codesign and System Synthesis* (*CODES*), 2011.

CHAPTER 1

Introduction

1.1 Motivation

Energy efficiency is one of the key considerations for various systems, from handheld devices to servers in a data center. Designing an energy-efficient processor that matches the target performance is currently one of our most important goals. Prior to the early 2000s, the performance of computers doubled every 18 months or 24 months while keeping the power constant through Dennard scaling [63]. However, after the early 2000s, frequency scaling could not sustain this momentium. The computing industry then entered into the multicore era, and performance improved through exploiting the parallelization. Such highly parallel, general-purpose computing systems still face serious challenges in terms of performance, power, heat dissipation, space, and cost. Recently, researchers have pointed out that general-purpose multicore scaling cannot be sustained due to the power budget limit [67, 160]. Domain-specific customization is a disruptive technology to attack that power wall [59].

Users or enterprises usually focus on a group of selected applications in one domain. For example, the media content provider has a high computing demand to provide audio and video streams. With application-specific accelerators customized for a specific domain, computation can be handled with better performance and en-

ergy efficiency. Normally, application-specific accelerators can provide 10 to 1000X energy-efficiency improvement over general-purpose processors through custimization and by exploiting the application parallelism [79, 31, 30]. In this dissertation we focus on (1) the medical imaging domain [25], and (2) the genomics and DNA sequencing domain [129]; both are discussed in Section 1.5.

In a customized computing system, the accelerators and general-purpose processors cooperate together to meet the demand of both high performance and low power. The design of the memory system is the key to providing sufficient data for accelerators and processors. The goal of the memory system design should be to provide short latency of memory accesses and high data bandwidth with low energy consumption.

This dissertations covers the memory system optimizations at two levels — the single-chip level and the datacenter level. In the single-chip level, we discuss two topics: (1) simulation and optimizations for energy-efficient multilevel caches, and (2) memory system optimization for accelerator-rich architectures. In the datacenter level, we provide customized optimizations for in-memory cluster computing frameworks, especially for the DNA sequencing domain.

The first focus of this dissertation is simulation and optimization for a high-performance and energy-efficient cache system. An on-chip multilevel CPU cache is widely used in modern processors. In order to model the CPU caches with multiple configurations efficiently, we provide an accurate and fast co-simulation framework, HC-Sim [40], for L1 caches and scratchpad memory, which is discussed in Chapter 2. Next, we try to provide an efficient and low-power cache architecture, which is important for general-purpose processors but can possibly be used for a customized-computing system in the future. We utilize disparate memory technologies to compose L2 caches or last-level caches (LLC) to reduce leakage power. This kind of cache design

is called "hybrid cache." We propose a reference architecture to implement a hybrid cache. To further improve energy efficiency, we design a dynamic reconfiguration scheme for so that a hybrid cache can adapt to the program behavior and reduce energy consumption while maintaining performance during runtime [37]; this will be discussed in Chapter 3. Moreover, we use the compiler hints and runtime hardware assist together to save the energy and enhance the lifetime of the proposed hybrid cache [38], as discussed in Chapter 4.

The second focus of this dissertation is the performance evaluation and memory system design for an accelerator-rich architecture (ARA). ARAs can provide energy-efficient solutions for domain-specific computing in the age of dark silicon [67] with the use of accelerators to replace energy-hungry general-purpose processors. Instead of running extremely time-consuming full-system simulations, we design a highly-automated and highly-parameterized prototyping flow together with runtime system and user APIs for fast performance and accurate energy evaluation. We called this plaform "ARAPrototyper" [34, 41, 36], which will be discussed in Chapter 5. A real ARA prototype can be deployed on the Xilinx Zynq platform [89], which is a system-on-a-chip (SoC) with processors and accelerators synthesized using FPGA. We further provide a solid theoretical basis for synthesizing the on-chip interconnects of the ARA memory system in Chapter 6. Our proposed algorithm for synthesizing partial crossbar is scalable enough for hundreds of accelerators with minimum switches [33].

The third focus of this dissertation is to optimization and customize support in the datacenter level. We especially customize our system for the domain of genomics and DNA sequencing. In general, the scale of computation and the data storage can be way beyond a single node in the big data era. For the genomics and DNA sequencing

domain, the amount of data can achieve the scale of hundreds of TBs to hundreds of PBs. Therefore, the single-node performance may not meet the requirement. We believe the scale-out solution that uses a cluster computing system is the way to bridge the performance gap in the genomics domain. To accelerate genomics applications, we use the in-memory cluster computing system, Spark [171], which distributes the computations across the nodes in a cluster. We design a cloud-scale aligner, called CS-BWAMEM [39], to accelerate the time-consuming alignment process. We further introduce multiple customized techniques to improve the performance in the DNA sequencing domain in Chapter 7. In addition to the scale-out solution, the scale-up strategy is also important for improving the system-wide performance. By using FPGA accelerators, we can further improve the performance by leveraging compute-intensive kernels into customized accelerators. The details of the runtime system and optimization strategies are discussed in Chapter 8.

Figure 1.1 demonstrates the overall scope of this dissertation. Chapter 2 discusses HC-Sim, a fast and exact simulator across both the processor plane and the accelerator plane. HC-Sim can efficiently determine the optimized L1 cache size for a processor when the buffers required by accelerators are provided. Chapter 3 introduces the hybrid cache architectures for LLCs and dynamic reconfiguration mechanism for reducing leakage. Chapter 4 discusses the software/hardware co-optimization scheme to enhance the endurance and reduce the high write power of hybrid caches. Hybrid caches can significantly reduce the energy and provide large density for on-chip memory in the processor plane in an ARA. Chapter 5 discusses the ARAPrototyper, an efficient prototyping and evaluation platform for ARAs. Chapter 6 provides the interconnect optimization for ARA memory system. In Chapter 7, we provide optimizations in the datacenter level. We use cloud-scale BWAMEM (CS-BWAMEM), an

aligner that uses the computing power of an in-memory cluster for accelerating DNA sequencing processes, to demonstrate our proposed optimizations in an in-memory cluster. In Chapter 8 we investigate how to use FPGA hardware accelerators in an in-memory cluster to further provide system-wide and cluster-wide speedup.

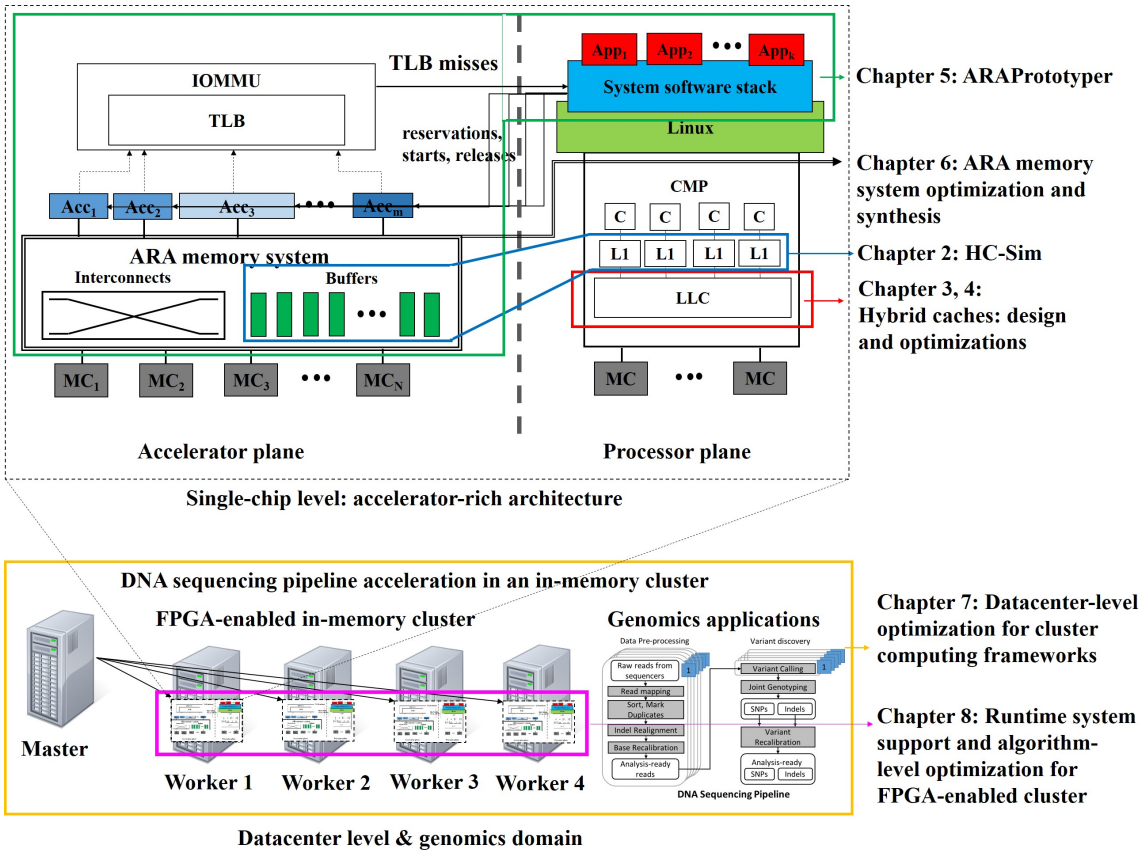


Figure 1.1: Dissertation overview

The remainder of this chapter gives an introduction to the topics covered in this dissertation. Section 1.2 describes simulation and optimizations in multilevel caches. We introduce the HC-Sim and the proposed hybrid cache architectures. Section 1.3 describes ARA memory system optimizations. We introduce a fast prototyping and evaluation flow, ARAPrototyper. We also discuss the interconnect synthesis

algorithm of the ARA memory system. Section 1.4 describes the datacenter-level optimizations and how we enable the FPGA acceleration in an in-memory cluster computing system for accelerating the applications in the genomics domain. Section 1.5 discusses the two domains of interest in this dissertation, including (1) the medical imaging pipeline, and (2) the DNA sequencing pipeline.

1.2 Simulation and Optimizations for Multilevel Caches

An on-chip multilevel CPU cache is widely used in modern processors. It usually occupies more than 50% of die area in modern processors [10]. In order to support efficient computation in modern processors and ARAs, the memory system needs to be designed efficiently considering both performance and energy. Therefore, accurate and fast performance modeling is important. We first investigate the modeling of the L1 caches together with scratchpad memories (SPMs). We build HC-Sim for fast evaluation and perform efficient design space exploration for multiple caches and SPM configurations.

In addition to system performance, power and energy consumption is another key concern in modern processors or ARAs. Most importantly, the energy consumption of memory systems is one of the major sources of consumption. Compared to the arithmatic logic units (ALUs) in CPU and accelerators, a memory system can consume more energy, especially the leakage power, when the system is idle. Therefore, the optimization considering energy efficiency is important. In this dissertation we provide a hybrid cache architecture from both architecture design and circuit-level implementation. Hybrid caches use both non-volatile memory technology and traditional SRAM cells to build L2 and LLC, which account for the largest portion of

an ARA, to reduce leakage. We further provide multiple optimization schemes to improve the energy efficiency and endurance of the hybrid caches while maintaining system performance.

1.2.1 HC-Sim

In order to build an efficient memory system for customizable domain-specific computing, the first step is to have an accurate performance model to guide the memory system design. However, it is very difficult to develop an accurate analytical model due to the complicated memory access patterns. This dissertation takes the simulation approach to obtain accurate cache miss rates as measurement to estimate the performance of a targeted application.

In domain-specific computing, the computation-intensive portions of an application are usually processed by customized accelerators. The memory access patterns of the accelerators can be known in advance during design time through compiler analysis. SPMs can be used as the on-chip memories for the accelerators to replace traditional CPU caches for better performance and energy consumption [43, 19, 97, 49]. The CPU caches are used to handle on-chip memory accesses from CPUs. Both SPMs and caches can have a significant impact on system performance.

The main contributions of HC-Sim can be summarized as follows.

(1) HC-Sim can simulate multiple L1 cache configurations with one SPM configuration simultaneously, and thus significantly improve the simulation speed. In HC-Sim we propose novel data structures and algorithms which incorporate a centralized hash table and a novel miss counter structure to avoid time-consuming tag searches on the stacks. Compared to the state-of-the-art algo-

rithms, SuSeSim [80] and CRCB [156], HC-Sim is 2.56X and 5.44X faster than those algorithms, respectively. Furthermore, we show that the efficiency of HC-Sim can be improved by using a miss-counter-based structure for a group of applications.

(2) To enhance scalability of HC-Sim, we build our simulation framework on the dynamic binary instrumentation tool, Pin [121]. The trace generation is embedded into HC-Sim and is performed on-the-fly to avoid the overhead of a huge trace file.

(3) We provide an interface for designers to specify the address range of SPM so that the SPM accesses can be bypassed, and the correct miss rates of L1 caches can be simulated. We also provide a mechanism to filter out prefetching loads, which arise from SPM prefetching, to maintain the correctness of L1 cache simulation.

1.2.2 Hybrid Caches—Design and Optimizations

The traditional SRAM-based on-chip cache has become a bottleneck for energy-efficient design due to its high leakage power. Designers have turned their attention towards emerging non-volatile memories (NVMs), such as the spin-torque transfer magnetoresistive RAM (STT-RAM) and phase change RAM (PRAM), to build future memory systems. Power, performance, and density characteristics of the new memory technologies differ dramatically compared to SRAM, and thus they enlarge the landscape of memory design.

Table 1.1 shows a brief comparison of SRAM, STT-RAM, and PRAM technologies. The exact access time and dynamic power depend on the cache size and the peripheral

circuit implementation. In sum, SRAM suffers from high leakage and low density while providing great endurance; STT-RAM and PRAM provide high density and low leakage at the cost of weak endurance. Moreover, STT-RAM outperforms PRAM in terms of the endurance, access time, and dynamic power, while PRAM has higher density. Based on the endurance, STT-RAM is more suitable for on-chip last-level cache [165, 154, 65, 143] design due to its higher endurance, while PRAM is promising as an alternative for DRAM in the main memory design due to its higher density [107, 140]. Therefore, in this dissertation we will focus on a hybrid cache architecture with STT-RAM.

Table 1.1: Comparison among SRAM, STT-RAM, and PRAM.

	SRAM	STT-RAM	PRAM
Density	1X	4X	16X
Read time	Very fast	Fast	Slow
Write time	Very fast	Slow	Very slow
Read power	Low	Low	Medium
Write power	Low	High	High
Leak. power	High	Low	Low
Endurance	10^{16}	4×10^{12}[32]	10^{9}

Hybrid caches can provide larger cache sizes than those of traditional SRAM caches under the same area [165, 154]. In the LLC design, a hybrid cache can have less cache misses with a larger cache size, and thus has better performance than a traditional SRAM cache. However, the previous work [165, 154] does not consider the opportunities to further reduce the energy by dynamically reconfiguring the hybrid cache In this dissertation we propose a novel reconfigurable hybrid cache design (*RHC*). Our design explores the use of NVM cache to replace the conventional all-

SRAM design in the last-level cache to efficiently reduce leakage energy. The proposed RHC design supports reconfigurable SRAM/NVM size, with the capability of powering on/off SRAM and NVM arrays in a way-based manner for better accommodation of memory requirements from different workloads. Hardware-based mechanisms are proposed to detect the cache demand for dynamic reconfiguration. On average, RHC significantly saves 64%, 46% and 28% energy over a non-reconfigurable SRAM cache, a non-reconfigurable hybrid cache and a reconfigurable SRAM cache, respectively, while maintaining the system performance (at most, only a 4% performance overhead).

As shown in Table 1.1, the endurance and the high dynamic write power of STT-RAM are the two disadvantages compared to SRAM. Although ITRS predicts that the write cycles of STT-RAM will be 10^{15} at 2024 [87], the best available write cycles of STT-RAM are 4 x 10^{12} at present [32]. Suppose we execute *segmentation* [25], a medical imaging application, on a 4GHz CPU with 32 KB L1 cache, 2MB STT-RAM L2 cache continuously, the lifetime of a STT-RAM cache can last only 2.17 years without any optimizations applied. Therefore, the hybrid cache architecture is a good alternative compared to the pure STT-RAM cache since the write-intensive blocks can be placed in SRAM instead of STT-RAM. This can be achieved through either static or dynamic schemes to optimize the block placement to reduce the average write frequency to STT-RAM cells.

In this dissertation we propose a combined approach to take the advantages from both static and dynamic schemes. For the initial cache block placement, the compiler provides hints of data placement to hardware to reduce the STT-RAM write frequency, while the hardware is designed to be able to correct compiler hints based on runtime cache capacity pressure. When initial placement does not work effec-

tively, some write-intensive cache blocks are still placed in STT-RAM. Once these write-intensive blocks are detected, dynamic migration starts to migrate these blocks from the STT-RAM array to the SRAM array. Experimental results show that the combined scheme improves endurance by 23.9x and 5.9x compared to pure static and pure dynamic optimizations, respectively, while maintaining similar performance. Furthermore, the system energy can be reduced by 17% compared to pure dynamic optimization since STT-RAM writes are reduced through initial placement from the proposed compiler flow.

1.3 Memory System Optimization for Accelerator-Rich Architectures

1.3.1 ARAPrototyper

Compared to conventional general-purpose processors, ARAs can provide orders-of-magnitude performance and energy gains [79, 31, 30] and are emerging as one of the most promising solutions in the age of dark silicon [67, 160]. However, many design issues related to the complex interaction between general-purpose cores, accelerators, customized on-chip interconnects, and memory systems remain unclear and difficult to evaluate. Therefore, a research platform that can enable rapid evaluation of such design spaces will be extremely useful.

In this dissertation we design and implement the ARAPrototyper to enable rapid design space explorations for ARAs in real silicons and reduce the tedious prototyping efforts far down to manageable efforts. First, ARAPrototyper provides a reusable baseline prototype with a highly customizable memory system, including intercon-

nect between accelerators and buffers, interconnect between buffers and last-level cache (LLC) or DRAM, coherency choice at LLC or DRAM, and address translation support. To provide more insights into performance analysis, ARAPrototyper adds several performance counters on the accelerator side and leverages existing performance counters on the CPU side. Second, ARAPrototyper provides a clean interface to quickly integrate a user's own accelerators written in high-level synthesis (HLS) code. The whole design flow is highly automated to generate a prototype of ARA on an FPGA system-on-chip (SoC) on Xilinx Zynq platform [89]. To quickly develop applications that run seamlessly on the ARA prototype, ARAPrototyper provides a system software stack, abstracts the accelerators as software libraries, and provides APIs for software developers. Our experimental results demonstrate that ARAPrototyper enables a wide range of design space explorations for ARAs at manageable prototyping efforts and 4,000 to 10,000X faster evaluation time than full-system simulations [52]. We believe that ARAPrototyper can be an attractive alternative for ARA design and evaluation.

1.3.2 Interconnect Synthesis for ARA Memory System

An ARA is composed of heterogeneous accelerators with an on-chip memory system [54, 41]. Compared to the general-purpose processors, an accelerator demands short and predictable latency to its local on-chip memory to satisfy its performance target. Moreover, an accelerator requires a much higher off-chip memory bandwidth than a CPU since it consumes much more data in a given time period. Therefore, a customized on-chip memory system design is one of the keys to an efficient ARA. In this dissertation we provide a two-layer interconnect synthesis method [33]. We first provide an optimal layer of partial crossbar that connects the heterogeneous

accelerators and shared memory banks with a minimum number of switches. The second layer of interconnect tries to interleave possible conflicting long-burst memory requests for prefetching data from off-chip memory. The experimental results show that we can reduce more than 45% of the switches of the partial crossbar compared to the best known method [60]. This further leads to a 53% reduction of LUTs and 34% reduction of slice utilization on a 30-accelerator FPGA prototype. The reduction of switches leads to a 16% - 53% reduction on LUTs and a 24% - 38% reduction in slices on FPGA prototypes. Furthermore, the performance of an ARA can be improved by 36% - 52% with a well-designed interleaved network in a real ARA prototype for medical imaging applications. This prototype also shows a 7.44x energy efficiency gain over the state-of-the-art Xeon processor [104].

1.4 Datacenter-Level Optimizations in an In-Memory Cluster Computing Framework

In the era of "Big Data," the amount of data grows exponentially and requires scalable technologies to process and store the data in a reliable way. The domain of genomics applications that we focus on is inherently a big data domain. For example, in the DNA sequencing pipeline, the raw data of an individual obtained from a sequencer can be around 300GB to 500GB before processing. We need the capability to provide scalable performance according to the performance target based on the demand of target biological genomics pipelines, such as variant calling [129] or differential gene expression [157]. Many research and clinical applications, such as precision medicines for cancers, are developed on top of these pipelines. The problem becomes more difficult when we have many cases to be analyzed. The number of individuals can be

from 100 to above one millon. At this problem scale, the data cannot even be stored in a single node.

In recent years, big data analysis and storage systems are burgeoning in the datacenters to handle large-scale problems. Google's MapReduce [62] and Google File System (GFS) [73] provide a simple programming model and a storage layer for developers to store data and distribute computations over a cluster with more than thousands of nodes. These infrastructures provide a scalable and efficient solution for applications with a huge amount data. However, Google's MapReduce and GFS are not open-source software and are used only internally in Google. Hadoop [163], with both the MapReduce framework and Hadoop Distributed File System (HDFS) [151] implemented in Java, is the most widely used open-source solution for big data analysis.

However, Google's MapReduce and Hadoop have a limitation on iterative algorithms which reuse a working set of data across multiple iterations. Machine learning algorithms such as K-means and logistic regression are the key examples in the category. In MapReduce and Hadoop, the reused data set and the output after a single iteration need to be written back to a distributed file system, which incurs substantial overheads on network bandwidth, disk I/O, and data serialization. Because of the reliability concern, the distributed file system maintains several copies of data, which makes the problem even worse. In order to solve the problem, Spark [172, 171], an in-memory cluster computing framework, is proposed to target the applications with reused data. The key idea is try to keep the input data and intermediate output results across iterations in memory instead of writing them back to the distributed file system.

The domain of DNA sequencing applications is one of the domains we are in-

terested in. With a significant amount of data-level parallelism, applications in the DNA sequencing pipeline can utilize in-memory cluster computing infrastructures such as Spark to fully exploit data-level parallelism. Table 1.3 shows motivational results of the runtime comparison of the state-of-the-art DNA sequencing pipeline and our proposed pipeline using the scale-out approach. Table 1.2 shows the settings of the state-of-the-art pipeline and our proposed pipeline. With a 30-node cluster, our target is to shrink the overall processing time from 8.85 days to be within one day (8 - 24 hours). Based on the current results, we demonstrated a superlinear 39.26x speedup over the state-of-the-art pipeline in the first three steps, including alignment, sort, and markduplicate. Note that ADAM [127] is a new framework built on Spark to provide cluster-scale computation for some applications in the DNA sequencing pipeline.

Table 1.2: A motivational example: the settings of the data pre-processing pipelines

	State-of-the-art pipeline [129]	Our proposed pipeline
Tools	BWA-MEM [111], SAMtools [114], Picard [7], and GATK [129]	CS-BWAMEM [39], ADAM [127] and our scale-out tools
# of Nodes	1	30
Data Size (FASTQ)	306.7GB	306.7GB

1.4.1 Datacenter-Level Optimizations — A Case Study of CS-BWAMEM

A next-generation sequencer can generate billions of small fragments (reads) of length in the range of a few hundred nucleotides in one run. Processing such a tremendous number of reads introduces significant computational challenges. For most of the

Table 1.3: A motivational example: runtime comparison of the data pre-processing pipelines

	State-of-the-art pipeline	Our proposed pipeline
Alignment	12.93 hrs	48 minutes
Sort	11.33 hrs	27 minutes (ADAM)
MarkDuplicate	103.33 hrs	2 - 10 hrs
IndelRealignment	12.50 hrs	on-going
BaseRecalibration	53.70 hrs	on-going
Extra Steps Between Stages	18.62 hrs	included
Total Runtime	**212.41 hrs (1.26 week)**	**Projected: 8 - 24 hours**

widely used sequencing pipelines, such as variant discovery [129] and differential gene expression [157], the first step would align these short reads back to a long human reference genome. After this alignment step, the aligned short reads can contain the coordinate information based on the reference genome. Usually, the alignment step is one of the most time-consuming steps due to its compute-intensive nature to align these reads back to the long reference genome.

The major limitation of the current state-of-the-art aligners, such as BWA-MEM [111] and Bowtie2 [105] is that their performance is limited by the computation power of a single node. Performing a high-coverage whole-genome sequencing can take tens of hours. In this dissertation we propose CS-BWAMEM, which is a scalable solution for latency-critical sequencing. For a 30x coverage for whole genome data, CS-BWAMEM can align it within 32 minutes by using a 30-node cluster, which is 18x faster than BWA-MEM.

CS-BWAMEM is built on top of the in-memory cluster computing framework,

Spark, [172, 171] and thus can meet different performance targets. We used CS-BWAMEM as an example to demonstrate our proposed customized optimizations at the datacenter level. These strategies can also be applied to the applications in the DNA sequencing pipeline, but may not be limited to the DNA domain. First, the large reference genome needs to be broadcast to each node in the cluster in CS-BWAMEM. However, it can introduce significant performance overhead when the size of the cluster is large. We propose a mechanism to bypass the original Spark broadcast. Instead, we load the reference genome from the local disk of each node to avoid the broadcast overhead. Second, the data is stored in a column store for better compression and partial data access. Third, to utilize accelerators or co-processors efficiently, we group small tasks in batches, and the accelerators can compute tasks in a batched way to avoid the overhead introduced by fine-grained data transfer. Finally, we introduce multiple strategies to reduce the significant I/O overhead through in-memory caching and pipelining the computation with I/O.

CS-BWAMEM demonstrates a motivational example showing that a similar methodology can also be applied to the other applications in the DNA sequencing pipeline.

1.4.2 Support and Optimizations of an FPGA-Enabled In-Memory Cluster

The scale-out approach mentioned in Section 1.4.1 can accelerate the target large-scale applications, such as CS-BWAMEM. However, in order to further improve the system performance and energy, we need to deploy energy-efficient architectures, such as ARAs mentioned in Section 1.3, in an in-memory cluster. Instead of realize ARAs from manufacturing ASICs, we select FPGA to implement customized accelerators for real-life deployment.

We use PCIe-based FPGA cards in our in-memory cluster. We aim to accelerate the most compute-intensive dynamic programming algorithm [111], a modified version of the original Smith Waterman (S-W) algorithm [152, 76], used in CS-BWAMEM. We use the array-based architecture developed in [35]. However, it is still unclear how PCIe-based FPGA accelerators can be efficiently used in a Spark cluster. In this dissertation we study both (1) algorithm-level improvement that adapts to FPGA accelerators, and (2) runtime system support of using FPGA accelerators.

The contributions can be summarized as follows.

- **Batched S-W algorithm for reducing communication overheads:** We realize the high-throughput S-W accelerator proposed in [35] in a PCIe-based FPGA card using Xilinx SDAccel flow [8], and deploy the accelerators over the cluster. We observe that a significant communication overhead occurs for the data transfer between (1) map tasks and the accelerator manager, and between (2) the accelerator manager and FPGA board. To overcome the communication overheads, we try to group a large number of reads and send them to the FPGA accelerator. However, due to the strong inner-task dependency of the S-W algorithm used in BWA-MEM, we cannot directly send data to FPGA without resolving the dependency. By redesigning the S-W algorithm, we are able to process data in a batched fashion and thus better utilize the FPGA accelerator with much less communication overhead.
- **Accelerator Manager:** In the SDAccel flow, we need a software accelerator manager (AM) to manage the accelerator requests from multiple Spark tasks in a node. The AM is in charge of receiving the requests from multiple processes, dispatching the requests, and sending input data to the FPGA accelerator. It is challenging to design the AM efficiently, since multiple tasks can issue requests

simultaneously. We design the handshaking protocol and use the POSIX shared memory to exchange data between the AM and the Spark tasks.

1.5 Application Domains

1.5.1 Domain 1: Medical Imaging Pipeline

Computerized tomography (CT) plays a major role in modern medicine. The medical imaging pipeline consists of a set of applications which are used to process the image produced by a CT scanner [25], as shown in Figure 1.2. The applications include image reconstruction, de-noise, de-blur, image registration, and image segmentation. Image reconstruction is the most time-consuming application; it uses the compressive sensing algorithm to reduce the required sample size for less radiation to patients.

After the reconstructed image is obtained, the next step in the pipeline is to remove the noise of an image. This step consists of two applications: (1) *denoise* and (2) *deblur*. *denoise* tries to remove the noise under Rician distribution while *deblur* tries to remove the blur. Next, image registration is performed to better align the two image studies and capture the progressive development of the illness (e.g., tumors). Fluid registration regularizes the deformation using a fluid PDE equation, and it allows registrations of large deformations. Finally, image segmentation tries to find and segment an object of interest for the analysis step in the pipeline. These medical imaging applications consists of a huge amount of stencil computations, and thus are data-intensive.

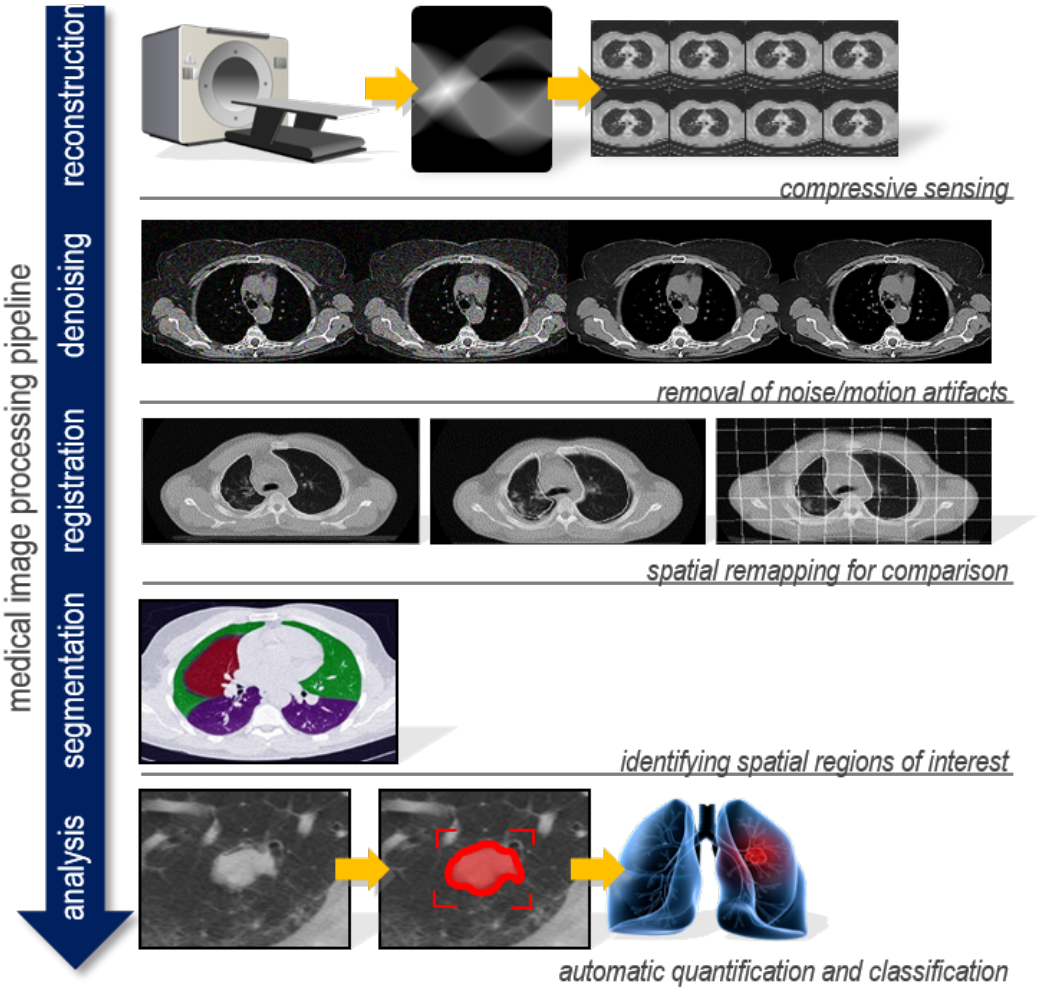


Figure 1.2: Medical imaging pipeline

1.5.2 Domain 2: DNA Sequencing Pipeline

The advent of next-generation sequencing technologies dramatically reduces the cost of genome sequencing. Today's sequencing technologies can obtain a genome for an individual for $1,000 or less. The technology can be widely used in research and is transitioning into clinics for applications such as precision medicine for cancer

treatment. Next-generation sequencers will be more cost-effective by generating the sequence from very small fragments (reads) of length in the range of a few hundred nucleotides. Combining these sequenced fragments into a genome sequence by taking advantage of the known human genome sequence is called resequencing and entails tremendous computational challenges.

Figure 1.3 shows the workflow of the DNA sequencing suggested in the Genome Analysis Toolkit (GATK) from the Broad Institute. In data preprocessing, the raw reads generated from sequencers are processed to be analysis-ready reads. The raw data of each individual is around 300GB – 500GB. In variant discovery, the analysis-ready reads are used as input to perform analysis-read variants. The amount of data can vary from 10TB – 1PB, which is far beyond the capability of a single server. The amount of data depends on the number of individuals that are involved in joint genotyping. The current GATK tool takes around 212 hours (about a week) to complete data preprocessing in a multicore server with the multithreaded software, while variant discovery may take several weeks. Our goal is to accelerate both data preprocessing and variant discovery to reduce the processing time to within one day and one week, respectively.

In data preprocessing, the raw reads from sequencers are first sent to an aligner to perform initial mapping. The goal is to align these short reads to possible locations of the reference genome. After alignment, a short read can be mapped to none, one, or many locations on the reference genome. Burrows-Wheeler Aligner (BWA) [112, 113, 111] and Bowtie 2 [106, 105] are the state-of-the-art aligners on general-purpose processors. In this dissertation we adopt the BWA [111] in our sequencing pipeline. The aligned reads then require sorting, and the duplicated reads need to be marked before we can further refine the quality of results. In this pipeline we use

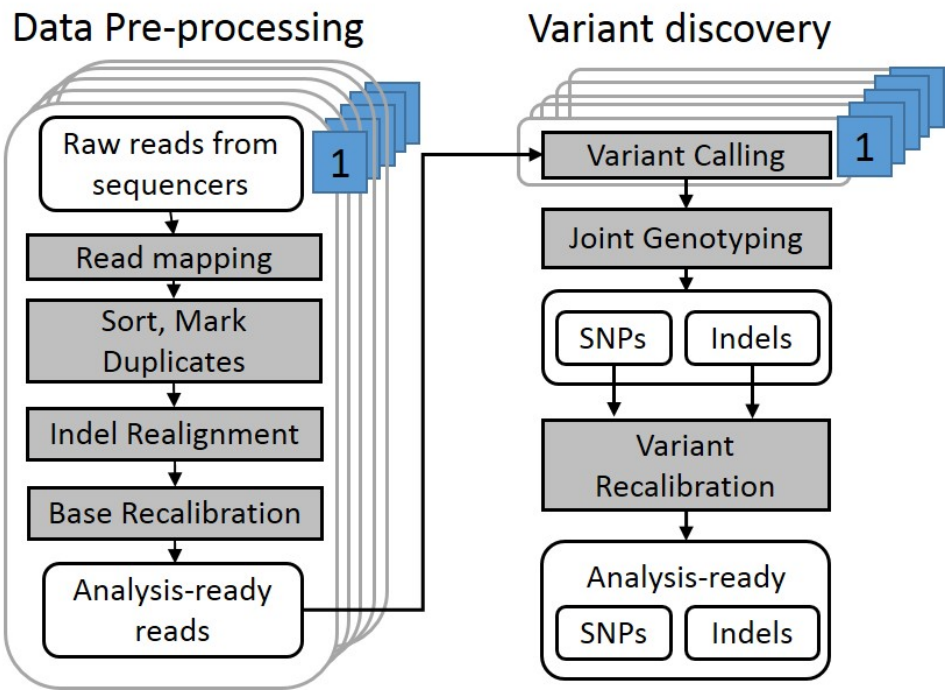


Figure 1.3: DNA sequencing pipeline

SAMtools [114] and Picard [7] to achieve sorting and mark duplication, respectively. After *sort* and *markduplicate*, *indelrealignment* is performed to locally realign reads such that the number of mismatching bases is minimized across all the reads. Due to the presence of an insertion or deletion from the individual's genome with respect to the reference genome, a large percent of intervals require local alignment. There are two main steps in *indelrealignment*. First, the suspicious intervals in need of realignment needs to be determined. Second, we need to call the local realigner on those intervals. The final step is base recalibration. *baserecalibration* can recalibrate base quality scores of the aligned reads. After recalibration, the quality score of each read is more accurate. The computation kernel of the *baserecalibration* is a hidden Markov model (HMM). For *indelrealignment* and *baserecalibration*, we use the Genome Analysis Toolkit (GATK) [129] from the Broad Institute. The applications

in data preprocessing inherit great data-level parallelism. Therefore, performance can be significantly improved by exploiting the data-level parallelism in the cluster level, which is way beyond a single node, as discussed in Section 1.4.

In variant discovery, the goal is to obtain variants relative to the reference genome. First, the analysis-ready reads of an individual are used for variant calling. The goal of variant calling is to try to find single nucleotide polymorphism (SNP). SNP is a single nucleotide variation, which is different from the reference genome, of an individual. *variantcalling* is performed on analysis-ready reads of every individual to get genotype likelihoods. After that, joint genotyping is performed to obtain SNPs and Indels (insertions and deletions) of the group of individuals. Finally, the variant recalibration step performs customized filtering according to the needs of each project.

CHAPTER 2

HC-Sim: A Fast Cache and SPM Co-Simulation Framework

2.1 Introduction

The speed gap between the processor and main memory has been increasing constantly. Caches are used as intermediate storage to mitigate the gap. The L1 cache, which is accessed most frequently by the processor, has a significant impact on the performance and energy consumption of a system. Increasing the cache size can improve locality, leading to a higher cache hit rate and possible performance improvement. However, a large cache will increase the access time and the energy of a cache access. The cache size determination depends on the requirements of system performance and the constraint on energy consumption.

An embedded system is designed for a specific application or a domain of applications. Therefore, the performance and energy of the system can be optimized by providing a customized cache hierarchy. One important issue is to find the optimal configuration for the L1 cache. A cache configuration can be defined by three parameters: the number of cache sets, associativity, and cache line size. Previous work has shown that a correct L1 cache configuration can significantly improve performance

and reduce energy consumption [12, 24, 70, 93]. To find the most energy/performance-efficient configuration, the L1 cache miss rate should be evaluated for each configuration. The distribution of L1 cache miss rates defines the working set of the application and is an important metric used for the performance and energy estimation [12, 24, 93]. It is also very useful for energy optimization of computer systems with heterogeneous [94] or customizable cache designs [12, 141, 43].

To obtain the miss rates of different cache configurations on a target application, two classes of approaches are proposed: simulation-based techniques [81, 153, 66, 93, 156, 80] and analytical approaches [70, 136, 21, 74, 75, 161]. The simulation-based techniques perform exact simulation on every cache access in a given program trace. The hit or miss status of every access is simulated exactly. However, the simulation-based techniques suffer from long simulation time since each configuration should be simulated for the target application. The analytical approaches provide analytical models for cache miss rate estimation. Generally, the analytical approaches are fast but inaccurate.

This chapter focuses on improving the performance of simulation-based techniques. Two reasons lead to the long simulation time of exact techniques. First, exhaustive simulation on each cache configuration is required. In [128, 81], the researchers discovered and applied the "inclusion property" to simulate multiple cache configurations simultaneously. This property specifies that when a cache hit occurs in a cache, then cache hits occur in all larger caches. For example, if a hit occurs in a fully associative cache with two cache blocks, then hits will definitely occur in a fully associative cache with more than two blocks. Therefore, multiple configurations can be simulated in one run, and this reduces the number of runs to simulate all configurations. Second, based on the inclusion property, the hit/miss status of a

cache access of multiple cache configurations can be simulated by using a stack [128]. For each cache access, a linear search is required to check if the tag of the access is still in the stack. If not, cache misses occur. The process of the linear search may be time-consuming since it depends on the total stack distance traversed by all accesses. In [156, 80], the authors developed efficient techniques to reduce the search space and thus improve simulation performance.

The scalability of the cache simulator is another important issue. The implementations of previous simulation-based techniques rely on a given trace file as an input [66, 93, 156, 80]. However, if the target application has a long execution time, the generated trace will be huge. The storage of memory trace consumes tremendous disk space. For example, the trace file of a medical imaging application, *riciandenoise*, has 3.5 billion data cache accesses, requiring 42.3GB to store the trace. For some SPEC2006 benchmarks [88], such as $h264ref$, the trace file is estimated to be 1.1TB.

For applications with predictable access patterns, scratchpad memory (SPM) can used to achieve better performance and energy consumption of an embedded system [43, 19, 97, 49]. The predictable patterns can be identified by programmers or compilers to provide optimized codes. Compared to a cache, SPM does not need to perform an expensive associative way driving and tag comparisons, and hence is more energy efficient. The recent NVIDIA Fermi GPU [9] also provides a mechanism to configure the ratio between L1 cache and SPM. However, currently no tool can efficiently simulate the hybrid memory system with L1 cache and SPM. The co-simulation is achieved by integrating SPM into the memory system in a full system simulator, which suffers from long simulation time [49].

In this chapter we propose *HC-Sim* (Hashing-based Cache Simulator), which efficiently simulates multiple L1 cache configurations simultaneously and supports fast

co-simulation for L1 caches and SPM. The contributions are summarized as follows.

(1) We propose novel data structures and algorithms, which incorporate a centralized hash table and a novel miss counter structure to avoid time-consuming tag searches on the stacks. Compared to SuSeSim [80] and CRCB algorithms [156], HC-Sim results can be up to 5.21X and 13.73X faster, respectively. On average, HC-Sim is 2.56X and 5.44X faster than the SuSeSim and CRCB algorithms, respectively. Furthermore, we show that the efficiency of HC-Sim can be improved by using a miss-counter-based structure for a group of applications.

(2) To enhance scalability of HC-Sim, we build our simulation framework on the dynamic binary instrumentation tool, Pin [121]. The trace generation is embedded into HC-Sim and is performed on-the-fly to avoid the overhead of a huge trace file.

(3) To observe the interaction between a L1 cache and SPM, such as the miss rates and the distribution of memory accesses on L1 caches and SPM, fast cache and SPM co-simulation are needed. This is non-trivial since the implementation of HC-Sim is based on Pin, and only instruction-level information can be obtained. Here, we provide an interface for designers to specify the address range of SPM so that the SPM accesses can be bypassed, and the correct miss rates of L1 caches can be simulated. We also provide a mechanism to filter out prefetching loads, which arise from SPM prefetching, to maintain the correctness of L1 cache simulation.

2.2 Background

In this section we first define the terminology that is used to describe a cache configuration and a memory access. Next, the inclusion property is reviewed. After that, we describe the stack simulation and the forest simulation used to simulate associative caches [128] and direct-mapped caches [81] respectively. Finally, we discuss the data structure extended to simulate set-associative caches [93] and the corresponding enhancements [156, 80].

2.2.1 Terminology

A cache configuration is determined by three parameters: *the number of cache sets* (s), *associativity* (a), and *cache line size* (b). The cache size is the multiple of the three parameters ($s \times a \times b$). The largest settings of the number of cache sets, associativity, and cache line size are denoted as S, A, and B respectively. We assume that the three parameters can only be set to powers of two. Hence, to find the optimal configuration, $(log_2(S) + 1) \times (log_2(A) + 1) \times (log_2(B) + 1)$ configurations should be simulated.

For a memory reference, the address can be divided into three fields: *tag*, *index*, and *offset*, as shown in Figure 2.1. The index is used to indicate in which cache set the datum is stored. We can use $log_2(s)$ bits for the index. In a set-associative cache, a datum can be stored in any cache way of the cache set. Therefore, tag comparison is performed to find the datum. Typically, the cache line size is larger than the size of one datum. The offset is used to retrieve the datum from the cache line. $log_2(b)$ bits are required for the offset. Hence, in a 64-bit system, the tag field contains (64 - $log_2(s)$ - $log_2(b)$) bits. The *block address* contains the fields of the tag and the index of a reference.

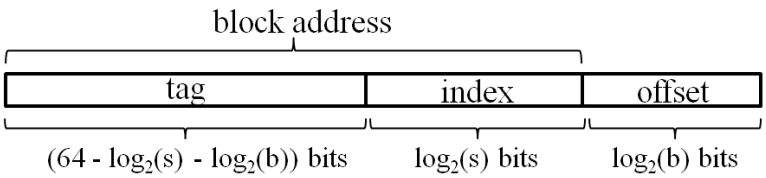


Figure 2.1: The address fields of a 64-bit memory reference

2.2.2 Inclusion Property

The inclusion property can be used to achieve fast simulation in multiple set-associative caches with LRU replacement policy [128, 81, 93, 156, 80]. The inclusion property holds when the cache line sizes of these set-associative caches are the same, and no prefetching is performed. We summarize the inclusion property into the following two cases. First, for caches with the same number of sets (s) and different associativity (a), whenever a hit occurs in a cache, hits are then guaranteed in all caches with larger associativity [128, 81]. For example, if a hit occurs in a cache where $s = 2$ and $a = 1$, it is then guaranteed that the same hit occurs in the caches with $s = 2$ and $a > 1$. This can be used to simulate associative caches simultaneously, as described in Section 2.2.3. Second, for caches with the same associativity and a different number of sets, if a hit occurs in a cache, then all caches with a larger number of sets guarantee the same hit [81, 93]. For example, if a hit occurs in a cache with $s = 2$ and $a = 1$, it is then guaranteed that hits occur in the caches with $s > 2$ and $a = 1$. Based on the property, the forest simulation is proposed to simulate direct-mapped caches [81] and set-associative caches [93], as reviewed in Section 2.2.4 and Section 2.2.5.

2.2.3 Stack Simulation for Associative Caches

By using the inclusion property, the authors in [128] showed that a stack can be used to model multiple associative caches/memories. Figure 2.2(a) shows the linked-list structure used to simulate the stack behavior. A linked-list with four nodes can be used to simulate 1-way to 4-way associative caches simultaneously. Note that we consider that all associative caches are in the same cache set. Each node in the list represents a cache way and stores the tag of a cache access. The most recently accessed cache access is stored in the head, while the second-most recently accessed is stored on the second node, and so on. Only the four most recently accessed ones can be stored in the stack. Here, we define the *stack distance* of a node to be the distance between the head and the node. To find a node in the list by linear search, the stack distance is the number of nodes that must be traversed.

Here, we illustrate the process of stack simulation with an example. Figure 2.2(b) shows the situation after a cache access sequence with addresses {0110, 1001, 1010, 1000} is performed. The initial tags stored in the linked-list are {1000, 1001, 0010, 1100}. Here, we assume the cache line size is one byte and the number of sets is one. When the first access '0110' comes in, a linear search is performed on the linked-list. Since no matched tag is found, misses occur at all associative caches. For the second access '1001,' a mapped tag is found in the third node. Based on the inclusion property, hits occur at the 3-way and 4-way associative caches while misses occur at 1-way and 2-way associative caches. After that, '1010' and '1000' are processed, and their corresponding status is shown in Figure 2.2(b). The total number of misses for 1-way to 4-way caches is four, four, three and two, respectively.

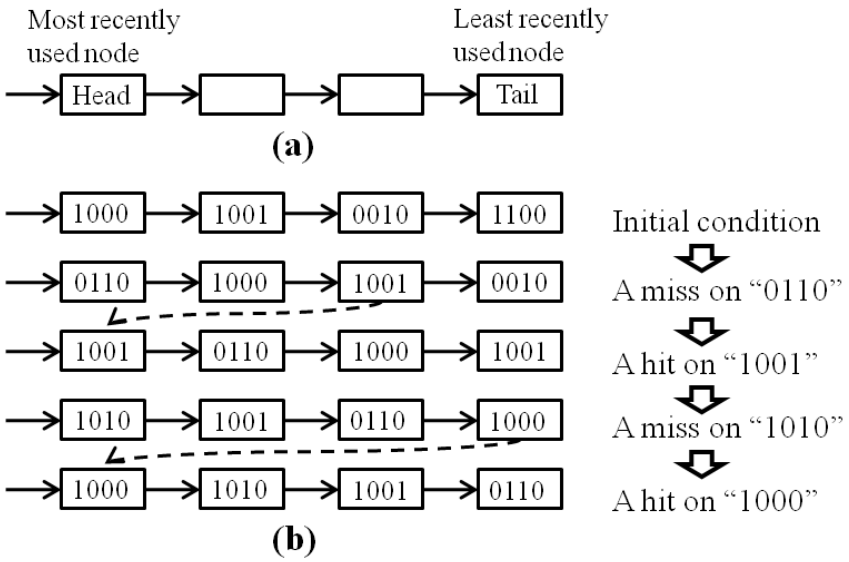


Figure 2.2: LRU stack simulation

2.2.4 Forest Simulation for Direct-Mapped Caches

Initially, forest simulation was proposed to simulate direct-mapped caches [81]. As mentioned in Section 2.2.1, we use $log_2(s)$ bits to encode the index field of a memory reference. We can use 1-bit to encode a direct-mapped cache with two sets, which are encoded by '0' and '1.' For a cache with four sets, we can use '00,' '01,' '10,' and '11' to encode them. It is natural to represent a group of direct-mapped caches of different sizes in a binary tree, as shown in Figure 2.3(a). The root of the tree represents a direct-mapped cache with one cache set, while two nodes in the second level represent a cache with two sets and so on. In Figure 2.3, a group of direct-mapped caches with one, two, four, and eight sets represented in a four-level binary tree.

Figure 2.3(a) also shows an intermediate state of the tree. Each node has an entry to store a tag. We assume the cache line size to be one byte. In forest simulation, the tree traversal starts at the root and proceeds to the leaves. However, only one node

in each tree level will be traversed since an access can only be mapped to one cache set. First, suppose the next address accessed is '1001,' where a hit occurs at the root. Based on the inclusion property, we do not need to traverse the tree any more since hits are guaranteed in all larger caches with two, four, and eight sets. Therefore, tag updates are not required. After that, the next access is '1100,' a miss occurs in the root and further traversal of the tree should be performed until either the tag is found or the leaf of the tree is arrived at. Since a matched tag is not found until the node '100,' we can conclude misses occur in direct-mapped caches with one, two, and four sets. The updated state of the tree is shown in Figure 2.3(b).

2.2.5 Forest Simulation for Set-Associative Caches

To simulate set-associative caches in one run, the researchers in [93] proposed a data structure based on the forest simulation framework described in Section 2.2.4. In [93], the tag entry is replaced by a linked-list described in Section 2.2.3. The linked-list is used to simulate a group of associative caches. Figure 2.4 shows the data structure used to simulate the caches where $0 \leq s \leq 8$ and $0 \leq a \leq 4$. If we want to simulate set-associative caches with multiple cache line sizes, we can replicate the structure for multiple copies.

The authors proposed an algorithm based upon this data structure to obtain miss rates for caches [93]. The algorithm can be briefly explained as follows. For each access in the trace, we traverse the tree from the root to the leaves in a top-down scheme. Note that only one of the nodes in each tree level is traversed because a cache access can only be mapped to one cache set in a set-associative cache. For example, the '1001' reference will be mapped to the tree node with set number '01' in the third tree level. After that, a linear search is performed on the linked-list pointed to by the

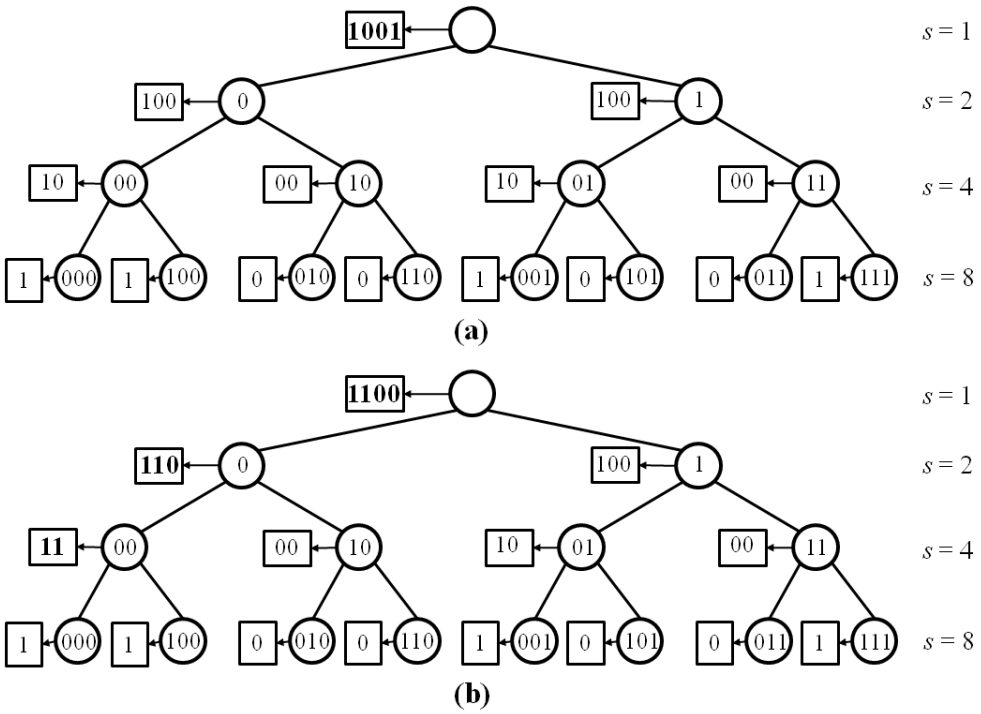


Figure 2.3: Forest simulation for direct-mapped caches

cache set '01.' If a matched tag is found at the n-th node of the linked-list, hits occur on caches with associativity larger than or equal to n. Finally, we need to update the order of the linked-list to maintain the LRU property. Note that the cache miss counters need to be updated if misses occur on corresponding cache configurations.

Assume that there are N accesses in the trace. For each access, it takes at most $O(log_2(S))$ time, which is equal to the height of the tree, to perform the tree traversal. The time complexity to perform linear search in a linked-list is $O(A)$ in the worst case if no matched tag is found. Note that we assume that the associativity can only be powers of two. Therefore, $log_2(A)$ counters are required for each linked-list. To update the miss counters, it takes $O(log_2(A))$ for each linked-list. The time complexity is $O(N + N(log_2(S))(log_2(A)) + N(log_2(S))A)$, which is equal to $O(N(log_2(S))A)$ [93].

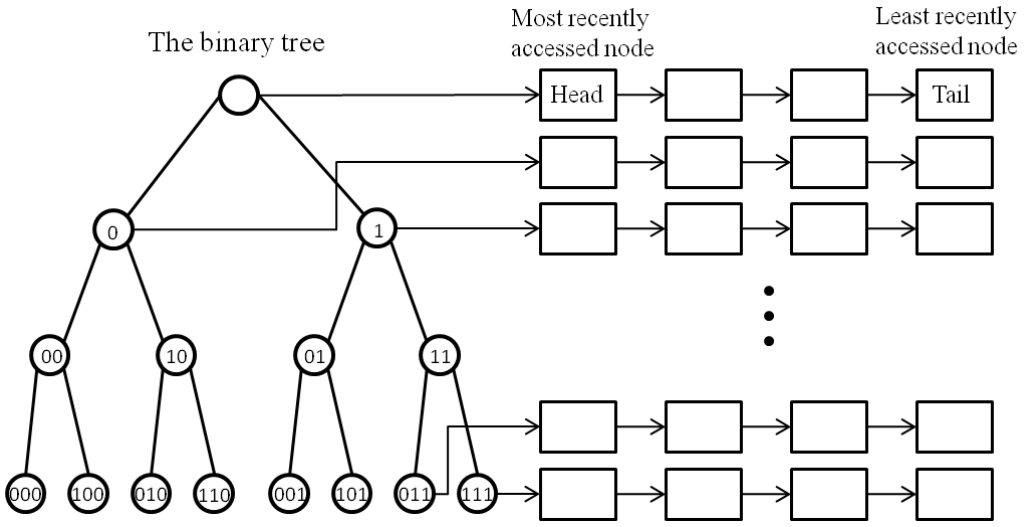


Figure 2.4: Forest simulation for set-associative caches

2.2.6 Strategies for Speedup

Based the framework described in Section 2.2.5, the researchers presented strategies utilizing the inclusion property to speed up the simulation [156, 80]. In [156], the researchers provided two enhancements which reduced the total number of nodes to be traversed in the linked-lists. First, if a hit occurs in the head of the linked-list with $s = i$, then we do not need to perform simulations for caches with $s > i$ since hits occur at all configurations. Second, if the block addresses of two consecutive cache accesses are the same, hits occur for all cache configurations.

In [80], the authors had two other observations. First, the inclusion property can also be interpreted in another way. If a miss occurs in caches with $s = i$ for all associativity, then misses will occur in all smaller caches with $s < i$ for all associativity. A bottom-up tree traversal scheme can be developed according to the observation. This removes a large amount of unnecessary tag searches on the linked-lists, and thus

reduces the simulation time. Second, we assume the largest setting of associativity is A. If a matched tag is found in X-th way where $X > (A/2)$, then cache hits can only occur after X-th way for caches with a smaller number of cache sets. Therefore, by performing linear search from the tail to the head, more unnecessary searches can be removed.

2.3 HC-Sim Design

2.3.1 Motivation

We observe that the most important factor that leads to the long simulation time is the linear search performed on the linked-lists. If we can improve the efficiency of searching, the simulation time can be significantly reduced. However, even if one can replace the linked-lists with alternative data structures to improve searching efficiency, it is still costly to update the linear order for every cache access. The linear order is naturally maintained by the LRU stacks, i.e. the linked-lists. Therefore, we focus on improving the searching efficiency of the linked-lists. A hash table is one of the data structures that can be used to improve the searching efficiency. In [101], the researchers provided a data structure using a hash table to simulate a memory trace. Figure 2.5 shows the data structure. A hash table is used to find the corresponding memory reference in the stack. The key in the hash table is the address of the memory reference, while the value of the hash table is a pointer used to point to the corresponding reference in the stack.

With the aid of the hash table, a memory reference can be found in constant time. However, to obtain the miss rates of different memory configurations, the stack

distance of a node should be found in constant time as well. Then, we can use counters to record the hit information for a cache configuration. Figure 2.5(a) shows that the *marker* is added as a field in each node of the stack. Normally, a designer only considers memory configurations to be power-of-two sizes. Therefore, we assume that the stack sizes can only be powers of two. A marker represents the *relative* stack distance of a node instead of the distance from the top of the stack. For example, the value of the marker of the fifth element in the LRU stack is '3,' which means the node can be found in the stack only when the stack size is greater than or equal to 2^3. The node cannot be found when the stack size is less than or equal to 2^2. Whenever a new memory reference is processed, the marker fields are required to be updated. It can be naively done by updating the fields from the top of the stack to the node found by the hash table. However, this does not improve the simulation efficiency compared to linear search.

To improve the efficiency, the *marker pointers* are used to point to the boundaries of markers. For example, the marker pointer of size = 2^2 is pointed to the fifth node which follows the last node with marker '2,' as shown in Figure 2.5(a). When a cache access occurs, the marker pointers and the corresponding hit counter should be updated. Consider the address of the next access as '110111. Figure 2.5(b) shows the update of the data structures. First, since '110111' is found in the hash table, the node in the LRU stack can be found by the pointer. The node should be moved to the head of the stack, and the marker should be set to '0.' Second, the marker pointers of 2^0 and 2^1 should be moved upward by one node such that the marker pointers point to the correct places. The marker pointers are moved from '1010' and '0110' to '1011' and '1010,' respectively. Also, the markers of the node '1011' and '1010' should be set to '1' and '2,' respectively. Third, the hit counter of 2^2 should be incremented by 1.

Based on the inclusion property, the hit counter of 2^3 should be also incremented to record the hit. However, we can save the increment to improve simulation efficiency. The total number of hits of a specific cache size can be calculated by accumulation after all cache accesses are processed.

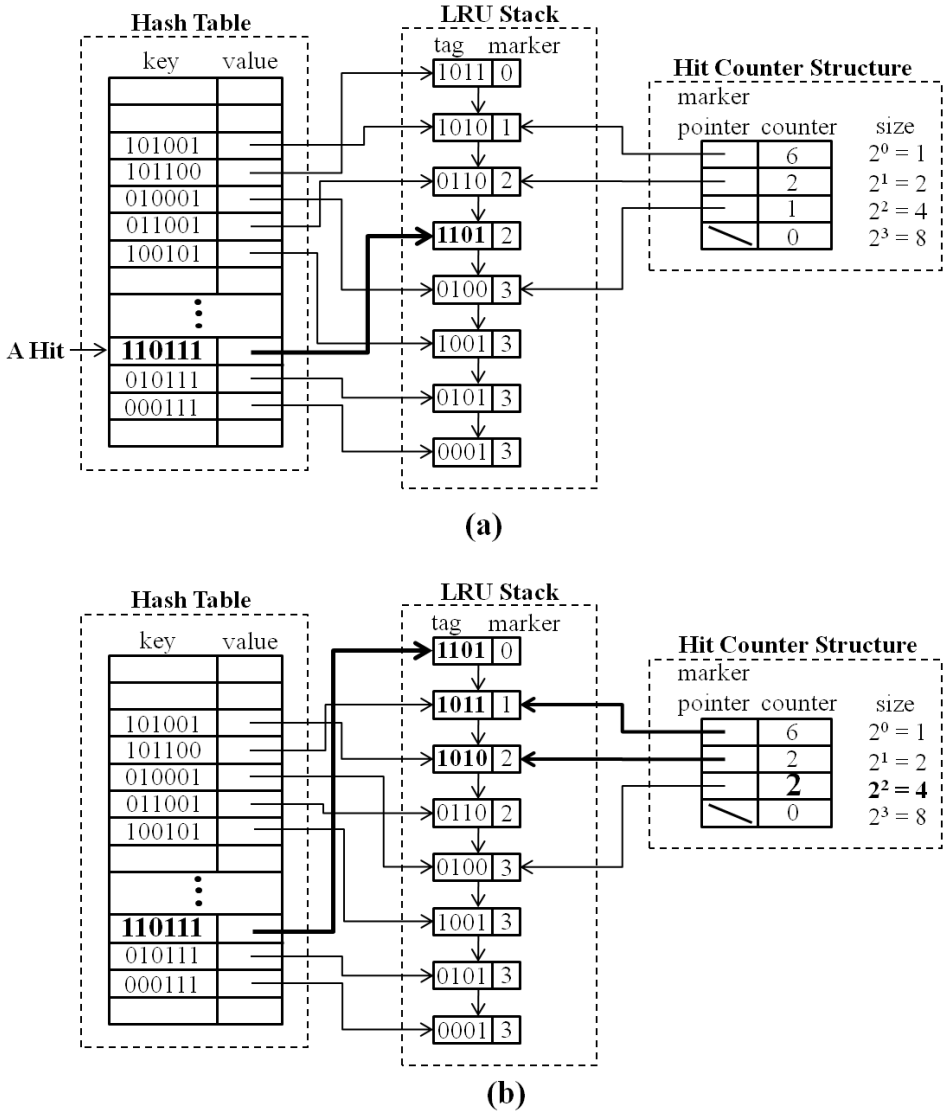


Figure 2.5: Hashing-based structure for stack simulation

If the maximum stack distance is A, the time complexity to update marker pointers is O($log_2(A)$). Compared to the time complexity of linear search, which is O(A), this method is more efficient. In addition, the hit counter takes O(1) to be updated, which is more efficient than the miss counters used in [93, 156, 80]. Therefore, this method has been used to simulate disk and file-system traces, which have long stack distances [101].

The hashing-base structure can be used to simulate associative caches efficiently. However, it is not clear how to extend the proposed data structure to simulate set-associative caches in one run. Simply replicating hash tables for all stacks is not feasible. Considering the forest simulation framework, if we want to simulate set-associative caches with up to 512 sets, there will be 1023 nodes in the forest. Each node requires a hash table, an LRU stack and a counter structure. However, the size of a hash table depends on the number of distinct cache accesses appearing in the hash table, which may be large in some applications. Therefore, it may be infeasible to realize the system due to the huge memory demand. We shall introduce an elaborate centralized hash table design, as described in Section 2.3.2.2, to handle the memory usage problem. Also, the extra data structure modifications are required and are discussed in Section 2.3.2.1.

2.3.2 HC-Sim Data Structures

2.3.2.1 Modifications of Data Structures

To simulate associative caches, the size of an LRU stack is the largest setting of associativity. For a cache access, if misses occur for all associative caches, the access will become the head of the linked-list. In the meantime, the original tail of the

linked-list should be removed from the linked-list since it is no longer in the LRU stack. Otherwise, the stack will grow continuously without the recycling mechanism. This issue is not mentioned in [101]. Therefore, we require a mechanism to invalidate the pointer in the hash table when the corresponding cache line is no longer available. Figure 2.6(a) shows a new field, called *reset pointer*, which is added to each node of the LRU stack for invalidation use. In this example, we assume the cache line size is four bytes, and the number of cache sets is one.

Moreover, we use block addresses as the keys of the hash table, as shown in Figure 2.6(b). It is possible that many references stored in the hash table point to the same node in the stack, as shown in Figure 2.6(a). In this example, '000110,' '000111,' and '000100' point to the same node in the stack since their tags and indexes are the same. The design of reset pointers becomes complicated since multiple entries in the hash table require invalidation when there is a miss. One feasible solution is to use a linked-list to model a set of reset pointers of a node in the LRU stack. However, this degrades the performance since O(B) times of invalidation are required. Also, extra memory overhead in O(B) is introduced. In a cache, the granularity of a hit or a miss is based on one cache line. Cache accesses in the same cache line share the same block address and thus the same hit/miss status. Therefore, it is natural to use the block address as the key. One important benefit is that the number of entries in the hash table can be greatly reduced, and thus the memory use is reduced. In Figure 2.6(b), two redundant accesses are removed from the hash table. Furthermore, the invalidation mechanism is simplified and can be realized by using one reset pointer per stack node.

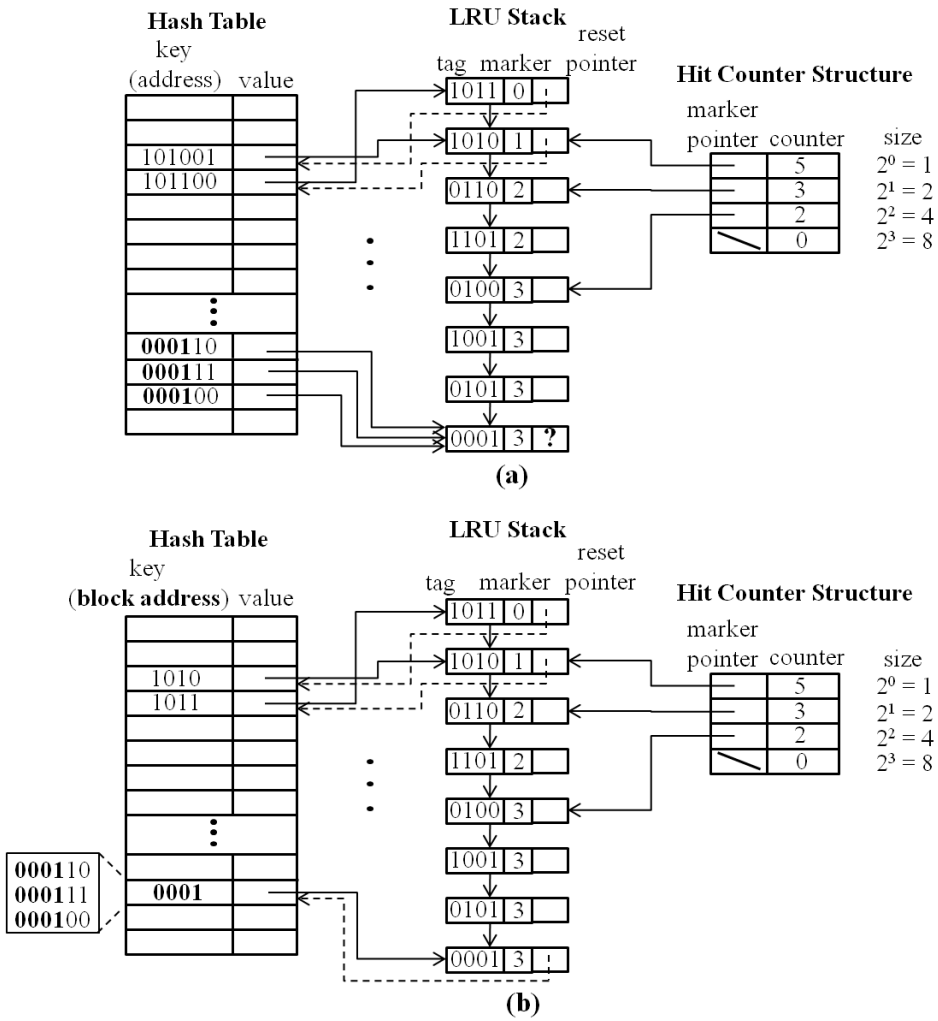


Figure 2.6: Modifications of data structures: (a) reset pointers and (b) use of block addresses

2.3.2.2 Centralized Hash Table Design

Instead of using independent hash tables for all stacks, which results in huge memory overhead, we adopt a centralized hash table structure. Figure 2.7 shows the data structures used by HC-Sim, which can be divided into three parts: the hash table,

the forest, and the counter structures. The key of each entry is the block address, as described in Section 2.3.2.1. For the value field, a *node pointer array* is used to store the *node pointers* that point to the corresponding nodes in the LRU stacks. Intuitively, it seems that the size of the node pointer array should be equal to the number of tree nodes in the forest. However, the array requires only $(log_2(S)) + 1$ elements, which is equal to the number of levels of the binary tree. This is because a cache line can only be mapped to one of the cache sets in any level of the binary tree. For example, the cache access '10011' is mapped to the sets with set number '1,' '11,' and '011' respectively. '1,' '11,' and '011' are the set numbers of three different levels in the binary tree. The data structures shown in Figure 2.7 can be used to simulate set-associative caches where $b = 1$, $0 \leq a \leq 4$, and $0 \leq s \leq 8$.

For the forest, we used the modified node with reset pointer to realize the invalidation of a hash-table entry. For each linked-list, a corresponding counter structure is provided for stack distance calculation, and thus the number of hits can be updated.

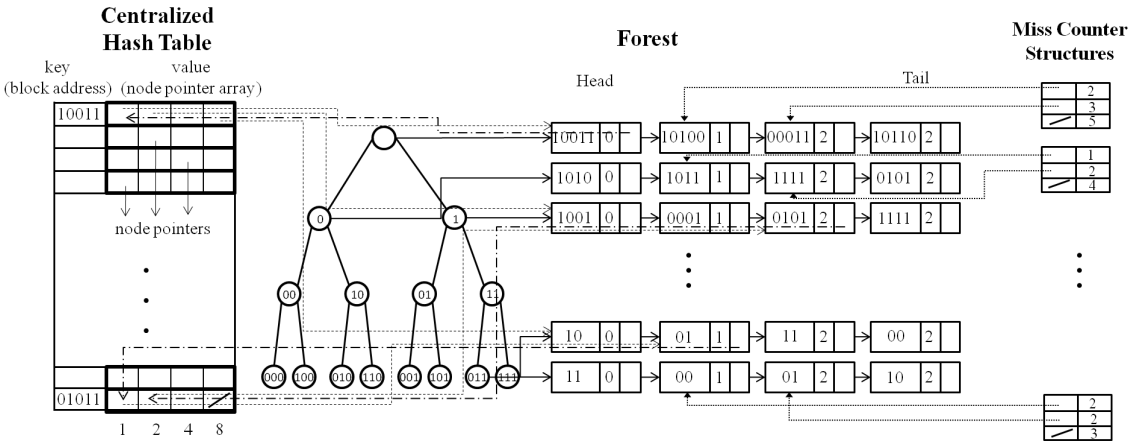


Figure 2.7: HC-Sim structure

The counter structures are used to record the number of hits or misses for miss

rates calculation. Figure 2.8 shows the process of updating the counters in three different counter structures. In this example, the access coming next, '0101101,' is found at the fourth node of the linked list. Therefore, hits occur when associativity equals 2^2, and 2^3 while misses occur when associativity equals 2^0 and 2^1. Figure 2.8(a) shows the hit counter structure used in [101]. As mentioned in Section 2.3.1, we only increment the counter of 2^2. The time complexity for the counter update is O(1).

In [93, 156, 80], authors used a centralized miss counter structure, as shown in Figure 2.8(b). Since they do not use the marker pointers to point to the boundaries of markers, the counter structure is a two- dimensional array that records the total number of misses of each cache configuration. For the counter updates, they simply increment the miss counters of all configurations with cache misses. The time complexity is O($log_2(A)$).

However, spatial locality exists in most applications. Normally, the desired miss rates for most L1 cache configurations are less than 10%. The total number of hits are an order larger than the the total number of misses. Therefore, the hit counter structure may or may not outperform the centralized miss counter structure. Based on this observation, we design a new miss counter structure, as shown in 2.8(c). In this example, only the miss counter of 2^1 is required to be incremented. Similarly, the miss rates can be calculated by accumulation. The miss counter structure inherits both advantages of the O(1) complexity and fewer increments on misses. We demonstrate the statistics of counter updates in Section 2.5.2.

For each cache access, at most O($log_2(S)$) LRU stacks are traversed in the binary tree. Therefore, the time complexity of counter updates for the hit counter structure and the miss counter structure is O($log_2(S)$) per access, while that for the centralized

miss counter structure is O($(log_2(S))(log_2(A)))$.

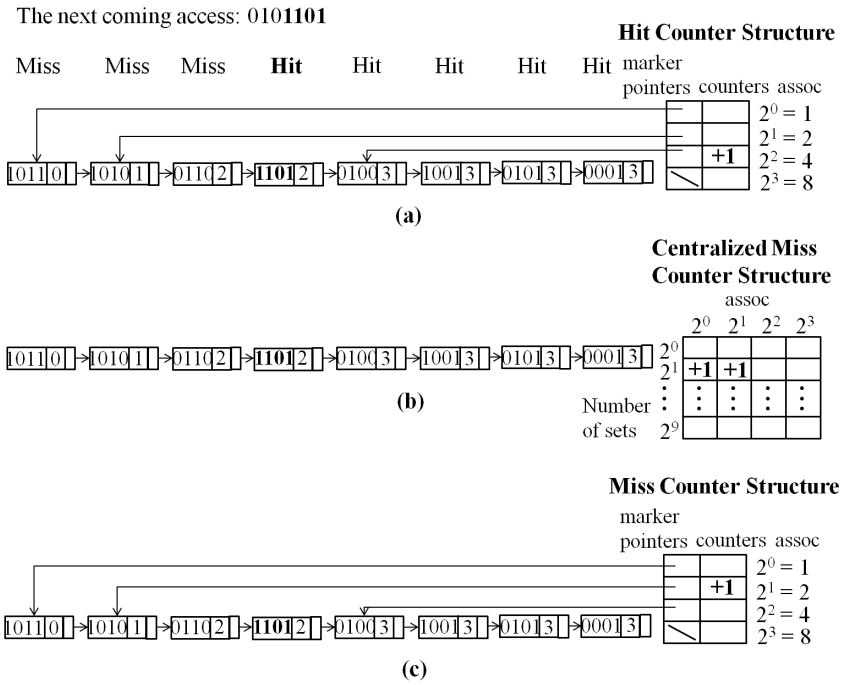


Figure 2.8: Counter structure design: (a) hit counter structure, (b) centralized miss counter structure, and (c) miss counter structure

2.3.3 HC-Sim Algorithm

Algorithm 1 is proposed to simulate set-associative caches based on the HC-Sim data structures. For each cache access, we will first check to determine if the block address of the access is already in the centralized hash table (line 12). If so, then we check the node pointers in the array to determine the hit/miss status for all configurations. If the node pointer is valid, it means that it points to a node in the corresponding LRU stack, and thus a hit is detected. If the node pointer is invalid, a miss is detected. Lines 14-29 show the checking and updating procedure. If the block address does not

exist in the hash table, then misses occur for all cache configurations. We need to insert the block into the hash table and perform corresponding updates (lines 30-40). Note that the observations of the CRCB algorithm [156] are also applied to avoid unnecessary accesses to node pointers.

To compute the time complexity of the algorithm, we assume that there are N cache accesses in the trace. Hence, there will be O(N) accesses on the centralized hash table. For each access, the node point array is accessed for at most O($log_2(S)$) times, and thus the complexity of updating miss counters is also O($log_2(S)$). However, the number of marker pointers required to be updated may be O($log_2(A)$) per linked-list access. Therefore, the time complexity of Algorithm 1 is O(N + $N(log_2(S))$ + $N(log_2(S))(log_2(A))$), which is equal to O($N(log_2(S))(log_2(A))$). In the worst case, HC-Sim outperforms previous methods [93, 156, 80] since the complexity of updating marker pointers is O($N(log_2(S))(log_2(A))$), which is more efficient than that of performing linear search. The time complexity of linear search is O($N(log_2(S))A$), as described in Section 2.2.5. For the counter updates, the time complexity of HC-Sim is also less than that of the previous work [93, 156, 80], which is O($N(log_2(S))(log_2(A))$). Note that Algorithm 1 can be slightly modified to be used for a hit-counter-based HC-Sim, while the time complexity does not change.

2.3.4 HC-Sim Implementation

For scalability concerns, HC-Sim is implemented based on the Pin framework. Pin is a dynamic binary instrumentation system that supports x86 architectures. We can place instrumentations on different types of instructions by using Pin's APIs. The implementation of HC-Sim can be partitioned into two parts, as shown in Figure 2.9. Trace Generator bypasses the instructions which are not memory accesses. This

Algorithm 1 HC-Sim Algorithm

```
1: H → the centralized hash table in HC-Sim
2: array_size → the size of a node pointer array
3: prev_x → the block address of the previous cache access
4: for each cache access x do
5:     block_x = get_block_address(x)
6:     /* CRCB2: Two consecutive accesses are in the same cache line */
7:     if block_x == prev_x then
8:         Break
9:     end if
10:    prev_x = block_x
11:    if block_x is found at entry h in H then
12:        for i = 1 to array_size do
13:            /* Cache hit(s) occur(s) in this cache set */
14:            if h.array[i] is valid then
15:                if h.array[i] does not point to the head of the stack then
16:                    Move the node pointed by h.array[i] to the head
17:                    Increment the miss counter in the corresponding counter structure
18:                    Update the marker pointers in the corresponding counter structure
19:                end if
20:                /* Cache miss(es) occur(s) in this cache set */
21:            else
22:                Push the new node of x into the corresponding stack
23:                Point h.array[i] to the head of the stack
24:                Increment the miss counter of largest associativity in the corresponding counter structure
25:                Update all marker pointers in the corresponding counter structure
26:                Pop out the tail and invalidate the corresponding node pointer in H
27:            end if
28:        end for
29:        /* Cache misses for all configurations */
30:    else
31:        Insert block_x into the hash table
32:        for i = 1 to array_size do
33:            Push the new node of x into the corresponding stack
34:            Point h.array[i] to the head of the stack
35:            Increment the miss counter of largest associativity in the corresponding counter structure
36:            Update all marker pointers in the corresponding counter structure
37:            Pop out the tail and invalidate the corresponding node pointer in H
38:        end for
39:    end if
40: end for
```

can be done by placing instrumentations before every load/store instruction. When a memory access is identified by Trace Generator, the address will be sent to Simulation Engine. Next, Simulation Engine simulates the hit/miss status for each configuration and finally gives the control back to Trace Generator. Simulation Engine implements the data structures and the corresponding algorithms described in Section 2.3.2 and Section 2.3.3 respectively.

In addition, the implementation of the hash table uses the $google - sparsehash$ [82], which is an extremely memory-efficient implementation. The overhead is only 2 bits/entry. The memory-efficient feature is important since the memory demand of HC-Sim is proportional to the number of entries in the hash table. The number of entries, which is usually a large number, depends on the number of distinct cache lines occurring in the execution of an application. Therefore, the memory efficiency is one of the important factors determining the scalability of HC-Sim.

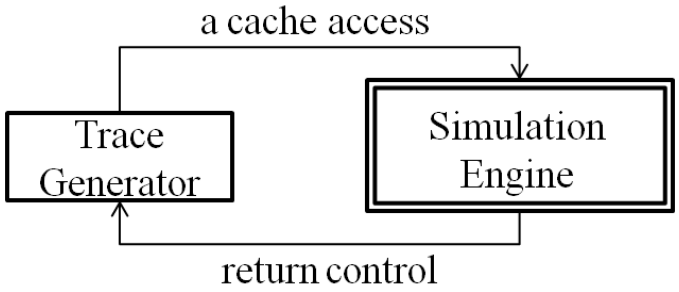


Figure 2.9: Implementation of HC-Sim

2.4 SPM Co-Simulation Support

In this section we discuss how to add the co-simulation support for scratchpad memory (SPM) into HC-Sim. Our motivation is to evaluate the miss rates of different L1

cache configurations when SPM is provided. The miss rates of caches can provide information for designers to select a suitable L1 cache configuration under a given SPM configuration for a target application. Based on the fast and exact L1 cache simulation framework, we show that the co-simulation can be done accurately and efficiently.

2.4.1 SPM Access Handling

In a hybrid cache system, a memory access will be directed to the L1 cache and SPM according to its memory addresses. Therefore, to obtain the miss rates of different L1 cache configurations, the accesses to SPM should be filtered out. Since the accesses to SPM have an address range which is not overlapping with the range of cache accesses, one feasible solution is to examine the address of each access before the cache simulation is performed. Figure 2.10(a) shows a code sample of *riciandenoise* [86] re-written by a programmer for performance optimization by using SPM. Here, we assume that a one-dimensional array is used to model SPM in the program, and the address range of the SPM array is contiguous. The accesses to SPM can be specified by a programmer or a compiler. We provide an interface function, which is called *spm_alloc*(), for programmers to specify the starting address and the size of the array, as shown in Figure 2.10(a) (line 72). This function is added to inform HC-Sim about the address range of SPM accesses. The same interface is used in [50, 49]. In [49], the interface is integrated with a full system simulator to support co-simulation. Figure 2.12 shows the design of the co-simulation support for HC-Sim. By intercepting the "spm_alloc()", the address range of SPM can be known in advance by an address filter, as shown in Figure 2.11. Therefore, SPM accesses can be filtered out, and they do not enter the simulation engine. Even if the address range is non-contiguous or

segmented, the interface function still can be applied to specify the address range of SPM.

```
0 double spm[4*128+2*14];
1 void riciandenoise(double *u, double g*, …)
2 {
3   for(p = 1; p < P-1; p++)  {
4     /* Prefetch data to SPM */
5     for(m=0; m<M; m++)  {
6       spm[offset+m]=u[m][n-1][p];
7       spm[offset+m+M]=u[m][n][p];
                  • • •
31    /* Load data from SPM and perform computations */
32    for(n = 1; n < N-1; n++)  {
33      for(m = 1; m < M-1; m++)  {
34        g(m, n, p) = 1.0/sqrt( EPSILON
35          + SQR(spm[offset+m+n%4*M] - spm[offset2+1])
36          + SQR(spm[offset+m+n%4*M] - spm[offset2+2])
37        +  ... )
                  • • •
70 int main()
71{
72     spm_alloc(&spm[0], 4*128+2*14);
73     riciandenoise(u, g, …);
74}
```

(a)

```
LOAD    EAX, u[m][n-1][p]
STORE   spm[offset+m], EAX
```

(b)

Figure 2.10: (a) An SPM code sample of *riciandenoise* (b) detection of prefetching loads

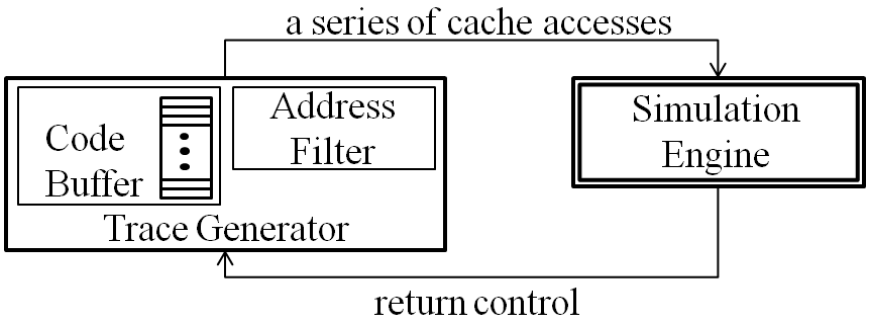


Figure 2.11: Co-simulation support for HC-Sim

2.4.2 Prefetching Support and Prefetching Loads

Prefetching is a common technique to improve the performance of SPM. In Figure 2.10(a), the expressions in line 6 and line 7 represent that the data in array u should be prefetched into SPM from lower-level cache or main memory for future computation. The expression in line 7 would be translated into instructions that contain one load and one store, as shown in Figure 2.10(b). First, a load instruction is generated since the data in u should be loaded into the register. After that, a store instruction is generated to store the data loaded from array u. The second store instruction would be filtered out as described previously. However, the first load instruction should never enter the simulation engine. It is used to model the behavior of prefetching, but it is not a real access on L1 caches. We call these loads prefetching loads.

However, it is not trivial to filter out these prefetching loads. One reason is that implementation of HC-Sim is based on Pin. Through binary- level instrumentation, we can only observe the instruction-level information, such as register names or the types of instructions. One way to handle this is by tracking the dependence between the SPM store and the prefetching load. However, another problem is that the prefetching load always occurs before the SPM store. Therefore, we introduce a

code buffer to deal with the problem. With the assistance of the code buffer, memory accesses can be temporarily stored. When an SPM store is detected, we can filter out the prefetching load corresponding to that store and then re-simulate the memory accesses stored in the code buffer in program order. Therefore, the correctness of cache simulation can be maintained. Figure 2.12 shows the co-simulation flow. There is one potential problem for the code buffer mechanism. A prefetching load may be simulated when the buffer is full. However, this case is rare if an adequate size of code buffer is given since the prefetching load and the SPM store are adjacent. For example, only 0.01% of prefetching loads are simulated when we perform simulation on the *riciandenoise* benchmark. The buffer size is 10000 elements. The simulation error is negligible.

2.5 Experimental Results

2.5.1 Simulation Setup and Workloads

As mentioned in Section 2.3.4, we implement HC-Sim based on the Pin framework for scalability concerns. Therefore, we also implement SuSeSim [80] and the CRCB algorithm [156] based on the Pin framework to perform a fair comparison. All methods are implemented in C++ and compiled by the g++ compiler with '-O3' optimization level. Simulations are performed on an Intel Xeon Quad 2Ghz processor with 8GB main memory.

Since the trace is dynamically generated during runtime by the tool itself, the addresses of memory references will be different every time, and that makes verification difficult. Therefore, we implement a golden version that takes a trace file as an input

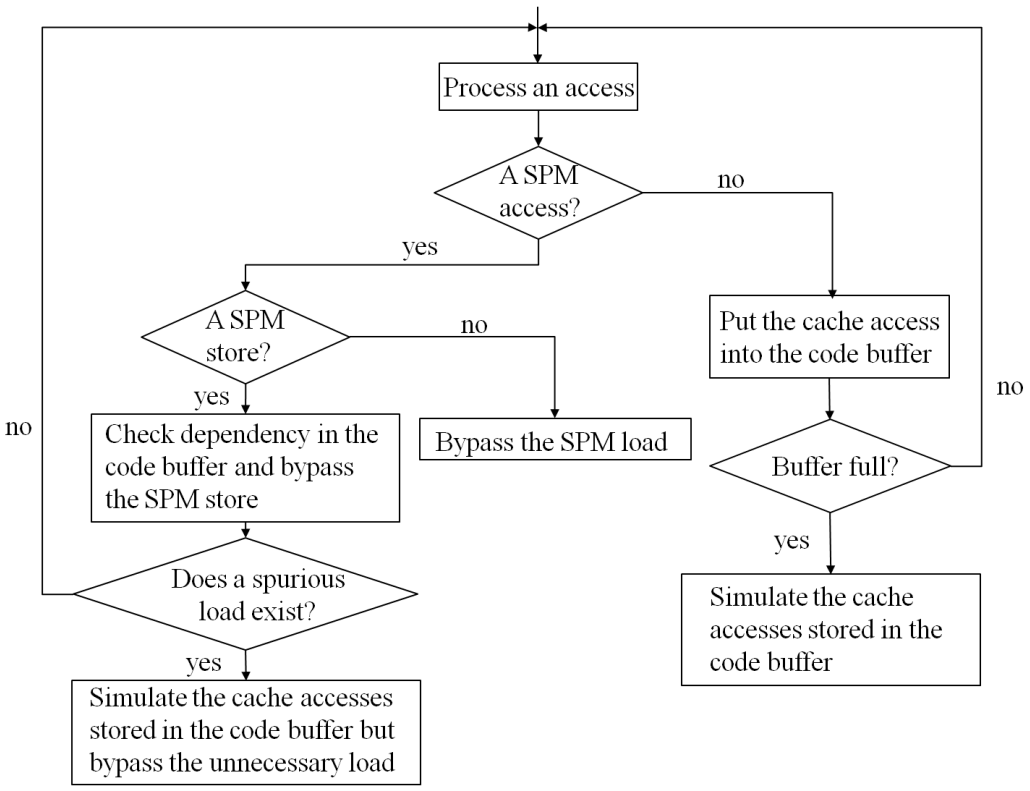


Figure 2.12: Flow of co-simulation on SPM and L1 caches

for each method, and the golden version is verified by DineroIV [66]. The golden version provides a reference to guarantee the correctness of the implementations on the Pin framework.

We simulated 400 configurations for each method on each workload. The range of each cache parameter is described in Table 2.1. Note that we consider only L1 data cache simulation in this dissertation. However, the proposed method can also be applied to L1 instruction cache simulation.

As shown in Table 2.2, the benchmarks we used in this dissertation cover memory-intensive workloads from SPEC2006 [88]. The medical imaging benchmarks are a set

Table 2.1: Cache configurations

The Number of Cache Sets (s) = 2^i	$0 \leq i \leq 9$
Associativity (a) = 2^i	$0 \leq i \leq 9$
Cache Line Size (b) = 2^i	$4 \leq i \leq 7$

of benchmarks that are used to process the image produced by a CT scanner [86]. The benchmarks include applications for image reconstruction, denoise, deblur, image registration, and image segmentation. These medical imaging benchmarks consist of stencil computation and are data-intensive. The total number of accesses and the simulation time of miss-counter-based HC-Sim for each benchmark are listed in Table 2.3. In our experiment, the largest trace contains 87.07 billion accesses, which results in a huge disk demand in terabyte scale. By using HC-Sim, we can eliminate the problem and can perform simulation on an even larger program.

Table 2.2: Workloads

Benchmark	Applications
SPEC2006	gcc, mcf, libquantum, h264ref, astar soplex, lbm
Medical Imaging	rician-denoise, rician-deblure, registration segmentation, compressive sensing

2.5.2 Performance Evaluation

Table 2.4 shows the normalized simulation time of HC-Sim, SuSeSim [80], and the CRCB algorithm [156]. Note that the simulation time of SuSeSim, CRCB, and hit-counter-based HC-Sim are normalized to that of miss-counter-based HC-Sim. Compared to hit-counter-based HC-Sim, SuSeSim and CRCB, miss-counter-based HC-

Table 2.3: Trace size and simulation time of HC-Sim (M)

Benchmarks	Num of accesses	Simulation time (Second)
mcf	4.84B	33712
lbm	3.40B	38526
gcc	5.53B	112523
h264ref	87.07B	324017
astar	64.30B	198116
soplex	128.94M	452
libquantum	70.14M	339
riciandenoise	3.34B	33892
riciandeblure	10.85B	61233
registration	10.46B	49557
segmentation	6.03B	30389
comp. sensing	35.55B	139727

Sim can achieve up to 1.27X, 5.21X and 13.73X reduction in runtime respectively. On average, miss-counter-based HC-Sim can run 1.12X, 2.56X and 5.44X faster than hit-counter-based HC-Sim, SuSeSim, and CRCB, respectively.

As mentioned in Section 2.3.3, the time complexity of HC-Sim is bounded by the marker update, which is $O(N(log_2(S))(log_2(A)))$. The complexities of previous methods [93, 156, 80] are bounded by linked-list traversal, which is $O(N(log_2(S))A))$. To show the effectiveness of the reduced complexity, we compare the number of marker updates of HC-Sim with the number of traversed list nodes for SuSeSim and CRCB, as shown in Figure 2.13. Here, we collect data for the cache configurations where $0 \leq s \leq 9$, $0 \leq a \leq 9$, and $b = 64$. The data are normalized to the number of marker updates of miss-counter-based HC-Sim. We can observe that the number of marker

Table 2.4: Normalized simulation time

Benchmarks	HC-Sim (M)	HC-Sim (H)	SuSeSim	CRCB1
mcf	1.00	1.03	3.74	8.57
lbm	1.00	1.14	5.21	13.73
gcc	1.00	1.10	2.38	3.33
h264ref	1.00	1.27	1.70	2.27
astar	1.00	1.19	1.22	1.69
soplex	1.00	1.25	1.50	2.04
libquantum	1.00	1.07	3.05	3.89
riciandenoise	1.00	1.09	2.52	7.49
riciandeblure	1.00	1.05	4.17	12.47
registration	1.00	1.05	2.28	5.92
segmentation	1.00	1.12	1.03	1.29
comp. sensing	1.00	1.09	1.86	2.58
Average	**1.00**	**1.12**	**2.56**	**5.44**

updates is from 5X to 13X and from 7X to 40X less than the traversed list nodes of SuSeSim and the CRCB algorithm, respectively. Therefore, the number of nodes accessed is significantly reduced through the hashing-based structure. Even if the constant overhead of a hash table access is large, the time complexity reduction from $O(N(log_2(S))A))$ to $O(N(log_2(S))(log_2(A)))$ still compensates for the overhead.

Figure 2.13(b) quantifies the benefit of using miss counters to further improve the efficiency of HC-Sim. First, we can observe that the number of counter updates of miss-counter-based HC-Sim is from 2.2X to 7.6X less than that of hit-counter-based HC-Sim. Moreover, compared to the centralized miss counter structure used in [156, 80], miss-counter-based HC-Sim reduces the number of counter updates from

2.1X to 5.5X since the time complexity is reduced from O($N(log_2(S))(log_2(A))$) to O($N(log_2(S))$). Note that we collect the data where the cache line size is equal to 64 bytes.

2.5.3 Evaluation of Co-Simulation Support

In this section a case study of *riciandenoise* is used to evaluate the efficiency of co-simulation support for HC-Sim. Figure 2.14 shows the miss rate distribution of a hybrid cache system with an L1 cache and SPM and a cache-only system. The cache line sizes for both systems are 64 bytes. In the hybrid system, the size of SPM is 4320 bytes as indicated in Figure 2.10(b). The input size of *riciandenoise* is an image with $64 \times 64 \times 64$ pixels. Figure 2.14(a)(b) shows the miss rates distribution of both systems. With SPM support, the hybrid cache system has a lower miss rate distribution as shown in Figure 2.14(b).

The simulation time of HC-Sim with co-simulation support takes only 122 seconds, while the simulation time for the cache-only system takes 309 seconds. The first reason is that SPM accesses are filtered out, and thus the number of accesses for cache simulation is reduced. Second, with the assistance of SPM, the miss rates are reduced, leading to faster cache simulation. In comparison, the authors in [50, 49] recently evaluated the performance of SPM with a full system simulator [126] which takes almost eight hours to obtain the result of only one cache configuration with SPM. With HC-Sim, we can evaluate the distribution of miss rates more efficiently and thus provide the capacity for an early stage design space exploration with more than 10,000X simulation runtime reduction. When the miss rate distribution of L1 caches is the metric we would like to measure, full system simulation is inefficient and is not required.

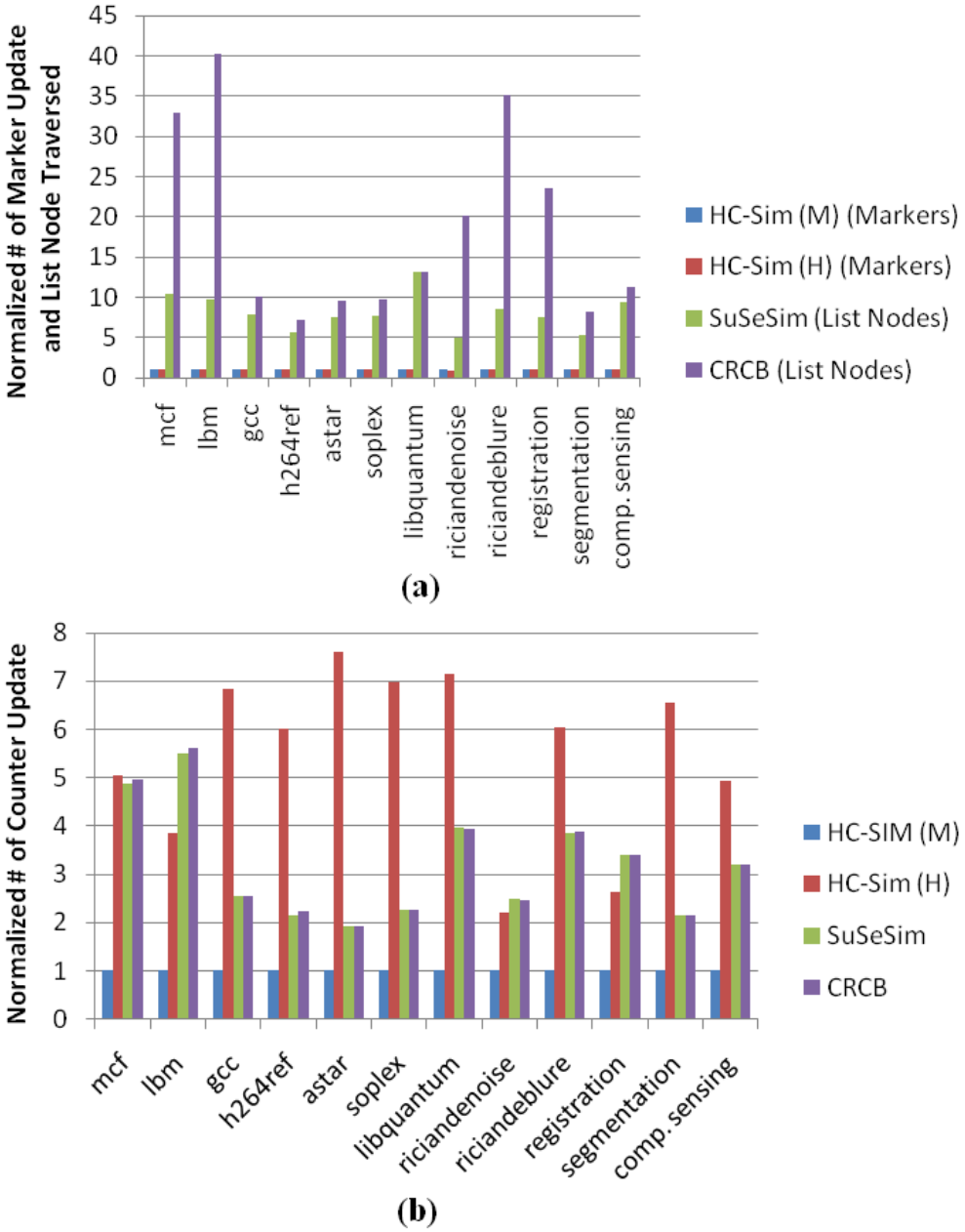


Figure 2.13: Normalized numbers of (a) marker updates and traversed list nodes and (b) counter updates ($b = 64$ bytes)

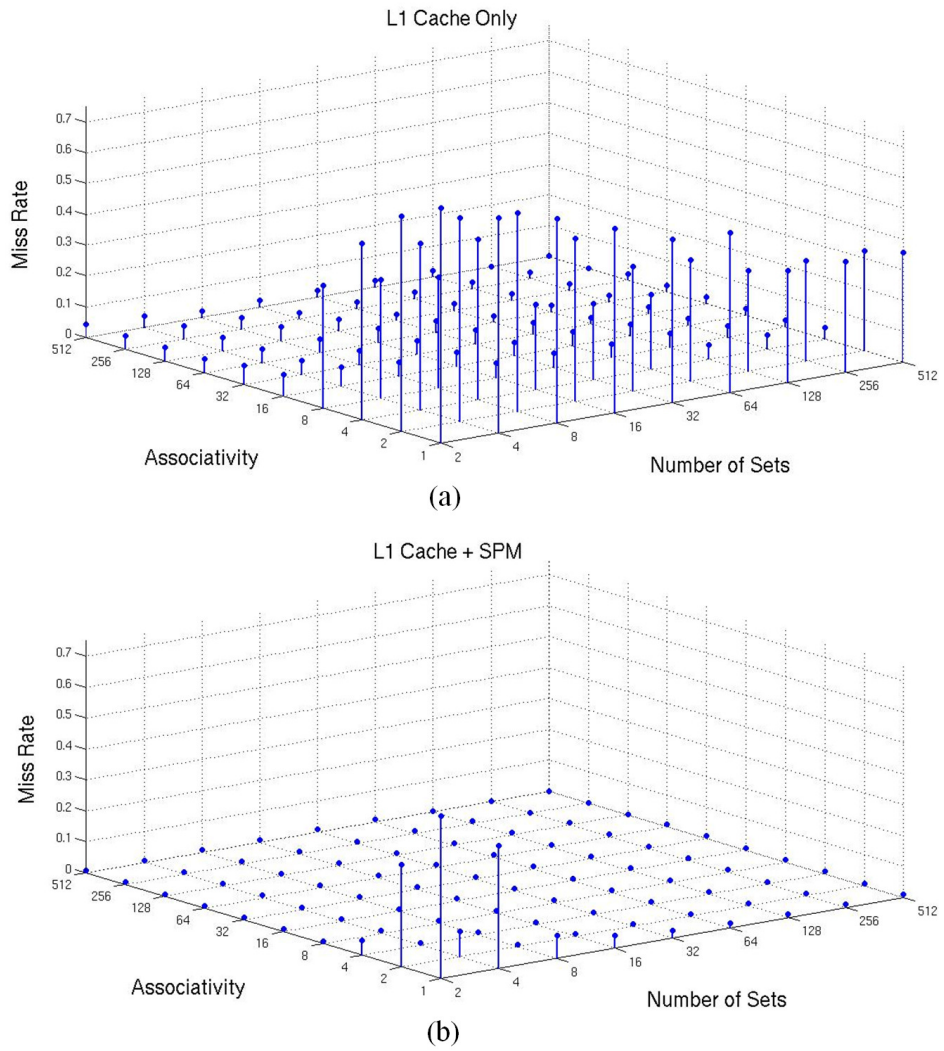


Figure 2.14: Miss rate distribution of (a) L1 caches only (b) L1 caches + SPM

2.6 Related Work

Section 2.2 provided a detailed review of simulation-based methods. In this section we discuss analytical approaches for cache miss rate estimation. In contrast to the exact simulation-based methods, the analytical approaches provide a fast estimation of cache misses.

The researchers in [70] use system-level simulations to estimate the energy-delay product under different cache configurations. In contrast to exploring the whole cache design space, they use a sensitivity-based analysis to optimize the three cache parameters sequentially. In their work, the optimization procedure is to give initial values of cache line size and associativity and first optimize the number of cache sets according to the ED metric. Next, the line sizes are optimized under the fixed number of sets. Finally, the associativity is optimized.

In [75] the cache miss equations (CME) are proposed to represent the cache misses of a loop nest. By analyzing the iteration space and reuse vectors of a loop nest, the CMEs can capture cache misses in a simple loop nest, such as a perfect nested loop, accurately. The cache parameters, such as associativity, line size, and the number of sets, are treated as inputs for the CMEs. In [161] the authors provided a fast and accurate approach to solving CMEs. By using a sampling technique, a small subset of the iteration space can be analyzed to approximate the miss ratio of each reference.

In [21], the authors developed a probability model to estimate miss rates by using the stack distance distribution of the cache accesses in a program. They showed that the miss rate can be estimated accurately with a 10^{-4} sampling rate. However, the approach can only be used on fully associative cache.

In [74], instead of calculating the cache miss rates for different cache configurations, the authors try to obtain the configurations that satisfy the constraint of a cache miss rate. They provided an exact method for analyzing the trace and finding feasible configurations for different miss rate targets.

In general, the estimation approaches may be inaccurate since they do not examine the hit/miss status of each cache access. In addition, some analytical models can be

only applied on a subset of caches [21] or a subset of programs [75, 161].

2.7 Conclusions

We propose a fast and exact L1 cache simulator, HC-Sim, to simulate multiple cache configurations in one run. HC-Sim adopts a centralized hash table and supplementary data structures to efficiently reduce the search time performed on the LRU stacks. Assuming no prefetching, HC-Sim can simulate multiple caches that adopt the LRU replacement policy simultaneously. For a collection of 12 workloads, HC-Sim can be 2.56X and 5.44X faster than the SuSeSim and CRCB algorithms on average. To enhance the scalability, HC-Sim is implemented based on the dynamic instrumentation framework, Pin, to generate traces during runtime. The overhead of huge trace files can thus be avoided. In addition, HC-Sim provides the capacity to perform co-simulation on L1 caches and SPM simultaneously. The miss rates of L1 caches can be efficiently calculated with a given SPM configuration. Therefore, designers can efficiently explore the design space of a hybrid memory system consisting of L1 cache and SPM.

CHAPTER 3

Reconfigurable Hybrid Cache: Architecture Design and Dynamic Reconfiguration

3.1 Introduction

The traditional SRAM-based on-chip cache has become a bottleneck for energy-efficient design due to its high leakage power. Designers have turned their attention towards emerging non-volatile memories such as the spin-torque transfer magnetoresistive RAM (STT-RAM) and phase change RAM (PRAM) to build future memory systems. Power, performance, and density characteristics of the new memory technologies differ dramatically compared to SRAM, and thus they enlarge the landscape of memory design.

Table 1.1 shows a brief comparison of SRAM, STT-RAM, and PRAM technologies. The exact access time and dynamic power depend on the cache size and the peripheral circuit implementation. In sum, SRAM suffers from high leakage and low density while providing great endurance; STT-RAM and PRAM provide high density and low leakage at the cost of weak endurance. Moreover, STT-RAM outperforms PRAM in terms of endurance, access time, and dynamic power, while PRAM has higher density.

With desirable characteristics on leakage power and density, NVMs have been

explored as an efficient alternative for either SRAM or DRAM in memory systems [165, 154, 65, 143, 107]. As can be seen in Table 1.1, compared to PRAM, STT-RAM has higher endurance (10^9 versus 4×10^{12} write cycles) [87, 32]. Based on the write cycles, we use a similar endurance model proposed in [92] to calculate the lifetime of PRAM and STT-RAM in an on-chip hybrid cache which consists of 1MB SRAM and 3MB NVM. Table 3.1 demonstrates the lifetime of the hybrid cache for three write-intensive workloads selected from medical imaging domains [25] and PARSEC [22]. For a PRAM-based hybrid cache, the lifetime is limited, ranging from 4.70 to 196.12 days; but the STT-RAM-based hybrid cache can last for more than tens of years. Thus, STT-RAM is more suitable for on-chip last-level cache [165, 154, 65, 143] design due to its higher endurance, while PRAM is promising as an alternative for DRAM in the main memory design due to its higher density [107, 140]. Therefore, in this dissertation we will focus on a hybrid cache architecture with STT-RAM as the NVM.

Table 3.1: Endurance (lifetime) of 4MB RHC and 2MB SRAM-based cache

Workloads	registration	segmentation	fluidanimate
PRAM (days)	4.70	196.12	39.33
STT-RAM (years)	12.88	537.32	107.76

A common problem in existing hybrid cache designs [165, 154] is the lack of adaptation to varied workloads. Previous studies show that different applications may exhibit different characteristics [142]. For example, if the targeted application accesses a 10MB working set in a streaming fashion, then a fixed hybrid cache design consisting of a 2MB SRAM and 8MB NVM (as discussed in [154]) may become inefficient in terms of both performance and energy compared to a 2MB SRAM-only design. In the 2MB SRAM-only design, all data blocks are put into the SRAM to achieve fast

access. However, in the 2MB SRAM with 8MB NVM design, the data blocks are distributed in both the SRAM and NVM regions, while most blocks are located in the NVM region. The cache miss rates are the same for the two architectures due to the streaming access pattern, but the performance degrades because of the longer access latency of NVM. The energy consumption in the hybrid design is larger since it consumes more leakage due to longer runtime and additional leakage from NVM arrays. Also, the greater dynamic write energy on NVM increases the total energy consumption. Therefore, if we can provide configurability on the hybrid cache design, it can be reconfigured to accommodate varied workloads. In this example the hybrid cache can be reconfigured into 2MB SRAM to achieve the best performance and energy efficiency.

In this chapter we propose a novel reconfigurable hybrid cache design (*RHC*). Our design explores the use of NVM cache to replace the conventional all-SRAM design in the last-level cache to efficiently reduce leakage energy. The proposed RHC design supports reconfigurable SRAM/NVM size, with the capability of powering on/off SRAM and NVM arrays in a way-based manner for better accommodation of memory requirements from different workloads. Hardware-based mechanisms are proposed to detect the cache demand for dynamic reconfiguration. On average, RHC significantly saves 64%, 46% and 28% energy over a non-reconfigurable SRAM cache, a non-reconfigurable hybrid cache and a reconfigurable SRAM cache, respectively, while maintaining the system performance (at most, only a 4% performance overhead).

The remainder of the chapter are organized as follows. Section 3.2 introduces the NVM technologies of both STT-RAM and PRAM. The architecture of the proposed RHC is demontrated in Section 3.3. Section 3.4 describes our evaluation methodology, and the evaluation results are shown in Section 3.5. Section 3.6 discusses the related

work. Section 3.7 concludes the chapter and discusses the future work.

3.2 Background: STT-RAM and PRAM

Figure 3.1 shows the schematic views of an STT-RAM and a PRAM cell. Both of them can be modeled as a 1T1R structure. In STT-RAM, a magnetic tunnel junction (MTJ) is used as the information carrier. Each MTJ consists of two ferromagnetic layers and one tunnel barrier layer. If the two ferromagnetic layers have different magnetic directions, the MTJ resistance is high, representing a binary 1 state. If the two ferromagnetic layers have the same direction, the MTJ resistance is low, representing a binary 0 state.

The storage mechanism in PRAM relies on two stable states of phase-change materials (GST). The crystalline state has low resistance, representing binary 1 (*set* state); the amorphous state has high resistance, representing binary 0 (*reset* state). The GST can be converted from one state to another by applying heat. like chalcogenide. $Ge_2Sb_2Te_5$ (GST) is the most widely used chalcogenide.

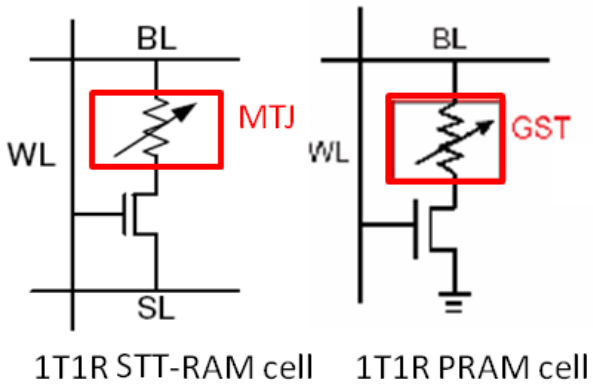


Figure 3.1: A STT-RAM and a PRAM cell

3.3 Reconfigurable Hybrid Cache Design

In this section we discuss the RHC design in the following way. First, we present the architecture and circuit design of the RHC with disparate SRAM and NVM technologies. Next, the reconfigurability support for the RHC is discussed. Finally, we propose hardware-based schemes for dynamic reconfiguration.

3.3.1 Hybrid Cache Architecture

Figure 3.2 shows an overall structure of RHC. In RHC the data array is partitioned into SRAM and NVM at a cache-way granularity. One concern of RHC design is that the access latency of a NVM cell is longer than that of SRAM [165]. In a simple hybrid cache design where the tag and data arrays of each cache way are implemented either with all SRAM cells or NVM cells, the cache critical path will always be dominated by the longer access latency to the NVM cache ways. To overcome this, RHC is designed in the following way. First, the accesses to the tag array and data array are done sequentially, (i.e., the data array will be accessed after the tag array). Such a serialized tag/data array access has already been widely adopted in a modern low-level large-scale cache for energy reduction. Second, the RHC tag array is fully implemented with SRAM cells. In RHC, each tag entry contains only four bytes, including the tag, coherence state bits, the dirty bits, etc., while each cache block in the data array contains 64 bytes. Hence, the SRAM-based RHC tag array will not create a large energy overhead.

The circuit design of RHC with STT-RAM as the NVM is as follows. First, an STT-RAM cell has a bitline (BL) and a source-line (SL) for its operation. This is similar to the bitlines (BL, BLB) used in SRAM. Therefore, the organization of an

STT-RAM data array is almost the same as a SRAM data array. Second, the sense amplifiers need to be modified due to the single-ended bitlines [65]. According to a recent implementation [159] of an STT-RAM array, the reference voltage is 1.2V, which is close to a SRAM-based design. Therefore, additional power-supply pins to support the read/write accesses of the STT-RAM array may not be required.

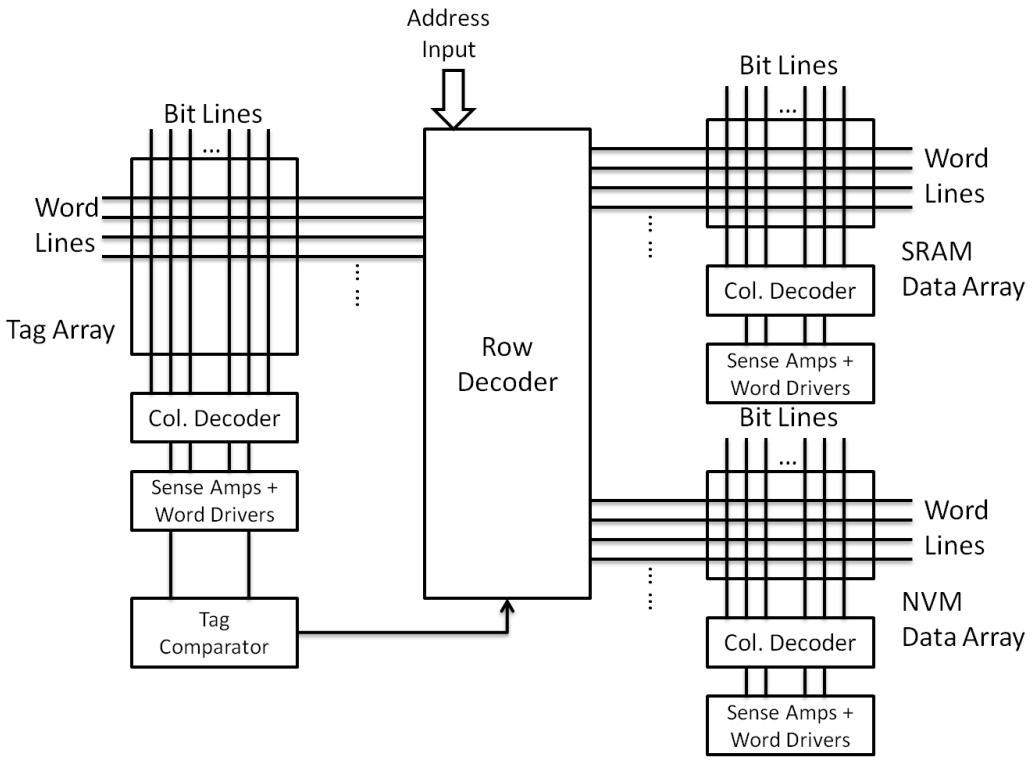


Figure 3.2: Reconfigurable hybrid cache (RHC) design

3.3.2 RHC Reconfiguration Design

The reconfiguration in RHC is realized by powering on/off SRAM and NVM arrays arbitrarily in a way-based manner. From an architectural point of view, the reconfiguration mechanism in RHC is similar to the existing way-based reconfigurable SRAM

cache [12]. Data accesses will not be directed to a disabled cache way, thus those ways in the data array dissipate no dynamic power. Note that the replacement decision logic within the cache controller must ensure that no data will be allocated to a disabled cache way.

In Figure 3.3 we illustrate the power-gating design adopted in RHC. A centralized power management unit (PMU) is introduced to send sleep/wakeup signals to power on/off each SRAM or NVM way. The power-gating circuits of each way in SRAM tag/data arrays are implemented with NMOS sleep transistors to minimize the leakage. In this design the stacking effect of three NMOS transistors from the bitline to GND substantially reduces leakage [137]. Note that in RHC, the SRAM cells in the same cache way will be connected to a shared virtual GND while the virtual GNDs among different cache ways are disconnected. This can ensure that the behaviors of cache ways that are powered-on will not be influenced by the powering-off process in other ways.

For the peripheral circuits, such as the row decoder, column decoders, word drivers, and sense amplifiers, we use PMOS sleep transistors to implement the power-gating design; this can provide better performance of the peripheral circuits in the active mode [29]. Since the NVM cell itself consumes little leakage, we do not introduce extra power-gating circuits for the cells of NVM data arrays. To power on/off a NVM cache way, PMU will send a sleep/wakeup control signal to the peripheral circuits of the corresponding NVM way. The design complexity of the PMU is highly related to the adopted wakeup scheme. In this dissertation we assume a daisy-chain wakeup scheme for each cache way [150]. For the PMU, we use Synopsys SAED 90nm technology, which is the most advanced process technology available, to obtain the energy and delay numbers. An RTL-level description of PMU is synthesized by Syn-

opsys Design Compiler^{TM}. The dynamic energy is 0.0135pJ for one reconfiguration, while the leakage power is 1.0378uW. The delay of the PMU is 0.28ns. The overhead of the PMU is small and thus can be neglected.

The overhead of the reconfiguration will be classified in the following two categories. First, when a cache way is disabled, the dirty blocks in that cache way need to be written back to lower-level memory. This will introduce both performance and energy overhead. Second, from a circuit-level perspective, the power-up process also involves extra energy consumption. The reason for this is that the accumulated charge during the standby mode in SRAM cells should be discharged.

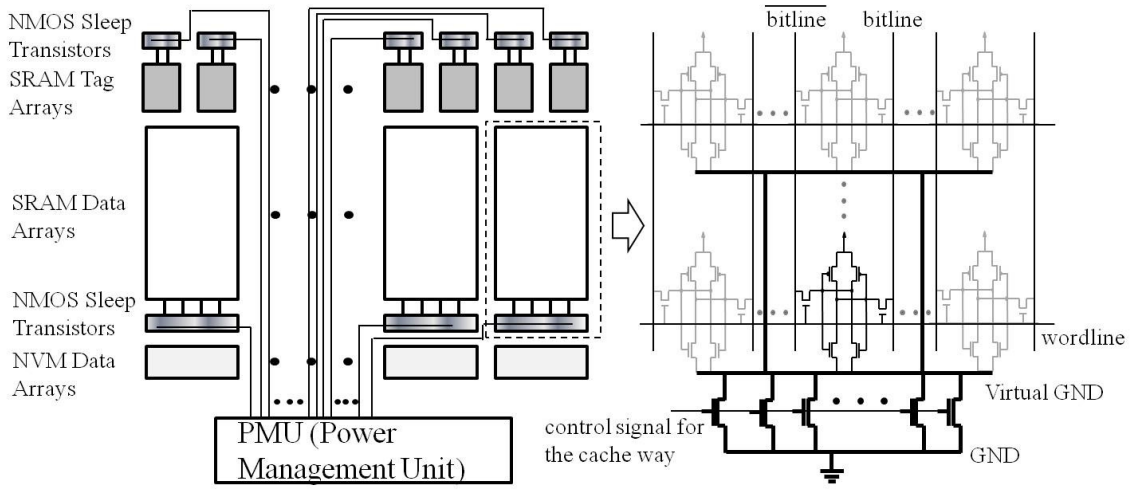


Figure 3.3: Power-gating design for RHC

3.3.3 Dynamic Reconfiguration

In this section we propose two hardware schemes to utilize the reconfigurability provided by RHC. The main idea is to detect the cache demand dynamically and reconfigure the RHC in a way-based manner to satisfy the demand. In the meantime, the

powered-off cache ways can provide energy savings in leakage.

3.3.3.1 Way-Based Decay Scheme

The reconfiguration scheme includes two dynamic decisions: (1) when to power off a cache way and (2) when to power on a cache way. To power off a cache way, we utilize the *cache decay* idea [98], and introduce the novel *way-based decay counters*, as shown in Figure 3.4. The main idea of cache decay is to power off a cache block which is not accessed for a long time period to save leakage. This time period is called the *decay interval*. Cache decay is implemented by a local 2-bit saturating counter for each block with a global cycle counter. The local counter is incremented when the global counter exceeds a certain number of clock cycles, which is used to model the decay interval. The local counter is reset to zero when there is an access on this block. Cache decay is initially used to provide a self-guided block-based power-on/off mechanism [98]. However, it is not feasible for PMU to arbitrate reconfiguration at the block-based granularity due to circuit design complexity. Therefore, we use the way-based decay counter to measure the number of decay blocks in that cache way during a time period. The way-based decay counter is incremented by one when any local 2-bit counter in that cache way saturates. Similarly, when a local 2-bit counter is reset to zero, the corresponding way-based decay counter is decreased by one. If the value of a way-based decay counter exceeds a given threshold in a given time period, such as 90% (used in this dissertation) of the blocks in that way, the whole cache way will be powered off due to the low cache demand.

To detect the demand for powering on more cache ways, we keep the whole tag array powered-on to record potential hits if those blocks are in RHC. The potential hit counter is increased by one when a hit occurs on a tag entry whose corresponding data

block is powered off (referred to as the *victim tags* [175, 49]). The victim tags reuse the same tag array, which does not create extra storage overhead. The replacement policy for the victim tags follows LRU policy. When the value of potential hit counter is greater than a threshold, a cache way is powered on to reduce cache misses. We denote this powering-on threshold as TH_{on}. Note that the power-on/off decision is made for every one-million cycles in this dissertation. This time period is called the *reconfiguration period*. Both the way-based decay counters and the potential hit counter are reset to zero after the decision is made.

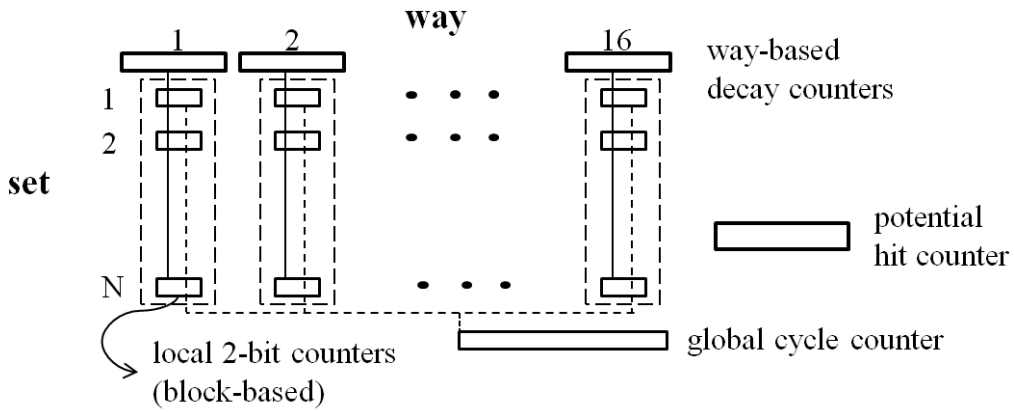


Figure 3.4: Counters for dynamic reconfiguration

3.3.3.2 Independent Potential Hit Counters Scheme

In this section, we provide an improved strategy to for dynamic reconfiguration. In the way-based decay scheme, a large number of cache ways can be powered off simultaneously since each cache way is controlled independently. However, we observe that this aggressive powering-off scheme may result in significant performance degradation due to the increase of L2 cache misses especially when the decay interval is small, such as one million cycles. Another potential problem is that a single decay interval

cannot accurately capture the varied decay intervals of all cache blocks, which also makes the way-based decay counter ineffective. When the decay interval is too large, such as 100 millions cycles, most of the blocks are accessed once during that interval. The powering-off decisions are seldom made and thus the energy reduction is limited.

The improved scheme takes both the hybrid nature of RHC and the aggressive powering-off issue into consideration. Considering the hybrid nature of RHC, it is beneficial to measure the cache demand for the SRAM and STT-RAM arrays independently to better accommodate the cache demand. Therefore, we use two potential hit counters to measure the cache demand of the SRAM and STT-RAM arrays independently.

The powering-off strategy is different from that of the way-based decay scheme. Here, we use the same potential hit counter to make the powering-off decision. We introduce another powering-off threshold (TH_{off}). When the value of the potential hit counter is less than or equal to TH_{off}, a cache way can be powered off. Based on the strategy, only one cache way can be powered off at a time period, and this greatly reduces the chance of cache thrashing. However, according to our observation, this strategy still generates considerable cache misses. To mitigate the aggressive powering-off strategy, we further restrict the powering-off condition. When the value of the potential hit counter reaches TH_{off}, we cannot power off a cache way immediately. A cache way can only be powered off after ten consecutive reconfiguration periods (ten-million cycles) have passed. Note that TH_{on} is set to 50 and TH_{off} is set to 0 for both SRAM and STT-RAM arrays for evaluation.

Furthermore, we consider the endurance of RHC when making decisions of reconfiguration. We achieve this by randomly selecting the cache way from all possible candidates. For example, when the decision is to power off a cache way, we will

randomly pick the victim from all powered-on cache ways.

3.4 Evaluation Methodology

3.4.1 Performance and Energy Models

We evaluate the proposed RHC design on a simulation platform built upon Simics [124] with GEMS [126]. Simics and GEMS provide the full-system simulation capability, especially for simulating complex cache coherency protocols, such as MESI. In Chapter 2 we proposed HC-Sim to simulate many cache configurations simultaneously for fast design space exploration. However, HC-Sim does not support fine-grained multilevel coherent cache simulation. Therefore, we still adopt the conventional full-system simulator for simulating hybrid caches.

Table 3.2 shows the parameters used in our model. The value K represents the number of cache ways that are powered on in a specific L2 cache configuration, which also equals the amount of "active" cache associativity. Notice that the configuration of the processor core, L1 caches, and main memory remains the same through all simulations.

For the energy of the memory technologies, we use the ITRS 32nm process model. The SRAM and STT-RAM energy/latency numbers used in our simulations are obtained from CACTI 6.5 [85] and the data scaled from [65], respectively. The energy numbers of a 4MB RHC and 2MB SRAM-based cache are listed in Table 3.3, where *Active* and *Standby* correspond to the power-on and power-off state. The standby leakage is estimated according to the ratio of active/standby leakage presented in [137]. This can be achieved through a careful power-gating design.

Table 3.2: Simulation parameters

	single-thread workload	multi-thread workload
#Core	1	4
Core	Sun UltraSPARC-III Cu processor core, 4GHz	
L1 Cache	32KB per core for I/D caches 4-way, 64-byte block, 1-cycle latency	
L2 Cache	RHC: 1MB SRAM + 3MB STT-RAM SRAM-based: 2MB K-way (K$\leq$16), 64-byte block	
L2 Cache Access Lat.	SRAM: 10 cycles STT-RAM read/write: 11/30 cycles	
Main Memory	4GB, 320-cycle access latency	

3.4.2 Benchmarks

Our testbench consists of 16 benchmark applications, which have been carefully chosen to represent memory-intensive algorithms in the fields of data processing, massive communication, scientific computation and medical applications. The applications include seven memory-intensive applications from SPEC2006 [88], four applications from PARSEC [22], and five applications from the medical imaging domain [25].

3.4.3 Reference Designs

To evaluate the effectiveness of RHC, we compare RHC with a traditional SRAM-based cache under the same area basis. RHC is set to 4MB, which is composed of 1MB SRAM and 3MB STT-RAM, while the SRAM-based cache is set to 2MB. This setting reflects the fact that STT-RAM is about four times denser than that of

Table 3.3: Energy of 4MB RHC and 2MB SRAM-based cache

L2 Cache Design	Tech.	Dyn. energy per acc. (nJ)	Active leak.(mW)	Standby leak.(mW)
4MB RHC	SRAM	0.137	431.30	14.38
	STT-RAM	Read: 0.278	116.92	3.897
		Write: 0.765		
2MB SRAM		0.288	711.29	23.71

SRAM. The area of the data arrays in 4MB RHC is about 0.875X that of the 2MB SRAM-based cache.

The associativity of both the 4MB RHC and 2MB SRAM-based cache are both 16-way. This setting provides the same reconfigurability on RHC and SRAM-based cache. RHC has four SRAM ways and 12 STT-RAM ways, while the SRAM-based cache has 16 SRAM ways. Both can be reconfigured from one cache way to 16 cache ways. To evaluate the effectiveness of RHC, we compare the performance and energy of RHC with three reference points: (1) *SC*: non-reconfigurable 2MB SRAM-based cache; (2) *HC*: non-reconfigurable 4MB hybrid cache (4-way SRAM + 12-way STT-RAM); and (3) *RSC*: reconfigurable 2MB SRAM-based cache. Note that the evaluation in Section 3.5.1 and Section 3.5.3 uses the scheme introduced in Section 3.3.3.2 for both RHC and RSC, while RSC uses a single potential hit counter.

3.5 Experimental Results

3.5.1 Effectiveness of RHC

Figure 3.5 shows the comparison results of L2 cache miss rates. Compared to the baseline SC, HC consistently has a lower miss rate (39% on average) because of the 2X larger cache capacity provided by the STT-RAM. But this consistent miss rate improvement is realized at the cost of more energy consumption compared to RSC and RHC, which will be discussed later. On the other hand, since RSC dynamically powers off the cache ways—although this is done based on the cache pressure—the reduced cache capacity consistently impairs the cache performance (77% more cache misses), especially when the dynamic reconfiguration scheme can not accurately capture the cache behavior. This can be observed in *bzip2* and *libquantum*. Compared to the baseline SC, there are two main scenarios with RHC: 1) for the cases where the applications have relatively large working sets (such as *bzip2*, *deblur* and *compressive sensing*), RHC can achieve a considerable miss rate reduction of 52% on average; 2) for the cases where the applications have relatively small working sets which can be held for a 2MB L2 cache, RHC will gradually power off half of the cache capacity. But during this process, some of the cache blocks with long reuse distance will be evicted, which results in slightly higher miss rates. Overall, RHC incurs 33% more cache misses compared to SC.

Figure 3.6 shows the comparison results of system performance in terms of runtime of the application on the system. These results are normalized to that of the baseline SC scheme. The runtime difference of the four design schemes mainly comes from the difference of the L2 cache miss rate. Compared to the baseline SC, HC consistently has better performance (0% to 36% less runtime) because of its consistently smaller miss

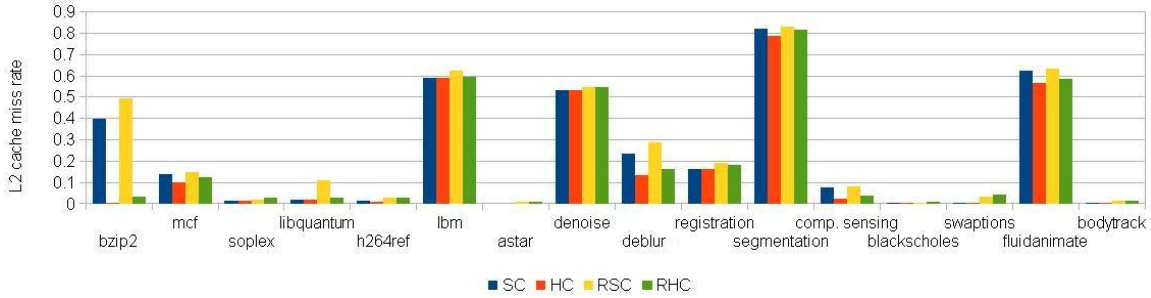


Figure 3.5: Comparison results of L2 cache miss rate

rate, while RSC consistently has worse performance (0% to 9% more runtime) due to its consistently larger miss rate. For the cases where RHC can achieve considerable L2 miss rate reduction, it also improves the performance (1% to 34% less runtime) over SC. For the other cases, RHC incurs a slight performance overhead (0% to 4% worse runtime).

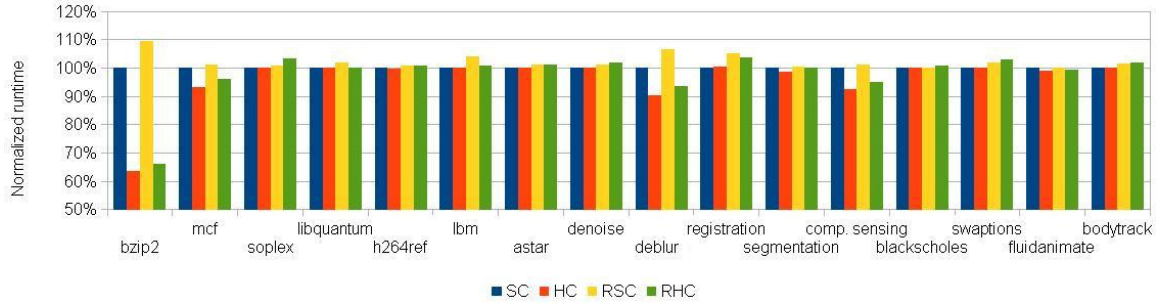


Figure 3.6: Comparison results of runtime

When it comes to energy, the power of the dynamic reconfiguration begins to show gain. Figure 3.7 shows the comparison of memory subsystem energy, The energy data is broken down into the L1 cache dynamic/leakage energy, and the L2 cache SRAM/STT-RAM dynamic/leakage energy for detailed illustration of the energy distribution. These results are normalized to that of the baseline SC scheme. The SRAM leakage dominates the memory subsystem energy in 32nm technology. Compared to

the baseline SC, HC reduces energy by 24% to 53% (30% on average), because the STT-RAM array consumes less leakage. In addition, HC consistently reduces the runtime, which reduces the SRAM leakage. By dynamically powering off the cache ways based on cache pressure, RSC can also reduce the energy by 7% to 88% (51% on average). In cases of *bzip2*, *deblur* and *compressive sensing* where the powering-on time of the remaining cache ways incurs an energy overhead which almost catches up with the reduction of the leakage in the powering-off cache ways, the energy reduction of RSC is much smaller than the other cases. By dynamically powering off the cache ways and maintaining the system performance, RHC achieves the least energy use among all the design schemes since RHC inherits the advantages of both the low leakage NVM array and dynamic reconfiguration to save leakage. It reduces energy by 63%, 48%, 25% compared to baseline SC, HC and RSC, respectively.

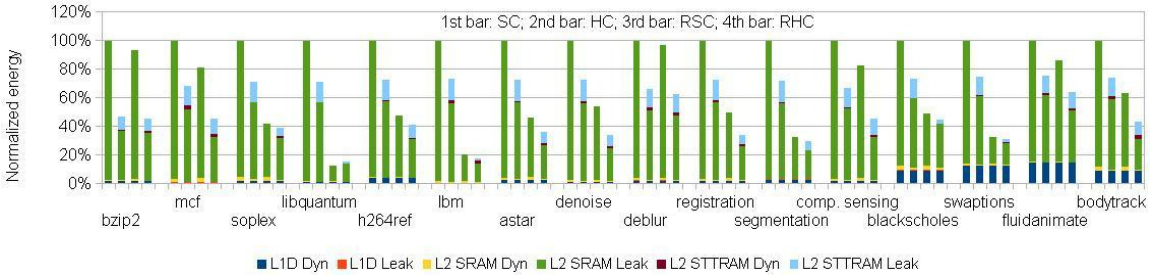


Figure 3.7: Comparison results of memory subsystem energy

To better illustrate the gain over other design schemes in terms of both energy and runtime, we use the metrics of energy-delay product (ED) to make the comparison, where the delay means the runtime. Figure 3.8 shows the comparison results of this metric over the four design schemes. All results are normalized to that of the baseline SC. The proposed RHC achieves the best ED among all the design schemes. On average, RHC improves the ED by 64%, 46%, and 28% compared to SC, HC and RSC, respectively.

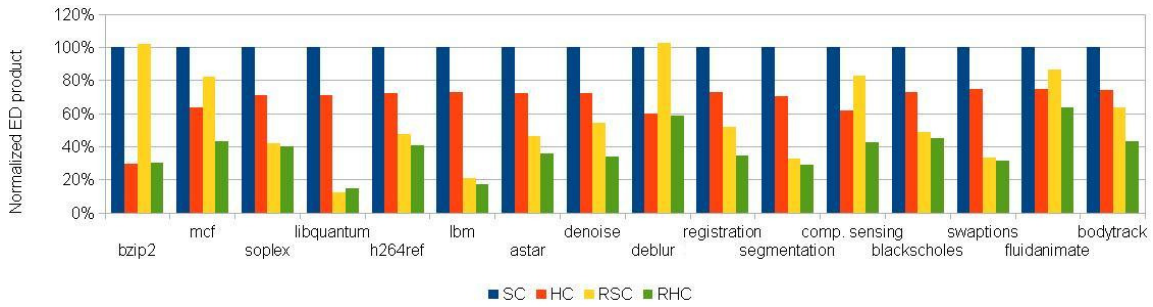


Figure 3.8: Comparison results of ED product

3.5.2 Comparison of Two Dynamic Schemes

For the way-based decay scheme with *independent potential hit counters* (IPHC), we evaluate three different decay intervals (1M, 10M, and 50M cycles). When the decay interval is larger, more cache ways are powered-on to maintain the performance. The largest interval we used is 50M cycles since the simulation results remain the same even when we enlarge the decay interval. Figure 3.9 shows the comparison results of runtime. The results are normalized to the baseline HC. The most critical disadvantage of the way-based decay scheme comes from the significant performance degradation, as shown in Figure 3.9. When the decay interval is set to 1M cycles, the performance drops from 2% to 131% compared to HC. Even when the decay interval is set to 50M cycles, *swaptions* still suffer from 27% performance degradation. In contrast, IPHC can provide stable performance within a 4% degradation compared to HC among all workloads.

Figure 3.10 shows the comparison of energy. IPHC can achieve better or at least similar energy reduction compared to the cases of 10M and 50M decay intervals. Therefore, IPHC can further provide energy savings when maintaining similar performance compared to the 50M decay interval case. When the decay interval is set to 1M

cycles, the way-based decay scheme achieves much better energy saving on *bzip2*, *segmentation*, and *comp. sensing*. However, *bzip2* and *comp. sensing* suffer from 122% and 24.2% performance overhead compared to IPHC. In summary, IPHC provides consistent performance compared to baseline HC while providing considerable energy saving. The way-based decay scheme suffers from potential performance degradation problems, and the choice of a suitable decay interval varies from workloads.

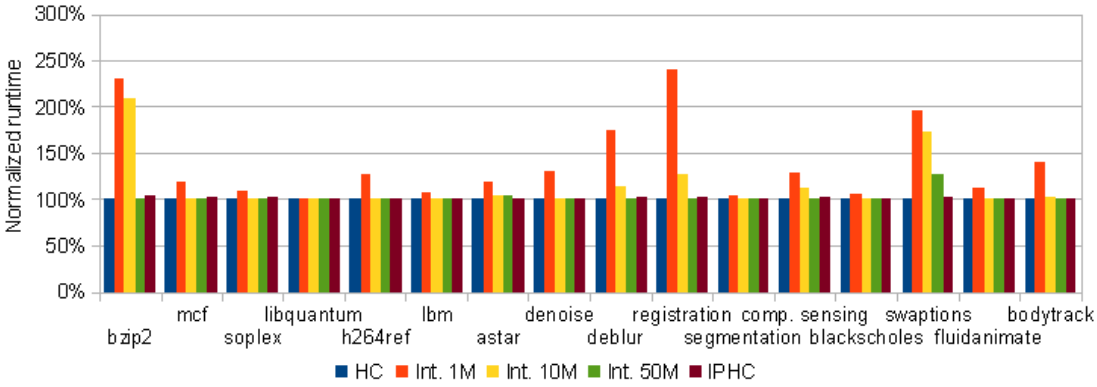


Figure 3.9: Comparison of runtime on two dynamic schemes

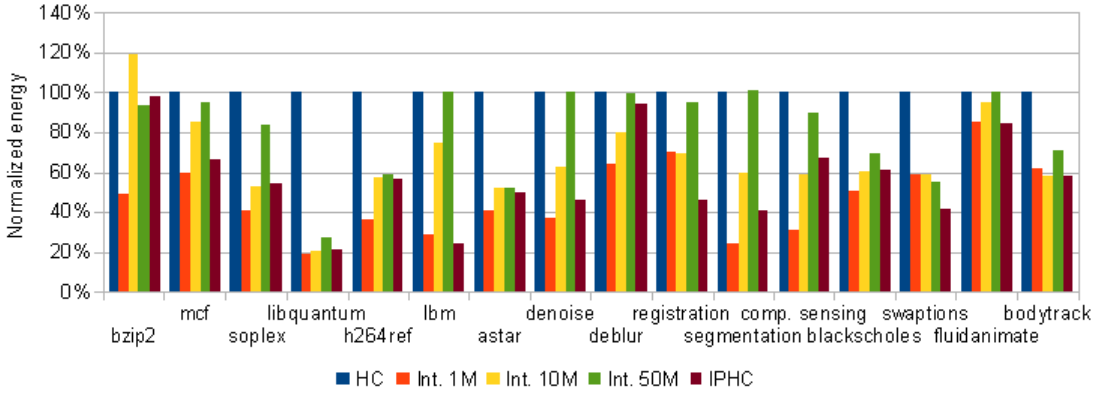


Figure 3.10: Comparison of energy on two dynamic schemes

3.5.3 Endurance Analysis

Table 3.4 shows the endurance comparison between HC and RHC. The lifetime calculation is based on the method from [92] and the write cycle is 4×10^{12} [32]. RHC can achieve from a 1.08X to 3.53X lifetime enhancement on most of the workloads except *bzip*2, *soplex*, *segmentation*, and *swaptions*. For *bzip*2 and *segmentation*, the lifetimes of our scheme are still in a reasonable range. Through the random selection of powered-on/off victims, some write-intensive data blocks may have a chance to be swapped out to main memory, and thus the pressures of the most write-intensive cache blocks can be alleviated. Therefore, our reconfigurable scheme can achieve a reasonable lifetime compared to HC even when available cache ways are decreased due to reconfiguration. However, we observe the non-uniform distribution of write accesses as mentioned in the previous work [32, 92]. A suitable wear-leveling technique is still required to achieve better endurance.

Table 3.4: Endurance comparison of 4MB non-reconfigurable hybrid cache (HC) and 4MB RHC (unit: year)

Workloads	HC	RHC	Workloads	HC	RHC
bzip2	299.92	200.24	g.-deblure	76.68	116.36
mcf	8.40	29.68	registration	12.88	30.68
soplex	4.64	4.6	segmentation	537.32	256.4
libquantum	2.82	4.2	comp. sensing	3.28	3.56
h264ref	22.76	41.88	blackscholes	3.144	5.44
lbm	228.96	253	swaptions	7.36	3.4
astar	16.00	30.08	fluidanimate	107.76	118.56
r.-denoise	53.44	118.8	bodytrack	9.76	10.28

3.6 Related Work

Because of the desirable characteristics on leakage power and density, NVMs have recently been intensively investigated as an efficient alternative for either SRAM in the on-chip caches or DRAM in main memory [165, 154, 65, 143, 107]. In [165], STT-RAM and PRAM are used to implement the lower-level cache. Two types of hybrid cache architectures are evaluated—inter-level and intra-level, in which NVMs are used either as the entire L3 cache or the slow-accessed region in L2 cache. In [154], 3D stacking STT-RAM is used to build a hybrid cache system with SRAM. The capacity for STT-RAM to be used alone in L2 cache is evaluated in [65, 143]. However, none of the prior work has considered dynamic powering on/off of SRAM and NVM arrays to adapt to varied workloads. Recent work shows that the write performance of STT-RAM can be improved by relaxing the retention time of STT-RAM cells. In [115], the authors propose Cache-Coherence-Enabled Adaptive Refresh (CCear) to minimize the number of refresh operations for volatile STT-RAM LLC to reduce refresh power.

Dynamically reconfigurable caches are investigated for pure SRAM caches to either reduce the energy consumption via power gating [137, 173, 175, 98, 29], or provide dynamic flexible support of software-managed memories to the core through a cache line control bit [43, 49]. The key to these approaches is the dynamic assessment of runtime cache pressure. In [137], researchers use a single miss counter to measure the demand of an instruction cache to perform reconfiguration. Missing tags or victim tags are used in [173, 175, 49] to assess the cache pressure. When a cache miss occurs, the tag of the victim block will overwrite the LRU tag of the same set in victim tags and will be marked as the MRU victim tag. If there is a cache miss and victim tag hit, this indicates that a potential hit would occur if the requested block were held in the cache. The authors in [98] use a time-based counter for each cache block,

which will be reset once there is a hit to that block. When the counter maintaining a non-zero status exceeds a given decaying period, the block will be turned off to save leakage. However, none of the existing dynamic reconfiguration schemes have considered hybrid memory technologies. In [29], an industry last-level cache design is proposed while providing reconfigurability through the set-based architecture [137]. In recent work, EECache [42] proposes slice-based power-on/power-off decisions based on three main factors: (1) utilization, (2) hotness, and (3) distribution of dirty cache lines. The proposed data migration policy is used to reduce the miss penalty due to the loss of data in powered-off slices.

To the best of our knowledge, this dissertation is the first work to explore the dynamic cache reconfiguration for hybrid memory technologies in order to reduce the cache energy consumption.

There is also work investigating endurance reduction for the NVMs. In [107, 140], wear-leveling techniques are proposed for a PRAM-based memory system to enhance the lifetime. Recent work in [32] uses periodic set-remapping to distribute the writes among sets in a STT-RAM cache. Another study migrates the write-intensive cache blocks to other cache lines in the same/different cache set or in the SRAM to reduce the average write frequency of the STT-RAM (or PRAM) cache lines [92]. This work is orthogonal and complementary to our proposed reconfigurable hybrid cache designs.

3.7 Conclusions

We propose an energy-efficient last-level cache design—the reconfigurable hybrid cache (RHC). In RHC different memory technologies (SRAM and NVM) are uni-

fied at the same cache level to form a hybrid design, and power gating circuitry is introduced to allow adaptive powering on/off of SRAM/NVM sub-arrays at way level. Experimental results show that the proposed RHC achieves an average 63%, 48% and 25% energy saving over non-reconfigurable SRAM-based cache, non-reconfigurable hybrid cache, and reconfigurable SRAM-based cache, while maintaining the system performance (at most 4% performance overhead).

CHAPTER 4

Static and Dynamic Co-Optimizations for Block Placement in Hybrid Caches for Energy and Endurance

4.1 Introduction

In Chapter 3 we introduced the reconfigurable hybrid cache architecture and provided optimization techniques to dynamically reconfigure hybrid caches for reducing leakage consumption. In addition to the reconfiguration, the use of hybrid caches is still limited by two important factors: endurance and high dynamic write energy on the NVM cells. In this chapter we will provide static and dynamic co-optimization to improve the endurance and reduce the dynamic write energy on NVM cells.

We select the STT-RAM as the NVM cells in hybrid caches. Compared to PRAM, STT-RAM has significantly higher endurance (109 versus 1012 write cycles) and shorter write latency [87], and is much more promising in the last-level cache design [154, 165, 92, 32, 116, 37]. Moreover, due to the intensive writes of caches, hybrid caches consisting of both SRAM and STT-RAM are investigated [154, 165, 92, 37], where the SRAM can accommodate write-intensive data and the STT-RAM can accommodate other data with its dense capacity.

However, the STT-RAM endurance is still an important issue to be considered in the last-level cache design. Although ITRS predicts that the write cycles of STT-RAM will be 10^{15} at 2024 [87], the best available write cycles of STT-RAM are 4 x 10^{12} at present [32]. Suppose we execute *segmentation* [25], a medical imaging application, on a 4GHz CPU with 32 KB L1 cache, 2MB STT-RAM L2 cache continuously, the lifetime of a STT-RAM cache can last only 2.17 years without any optimizations applied. The endurance problem becomes even worse in the multilevel cell (MLC) STT-RAM technology [32]. Block placement optimization is very important to shrink the large endurance gap between STT-RAM and SRAM in a hybrid cache. Recent work considers either static or dynamic schemes to optimize the block placement to reduce the average write frequency to STT-RAM cells, while maintaining the overall performance by making use of higher density of STT-RAM. Some of the proposed approaches were targeted at PRAM, and those ideas can also be applied to STT-RAM with the same objective.

The first category of the prior work uses static schemes. In [90] the authors introduce data migration and recomputation to reduce the write frequency on PRAM main memory. In [119] the partitioning of the application working set into SRAM and PRAM can reduce 79% of the writes to PRAM.

The second category of the prior work uses dynamic schemes. Recent work in [32] uses periodic set-remapping to distribute the writes among sets in a STT-RAM cache. Another set of work migrates the write-intensive cache blocks to other cache lines in the same/different cache set or in the SRAM in order to reduce the average write frequency of the STT-RAM cache lines [92]. In [162], the authors use an access pattern predictor to direct block placement and migration. The LLC write accesses can be categorized into three classes: prefetch-write, demand-write, and core-write.

However, there exists intrinsic limitations in both approaches, which cannot be resolved independently. The static optimization decisions are made at compile time without run-time information; thus the compiler may generate misleading hints to the hardware. On the other hand, pure dynamic optimization uses blocked-based counter structures to learn the memory reference patterns on-the-fly. However, the dynamic scheme lacks a global view of the whole program and has no knowledge of future access patterns.

In this chapter we propose a combined approach in which the static and dynamic optimizations can compensate each other. The compiler tries to provide data placement hints to hardware to reduce STT-RAM write frequency, while the hardware is designed to correct compiler hints based on runtime cache behavior. Experimental results show that the combined scheme improves the endurance by 23.9x and 5.9x compared to pure static and pure dynamic optimizations, respectively, while maintaining similar performance. Furthermore, the system energy can be reduced by 17% compared to pure dynamic optimization since STT-RAM writes are reduced through initial placement from the proposed compiler flow.

4.2 Problem Formulation

In this chapter we assume that the L2 cache is a hybrid cache architecture with 4-way SRAM and 12-way STT-RAM, which is similar to the setting described in [92, 37]. The block-level initial placement and dynamic migration is allowed to place the data blocks in either SRAM or STT-RAM. The initial placement is given by the compiler hints and the runtime cache pressure while the dynamic migration is designed with hardware mechanisms. Our assumptions are described in detail in Section 4.4.1 and

Section 4.4.3 for better illustration of our co-optimization strategy.

The objective of this chapter is to improve the endurance of the hybrid cache and reduce system energy while maintaining performance through the combined scheme. Another meaningful objective is to co-optimize performance and energy while under the endurance constraint. A storage-efficient way to monitoring the endurance of each cache block is required under the second scenario. The discussion of this formulation is not included in this dissertation but may be worthwhile for further investigation.

4.3 Motivational Examples

In this section we use real-life examples to illustrate how pure static optimization and dynamic optimization may produce sub-optimal block placement decisions in a hybrid cache design.

The deficiency of pure static optimization comes from the fact that the runtime write frequencies of L2 cache (which is the hybrid last-level cache in our evaluated system) blocks are input-dependent, which cannot be fully obtained offline. First, the input data may change the control flow of the program, and this will change the write frequency of the data that are affected by the control flow variation. Second, since part of the writes to the L2 cache come from the write-back operations from the L1 caches, the compiler cannot accurately capture the L2 cache write behavior with the existence of the L1 caches. For example, given the LRU replacement policy, the data which is written fewer times in the code may be frequently evicted by the L1 cache and behaves much more write-intensive in the L2 cache than other data which are written more frequently. Note that the lifetime of the STT-RAM mainly depends on the peak write count of all the cells [92, 32]. Even static optimizations

can reduce the total STT-RAM writes compared to dynamic optimizations via global optimization; the potential mis-predictions can still severely degrade the STT-RAM lifetime, since there is no dynamic scheme to migrate mis-predicted write-intensive blocks into SRAM. This causes a large peak write count to this cell. As an example, Table 4.1 shows the STT-RAM cell write count distribution of the segmentation application [25] for both of the pure static [119] and the pure dynamic [92] schemes. The peak write count of static optimization is significantly larger than that of the dynamic one.

Table 4.1: STT-RAM write count distribution

# writes	0-100	100-200	200-300	300-400	400-1000	1000-5000	> 5000	max
static	12262	6	5	0	0	10	5	5470
dynamic	12207	38	16	27	0	0	0	395

The deficiency of dynamic migration comes from the fact that it lacks the future memory access information and highly relies on the application to exhibit a bipolar L2 write frequency pattern—the L2 cache blocks are either rarely written or intensively written. Then the write-intensive blocks can swap their places with the rarely written blocks through dynamic migration. However, based on our observation, not all the applications have such characteristics, especially in the three medical imaging applications [25] used in our study. As shown in Figure 4.1, most of the blocks are uniformly written 2-3 times. Under this circumstance, if the migration threshold is set to be higher than 3, then there will be little migration; both the SRAM and STT-RAM will be evenly written based on the LRU scheme, which may impair the endurance. However, even if the migration threshold is set to be 2 or 3, the migration for most blocks does not save any writes, since most blocks behave similarly. A correct approach is to place all these streaming accessed blocks into the SRAM, which

can be obtained via static optimization in the compiler. After that, the expensive writes on STT-RAM can be significantly reduced, and thus dynamic energy can be reduced.

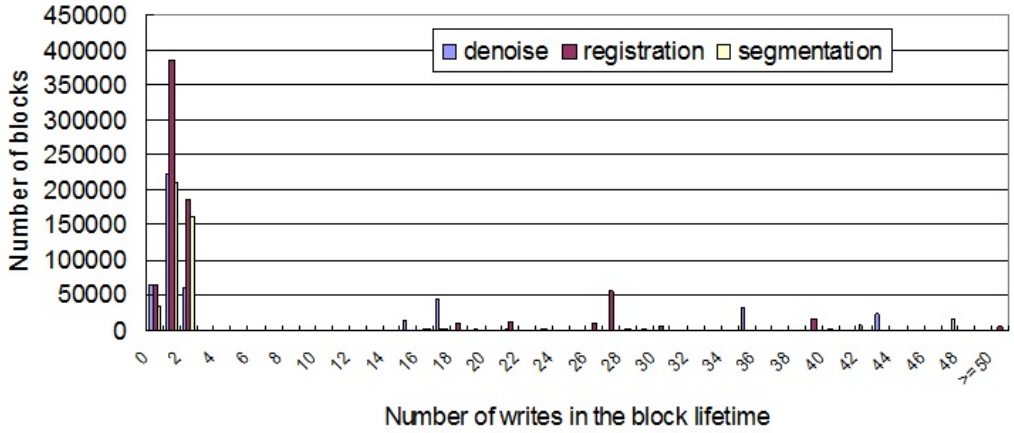


Figure 4.1: Write frequency distribution of the L2 cache blocks

To overcome these limitations, while taking advantage of both static and dynamic schemes, we use a combined strategy: the compiler tries to guide the hardware in order to rapidly achieve the desired placement, while the hardware corrects the compiler hints based on the run-time cache behavior. To the best of our knowledge, we are the first one to take such a hybrid approach.

4.4 The Combined Approach

4.4.1 Compiler Support

In this dissertation we develop an automatic compilation flow to generate data placement hints for each memory reference. Here, we assume that LRU replacement policy is used, and L2 is an inclusive cache with the same block size of L1, which is widely

used in modern processors because of easy coherence implementation.

Similar to [119], our compiler tends to place write-intensive references into SRAM and non-write-intensive data into STT-RAM. Based on our inclusive cache assumption, the write accesses on L2 STT-RAM cells occur in only two situations: (1) L1 dirty evictions due to L1 cache replacement, and (2) L2 cache replacement. However, the work in [119] assumes there is no cache in the memory system, and thus does not consider the effect of higher-level (L1) cache on the memory access behavior. Figure 4.2 shows an example code and its corresponding memory access behavior. We can find that both arrays A and B are written twice. However, since array A is more frequently accessed and can be kept in the L1 cache (we assume LRU replacement policy is used here), neither of the two writes falls into the L2 cache. On the other hand, since array B is evicted from the L1 cache before its next access, one write-back operation will be issued into L2.

loop 1: A[i] = …; B[i] = …; (*write array A and B in L1*)
loop 2: … = A[i] …; (*read array A in L1*)
loop 3: … = C[i] …; (*array B is evicted into L2*)
loop 4: A[i] = …; B[i] = B[i]…; (*write array A and B in L1*)

Figure 4.2: One sample code and its memory access behavior

To capture this effect, we use the concept of *memoryreusedistance* (MRD) [64], which equals the total size of unique data elements accessed between two references to X. A larger memory reuse distance of X implies that X will not be accessed in the near future, and thus X is more likely to be evicted from the L1 cache.

$Definition1$: For a write operation w of memory instruction X, assuming the future access sequence of X is w, r_1, r_2, ..., r_n, w', etc., (r and w correspond to

read and write operations), w is called an L1-writeback write if one of the following conditions is satisfied: (1) there exists $MRD(r_i, r_{i+1})$ ¿ $dist_{L1}$ (i = 1, ..., n-1), (2) $MRD(r_n, w')$ ¿ $dist_{L1}$, (3) $MRD(w, r_1)$ ¿ $dist_{L1}$. ($dist_{L1}$ is the average reuse distance to keep X in L1 cache.)

From Definition 1 we can see that if the memory reuse distance between two accesses into a dirty data X is larger than a threshold value $dist_{L1}$, the compiler will treat the first write (w) to X as a $L1 - writeback$, since X will be written back into L2. The other set of L2 write accesses comes from L2 misses, and data are written into L2 from main memory. Here we use $dist_{L2}$ to indicate the average reuse distance to keep X in L2 cache. For two adjacent accesses to X, if the memory reuse distance between them is larger than $dist_{L2}$, the compiler will treat the second access as an L2 miss, which will introduce one L2 write operation.

In our flow, we provide a 2-bit compiler hint for each memory instruction to guide its data placement in L2 cache. For each access to reference X, we count the total number of future L2 writes to X including both $L1 - writeback$ writes and L2 misses. If there are frequent L2 writes, our compiler will generate hint "01" for X. On the other hand, hint "00" is generated to indicate that X will not be written frequently. For those accesses that the compiler cannot analyze accurately (e.g., due to unknown loop bound), hint "1x" is generated, and the data placement is controlled by hardware. Note that a memory instruction in a regular loop is accessed multiple times with repeated access patterns [50]; therefore we can apply the same hint to all the accesses to it.

It should be noted that the compiler just tries its best to predict the write frequency. The value of $dist_{L1}$ and $dist_{L2}$ can be obtained from profiling on representative input or set to a fixed value by default. However, it is not feasible to profile

all input sets. In this dissertation a conservative approach is used to set $dist_{L1}$ to L1 set associativity. This can ensure that data X will not be evicted from L1 between two accesses with reuse distance less than $dist_{L1}$. Since the compiler cannot make an optimal decision without knowing the runtime cache behavior, these generated hints may not be followed in the hardware. We will discuss our combined scheme in Section 4.4.3.1.

4.4.2 Compiler-Hardware Interface

In our implementation, the compiler passes the hints to the hardware through setting two bits in the 32-bit instruction code. We assume that there are two extra bits in each memory instruction that the compiler can use to assist the run-time cache block replacement. Existing architectures already use these kinds of extra bits in the instruction, such as the $prefetch$ and $evict - next$ instruction in the Alpha 21264. We believe that in most architectures, the increasing speed gap between memory and processor will justify the inclusion of additional bits in the instruction code to facilitate the reduction of this gap.

Once a memory reference instruction is executed, if this is a L1 cache hit, these 2 bits will be discarded. If this is a L1 miss, then the bits will be passed to the L2 cache controller, if this is a L2 hit, then these 2 bits will be discarded; if this is a L2 miss, these 2 bits will be used as hints for the initial placement of the new block.

4.4.3 Hardware Support

In this chapter we use a 1MB 16-way hybrid cache including a 4-way SRAM data array and a 12-way STT-RAM data array similar to the configurations used in [154, 92, 37].

The asymmetric configuration is chosen since smaller SRAM contributes less leakage, while the bigger STT-RAM provides the advantage of higher density. We use separate replacement units on SRAM and STT-RAM in order to perform block replacement in these two arrays independently. We also introduce a global replacement unit for them in order to perform a global replacement among them if required. All of these replacement units use LRU policy.

4.4.3.1 Capacity-Pressure and Compile-Hints-Based Initial Placement

The initial block placement decision is made based on both the compiler hint and also the SRAM/STT-RAM capacity pressure monitored in the hardware.

Before discussing the decision-making process, we first show our capacity pressure assessing hardware. To assess the SRAM/STT-RAM capacity pressure, we introduce two additional hardware structures: *missingtags* (MTs) and MT counters. The proposed structures are similar to the missing tags [173] and victim tags [49]. MTs and MT counters are integrated with the tag array design, as shown in Figure 4.3. In addition to the original 4-way SRAM tag array and the 12-way STT-RAM tag array, 4-way SRAM MTs and 4-way STT-RAM MTs are introduced. Moreover, for each cache set, there are a SRAM MT counter and a STT-RAM MT counter, and these counters indicate the capacity pressure of the SRAM and STT-RAM portions in that cache set, respectively. Note that the sizes of MTs are the same for both SRAM and STT-RAM arrays to provide similar pressure-monitoring criterion.

We use the SRAM MT to illustrate the MT and MT counter functionality, and the STT-RAM MT and MT counter work in the same way. When a cache miss occurs in the SRAM, the tag of the victim block will overwrite the LRU tag in the same

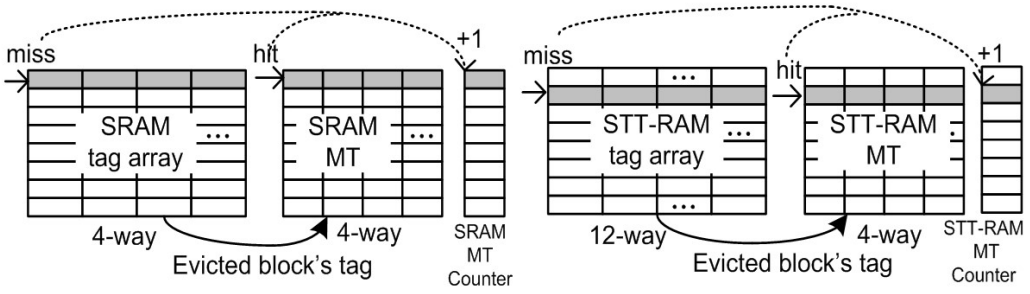


Figure 4.3: SRAM and STT-RAM missing tag and counter

set in SRAM MTs and be marked as most recently reused tag. If there is a miss in the SRAM array and there is a hit in the SRAM MTs, this indicates that a potential hit will occur if the requested block were placed in STT-RAM. Then the SRAM MT counter in the corresponding cache set is incremented by one.

We use an interval-based assessing approach, i.e., the value of the MT counters in the current interval will be used to guide the initial placement in the next interval. Considering the less-frequent access to the L2 cache, the interval length cannot be too short, but it can also be too long in terms of timely assessment. In this dissertation we set it to be 10K cycles. At the end of each 10K cycle, the value of all the MT counters will be evaluated to fill the *capacitypressuretable* (CPT). The number of entries of CPT equals the number of sets in the cache, and each entry contains two bits: the SRAM capacity pressure and the STT-RAM capacity pressure of that cache set. If the value of the SRAM (STT-RAM) MT counter of a cache set is greater than a threshold, then the SRAM (STT-RAM) bit for that set in the CPT is set to 1 (*high*), and otherwise set to 0 (*low*). Then all the MT counters are reset to 0. In the next interval, the CPT is accessed together with the compiler hints to decide the initial block placement.

Given the capacity pressure from the CPT and also the compiler hints, the L2 cache controller makes the initial placement as shown in Table 4.2. If a block is going to be placed in SRAM (STT-RAM), then the LRU replacement unit of SRAM (STT-RAM) will be triggered to evict the victim in that cache set of SRAM (STT-RAM). If a block is going to be placed globally, the global LRU replacement unit is triggered to evict the LRU block of all the SRAM and STT-RAM cache lines in that cache set.

Table 4.2: Initial placement decision based on compiler hints and SRAM/STT-RAM capacity pressure

Capacity pressure		**Compiler hint**		
SRAM	STT-RAM	infrequent writes	frequent writes	unknown
High	**Low**	STT-RAM	STT-RAM	STT-RAM
Low	**High**	SRAM	SRAM	SRAM
High	**High**	STT-RAM	SRAM	Global
Low	**Low**	SRAM	SRAM	SRAM

4.4.3.2 Write-Frequency-Based Dynamic Migration

As pointed out in Section 4.4.1, the compiler hints are not absolutely accurate due to the input variation and the L1 cache impact. Moreover, according to Section 4.4.3.1, when capacity pressure unbalance occurs, blocks may be initially placed in the less-intensive used portion of the hybrid cache, instead of based on the write-frequency of the block itself, as shown in Table 4.2. Thus it is possible that a block is incorrectly initially placed.

We use dynamic migration to correct the initial placement by migrating the actually write-intensive STT-RAM data blocks to SRAM. We use the dynamic migration

scheme similar to [92], which is briefly described as follows. Each L2 cache block is associated with a saturate 2-bit write counter to indicate the number of writes during its on-chip lifetime. If the write counter of a STT-RAM block saturates (three writes), the migration unit will check the write counters of the SRAM blocks in the same cache set. If there is any counter that is less than 3, then the corresponding SRAM block is swapped with that STT-RAM block. After that, all the write counters in this cache set are reset to 0. If the counters of all the SRAM blocks in a set are saturated, no migration will be performed. Therefore, the possibility that another write-intensive block could be swapped from the SRAM back to STT-RAM is avoided.

In sum, in our combined approach, if the compiler provides correct hints, the hardware can use them to rapidly achieve correct block placement. If the compiler makes mis-predictions, the hardware corrects the compiler hints as shown in Table 4.3. Note that all the hardware corrections are automatically triggered by our introduced hardware counters.

Table 4.3: Hardware corrections to the compiler mistakes

Compiler mis-predictions			**Hardware corrections**		
a	**b**	**c**	**i**	**ii**	**iii**
X					X
	X			X	
		X	X		X
X		X	X		X
X	X			X	X

Compiler mis-predictions: (a) Mis-predicts some write-intensive blocks as non-write-intensive. (b) Generates larger percent of non-write-intensive blocks than there

actually is. (c) Generates larger percent of write-intensive blocks than there actually is.

Hardware corrections: (i) Distributes blocks to STT-RAM. (ii) Distributes blocks to SRAM. (iii) Migrates write-intensive blocks from STT-RAM to SRAM.

4.5 Evaluation Methodology

4.5.1 Compilation and Simulation Infrastructure

The compiler support for hint generation is implemented based on the LLVM compiler infrastructure [83]. Omega library [84] is used in this flow to perform memory dependency analysis. Given a source program written in C/C++, we parse it into LLVM IR using LLVM's frontend. All standard optimizations in O3 are applied. Our hint-generating flow is invoked as a pass on the optimized LLVM intermediate representation (IR) code and will automatically generate data placement hints for each load/store instruction. We also modify LLVM backend to emit hint-included load/store instructions in the final assembly code. A potential issue of this LLVM frontend analysis is that some load/store instructions cannot be captured in the IR level. For example, the loads/stores in pre-compiled library functions cannot be analyzed under this framework. Moreover, the loads/stores from the operating system cannot be analyzed during compile time. Therefore, a hardware support mentioned in Section 4.4.3 is required to provide better optimization.

We extend the full-system cycle-accurate Simics [124] and GEMS [126] simulation platforms to model the proposed hardware support. The system configurations of SIMICS/GEMS are shown in Table 4.4. We obtain the energy data of the SRAM

Table 4.4: Simics/GEMS simulator configurations

Core	Sun UltraSPARC-III Cu processor core
L1 Instruction/ Data Cache	32KB, 2-way set-associative, 64-byte block, 2-cycle access latency, pseudo-LRU
L2 Cache (Hybrid cache)	1MB, 16-way set-associative (4-way SRAM, 12-way STT-RAM), 64-byte block, access latency: 10-cycle for SRAM, 11-cycle (read) and 30-cycle (write) for STT-RAM
Main Memory	4GB, 320-cycle access latency

array and MTs/MT counters through Cacti 6.5 [131] with 32nm process technology at 330K. The energy data of the STT-RAM array are obtained from NVSim [169]. Table 4.5 shows the energy model we use in our evaluation. Note that the low leakage cells (*itrs-lstp*) are used in SRAM data array and tag array. For peripheral circuitry, we use high performance cells (*itrs-hp*) to optimize performance and area. Note that we also try to implement the peripheral circuitry with low leakage cells for further leakage minimization. However, we observed that considerable area overhead may arise since the width of an *itrs-lstp* transistor needs to be large enough to provide enough current for STT-RAM write operation.

Table 4.5: Energy/power data of the evaluated hybrid cache

	Read energy	**Write energy**	**Leakage power**
SRAM (4-way)	0.0603nJ	0.0603nJ	15.017mW
STT-RAM (12-way)	0.231nJ	1.306nJ	11.173mW
MTs (8-way)	0.0020nJ	0.0020nJ	2.805mW

4.5.2 Benchmarks

Our testbenchs consist of eight benchmark applications, which have been carefully chosen to represent-memory intensive algorithms in the fields of data processing, massive communication, scientific computation and medical applications. The benchmark applications include three memory-intensive applications from SPEC2006 [88] (*bzip2*, *mcf* and *lbm*) and five applications from the medical imaging domain [25].

4.5.3 Reference Schemes

To demonstrate the effectiveness of our combined scheme (*combined*), we compare it to two representative prior approaches:

Pure static optimization (*static*): The hardware will strictly follow the compiler-generated block placement hint. The compiler hints are generated based on the approach proposed in [119], and we further take the effect of L1 cache into consideration using the techniques discussed in Section 4.4.1.

Pure dynamic optimization (*dynamic*): We use the dynamic migration scheme proposed in [92]. Our dynamic migration scheme described in Section 4.4.3 follows the same migration threshold used in [92]. The migration threshold is set to three. There is no compiler hint in this scheme.

Note that the energy overhead of MTs and MT counters is only applied on the *combined* scheme.

4.6 Experimental Results

4.6.1 Endurance

In this work, we assume that the maximum write cycles of a STT-RAM cell is 4 x 10^{12} [32]. We assume that a workload continuously runs on the system. To model the endurance in a more sophisticated way, one can provide a loading factor, which is the percentage of the overall runtime occupied by the workload. The lifetime is measured from the start of the simulation until the first STT-RAM line becomes defective, which is similar to the estimation methodology proposed in [92, 32].

Figure 4.4 demonstrates the lifetime which is normalized to the *static* scheme. The *static* scheme typically performs the worst among the three schemes (up to 1.2x - 148x worse than the *combined* scheme). This is because once a compiler mis-predicts a write-intensive block as a non-write-intensive one and places it into the STT-RAM, this block will be intensively written, and there is no dynamic migration to mitigate it. The lifetime of the STT-RAM mainly depends on the peak write count of the cells. The exceptions are *fft*, *lbm* and *denoise*, where the program only has negligible input-variation, so the *static* scheme can have a longer lifetime than the other two schemes. Note that the *static* scheme can only reduce the total writes instead of the peak write count among all blocks, as shown in Figure 4.5. Therefore, the *static* scheme is the worst in terms of endurance, but it can save STT-RAM write energy, which is discussed in Section 4.6.2.

With the dynamic migration to average the writes to STT-RAM blocks, the *dynamic* scheme achieves up to 14x improvement of lifetime compared to the *static* scheme. However, the reduction of the peak write count of STT-RAM is accompanied

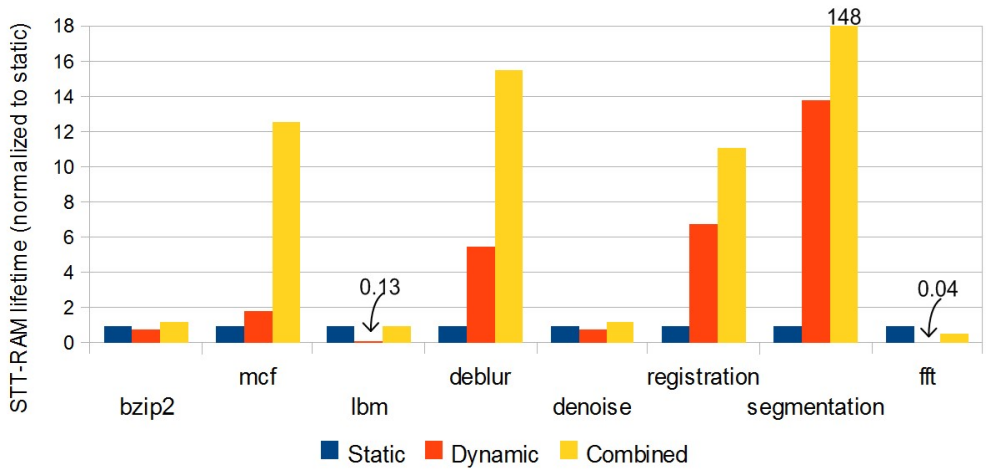


Figure 4.4: Comparison results of STT-RAM lifetime

with the cost of much more total STT-RAM writes, since it lacks global information to reduce the total STT-RAM writes. Figure 4.5 shows that the *dynamic* scheme has 1.6x - 36.6x more STT-RAM writes than the *static* scheme. In cases of *fft* and *lbm*, the data blocks are all uniformly written less than 3 times on either SRAM or STTRAM. Therefore, there is little migration in the *dynamic* scheme, and it has a lifetime which is only 4% - 13% of that of the *static* scheme.

The *combined* scheme has a 1.6x - 14.7x lifetime compared to that of the *dynamic* scheme. By following the correct compiler hints, the *combined* scheme rapidly achieves the optimal block placement without additional migrations, especially in the cases where most of the blocks are uniformly written less than two or three times, as shown in the motivational examples in Section 4.3. This can save both the peak write count and also the total writes of the STT-RAM. Although the *combined* scheme has 0.8x - 4.1x more total STT-RAM writes than *static*, it achieves 1.2x - 148x lifetime due to averaging the writes to the STT-RAM cells (except *fft* where *static* has a 1.8x longer lifetime than *combined*).

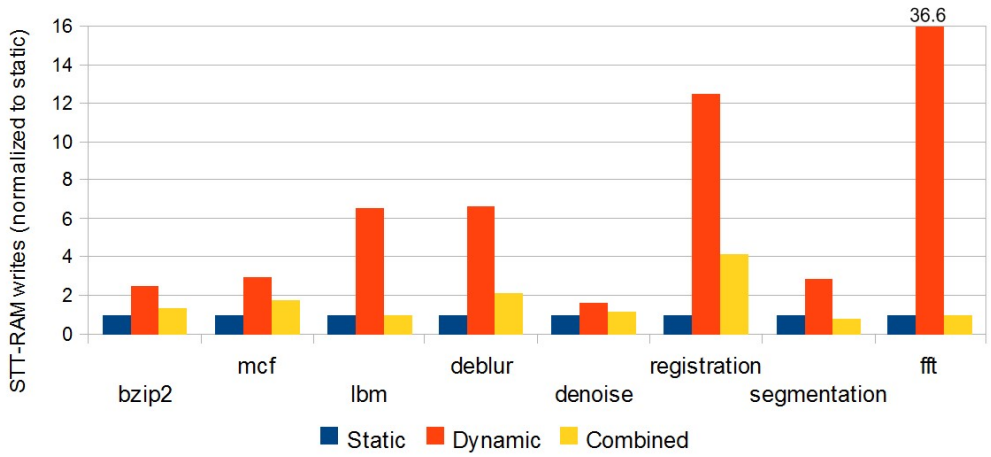


Figure 4.5: Comparison results of total STT-RAM writes

4.6.2 Energy

Figure 4.6 shows the distribution of hybrid cache (L2 cache) energy that is normalized to the *static* scheme. The leakage consumption of three schemes is similar. This is because leakage is proportional to program runtime and the runtime (as shown in Figure 4.7) of the three schemes is similar. Therefore, the key factor that influences the system energy is the L2 STT-RAM dynamic energy. The *static* scheme has the least energy, because the reduced STT-RAM writes (as shown in Figure 4.5) bring in considerable dynamic energy savings. Without the hints of initial placement, a large number of writes arises in the *dynamic* scheme, leading to 9% - 80% energy overhead (38% overhead on average) compared to the *static* scheme.

The combined scheme achieves similar energy consumption to that of the *static* scheme (7% - 20% energy overhead, 11% overhead on average) and outperforms the *dynamic* scheme (2% - 39% energy reduction, 17% reduction on average). Note that the energy overhead of the *combined* scheme comes from both the leakage of the

introduced MTs and the extra dynamic STT-RAM writes energy compared to the *static* scheme.

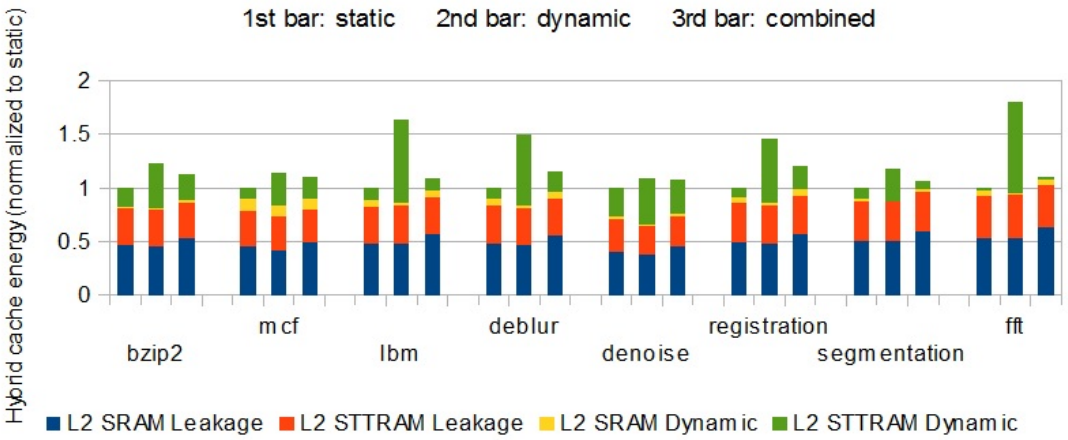


Figure 4.6: Comparison results of hybrid cache energy

4.6.3 Performance

Performance is measured by the runtime of a workload (in terms of number of clock cycles obtained from our simulation infrastructure). Figure 4.7 shows the comparison results of runtime that are normalized to the *static* scheme. Since the total cache size for the three schemes is the same, the runtime does not vary significantly. The differences among the three schemes come from how efficiently they make use of the aggregate capacity of both SRAM and STT-RAM to reduce the cache misses. The *dynamic* scheme typically performs the best due to the equivalent initial placement to SRAM and STT-RAM, which best utilizes the STT-RAM capacity. As mentioned in Section 4.4.3.1, the compiler may generate larger write-intensive data on SRAM due to the input variation, thus imposing high-capacity pressure on the SRAM resulting in high cache misses (as shown in Figure 4.8). Therefore, the *static* scheme performs

-1% to 9% worse than the *dynamic* scheme (with a -1% to 30% increase in the L2 cache misses). The only exception is *fft* where *static* outperforms *dynamic* due to accurate compiler hints.

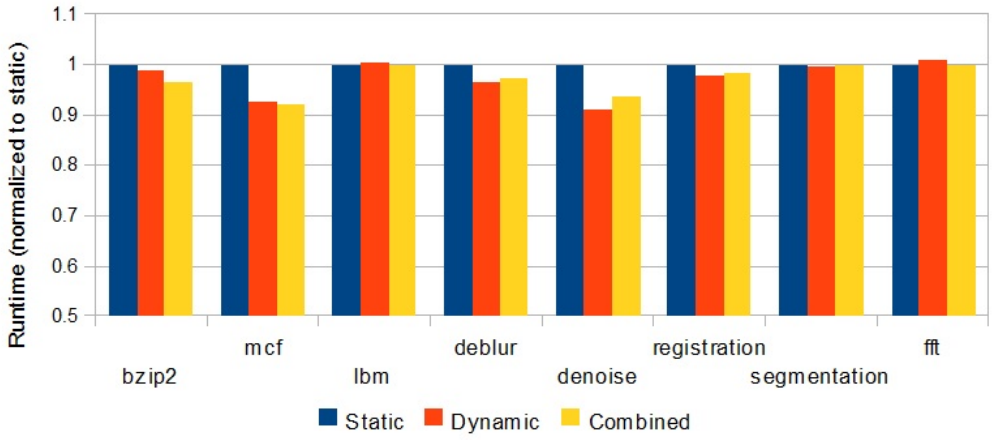


Figure 4.7: Comparison results of runtime

In the *combined* scheme, the hardware can automatically correct the compiler mis-predictions as discussed in Section 4.4.3. Therefore, it achieves a runtime similar to that of the *dynamic* scheme (within a -5% to 5% variation). These analyses are summarized in Table 4.6.

Table 4.6: Comparison summary of the experimental results

	static	*dynamic*	*combined*
Endurance	worst	fair	best
Performance	fair	best	best
Energy	best	worst	$\tilde{\text{b}}$est

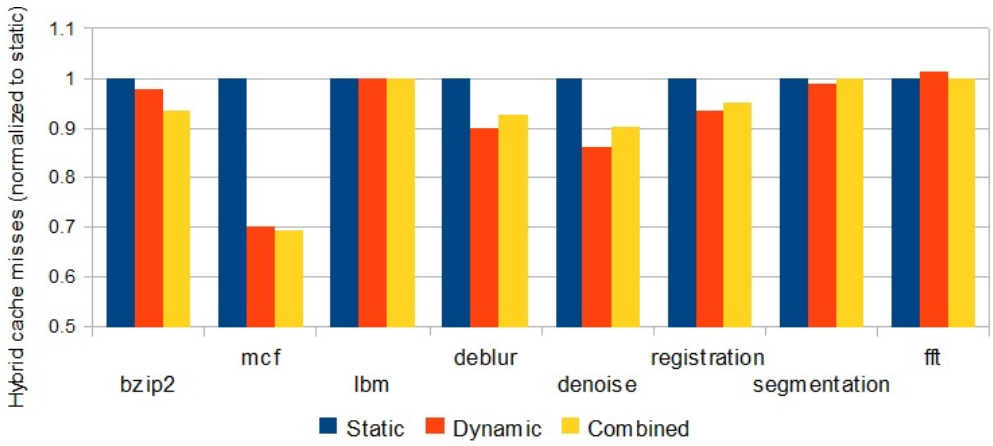


Figure 4.8: Comparison results of hybrid cache misses

4.6.4 Different Bit Widths of Write Counters

We perform the sensitivity analysis on different bit widths of the write saturation counters in our proposed *combined* scheme. The write counters are used for dynamic migration to improve the endurance. We justify that 2-bit counters are adequate enough for write counters. Figure 4.9 shows that the lifetime can be significantly enhanced in *mcf*, *deblur*, *registration*, and *segmentation* when 2-bit counters are applied. The 1-bit counters are inefficient since the SRAM blocks in the same set may easily saturate and thus prevent the migration of write-intensive STT-RAM blocks into SRAM ones. For the rest of workloads, the lifetime is insensitive to the bit width. According to our experimental results, the bit widths of write counters are insensitive to both energy and runtime among all workloads (less than 1% difference). In terms of energy, the only exception is *mcf*, where most of write-intensive blocks cannot be migrated into SRAM when 1-bit counters are used. Therefore, the STT-RAM energy increases by 15% in the 1-bit counters case compared to the others (2- to 5-bit). For performance, it is insensitive to the widths of write counters since performance

is maintained through cache capacity pressure monitoring, as described in Section 4.4.3.1.

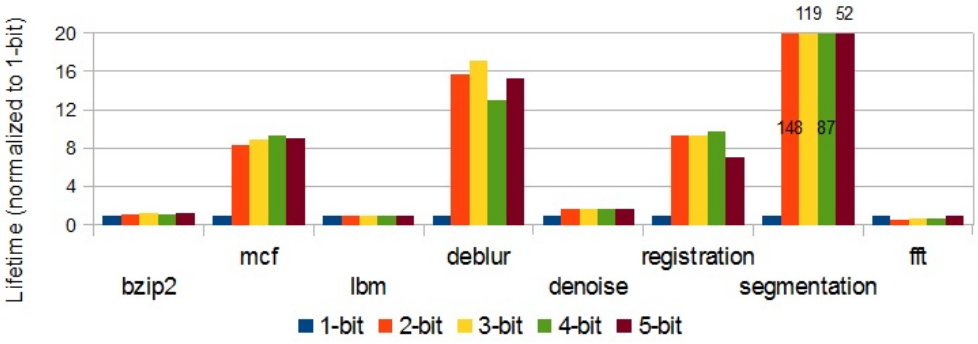


Figure 4.9: STT-RAM lifetime over different bit widths

4.7 Conclusions

In this chapter a combined static and dynamic scheme is proposed to optimize the block placement in a hybrid SRAM and STT-RAM cache, so that endurance and energy are co-maximized. The compiler tries to guide the hardware to rapidly achieve the desired placement, while the hardware corrects the compiler hints based on the runtime cache behavior. Experimental results show that the combined scheme improves the endurance by 23.9x and 5.9x compared to pure static and pure dynamic schemes, respectively, while maintaining similar performance. Meanwhile, the system energy can be reduced by 17% compared to the pure dynamic scheme.

CHAPTER 5

Accelerator-Rich Architectures and ARAPrototyper

5.1 Introduction

The scaling of conventional multicore processors has been limited by the power and utilization walls because most portions of future chips cannot be simultaneously powered up. This unpowered portion is referred to as dark silicon [67, 160]. Customized acceleration [30, 31, 45, 77, 79, 102, 118, 139, 147, 160] has proved to be one of the most promising solutions to address this issue. Compared to conventional general-purpose processors, these customized accelerators can provide orders-of-magnitude performance improvement and energy savings. Recently, more accelerators are being integrated into the general-purpose processors; this new architecture is referred to as the accelerator-rich architecture (ARA) [53, 54, 122]. Due to the significant performance and energy gains, numerous ARA efforts have been reported from both academia (such as research in [53, 54, 122]) and industry (such as the IBM wire-speed processors in server markets [72] and the Intel video streaming processors in consumer markets [104]).

However, the accelerator-rich architectures (ARAs) are still in the early stages of

development and many design issues, especially system-level issues, remain unclear and difficult to evaluate. Examples include efficient accelerator resource management, design choices of interconnect between accelerators and scratchpad buffers, interconnect between scratchpad buffers and LLC or DRAM, efficient address translation support, etc. Therefore, a research platform that can enable rapid ARA design space explorations will be extremely useful.

In prior work, there are two major approaches used to explore the ARA design spaces: 1) full-system simulation [52, 53, 54, 56, 77, 102], and 2) FPGA prototyping [20, 34, 41, 45, 71, 118]. As shown in Figure 5.1, full-system simulators are very flexible when changing configurations and require little development effort to conduct design space explorations. However, the simulation time is very long and usually three to four orders-of-magnitude slower than native execution. On the other hand, FPGA prototyping provides rapid evaluation from real silicons, and it has gained increased attention. An FPGA prototype is a realization of the targeted ASIC design, which allows users to run real-life applications on the prototype at native speed and helps developers to verify the robustness of the design before taping out a chip. However, the tedious efforts for existing FPGA prototyping flows have impeded the wide adoption of FPGA prototyping for architectural design space exploration. The goal of the ARAPrototyper is to reduce the prototyping efforts to efforts that are manageable and enable both rapid prototyping and rapid evaluation/verification for ARAs.

The major burden of FPGA prototyping for full-system evaluation involves significant design, implementation, and verification efforts. A robust FPGA prototype developed from scratch usually needs a very long development cycle because it requires a wide range of background knowledge, such as hardware accelerator design, system software stack support (including drivers), and application programming inter-

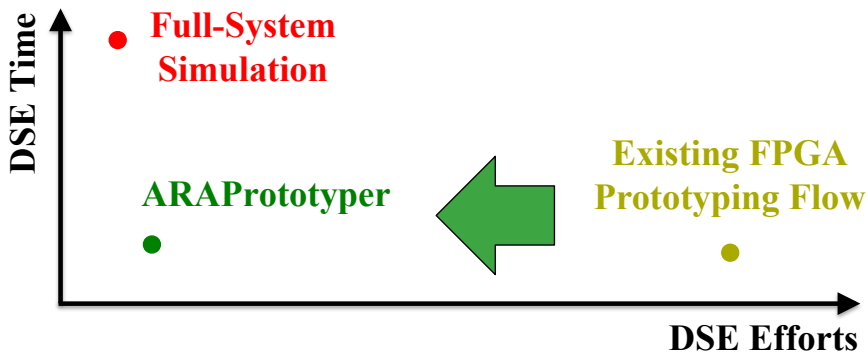


Figure 5.1: Position of ARAPrototyper: rapid prototyping and evaluation for ARA design space exploration (DSE).

faces (APIs) design. Existing FPGA prototypes, like LegUp [28, 27, 71, 78], CoRAM [46, 47], and PARC/ARACompiler [34, 41], take years of engineering efforts for initial development and continuous improvement. An FPGA prototype developed for architectural design space exploration purposes imposes more challenges. First, architects usually want to explore different designs of ARAs or improve their ARAs in an incremental way. To reduce their burden, we should design our ARAPrototyper such that our baseline ARA prototype is highly reusable and customizable to avoid rebuilding the system from scratch. Second, users may want to add their own accelerators into the reusable baseline prototype for system-level evaluation, which stil requires hundreds of lines of HLS code simply for integration in state-of-the-art FPGA prototyping flows—such as our prior efforts, PARC [34] and ARACompiler [41]. Therefore, a decent automation flow with a clean customization interface should be provided so that users can change a few lines of code and push a button to generate their own ARAs.

In this chapter we present ARAPrototyper, a prototyping flow to enable rapid design space explorations for ARAs in native execution time. We choose the modern

Xilinx Zynq SoC [89], which is composed of a dual-core ARM Cortex-A9 CPU and FPGA fabrics,[1] as our underlying prototyping platform. Recently, Xilinx proposed the SDSoC development environment [166] to enable an automated design process for the embedded system. However, it still cannot satisfy our need for rapidly prototyping complex ARAs. To reduce the prototyping efforts for ARA design space explorations, we provide the following features in ARAPrototyper.

1. We develop a reusable and highly customizable baseline prototype for users to evaluate the performance of their ARAs. First, a shared memory architecture has been provided as highly parameterized hardware templates in the baseline prototype. Users can easily configure the interconnect topology between the accelerators and buffers, the interconnect topology between buffers and LLC or DRAM, coherency choice at LLC or DRAM, and TLB (translation-aside buffer) sizes in the ARA specification file without writing RTL or HLS codes. Second, to gain more insight into the performance evaluation, we add a few performance counters at the accelerator side to monitor DRAM and TLB accesses. We also leverage the existing performance counters on the ARM CPU. These can significantly reduce system design efforts and improve the quality of evaluation.

2. To further reduce the efforts of the accelerator design, we support the integration of accelerators that are written in high-level synthesis (HLS) into our ARAPrototyper. More importantly, we provide a clean accelerator integration interface for users to integrate their own accelerators by abstracting away common functionalities such as issuing memory access requests and invoking address

[1] According to the Xilinx UltraScale MPSoC roadmap [167], the next generation of Zynq boards will include a quad-core ARM CPU and ultra-scale FPGA fabrics which will enable the design space explorations for even larger-scale ARAs.

translations. Users just have to specify a few parameters and invoke the computation kernels of their own accelerators. The whole flow to integrate users' own accelerators with their customized ARA prototype is highly automated.

3. We provide a system software stack that supports users in compiling and running their applications seamlessly on their customized ARA prototype. For users to quickly develop their applications that use the accelerators, we abstract the accelerators as software libraries and provide them with user-friendly C/C++ APIs to manipulate accelerators.

To demonstrate the above design space exploration capability of the ARAPrototyper, we choose the medical imaging pipeline [25] as our main application domain for case studies. In order to illustrate the manageable prototyping efforts, we further integrate existing HLS-synthesisable accelerators from the widely used accelerator benchmark suite MachSuite [144] into ARAPrototyper. Only a few lines of code (LOCs) are needed for the integration compared to hundreds of LOCs in recent ARA prototyping work such as PARC [34] and ARACompiler [41]. Finally, we also compare the evaluation time of ARAPrototyper to that of the state-of-the-art full-system ARA simulator PARADE [52] by running a set of common medical imaging applications with different input sizes. ARAPrototyper achieves a 4,000X to 10,000X faster evaluation time. We believe that ARAPrototyper can be an attractive alternative to current full-system simulators for rapid ARA design and evaluation. In summary, this dissertation makes the following contributions:

1. Rapid FPGA prototyping for ARA design space explorations by providing a highly customizable baseline prototype with performance counters, a clean interface and automation flow to integrate the users' own HLS-synthesisable ac-

celerators, a system software stack and accelerator APIs to quickly develop applications that can run seamlessly on the prototype.

2. Rapid evaluation of ARA designs in native execution time, which is about 4,000X to 10,000X faster than the state-of-the-art full-system ARA simulator PARADE.

3. Case studies demonstrating ARAPrototyper's capability for a wide range of ARA design space explorations, manageable prototyping efforts, and rapid evaluation time.

5.2 Background and Motivation

Table 5.1 summarizes the evaluation methodologies in existing accelerator-related research. Basically, we can divide them into two major categories: simulation-based evaluation and FPGA prototyping-based evaluation.

The simulation methodologies can be further divided into the following four categories: 1) pre-RTL simulation [149], 2) RTL simulation [122, 147], 3) cycle-accurate simulation [79, 139, 160], and 4) full-system cycle-accurate simulation [52, 53, 54, 56, 77, 102]. First, except for the pre-RTL simulation, all other simulations take a very long evaluation time that is orders-of-magnitude slower than the native execution. Second, the pre-RTL simulator Aladdin [149] uses dynamic data dependence graphs to model an accelerator, where the model depends on the input changes. More importantly, Aladdin only simulates the accelerator itself and lacks integration with full-system simulators to enable system-level exploration. Third, except for PARADE [52], all (full-system) cycle-accurate simulators also need to implement the accelerator design in RTL, which results in tedious efforts. Finally, PARADE is

Table 5.1: Evaluation methodologies in existing accelerator-related research.

Full-system	Methodology	Related work
N	pre-RTL simulation	Aladdin [149]
N	RTL simulation with SPICE models	AccStore [122] Sonic Millip3De [147]
N	Cycle-accurate simulation	H.264 [79] Convolution Engines [139] Conservation Cores [160]
Y	Full-system cycle-accurate simulation	Walker [102] DySER [77] ARC [54] CHARM [53] BiN [56] PARADE [52]
N	FPGA prototyping	LegUp [28][27][78] FPCA [57] CoRAM [46][47]
Y	Full-system FPGA Prototyping	DySER [20] TSSP [118] LINQits [45] PARC [34] ARACompiler [41]

the state-of-the-art full-system cycle-accurate ARA simulator that provides various design space exploration choices. PARADE extends the widely used gem5 [23] simulator with HLS support to reduce the efforts of modeling the accelerators. We will compare the evaluation time of our ARAPrototyper to PARADE in Section 5.6.2.

Compared to the long-running simulation, FPGA prototyping [20, 28, 27, 34, 41, 45, 57, 71, 78, 118] is gradually gaining increased attention because it enables native measurement of the performance and power in real silicons. However, the tedious prototyping efforts impede the wide adoption of FPGA prototyping for ARA design and evaluation. In this chapter we exploit full-system FPGA prototyping to enable rapid design space explorations for the emerging ARAs. Our goal of ARAPrototyper

is to reduce the tedious prototyping efforts far down to manageable efforts.

5.2.1 Comparison to Recent Prototyping Work

In this subsection we compare the ARAPrototyper to the four most related prototyping systems: PARC [34], ARACompiler [41], LegUp [28, 27, 71, 78], and CoRAM [46, 47]. In addition, we also discuss commercial accelerator design tools such as Xilinx SDSoC [166].

1. **PARC [34].** PARC is our first-generation FPGA prototype designed to evaluate the ARA architecture described in [54]. ARAPrototyper shares some methodologies that are similar to PARC: the integration with high-level synthesis flow, shared memory architecture, and accelerator API support. However, ARAPrototyper provides many more new features. First, ARAPrototyper significantly reduces the prototyping efforts (hundreds of LOCs to a few LOCs as compared in Section 5.6.3) by providing a clean accelerator integration interface and automation flow. Second, ARAPrototyper significantly enlarges the scope of design space explorations for ARAs: 1) it adds the customizable interconnect layer between buffers and DRAM ports to explore the efficiency of off-chip accesses; 2) it adds the coherency choice at either LLC or DRAM. Third, ARAPrototyper adds performance counter support to provide more insights into the performance evaluation. Finally, ARAPrototyper is implemented in the newer Xilinx Zynq SoC board [89] and has stronger ARM processor support (PARC uses a much weaker MicroBlaze processor), and thus models a real-life ARA more closely.

2. **ARACompiler [41].** The ARACompiler is our early enhanced version over

PARC, and lies in the middle between PARC and ARAPrototyper. Compared to ARACompiler, ARAPrototyper has the advantages of 1) manageable prototyping efforts (quantitative comparison of efforts will be presented in Section 5.6.3) and 2) performance counter support to provide more insights, as explained above. More importantly, ARACompiler [41] is published as a poster paper. This discussion of ARAPrototyper provides many more implementation details and extensive design space explorations, which can provide more insights to the community.

3. **LegUp [28, 27, 71, 78].** LegUp takes a standard C program as input and automatically compiles the program into a hybrid architecture with a MIPS soft processor and customized accelerators. The more recent update [71] uses an ARM processor in the Altera FPGA-SoC and can take OpenMP and pthread functions as input. LegUp can perform self-profiling on the processor and identify program sections that would benefit from hardware acceleration. The identified sections are synthesized by its own HLS engine. Compared to LegUp, ARAPrototyper takes a different design philosophy. ARAPrototyper allows users to design the accelerators themselves (also adopted in [27]) or leverage existing accelerators that other hardware developers provided. More importantly, ARAPrototyper models the emerging ARA architectures that have the global accelerator management (GAM), customizable interconnect between accelerators and buffers, interconnect between buffers and DRAMs, coherency choice at LLC or DRAM, etc. In addition, ARAPrototyper adds the performance counter support to provide more insights into ARA design space explorations. This is totally from a different perspective and none of these ARAPrototyper features are supported by LegUp.

4. **CoRAM [46, 47].** The goal of CoRAM is to provide a scalable and portable memory architecture so that designers can focus on the accelerator design instead of building the memory architecture from scratch. A 2D-mesh interconnect is used to provide the connectivity between CoRAM blocks, which is different from the partial crossbar architecture explored in ARAPrototyper. CoRAM provides the flexibility for designers to customize the on-chip SRAM blocks into caches, FIFOs or buffers. But designers still need to expend considerable effort to design these customized memories, which impedes the goal of rapid prototyping and evaluation. Similar to LegUp, the full-system evaluation capability is not supported. Instead, ARAPrototyper provides the capability to observe interactions between hardware and OS, such as the performance impacts on TLB misses, which enlarges the scope of design space explorations.

5. **Commercial tools [166].** FPGA vendors also provide tools to design and prototype customized SoCs. For example, designers can use Xilinx SDSoC [166] to build their own accelerators using FPGA fabrics that work together with hard ARM cores. However, it does not support most features that an ARA needs, such as the global accelerator manager, customized interconnect between accelerators, buffers, and DRAM, performance counters, to name just a few. ARAPrototyper provides a reusable baseline with highly customizable parameters for a typical ARA, and provides easy accelerator integration for rapid prototyping.

5.3 The Baseline ARA Prototype

We first present an overview of the ARA that we are prototyping, based on the architecture proposed in [54, 56]. As shown in Figure 5.2, the ARA mainly contains

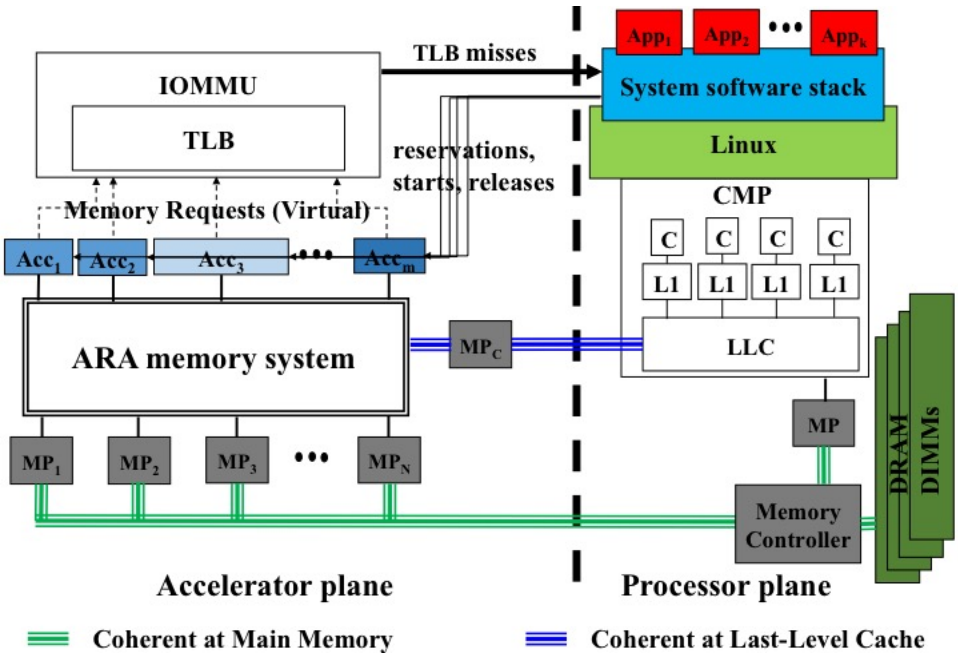


Figure 5.2: ARA overview: accelerator plane and processor plane.

two planes: 1) the accelerator plane, and 2) the processor plane. The accelerator plane is composed of the heterogeneous accelerators, the ARA memory system to support the high memory demand of accelerators, and IOMMU for address translation. The processor plane is composed of a conventional multicore processor with a multilevel cache. From a system perspective, the user applications are launched in the processor plane, and the compute-intensive tasks can be offloaded to the accelerator plane. The system software stack acts as the interface between the two planes. It provides the services of reservations, starts, and releases for the accelerators. The system software stack is implemented in the privileged mode and transparent to users.

Next, in our baseline ARA prototype, we will present the detailed design of the customizable ARA memory system in the accelerator plane and the system software stack connecting the two planes. To gain more insights into the performance evalu-

ation, we also add a few more performance counters in the accelerator plane, while we can leverage the existing performance counters for the processor plane. Finally, we will introduce some important features of Xilinx Zynq SoC [89], which is used for ARA prototyping.

5.3.1 ARA Memory System

The ARAPrototyper can generate a shared memory (buffer) architecture for heterogeneous accelerators to share the on-chip memory resources, which is similar to the architectures discussed in [56, 122]. We believe the on-chip memories need to be shared when the number of accelerators increases. To share on-chip memory resources, we provide a customized two-layer interconnect, which can be synthesized automatically by specifying the parameters in the hardware templates. We also provide the flexibility if users desire to fully customize the interconnects, even to support private buffers.

Figure 5.3 presents the accelerator plane and its ARA memory system design in detail. The major components include 1) heterogeneous accelerators, 2) homogeneous shared buffers, 3) direct memory access controllers (DMACs), 4) physical memory ports, 5) a customized partial crossbar between accelerators and buffers, 6) a customized interleaved network between buffers and DMACs, and 7) an input/output memory management unit (IOMMU) and a dedicated TLB. We can have different types and different numbers of accelerators of each type. Each type of accelerator has its own input and output port demands. Each port can connect to one or multiple buffers based on the generated partial crossbar topology.

The ARAPrototyper provides a pool of homogeneous buffers to be shared by the

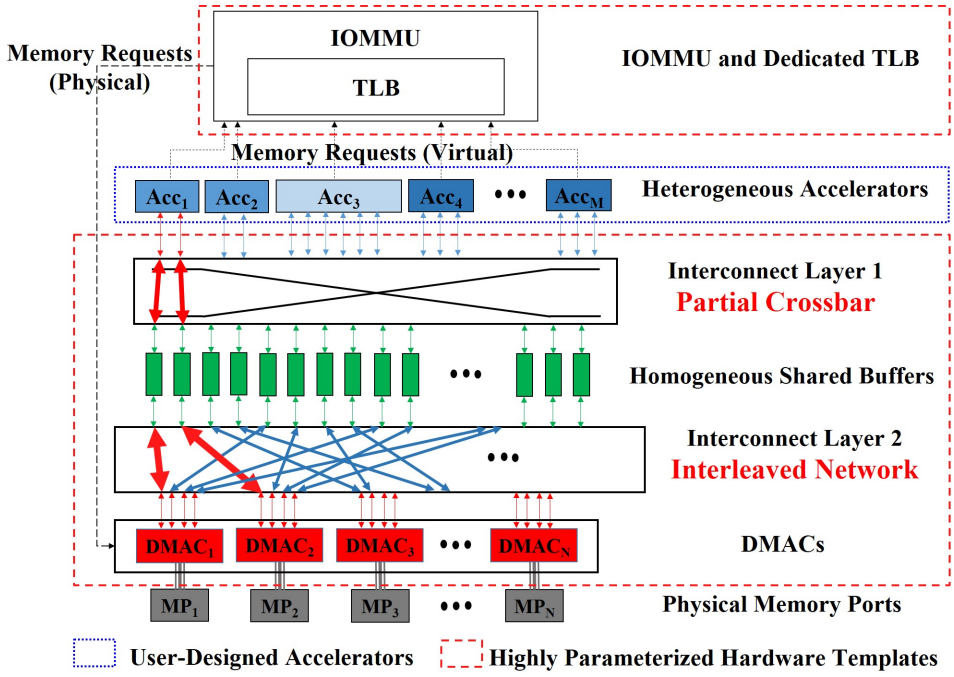


Figure 5.3: Accelerator plane and the ARA memory system.

accelerators. Before computation, an accelerator needs to send requests to IOMMU to perform page translations. After that, IOMMU assigns corresponding DMACs to issue memory requests to fetch data through physical memory ports (MPs). The off-chip long burst requests are interleaved with the interleaved network to minimize possible conflicts. The memory requests are at the page granularity (4KB). The buffer size is 16KB by default, but can be configured by users.

5.3.1.1 Customizable Optimal Partial Crossbar

The goal of the partial crossbar is to provide sufficient connectivity between the accelerators and shared buffers. The partial crossbar avoids extra arbitration cycles that occur in a conventional bus. Therefore, a deep-pipelined accelerator can achieve

initiation interval (II) as low as one with the partial crossbar support. Figure 5.4 demonstrates a real interconnect topology generated from ARAPrototyper. In this example, the accelerator plane contains six heterogeneous accelerators. The numbers inside each parentheses represent the assigned buffer IDs to the accelerator, which forms the topology of the customized partial crossbar. When an accelerator is reserved by an application, the accelerator has the privilege of using the assigned buffers as its own local buffers. The accelerator can fetch one element from each buffer per cycle (II = 1) since a dedicated connection is built.

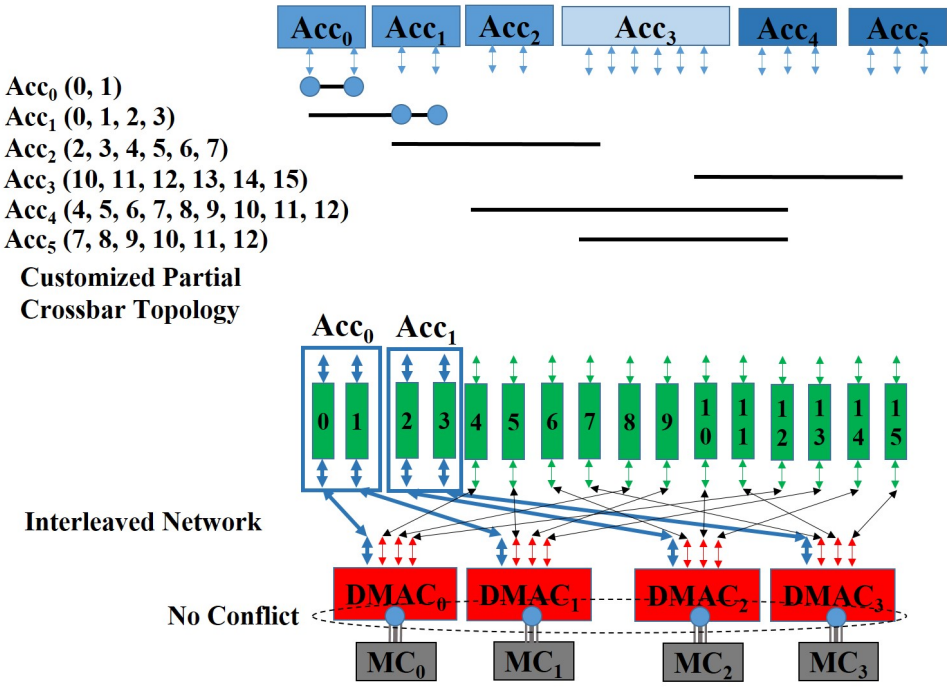


Figure 5.4: A real example of the interconnect topology generated from ARAPrototyper.

The ARAPrototyper provides a built-in optimization flow [33] for the customized partial crossbar. This optimizer takes the number of ports of each accelerator and the number of shared buffers as input. Designers need to provide the maximum number of

simultaneous active accelerators as another constraint. The number of simultaneous active accelerators can influence 1) the power budget and 2) the complexity of the partial crossbar, which reflect the two important design criteria—dynamic power and area. Our optimizer can guarantee the optimality of the crossbar with the minimum number of cross points based on the input and constraints. In PARC [60], the authors can generate an optimal crossbar topology when the number of accelerator ports of each accelerator are equal. In ARAPrototyper, we provide a more generalized optimal partial crossbar design for accelerators with heterogeneous port demands. Buffer demand information can also be reported by our built-in optimizer.

5.3.1.2 Private Buffer Architecture Support

Though we mainly target the shared buffer architecture as previously explained, ARAPrototyper also supports the private buffer architecture: each accelerator has its own buffers without sharing. Users can simply set the number of shared buffers to be equal to the number of ports of all accelerators. In this case, the shared buffer architecture is customized to the private buffers while still benefiting from the interleaved network. This can be used to evaluate an ARA with abundant buffer resources and a large power budget.

5.3.1.3 Customizable Interleaved Network

The main purpose for an interleaved network is to minimize possible conflicts for the long off-chip burst requests. An accelerator usually issues multiple requests simultaneously to prefetch data from the off-chip DRAM to on-chip buffers for near-future computation. For example, in a stencil computation, multiple data elements, e.g., five

or seven, are required for a single computation. These data are prefetched into buffers in advance. If the simultaneous requests are not distributed evenly across physical memory ports, significant performance degradation can occur. First, an accelerator can start to work only when all required data are prefetched into its buffers. Second, a memory request is at the page granularity (4KB), and thus the latency is very large. The uneven distribution of requests can cause serious delay for the pending requests.

Figure 5.4 shows how the interleaved network successfully distributed four simultaneous accesses into four DMACs. Note that the topology of the interleaved network depends on the topology of the customized partial crossbar. In ARAPrototyper, we support two design strategies for design space exploration: 1) interleaving the requests within an accelerator, and 2) interleaving the inter-accelerator requests.

5.3.1.4 Coherency Choice at LLC or DRAM

The ARAPrototyper supports two types of coherency. Users can select either one in our flow. First, the ARA memory system can be coherent with the last-level cache (LLC) residing in the processor plane. In this case, users do not need to worry about the coherency. Second, the ARA memory system can directly exchange data with the off-chip DRAM (i.e., coherent at DRAM). Compared to the LLC coherent case, the burst DMA transfer may provide higher memory bandwidth because of larger burst sizes and more physical memory ports. However, users need to invalidate the corresponding cache lines if the data is updated in the DRAM.

5.3.1.5 TLB Support in IOMMU

Since accelerators in an ARA share the physical memory with the processor and use virtual memory for simplicity, a hardware IOMMU and a dedicated TLB are provided in the accelerator plane to support the virtual to physical address translation. The TLB size is configurable by users. We leverage the system software stack to handle a TLB miss, which will be explained in Section 5.3.2.4. To gain more insights, we also add two performance counters to monitor the TLB accesses and TLB misses. Since in our cases, the data is accessed consecutively in a streaming fashion, we can also use the TLB access counter to calculate the total DRAM accesses and achieved memory bandwidth from the accelerator plane. One can add a DRAM access counter if necessary.

5.3.2 System Software Stack

Figure 5.5 presents an overview of the ARA system software stack. ARAPrototyper can automatically generate the related software modules based on the ARA specification file. The five major components in the system software stack are: 1) global accelerator manager (GAM), 2) dynamic buffer allocator (DBA), 3) coherence manager, 4) TLB miss handler, and 5) performance monitor (PM). Next, we present more details of these components. Users may further customize the software stack based on their needs.

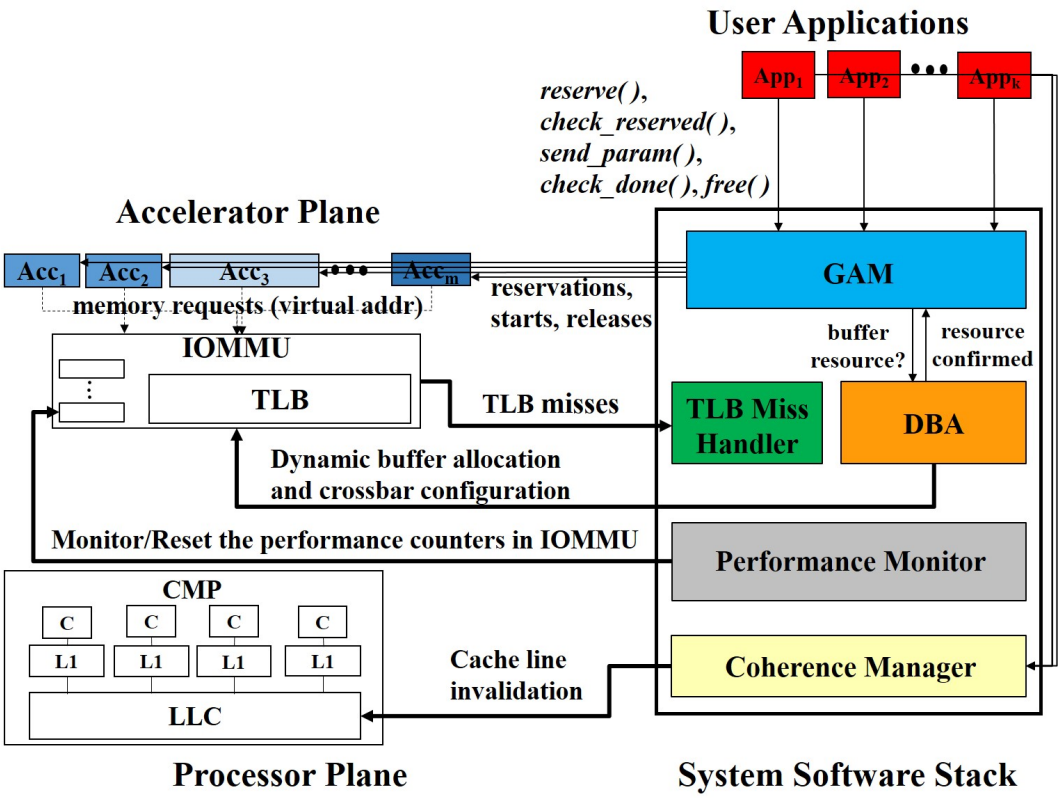


Figure 5.5: System software stack and the interactions with the ARA and user applications.

5.3.2.1 Global Accelerator Manager

GAM is responsible for 1) interfacing with user applications, 2) accelerator resource management and task scheduling, and 3) requesting buffer resources. User applications can talk to GAM with the provided APIs, which will be discussed in Section 5.5. In GAM, we use a table to keep track of the available accelerators of each type. The incoming requests from user applications are scheduled in a first-come, first-serve basis. GAM would make requests for the shared buffer resources to the dynamic buffer allocator before reserving a target accelerator.

5.3.2.2 Dynamic Buffer Allocator

In the shared buffer architecture, discussed in Section 5.3.1, a buffer bank can be shared by multiple accelerator ports. DBA is in charge of the dynamic buffer assignment during runtime based on the requests from user applications. Static assignment, such as the work in [34], can no longer handle the dynamic cases and can limit the framework scalability for evaluation.

DBA receives the buffer requests from GAM. As shown in Figure 5.6, DBA uses a list structure, called *task list*, to store the requests that have not been processed. With information of the incoming tasks, DBA is able to provide different kinds of allocation policies to influence task scheduling. Users are able to modify the allocation policy of DBA based on their demand; this is done by modifying the policies, such as throughput-driven or deadline-driven scheduling, to manipulate the *task list*.

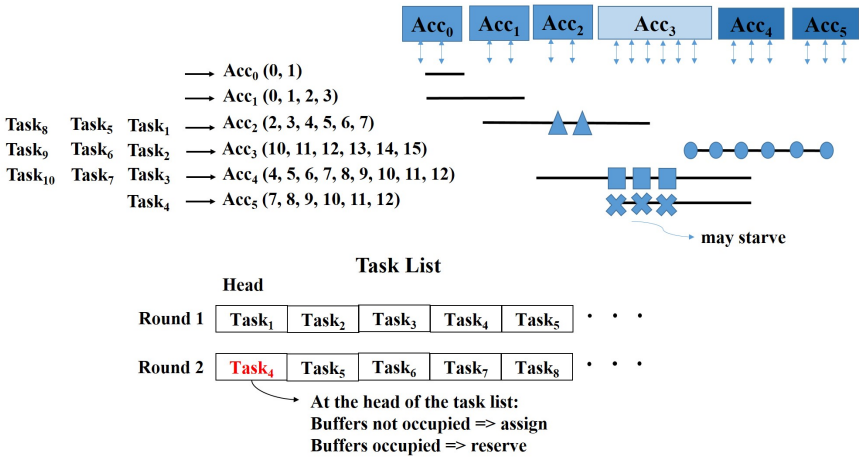


Figure 5.6: Dynamic buffer allocation: a starvation-free scheme.

In ARAPrototyper, we provide a starvation-free buffer allocation policy, as illustrated in Figure 5.6. The tasks come in numerical order. It is possible that Acc_5 can

starve since Acc_2, Acc_3, and Acc_4 occupy the buffers to serve the continuous incoming tasks. To prevent starvation, we use two flags for each buffer: occupied and reserved. A buffer can only be allocated by DBA when the buffer is neither occupied nor reserved. When a buffer is assigned to an accelerator, it will be marked as occupied. The reserved flag is used when a buffer is occupied but another accelerator would like to reserve it. The starvation can be resolved by providing the "reserved" privilege only for the task at the head of the *task list*. This guarantees that the task at the head can always occupy or reserve the required buffers. After that, the algorithm greedily allocates buffers to the tasks in the *task list* in order until no feasible allocation can be found. This algorithm is lightweighted, and the overhead is negligible.

5.3.2.3 Coherency Manager

The RAPrototyper offers a coherency manager for coarse-grained software-based coherence handling. When users try to directly write data to DRAM for higher memory bandwidth, the overlapping pages residing in multilevel caches in the processor need to be invalidated. We abstract the cache invalidation details in the coherency manager in our system software, so users only need to call the coherency manger to handle the possible coherency issue.

5.3.2.4 TLB Miss Handler

For the TLB misses arising from the accelerators, we currently use a software-based handler to handle the miss. To reduce overhead incurred in the communication between IOMMU and the TLB miss handler in the privileged mode, IOMMU groups multiple TLB misses and sends them to the handler together. Instead of using the

slow kernel API to do page translation, we write our own version by leveraging the ARM architecture support for page table walk. Table 5.2 shows our profiling results on the average TLB miss handling time. Our efficient walker reduces the miss penalty from 4278 cycles to 458 cycles, and thus performance degradation from TLB misses in an ARA can be significantly reduced.

Table 5.2: Average TLB miss penalty; kernel APIs vs. software page table walk (Acc@100MHz).

	Microblaze	Cortex-A9	Cortex-A9
Method	Kernel APIs	Kernel APIs	pgtwalk
Freq.	100MHz	667MHz	667MHz
Cycles	4975	4278	458
Time(us)	49.75	6.41	0.69

We are also considering implementing a hardware-based page walker. It can be a scalable version when the number of accelerators is large. However, the hardware-based walker in our Zynq board [89] can lead to three sequential DRAM accesses (600 cycles) per TLB miss because of the walk on multilevel page tables.

5.3.2.5 Performance Monitor

To provide more insights into the system bottleneck analysis, we add performance counters in the IOMMU so that the TLB hit/miss events and memory bandwidth can be monitored on-the-fly, as mentioned in Section 5.3.1.5. We add a PM module in the system software stack to handle requests from applications and interact with the IOMMU to monitor or reset the performance counters. These performance counters can provide more in-depth performance characterization in addition to simple runtime

numbers. The impacts of different architecture parameters can also be observed and analyzed.

In addition to using the accelerator-side performance counters supported by PM, users can also use OProfile [135] to obtain the performance counter information inside CPU cores. We successfully ported OProfile on top of our ARA baseline prototype under the Zynq platform.

5.3.3 Prototyping Platform: Xilinx Zynq SoC

We choose the Xilinx Zynq ZC706 evaluation board [89] with 1GB DRAM as our underlying prototyping platform. Figure 5.7 shows the architecture of Zynq SoC. It is composed of FPGA fabrics for accelerator implementation and a dual-core ARM for the system software implementation. The FPGA contains around 2MB on-chip block RAM, which can be used to implement the shared buffers. User applications can be launched on the ARM processor with Linux support. The Zynq architecture has the following advantages to support ARAPrototyper.

1. **Faster processor cores.** The hard ARM cores can run up to 800MHz, which is much more efficient than the soft Microblaze cores synthesized from FPGA. Linux can be ported on ARM cores and executed fluently. The system software stack provided in ARAPrototyper, including global accelerator manager, dynamic buffer allocator, coherence manager, TLB miss handler, and performance monitor, can all leverage the faster processor.

2. **A coherent LLC support.** The dual-core ARM provides a shared L2 cache. The shared buffers can be coherent with L2 cache through the accelerator-coherent port (ACP). This provides an alternative ARA design opportunity (as

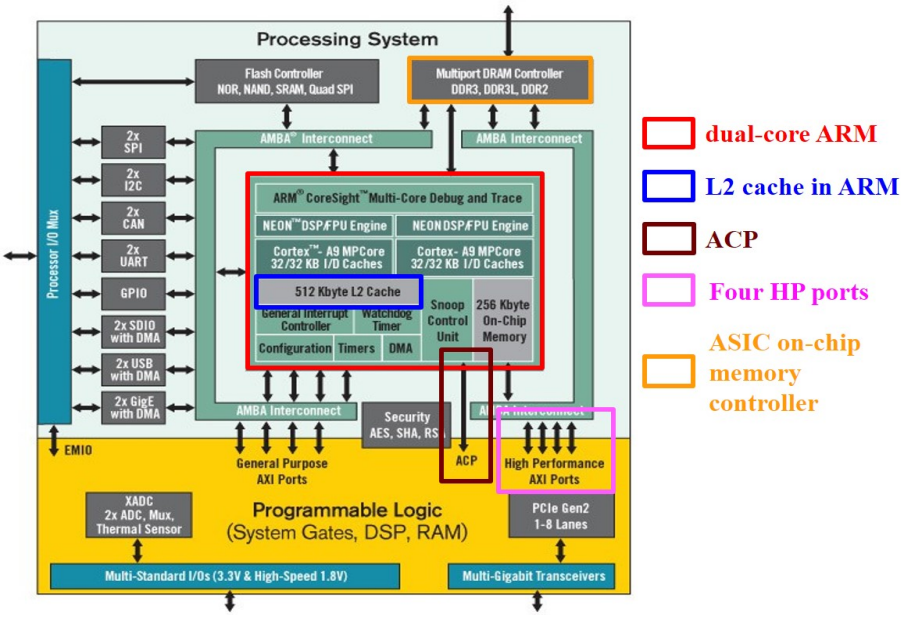


Figure 5.7: The prototyping platform: Xilinx Zynq SoC. (Taken and modified from the Xilinx website.)

described in Section 5.3.1.4).

3. **A fast ASIC on-chip memory controller.** The ASIC on-chip memory controller provides higher memory bandwidth compared to a memory controller synthesized in FPGA. In order to efficiently exploit the available memory bandwidth, Zynq provides four high-performance (HP) ports in FPGA fabrics. This gives us opportunities to explore the topology of the interleaved network to better utilize the off-chip bandwidth.

Prior PARC work [34] is prototyped on the ML605 board with Virtex 6 FPGA. Compared to the Virtex 6 with only FPGA fabrics, Zynq SoC enables a wider range of ARA design explorations.

5.4 Design Automation Flow and ARA Customization Interface

The main challenge to architectural design space exploration through FPGA prototyping is the long development cycle for each generation of an ARA, which requires extensive coding in RTL. To further reduce the prototyping efforts, we develop a highly automated design flow for users to customize the baseline ARA prototype and integrate their own accelerators. Users only have to configure an ARA specification XML file to customize their ARA, and specify a few parameters in the acceleration integration interface to add their own accelerators that are written in HLS. Our design flow can automatically generate users' customized ARAs and deploy them on the underlying FPGA prototyping platforms.

5.4.1 Design Automation Flow

We classify the components in the ARA prototype into the following three parts.

1. **Platform-specific modules.** The platform-specific modules are mainly bonded to the hard modules in the FPGA chip and evaluation platform (board), such as the dual-core ARM processor. ARAPrototyper can adapt to different platforms, and thus users can spend minimum effort on the platform issues.

2. **Platform-independent modules.** The platform-independent modules include two-layer interconnects, shared buffers, IOMMU and TLB, and DMACs, which are the major components in the ARA memory system. ARAPrototyper provides highly parameterized hardware templates for these platform-independent components. Users can easily customize them in the ARA specifi-

cation file that will be explained in Section 5.4.2.

3. **User-designed accelerators.** In the ARAPrototyper, we provide a group of highly optimized accelerators in the medical imaging pipeline that will be further explained in Section 5.6. Users can easily develop their own accelerators in HLS. Furthermore, we provide a clean accelerator integration interface (to be explained in Section 5.4.3) for users to easily add their own accelerator into our ARAPrototyper.

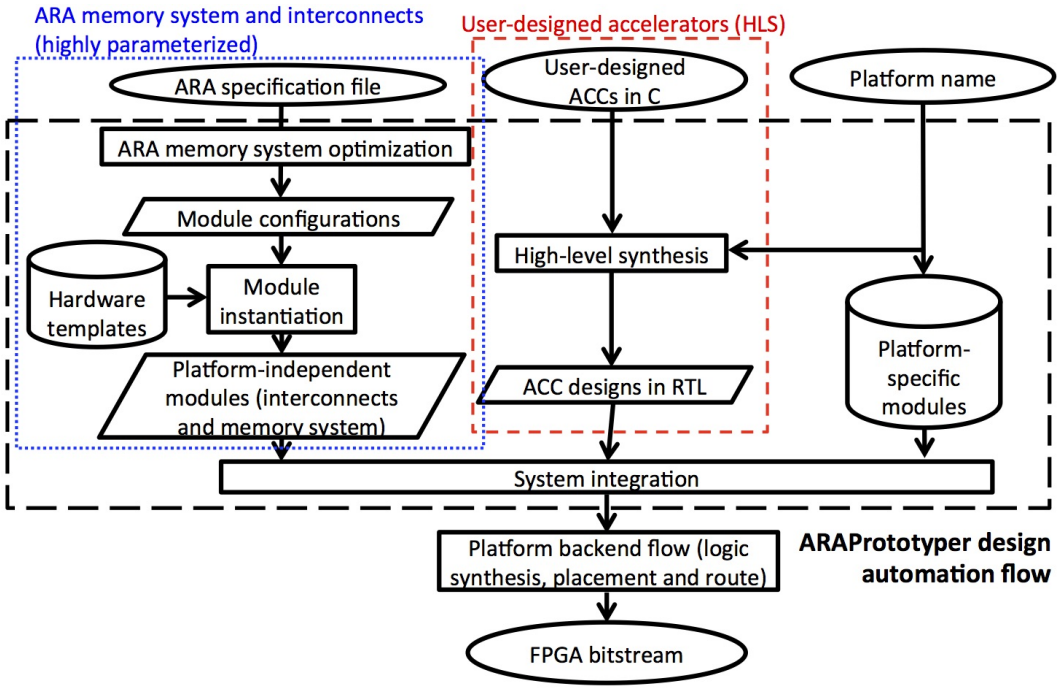


Figure 5.8: ARAPrototyper design automation flow.

Figure 5.8 presents our design automation flow. It begins with the ARA specification file, and all following steps can be executed automatically upon a single "make" button. In the left branch, we apply ARA memory system optimizations to our platform-independent modules by using the configurations in the ARA specifica-

tion file. This is combined with our hardware templates to create the ARA memory system. In the middle branch, HLS tools are applied to user-designed accelerators coded in C/C++ for generating RTL designs. In the right branch, the platform information is used to generate platform-specific modules. Depending on the target platform, e.g., Xilinx Zynq FPGA in our case, the flow can be seamlessly integrated with the corresponding back-end process (e.g., Xilinx PlanAhead flow for bitstream generation).

5.4.2 ARA Specification File

The ARA specification file is provided for users to specify components and configure the parameters in the accelerator plane. Users can easily evaluate their new accelerators by integrating them into the reusable baseline prototype, and perform system-level design space explorations. This is composed of six major sections: 1) accelerator specification (including the number of read and write ports and the number of parameters sent from the application), 2) the number of shared buffers and DMACs, 3) two-layer interconnects configurations, 4) IOMMU configurations, 5) coherence type, and 6) target frequency. The ARA specification file is recorded in a XML format, with an example shown in Listing 5.1. Users can easily modify it on top of the XML template we provide, including (1) accelerator specification, (2) shared buffer and DMAC specification, (3) interconnect specification, (4) IOMMU specification, (5) coherence specification, and (6) target frequency.

```
<system>

<ACCs>
   <acc type="gradient" num="2" num_params="5">
       <port size="16K" num="6" />
   </acc>
```

```
    <acc type="segmentation" num="1" num_params="13">
        <port size="16K" num="8" />
    </acc>
    <acc type="rician" num="1" num_params="7">
        <port size="16K" num="12" />
    </acc>
    <acc type="gaussian" num="1" num_params="7">
        <port size="16K" num="5" />
    </acc>
</ACCs>

<SharedBuffers size="16K" num="32" numDMACs="4" />

<Interconnects>
    <ACCs_to_Buffers type="crossbar" connectivity="3" auto="1" />
    <Buffers_to_DMACs type="interleaved" use="1" auto="1" />
</Interconnects>

<IOMMU>
    <TLB size="8K" evict="LRU" />
<IOMMU/>

<CoherentCache use="0" />

<AccFrequency hz="75MHz" />

</system>
```

Listing 5.1: An example ARA specification file created for design space exploration with four types of accelerators via ARAPrototyper.

5.4.3 Accelerator Integration Interface

In order to further reduce the development cycle of adding users' own accelerators, ARAPrototyper supports the integration of accelerators developed in HLS. Moreover, ARAPrototyper provides a clean accelerator integration interface that only needs a

few lines of code to integrate users' own accelerators.

Accelerator development in HLS. Design productivity of accelerators can be improved by raising the level of design abstraction beyond RTL. HLS tools [51, 168] enable automatic synthesis of high-level, untimed or partially timed specifications (such as in C, C++, or SystemC) to low-level cycle-accurate RTL code. As reported in [51], the code density can be easily reduced by 7 to 10X when moved to high-level specification in C, C++, or SystemC, and at the same time, resource usage can also be reduced by 11 to 31% in an HLS solution, compared to a hand-coded RTL design.

```
void vecsqr (volatile int* vaddr_port0, volatile int* vaddr_port1,
  volatile int* buf_id_port0, volatile int* buf_id_port1,
  ...
  volatile int* IOMMU_FIFO,
  volatile float port0[CHUNK_SIZE],
  volatile float port1[CHUNK_SIZE]
  ...) {
    // read function parameters
    int vaddr0  = *vaddr_port0;
    ...
    // issue read memory access
    memory_request0(READ, IOMMU_FIFO, vaddr0, buf_id0, Req_Length0);
    ...
    // computation kernel to be plugged in
    for(int i = 0; i < Req_Length1; i ++)
      port1[i] = port0[i] * port0[i];
    ...
    // issue write memory access
    memory_request1(WRITE, IOMMU_FIFO, vaddr1, buf_id1, Req_Length1);
}
```

Figure 5.9: Accelerator integration template in HLS-compatible C.

Accelerator integration interface. To allow users to easily integrate existing accelerators into the ARAPrototyper, we provide the following flow. First, designers need to specify the accelerator port information in the ARA specification file, such as the number of parameters sent from CPU and the number of demanded buffers.

Next, our tool can generate the port names and the corresponding HLS pragma for the control and data ports between the accelerator, IOMMU, and CPU automatically using the ARA specification file. This generated file in HLS-compatible C format is called accelerator integration template, as shown in Figure 5.9. Designers need to place the computation kernel and the invoking of read and write memory requests explicitly in the corresponding locations in the template.

Control and data ports are generated as function parameters in the HLS codes. There are three kinds of ports. First, the input parameters sent from CPU (such as "vaddr_port0") are generated. These parameters are sent from the CPU through the AXI-Lite port and are stored in the registers of the accelerator. Second, the communication channel ("IOMMU_FIFO") realized by FIFO is generated. The accelerator uses the FIFO to send read and write requests to IOMMU to fetch data from DRAM (or L2 cache) to its own buffers and write data back from its own buffers to DRAM (or L2). Third, the ports to input and output buffers such as "port0" and "port1" are generated. These ports are connected to the shared buffers through the automatically synthesized crossbar described in Section 5.3.1.1.

The only changes that designers need to specify are the memory requests in reading data from DRAM (or L2 cache) and writing data back to DRAM (or L2) in the accelerator integration template. For example, a memory request ("memory_request0") needs to be specified explicitly before the computational kernel reads data from its own buffer. Similarly, the output results need to be written back after computation is done in "memory_request1." This only involves a few lines of code (LOCs) change. As shown in Figure 5.9, after the existing accelerator computation kernel is plugged in, only the two lines with "Req_Length0" and "Req_Length1" (shown in red color and bold italic font) are added. Detailed prototyping efforts (LOCs) will be presented

in Section 5.6.3.

5.5 Application Development API

To enable rapid development of applications that use the accelerators in ARAPrototyper, we abstract accelerators as software libraries and provide the user-friendly C/C++ APIs. With the information provided by the ARA specification file, ARAPrototyper can automatically generate the header file of accelerator APIs for programmers, which is similar to [34][41]. For each type of accelerator, we provide the following APIs with fine-grained accelerator control as done in ARACompiler [41]: 1) *reserve()*, 2) *check_reserved()*, 3) *send_param()*, 4) *check_done()*, and 5) *free()*. Users can develop their applications with the C++ classes and member functions to manipulate the accelerators in the ARA. After applications are developed, users can simply set up a g++ cross compiler to compile applications into ARM executable, which can be seamlessly executed on the ported Linux on the Zynq board.

Figure 5.10(a) is an application code example using an accelerator. The application can first use the *reserve()* function to make requests to GAM for reserving an accelerator. After the reservation is confirmed, the required parameters are sent through the *send_param()* function, and the accelerator will be started. The application should periodically check the status of accelerators with *check_done()*. Once the accelerator finishes its task, it should be freed for future use. With these APIs, a programmer can explore more complicated settings, such as using multiple accelerators simultaneously in the user application.

Compared to ARACompiler [41], ARAPrototyper provides a simplified API, called *run()*, which is intended for software developers who do not want to dig into the

```
#include "accelerator_type.h"
void main() {
   class Acc_Gaussian acc; Image a;
   acc.reserve();
   while(acc.check_reserved() == 0);
   acc.send_param(7, Image::get_M(), Image::get_N(),
      Image::get_P(), a.get_ptr(), 1, 1, 1);
   while(acc.check_done() == 0) wait(1000);
   acc.free();
}
```

(a)

```
#include "accelerator_type.h"
void main() {
   class Acc_Gaussian acc;  Image a;
   acc.run(7, Image::get_M(), Image::get_N(),
      Image::get_P(), a.get_ptr(), 1, 1, 1);
}
```

(b)

```
#include "accelerator_type.h"
#include "TLB_PM_api.h"
void main() {
   class Acc_Gaussian acc;  Image a;
   class TLB_Performance_Monitor pm;
   pm.reset_tlb_counters();
   acc.run(7, Image::get_M(), Image::get_N(),
      Image::get_P(), a.get_ptr(), 1, 1, 1);
   int access_num = pm.get_tlb_access_num();
   int miss_num = pm.get_tlb_miss_num();
}
```

(c)

Figure 5.10: Code examples of using APIs to develop applications.

hardware accelerator details. This API covers the functionality, from reserving to releasing the accelerator using a single function. Figure 5.10(b) is a code example, which achieves the same functionality as the code of Figure 5.10(a).

As presented in Section 5.3.2.5, ARAPrototyper provides a PM module to monitor performance counters added in our prototype. We provide several APIs built on top of the PM module so that designers can use these APIs to monitor those key performance counters for analyzing and improving the ARA design. Figure 5.10(c) shows how TLB accesses and misses can be monitored by using the provided APIs.

```
#include "accelerator_type.h"
void main() {
   class Acc_Gaussian acc1;  class Acc_Gradient acc2;
   Image a, b, c;
   acc1.reserve();
   acc2.reserve();
   while(acc1.check_reserved() == 0);
   while(acc2.check_reserved() == 0);

   // start two accelerators at the same time
   acc1.send_param(7, Image::get_M(), Image::get_N(),
      Image::get_P(), a.get_ptr(), 1, 1, 1);
   acc2.send_param(5, Image::get_M(), Image::get_N(),
      Image::get_P(), b.get_ptr(), c.get_ptr());

   int acc1_done = 0;
   int acc2_done = 0;
   int all_done = 0;
   while(!done) {
       if(!acc1_done) acc1_done = acc1.check_done();
       if(!acc1_done) acc1_done = acc1.check_done();
       done = acc1_done & acc2_done; wait(1000);
   }
   acc1.free();
   acc2.free();
}
```

Figure 5.11: Code examples of using APIs to develop applications when utilizing more than one accelerator in the ARA.

Furthermore, programmers can write a complex application by using the provided

APIs. Figure 5.11 shows an example where a programmer can start two accelerators simultaneously. This allows users to explore the latency and the throughput when more than one accelerator is used in the given ARA.

5.6 Experimental Results

In this section we present a quantitative evaluation of the rapid evaluation time and manageable prototyping efforts of ARAPrototyper for ARA design space explorations. We choose the medical imaging processing pipeline as our target application domain. To demonstrate the capability and usage of ARAPrototyper, we conduct a number of case studies for ARA design explorations.

5.6.1 Target Domain: Medical Imaging

The target application domain we choose to accelerate is primarily the medical imaging processing pipeline, which is one of the most important application domains in personalized medical care. This pipeline is used to process the raw data obtained from computerized tomography (CT) [25]. We are motivated to accelerate it with a scenario in which a doctor is able to show the CT analysis interactively to patients with a tablet. Current mobile processors without accelerators cannot provide real-time and energy-efficient solutions.

The medical imaging pipeline can be divided into the following stages. After image reconstruction, it will 1) remove noise and blur; 2) align the current image with previous images from an individual; and 3) segment a region of interest for diagnosis [25]. The three tasks can be implemented by four accelerator kernels: *gradient*,

gaussian, *rician*, and *segmentation*. In the following subsections, we will mainly demonstrate the benefits of ARAPrototyper using these four accelerator kernels for case studies.

5.6.2 Evaluation Time

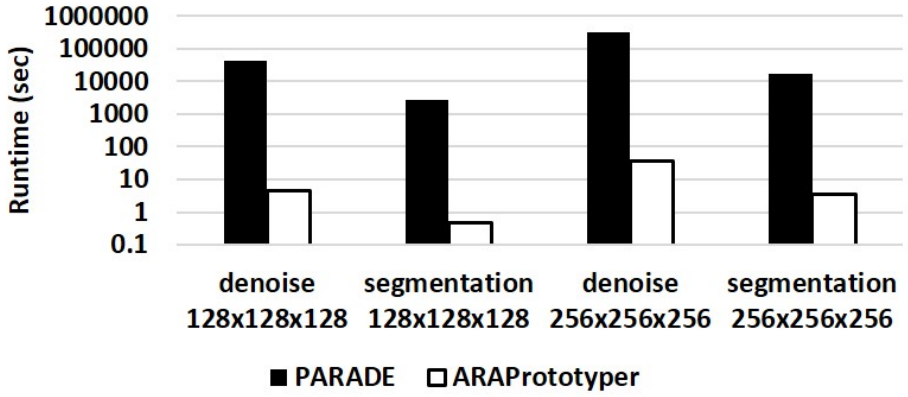


Figure 5.12: Evaluation time on ARAPrototyper and PARADE.

To demonstrate the rapid evaluation time of ARAPrototyper, we compare it against the state-of-the-art full-system ARA simulator PARADE [52] for a typical ARA configuration. Figure 5.12 compares the execution time of two common medical imaging applications with different input sizes on ARAPrototyper and PARADE. For the larger input size, it takes a couple of days for PARADE to simulate one single ARA configuration, while it only takes a minute or so for ARAPrototyper to run the configuration. We should mention that our flow generation time (generating the ARA configuration to the FPGA bitstream) takes around four hours, but it is a one-time effort for one ARA configuration that can run multiple applications with multiple inputs. Usually, the native executions on our FPGA prototype are 4,000X to 10,000X faster than full-system simulations, and we believe ARAPrototyper can

be an attractive alternative for design space explorations.

5.6.3 Prototyping Efforts

To demonstrate the manageable prototyping efforts of ARAPrototyper, we present the lines of code (LOCs) that users have to change or add to customize their own ARA using existing accelerators or integrating their own accelerators.

We first present LOCs for users to configure their own ARA by leveraging our reusable baseline prototype and existing accelerators. As shown in Table 5.3, users can simply configure the ARA specification file with up to 33 lines of XML code to set up the parameters of the shared memory architecture and operating frequency. There is no C/C++ description or RTL code required. Users can start the push-button ARAPrototyper flow after the specification file is set to obtain an FPGA prototype in hours. We also present the LOCs for automatically generated RTL from the baseline prototype in Table 5.3, which is more than 37,000 lines. It reflects the huge engineering efforts required if everything is built from scratch.

Table 5.3: Lines of code (LOCs) to customize users' own ARA prototype using existing accelerators.

		# line of XML code
input	ARA description file	33
	components	# line of RTL code
automatically generated	IOMMU	21407
	crossbar	1526
	top module	14253
	total	37186

Next, we present LOCs for users to integrate their own accelerators. To demonstrate the benefits of reduced prototyping efforts of ARAPrototyper compared to prior work such as PARC [34] and ARACompiler [41], we also include them for a quantitative comparison. Table 5.4 presents the LOCs to integrate our medical imaging accelerators and third-party MachSuite [144] accelerator kernels into PARC/ARACompiler/ARAPrototyper, including total generated RTL code, total HLS C/C++ code, kernel-only HLS code, and integration-only code. We include eight more accelerator kernels from a widely used third-party accelerator benchmark suite MachSuite to better illustrate our manageable prototyping efforts.[2] As shown in Table 5.4, compared to the hundreds of LOCs for accelerator integration in PARC and ARACompiler, users only need to add a few LOCs (most of the time less than 10 LOCs) to integrate their own accelerators into ARAPrototyper due to its clean accelerator integration interface and automation flow.

5.6.4 Design Space Exploration

To demonstrate the capability and use of ARAPrototyper, we conduct the following case studies for ARA design explorations.

5.6.4.1 Private vs. Shared Buffer Architecture

First, users can configure the ARAPrototyper to achieve either a private or shared buffer architecture (as explained in Section 5.3.1.1 and Section 5.3.1.2). To demonstrate this, we use an example ARA with a total of five accelerators. In the private

[2]We do not include MachSuite accelerators for more studies because their performance is far below optimal. However, users can engage in further accelerator microarchitecture explorations based on ARAPrototyper.

Table 5.4: Lines of code (LOCs) to integrate medical imaging and third-party MachSuite kernels into PARC/ARACompiler/ARAPrototyper, including total generated RTL code, total HLS C/C++ code, kernel only HLS code, and integration-only code.

Domain	Accelerator	Total RTL	Total HLS	Kernel	PARC/ ARACompiler Integration	ARA-Prototyper Integration
Medical Imaging	gaussian	15107	513	363	150	5
	gradient	32538	778	616	162	6
	segmentation	63857	1304	1070	234	8
	rician	42291	1140	850	290	12
MachSuite [144] (third-party)	FFT/TRANSPOSE	17072	530	412	118	4
	GEMM/NCUBED	3201	121	23	98	3
	GEMM/BLOCKED	5226	158	20	138	5
	KMP/KMP	3593	167	45	122	4
	MD/KNN	7023	243	53	190	7
	SORT/MERGE	2996	128	54	74	2
	SPMV/CRS	4080	160	18	142	5
	VITERBI/VITERBI	4212	177	35	142	5

buffer architecture, each accelerator needs its own buffer resources, regardless of how many of the accelerators are simultaneously powered on dynamically. While in the shared one, with the maximum number of simultaneously powered-on accelerators in mind, we can allocate the minimum buffer resources to support any combination of

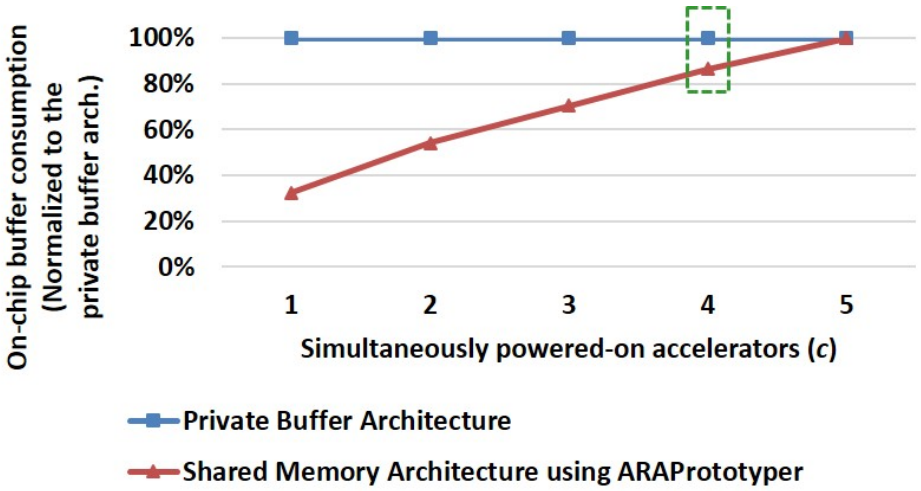


Figure 5.13: Buffer consumption: private vs. shared buffer arch.

accelerators running. Figure 5.13 shows that the shared buffer architecture can use much less physical buffer resources (thus less area and power) when not all accelerators are running simultaneously. On the other hand, if the shared buffer architecture is designed to support at most four simultaneous accelerators, but users need to run five tasks, then it would degrade the performance by 12.6% compared to the private buffer architecture (with 15.6% less buffer resources) based on our profiling.

5.6.4.2 Interleaved Network: Inter-Acc vs. Intra-Acc

Second, ARAPrototyper provides the flexibility for users to evaluate different interconnects between buffers and DRAM. Users can (statically) configure this interconnect to interleave inter-accelerators to achieve fairness among accelerators or within an accelerator to achieve better performance for the accelerator, as explained in Section 5.3.1.3. Figure 5.14(a) presents the performance of inter-accelerator and intra-accelerator interleaved networks for our medical imaging accelerators. As discussed

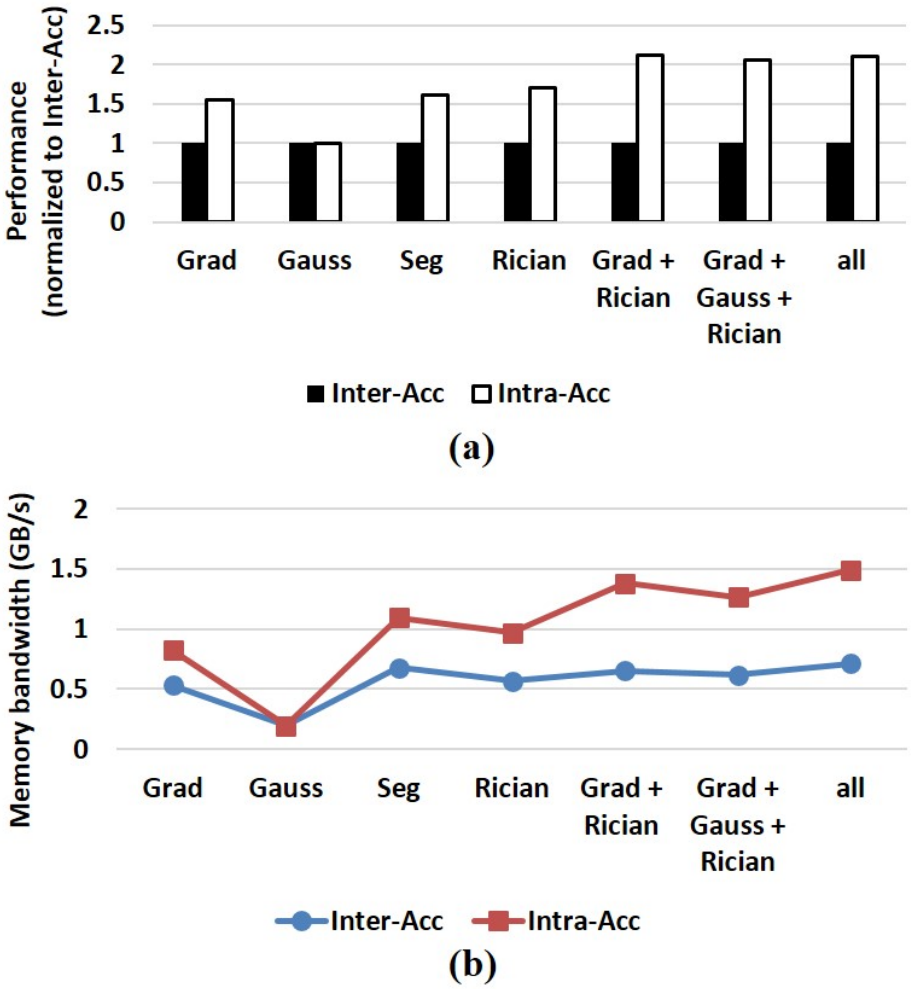


Figure 5.14: Evaluation on (a) performance and (b) memory bandwidth between Inter-Acc and Intra-Acc interleaving networks.

in Section 5.3.1.3, the intra-accelerator interleaving can prevent the case in which all long-burst requests from the accelerator are issued to the same DMACs.

To gain more insights into this performance speedup, we further compare the achieved bandwidth of both cases; this can be obtained using the added performance counters in ARAPrototyper. As shown in Figure 5.14(b), intra-accelerator interleav-

ing can achieve better bandwidth than inter-accelerator interleaving in our case, and thus achieves better performance. We can also observe that the available memory bandwidth is not the performance bottleneck. When we launch two, three, or four accelerators simultaneously, the utilized memory bandwidth still increases. Note that *gaussian* is a special case since it only fetches four pages of data, and thus the impact is negligible.

5.6.4.3 Coherency Choices

Third, ARAPrototyper provides the flexibility for coherency choices at either LLC or DRAM depending on the application locality. Figure 5.15(a) presents the performance of both coherency choices. In our case, coherency at DRAM achieves up to 1.7X performance speedup compared to coherency at LLC. The major reason is that our medical imaging applications behave in a streaming fashion and have poor locality at LLC. Another reason is due to current Zynq board limitation, where the LLC has only one port while DRAM has four ports. As a result, coherency at DRAM can achieve higher bandwidth, as shown in Figure 5.15(b).

5.6.4.4 Impact of TLB sizes

Fourth, The ARAPrototyper provides the flexibility for users to configure different TLB sizes. In addition, we also provide performance counters in ARAPrototyper to get the number of TLB accesses and misses for further performance analysis. A TLB miss handling (in software) penalty can also be collected in our system software stack. Figure 5.16(a) and Figure 5.16(b) present the TLB miss rate and TLB miss handling penalty (in terms of percentage of total execution time) for different TLB sizes. In

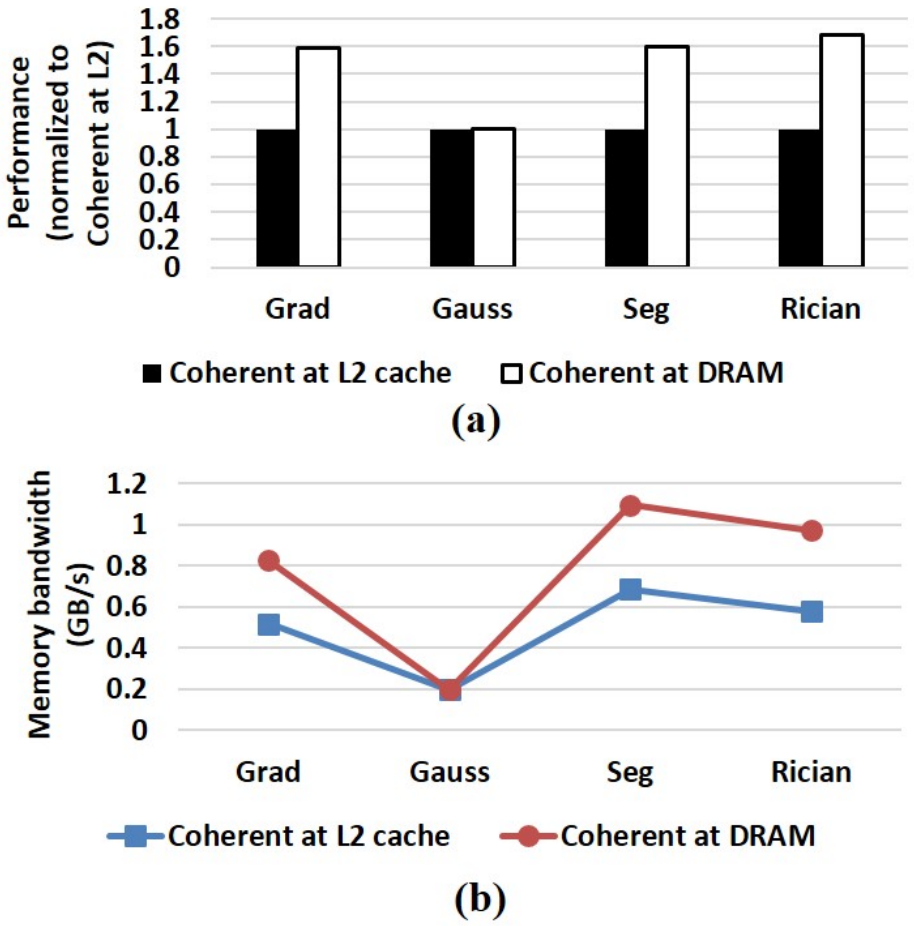


Figure 5.15: Evaluation on (a) performance and (b) memory bandwidth for different coherency choices.

our case, we choose 32K TLB entries in the design since the TLB miss rate and miss handling penalty will stop to shrink after this point. Another point we want to mention is that TLB misses can cause up to a penalty of 24% of the whole execution time in an ARA due to the streaming access behavior and accelerated computation. Therefore, this needs more attention to address translation support when designing an ARA compared to a general-purpose CPU.

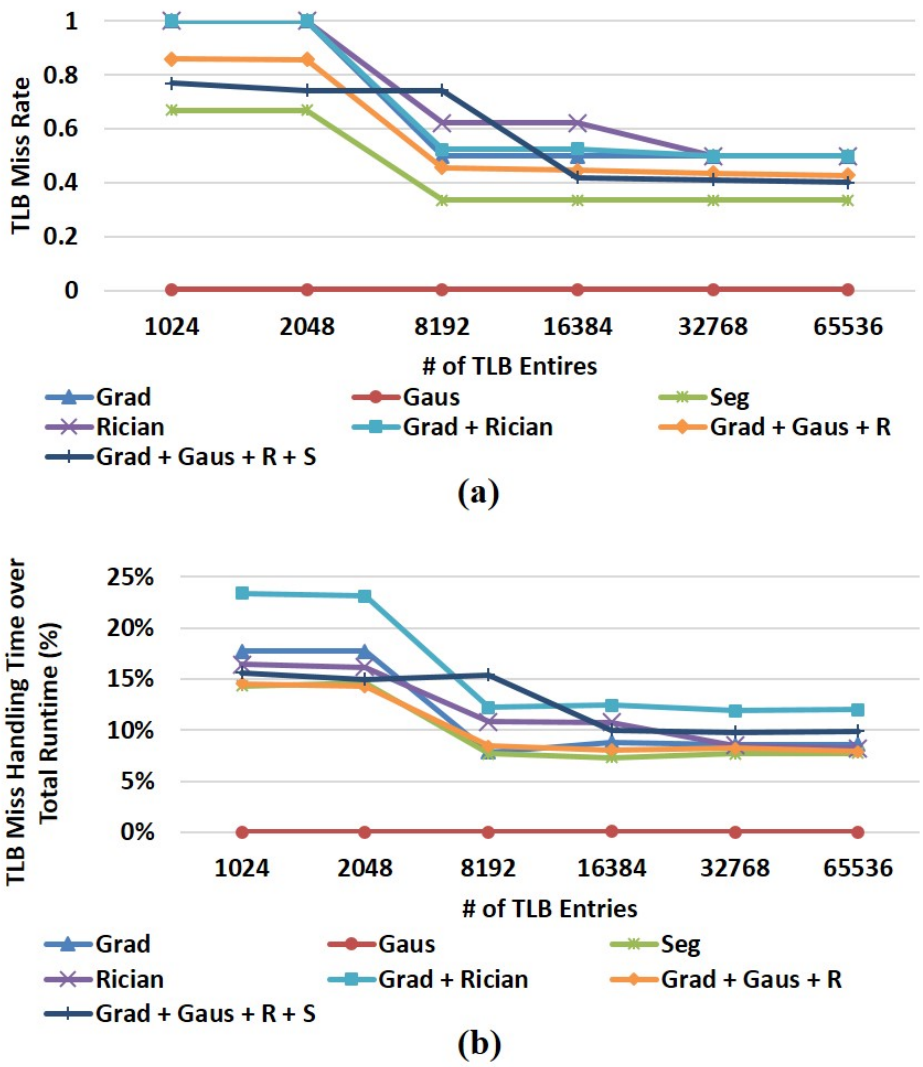


Figure 5.16: The impact of TLB sizes on: (a) TLB miss rates and (b) TLB miss penalty over total runtime.

5.6.4.5 Accelerator Microarchitecture Exploration

Finally, without loss of generality, users can conduct accelerator microchitecture explorations such as 1) algorithm-level changes, and 2) HLS pragmas tuning. Actually,

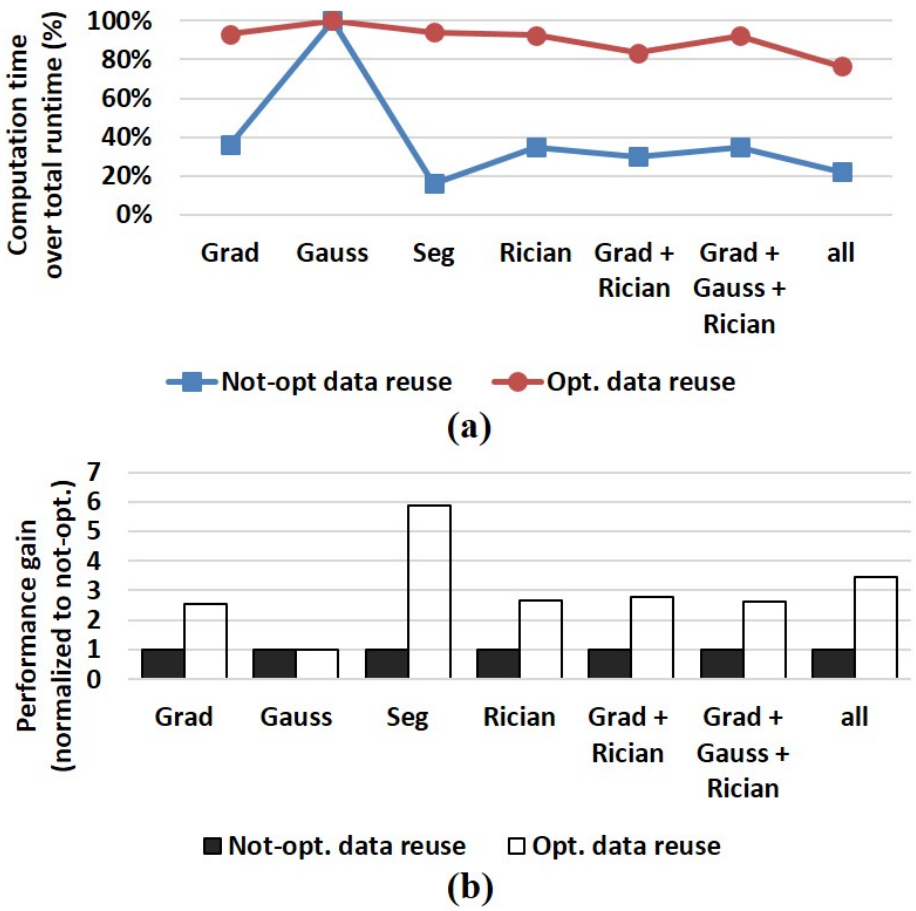


Figure 5.17: Evaluation of accelerator data reuse optimization: (a) the ratio of computation in total runtime; (b) performance speedup.

ARAPrototyper makes the accelerator microarchitecture explorations easier by providing more profiling statistics through performance counters and pointing out the optimization directions.

In this subsection we demonstrate a data reuse optimization for accelerators motivated by the profiled low computation percentage of total execution time in initial accelerator design. As shown in Figure 5.17(a), in our initial accelerator design, before

data reuse optimization, the computation ratio is below 40% for most accelerators, which suggests the accelerators are not fully utilized but are waiting for data. Therefore, we apply the data reuse optimization presented in [58] to the accelerators. After this optimization, the computation ratio can be significantly increased, in most of the cases above 80%. As shown in Figure 5.17(b), the data reuse optimization can achieve up to 6X performance speedup.

5.7 Conclusions

In this work we designed and implemented ARAPrototyper to enable rapid design space explorations for ARAs in FPGA prototypes with manageable efforts. Designers can easily integrate their HLS-compatible accelerator designs into our reusable baseline prototype for a few lines of code, and customize their own ARAs with up to 33 lines of XML code. The memory system of our ARA prototype is highly customizable and enables numerous design space explorations with insights provided by our added performance counters. Furthermore, we provide user-friendly APIs and the underlying system software stack for users to quickly develop their applications and deploy them seamlessly on our prototype. Finally ARAPrototyper achieves a 4,000X to 10,000X faster evaluation time than full-system simulations. We believe that ARAPrototyper can be an attractive alternative for ARA design and evaluation.

The success of ARAPrototyper involves continuous development efforts from our lab. ARAPrototyper inherits some key features in [34] but provides significant modifications to adapt to the Zynq platform in order to support efficient full-system evaluation. The design of the accelerator integration interface in Section 5.4.3 is a collaborative effort between Peipei Zhou and myself. I also thank Dr. Fang for his

feedback on improving ARAPrototyper.

CHAPTER 6

Memory System Optimizations for Accelerator-Rich Architectures

6.1 Introduction

In Chapter 5 we discussed the high-level view of ARAs. We also discussed how to use ARAPrototyper to prototype and evaluate ARA performance and energy efficiently. In this chapter we will discuss the details of the ARA memory system synthesis, which is the key for building an efficient ARA.

Accelerators improve performance by exploiting the application parallelism and data locality. An accelerator utilizes customized and deep pipelines to process a series of data. To maximize performance, the accelerator is usually designed with its initiation interval (II) to be one, i.e., fully pipelined to maximize its throughput. To enable a fully pipelined design, an accelerator must be able to simultaneously fetch multiple data elements from all its owned memory banks within one cycle. This can be easily achieved if all accelerators have their own private memory banks. However, as the number of accelerators in an ARA increases, the on-chip memory resource needs to be shared. The interconnects between the accelerators and shared memory banks have to provide (1) sufficient connectivity, and (2) the fixed one-cycle latency.

Figure 6.1 shows three conventional on-chip shared memory architectures. Multi-level caches (Figure 6.1(a)) are commonly used in multi-core processors. However, the fixed latency demand cannot be met since a cache miss can lead to uncertain access latency. Furthermore, a cache may not be able to service a number of simultaneous requests efficiently, even if interval banking is performed. A conventional SoC uses a system bus (Figure 6.1(b)) to share memory resource. However, when the number of devices increases, significant arbitration latency and area overhead to synthesize memory interfaces become the bottleneck [60]. Other interconnect topologies, such as ring and mesh, usually cannot meet the one-cycle latency constraint. A partial crossbar, shown in Figure 6.1(c), can provide sufficient connectivity and the one-cycle latency with moderate area overhead. In this dissertation we conduct optimization for the partial crossbar for heterogeneous accelerators.

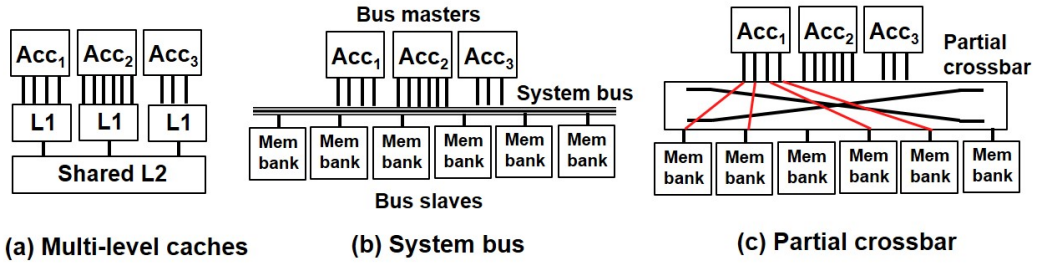


Figure 6.1: Three conventional on-chip shared memory architectures.

Another important issue is how to efficiently fetch the data from off-chip memory. In CPU cores, the off-chip memory accesses are issued when misses occur in the last-level caches. An accelerator improves performance by grouping a series of memory accesses together into a burst request and then issuing multiple long burst requests to prefetch data into its own on-chip memory banks. In an ARA, it is common to have multiple memory ports to fully use the available memory bandwidth. However, if multiple burst requests are issued to the same interface, the outstanding requests

must wait.

In this chapter we aim to design the interconnects to satisfy the need for sufficient connectivity, fixed latency, and efficient off-chip prefetching capabilities for the shared ARA memory system.

Our contributions can be summarized as follows:

- **An Optimal Partial Crossbar Between Heterogeneous Accelerators and Memory Banks:** Assuming the number of accelerators that can be simultaneously powered on is bounded, we provide a novel algorithm to synthesize the interconnect between heterogeneous accelerators and memory banks as a partial crossbar with the minimum number of switches. Also, the number of required memory banks is minimum. Compared to the state-of-the-art synthesis algorithm [60], we further generalize the optimal solution for accelerators with heterogeneous memory bank demands.
- **Interleaved Network Between Memory Banks and Memory Interfaces:** We generate the interleaved network to interleave the simultaneous long burst requests to limited memory interfaces based on the proposed optimal partial crossbar topology. Performance can thus be significantly improved when request conflicts are reduced.

We first evaluate our synthesis algorithm by comparing the number of switches in the partial crossbar with [60], the full-capacity crossbar and full crossbar. Based on experimental results, we can reduce more than 45% of switches compared to the work in [60]. To validate the effectiveness of the reduction of switches, we synthesize our partial crossbar design with 30 accelerators on the Xilinx Zynq platform. Our method can reduce 47% of switches, 53% of LUTs and 34% of slices when compared to [60].

We further demonstrate the efficiency of our interleaved network through a real ARA prototype with heterogeneous medical imaging accelerators. The interleaved network can reduce the execution time 36% - 52% by improving the prefetching process. The ARA prototype also shows a 7.44x energy efficiency gain over the state-of-the-art Xeon processors.

6.2 Preliminary

6.2.1 Accelerator-Rich Architectures

An ARA is composed of the general-purpose cores, heterogeneous accelerators, on-chip memories, and interconnects [160, 122, 55, 41], as discussed in Section 5.3. Figure 6.2 demonstrates the accelerator plane of an ARA, which can be decomposed into the following components: (1) heterogeneous accelerators, (2) shared memory banks, (3) direct memory access controllers (DMACs), (4) physical memory ports (interfaces), and (5) two layers of interconnects [41]. The heterogeneous accelerators can have a different number of memory bank demands. For example, Acc_1 needs four memory banks, while Acc_2 requires six. These heterogeneous demands make the design of interconnects more difficult than that of the "homogeneous" demands.

In the dark silicon era, the power wall limits the number of transistors that can be powered on simultaneously. For an accelerator island, the power budget is determined by (1) the number of powered-on accelerators at a certain time period, and (2) the number of memory banks in this island. The number of powered-on accelerators leads to the dynamic power consumption [60], while the number of memory banks significantly contributes to leakage power [122]. Therefore, the number of on-chip memory

banks and the maximum number of powered-on accelerators should be limited under a given power budget.

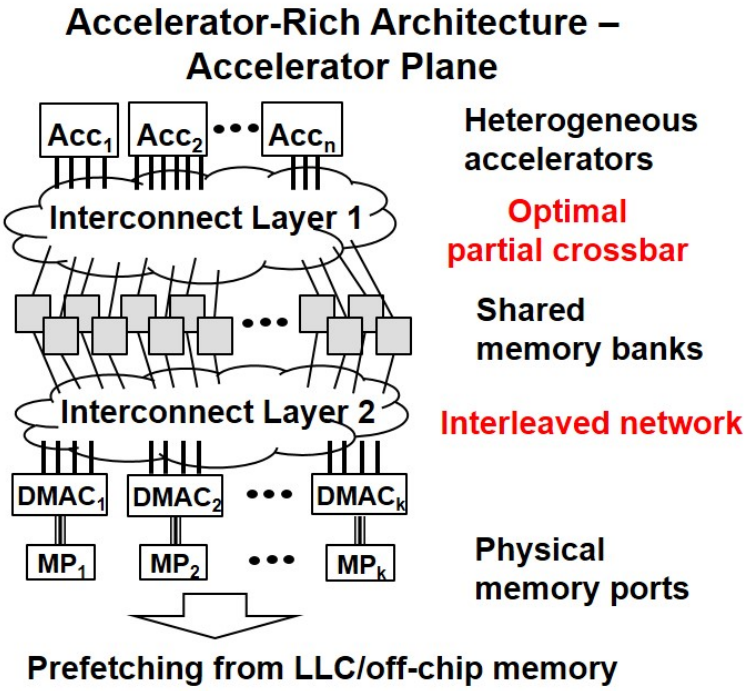


Figure 6.2: An accelerator-rich architecture (ARA)—the accelerator plane.

6.2.2 Limitations of Existing Methods and Motivation

The authors in [109] summarized the complexity for several types of crossbar designs, such as the full-capacity crossbar [132], for general signal routing in FPGA. However, these methods are not efficient enough when the constraint on the number of powered-on accelerators is considered. The work in [60] first investigated the partial crossbar synthesis when the power budget is limited, which is the work that is most relevant to ours. However, the method proposed in [60] can only generate the minimum partial crossbar for accelerators with homogeneous memory bank demands, as shown

in Figure 6.3(a). The homogeneous bank demands do not match the heterogeneity nature of accelerators. Therefore, we are motivated to synthesize the minimum partial crossbar for accelerators with heterogeneous bank demands, as shown in Figure 6.3(b).

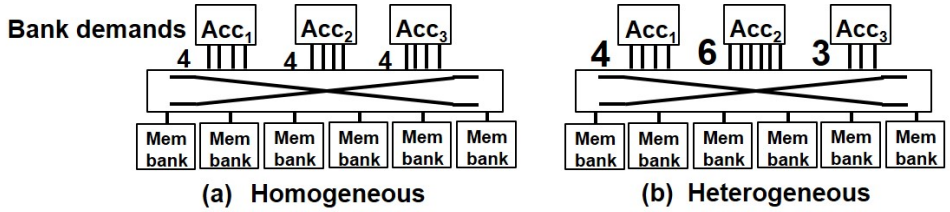


Figure 6.3: Limitation of the previous partial crossbar synthesis method.

The crossbar (bus matrix) is also used to provide sufficient connectivity for buses in high-performance systems [95]. The crossbar network is usually designed in a cascaded fashion, while providing full connectivity [96] or partial connectivity [95]. However, this design style cannot meet the demand for accelerators to fetch multiple data every cycle to maintain high throughput. Similarly, the network-on-chip (NoC) topologies [18][148] and the combination of buses and NoC [61] for large-scale multi-core processors cannot satisfy the high-throughput need.

In this dissertation the first important question that we address is how to synthesize the interconnects between heterogeneous accelerators and shared memory banks. We use the configurable partial crossbar to provide one-cycle fixed latency and guaranteed connectivity. The second question is how to design an interleaved network between memory banks and DMACs based on the optimal partial crossbar topology. The topology synthesis in this layer has not been considered together with a given partial crossbar topology from the existing work.

6.3 Optimal Partial Crossbar Design

In this section we first discuss the crossbar configurability and then define the problem formulation of the partial crossbar synthesis. Following that, we propose a novel algorithm to synthesize the optimal partial crossbar between the accelerators and the shared memory banks. The algorithm guarantees that the number of switches in the partial crossbar is minimum, while supporting at least any c accelerators in the island that can be simultaneously powered on. For simplicity of discussion, we summarize the key notations used in this chapter in Table 6.1.

Table 6.1: Major notations

Notation	Explanation
n	the number of heterogeneous accelerators in the island
a_i	accelerator i, $1 \leq i \leq n$ $\{a_1, a_2, ..., a_n\}$ are sorted in descending order based on the memory bank demand
d_i	the number of memory bank demand of a_i
c	the number of simultaneous powered-on accelerators
m	the number of shared memory banks
b_i	memory bank i, $1 \leq i \leq m$
k	the number of DMACs and MPs

6.3.1 Crossbar Configurability

Figure 6.4 is an example of the partial crossbar design between the heterogeneous accelerators and the shared memory banks. In this example, banks 1 to 4 are assigned to Acc_1, while banks 3 to 8 are assigned to Acc_2. Acc_1 and Acc_2 cannot be powered on simultaneously since they share bank 3 and bank 4. In our assumption, one memory

bank cannot be simultaneously used by two accelerators or any two ports in one accelerator. This is because a memory bank only has two ports. One is connected to an accelerator while the other is connected to a DMAC. Also, an accelerator accesses its own memory banks every cycle to achieve high throughput. The crossbar is designed to be configurable for sharing these two memory banks for Acc_1 and Acc_2. We assume that two-port memory banks are used. One port of a memory bank is connected to the accelerators while the other port is connected to one DMAC.

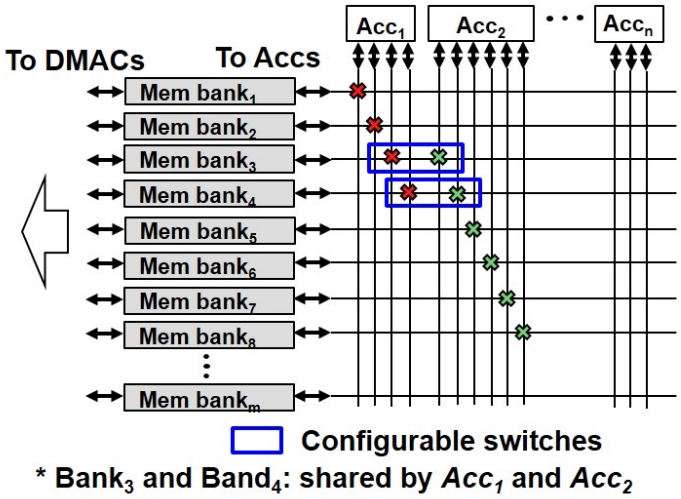


Figure 6.4: An example of a configurable crossbar.

6.3.2 Minimum Required Memory Banks

Suppose that the number of allowed simultaneous powered-on accelerators within the power budget is c. The primary goal is to provide a configurable partial crossbar which makes any c accelerators in this island simultaneously work together. The first question we try to answer is what is the minimum number of required memory banks to support c accelerators to be powered on simultaneously.

Lemma 1. *Given a set of accelerators, $\{a_1, a_2, ..., a_n\}$, with non-increasing memory band demands $d_1 \geq d_2 \geq ... \geq d_n$, and c, the minimum number of required memory banks is at least $\sum_{i=1}^{c} d_i$.*

Proof. The goal is to simultaneously power on any c accelerators in this island. The c accelerators with the largest memory bank demands need to be satisfied. Therefore, at least $\sum_{i=1}^{c} d_i$ memory banks are required. □

6.3.3 Problem Formulation

Given the number of simultaneous powered-on accelerators c, the number of shared memory banks m ($m = \sum_{i=1}^{c} d_i$), and n accelerators, our goal is to minimize the total number of switches of the partial crossbar.

6.3.4 Optimal Partial Crossbar Synthesis

We propose Algorithm 2 to synthesize the partial crossbar, which meets these two requirements: (1) the minimum number of memory banks shown in Lemma 1, and (2) the minimum number of switches equal to the lower bound shown in Theorem 1. In this section we illustrate the high-level concepts of the proposed algorithm. We will discuss (1) the lower bound of the minimum required number of switches and (2) the optimality analysis of the proposed algorithm in Section 6.3.5 and Section 6.3.6, respectively.

We have n heterogeneous accelerators, and only c accelerators can be powered on simultaneously. In Algorithm 2, we first assign the crossbar switches for the c accelerators, $\{a_1, a_2, ..., a_c\}$, with the largest memory bank demand. For these c

Algorithm 2 Optimal partial crossbar synthesis

```
1: port_map: the mapping between the accelerator ports and shared memory banks; 1st dimension:
   the accelerator index; 2nd dimension: the accelerator's port index; the mapping: memory bank
   id
2: d: the array recording the memory bank demands for all accelerators
3: n: the number of accelerators
4: m: the number of memory banks
5: c: the number of simultaneous powered-on accelerators
6: procedure OptCrossbar(d, n, m, c)
7:     bank_index ← 0
8:     for i ← 1 to c do                          ▷ 1st nested loop: for the largest c accelerators
9:         for j ← 1 to d[i] do                   ▷ Assign consecutive memory banks
10:            port_map[i][j] ← bank_index
11:            bank_index ← bank_index + 1
12:        end for
13:    end for
14:    for i ← 1 to c do                          ▷ 2nd nested loop: for the rest n - c accelerators
15:        bank_index ← port_map[i][1]
16:        for j ← c + 1 to n do
17:            if bank_index + d[j] > port_map[i][d[i]] then
18:                bank_index ← port_map[i][1]
19:            end if
20:            for k ← 1 to d[j] do               ▷ Assign consecutive memory banks
21:                port_map[j][k] ← bank_index
22:                bank_index ← bank_index + 1
23:            end for
24:        end for
25:    end for
26:    return port_map
27: end procedure
```

accelerators, one memory port exactly maps to one memory bank (lines 8 - 13). Next, we assign the switches for the remaining $n-c$ accelerators (lines 14 - 25). Each port in accelerators $\{a_c+1, ..., a_n\}$ is mapped to c memory banks. The number of switches generated from Algorithm 2 is $m + c \times \sum_{i=c+1}^{n} d_i$ (Theorem 2), which is the minimum number of required switches (Theorem 1). Note that the memory ports of an accelerator are assigned in a contiguous way to one *Region* (Definition 1) for accelerators $\{a_1, a_2, ..., a_c\}$ or to c *Regions* for accelerators $\{a_c+1, ..., a_n\}$. Figure 6.5(a) shows an example of the partial crossbar topology generated from Algorithm 2.

6.3.5 The Lower Bound of the Required Switches

To design the minimum (optimal) partial crossbar, we first find the lower bound of the minimum required switches and then provide a proof for this lower bound. Following that, we will provide an algorithm to synthesize the partial crossbar with the minimum switches.

Theorem 1. *The lower bound for the number of switches required in the partial crossbar is* $m + c \times \sum_{i=c+1}^{n} d_i$, *where* m *is equal to* $\sum_{i=1}^{c} d_i$.

Proof. Based on Lemma 1, the number of required memory banks is at least $m = \sum_{i=1}^{c} d_i$. To power on the c accelerators with the largest memory demands ($\{a_1, a_2, ..., a_c\}$), we need exactly m switches to map each port from $\{a_1, a_2, ..., a_c\}$ to one memory bank. Considering the case where one accelerator in $\{a_{c+1}, a_{c+2}, ..., a_n\}$ and any (c - 1) accelerators out of $\{a_1, a_2, ..., a_c\}$ are powered on simultaneously, the accelerator a_j in $\{a_{c+1}, a_{c+2}, ..., a_n\}$ needs to have at least c copies of memory banks ($c \times d_j$) to avoid the memory bank conflicts. Therefore, we require at least $c \times \sum_{i=c+1}^{n} d_i$ switches for all

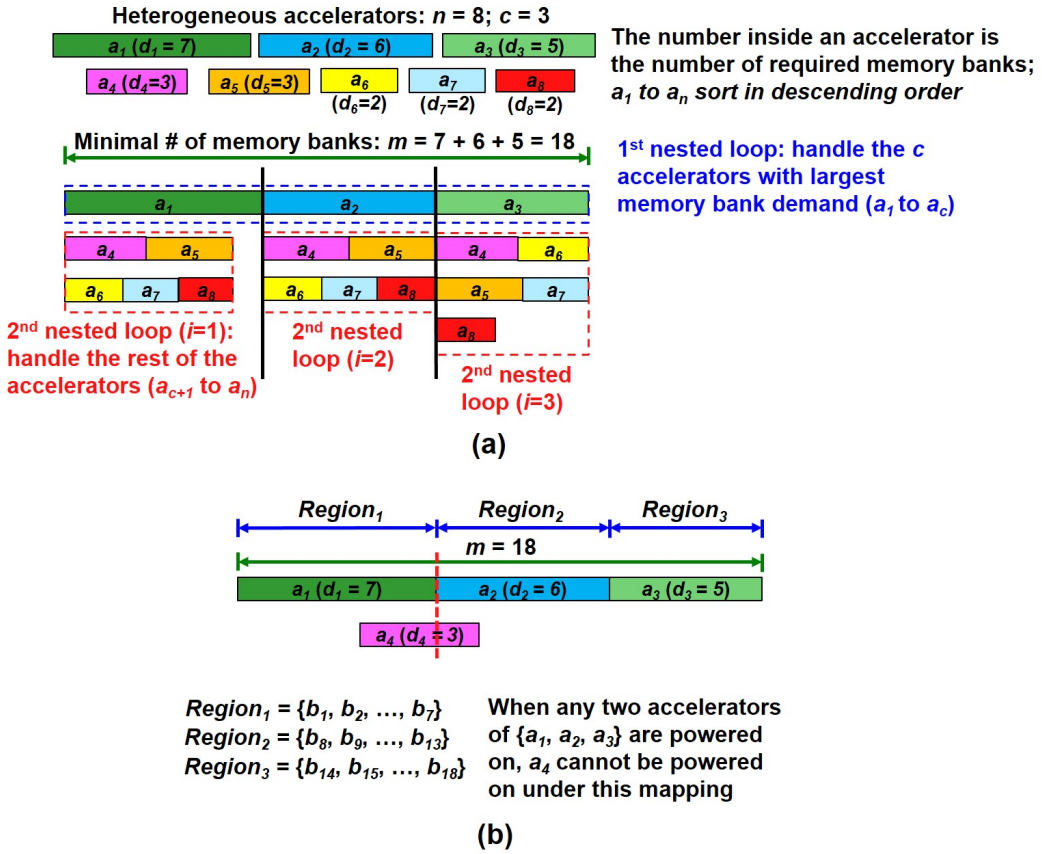


Figure 6.5: (a) An example of partial crossbar synthesis using Algorithm 2. (b) An example demonstrating the insight of Algorithm 2 design.

accelerators in $\{a_{c+1}, a_{c+2}, ..., a_n\}$ to support c simultaneous powered-on accelerators. The lower bound of the required switches is $m + c \times \sum_{i=c+1}^{n} d_i$. □

Theorem 1 can be easily used to find the minimum number of switches when all accelerators have the same number of memory bank demands, as described in [60]. We use r to denote the homogeneous memory bank demand. The minimum required memory banks, m, is equal to $c \times r$. As demonstrated in Equation 6.1, Theorem 1 can further generalize the theorem derived from [60] for synthesizing accelerators

with homogeneous bank demands.

$$\begin{aligned} m + c \times \sum_{i=c+1}^{n} d_i &= m + c \times (n - (c+1) + 1) \times r \\ &= c \times r + c \times (n - c) \times r = m \times (1 + n - c) \end{aligned} \tag{6.1}$$

6.3.6 Algorithm Optimality Analysis

In this section we first prove that Algorithm 2 can synthesize a partial crossbar with a minimum number of switches by Theorem 2.

Theorem 2. *Algorithm 2 synthesizes the partial crossbar with* $m+c\times \sum_{i=c+1}^{n} d_i$ *switches, which is equal to the lower bound described in Theorem 1.*

Proof. Based on Algorithm 2 (lines 9 - 14), the number of required switches for the c accelerators with the largest bank demand is $\sum_{i=1}^{c} d_i = m$. The number of required switches for the rest of $n - c$ accelerators is $c \times \sum_{i=c+1}^{n} d_i$, according to the second nested loop in Algorithm 2. Therefore, we can synthesize the partial crossbar with $m + c \times \sum_{i=c+1}^{n} d_i$ switches. □

To further prove that the crossbar generated from Algorithm 2 can power on any c accelerators, we first define the term *Region*.

Definition 1. *"Regions" are the ranges for the contiguous memory bank assignments for the c accelerators with the largest memory bank demands. All the c Regions are not overlapped with one another based on the Algorithm 2.*

Lemma 2. *For any two accelerators, if their bank assignments reside at two different Regions, i.e., non-overlapped Regions, the two accelerators can be powered on simultaneously. (We suppose the other accelerators are currently powered off.)*

Lemma 3. *Based on Lemma 2, a partial crossbar with c Regions can support at least c accelerators that are powered on simultaneously.*

Based on Definition 1, we have three *Regions* for the synthesized partial crossbar shown in Figure 6.5(a). We can further deduce Lemma 2 and Lemma 3 based on the definition. The key insight of Algorithm 2 is to avoid the case of the cross-Region bank assignment such as a_4, shown in Figure 6.5(b). By using Algorithm 2, the bank assignment of the accelerators is aligned well inside the *Regions*, which avoids the cross-Region assignment. We show that Algorithm 2 can provably power on any c accelerators by Theorem 3.

Theorem 3. *The crossbar generated from Algorithm 2 can power on any c heterogeneous accelerators simultaneously.*

Proof. Omitted. The following three cases, which cover all possible conditions, provide the proof for Algorithm 2.

Case 1: The c accelerators are all from $\{a_1, a_2, ..., a_c\}$.

Based on Lemma 3, the c accelerators can be powered on simultaneously since they are allocated at three different *Regions*.

Case 2: The c accelerators are all from $\{a_{c+1}, a_{c+2}, ..., a_n\}$.

Based on Algorithm 2, each accelerator in the group owns c copies of memory banks distributed in the c *Regions*. Therefore, we can provide a given order to assign these c accelerators to the c *Regions*, and thus the accelerators can all be powered on (Lemma 3).

Case 3: The w accelerators are from $\{a_1, a_2, ..., a_c\}$ while the rest of $c-w$ accelerators are from $\{a_{c+1}, a_{c+2}, ..., a_n\}$.

We first assign the w accelerators to their corresponding *Regions*. The rest of $c-w$ accelerators still own $c-w$ copies of memory banks in the rest of $c-w$ *Regions*. Therefore, we can provide a given order to assign these $c-w$ accelerators to the rest of $c-w$ *Regions* to power on all c accelerators (Lemma 3). □

6.4 Interleaved Network

6.4.1 Conflicts of Burst Prefetch Requests

In an ARA, the off-chip data prefetching memory access patterns have the following properties. First, the prefetch requests issued from accelerators are usually long memory bursts to prefetch a group of consecutive data elements for performance concerns. In our design, it is at the page granularity (4KB), ranging from one to four pages (4 - 16KB). Second, an accelerator issues multiple burst requests simultaneously. The number of simultaneous burst requests depends on the memory bank demands of an accelerator; this ranges from 5 to 12 in our design. Third, the partial crossbar described in Section 6.3 maps the ports of an accelerator to contiguous memory banks. The interconnects between memory banks and DMACs need to be carefully designed considering the partial crossbar topology.

Figure 6.6 shows a real topology synthesized using Algorithm 2. In this example, a_1 is powered on and six simultaneous burst requests are issued from a_1 to prefetch the required input data into $\{b_1, b_2, ..., b_6\}$. If the interconnect layer between memory banks and DMACs is not designed carefully, request conflicts will arise. Figure 6.6(a)

shows that four burst requests are sent to $DMAC_1$, while two requests are sent to $DMAC_2$. The four requests issued to $DMAC_1$ will become the bottleneck since MP_1 services the requests in a sequential way. Each request is a 4KB to 16KB long burst request, which leads to a significant performance degradation.

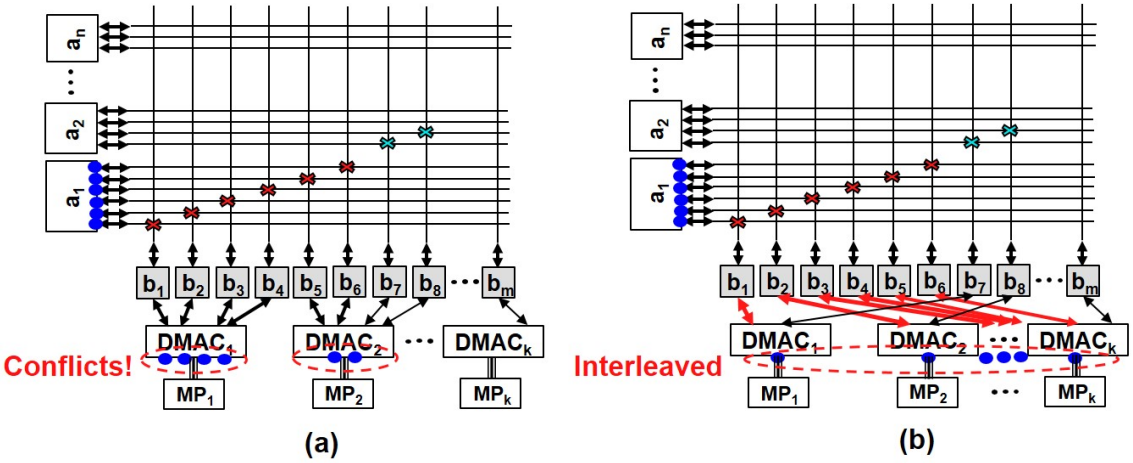


Figure 6.6: (a) Burst requests conflict at $DMAC_1$. (b) Burst requests are interleaved to different DMACs.

6.4.2 Interleaved Network Design

We consider the following important properties for interleaved network design in order to resolve possible conflicts.

1. Partial crossbar topology. According to Algorithm 2, the memory banks assigned to an accelerator are contiguous. We design the mapping function described in Equation 6.2 to map the memory banks to DMACs. The mapping function can guarantee the distribution of simultaneous burst requests distributed uniformly to different DMACs. Figure 6.6(b) shows an example of our interleaved network design. In this example ($k = 6$), the memory banks $\{b_1, b_2, ..., b_6\}$ are mapped to

$\{DMAC_1, DMAC_2, ..., DMAC_6\}$, respectively. The six interleaved burst requests can be serviced in parallel with available memory bandwidth without conflicts.

$$MemBankToDMAC(i) = i \bmod k, \quad i \in 1..m \tag{6.2}$$

2. Accelerator usage pattern. We believe that the priority of interleaving requests within an accelerator is more important than that of interleaving requests across accelerators. This is because not all of the accelerators can start simultaneously with limited power budgets. The heterogeneous nature of accelerators also reduces the possibility that accelerators launch simultaneously. Even if multiple accelerators are running simultaneously, the requests from multiple accelerators may still interleave. However, a single accelerator cannot start to work until all data are prefetched and ready. Therefore, we choose the topology generated in Equation 6.2 to ensure that the burst requests of a specific accelerator can be distributed uniformly across DMACs.

6.5 Experimental Results

6.5.1 Case Study: A Real Medical Imaging ARA Prototype

6.5.1.1 Applications and Prototyping Platform

We are interested in accelerating the medical imaging processing pipeline for computerized tomography (CT) images [25]. First, the noise and blur need to be removed. Second, the process would align the current image with previous images of an individual. Third, a region of interest for diagnosis is segmented. To accelerate the pipeline, we include four accelerator kernels—*gradient*, *gaussian*, *rician*, and *segmentation*

in our ARA design.

We use the Xilinx Zynq ZC706 as our prototyping platform. The Zynq SoC, which is composed of a dual-core ARM Cortex-A9 and FPGA fabrics, is used to prototype an ARA (Figure 6.2). We use ARAPrototyper [41, 34] to prototype the ARA for the medical imaging pipeline. In this ARA, we have five heterogeneous accelerators: (1) two *gradient*, (2) one *gaussian*, (3) one *rician*, and (4) one *segmentation*. The memory bank demand of each accelerator is shown in Table 6.2. The shared memory banks are synthesized using the on-chip BRAMs, while both interconnect layers are realized using FPGA LUTs and routing resources. In Zynq, there are four physical memory ports ($k = 4$) for accelerators to prefetch data from off-chip DRAM.

Table 6.2: Memory bank demands (i.e., the number of ports of each accelerator)

Type	gradient	gaussian	rician	segmentation
Bank demand	6	5	8	12

6.5.1.2 Optimal Partial Crossbar

Figure 6.7(a) shows the minimum number of required memory banks, while Figure 6.7(b) shows the number of switches generated from Xiao's algorithm and Algorithm 2. We set c to four in the ARA prototype, and thus 8.8% of switches are saved.

6.5.1.3 Effectiveness of Interleaved Network

To evaluate the effectiveness of the interleaved network, we measure performance from our FPGA prototype. This is because the request conflicts occur during runtime based on the accelerator utilization, which is difficult to model in an analytical way. We

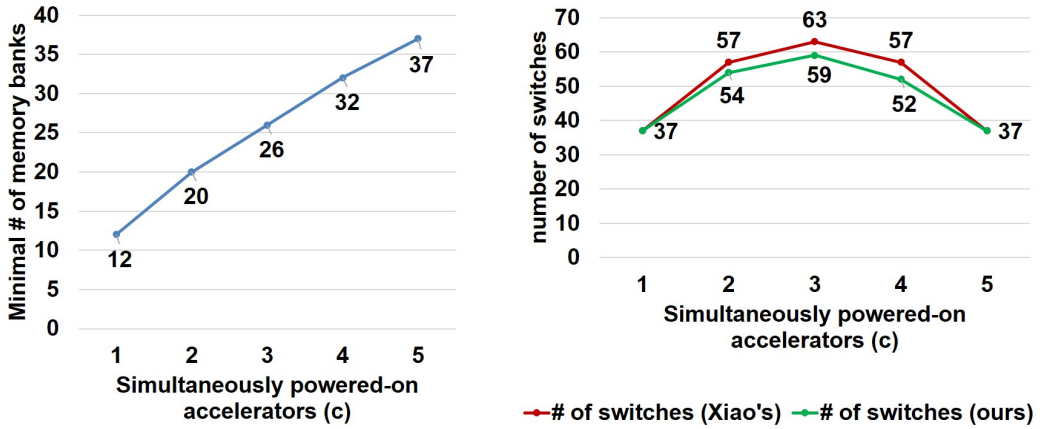


Figure 6.7: (a) The minimum number of required memory banks for this ARA. (b) The number of switches generated from Xiao's algorithm and our algorithm.

design the interleaved network based on the partial crossbar topology generated from Algorithm 2. We compare our interleaved network design from Equation 6.2 to a non-interleaved design described in Equation 6.3. The non-interleaved mapping is similar to the case demonstrated in Figure 6.6(a). The interleaved network can reduce the runtime from 36% to 52%, as shown in Figure 6.8. This is because the interleaved network improves the performance of prefetching by utilizing the available memory bandwidth better.

$$MemBankToDMAC(i) = i \text{ div } k, \quad i \in 1..m \tag{6.3}$$

6.5.1.4 Energy-Efficiency of the ARA Prototype

Table 6.3 shows the performance and power results of the *denoise* application in our ARA prototype and the state-of-the-art processors. *denoise* is composed of *gradient*

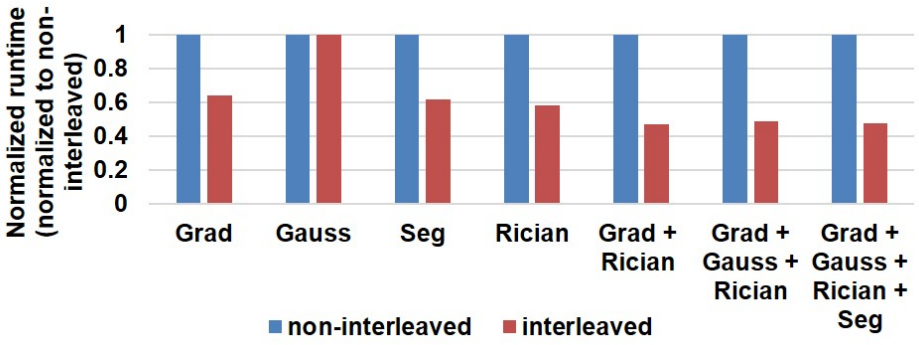


Figure 6.8: Effectiveness of interleaved network

and *rician* kernels and executes the two kernels sequentially for 10 iterations. We use OpenMP to implement *denoise* for evaluating Xeon processors. The result of Cortex-A9 uses single thread. *denoise* is compiled using gcc with -O2 option. Table 6.3 shows that our prototype can achieve 7.44x and 2.22x energy efficiency over Xeon and ARM, respectively. As reported in [103], the power gap between FPGA and ASIC is around 12X. If our ARA is implemented in ASIC, 24-84X energy savings over Xeon processors are expected.

Table 6.3: Performance and power comparison over (1)ARM Cortex-A9, (2)Intel Xeon (Haswell), and (3)ARA

	Cortex-A9	Xeon (24 threads)	ARA
Freq.	667MHz	1.9GHz	Acc@100MHz CPU@667MHz
Runtime(s)	28.34	0.55	4.53
Power	1.1W	190W(TDP)	3.1W
Total Energy	2.22x	7.44x	1x

6.5.2 Scalability Study of the Optimal Crossbars

To evaluate our crossbar design, we compare the number of switches of (1) full crossbar, (2) full-capacity crossbar [132], (3) the crossbar generated from [60], and (4) the crossbar generated from Algorithm 2 over different numbers of powered-on accelerators. A full-capacity crossbar can only connect any m inputs to m outputs, which is less flexible than a full crossbar. In our case, m is the number of memory banks. The algorithm in [60] can only guarantee the minimum design when all accelerators have the same memory bank demands. Note that the number of switches is measured based on the minimum required memory banks derived from Lemma 1 for all the cases in Figure 6.9 and Figure 6.10. We generate the memory demand of each accelerator at random, ranging from 4 to 16.

We evaluate Algorithm 2 using three sizes of designs: (1) 10 accelerators, (2) 50 accelerators, and (3) 100 accelerators, as shown in Figure 6.9. For larger designs, Algorithm 2 generates only about 1% to 10% of switches for full crossbar and full-capacity crossbar, respectively. Compared to the state-of-the-art method [60], we can reduce the number of switches by more than 45% in larger designs, as shown in Figure 6.9(b)(c). This means that the method in [60] is still far from the optimal solution for many cases.

Note that the method in [60] can only generate optimal results when $c = 1$ and $c = n$, where n is the number of accelerators in the ARA. For all the other cases, only Algorithm 2 can generate a crossbar with the minimum number of switches and outperform [60]. When c is equal to 1, the accelerator with the largest memory demand shares its memory banks with all the other accelerators. When c is equal to n, each accelerator has its own memory banks. In both cases, the number of switches

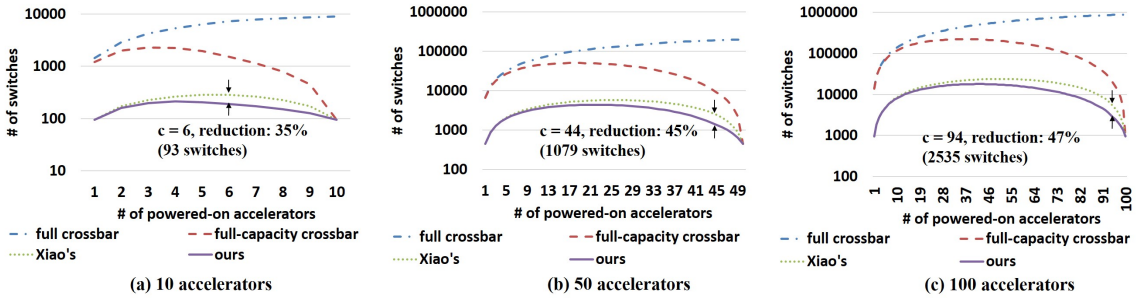


Figure 6.9: The number of switches of (1) the full crossbar, (2) the full-capacity crossbar, (3) the crossbar generated using Xiao's algorithm, and (4) the crossbar generated using Algorithm 2, over different numbers of powered-on accelerators.

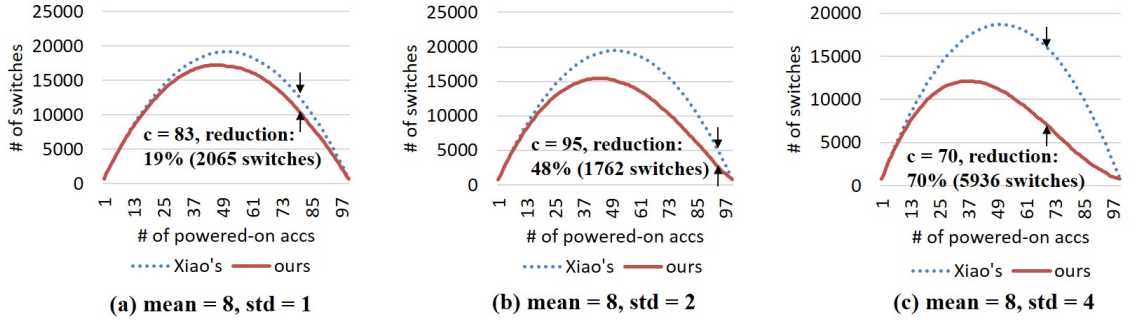


Figure 6.10: The comparison of Algorithm 2 and Xiao's algorithm based on the different memory demand deviations.

is equal to the sum of the memory bank demands of all accelerators.

We further evaluate our algorithm based on the different deviations of memory bank demands. We generate the bank demands of 100 accelerators under normal distribution, and the mean of the demands is equal to eight. We vary the standard deviations from one, two, to four. As shown in Figure 6.10, our algorithm becomes much more effective when the deviation increases. Note that when the standard deviation is equal to zero, both Algorithm 2 and [60] can achieve the optimal solution.

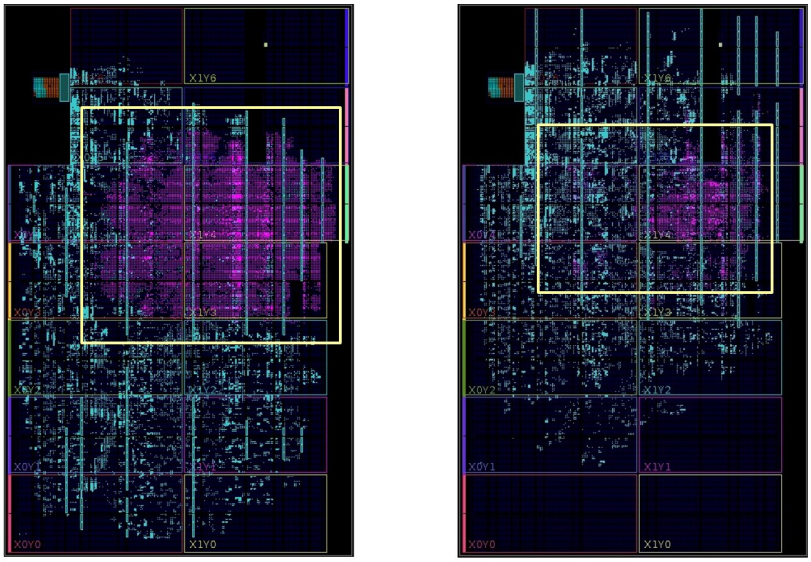
6.5.3 FPGA Validation of the Partial Crossbars

To further validate the effectiveness of switch reduction, we prototype the partial crossbar designs generated from Algorithm 2 and [60] on the Xilinx ZC706. We generate a large design with 30 heterogeneous accelerators with 156 memory banks, and only 20 accelerators can be powered on simultaneously (n=30, c=20, m=156). In order to save the FPGA resource for synthesizing the partial crossbar, we use dummy accelerators instead of real ones. Figure 6.11 shows the resource utilization of the FPGA of the generated partial crossbars of Algorithm 2 and [60]. Algorithm 2 can generate a much less congested partial crossbar by reducing 47% of switches, which leads to 53% LUT and 34% slice reduction when compared to [60].

Table 6.4 shows the resource utilization, including LUTs and slices, of the FPGA prototypes on six configurations. The memory demand of each heterogeneous accelerator is generated under normal distribution ($N(8, 4)$). In order to save the resource for synthesizing the partial crossbar, we use dummy accelerators instead of real accelerators.

Table 6.4: Evaluation of the resource utilization of partial crossbar on FPGA (n: the number of accelerators, c: the number of simultaneously powered-on accelerators, m: the number of memory banks)

	[60]			Algorithm 2			Reduction by using Alg. 2		
Configurations	switches	LUTs	slice(%)	switches	LUTs	slice(%)	switch(%)	LUT(%)	slice(%)
n=30, c=10, m=92	1284	65K	33	1009	50K	25	21.42	23.68	24.24
n=30, c=15, m=128	1479	77K	40	965	49K	25	34.75	37.19	37.50
n=30, c=20, m=156	1348	82K	41	713	38K	27	47.11	53.27	34.15
n=50, c=10, m=120	3041	145K	41	2628	122K	29	13.58	16.32	29.27
n=50, c=16, m=180	4187	NA	NA	3204	153K	33	NA	NA	NA
n=50, c=28, m=280	4836	NA	NA	2799	142K	34	NA	NA	NA



(a) FPGA snapshot using Xiao's alg. (b) FPGA snapshot using Algorithm 2

Figure 6.11: Snapshots of the Zynq FPGA using Xiao's algorithm and (b) Algorithm 2 for partial crossbar synthesis for the configuration (n=30, c=20, m=156). The purple color represents the resources used by the partial crossbar, while the blue color represents the rest of utilized resources.

Table 6.4 demonstrates that the switch reduction leads to lower resource utilization on FPGA, including both LUTs and slices for FPGA routing. Compared to [60], our algorithm can effectively reduce the number of LUTs by 16% - 53% and slice utilization by 24% - 38%. It also shows that the number of switches is a good estimate for resource utilization. Note that for the two larger designs ([n=50, c=16, m=180], [n=50, c=28, m=280]), the method in [60] cannot generate a feasible partial crossbar. This is because the generated design is too congested, and thus the Xilinx router cannot find a feasible solution. Therefore, our algorithm is much more scalable than [60] when the number of accelerators and the powered-on constraints are large. In Figure 6.11, we can see that our algorithm can generate a less congested partial crossbar with less resource utilization.

6.6 Conclusions

In this work we first provide an optimal partial crossbar synthesis algorithm that guarantees the required number of switches to be minimum while supporting at least a given number of accelerators that can be powered on simultaneously. Second, we improve the data prefetching efficiency by interleaving multiple simultaneous long burst requests into different physical memory ports for better bandwidth utilization. With the optimal synthesis algorithm, we can reduce 45% of switches when compared to previous work. Our prototyping results show that we can improve performance by 36% - 52% using the interleaved network for a real ARA.

CHAPTER 7

Datacenter-Level Optimizations for Cluster Computing Frameworks

7.1 Introduction

In the era of "big data," the amount of data grows exponentially and requires scalable technology to process and store the data in a reliable way. With a significant amount of data, valuable information can be found by intelligently analyzing the data. The major goal of this dissertation is to provide customized solutions for our target application domains. In this chapter we will use an important application, read alignment, in the genomics domain to demonstrate our proposed optimizations in the datacenter level. The domain of genomics applications that we are targeting is inherently a big data domain. For example, in the DNA sequencing pipeline, the raw data of an individual obtained from a sequencer before processing is around 300GB to 500GB. In order to improve performance of sequencing, the data and computations need to be processed in an efficient and scalable way. We need the capability to tune performance according to the performance target. The problem becomes more difficult when we have multiple individuals to analyze. The number of individuals can be from 100 to above one million. Under this problem scale, the data cannot even be stored in a single node. Therefore, a scalable methodology is required for the applications with

significant amounts of data to be processed. With a significant amount of data-level parallelism, our applications can utilize a big data infrastructure such as Spark to exploit data-level parallelism.

In Section 1.5.2 we discussed that read alignment is the first step in the DNA sequencing pipeline for variant discovery, as shown in Figure 1.3. Aligning the reads to a reference genome is usually one of the most time-consuming steps in many genomics pipelines, such as the pipelines for variant discovery [129] and differential gene expression [158]. Recent high-throughput sequencing systems, such as Illumina HiSeq X Ten, can deliver more than 18,000 genomes annually. As the data throughput increases—more than doubling each year—a fast and scalable aligner is needed to align the ever-increasing data.

The state-of-the-art aligners, such as BWA-SW [113], Bowtie2 [105], and BWA-MEM [111], can perform gapped-read alignment while using pair-end reads to improve alignment quality. These aligners usually proceed in two steps for each read. The first step is called the seeding process. The full-text minute index (FM-index) [69] and Burrows-Wheeler transform [26] (BWT) are used to find the possible candidate locations on the reference genome of a read very efficiently. These ungapped and exact-matched alignments collected from a read are used as seeds in the second step. To allow gaps, such as mismatches, insertions, and deletions, the Smith-Waterman algorithm [152] is applied to extend the seeds with gaps. The score of each can be used to filter out poor gap-allowed alignments. However, the seed extension step involves significant computation since dynamic programming is used. Recent aligners, like Bowtie 2 and BWA-MEM, leverage the efficiency of single-instruction multiple-data (SIMD) built into modern processors to accelerate this compute-intensive part.

Though the efficiency of the aligners can be significantly improved by the FM-

index aided method together with the SIMD acceleration on dynamic programming algorithm, the tremendous number of reads delivered from sequencers nowadays still takes a significant amount of time to be aligned using existing aligners. Existing aligners use multi-threaded parallelization to boost performance. However, they can only use the computation power from a single server, which is limited when the number of reads to be processed is significantly large. For the deep-coverage whole-genome sequencing (WGS) data, billions of reads can be generated in one run. The alignment of 30x coverage WGS data can take more than 10 hours if BWA-MEM is used on a single server with two state-of-the-art Intel six-core processors. In fact, the alignment process is highly parallelizable but we haven't seen the prior work using private or public cloud to fully exploit the huge parallelism arising from billions of reads to accelerate the alignment process. To build a cluster-scale aligner, the first difficulty is the software development complexity. BWA-MEM, the most widely used aligner for DNA sequencing, is composed of 15,741 lines of C code. We need to analyze the tool and develop the software under the MapReduce programming model [62]. In order to make the tool adoptable in the bioinformatics community, the second difficulty will be to achieve almost the same alignment quality over BWA-MEM. It takes a significant effort to develop this software.

In this chapter, we developed a new aligner, CS-BWAMEM, which can leverage the abundant computing resources in a computing cluster. CS-BWAMEM uses the BWM-MEM algorithm and is developed using the MapReduce [62] programming model, which is one of the most frequently used models in developing distributed software packages in modern datacenters. CS-BWAMEM is built on top of Apache Spark [172, 171], which realizes a highly efficient MapReduce system that can cache intermediate data in memory to avoid unnecessary disk I/O accesses. By using Spark, CS-

BWAMEM can distribute read alignment tasks to multiple servers within a cluster, providing the capability to accelerate the alignment process beyond one single server. Therefore, CS-BWAMEM can provide scalable and ultra-fast alignment speed. Users can deploy CS-BWAMEM based on the computing resources they have in a cluster or the runtime target they set up. With such scalability, CS-BWAMEM can finish deep-coverage (30x) whole-genome sequencing within 32 minutes by using a 30-node cluster.

Our contributions can be summarized as follows.

- **Ultrafast and scalable aligner—CS-BWAMEM:** CS-BWAMEM is built on top of the in-memory cluster computing system, Spark. With a 30-node cluster deployment, CS-BWAMEM is 18x faster than the state-of-the-art aligner, BWA-MEM.
- **Broadcast avoidance of the large human reference genome:** We avoid broadcasting large human reference genome data to every node before computation. Instead, we load it from its local disk of each Spark worker during runtime. The time complexity of broadcast can be limited to constant time instead of linear time or logarithmic time.
- **Use of column store:** We adopt the column store to the raw sequences (FASTQ format) for better compression and partial data access.
- **Batch processing for reducing data transfer overhead:** In order to efficiently utilize accelerators or co-processors, we group small tasks into a batch, and the accelerators then can compute tasks in a batched way to avoid the overhead introduced by fine-grained data transfers. We used the Intel SSE to accelerate the Smith-Waterman algorithm as a case study in this chapter. We will provide more details of using FPGA accelerators in the cluster in Chapter

8.

- **I/O performance improvement:** We show that I/O time accounts for a significant amount of runtime in CS-BWAMEM. We optimize I/O efficiency by using (1) caching intermediate data in memory and (2) pipelining file system I/O with MapReduce computation.

By using all the customized strategies in a 30-node cluster, CS-BWAMEM can align the whole-genome in 32 minutes, which is about 18x faster than the state-of-the-art aligner, BWA-MEM, running on a single server.

In this chapter we first discuss the background of state-of-the-art read aligners and cluster computing frameworks. Next, we describe the software architecture of CS-BWAMEM, where CS-BWAMEM provides a valuable case study to demonstrate the effectiveness of using the in-memory cluster computing system. After that, we introduce the details of the customized optimizations for improving performance for CS-BWAMEM. Finally, we demonstrate the effectiveness of our optimizations in the experimental result section.

7.2 Background

7.2.1 Big Data Infrastructures

In recent years, big data analysis and storage systems have burgeoned in the data-centers in order to handle large-scale problems. Google's MapReduce [62] and Google File System (GFS) [73] provide a simple programming model and a storage layer for developers to store data and distribute computations over a cluster with more than thousands of nodes. These infrastructures provides a scalable and efficient solution

for applications with a huge amount of data. However, Google's MapReduce and GFS are not open-source software and are used only internally in Google. Hadoop [163], with both a MapReduce framework and the Hadoop Distributed File System (HDFS) [151] implemented in Java, is the most widely used open-source solution for big data analysis.

However, Google's MapReduce and Hadoop have a limitation on iterative algorithms which reuse a working set of data across multiple iterations. Machine learning algorithms such as K-means clustering and logistic regression are the key examples. In MapReduce and Hadoop, the reused data set and the output after a single iteration need to be written back to a distributed file system, which incurs substantial overheads on network bandwidth, disk I/O, and data serialization. Because of the reliability concern, the distributed file system maintains several copies of data, which makes the problem even worse. In order to solve the problem, Spark [172, 171], an in-memory cluster computing framework, is proposed to target the applications with reused data. The key idea is to try to keep the input data and intermediate output results across iterations in memory instead of writing them back to the distributed file system.

7.2.1.1 Spark: An In-Memory Cluster Computing System

For iterative algorithms and iterative machine learning models, the input data or training are reused many times in each iteration. The intermediate data generated from one iteration can be reused in the next iteration. However, in most current frameworks, the only way to reuse data between iterations is to write this intermediate data back to a reliable file system, such as a distributed file system like GFS or HDFS. This incurs significant overheads due to data replications, disk I/O, and

data serialization. The shared input data also needs to be read from HDFS in each iteration. Instead of writing data back to the distributed file system on disks, the capability to cache the reused data in memory can significantly improve performance.

Spark provides a framework for programmers to develop iterative algorithms with data reuse among iterations in the MapReduce programming model. Spark provides a new abstraction, resilient distributed datasets (RDDs), which is a read-only collection of objects partitioned across a set of nodes in a cluster. RDDs are fault-tolerant and parallel data structures. Users can explicitly specify that RDDs be persisted in memory so that the RDDs are not written back to the distributed file system, resulting in better performance. RDDs guarantee data reliability through recomputation on faulty elements instead of replication, which also significantly improves performance [171].

7.2.1.2 Parquet: Column-Oriented Storage at Cloud Scale

The MapReduce programming model provides the capability to exploit data-level parallelism. Multiple data objects can be processed simultaneously in a cluster. These independent data objects are called *records*, which are stored in a distributed file system, such as HDFS. The MapReduce underlying system can fetch records from the distributed file system, and then the user MapReduce program can process the records in a parallel way. This type of storage method is called record-oriented storage.

However, when the data structure of a record becomes complicated, not all of the field of a record would be used in computation. Only a subset of the fields is required for a specific type of data analysis. Compared to the flattened record-oriented data storage, column-oriented storage, which has been studied in the database field, is

another way to store data and retrieve data efficiently [11]. Google's Dremel [130] is the first system to provide the nested data representation for the column-oriented storage at cloud scale. Parquet [4] is an open-source implementation that inherits the main spirit of Dremel.

Figure 7.1 illustrates the record-oriented storage and column-oriented storage with the same data structure. FASTQ [48] is the input format of the raw reads to be sequenced. A FASTQ record contains five fields: *name*, *seq*, *quality*, *seqLength*, and *comment*. In the record-oriented storage, FASTQ records are stored in the granularity of a record, as shown in Figure 7.1(b). Every time, all the fields in a record are fetched before computations start. In the column-oriented storage, data of the same field are stored together, as shown in Figure 7.1(c). The column-oriented storage forms a tree-like structure. Programmers can fetch only a few columns (fields) to reduce I/O demand.

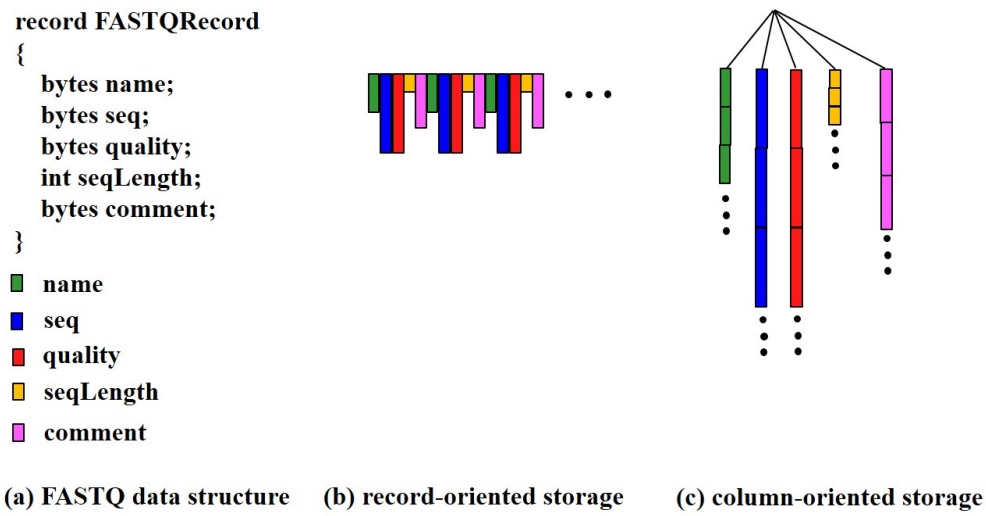


Figure 7.1: Record-oriented storage vs. column-oriented storage

The major benefits of column stores are as follows:

- **Nested representation:** The data for analysis are usually complex and can contain many fields. The organization of data can be hierarchical. Therefore, a general and flexible way to represent data is required. Parquet provides nested representation for describing data flexibly.
- **In situ data access:** *In situ* refers to the capability to access data "in place," which means, in this case, the ability to access data in a distributed file system. Parquet can work seamlessly with HDFS and provides APIs that can be used for Hadoop and Spark programmers. Programmers can easily develop MapReduce programs and obtain structure data.
- **On-demand data fetching:** Column-oriented storages pack the items of the same field together. It is common that only a few fields of a record are required during computation. Programmers only need to fetch the required fields of the records. Therefore, significant disk I/O and network bandwidth can be reduced.
- **Better compression and less storage:** The data items in the same field share the same properties. Therefore, column-oriented storages can have higher compression rates to compress items in the same column. This reduces the storage demand and improves the disk I/O and network efficiency.

We are the first to use Parquet for storing FASTQ format data. The ADAM [127] project also uses Parquet to store the ADAM format data, which is similar but not 100% compatible to the SAM format. Another related work is Hadoop-BAM. Hadoop-BAM is a library for manipulating the SAM/BAM format data in the Hadoop system.

7.2.2 Target Domain: Read Alignment

We have discussed the DNA sequencing pipeline in Section 1.5.2. In this section, we will focus on the discussion of the read aligner.

Next-generation sequencing (NGS) is an area of chemical/biological engineering, bioinformatics and computer science. To sequence the entire genome of a human, a number of copies of the individual's genome are fragmented into small pieces called *reads*, and sequencers then determine the order of nucleotides for all the reads. Nucleotides are the building blocks of a DNA molecular structure that is comprised of only four kinds of nucleotides—generally represented by four letters, A, C, G and T. Therefore, the sequenced reads generated by NGS sequencers, such as Illumina sequencers, are stored as ASCII strings, and a software program, called *read aligner*, assembles the reads into an entire DNA sequence by aligning each read onto a known human genome, called *reference genome*, which is also represented by a string of the four letters.

The key advantage of NGS is that the reads can be sequenced, as well as aligned, completely independently. A genome sequencing task for an individual usually generates billions of reads for sequencing and alignment, creating tremendous amounts of parallelism to explore. The state-of-the-art aligners have exploited the parallelism to some extent; for example, the Burrows-Wheeler Aligner (BWA) [112, 113, 111], by far the most prevalent read alignment software package, is able to fully utilize the maximum number of threads supported by a server to perform read alignment tasks in parallel by using POSIX threads. However, BWA cannot be run across multiple servers, which prevents the scalable parallelism from being further exploited.

For each read, an alignment task is actually an approximate string-matching prob-

lem. Given a pre-defined scoring function, approximate string matching generates the gap/mismatch-tolerant mapping(s) between a read and the reference that maximizes the scoring function. The Smith-Waterman (S-W) algorithm [152] is a commonly used method to address this problem. This dynamic-programming algorithm tolerates an arbitrary number of mismatches and gaps, and is guaranteed to generate all the optimal mapping(s) with the highest score. However, this algorithm has quadratic time complexity, proportionate to the length of the read (m) by the length of the reference (n), i.e., O(mn). A read typically consists of hundreds of nucleotides, while the reference genome contains 3.2 billion. Worse still, as mentioned before, billions of reads need to be processed for only one individual. Assuming that the basic computation of the S-W algorithm consumes 1 ns, it would take over 10,000 years for one thread to align one billion reads along the 3.2-billion-long reference. Therefore, it is practically impossible for read aligners to align reads by only using the S-W algorithm, even though some degree of parallelism can be exploited to accelerate the process.

A two-step heuristic approach is widely adopted by the state-of-the-art aligners, such as Bowtie2 [105] and BWA-MEM [111], as an acceptable alternative for the pure S-W method [113]. In the first step, a read is chopped into small segments, called *seeds*, which are exactly mapped to the reference genome. An algorithm, based on backtracking with the Burrows-Wheeler Transform (BWT) proposed in [112], can achieve linear time complexity for exact string matching, proportionate to the length of a short seed (usually tens of nucleotides) and independent of the length of the long reference. Recently, the state-of-the-art aligners adapted the Burrows-Wheeler Transform (BWT) as an engine to find exact mappings [26]. The state-of-the-art aligners, such as SOAP2 [117], Bowtie [106], and Burrows-Wheeler Aligner (BWA) [112], apply BWT on the reference genome and can significantly improve sequencing

performance with a small memory footprint. Consider the size of a read as m and the size of the reference genome as n, the beauty of the BWT-based algorithm is that it can find the exact match of a read in O(m) time, independent of the size of the genome. Note that the genome size (n) is about 3.2 billions bp, which is much larger than the size of a read (m).

In the second step, the exact mappings serve as seeds and extend both leftward and rightward. The extensions fall into the approximate matching category and are performed using the S-W algorithm. Differing from the pure S-W method in which the whole reference is examined, the seeds generated in the first step locate the S-W tasks in only several candidate sections, i.e., approximately mapping several pairs of strings with hundreds of letters. This canonical paradigm, seed-and-extend, first introduced in the BLAST algorithm [13], is deemed a milestone for read alignment.

7.2.2.1 From BWA, BWA-SW to BWA-MEM

The first version of BWA uses BWT to efficiently perform exact and inexact mappings of short reads [112]. As sequencing technologies improve, the size of a read increases gradually for better sequencing accuracy. In order to align long reads ($>$ 200 bp), the authors of BWA further propose Burrows-Wheeler Aligner's Smith-Waterman Alignment (BWA-SW). BWA-SW follows the seed-and-extend paradigm, such as BLAST [14], BLAT [99], and SSAHA2 [133]. BWA-SW finds alignments by using the Smith-Waterman algorithm allows mismatches and gaps in seeds. The Smith-Waterman algorithm is a classic method for performing local sequence alignment, which is used for string matching or nucleotide sequences or protein sequences. The algorithm was first proposed in [152] and improved in [76] and [15] for better scaling and accuracy, respectively. Smith-Waterman is a dynamic programming algorithm and has O(m^2)

time complexity, where m is the length of a read or a seed. Therefore, it is commonly used for aligning two short sequences due to the quadratic time complexity.

BWA-MEM is the most recent version of BWA. It also follows the seed-and-extend paradigm. Conventionally, the seed-and-extend paradigm only uses a constant size of a seed, e.g., 16 bp, while shifting the seeding window in a constant size, e.g., 10 bp [105]. Instead, BWA-MEM uses an algorithm to find supermaximal exact matches (SMEMs) [110] as seeds of a read in the seed-and-extend process. For seed extension, a dynamic programming algorithm adapted from the Smith Waterman algorithm is used.

7.2.2.2 Input/Output File Formats

Table 7.1 shows the FASTQ format, which is used for describing the raw reads from sequencers [48]. A FASTQ file is one of the inputs of an aligner. The aligner tries to map each seed, collected from a read, to the possible locations on the reference genome. Therefore, a read can have multiple alignments in the final output. The output alignments are stored in the sequence alignment/mapping (SAM) format [155]. To save storage demand, the BGZF compression format [155] is used to compress a SAM file. The compressed format is called BAM [155]. Table 7.2 shows the mandatory fields of an alignment. The detailed SAM format is very complex and also includes the header description. The detailed specification can be found in [155]. Note that both FASTQ and SAM formats have several fields, but not all of the fields need to be used during various types of analyses.

For the whole genome DNA sequencing, the input data sizes in the FASTQ format can reach 250GB to 350GB per individual sample, while the uncompressed SAM

Table 7.1: FASTQ format

Line Number	Brief description
Line 1	sequence identifier and an optional description
Line 2	the raw sequence letters (A, T, C, G, N)
Line 3	(optional) the same sequence identifier and any description
Line 4	the quality values for the sequence in Line 2

format output file after read alignment can achieve 300GB to 500GB. In order to perform read alignment and store data beyond a single node in a scalable way, we choose the open source Apache Hadoop Distributed File System (HDFS) [151] and Parquet [4] for storage purposes. For the computation infrastructure, we use Apache Spark [172, 171]. For the DNA sequencing, we use Avro [2] with Parquet to model both FASTQ and SAM format schemas. More importantly, the computation of each read does not directly depend on the result of the other reads. Therefore, we can use the a cluster to handle these large-scale computation challenges.

7.3 The Baseline Architecture of Cloud-Scale BWAMEM

7.3.1 Overview of CS-BWAMEM

In this section we will introduce the software architecture of our proposed aligner, CS-BWAMEM. This aligner provides two major functions. First, the large FASTQ files stored in the local Linux file system can be automatically partitioned into small file fragments and then uploaded to a distributed file system in a cluster. CS-BWAMEM uses the Hadoop distributed file system (HDFS) [151] as the storage backend. Second, the small FASTQ fragments are fetched from HDFS and aligned in parallel across the

Table 7.2: SAM format (mandatory fields)

Field	Brief Description
QNAME	Query template NAME
FLAG	bitwise FLAG
RNAME	Reference sequence NAME
POS	1-based leftmost mapping POSition
MAPQ	MAPping Quality
CIGAR	CIGAR string
RNEXT	Ref. name of the mate/next read
PNEXT	Position of the mate/next read
TLEN	observed Template LENgth
SEQ	segment SEQuence
QUAL	ASCII of Phred-scaled base QUALity+33

allocated servers in the cluster.

For the read alignment, CS-BWAMEM is built on top of Apache Spark, Parquet, and HDFS. We use Apache Spark to leverage the computation power we have in an in-memory cluster. In CS-BWAMEM, we follow and use the algorithms directly from the state-of-the-art aligner, BWAMEM. In the Spark cluster, the reads are processed in a distributed way to improve the throughput of the aligner. The input and output data of CS-BWAMEM are stored in HDFS using Parquet columnar storage.

For each read, CS-BWAMEM proceeds through two MapReduce stages, which are composed of six steps, as shown in Figure 7.2. In the first map stage, CS-BWAMEM first uses a super maximal exact match (SMEM) method [110] to generate the seeds from the read. The generated seeds are then extended by the Smith-Waterman-like

dynamic programming algorithm. The estimated insert distance of all the reads in this group is calculated in the first reduce stage, where the insert distance is used as a statistical estimate of the distance between the read pairs. In the second map stage, the pair-end alignments are performed, and then the output data are written back to HDFS in the ADAM distributed format [127]. Users can also collect the alignment data in the more widely used SAM format [114].

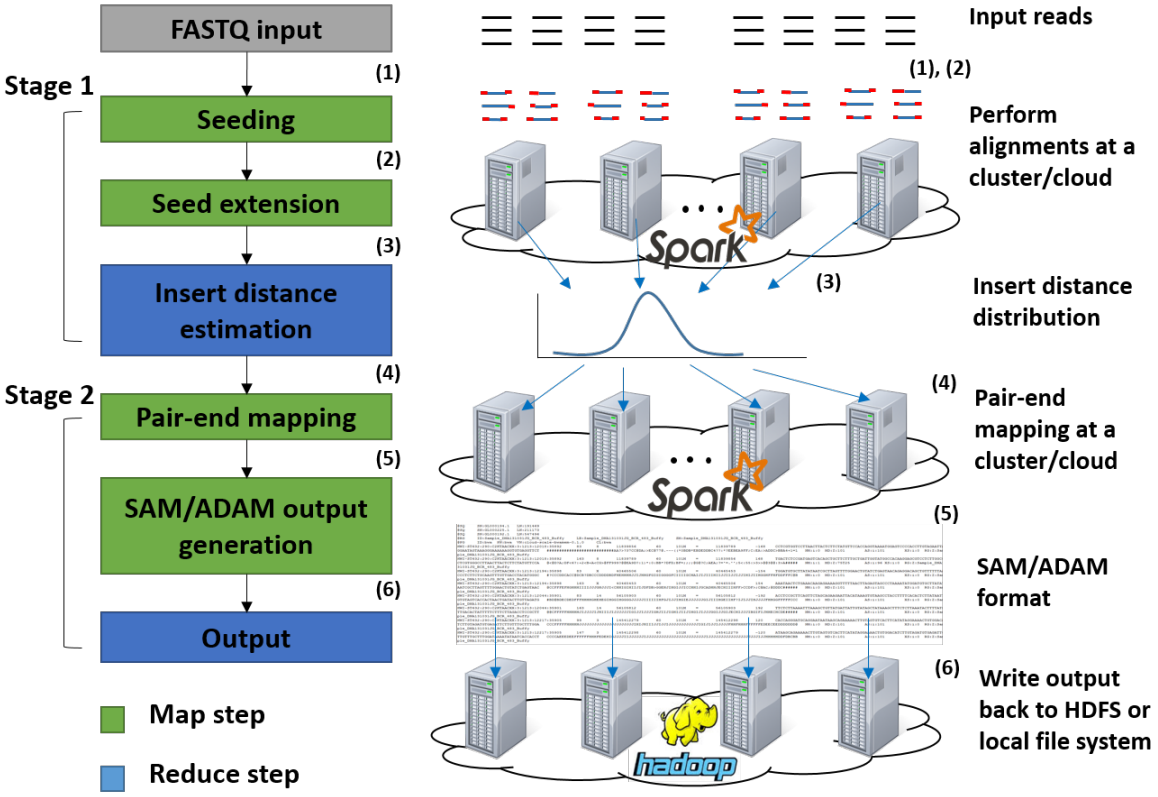


Figure 7.2: Overview of CS-BWAMEM

7.3.2 Data Organization and Computation Patterns in CS-BWAMEM

Figure 7.3 shows the data organization and distribution of CS-BWAMEM. Three types of nodes are in charge of different functions in Spark. A driver is a node where an application is launched and usually where the local data are stored. A master node performs the management and arbitration for Spark workers and HDFS datanodes. A worker node has as two roles: (1) a Spark worker and (2) an HDFS datanode. All worker nodes will interact with the master node for both storage and computation purposes.

CS-BWAMEM works as follows. The reference genome data and short reads generated from sequencers are initially stored in a driver node. Before launching the BWA-MEM algorithm for alignment, the short reads need to be uploaded to HDFS and will be stored in a distributed way. In the very beginning of CS-BWAMEM, the whole reference genome needs to be broadcast from the driver node to every single worker node. The reference genome acts as a large lookup table for performing read alignment for each short read. The reference genome is accessed intensively so each node needs to have its own copy in memory. We will show further optimization in Section 7.4.2. After the broadcast process, the core computation of BWAMEM starts independently on each worker.

Figure 7.4 shows the data flow and computation patterns in CS-BWAMEM. The goal is to align billions of input reads from FASTQ files and generate output SAM/ADAM alignments. From our profiling, the SMEM algorithm for seed generation and three dynamic programming Smith-Waterman kernels (SW, P-SW, and SW') are the four major steps which contribute more than 80% of the total application runtime.

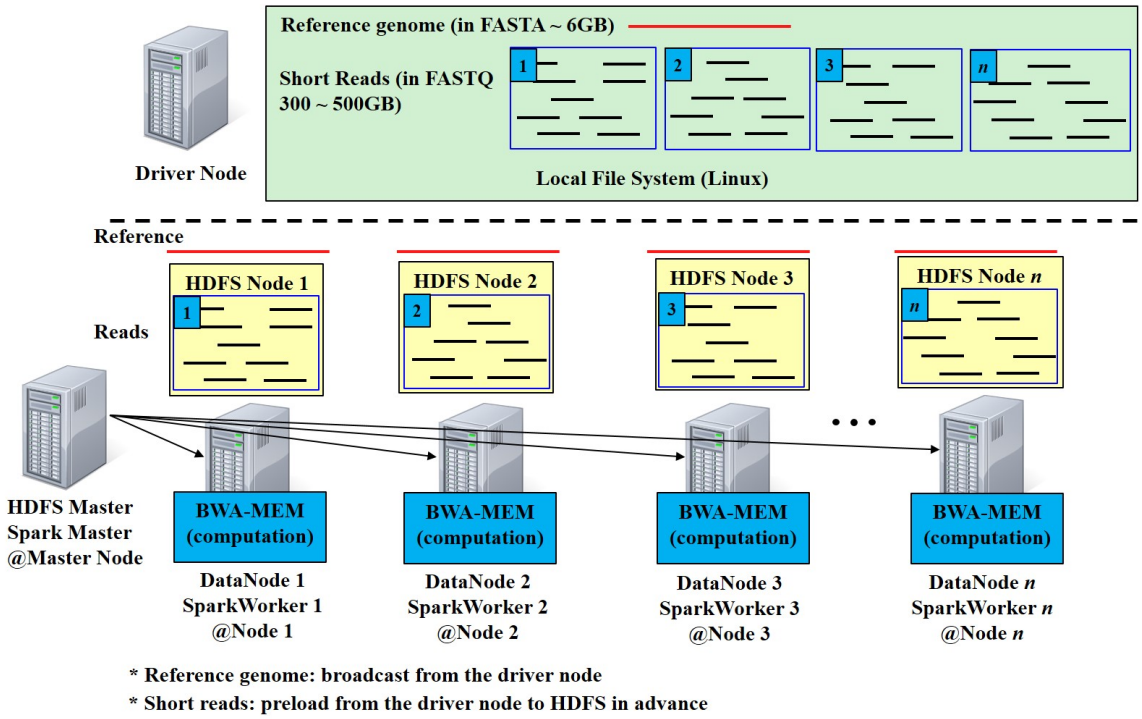


Figure 7.3: Data organization and distribution in CS-BWAMEM

The first "Map" stage in the MapReduce programming model is composed of the SMEM and SW kernels. The seeds, i.e., exact matches, can be generated from input reads by using the SMEM algorithm, as discussed in Section 7.2.2.1. The first Smith-Waterman kernel (SW) then takes the seeds as input and tries to extend the seeds with gaps (insertions and deletions) that are allowed. The mapping positions and the lengths of an extended seed can be known after the first Map "stage." The first "Reduce" stage and acts as a synchronization point and collects the pair-end statistics to calculate insert distance distribution, which is needed for the P-SW kernels. Note that our CS-BWAMEM can do both single-end and pair-end alignments. In the single-end BWAMEM, Step 2 and the P-SW kernel are skipped. In the second "Map" stage, pair-end Smith-Waterman (P-SW) kernels are used to perform pair-end pairing. P-

SW can find more possible missing alignments and filter out incorrect alignments. Finally, the SW' kernel is used to generate the alignments and quality scores and produce final output. The final output can be stored in a distributed fashion in HDFS in an ADAM format or be collected back to the Spark driver node in a SAM format.

We use the in-memory caching capability of Spark. Although CS-BWAMEM does not behave like common machine-learning algorithms, which have iterative behaviors, it is still very important to keep the data in memory since the intermediate data generated from the first MapReduce stage is 20 - 30x larger than the input reads. Without in-memory caching, we would expect to incur a huge disk I/O penalty by writing the intermediate data (6TB - 15TB) out to HDFS or a local Linux file system for each whole-genome sample.

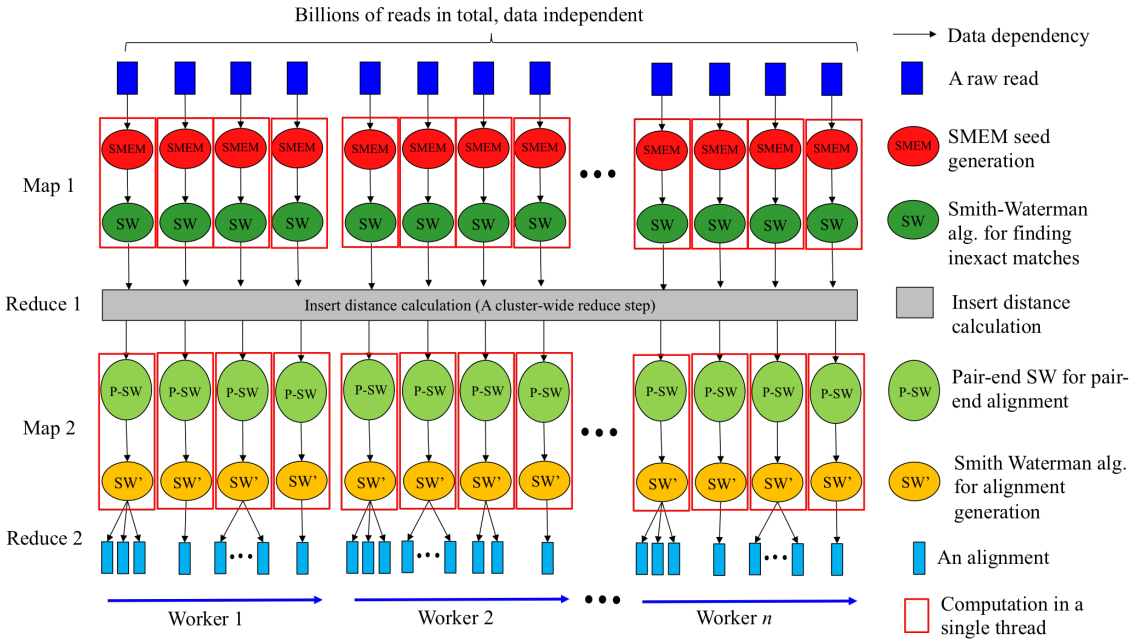


Figure 7.4: CS-BWAMEM: data flow and computation patterns

7.4 Optimizations in the Datacenter Level: a Case Study in CS-BWAMEM

In this section we introduce several customized strategies for the applications of the DNA sequencing pipeline deployed on an in-memory cluster. The proposed strategies can be applied to applications, including, but not limited to, DNA sequencing. We use CS-BWAMEM as a case study to demonstrate the effectiveness of these strategies.

7.4.1 Data Organization: Column-Oriented Storages

The first strategy we adopted is the column-oriented storage, which is introduced in Section 7.2.1.2. We use the Parquet framework for the column-oriented storage. CS-BWAMEM can enjoy two benefits from column-oriented storage: (1) *in situ* data access and (2) better compression and less storage. First, the *in situ* data access is required for data processing since the FASTQ data is distributedly stored in HDFS. Second, the FASTQ data is highly structured and can be compressed well. For example, the *seq* field only contains five different characters: A, T, C, G, N.

We do not use the benefit of on-demand data fetching for CS-BWAMEM. In CS-BWAMEM, the BWA, SW, and P-SW kernels described in Section 7.3 only require the *seq* field, while the SW' kernel requires all fields. For the other applications in the DNA sequencing pipeline described in Section 1.5.2, SAM or ADAM formats are widely used. Like the FASTQ format, the SAM format is also highly structured and shares the same benefits—like better compression and less storage. The effectiveness of on-demand data fetching highly depends on the behavior of an application. Also, Berkeley's ADAM has an implementation for *sort*, *markduplicate*, *baserecalibration*,

and *indelrealignment* [127].

7.4.2 Broadcast Elimination: Loading The Large Reference from Local Worker Nodes

It is common to have shared variables in a multi-threaded program in a single-node. In the distributed environment, broadcast can be used to provide share variables or tables for every node in a cluster. Normally, the shared variables are initially loaded from the driver node and then are broadcast to all worker nodes through networking. The efficiency of broadcast can influence the performance and scalability of a cluster system. As the number of nodes and the data size to be broadcast increase, the efficiency of broadcast is bounded by network bandwidth.

CS-BWAMEM give us a strong motivation to improve the efficiency of broadcast. The reference genome is required to be physically located in each node for fast and frequent accesses during the alignment process. Figure 7.5(a) shows the initial mechanism provided in Spark to broadcast the reference genome across all worker nodes. CS-BWAMEM is initially launched from the driver node and loads the 6 GB reference genome from the local Linux file system. The reference genome is then broadcast to each worker node through the network and becomes a shared variable in the distributed environment. Spark provides two types of broadcasting mechanisms: (1) HTTP broadcast [172] and (2) Torrent broadcast [44]. The default HTTP broadcast creates HTTP service for variable transfer, which significantly reduces performance. The Torrent broadcast use a BitTorrent session [5] to distribute shared variables from the seed to multiple receivers. However, the network bandwidth is still the bottleneck for the efficiency of broadcast.

In this dissertation we propose a novel method to load the shared variables directly from each worker node. Figure 7.5(b) demonstrates the mechanism by which each worker node can fetch the reference genome from its local file system. In this case, the limitation of aggregate network bandwidth and computation power for data transfer are transformed to the disk I/O bandwidth on each node, which improves performance by close to 13.5% in our cluster.

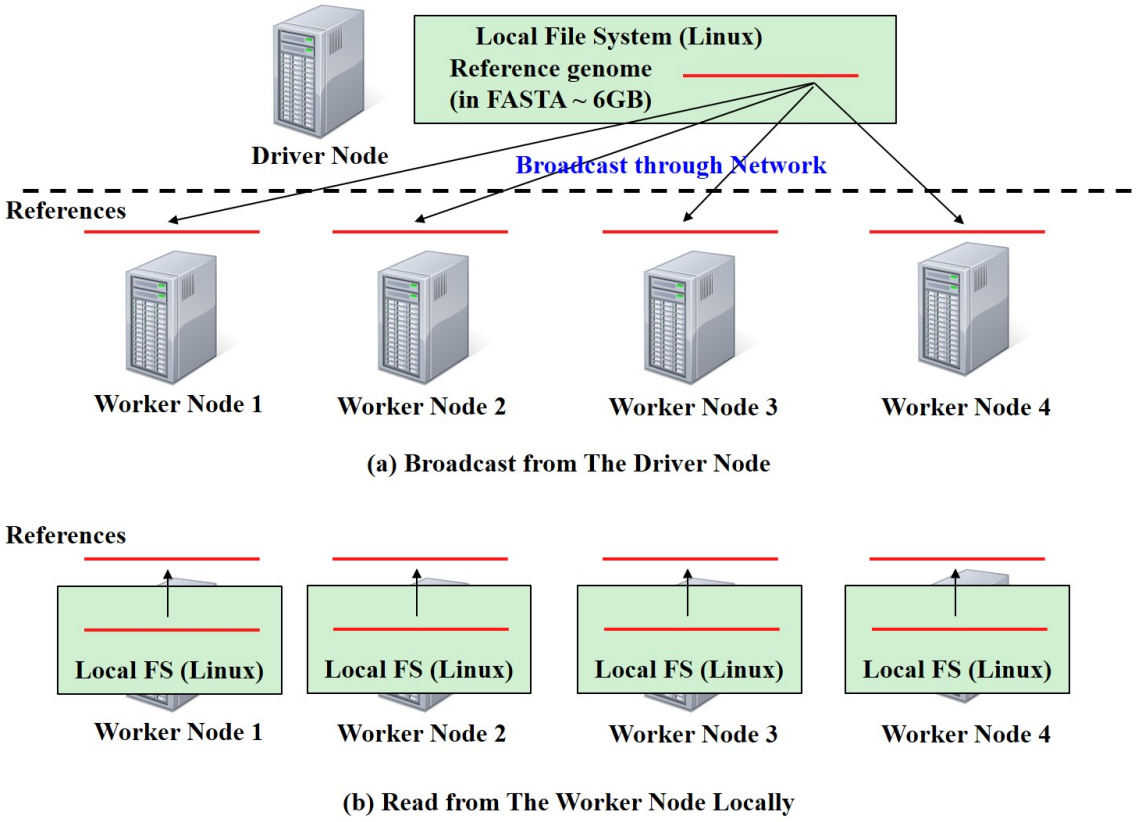


Figure 7.5: Broadcast vs. loading from the local file system

This strategy works great for large and infrequently changed data like the human reference genome used in CS-BWAMEM. The human reference genome is only updated every one or two year(s). The one-time uploading effort can be negligible

since we rarely update the data. The only limitation of the mechanism is that it may not be useful in a highly virtualized cluster. In a virtualized cluster, we do not have prior knowledge of where the virtualized machines physically locate. However, for a private cluster designed to handle a specific application, such as CS-BWAMEM, this strategy is very useful. We believe the strategy can be applied in other applications with similar settings.

7.4.3 Offloading Compute-Intensive Kernels: Reducing Data Transfer Overhead through Batch Processing

In a cluster system, computations are generally done in the general-purpose processors. As discussed in Chapter 5, accelerators are attractive alternatives to improve performance and energy efficiency. The built-in vector engines, like Intel SSE and Intel AVX, are the accelerators that can be directly used in today's modern processors. The co-processors like GPUs, Intel Xeon Phi co-processors, and FPGAs are also common accelerators. By designing suitable algorithm and customized accelerators, we can map compute-intensive kernels into such accelerators.

For coprocessor-based accelerators, such as GPUs and FPGA accelerators, the communication overhead to transfer data from the CPU host program to the coprocessors can dominate the total computation time. Normally, we need to send data from the CPU host program to the device memory on coprocessors. Modern coprocessors usually use the PCI Express technology for data transfer. If the data to be computed is not large enough, the communication overhead can dominate.

Even for the built-in vector engine, which is very close to the processor, the data transfer overhead can occur. This is because a lot of open-source cloud computing

infrastructures, such as Hadoop, Spark, and Parquet, are all developed in Java. The whole software ecosystem is developed using Java as well. Java applications run on top of the Java Virtual Machine (JVM), which does not support these architecture-dependent features due to the cross-platform concern. Therefore, we cannot directly utilize the accelerators that are built in the process, such as Intel AVX. We need to use Java Native Interface (JNI) to transfer data to the native implementation that maps the computation kernel into Intel SSE or AVX. The data transfer between JVM and a native environment also leads to performance degradation. We call this process data serialization and deserialization.

7.4.3.1 Pair-End Smith-Waterman Kernel Acceleration: Using Vector Engines

We implement CS-BWAMEM on top of Spark and use Scala as the programming language. Scala is an object-oriented functional programming language and can be compiled into Java bytecodes and run on JVM. Spark has very good support for Scala and is also implemented using Scala; thus, our decision was to use Scala. In the original BWA-MEM aligner, the developer uses Intel SSE vector engines to accelerate the compute-intensive Smith-Waterman algorithm that is used for pair-end mapping.

For the pair-end Smith-Waterman kernel (P-SW), as shown in Figure 7.4, the striped Smith-Waterman SIMD algorithms are developed to utilize the vector engines for acceleration [68, 146]. However, these SIMD algorithms cannot be directly implemented in Java due to its platform-independent characteristics. To leverage the power of vector engines, like Intel SSE, one feasible way is to use the Java Native Interface (JNI) support. JNI allows programmers to call native binaries wrapped in a shared library to achieve native performance. With JNI support, the striped Smith-

Waterman SIMD algorithm can be implemented in C and compiled into a shared library. Through JNI, the input data of the SIMD engines can be transferred from JVM to native machine.

However, if we use JNI in a fine-grained way with many tiny computations, the overhead of JNI to send data back and forth can dominate the overall runtime. Therefore, we take the batch processing strategy to reduce the communication overhead between Java and native SIMD functions. It is important to choose the batch size carefully because a large batch can introduce huge intermediate data structures to be allocated in main memory. For example, if the batch size is set to 100, we may need to increase the memory use by 100x for storing these data structures during the batch code section. This can lead to significant performance degradation when we run out of memory. Garbage collection can dominate the whole program runtime in this case. The batch size needs to be carefully set based on the input data size and the physical memory we have in the Spark worker node. In CS-BWAMEM, we set the batch size to be 10 or 100 for using the Intel SSE. In Chapter 8, we will have an in-depth discussion and analysis for applying batch processing for PCI-e based FPGA accelerators.

7.4.4 Improving I/O Performance: RDD Caching and Latency Hiding

Due to the large amount of data in whole-genome sequencing, a significant performance degradation can occur because of the I/O overhead. In this section we provide three strategies that significantly hide the potential I/O overhead and improve the performance of CS-BWAMEM.

The first proposed strategy is to cache the intermediate data structures, generated

from the first MapReduce stage, in the in-memory cluster by using RDD, as illustrated in Figure 7.6. CS-BWAMEM is not like the machine learning algorithms, which usually have iterative behaviors. However, the intermediate data structures generated after seed extension is about 5x to 10x larger than the original input FASTQ files. This is because after the seed extension step, more information, such as the coordinate of each alignment, is included. The intermediate data structures are more like the SAM format, which is more complex than the FASTQ format, and thus demands more memory. Second, in the seeding stage, a read can generate many seeds, where each seed may have its own alignment. Therefore, the number of alignments is larger than the original numbers of seed. Third, Java data structures are more memory-hungry than the ones in C/C++. Based on the above reasons, if we do not apply an in-memory caching strategy, like Spark RDD, these intermediate data need to be written back to local disks or even HDFS (if we use Hadoop MapReduce). Significant I/O overhead can result.

Before the alignment stage, the pair-end FASTQ input files stored in the local Linux file system need to be uploaded to HDFS for later pipelining. Before the data can be uploaded to HDFS, we need to first read it from local file system. However, the local file read sequentially followed by the HDFS upload is very time-consuming. However, since the input FASTQ files can be very large, about 300GB - 500GB per genome, we have to split the whole upload process into multiple groups, as shown in Figure 7.7.

Therefore, our second proposed strategy is to pipeline the sequential read of the FASTQ files and the write (upload) to HDFS between different groups of reads, as shown in Figure 7.7. This is because the local file system read and HDFS write operate in different I/O devices. Therefore, we can utilize the I/O bandwidth in both

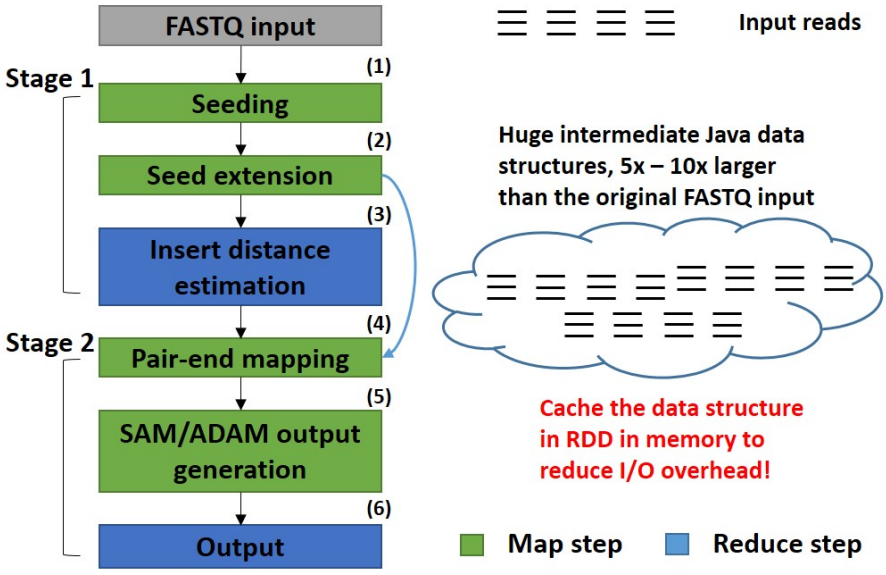


Figure 7.6: Strategy 1: caching intermediate data structures between MapReduce stages using Spark RDDs

local file system and HDFS.

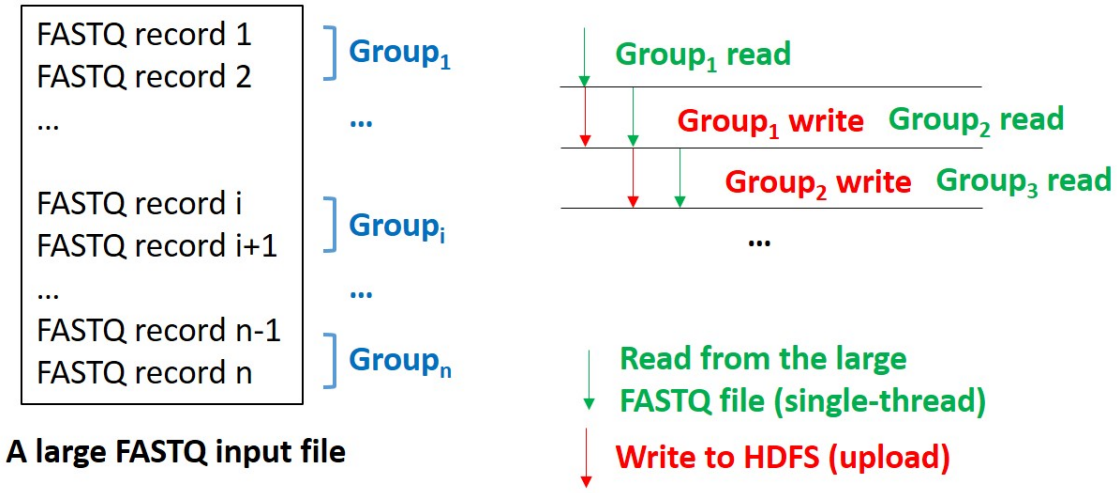


Figure 7.7: Strategy 2: pipelining the I/O of (1) read from a large SAM input file and (2) write to HDFS in the CS-BWAMEM upload

Similarly, in the alignment stage we can also leverage the same pipeline strategy. The alignment MapReduce stage can be overlapped with the write to the local Linux file system, as demonstrated in Figure 7.8. This happens when users want to collect the SAM format file and use the traditional GATK pipeline. Here, we pipeline the computation with file I/O to improve performance.

Note that it may be possible to overlap the alignment stage with the HDFS write stage if we want to generate output in a distributed fashion in ADAM format. However, it will require an approximate 2x computation resources since we are overlapping two MapReduce stages, the alignment and the HDFS output stage together, as demonstrated in Figure 7.7. We do not consider this in the current implementation.

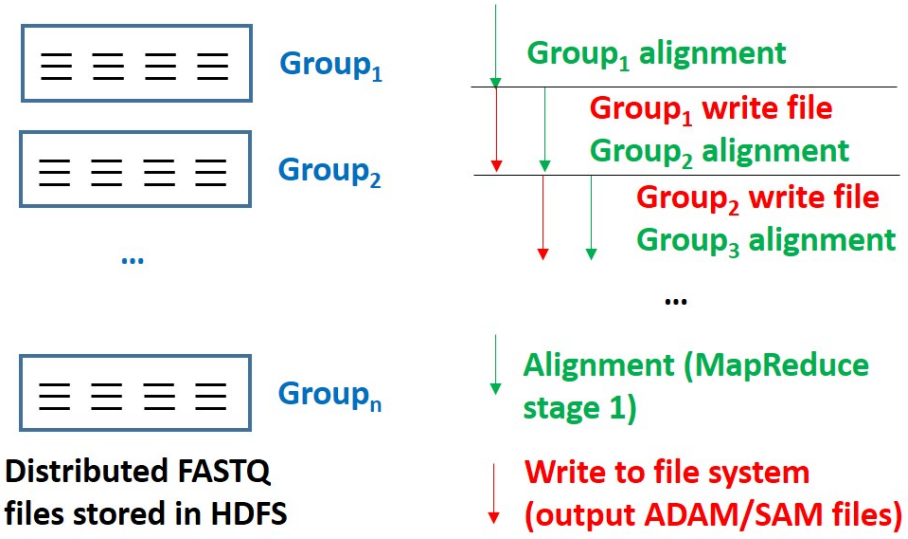


Figure 7.8: Strategy 3: pipelining (1) the MapReduce computation with (2) write output to HDFS in the CS-BWAMEM alignment

7.5 Experimental Results

7.5.1 CS-BWAMEM vs. State-of-the-Art Aligners

We compare our CS-BWAMEM with the two state-of-the-art aligners, BWA-MEM [111] and Bowtie 2 [105], which are used in variant discovery pipelines (DNASeq) [129] and RNASeq [158], respectively. Table 7.3 shows the environment setup of CS-BWAMEM, BWA-MEM, and Bowtie2, for the pair-end read alignment for a 30x coverage of whole-genome data. Our CS-BWAMEM is implemented in Scala on top of Spark and runs on JVM. BWAMEM and Bowtie2 are implemented in C [111] and run natively. Note that the results of CS-BWAMEM are based on the discussion in Section 7.3 and the column stores described in Section 7.2.1.2. With 30 nodes, CS-BWAMEM can finish whole-genome alignment within 32 minutes, which leads to 18x and 20.3x speedups over state-of-the-art BWAMEM and Bowtie 2, respectively. The major reason that CS-BWAMEM cannot achieve the ideal 30x speedup with 30 nodes is because the performance of JVM execution still falls behind the native execution by around 2x. In terms of the overall throughput, CS-BWAMEM does not outperform BWAMEM or Bowtie 2, with the same computing resources. However, CS-BWAMEM can be used to target the latency-critical applications, especially when users have enough computing resources. It is much easier to add more nodes in a cluster rather than add more processors in a node to reduce latency. Another advantage of using CS-BWAMEM is because users can leverage the benefits brought from cluster computing. For example, in the next step, "Sort," the scale-out version of software also demonstrates significant performance benefits. We will include the results of CS-BWAMEM alignment and Sort in the next section.

Table 7.3: CS-BWAMEM vs. BWA-MEM vs. Bowtie2 (pair-end)

	BWAMEM 0.7.12, Bowtie2 2.2.6	CS-BWAMEM
FASTQ size	300GB	300GB
Implementation	C, native execution	Java, JVM execution
Configuration	One node, 24 threads	30 nodes, 720 threads

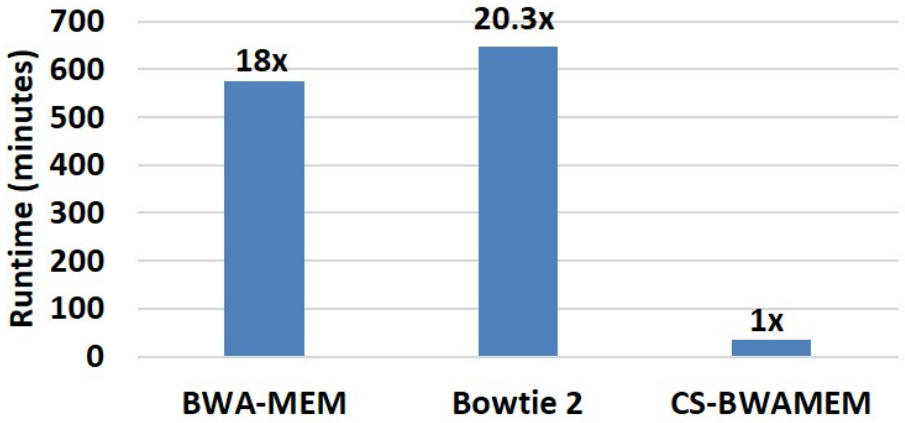


Figure 7.9: Performance comparison of (1) BWA-MEM, (2) Bowtie 2, and (3) CS-BWAMEM, for 30x coverage of whole-genome data.

7.5.2 Scalability and Runtime Distribution of CS-BWAMEM

In this section we provide the scalability analysis of CS-BWAMEM and then provide the runtime of CS-BWAMEM for both whole-genome data and whole-exome data. Note that the whole-genome data are commonly used for research purposes, while the whole-exome data are used in more clinical applications.

Figure 7.10 shows the scalability of CS-BWAMEM when we use different cluster sizes. CS-BWAMEM can provide scalable performance when we have more computing resource in a cluster. By using 15 nodes, CS-BWAMEM can provide 2.6x speedup

over a five-node cluster. When we use the largest available cluster size, 30 nodes, we still can achieve 4.88x speedup over five nodes. With good scalability demonstrated in Figure 7.10, users can determine a suitable cluster size based on their performance target.

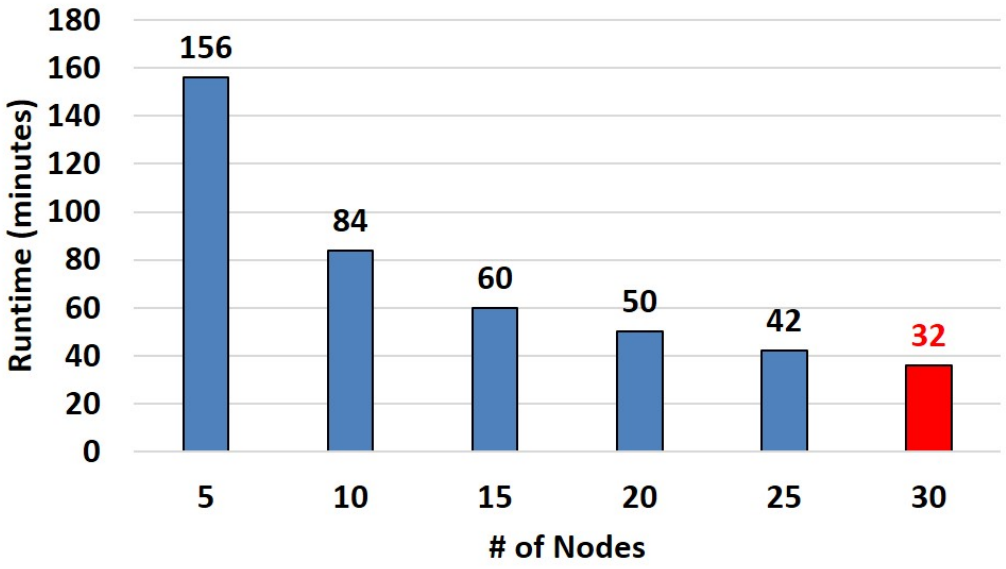


Figure 7.10: Scalability of CS-BWAMEM over different cluster sizes

Figure 7.11 shows the runtime distribution of 13 whole-genome data with 30x coverage. In addition to CS-BWAMEM alignment, we also provide the runtime distribution in the next step, Sort. For alignment, we can observe that the runtime is stable among all the whole-genome data, ranging from 36 to 49 minutes. For "Sort," we are also able to finish this step within 9.6 to 27 minutes. The performance is stable and does not vary a lot among different data samples.

We mentioned that recent clinical applications almost always use whole-exome data since the cost is cheaper. Also, the amount of data is only about 1/10 of the whole-genome data, and thus both the storage and computation demands are less.

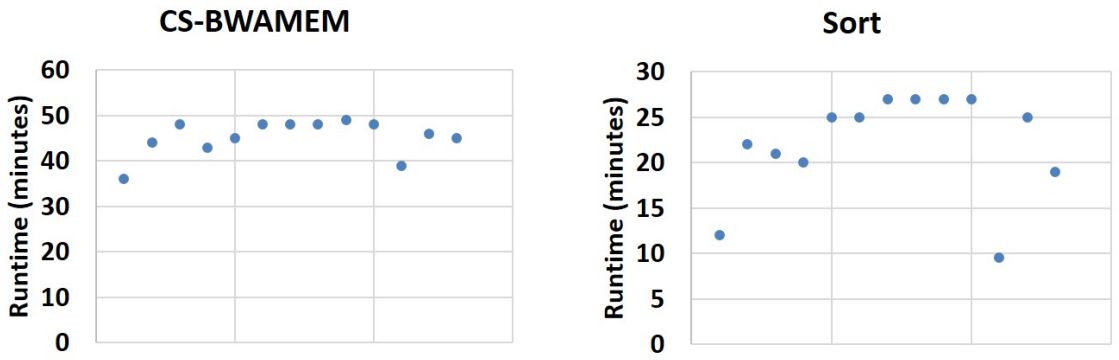


Figure 7.11: The runtime distribution of 13 30x coverage whole-genome data on (1) CS-BWAMEM and (2) Sort

Our collaborators at UCLA also collected around 800 whole-exome cases to detect rare Mendelian disorders [108]. The results shows that our CS-BWAMEM and the proposed flow of "Sort" in the cluster scale can efficiently finish the alignment and sort steps in around 10 minutes!

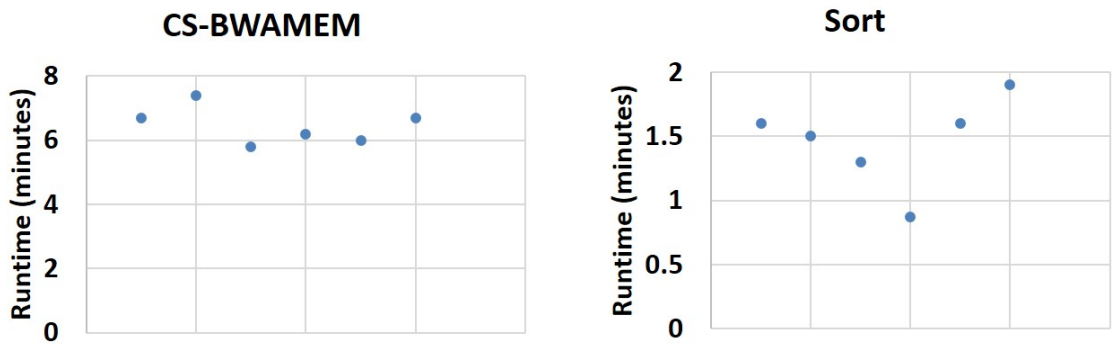


Figure 7.12: The runtime distribution of six whole-exome data on (1) CS-BWAMEM and (2) Sort

We project that if we can also complete the steps of *markduplicate*, *indelRealignment*, and *baseRecalibration*, we can achieve the whole data pre-processing pipeline,

as demonstrated in Figure 1.3, within eight hours and one hour, for whole-genome data and whole-exome data, respectively.

7.5.3 Validation of CS-BWAMEM's Alignment Quality

In general, our CS-BWAMEM algorithm basically follows the original BWA-MEM algorithm [111]. However, the final alignment results are still slightly different when compared to BWA-MEM. Fortunately, the difference is almost negligible. In this section we provide a quantitative analysis to validate the alignment quality of CS-BWAMEM.

These are the two implementation issues that make the alignment results different. First, in the insert distance calculation step for pair-end alignment, as shown in Figure 7.2, we need to calculate the insert distance distribution of a group of reads. In BWA-MEM, the group size depends on the number of CPU threads to be used. In CS-BWAMEM, it depends on the HDFS FASTQ file partition size and the number of partitions to be used, which generally contains more reads in a group than that of BWAMEM. Therefore, the insert distance distribution can be different and it slightly influences the pair-end alignment. Second, in the seed extension stage, the original BWA-MEM uses its own library for sorting seeds, while we directly use the built-in Java library for sorting in our implementation. It causes slightly different alignment results since the algorithm itself relies on the order of the seeds.

We used simulated reads generated from a read simulator, ART [91]. The reason to use simulated reads instead of real reads is because the simulated reads can provide the ground truth of alignment. We generate about 16 million Illumina reads at 100bp and 150bp for validating our CS-BWAMEM. We follow a similar validation method

described in BWA-SW [113].

Table 7.4 and Table 7.5 show the alignment quality results on 100bp and 150bp reads, respectively. MAPQ stands for the mapping quality score, which is introduced by Heng Li. MAPQ is used to estimate the probability of a read being placed at a wrong position. As suggested in [113], the author used score 20 as the boundary to differentiate if the aligner is confident of the alignment result. We follow this rule and evaluate our alignment result together with BWA-MEM. We find that CS-BWAMEM can provide the very similar alignment quality as BWA-MEM does for both 100bp and 150bp reads.

First, CS-BWAMEM can generate similar numbers of confidently aligned reads (MAPQ $\geq$ 20). As [113] suggested, we compared the coordinates of the simulated reads with our aligned reads to find out the error alignments. Note that the errors are the wrong alignments out of the alignments of high confidence reads (MAPQ $\geq$ 20). As shown in Table 7.4 and Table 7.5, the results show that the error rates are low and similar in both aligners. The alignment differences at Column 4 in both tables mainly come from the reads that can be mapped to multiple possible locations on the reference genome. In this case, the aligner cannot tell where should these reads are mapped, and thus the MAPQ of this kind of reads will be zero.

Table 7.4: Validation of alignment data of CS-BWAMEM and BWA-MEM on 16M 100bp simulated reads

100bp	MAPQ $\geq$ 20(%)	Error Aln.(%) (MAPQ $\geq$ 20)	Aln. diff.(%)
CS-BWAMEM	64.56	0.0134	18.07
BWA-MEM	64.54	0.0137	19.20

Table 7.5: Validation of alignment data of CS-BWAMEM and BWA-MEM on 16M 150bp simulated reads

150bp	MAPQ $\geq$ 20(%)	Error Aln.(%) (MAPQ $\geq$ 20)	Aln. diff.(%)
CS-BWAMEM	65.05	1.93	20.48
BWA-MEM	65.04	1.93	21.50

7.5.4 Effectiveness of Avoiding Broadcast

We compare the performance of our broadcast avoidance method with the state-of-the-art Torrent broadcast used in Spark. Table 7.6 demonstrates the effectiveness of avoiding broadcast. For a 30-node cluster, our method can effectively improve the performance by 13.51%. We also find that broadcast performance overhead grows as the cluster size increases. It is an important strategy to improve system-wide performance, especially when the cluster size is large.

Note that since Spark 1.x, the Torrent broadcast mechanism has replaced the original HTTP broadcast one. The performance has been significantly improved, but is still not good when the size of a cluster is large.

Table 7.6: Spark torrent broadcast V.S. load from local file system

	CS-BWAMEM + broadcast	CS-BWAMEM + load locally
Broadcast Mechanism	Torrent broadcast	Load from local file system
Configuration	30 nodes, 720 threads	30 nodes, 720 threads
Runtime	37 minutes	32 minutes

7.5.5 Effectiveness of SIMD Acceleration and Batch Processing in Pair-End S-W Algorithm

As discussed in Section 7.4.3.1, we use the Intel SSE SIMD engine to accelerate the pair-end S-W algorithm. Note that we only apply SIMD acceleration to the pair-end S-W algorithm. For the seed extension step shown in Figure 7.2, a modified version of the S-W algorithm, which cannot be accelerated using a SIMD S-W algorithm like [68], is used in the original BWA-MEM. In order to match the results of BWA-MEM, CS-BWAMEM keeps the original algorithm of seed extension. Instead of using the SIMD S-W algorithm, we utilize customized FPGA acceleration, which will be discussed in Chapter 8, to accelerate the seed extension process.

Without SIMD acceleration, the pair-end S-W algorithm would become the bottleneck of the whole application, as demonstrated in Table 7.7, which can slow down CS-BWAMEM by more than 2x. With pair-end SIMD acceleration with batch processing, we can achieve the 32 minutes of runtime, improving the performance by 2.19x.

Table 7.7: Batch processing: Java native execution vs. Intel SSE acceleration

	CS-BWAMEM (Java)	CS-BWAMEM (Java + Intel SSE)
Java vs. SIMD Acc.	JVM execution	Intel SSE Acceleration
Configuration	30 nodes, 720 threads	30 nodes, 720 threads
Runtime	70 minutes	32 minutes

Figure 7.13 shows the importance of carefully selecting the batch size. As discussed in Section 7.4.3.1, a large batch can reduce the communication overhead. However, a large batch size can increase memory usage and reduce JVM garbage collection performance. Therefore, a careful design space exploration needs to be performed to

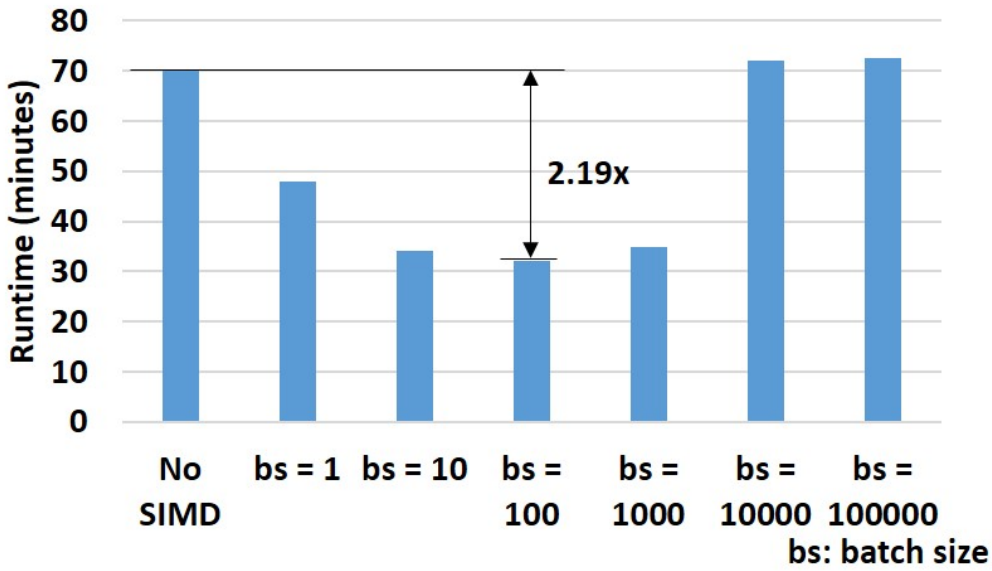


Figure 7.13: The comparison of the overall CS-BWAMEM runtime over different batch sizes

select the best batch size. We find that the best performance occurs when batch size is equal to 100, which is 33.3% faster than the case of not utilizing batch processing (bs = 1). We can observe that when the batch sizes are too large, such as 10,000 or 100,000, the performance degrades as well due to worse memory behavior in JVM. For example, when the batch size is equal to 10,000, performance is reduced by 50% compared to the case of when no batch processing is applied (bs = 1).

Figure 7.14 shows the performance comparison for the second MapReduce stage, i.e., the stage of paring and output generation. With SIMD acceleration, this stage can be accelerated by 3.88x. The performance of this stage also improves by 2x with a careful selection of batch size (bs = 100).

Table 7.8 demonstrates the runtime and garbage collection (GC) time distribution

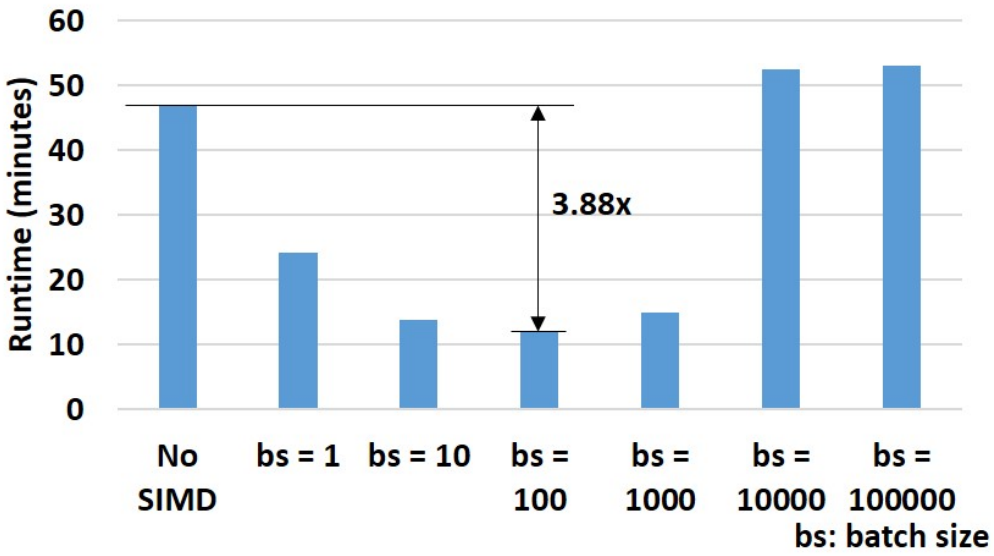


Figure 7.14: The comparison of the second MapReduce stage (the pairing and output generation stage) runtime of CS-BWAMEM over different batch sizes

of a sample of 4,000 HDFS file partitions. We compare the two setting: (1) bs = 100 and (2) bs = 10,000. We can determine that the setting of "bs = 100" is 4x and 5x faster than the setting of "bs = 10,000" for the mean runtime and mean GC time, respectively.

7.5.6 Effectiveness of I/O Optimizations

In Section 7.4.4, we discussed several I/O optimization strategies, including RDD caching and I/O pipelining. Based on the experimental results, the proposed Strategy 2, as shown in Figure 7.7, can improve the performance of the uploading stage by 40% (2 hours to 1.2 hours) for 30x coverage of whole-genome data.

The proposed Strategy 3 (Figure 7.8), which overlaps the alignment stage with

Table 7.8: Runtime and GC time distribution of the second MapReduce stage of a sample of 4,000 HDFS file partitions.

	Min	25th percentile	Median	75th percentile	Max
Runtime, bs=100	5s	9s	9s	10s	15s
Runtime, bs=10,000	10s	31s	36s	42s	1.4min
GC Time, bs=100	0.1s	2s	3s	4s	17s
GC Time, bs=10,000	39ms	0.5s	0.6s	0.8s	2s

file system I/O, can reduce the runtime from 80 minutes to 52 minutes (35%). Based on our observation, the local file system I/O bandwidth is about 120 MB/s in our system. The file size of the whole-genome alignment is about 360GB. Even if we can write the output file at full speed consistently, it still takes 50 minutes to finish it at 120MB/s bandwidth. Therefore, our results demonstrate that we have already pipelined the I/O with computation efficiently.

Figure 7.15 shows the alignment time of generating SAM format outputs over different cluster sizes. In this setting, the final alignment results are collected at the driver node in an aggregated SAM file. When we use a large cluster, e.g. a cluster with 25 - 30 nodes, we observe the diminishing return of performance gain. This is because the available network bandwidth and disk bandwidth on the single driver node limit the scalability.

With the three proposed strategies, we can significantly improve the system-wide performance. We can conclude that I/O optimization is the key to reducing the overall runtime.

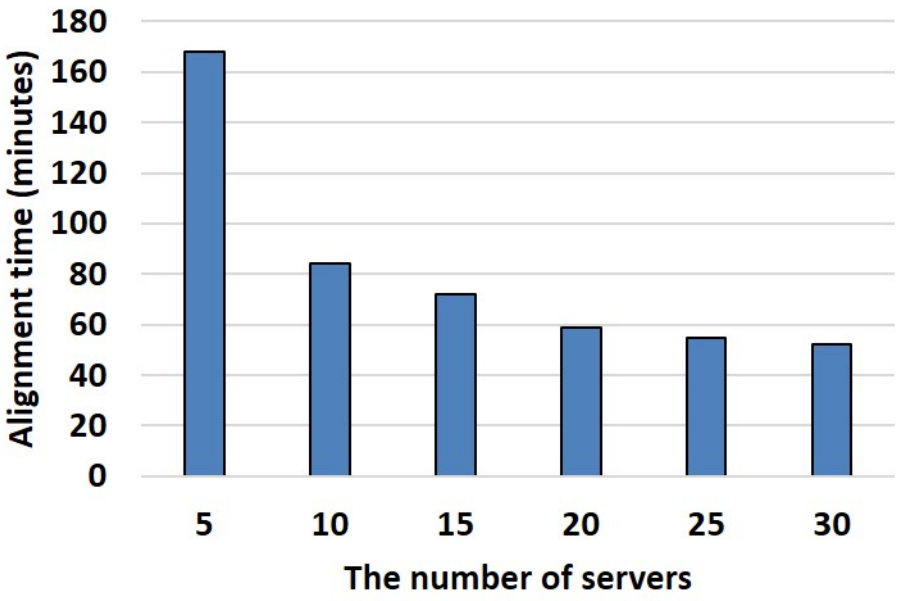


Figure 7.15: Scalability of CS-BWAMEM in SAM format outputs over different cluster sizes

7.5.7 Current Progress of DNA Sequencing Pipeline Acceleration

In this chapter we proposed CS-BWAMEM to accelerate the read alignment through the scale-out in-memory cluster approach. Our final goal is to accelerate the whole DNA sequencing pipeline, as shown in Figure 7.16. Our current progress shows that we can reduce the first three steps in the pipeline (alignment, sort, markduplicate) by 40x when we use a 30-node cluster, which is a super-linear speedup. This is through efficient in-memory cluster computing, algorithm improvement in markduplicate, and efficient cluster-level in-memory sorting in Spark. We clearly demonstrate the scale-out approach can help us process the DNA sequencing data more efficiently.

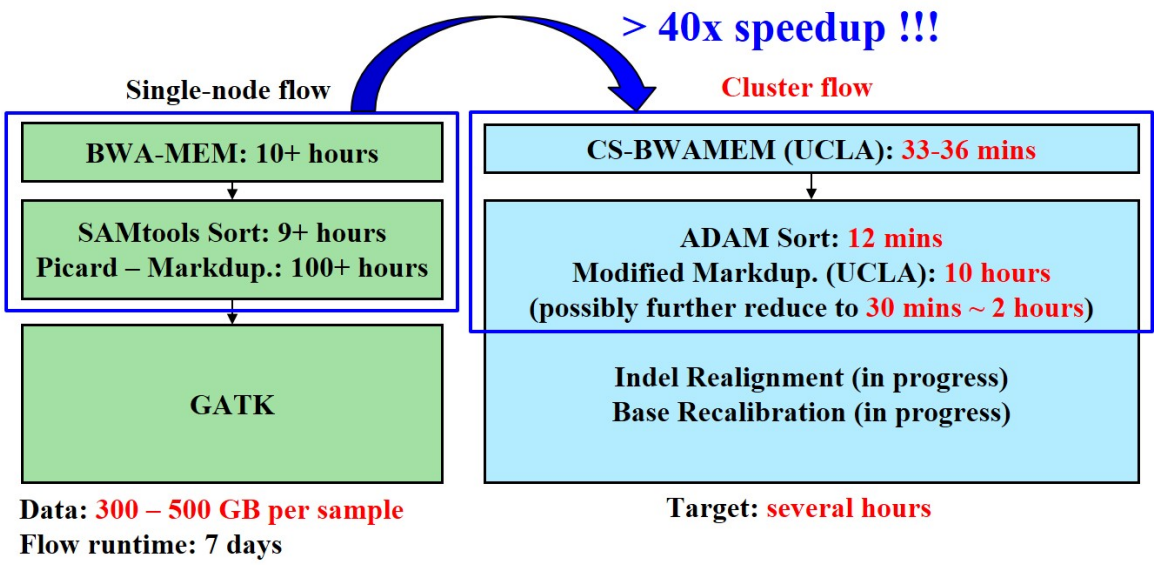


Figure 7.16: Current progress of accelerating the DNA sequencing pipeline

7.6 Conclusions

In this chapter we first introduce the software suite, cloud-scale BWAMEM (CS-BWAMEM), where CS-BWAMEM is an ultrafast aligner that provides around 20x speedup over the state-of-the-art aligner. CS-BWAMEM uses the in-memory cluster computing system, Spark, to provide scalable performance that matches a user's performance target. CS-BWAMEM is also a valuable case study to demonstrate the effectiveness of using an in-memory cluster computing system to accelerate the DNA sequencing pipeline.

Next, we introduce the details of customized strategies for improving the performance of CS-BWAMEM. Some of them can be further applied to other applications in the DNA sequencing pipeline or the other domains. First, we avoid broadcasting the large human reference genome data to every node before computation. Instead,

we load it from the local disk of each Spark worker during runtime. Second, the data is stored in a column store for better compression and partial data access. Third, to utilize accelerators or coprocessors efficiently, we group small tasks into a batch; the accelerators then can compute tasks in a batched way to avoid the overhead introduced by fine-grained data transfers. We used the Intel SSE vector engine as a case study in this chapter. Finally, we try to optimize I/O efficiency by using RDD caching and pipelining file system I/O with MapReduce computation. By using all the customized strategies in a 30-node cluster, CS-BWAMEM can align the whole-genome in 32 minutes—which is about 18x faster than the state-of-the-art aligner, BWA-MEM. The CS-BWAMEM is available online at https://github.com/ytchen0323/cloud-scale-bwamem.

I am the main architect of CS-BWAMEM and contributed 80% of the implementation. CS-BWAMEM is a collaborative effort between Sen Li, Peng Wei, Peipei Zhou and myself. Peng Wei and I have had many discussions on the mechanism of avoiding broadcast and CS-BWAMEM design. The improvement of markduplicate, described in Section 7.5.7, is a collaborative effort between Qi Zhao and myself.

CHAPTER 8

FPGA Acceleration in an In-Memory Cluster

8.1 Introduction

In Chapter 7 we discussed about our motivation to design CS-BWAMEM. We also show the overall software architecture in CS-BWAMEM. In this chapter we will focus more on how to use FPGA-based accelerators for further scaling up the performance of each node in a cluster for CS-BWAMEM. As discussed in Section 7.2.2, recent state-of-the-art aligners decompose the read alignment process into two steps [111][105]. The classic Smith-Waterman (S-W) algorithm is used to find inexact mappings by using dynamic programming. The time complexity is quadratic, and it thus becomes one of the main computation bottlenecks in the state-of-the-art aligners. Therefore, we decide to leverage this compute-intensive kernel into hardware accelerators.

The acceleration techniques have been studied in various architectures, such as SIMD vector engines in general-purpose processors, GPUs, and FPGAs. In [164, 68, 145], the authors redesigned the S-W algorithms based on the architecture of the SIMD vector engines. The wavefront-based [164] and column-based [68, 145] algorithms were proposed to map to the vector engines. Several GPU implementations, such as the work in [125] and CUDASW++ 3.0 [120], demonstrated the speedups over SIMD CPU implementations [68][145]. The research in [138, 123, 174, 170,

100] exploited the anti-diagonal parallelism of the S-W algorithm and implemented wavefront-based architectures on the FPGA. In a very recent work [35], the authors proposed an FPGA-based high-throughput acceleration engine for accelerating the S-W algorithm used in BWA-MEM, which best suits our needs. However, it is unclear how to modify the current S-W algorithm in BWA-MEM to efficiently use the FPGA accelerator. Furthermore, the runtime system and middleware need to be developed to manage the accelerator requests from multiple map tasks under a MapReduce framework.

In this dissertation we apply hardware acceleration to CS-BWAMEM in order to accelerate the compute-intensive Smith-Waterman kernels. Our contributions can be summarized as follows.

- **High-throughput FPGA S-W acceleration engine:** We propose an array-based architecture for processing the enormous number of reads in a high-throughput fashion, adapting better to inputs with widely varied sizes. A two-level hierarchical architecture for resource management is provided. It can reduce the amount of resources needed for synthesizing bus interfaces, while satisfying the off-chip bandwidth demand. Furthermore, our design supports the pruning technique, shortening the runtime of the S-W algorithm significantly. Our FPGA implementation demonstrates a 343.8x speedup over the state-of-the-art Intel Xeon server running under a single thread.
- **Batched S-W algorithm optimized for hardware accelerator:** We realize the high-throughput S-W accelerator proposed in [35] in a PCIe-based FPGA card using the Xilinx SDAccel flow [8] and deploy the accelerators over the cluster. We observe that a significant communication overhead occurs for the data transfer between (1) map tasks and the accelerator manager and between

(2) the accelerator manager and FPGA board. To overcome the communication overheads, we try to group a large number of reads and send them to the FPGA accelerator. However, due to the strong inner-task dependency of the S-W algorithm used in BWA-MEM, we cannot directly send data to FPGA without resolving the dependency. By redesigning the S-W algorithm, we are able to process data in a batched fashion and thus better utilize the FPGA accelerator with much less communication overhead.

- **Accelerator manager:** In the SDAccel flow, we need a software accelerator manager (AM) to manage the accelerator requests from multiple Spark tasks in a node. The AM is in charge of receiving the requests from multiple processes, dispatching the requests, and sending input data to the FPGA accelerator. It is challenging to design the AM efficiently since multiple tasks can issue requests simultaneously. We design the handshaking protocol and use the POSIX shared memory to exchange data between the AM and the Spark tasks.

With hardware acceleration, we can further reduce the runtime by more than 50% of the S-W algorithm process.

8.2 Background

8.2.1 Cluster Computing and Storage Frameworks

In recent years, cluster computing and storage frameworks began to proliferate in datacenters in order to handle large-scale problems. Google File System (GFS) [73] provides a storage layer for developers to store data in a distributed way and use replicas for maintaining reliablilty. We use the Apache Spark [171] in-memory MapReduce

framework as our underlying computation system. Google first proposed MapReduce [62], which provides a simple map/reduce programming model with the underlying system to distribute computations over a cluster with more than thousands of nodes. Apache Spark keeps the MapReduce programming model while further providing the capability to cache reusable data in memory instead of writing data back to the distributed file system at the reduce step. Therefore, Spark demonstrates significant performance improvement over Hadoop [3], an open source implementation of MapReduce. This capability of caching reusable data can be beneficial since we can cache the intermediate data between the steps in the DNA sequencing pipeline.

8.2.2 Hardware Acceleration for Read Alignment

As a fundamental operation in computational biology, accelerating read alignment has captured much attention from academia [16]. The hardware acceleration techniques of both the backtracking algorithm and the S-W algorithm are studied in various kinds of hardware platforms. In [134] the authors proposed a scalable FPGA-based solution, including both the backtracking and the S-W algorithm, and demonstrated a one-order-of-magnitude speedup versus Bowtie. In [17] the authors implemented a fully pipelined accelerator for the backtracking algorithm, achieving a 18.1x speedup over BWA. A systolic-array-based implementation for accelerating the S-W algorithm was described in [174], gaining up to 250x speedup. The authors in [123] proposed a novel computing approach, called race logic, and demonstrated up to 4x faster than the systolic array implementation. These solutions utilize the anti-diagonal parallelism inherent in the S-W algorithm, and work efficiently when the sizes of inputs are relatively homogeneous. In [35] the authors pointed out that the S-W kernel in BWA-MEM is fed with sharply varied size inputs, and is optimized by pruning. A

high-throughput accelerator is implemented specifically for the S-W kernel in BWA-MEM.

These hardware acceleration approaches show great potential to serve as building blocks in state-of-the-art read aligners to improve the overall performance. However, contemporary aligners are all purely software-based, leaving the potential for further acceleration unfulfilled. One important reason for this is that prior work ignored the complexity of integrating a standalone hardware accelerator into state-of-the-art aligners. This chapter shares our experiences of integrating a hardware accelerator into a prevalent aligner to fulfill the potential.

8.3 Architecture of the FPGA Accelerator

In this section we will discuss our design for accelerating the modified Smith-Waterman algorithm used for the seed extension step.

8.3.1 Key Observations

Our acceleration engine design is based on the following three important observations derived from an analysis of the S-W implementation in BWA-MEM.

Observation 1: Enormous Task-Level Parallelism. A sequencer can generate billions of reads from a single individual for analysis in todays NGS flow. The huge amount of data generates enormous task-level parallelism, prompting us to reexamine the conventional wavefront technique for S-W acceleration. The conventional wavefront technique exploits the inner-task anti-diagonal parallelism to maximize the speedup for a single task. However, the wavefront implementation is not optimal

when the task-level parallelism is several orders of magnitude larger than the inner-task parallelism. For an accelerator platform with limited resources, e.g., an FPGA board with a certain amount of LUTs (DSPs, BRAMs, etc.), we must decide if the resources should be allocated for exploiting the task-level parallelism or the inner-task parallelism.

Observation 2: Significantly Varied-Size Inputs. For simplicity without losing generality, we abstract the total resources of an accelerator platform into a given number of unified processing elements (PEs). We also assume that each PE is capable of producing one value per cycle to fill the 2D table in the DP algorithm. A kernel is composed of a group of PEs and can be assigned to execute one S-W task. We denote mxn as a pair of input strings for the S-W algorithm with length m and n.

The sharply varied input sizes of S-W in BWA-MEM result in a considerable waste of resources in wavefront-based designs. For example, a kernel of 10 PEs is only able to reach a maximum of 65% resource utilization for a 13x103 input because the length of the maximal antidiagonals (13) is not divisible by the number of PEs (10). It takes 2 cycles for 10 PEs to fill an anti-diagonal with 13 elements. Figure 8.1 provides a histogram of the sizes of the shorter strings (bounding the maximum degrees of parallelism) over 10M inputs of randomly selected BWAMEM S-W tasks. The sizes range from 1 to 84, and none of them has more than 5% of the 10M inputs. The significant diversity of input sizes makes it prohibitive to choose one or a few kinds of PEs to avoid wasting resources. A better choice is to have each kernel restricted to only one PE, which means the anti-diagonal parallelism gets totally ignored.

Observation 3: Pruning Strategies. Derived from the X-dropoff pruning strategy in BLAST [13], BWA-MEMs pruning strategy is able to save over 50% in

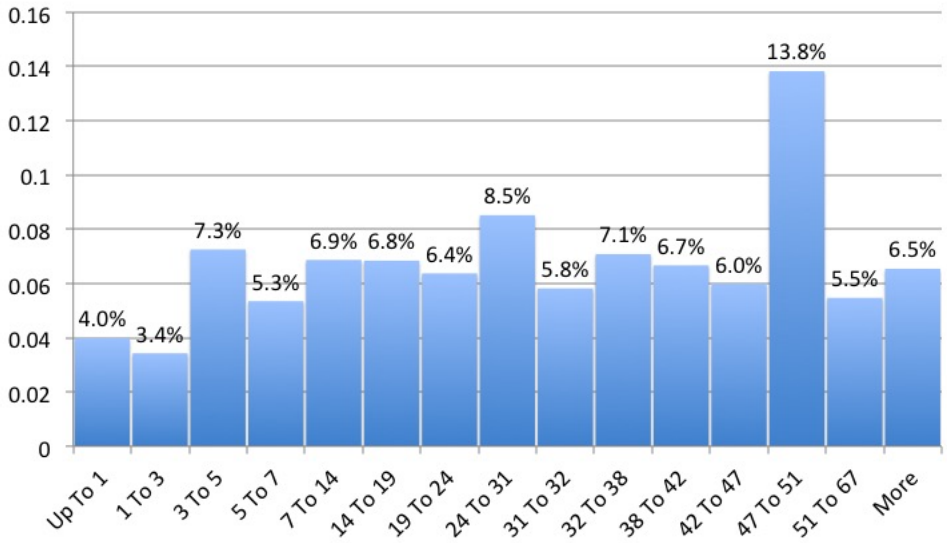


Figure 8.1: Histogram of the lengths of the shorter input strings collected from 10 million randomly selected BWA-MEM S-W tasks.

computation efforts for S-W tasks, as illustrated in Figure 8.2. However, the pruning strategy destroys the basis of the anti-diagonal parallelism, as described in detail in [111]. Moreover, the results generated by the optimized S-W algorithm in BWA-MEM are slightly different from those obtained by the standard S-W algorithm. This increases the difficulty of integrating existing wavefront-based work into BWA-MEM for concerns of result credibility. Even if the difficulty of integration can be overcome, the potential speedup from pruning would have to be sacrificed due to its incompatibility with the wavefront technique. It will be even worse when the sizes of seeds become longer, which is the future trend for NGS.

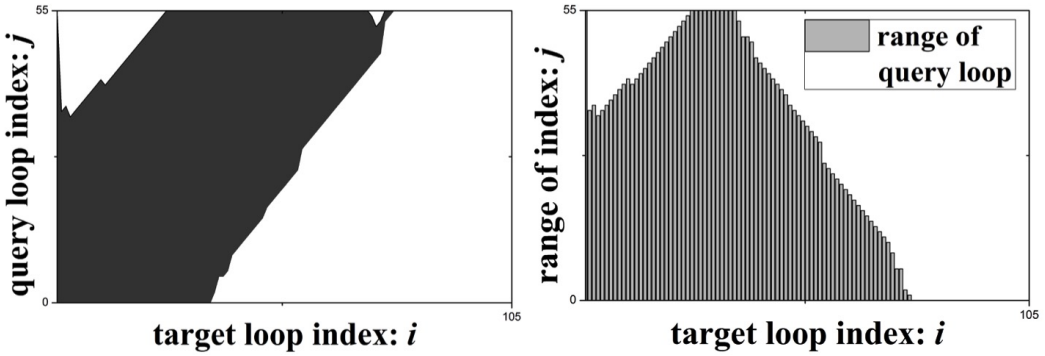


Figure 8.2: A 55x105 BWA-MEM Smith-Waterman task. The general S-W algorithm requires filling up a 55x105 matrix (5775 elements), but only 2836 elements (49%) were actually filled with the help of pruning. The black area in the left graph illustrates the elements that got filled, and the right graph shows how many elements for each target loop index are actually calculated.

8.3.2 Architecture Design

8.3.2.1 Overall Architecture

Figure 8.3 shows the overview of our Smith-Waterman accelerator engine. It consists of multiple PE arrays. Each PE array has a task distributor connected to the off-chip memory via an AXI bus interface. Each PE acts as one kernel and can take one S-W task at a time to maximize throughput. This strategy is used to take the advantage of the first observation of enormous data-level parallelism, as discussed in Section 8.3.1. In order to reduce the round-robin scheduling overhead rising from the large number of PEs, we design our architecture to be two-level task distribution. Note that our accelerator engine is customized for the modified Smith-Waterman algorithm used in BWA-MEM and CS-BWAMEM, which is different from the general-purpose Smith-Waterman algorithm.

Before the accelerator starts to operate, the host processor assembles a set of query and target sequence pairs and streams them to the on-board DDR3 memory via the PCIe bus. After that, each PE arrays task distributor fetches a certain number of sequence pairs and distributes them to the idled PEs. The mapping results are stored in the on-board memory and then sent back to the host.

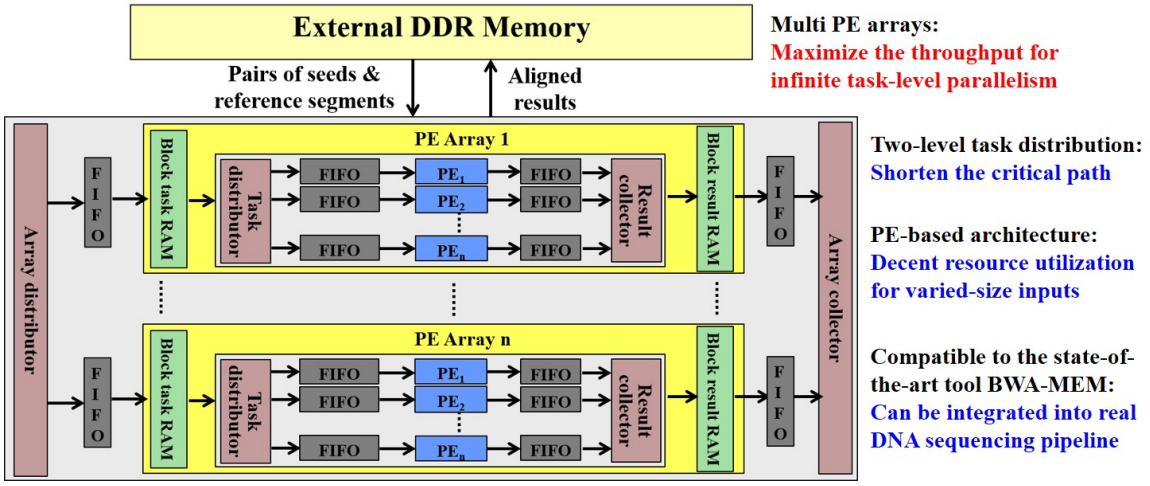


Figure 8.3: The architecture of the Smith-Waterman accelerator.

8.3.2.2 Processing Element (PE) Design

By using the high-level synthesis methodology, the hardware structure of a PE is created based on the S-W software code obtained from the software implementation of BWA-MEM. Unlike wavefront-based solutions that look completely different from the software structure, our design naturally follows the original software structure, which considerably shortens the development cycle.

To maximize throughput, we make the initiation interval (II) of the PE design be equal to one (II=1), i.e., calculating one cell of the dynamic programming score

matrix per cycle. Moreover, a conditional branch logic is implemented to realize pruning. It enlarges the size of each PE by 20%, but reduces computation effort by over 50%.

8.3.2.3 Two-Level Task Scheduling

The first level of task scheduling resides in a PE array. A PE array includes three types of major components: (1) a *task distributor*, (2) a set of *PEs*, and (3) a *result collector*. The task distributor fetches data from the on-board DRAM to the on-chip BRAM and dynamically dispatches the tasks to each PE through a FIFO. The result collector receives mapping results from each PE through a FIFO and packs them together for the host.

If a centralized task distributor is not provided, each PE needs to fetch data from on-board memory independently. This incurs a significant (25%) area overhead per PE to synthesize its own AXI interface. To reduce the overhead, we use only one AXI bus interface for each task distributor per PE array. A PE array, acting as an AXI bus master, fetches a group of data that satisfy the off-chip bandwidth demand of all PEs in the array. We also implement pingpong buffering by using a pair of BRAM blocks for better performance. After the data are prefetched to BRAM, the distributor checks the flag of the FIFO of each PE one by one to find an available PE. The task distributor then transfers the S-W tasks from BRAM to the FIFOs of the PE. This process continues until all the tasks are processed. The result collector continues monitoring the output FIFO of each PE and obtaining results until all the results are received.

If the number of PEs is small, the one-level task scheduling is sufficient. How-

ever, when the number of PEs is increased to a certain point, e.g., 50, the distributor becomes a performance bottleneck due to its round-robin task scheduling scheme. Therefore, we introduce a two-level task scheduling scheme which feeds tasks to multiple arrays, each with its own task distributor. This two-level hierarchy provides us with a scalable design methodology for obtaining scalable speedup.

8.3.2.4 Performance of the Acceleration Engine

To demonstrate the speedup of our accelerator engine over the original Smith-Waterman kernel used in BWA-MEM, an Intel Haswell Xeon server with two 6-core CPUs is used. The server can run 24 threads in parallel, with hyper-threading support. We compare the execution time of our FPGA acceleration engine to pure software-optimized S-W algorithm runtimes with 1, 2, 4, 8, 16 and 24 threads. To make a fair comparison, only the computation time of the S-W calls (after loading inputs from memory and before storing outputs to memory) is collected.

Figure 8.4 shows the performance comparison with results normalized to the single-threaded CPU performance. Our FPGA design outperforms the single-thread CPU and 24-thread CPU by 343.8x and 26.4x, respectively. This justifies the efficiency of our FPGA acceleration engine.

8.4 CS-BWAMEM with FPGA Acceleration

In this section we first describe our cluster setup, including computing and storage frameworks and hardware accelerators. Next, we discuss our cluster-scale aligner design and proposed optimization strategy. We mainly focus on (1) the uploading stage

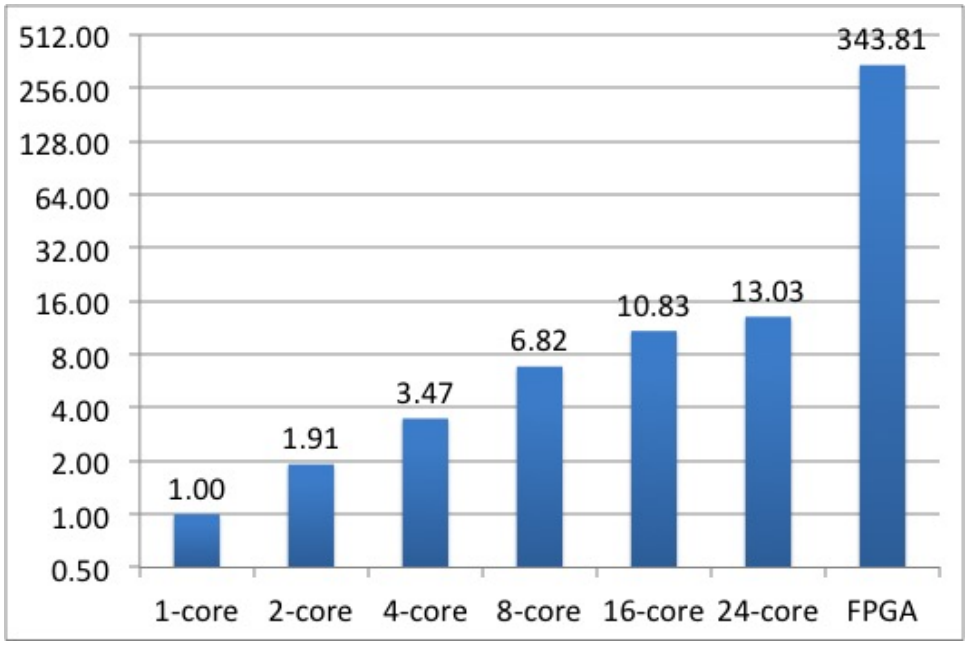


Figure 8.4: Performance comparison between our FPGA acceleration engine and the multi-threaded S-W software kernel.

and (2) the first MapReduce stage, including seeding and seed extension, described in Section 7.3. We will not discuss the pairing and output generation stage.

8.4.1 Cluster Setup

We use Apache Spark as our cluster computing framework and HDFS as the storage framework. Figure 8.5 shows a cluster with one master node and N worker nodes. Note that a HDFS datanode and a Spark worker are deployed together in the same node so that the Spark worker can fetch data from the local node for better data locality. We launch our jobs on the driver node. Then, the driver can send job requests to the master node. Each aligner job is decomposed into many small map tasks running on the Spark workers. Each node has an accelerator manager, which

handles the requests for the hardware accelerator that were sent from the map tasks. The accelerator manager is also in charge of the data transfer between the map tasks and the accelerator by copying data from the host program to the device memory. The interconnection network of the cluster is at the 10Gb Ethernet speed.

For each node, we have one PCIe-based FPGA card, which works like a coprocessor, for providing hardware acceleration for the S-W algorithm. We use the Alpha Data ADM-PCIE-7V3 card designed for datacenter applications [1]. Each card has a Xilinx Virtex 7 FPGA and 16GB on-board DDR3 DRAM. The host CPU process can send data to the on-board DRAM through the PCI-e Gen 3 interface. We use the Xilinx SDAccel flow with the Vivado high-level synthesis tool to design our accelerators and program the generated bitstream to the FPGA card [8]. The bitstream is preloaded on board before the read alignment jobs are launched.

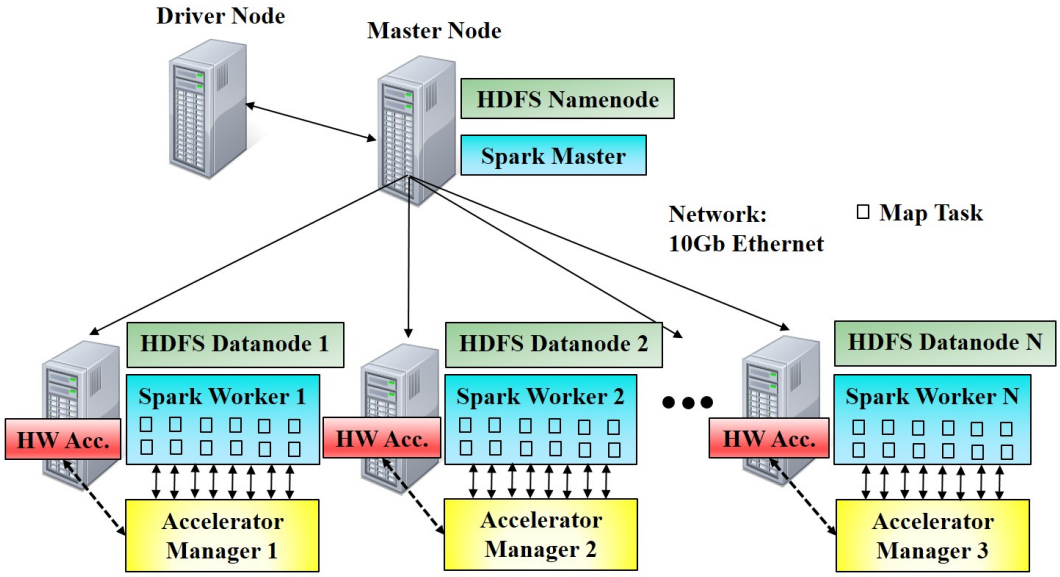


Figure 8.5: Cluster setup

8.4.2 Aligner Software Architecture

Figure 8.6 demonstrates our first two MapReduce stages. We partition the input FASTQ files into many independent data partitions In Spark, each data partition is scheduled as a map task, and computation is executed independently and distributedly in a cluster. Thus, we can exploit the huge task-level parallelism by leveraging the computation power all over a cluster.

In the first MapReduce stage, the reads in the FASTQ format need to be uploaded to HDFS. The FASTQ format is a standard text-based format to represent DNA reads. Conventionally, the whole genome FASTQ data is stored in a Linux file system. In our aligner, Avro [2] is used to specify the data schema of the FASTQ format. With Avro, the FASTQ reads can be stored and sent during computation between nodes in the cluster.

In the second MapReduce stage, the major computation of the read alignment is performed. It can be decomposed into two compute-intensive functions: exact mapping and inexact mapping, as discussed in Section 7.2.2. The exact mapping is developed using pure software implementation while the inexact mapping can be accelerated through the FPGA acceleration engine. Our aligner supports both the widely used SAM format [114] and the distributed ADAM format [127] for further analysis in the DNA sequencing.

8.5 Batch Smith-Waterman Algorithm

In this section we describe our approach for exploiting the potential of FPGA hardware acceleration solutions. In sum, the gap between an impressive PCI-e based

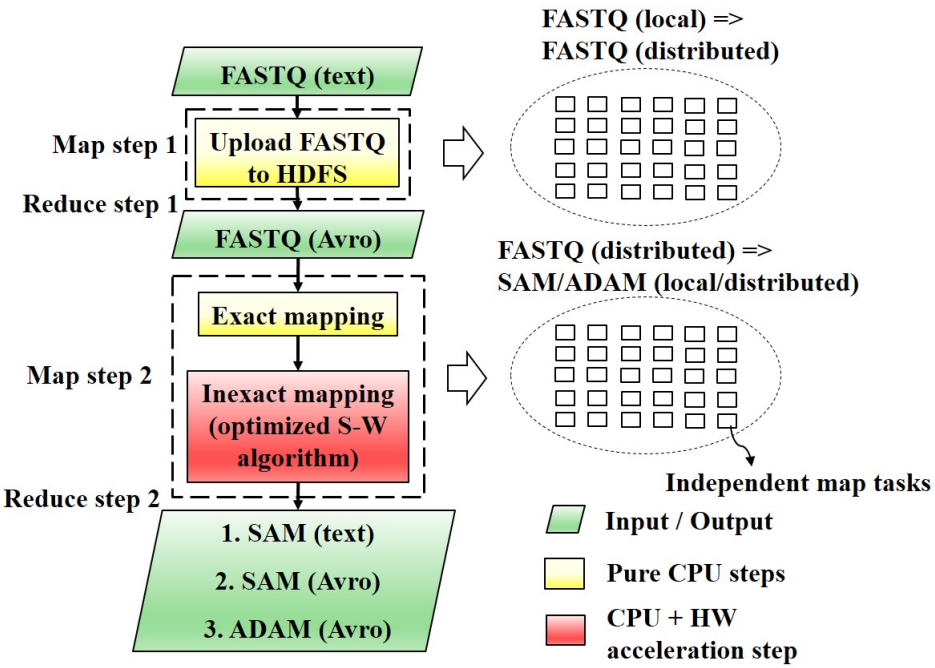


Figure 8.6: Cluster-scale read aligner: overview

accelerator and an integrated software/hardware system is surprisingly large. In this section we discuss how to design an efficient algorithm that maps well to our proposed FPGA accelerators.

8.5.1 Overhead of Data Transfer Between CPU Host and FPGA

In order to reduce the communication overhead between accelerators and processors, we proposed the batch processing strategy, which groups a series of data and sends it to accelerators all at once. In this case, the accelerators can perform a series of computations on the received data without interruption. Table 8.1 shows the communication overheads introduced between the host program and the FPGA accelerators in different batch sizes. The batch size is the number of tasks in a batch. The total

number of tasks in 512K in the example. For example, if the batch size is eight, the number of batches is $\frac{512K}{8} = 64K$. These batches are processed in a sequential way. The processor would communicate with FPGA through PCI-e bus back and forth to process these batches. We find that the communication overhead of the smallest batch (batch size = 1) is around 22.6x larger than the largest batch (batch size = 512K). Therefore, batch processing is an important way to reduce data transfer overhead.

Table 8.1: Overhead of communications between processors and FPGA accelerators

batch size	# of batches	overheads (s)
1	512K	**13.8**
2	256K	6.72
4	128K	3.45
8	64K	1.81
16	32K	0.995
...	...	...
64K	8	0.610
128K	4	0.610
256K	2	0.609
512K	1	**0.611**

8.5.2 Building Block: FPGA Smith-Waterman Acceleration Engine

Most hardware acceleration work focuses on accelerating the general versions of the algorithms to make the accelerators widely used; however, software aligners modify the algorithms for their own optimization purpose. BWA-MEM adopts a pruning strategy, called X-dropoff, to improve the performance of its S-W kernel, but the

results generated by the revised S-W kernel are slightly different from those generated by the general S-W algorithm. Moreover, the input size of the S-W kernel in BWA-MEM varies drastically. Therefore, the accelerators or SIMD-based algorithm designed for accelerating the general Smith-Waterman algorithm cannot be directly applied [68, 145, 138, 123, 174, 170, 100].

The BWT-based backtracking kernel encounters an even worse situation. While the acceleration for the general backtracking algorithm proposed in [112] has been well studied, BWA-MEM adopts the SMEM algorithm [110], which has not received enough attention yet. It is even extremely difficult to find a roughly similar hardware accelerator to start with.

As a result, we first focus on designing a customized FPGA accelerator for BWA-MEM and CS-BWAMEM. The detailed architecture design of the FPGA accelerator has been published in [35]. We use it as a building block for accelerating the S-W kernel in our integrated software/hardware system. The modified Smith-Waterman algorithm consumes 30%-40% of the overall execution time of BWA-MEM. Our system does not include a backtracking building block for now. Nevertheless, we are still working on designing accelerators for the SMEM algorithm in order to obtain higher aggregate speedup.

8.5.3 Batch Processing: Reduce Communication Overhead

When integrating a hardware accelerator into a software system, one critical issue that has a strong impact on performance is the data transfer overhead between the accelerator engine and the application host, as discussed in Section 7.4.3. The data transfer overhead can be significantly reduced if the data are sent in a coarse-grained

fashion to the accelerators; for example, a software program spends some time preparing input data for an accelerator. In this case, the overhead is even negligible if the computation time is orders-of-magnitude longer than the data transfer time. However, the behavior of calling the S-W kernel in BWA-MEM reaches the opposite extreme. By analyzing the original algorithm in BWA-MEM, we observed that the S-W kernel is called in a drastically finer-grain in BWA-MEM. On average, each S-W function call is fed by an 30x100 input and consumes about 20 μs in CPU, based on our profiling results. Meanwhile, a sequencing task calls the kernel hundreds of billions of times. Since the time to make a request to the FPGA board typically needs thousands of nanoseconds, it takes days for all the S-W calls to merely communicate with FPGA if we send data in such a fine-grained way.

Given this fine-grained pattern, our system processes the S-W function calls in a batched fashion. Figure 8.7 illustrates the effectiveness of batch processing by measuring the utilization of the bandwidth of an FPGA's private DRAM in different batch sizes. We initially feed a certain number of input data for the S-W kernel into an FPGA's private DRAM, and then collect the actual bandwidths of different batch sizes. We can see that the utilized bandwidth improves significantly with the increase of the batch size. Although the communication between an FPGA and its private DRAM is not the only communication overhead, it is sufficient to show the power of batch processing.

The proposed batch processing mechanism works well for BWA-MEM as the alignment tasks of different reads are completely independent, enabling us to process a batch of S-W calls from different reads in parallel. (The S-W calls generated by the same read, however, are strongly dependent on each other. See Section 8.8 for more details). Since billions of reads need to be aligned in a sequencing task, the degree

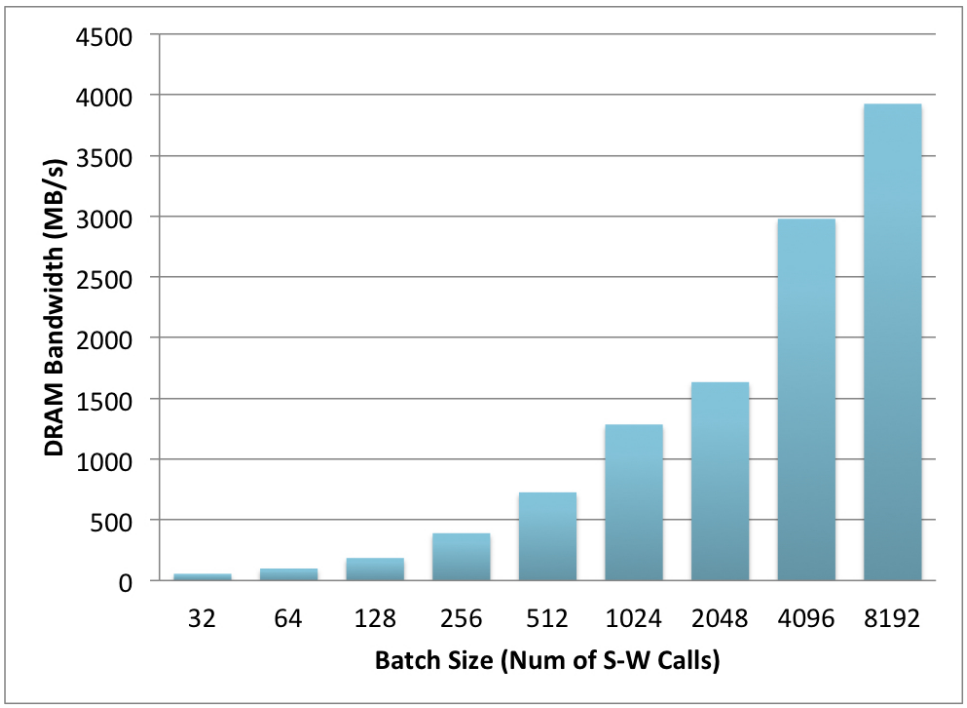


Figure 8.7: DRAM bandwidths for different batch sizes

of parallelism is adequate for our system to form large batches to reduce the communication overhead. We reimplement the BWA-MEM algorithm, realizing batch processing along with the MapReduce programming model. Each map task consists of a configurable number of reads to be processed in batch, which can be as large as a whole HDFS partition and as small as containing only one read. Our implementation not only maintains good scalability, but enables hardware accelerators to be integrated into our system.

8.5.4 Batch Size Selection: Larger Not Always Better

Batch processing of the S-W kernel in BWA-MEM is not as straightforward as simply unrolling a loop. BWA-MEM is a highly complicated algorithm with 15,741 lines of

C code in total, and the S-W kernel is deeply located inside the hierarchy of BWA-MEM's function calls—that is, the eighth nest from the main function. Extracting the kernel for batch processing not only takes considerable engineering effort, but generates more intermediate data, which is in proportion to the batch size. A large batch size incurs a large memory footprint, resulting in a longer garbage collection time. This reduces the performance gain from the reduction of the communication overhead, especially when many threads are used. Therefore, there is a trade-off between the reduction of data transfer overhead through a large batch size and an adequate memory footprint through a small batch size. Moreover, different system settings may lead to different optimal batch sizes. The batch size is implemented as a configurable parameter in our system with the default value 32,768, based on the experiment described in Section 8.8.2.

8.5.5 Thresholding: Addressing the Long-Tail Problem

It is not always efficient to send all the tasks to FPGA for acceleration when the gain from acceleration is smaller than the loss obtained from data transfer. In BWA-MEM, the number of S-W function calls generated by a read is data-dependent. Different reads may trigger enormously varied numbers of S-W calls, ranging from 0 to over 8000. Based on the algorithm of BWA-MEM, the S-W call of seed in one read strongly depends on the extension result from the previous seed. Therefore, the alignments of all seeds in a read can only be done sequentially. Figure 8.8 illustrates this dependency inside a read. For each row, it represents the number of seeds of a read. For example, read 0 contains 11 seeds while read 2 has only two seeds. Our batch processing algorithm would process the first seeds from all the reads in the batch together. After the first seeds of all reads are processed, the algorithm will try

to process the second seeds (if there are any) of all reads. This process will continue until all the seeds are processed.

Due to the significantly varied behaviors of the read, each read has different numbers of seeds. We can observe that the batch size will be gradually reduced. However, we observed that when the batch size is too small, the communication and data transfer overhead actually dominate the execution time, offsetting the performance gain brought from FPGA acceleration. To address this issue, a threshold is needed to determine whether a batch needs to be processed on FPGA or we can simply use CPU for computation. An appropriate threshold is critical to the performance of our system. Like the batch size, the threshold is also implemented as a configurable parameter in our system with the default value 64, based on the experiment described in Section 8.8.2.

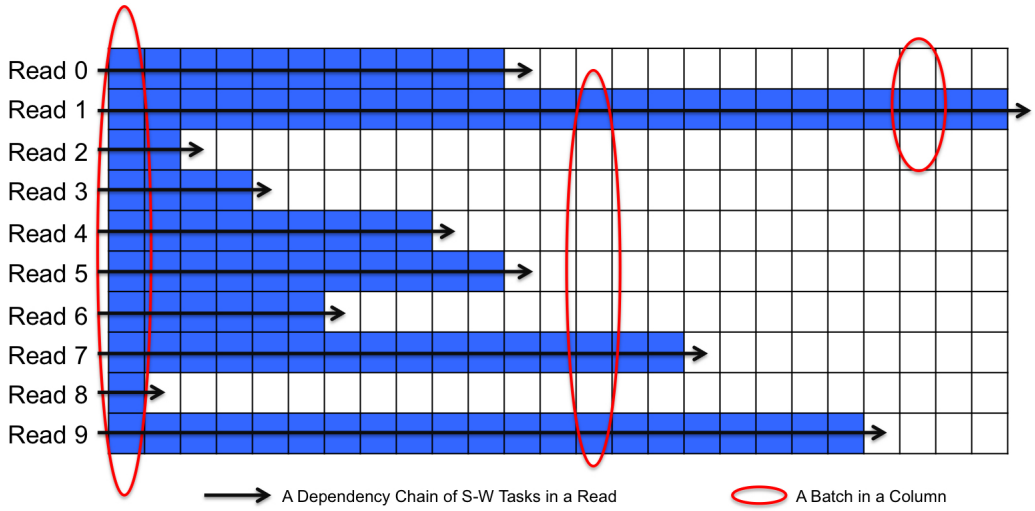


Figure 8.8: Dependencies and batching in a batch of reads

In sum, in order to leverage the acceleration benefits brought by FPGA, we need to carefully redesign the algorithm to avoid communication and data transfer overhead.

Batch processing is a powerful way to resolve this problem. However, the parameters, such as an adequate batch size and a threshold for using FPGA accelerator, need to be carefully determined to achieve the best performance.

8.6 Accelerator Manager Design

In the current SDAccel flow, we need to have a dedicated software accelerator manager (AM) to send requests, copy data from the host program to device memory, and wait for the results from the FPGA accelerator. This is because many threads inside a Spark worker will contend for the FPGA resources. AM is needed for arbitration purposes. In this section we first discuss the challenges encountered for supporting intense accelerator accesses from multiple S-W tasks. Next, we demonstrate our AM design that addresses these issues.

8.6.1 Design Challenges

The goal of the AM design is to provide a handshaking protocol between S-W map tasks and the FPGA accelerator. The AM is also in charge of the data transfer. For performance concerns, the AM needs to minimize the communication overheads so that the map tasks can efficiently leverage the power of the FPGA accelerator. We find the that following requirements need to be satisfied.

The first design challenge we encounter is that the accelerator access pattern is a large number of fine-grained and discontiguous accesses. A conventional way to efficiently use GPU-like coprocessors is to copy a large amount of data from the host program to the device memory, and then acceleration can be achieved by using abun-

dant SIMD engines. The data transfer and the computation is usually done at a coarse granularity. However, since the S-W algorithm used in BWA-MEM introduces strong inner-task dependency, as discussed in Section 8.5, we are not able to simply send a large number of reads directly to the FPGA coprocessor. The proposed batched Smith-Waterman algorithm can effectively group data and send it to the accelerator. However, the batched algorithm still generates a series of fine-grained and discontiguous requests for the FPGA accelerator.

Second, the accelerator needs to service the requests from multiple map tasks. In our cluster-scale aligner, the FASTQ data are stored and computed as small tasks in a distributed way. Each task is processed by one CPU core. Therefore, multiple requests can be sent from different map tasks to the AM simultaneously. The Hyper-Q technology in the Kepler GPU provides the capacity that allows 32 simultaneous connections to share the GPU resources [6]. However, neither the current SDAccel flow nor the S-W accelerator engine we select can support simultaneous accesses. The AM needs to act as an intermediate layer to arbitrate the accesses from multiple processes.

Third, our cluster-scale aligner is developed using Scala and built on top of the Spark framework, which runs on the Java Virtual Machine (JVM). However, the accelerator is a native application built from C to interact with the FPGA accelerator. We need to provide a handshaking protocol and data transfer mechanism between the Spark tasks and the AM.

8.6.2 Accelerator Manager Design

Figure 8.9 shows the AM design and the handshaking protocol. We launch one AM per node to handle the accelerator requests from the map tasks in this node. The POSIX shared memory is used for exchanging data and the done signals of data transfer between the AM and the map tasks. Since the shared memory is allocated by the map tasks during runtime, we use sockets to send the shared memory ID to the AM so that the AM can fetch data from and write results back to the assigned shared memory region.

When a map task enters the batched S-W algorithm section, it will send a series of batched requests to the AM for FPGA acceleration, as described in Section 8.5.3. This process is implemented through the Java Native Interface (JNI) and built as a library to be loaded in JVM during runtime. For each request, a shared memory is created and the required input data for FPGA acceleration is loaded to the allocated shared memory region (step (1)). Next, the map task tries to send the shared memory ID to the AM through a socket. The map task will keep trying to deliver the ID until the AM is not busy (step (2)). After that, the map task will continue checking the register v_2 until the AM finishes the request. The shared memory needs to be freed once the task is completed.

In the AM, we first program the bitstream to FPGA and create the default read-/write device buffers during initialization time. This is a one-time process. Next, the AM starts to listen to the socket to receive the requests from map tasks. After receiving the shared memory ID, the AM begins to process the current request for the specific map task. The register v_1 is set by the map task when data is ready. Once v_1 is set, the AM can read the data from the designated shared memory and

then write the data to the on-board device memory using the API, *write_dev_buffer()*, provided by the SDAccel flow. The *enqueue_task()* function sends a start signal to the FPGA accelertor. The AM will wait until the task finishes and then the results can be read from device memory. At the end, the AM writes the results back to the shared memory and updates the v_2 register so that the map task can get the results.

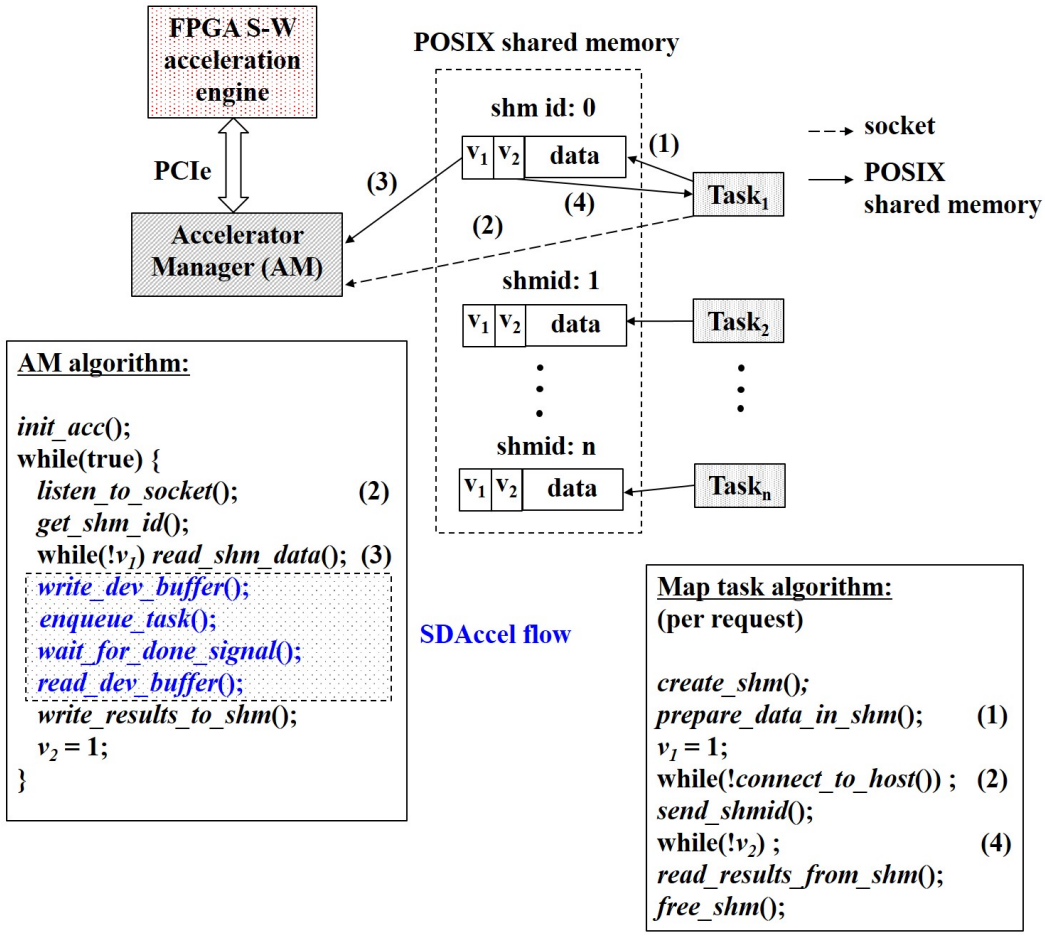


Figure 8.9: Accelerator manager design and the handshaking protocol

Since we have only one FPGA accelerator per server node, the accelerator resource cannot be shared once it is occupied. Therefore, the sequential FPGA accelerator accesses do not harm performance since the accelerator can only service one request

at one time. The potential inefficiency of the current AM design is that it needs to read the shared memory of the requests from multiple map tasks in a sequential way. This can be hidden if each map task can directly communicate to the FPGA accelerator with a suitable arbitration scheme. However, it cannot be achieved based on the current SDAccel flow, and SDAccel is the only way to use the Alpha Data card. Based on our evaluation, the data transfer time is currently not the performance bottleneck. Compared to the time spent on the FPGA accelerator, it accounts for only 10% of the time spent on FPGA hardware.

8.7 Evaluation Methodology

8.7.1 Software and Hardware Overview

Our system comprises a cluster of 12 Intel Haswell Xeon servers as well as six Xilinx Alpha Data FPGA boards. Each server equips two Intel Xeon E5-2420 microprocessors with 12 cores in total and 64GB memory. The system implements the BWA-MEM algorithm presented in BWA 0.7.8 by harnessing the Apache Spark MapReduce framework.

The system is tested on our cluster with Spark 1.2.1 and Hadoop 2.5.2. The testcases are derived from a real genome sample with breast cancer (HCC1954), which contains over 1 billion reads (300GB large). While the software implementation can fully utilize all of the 12 server nodes, the integrated software/hardware system is only tested under a subset of six nodes due to the insufficient number of FPGA boards. However, the effectiveness of the joint system can still be demonstrated in the 6-node cluster and easily projected to the entire cluster.

The hardware accelerator proposed in [35] is adopted as the building block for accelerating the S-W kernel in BWA-MEM. Accelerators are synthesized on the Xilinx Alpha Data ADM-PCIE-7V3 boards which are equipped with Xilinx Vertex 7 FPGAs. The six Alpha Data boards are located in six servers, forming a 6-node heterogeneous cluster for launching our joint software/hardware system. The communication between a server and an FPGA board is facilitated through the PCIe bus. The SDAccel development environment is used for accelerator configuration, synthesis and integration. We mainly test our system using an accelerator with 60 processing elements (PEs). We also use 40 PEs for performance comparison.

8.7.2 Profiling Methodology

In Spark, a large number of map tasks are created and launched in the cluster. These map tasks are executed in parallel, and the executions are interleaved with each other. In order to get a detailed profile of our cluster-scale aligner, we design a profiler to extract the runtime of each step. In the reduce stage, we collect the statistics from all map tasks and accumulate them. Therefore, we can identify the performance bottlenecks and reduce the design cycle easily.

In the evaluation section, the speedups of different numbers of nodes and the time breakdown of the AM statistics are the wall-clock time. The other detailed time breakdowns are the statistics collected from the profiler.

8.8 Evaluation and Analysis

8.8.1 Effectiveness of Hardware Acceleration

In order to understand the effectiveness of the hardware acceleration of the S-W algorithm, we need to first analyze the runtime breakdown between the exact mapping (using BWT) and the inexact mapping (using the S-W algorithm). We use our profiler to get the accumulated runtime from all the map tasks of the exact and inexact mapping steps. Figure 8.10 shows the statistics collected from a single node in the cluster. We compare the accumulated runtime breakdown between the pure-CPU aligner and the FPGA-accelerated aligner when using different numbers of cores. The performance of FPGA-accelerated aligner is normalized to the pure-CPU one.

We can observe important characteristics based on Figure 8.10. First, the BWT part accounts for more than 60% of the total runtime. The amount of time that can be accelerated by using the S-W accelerator is about 30% - 35%. Second, we can reduce the runtime of the S-W algorithm by more than 50% after all the communication overheads between the software map tasks and the PCIe-based FPGA accelerators are included. Third, we can reduce the overall aligner runtime by around 20% even when we launch map tasks on 12 cores in node. In Figure 8.10 we can observe that the runtime reduction only decreases slightly from 1-core (22.8%) to 12-core (16.2%). This means we can launch multiple map tasks concurrently with only one accelerator. It further demonstrates the effectiveness of our batched S-W algorithm and the accelerator manager.

To show the scalability of the hardware-accelerated cluster, we evaluate the effectiveness of hardware acceleration with a 6-node cluster. Figure 8.11 demonstrates

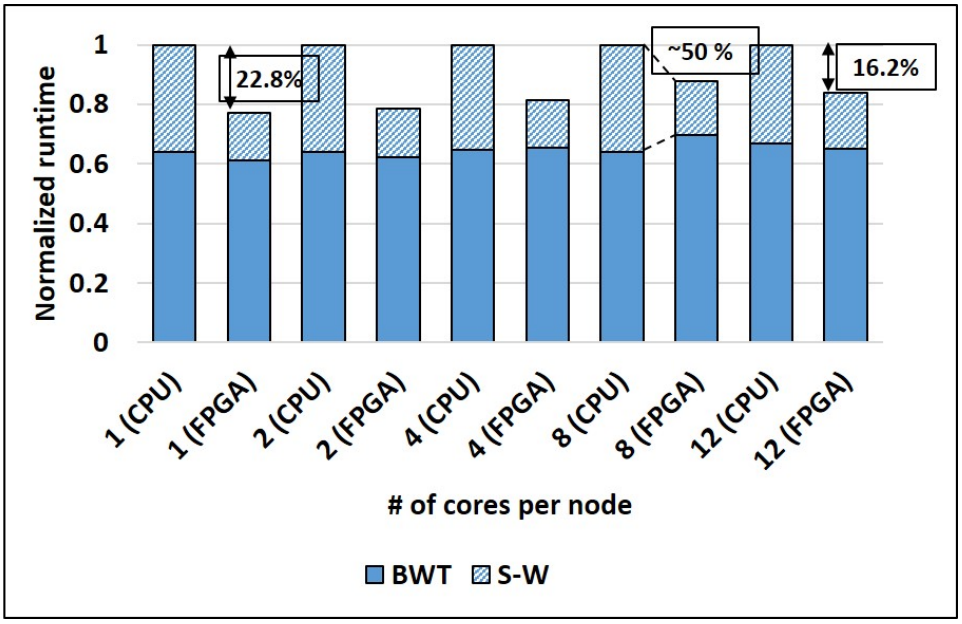


Figure 8.10: Runtime breakdown between the pure-CPU and the FPGA-accelerated aligners over different numbers of cores used in one node.

the performance gain of FPGA acceleration over different sizes of cluster. For each cluster size, the performance of the FPGA-accelerated aligner is normalized to the pure-CPU one. The FPGA accelerator can consistently reduce the runtime from 9% to 17% over the pure-CPU aligner. This leads to an extra speedup which is orthogonal to the benefits of using a cluster.

Figure 8.12 shows the scalability of the aligner with hardware acceleration. With six nodes, we can demonstrate a 5.23x and 4.70x speedup over a single-node pure-CPU aligner and a single-node FPGA-accelerated aligner, respectively.

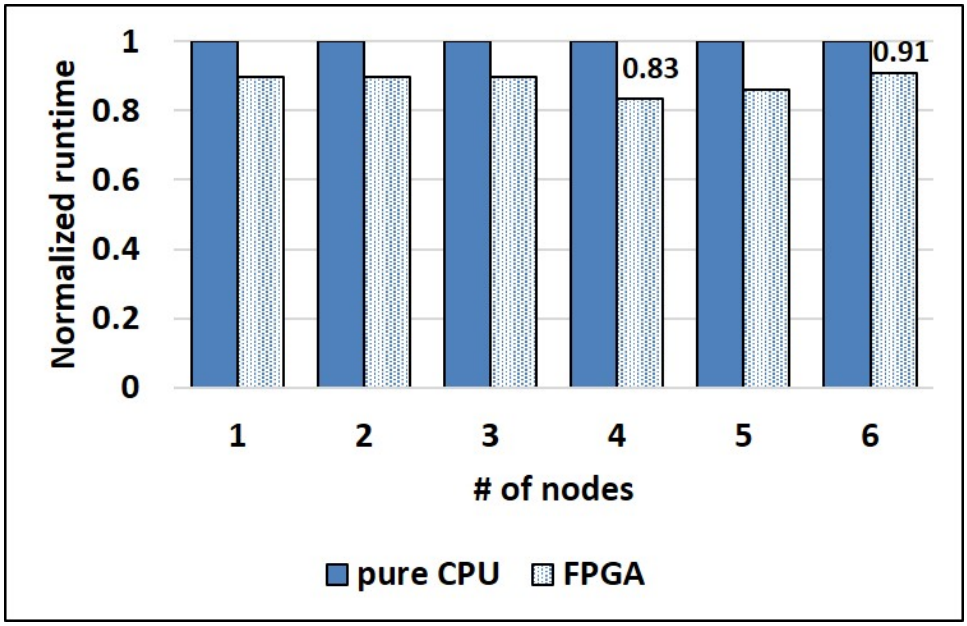


Figure 8.11: Effectiveness of hardware acceleration

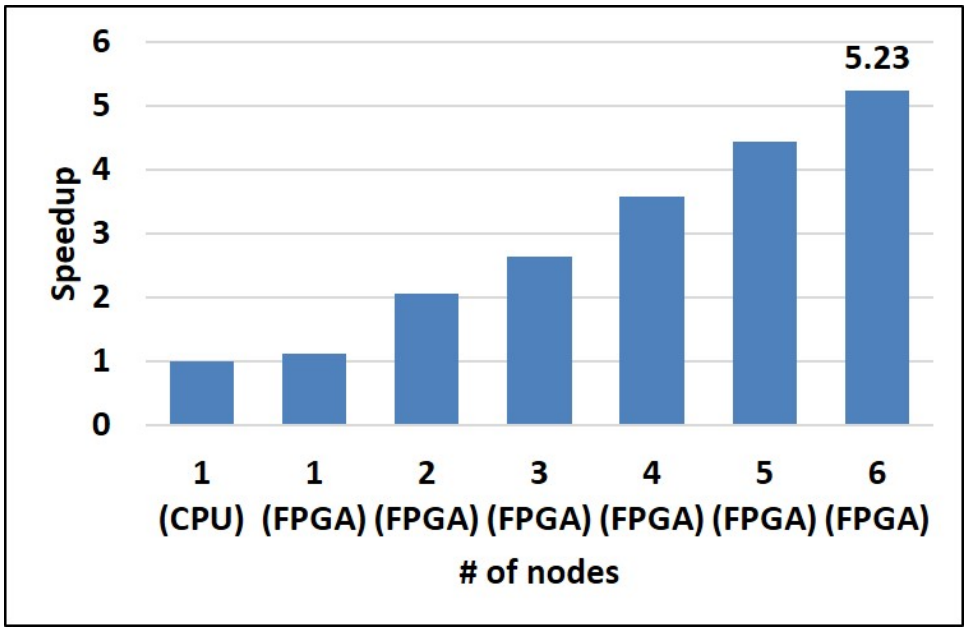


Figure 8.12: Scalability of the cluster-scale aligner with hardware acceleration

8.8.2 Performance of Batched S-W Algorithm

As discussed in Section 8.5, the batch size and threshold are two critical parameters that affect the overall performance of the joint software/hardware system. The following two sections evaluate the impact of the two parameters on performance.

8.8.2.1 Evaluation on Different Batch Sizes

Figure 8.13 shows the performance comparison among a group of joint hardware/software implementations with different batch sizes. The performance of each implementation is denoted by its execution time, which is collected from our 4-node, 48-core heterogeneous cluster and normalized to that of the implementation with batch size equal to 65,536.

We can see that the execution time gets shorter as the batch size increases, which can be as much as 26x, but becomes longer after reaching a certain amount, i.e., 32,768. This variation is consistent with the trade-off between lower communication overhead and larger memory footprint, as discussed in Section 8.5.4. We also find that the interference in contending for the FPGA accelerator by the 12 cores plays an important role in the overall performance. A smaller batch size leads to more batched tasks, triggers more severe interference, and eventually harms the performance. In addition, we observe that the garbage collection time of the 65,536-batch-size implementation is 33% more than that of the 32,768-batch-size implementation. A better garbage collection mechanism or a large physical memory is expected to increase the optimal batch size.

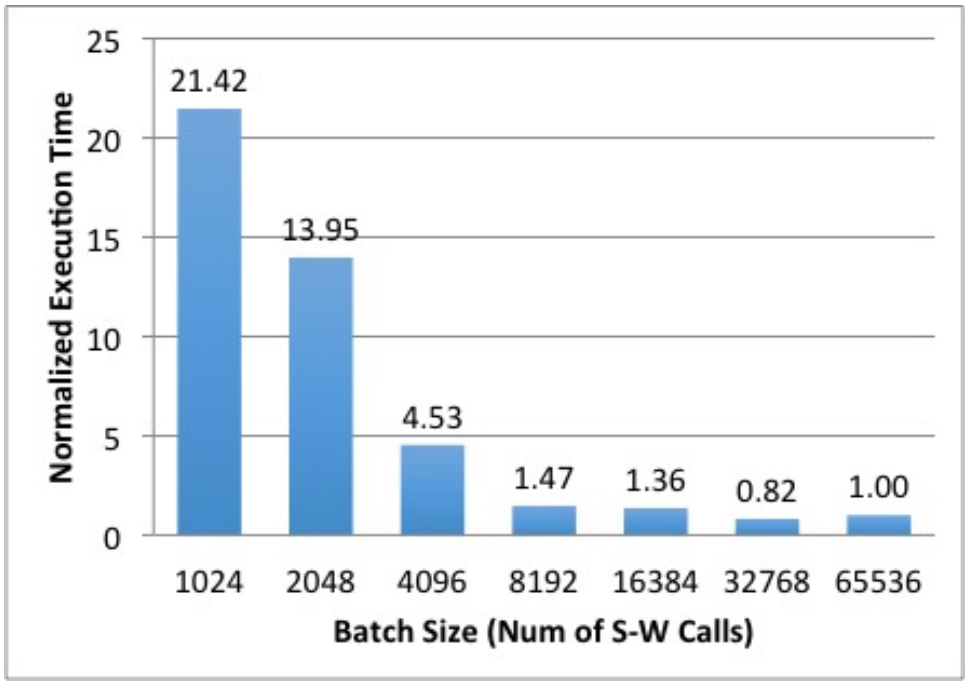


Figure 8.13: Performance comparison between different batch sizes

8.8.2.2 Evaluation on Different Threshold

Figure 8.14 shows the performance comparison among a group of joint hardware/software implementations with different thresholds. The performance of each implementation is denoted by its execution time, which is collected from our 4-node, 48-core heterogeneous cluster and normalized to that of the implementation with threshold equal to 256.

We can see that neither a too-large nor a too-small threshold can achieve optimal performance. Figure 8.15 shows the proportions between the number of S-W calls run on an FPGA and that of S-W calls run on CPU. We note that a larger threshold leaves more S-W calls on a CPU, some of which may run faster on an FPGA. On the other hand, as illustrated in Figure 8.16, the average execution time of a S-W

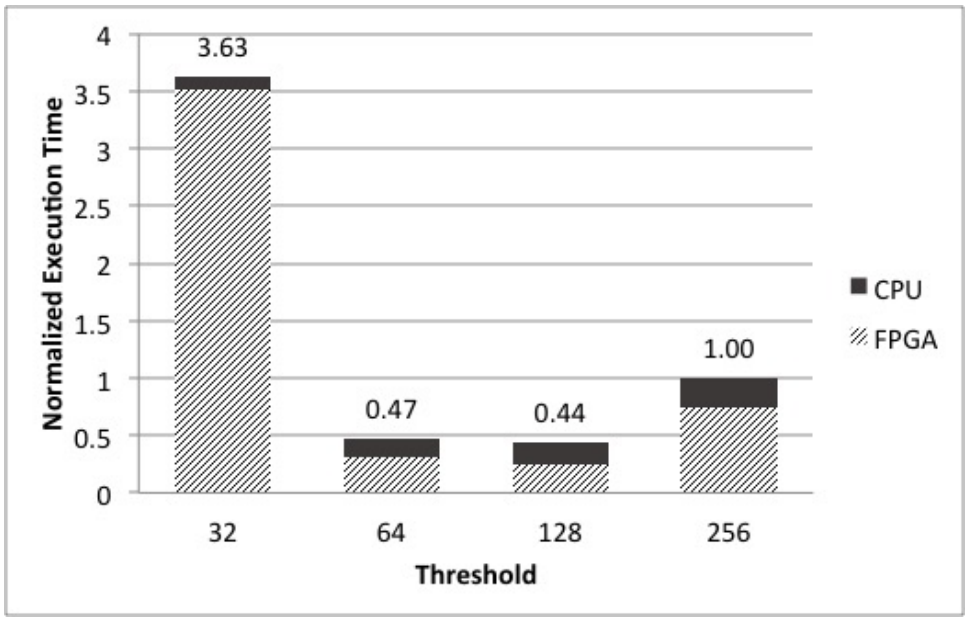


Figure 8.14: Performance comparison between different thresholds

function call in a small batch is considerably longer than that in a large batch.

Ideally, an optimal threshold is the batch size that equals the execution times of a batch on both CPU and FPGA. Nevertheless, the runtime of a batch is not only determined by the batch size, but the S-W calls inside the batch. The default threshold in our system, 128, is chosen based on the experiment results, and is expected to work well in the average case, though not in every case, for our system configuration.

8.8.3 Accelerator Manager Runtime Breakdown

Figure 8.17 shows the runtime breakdown of the AM. The runtime can be decomposed into three parts: (1) writing input data to FPGA device memory, (2) reading results from FPGA device memory, and (3) the real FPGA accelerator execution time. As discussed in Section 8.9, the real FPGA acceleration execution time accounts for 91%

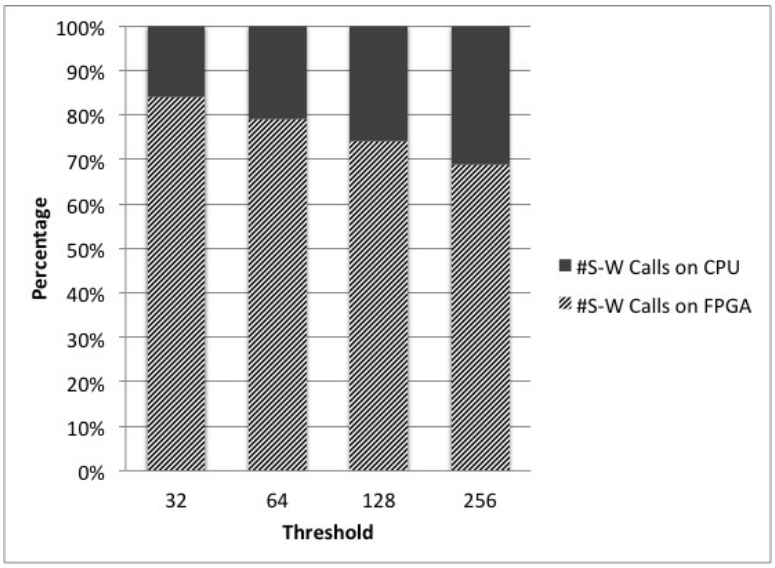


Figure 8.15: Proportions between the number of S-W calls on FPGA and on CPU under different thresholds

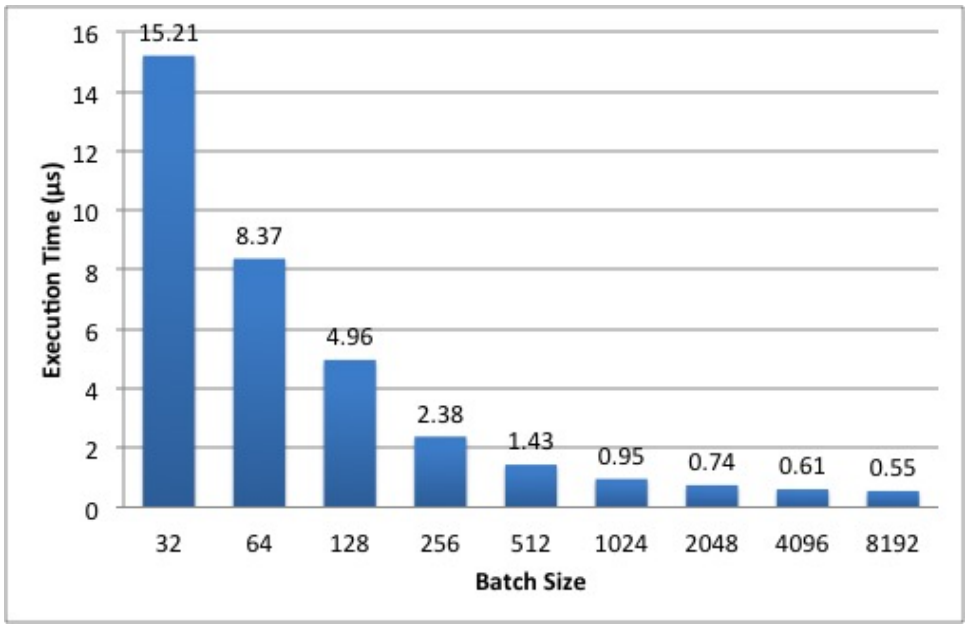


Figure 8.16: The average execution times of a S-W call in different sizes of batches

of time. The write/read device memory time is not the bottleneck for the AM.

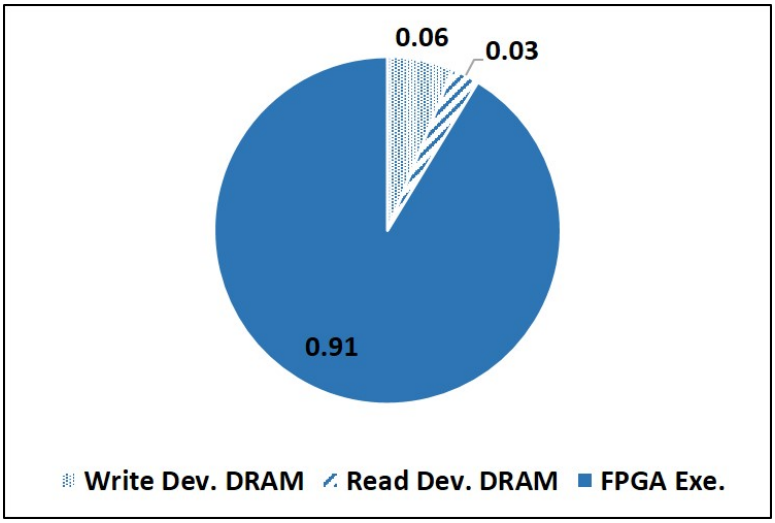


Figure 8.17: Time breakdown of the AM.

Figure 8.18 shows the utilization of the accelerator upon different numbers of cores, i.e., map tasks. When there is only one core access to the accelerator, most of the time the AM is listening to the socket channel and waiting for the accelerator request. When the number of cores increases, the demand to the accelerator increases, and thus the socket listening time shrinks signficantly.

8.8.4 Impact of Faster Accelerators

Generally, a faster accelerator is always better in terms of performance, but the effectiveness of a faster accelerator decreases based on Amdahl's Law. For example, if a computation kernel consumes 20% of the overall runtime, the replacement of a 10x-speedup accelerator with a 100x-speedup accelerator will have almost no benefit.

The situation is somewhat complicated in our system since an accelerator platform

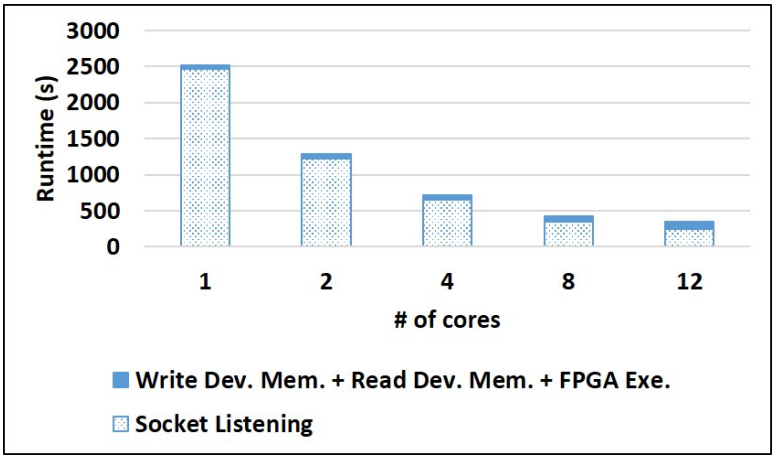


Figure 8.18: Socket listening vs. the data transfer with FPGA execution time

is shared by up to 12 cores. Figure 8.19 shows the performance improvement obtained by replacing a faster accelerator with more processing elements (PEs), i.e., 60 instead of 40, under different numbers of cores. Three kinds of performance improvements are collected and illustrated: 1) one that is directly related to FPGA, i.e., sending input to and receiving output from FPGA, as well as the FPGA execution time; 2) S-W kernel in BWA-MEM, which takes 30%-40% of the overall execution time; 3) performance gain of the whole BWA-MEM program.

We can see that the configuration with the larger number of cores benefits more from the faster accelerator. This is because a faster accelerator reduces the interference between multiple cores along with the acceleration of the FPGA runtime. As a result, the S-W kernel can be executed over 50% faster in the 12-core-per-node configuration. We currently observe that the faster accelerator is able to improve the overall performance by up to 7%, but it is expected to gain more if a suitable accelerator for the BWT kernel is developed and integrated into our system.

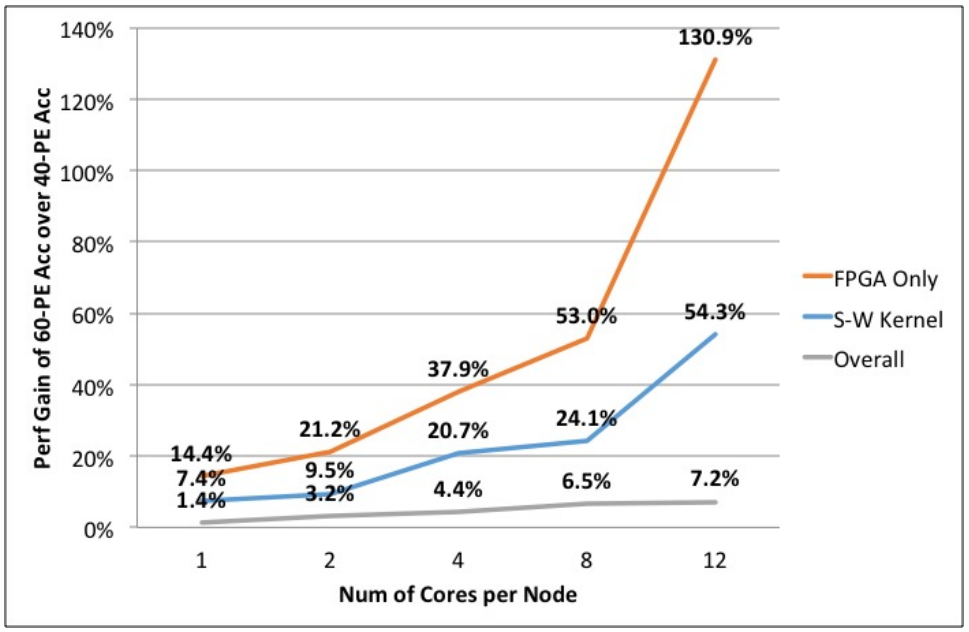


Figure 8.19: Performance improvement of a faster accelerator

8.9 Conclusions

In this chapter we aim to provide a more efficient read aligner than the state-of-the-art aligners, such as BWA-MEM and Bowtie 2. We first develop an FPGA-based accelerator to accelerate the modified Smith-Waterman algorithm that can be used in BWA-MEM and CS-BWAMEM. We further integrate the FPGA accelerator to provide hardware acceleration on the S-W algorithm. Furthermore, we develop a novel batched S-W algorithm and an accelerator manager to efficiently utilize the accelerator and remove the communication and data transfer overhead between map tasks and the FPGA accelerator. With hardware acceleration, we can reduce more than 50% of the runtime of the S-W algorithm and further reduce the system runtime by 10%.

The architecture design of the Smith-Waterman accelerator is an collaborative

effort among Prof. Jie Lei, Peng Wei, and myself. I contributed the key observations and the high-level architecture design, while Prof. Lei and Peng Wei determined the detailed design and implemented the FPGA accelerator. Peng Wei and I both contributed to the Batched Smith-Waterman algorithm design. Peng Wei realized the algorithm and integrated it into the CS-BWAMEM software suite.

CHAPTER 9

Concluding Remarks

Technical summary

In this dissertation we investigate the optimizations of memory system and in-memory cluster for customized computing. The efficiency of memory system determines the chip performance and energy efficiency. Our goal is to design a high-performance and energy-efficient memory system that supports customized computing in both general-purpose processors and accelerator-rich architectures (ARAs).

Many big data applications need the computation power beyond a single server. In order to provide acceleration on such applications, we adopt the in-memory cluster and provide corresponding optimization strategies. Furthermore, we provide support to deploy the customized accelerators in a cluster to improve system performance.

Table 9.1 summarizes our exploration efforts. From Chapter 2 to Chapter 4, we focus on the simulation techniques and energy-efficient hybrid caches design. In Chapter 2, we propose HC-Sim [40], which can simulate multiple L1 cache configurations with scratchpad memories simultaneously. In Chapter 3 and Chapter 4, we discuss energy-efficient L2 cache and LLC cache designs by using non-volatile memory technologies. The proposed hybrid cache design can significantly reduce leakage. With dynamic reconfiguration on hybrid caches, we can further reduce leakage when the

caches are not accessed frequently by processors [37]. To further reduce the dynamic write energy and the endurance of hybrid caches, we introduce a combined scheme by using compiler hint and monitoring hardware counter dynamically to reduce the writes on NVM cells [38]. We believe that, in the near future, NVM technologies can be widely used in on-chip memories. They can be used for both CPU caches or accelerator buffers.

From Chapter 5 to Chapter 6, we focus on the ARA memory system design, where an ARA is composed of a sea of accelerators, CPUs, and the memory system. In order to evaluate the system performance, we first proposed ARAPrototyper [36, 41, 34] in Chapter 5. ARAPrototyper provides a fast and a highly parameterized ARA design flow together with system software and user APIs. We can easily develop an ARA and utilize these APIs to develop applications that run on top of an ARA to collect performance statistics. In Chapter 6 we introduce an optimized crossbar synthesis algorithm, which highly reduces the crossbar complexity between accelerators and memory banks. We also introduce a two-level interconnect synthesis method for ARA memory system design [33].

In Chapter 7 and Chapter 8, we are interested in accelerating the DNA sequencing pipeline [111, 114, 129]. However, due to the big data behaviors and compute-intensive kernels of the applications in the pipeline, it is extremely time-consuming to run the pipeline in a single server. Therefore, we decide to use the in-memory cluster computing to provide a scalable speedup. In Chapter 7 We first try to accelerate the alignment step [111] in the pipeline. We proposed CS-BWAMEM [39] that utilizes the Spark in-memory cluster computing framework together with multiple customized optimization strategies, which can reduce the alignment time from 9.5 hours to 33 minutes. In Chapter 8 we provide runtime support and a customized

batch-processing algorithm that can be mapped to the PCIe-based FPGA accelerators. We demonstrate the capability and required support of using FPGA accelerators in an in-memory cluster.

Table 9.1: Summary of the proposed optimizations in the dissertation

Optimization target	Architecture components	Optimization goals	Simulation or emulation platform
On-chip caches **(Chapter 2-4)**	CPU, multilevel cache	energy-efficiency, endurance	HC-Sim [40]
ARA memory system **(Chapter 5-6)**	Accelerators, memory banks, CPU	performance, energy-efficiency	ARAPrototyper [36]
In-memory cluster **(Chapter 7-8)**	CPU, main memory (DRAM), FPGA boards (PCIe)	performance, energy-efficiency	evaluation on a real cluster

Future work

In this dissertation we discussed the memory system optimizations in both single-chip level and the datacenter level. For the single-chip level, we covered both energy-efficient hybrid caches and ARA memory systems. In the datacenter level, we provided customized optimization for the genomics domain in an in-memory cluster and supported the deployment of FPGA accelerators in the cluster. We summarize the lessons we learned and the possible future directions to be further explored.

1. **Cache simulation and hybrid caches**

   HC-Sim can efficiently simulate hundreds of single-level caches with different

architecture parameters. Meanwhile, HC-Sim can also co-simulate one scratchpad configuration with many L1 caches. One possible extension to HC-Sim is to provide the capability of simulating multiple scratchpad configurations at the same time. Another possible extension is to explore the miss rates of many multilevel caches simultaneously.

We provide the architecture and circuit-level implementation of dynamically reconfigurable hybrid caches. We show that dynamic reconfiguration can significantly reduce the leakage consumption. Furthermore, we provide compiler-assisted optimization with dynamic hardware monitoring to reduce the high dynamic write energy on the NVM cells while improving the endurance of hybrid caches. The important lessons we learned are about (1) the heterogeneity of SRAM and NVM cells, and (2) the dynamic behavior monitoring and tuning that can provide huge opportunities for optimizing energy efficiency.

In recent years, the research field of NVM caches and hybrid caches has been very active and has attracted much focus. However, the real product development of NVM caches is still ongoing. A possible research direction could be the optimization of the interaction between the L1 cache and the hybrid cache (usually used in L2 caches or LLCs). This is not addressed in this dissertation.

2. **ARAPrototyper and ARA memory system**

We build the ARAPrototyper to prototype an ARA in a highly automatic way, and users can also evaluate their own developed applications efficiently on top of their ARA prototype. This provides an alternative to full-system simulation. Compared to the full-system simulation, one advantage of prototyping is that native time execution speed can be achieved, which is four to five orders of magnitude faster than full-system simulation. This can be especially useful

when the input size is large. Also, ARAPrototyper provides a framework for users to test their ARAs on real silicon with their own applications deployed.

Compared to full-system simulation, the prototyping method still has limitations related to when we want to observe or evaluate fine-grained microarchitecture parameters. In order to probe for more detailed information, ARAPrototyper provides several built-in hardware counters for users to obtain more insights for further improvement. One piece of future work is to provide more hardware counters for users to help them analyze and improve their ARAs.

In Chapter 6 we provide an optimal crossbar synthesis algorithm to minimize the crossbar complexity. Also, we introduce the interleaved network for better off-chip bandwidth utilization. A possible future direction is to consider the co-optimization of both the crossbar layer and the interleaved network.

3. **Datacenter-level optimization in genomics domain**

   Research concerning the customization and optimization in a domain-specific datacenter or cluster is still in the very beginning stage. In this dissertation we provide several customized optimizations in an in-memory cluster computing system and seek both the algorithm-level optimizations and runtime system support for FPGA-enabled datacenters. Here are several lessons we learned. We also discuss several on-going projects we are working on and possible future directions.

   (a) *Amdahl's Law — the fundamental limiting factor of acceleration*

   We learn that the overall speedup of an application can be limited by the time spent on the code sections that cannot be accelerated, i.e., the code sections that need to be computed by general-purpose cores. We

successfully show that customized accelerators can be much more powerful than general-purpose cores. For example, our customized Smith-Waterman accelerators [35] can be 340x faster than a state-of-the-art Xeon processor running under a single thread. However, based on our profiling results, the runtime spent on the Smith-Waterman kernel in CS-BWAMEM only accounts for 28%. Even if we can reduce the runtime of the given kernel to zero, there is still 72% of the remaining part to be accelerated.

To achieve more speedup in an application, we need to identify more computation kernels to be accelerated. We identified the seeding kernel that uses the SMEM algorithm [110], as demonstrated in Figure 7.2. The SMEM algorithm accounts for about 30% runtime of BWA-MEM and CS-BWAMEM. With both seeding and seed extension steps accelerated, we are able to achieve more than 2x speedup in the alignment stage (CS-BWAMEM).

(b) *Data transfer overhead from JVM to the FPGA device memory*

In Chapter 8 we demonstrated that the data transfer from Spark workers running on JVM to the FPGA device memory can generate significant overhead. With this overhead, we can only accelerate the Smith-Waterman kernel by 2 to 3x instead of a 24x S-W kernel speedup over 24 CPU threads measured in [35].

We provide both (1) algorithm-level improvement through batch processing, and (2) accelerator manager support for FPGA-enabled clusters. However, we believe more optimizations can be provided in algorithm-level to map the algorithm to the FPGA accelerator in a more efficient way. Furthermore, we only provide a primitive design of the accelerator manager in this dissertation in order to support FPGA-enabled clusters. However, a

generalized and optimized accelerator manger needs to be provided. This node-level manager can take the requests from multiple applications and manage more than one accelerator. The optimization on both task scheduling and the data transfer reduction are two important optimization goals.

(c) *Acceleration of the whole DNA sequencing pipeline*

We discussed our current progress in accelerating the DNA sequencing pipeline in Section 7.16. We showed that more than 40x speedup can be achieved in the first three steps. However, we have to accelerate all of the five steps in the pipeline. Otherwise, the unaccelerated steps will become the bottlenecks, e.g., the *Indel Realignment* and *Base Recalibration* steps. This will be an important future work that needs to be addressed for the DNA sequencing domain.

We can take a methodology similar to what we did when developing CS-BWAMEM. First, we need to have a scale-out version of software in both *Indel Realignment* and *Base Recalibration*. Next, we also profile the application and then extract and design accelerator kernels. In Figure 9.1 we demonstrate the identified accelerator kernels and the current progress we have achieved.

Finally, in addition to accelerating the DNA sequencing pipeline, variant calling [129] is an important next-coming stage to be accelerated. Based on our profiling results, the runtime of the variant calling stage is from 62 to 203 hours, which is comparable to the current runtime of the DNA sequencing pipeline.

Conclusions

The journey is just beginning. Currently, the research of accelerator-rich architec-

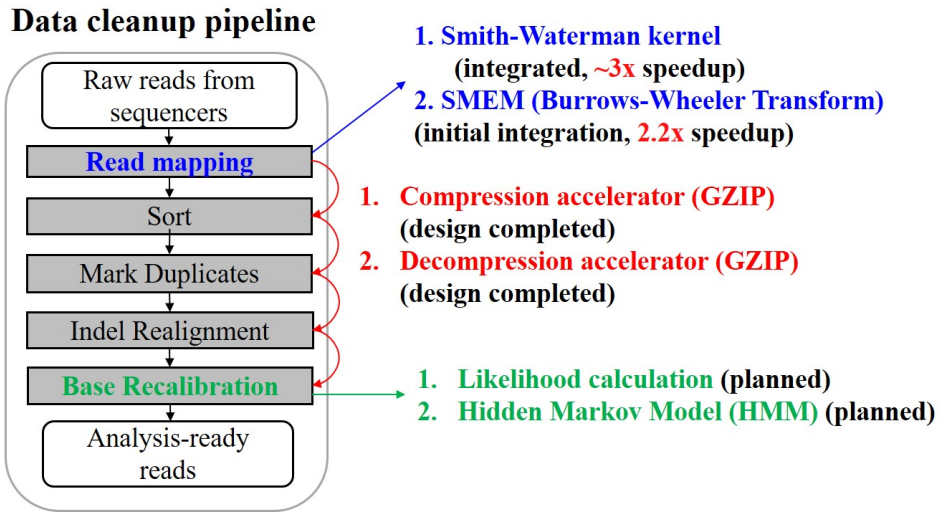


Figure 9.1: Identified accelerator kernels in DNA sequencing pipeline.

tures and FPGA-enabled datacenters is burgeoning. This is because energy-efficient architectures have become more and more important—from hand-held devices to datacenters. However, the research on such customized techniques is still far from mature. This dissertation launches an exploration into the optimizations of several key components: (1) multilevel caches, (2) ARA memory systems, (3) customized optimizations of the in-memory clusters, and (4) FPGA-enabled datacenters. It is our hope that the proposed methodologies—including their limitations, of course—will inspire more innovations in future accelerator-rich architectures and FPGA-enabled in-memory clusters.

References

[1] Alpha Data FPGA accelerator boards. http://www.alpha-data.com/dcp/.

[2] Apache Avro. http://avro.apache.org/.

[3] Apache Hadoop. https://hadoop.apache.org/.

[4] Apache Parquet. https://parquet.incubator.apache.org/.

[5] BitTornado. http://www.bittornado.com/.

[6] NVIDIA Kepler GK110 Architecture Whitepaper. http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf.

[7] Picard. http://broadinstitute.github.io/picard/.

[8] SDAccel Development Environment. http://www.xilinx.com/products/design-tools/sdx/sdaccel.html.

[9] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi (Whitepaper)*, 2009.

[10] A 32 nm, 3.1 billion transistor, 12 wide issue Itanium Processor for Mission-Critical Servers, 2011.

[11] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proc. VLDB Endow.*, 2(2):1664–1665, August 2009.

[12] D. H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *Proc. MICRO*, pages 248–259, 1999.

[13] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403 – 410, 1990.

[14] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, Oct 1990.

[15] StephenF. Altschul and BruceW. Erickson. Optimal sequence alignment using affine gap costs. *Bulletin of Mathematical Biology*, 48(5-6):603–616, 1986.

[16] S. Aluru and N. Jammula. A review of hardware acceleration for computational genomics. *Design Test, IEEE*, 31(1):19–30, Feb 2014.

[17] J. Arram, K.H. Tsoi, Wayne Luk, and P. Jiang. Hardware acceleration of genetic sequence alignment. In Philip Brisk, JosGabriel de Figueiredo Coutinho, and PedroC. Diniz, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 7806 of *Lecture Notes in Computer Science*, pages 13–24. Springer Berlin Heidelberg, 2013.

[18] James Balfour and William J. Dally. Design tradeoffs for tiled cmp on-chip networks. In *ICS*, pages 187–198, 2006.

[19] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proc. CODES*, pages 73–78, 2002.

[20] Jesse Benson, Ryan Cofell, Chris Frericks, Chen-Han Ho, Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. Design, integration and implementation of the dyser hardware accelerator into opensparc. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[21] E. Berg and E. Hagersten. Statcache: A probabilistic approach to efficient and accurate data locality analysis. In *Proc. ISPASS*, pages 20–27, 2004.

[22] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. PACT*, pages 72–81, 2008.

[23] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[24] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. ISCA*, pages 83–94, 2000.

[25] A. Bui, Kwang-Ting Cheng, J. Cong, L. Vese, Yi-Chu Wang, Bo Yuan, and Yi Zou. Platform characterization for domain-specific computing. In *Design*

Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific, pages 94–99, Jan 2012.

[26] M. Burrows, D. J. Wheeler, M. Burrows, and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[27] A Canis, Jongsok Choi, B. Fort, Ruolong Lian, Qijing Huang, N. Calagar, M. Gort, Jia Jun Qin, M. Aldham, T. Czajkowski, S. Brown, and J. Anderson. From software to accelerators with legup high-level synthesis. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*, CASES '13, pages 1–9, Sept 2013.

[28] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: High-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36, 2011.

[29] J. Chang, M. Huang, J. Shoemaker, J. Benoit, S.-L. Chen, W. Chen, S. Chiu, R. Ganesan, G. Leong, V. Lukka, S. Rusu, and D. Srivastava. The 65-nm 16-MB Shared On-Die L3 Cache for the Dual-Core Intel Xeon Processor 7100 Series. In *JSSC*, pages 846–852, 2007.

[30] Jianwen Chen, Jason Cong, Ming Yan, and Yi Zou. Fpga-accelerated 3d reconstruction using compressive sensing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 163–166, New York, NY, USA, 2012. ACM.

[31] Tung-Chien Chen, Shao-Yi Chien, Yu-Wen Huang, Chen-Han Tsai, Ching-Yeh Chen, To-Wei Chen, and Liang-Gee Chen. Analysis and architecture design of an hdtv720p 30 frames/s h.264/avc encoder. *IEEE Trans. Cir. and Sys. for Video Technol.*, 16(6):673–688, September 2006.

[32] Yiran Chen, Weng-Fai Wong, Hai Li, and Cheng-Kok Koh. Processor caches built using multi-level spin-transfer torque ram cells. In *Proc. ISLPED 2011*, pages 73–78, 2011.

[33] Yu-Ting Chen and J. Cong. Interconnect synthesis of heterogeneous accelerators in a shared memory architecture. In *Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on*, pages 359–364, July 2015.

[34] Yu-Ting Chen, J. Cong, M.A. Ghodrat, M. Huang, Chunyue Liu, Bingjun Xiao, and Yi Zou. Accelerator-rich cmps: From concept to real hardware. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 169–176, Oct 2013.

[35] Yu-Ting Chen, J. Cong, Jie Lei, and Peng Wei. A novel high-throughput acceleration engine for read alignment. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 199–202, May 2015.

[36] Yu-Ting Chen, Jason Cong, Zhenman Fang, and Peipei Zhou. ARAPrototyper: Enabling rapid prototyping and evaluation for accelerator-rich architecture. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '16, 2016.

[37] Yu-Ting Chen, Jason Cong, Hui Huang, Bin Liu, Chunyue Liu, Miodrag Potkonjak, and Glenn Reinman. Dynamically reconfigurable hybrid cache: An energy-efficient last-level cache design. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 45–50, San Jose, CA, USA, 2012. EDA Consortium.

[38] Yu-Ting Chen, Jason Cong, Hui Huang, Chunyue Liu, Raghu Prabhakar, and Glenn Reinman. Static and dynamic co-optimizations for blocks mapping in hybrid caches. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pages 237–242, New York, NY, USA, 2012. ACM.

[39] Yu-Ting Chen, Jason Cong, Sen Li, Myron Peto, Paul Spellman, Peng Wei, and Peipei Zhou. CS-BWAMEM: A fast and scalable read aligner at the cloud scale for whole genome sequencing. 2015.

[40] Yu-Ting Chen, Jason Cong, and Glenn Reinman. HC-Sim: A fast and exact l1 cache simulator with scratchpad memory co-simulation support. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, pages 295–304, New York, NY, USA, 2011. ACM.

[41] Yu-Ting Chen, Jason Cong, and Bingjun Xiao. ARACompiler: a prototyping flow and evaluation framework for accelerator-rich architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2015)*,, pages 157–158, March 2015.

[42] Hsiang-Yun Cheng, Matt Poremba, Narges Shahidi, Ivan Stalev, Mary Jane Irwin, Mahmut Kandemir, Jack Sampson, and Yuan Xie. Eecache: Exploiting design choices in energy-efficient last-level caches for chip multiprocessors. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ISLPED '14, pages 303–306, New York, NY, USA, 2014. ACM.

[43] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *Proc. DAC*, pages 416–419, 2000.

[44] Mosharaf Chowdhury. Performance and scalability of broadcast in Spark.

[45] Eric S. Chung, John D. Davis, and Jaewon Lee. Linqits: Big data on little clients. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 261–272, New York, NY, USA, 2013. ACM.

[46] Eric S. Chung, James C. Hoe, and Ken Mai. CoRAM: An in-fabric memory architecture for fpga-based computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 97–106, New York, NY, USA, 2011. ACM.

[47] Eric S. Chung, Michael K. Papamichael, Gabriel Weisz, James C. Hoe, and Ken Mai. Prototype and evaluation of the CoRAM memory architecture for fpga-based computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 139–142, New York, NY, USA, 2012. ACM.

[48] Peter J A Cock, Christopher J Fields, Naohisa Goto, Michael L Heuer, and Peter M Rice. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Res*, 38(6):1767–1771, April 2010.

[49] J. Cong, K. Gururaj, H. Huang, C. Liu, G. Reinman, and Y. Zou. An Energy-Efficient Adaptive Hybrid Cache. In *Proc. ISLPED*, pages 67–72, 2011.

[50] J. Cong, H. Huang, C. Liu, and Y. Zou. A Reuse-Aware Prefetching Algorithm for Scratchpad Memory. In *Proc. DAC*, pages 960–965, 2011.

[51] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, April 2011.

[52] Jason Cong, Zhenman Fang, Michael Gill, and Glenn Reinman. PARADE: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration. *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2015)*, pages 380–387, November 2015.

[53] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. Accelerator-rich architectures: Opportunities and progresses. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 180:1–180:6, New York, NY, USA, 2014. ACM.

[54] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Architecture support for accelerator-rich cmps. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 843–849, New York, NY, USA, 2012. ACM.

[55] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. CHARM: A composable heterogeneous accelerator-rich microprocessor. In *ISLPED*, pages 379–384, 2012.

[56] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Chunyue Liu, and Glenn Reinman. BiN: A buffer-in-nuca scheme for accelerator-rich CMPs. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pages 225–230, New York, NY, USA, 2012. ACM.

[57] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. A fully pipelined and dynamically composable architecture of cgra. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 9–16. IEEE, 2014.

[58] Jason Cong, Peng Li, Bingjun Xiao, and Peng Zhang. An optimal microarchitecture for stencil computation acceleration based on non-uniform partitioning of data reuse buffers. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 77:1–77:6, New York, NY, USA, 2014. ACM.

[59] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. Customizable domain-specific computing. *IEEE Des. Test*, 28(2):6–15, March 2011.

[60] Jason Cong and Bingjun Xiao. Optimization of interconnects between accelerators and shared memories in dark silicon. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, pages 630–637, 2013.

[61] R. Das, S. Eachempati, A.K. Mishra, V. Narayanan, and C.R. Das. Design and evaluation of a hierarchical on-chip interconnect for next-generation cmps. In *HPCA*, pages 175–186, 2009.

[62] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

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

[64] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 245–257, New York, NY, USA, 2003. ACM.

[65] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen. Circuit and Microarchitecture Evaluation of 3D Stacking Magnetic RAM (MRAM) as a Universal Memory Replacement. In *Proc. DAC*, pages 554–559, 2008.

[66] J. Edler and M. D. Hill. *Dinero IV Trace-Driven Uniprocessor Cache Simulator*. http://pages.cs.wisc.edu/ markhill/DineroIV/, 1998.

[67] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, 2011.

[68] Michael Farrar. Striped Smith–Waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, January 2007.

[69] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 390–, Washington, DC, USA, 2000. IEEE Computer Society.

[70] W. Fornaciari, D. Sciuto, C. Silvano, and V. Zaccaria. A design framework to efficiently explore energy-delay tradeoffs. In *Proc. CODES*, pages 260–265, 2001.

[71] B. Fort, A. Canis, J. Choi, N. Calagar, Ruolong Lian, S. Hadjis, Yu Ting Chen, M. Hall, B. Syrowik, T. Czajkowski, S. Brown, and J. Anderson. Automating the design of processor/accelerator embedded systems with legup high-level synthesis. In *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, pages 120–129, Aug 2014.

[72] H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson. Introduction to the wire-speed processor and architecture. *IBM J. Res. Dev.*, 54(1):27–37, January 2010.

[73] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[74] A. Ghosh and T. Givargis. Analytical design space exploration of caches for embedded systems. In *Proc. DATE*, pages 650–655, 2003.

[75] S. Ghosh, M. Martonosi, , and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):703–746, 1999.

[76] O. Gotoh. An improved algorithm for matching biological sequences. 162:705–708, 1981.

[77] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):38–51, September 2012.

[78] Stefan Hadjis, Andrew Canis, Jason H. Anderson, Jongsok Choi, Kevin Nam, Stephen Brown, and Tomasz Czajkowski. Impact of FPGA architecture on resource sharing in high-level synthesis. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 111–114, 2012.

[79] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM.

[80] M. S. Haque, A. Janapsatya, and S. Parameswaran. Susesim: A fast simulation strategy to find optimal L1 cache configuration for embedded systems. In *Proc. CODES+ISSS*, pages 295–304, 2009.

[81] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[82] http://code.google.com/p/google-sparsehash/. *Google Sparsehash*.

[83] http://llvm.org/. *LLVM Compiler*.

[84] http://www.cs.umd.edu/projects/omega/. *Omega Library*.

[85] http://www.hpl.hp.com/research/cacti/. *CACTI 6.5*.

[86] http://www.itk.org/ItkSoftwareGuide.pdf. *ITK Software Guide*.

[87] http://www.itrs.net/. *International Technology Roadmap for Semiconductors*, 2011.

[88] http://www.spec.org/cpu2006. *SPEC Benchmark*, 2006.

[89] http://www.xilinx.com/products/silicon-devices/soc/zynq-7000. *Xilinx Zynq Platform*.

[90] Jingtong Hu, Chun Jason Xue, Wei-Che Tseng, Yi He, Meikang Qiu, and Edwin H.-M. Sha. Reducing write activities on non-volatile memories in embedded cmps via data migration and recomputation. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 350–355, New York, NY, USA, 2010. ACM.

[91] Weichun Huang, Leping Li, Jason R. Myers, , and Gabor T. Marth. ART: a next-generation sequencing read simulator. *Bioinformatics*, 28(4):593–594, Feb 2012.

[92] Amin Jadidi, Mohammad Arjomand, and Hamid Sarbazi-Azad. High-endurance and performance-efficient design of hybrid cache architectures through adaptive line replacement. In *Proc. ISLPED*, pages 79–84, 2011.

[93] A. Janapsatya, A. Ignjatović, and S. Parameswaran. Finding optimal L1 cache configuration for embedded systems. In *Proc. ASPDAC*, pages 796–801, 2006.

[94] X. Jiang, A. Mishra, L. Zhao, R. Iyer, Z. Fang, S. Srinivasan, S. Makineni, P. Brett, and C. R. Das. Access: Smart scheduling for asymmetric cache cmps. In *Proc. HPCA*, 2011.

[95] Minje Jun, Deumji Woo, and Eui-Young Chung. Partial connection-aware topology synthesis for on-chip cascaded crossbar network. *IEEE Trans. Comput.*, 61(1):73–86, January 2012.

[96] Minje Jun, Sungjoo Yoo, and Eui-Young Chung. Topology synthesis of cascaded crossbar switches. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(6):926–930, June 2009.

[97] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *Proc. DAC*, pages 628–633, 2002.

[98] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *Proc. ISCA*, pages 240–251, 2001.

[99] W. James Kent. BLAT the BLAST-like alignment tool. *Genome Research*, 12(4):656–664, Apr 2002.

[100] Maria Kim. Accelerating next generation genome reassembly: Alignment using dynamic programming algorithms in fpgas. *University of Washington, Dept. of EE, MS Thesis*, 2011.

[101] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. In *Proc. SIGMETRICS*, pages 212–213, 1991.

[102] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 468–479, New York, NY, USA, 2013. ACM.

[103] Ian Kuon and Jonathan Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, February 2007.

[104] N. Kurd, M. Chowdhury, E. Burton, T.P. Thomas, C. Mozak, B. Boswell, M. Lal, A. Deval, J. Douglas, M. Elassal, A. Nalamalpu, T.M. Wilson, M. Merten, S. Chennupaty, W. Gomes, and R. Kumar. 5.9 Haswell: A family of ia 22nm processors. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 112–113, Feb 2014.

[105] Ben Langmead and Steven Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, pages 357–359, 2012.

[106] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[107] B.C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *Proc. ISCA*, pages 2–13, 2009.

[108] H. Lee, J.L. Deignan, N. Dorrani, S.P. Strom, S. Kantarci, F. Quintero-Rivera, K. Das, T. Toy, B. Harry, M. Yourshaw, M. Fox, B.L. Fogel, J.A. Martinez-Agosto, D.A. Wong, V.Y. Chang, P.B. Shieh, C.G. Palmer, K.M. Dipple, W.W. Grody, E. Vilain, and S.F. Nelson. Clinical exome sequencing for genetic identification of rare mendelian disorders. *The Journal of the American Medicine Association (JAMA)*, 312(18), November 2014.

[109] Guy Lemieux, Paul Leventis, and David Lewis. Generating highly-routable sparse crossbars for plds. In *FPGA*, pages 155–164, 2000.

[110] Heng Li. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, July 2012.

[111] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *Preprint at arXiv:1303.3997v2 [q-bio.GN]*, 2013.

[112] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, July 2009.

[113] Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, March 2010.

[114] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, August 2009.

[115] Jianhua Li, Liang Shi, Qingan Li, Chun Jason Xue, Yiran Chen, Yinlong Xu, and Wei Wang. Low-energy volatile STT-RAM cache design using cache-coherence-enabled adaptive refresh. *ACM Trans. Des. Autom. Electron. Syst.*, 19(1):5:1–5:23, December 2013.

[116] Jianhua Li, C.J. Xue, and Yinlong Xu. STT-RAM based energy-efficiency hybrid cache for cmps. In *VLSI and System-on-Chip (VLSI-SoC), 2011 IEEE/IFIP 19th International Conference on*, pages 31–36, Oct 2011.

[117] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2. *Bioinformatics*, 25(15):1966–1967, August 2009.

[118] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin servers with smart pipes: Designing soc accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 36–47, New York, NY, USA, 2013. ACM.

[119] Tiantian Liu, Yingchao Zhao, C.J. Xue, and Minming Li. Power-aware variable partitioning for dsps with hybrid PRAM and DRAM main memory. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 405–410, June 2011.

[120] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. CUDASW++ 3.0: accelerating smith-waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 14:117, 2013.

[121] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI*, pages 190–200, 2005.

[122] Michael J. Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. The accelerator store: A shared memory framework for accelerator-based systems. *ACM Trans. Archit. Code Optim.*, 8(4):48:1–48:22, January 2012.

[123] A. Madhavan, T. Sherwood, and D. Strukov. Race logic: A hardware acceleration for dynamic programming algorithms. In *ISCA*, pages 517–528, June 2014.

[124] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. In *IEEE Computer*, pages 50–58, 2002.

[125] Svetlin Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(S-2), 2008.

[126] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[127] Matt Massie, Frank Nothaft, Christopher Hartl, Christos Kozanitis, Andre Schumacher, Anthony D. Joseph, and David A. Patterson. Adam: Genomics formats and processing patterns for cloud scale computing. Technical Report

UCB/EECS-2013-207, EECS Department, University of California, Berkeley, Dec 2013.

[128] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[129] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark A. DePristo. The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome Res*, 20(9):1297–303, 9 2010.

[130] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(1-2):330–339, September 2010.

[131] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical report, HP Laboratories, 2009.

[132] S. Nakamura and G. M. Masson. Lower bounds on crosspoints in concentrators. *IEEE Trans. Comput.*, 31(12):1173–1179, December 1982.

[133] Zemin Ning, Anthony J. Cox, and James C. Mullikin. Ssaha: A fast search method for large dna databases. *Genome Research*, 11(10):1725–1729, Oct 2001.

[134] C.B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W.L. Ruzzo. Hardware acceleration of short read mapping. In *FCCM*, pages 161–168, 2012.

[135] OProfile. A system profiler for linux.

[136] J. J. Pieper, A. Mellan, J. M. Paul, D. E. Thomas, and F. Karim. High level cache simulation for heterogeneous multiprocessors. In *Proc. DAC*, pages 287–292, 2004.

[137] M. Powell, S. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *Proc. ISLPED*, pages 90–95, 2000.

[138] T.B. Preusser, O. Knodel, and R.G. Spallek. Short-read mapping by a systolic custom fpga computation. In *FCCM*, pages 169–176, 2012.

[139] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. Convolution engine: Balancing efficiency and flexibility in specialized computing. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 24–35, New York, NY, USA, 2013. ACM.

[140] M.K. Qureshi, M.M. Franceschini, L. A. Lastras-Montaño, and J.P. Karidis. Morphable Memory System: A Robust Architecture for Exploiting Multi-Level Phase Change Memories. In *Proc. ISCA*, pages 153–162, 2010.

[141] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *Proc. ISCA*, pages 214–224, 2000.

[142] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable Caches and their Application to Media Processing. In *Proc. ISCA*, pages 214–224, 2000.

[143] M. Rasquinha, D. Choudhary, S. Chatterjee, S. Mukhopadhyay, and S. Yalamanchili. An Energy Efficient Cache Design Using Spin Torque Transfer (STT) RAM. In *Proc. ISLPED*, pages 389–394, 2010.

[144] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks.

[145] Torbjorn Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation. *BMC Bioinformatics*, 12:221, 2011.

[146] Torbjørn Rognes and Erling Seeberg. Six-fold speed-up of smith-waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, 2000.

[147] Richard Sampson, Ming Yang, Siyuan Wei, Chaitali Chakrabarti, and Thomas F. Wenisch. Sonic Millip3De: A massively parallel 3D-stacked accelerator for 3D ultrasound. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 318–329, Washington, DC, USA, 2013. IEEE Computer Society.

[148] Daniel Sanchez, George Michelogiannakis, and Christos Kozyrakis. An analysis of on-chip interconnection networks for large-scale chip multiprocessors. *ACM Trans. Archit. Code Optim.*, 7(1):4:1–4:28, May 2010.

[149] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A Pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceeding of the 41st*

Annual International Symposium on Computer Architecuture, ISCA '14, pages 97–108, 2014.

[150] K. Shi and D. Howard. Challenges in Sleep Transistor Design and Implementation in Low-Power Designs. In *Proc. DAC*, pages 113–116, 2006.

[151] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[152] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[153] R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Transactions on Computer Systems (TOCS)*, 13(1):32–56, 1995.

[154] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen. A Novel Architecture of the 3D Stacked MRAM L2 Cache for CMPs. In *Proc. HPCA*, pages 239–249, 2009.

[155] The SAM/BAM Format Specification Working Group. Sequence Alignment/Map Format Specification, 2014.

[156] N. Tojo, N. Togawa, M. Yanagisawa, and T. Ohtsuki. Exact and fast l1 cache simulation for embedded systems. In *Proc. ASPDAC*, pages 817–822, 2009.

[157] Core Trapnell, Adam Roberts, Loyal Goff, Geo Pertea, Daehwan Kim, David R. Kelley, Harold Pimentel, Steven L. Salzberg, Rinn John L., and Lior Pachter. Differential gene and transcript expression analysis of RNA-seq experiments with TopHat and Cufflinks. *Nature Protocols*, 7:562–578, 2012.

[158] Core Trapnell, Brian A. Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J. van Baren, Steven L. Salzberg, Barbara J. Wold, and Lior Pachter. Transcript assembly and quantification by RNA-Seq reveals unannotated transcripts and isoform switching during cell differentiation. *Natural Biotechnology*, 28(5):511–515, May 2010.

[159] K. Tsuchida and et al. A 64Mb MRAM with Clamped-Reference and Adequate-Reference Schemes. In *Proc. ISSCC*, pages 258–259, 2010.

[160] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford

Taylor. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 205–218, 2010.

[161] X. Vera, N. Bermudo, J. Llosa, and A. González. A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(2):263–300, 2004.

[162] Zhe Wang, D.A Jimenez, Cong Xu, Guangyu Sun, and Yuan Xie. Adaptive placement and migration policy for an STT-RAM-based hybrid cache. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 13–24, Feb 2014.

[163] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[164] A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer Applications in the Biosciences*, 13(2):145–150, 1997.

[165] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie. Hybrid Cache Architecture with Disparate Memory Technologies. In *Proc. ISCA*, pages 34–45, 2009.

[166] Xilinx. Sdsoc development environment. http://www.xilinx.com/products/design-tools/software-zone/sdsoc.html.

[167] Xilinx. Ultrascale zynq. http://www.xilinx.com/products/technology/ultrascale-mpsoc.html.

[168] Xilinx. Vivado high level synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/index.htm.

[169] Cong Xu, Xiangyu Dong, N.P. Jouppi, and Yuan Xie. Design implications of memristor-based rram cross-point structures. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[170] C.W. Yu, K.H. Kwong, K.H. Lee, and P.H.W. Leong. A smith-waterman systolic cell. In Patrick Lysaght and Wolfgang Rosenstiel, editors, *New Algorithms, Architectures and Applications for Reconfigurable Computing*, pages 291–300. Springer US, 2005.

[171] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[172] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[173] M. Zhang and K. Asanovic. Fine-grain CAM-tag cache resizing using miss tags. In *Proc. ISLPED*, pages 130–135, 2002.

[174] Peiheng Zhang, Guangming Tan, and Guang R. Gao. Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform. In *Proceedings of the 1st International Workshop on High-performance Reconfigurable Computing Technology and Applications: Held in Conjunction with SC07*, pages 39–48, 2007.

[175] Huiyang Zhou, Mark C. Toburen, Eric Rotenberg, and Thomas M. Conte. Adaptive mode control: A static-power-efficient cache design. *ACM Trans. Embed. Comput. Syst.*, pages 347–372.