University of California
Los Angeles

Resource and Data Management in Accelerator-Rich Architectures

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

by

Muhuan Huang

2016

Abstract of the Dissertation

Resource and Data Management in Accelerator-Rich Architectures

by

Muhuan Huang

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2016

Professor Jingsheng Jason Cong, Chair

In many domains, accelerators—such as graphic processing units (GPUs) and field programmable gate arrays (FPGAs)—provide a significantly higher performance than general-purpose processors and at a much lower power. Accelerator-rich architectures are thus much more energy-efficient and are becoming mainstream.

This dissertation investigates two important keys to the performance and power efficiency of accelerator-rich architectures—resource and data management. Three broad classes of accelerator-rich architectures are considered: chip-level accelerator-rich architectures such as systems-on-chips(SoC), node-level accelerator-rich architectures, and cluster-level accelerator-rich architectures.

We first study SoC resource management for a broader class of streaming applications. On accelerator-rich SoCs, where multiple computation kernels space-share a single chip, we target the exploration of tradeoffs of on-chip resources and system performance, and find the best combination of accelerator implementations and data communication channel implementations to realize the application functionality.

We continue our study of node-level accelerator-rich architectures where we consider orchestrating two kinds of computation resources, CPU and accelerator, in the PCIe-integrated CPU-accelerator platform and explore the CPU-FPGA collaboration approach to improve application performance.

Then we study the resource allocation problem on accelerator-rich clusters, where accelerators are time-shared among multiple tenants. Unlike traditional cluster resource management, we propose to consider accelerators as the first-class citizen in the cluster

resource pool, and develop an accelerator-centric resource scheduling policy to enable fine-grained accelerator sharing among multiple tenants.

Finally, we investigate data shuffling on accelerator-rich clusters and evaluate the possibility of using accelerators during data shuffling. We find that although data shuffling involves a large amount of computation, using accelerators does not necessarily improve system performance due to the data serialization and deserialization overhead introduced by accelerators.

The dissertation of Muhuan Huang is approved.

Wotao Yin

Wei Wang

Tyson Condie

Jingsheng Jason Cong, Committee Chair

University of California, Los Angeles

2016

To my parents

Table of Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I wish to express my deepest gratitude to my advisor, Professor Jason Cong, for his continuous support and guidance throughout my education. I first met Jason during an undergraduate exchange program at UCLA in my junior year. Jason guided me into the area of FPGA acceleration and has sparked my desire to pursue research in the field of customized computing. Throughout my PhD study, Jason has provided many insightful suggestions for my research projects. He also generously provided various amazing opportunities towards conducting better research and producing better industry-level products. It is my great fortune to have him as my advisor.

I wish to express my appreciation to my doctoral committee members, Professor Tyson Condie, Professor Wei Wang, and Professor Wotao Yin, for their time, interest and advice to improve the quality of this dissertation. I especially wish to thank Tyson for his guidance on the runtime of large-scale distributed data processing system.

I also wish to thank all my collaborators at UCLA, especially Zhenman Fang, Karthik Gururaj, Hui Huang, Matteo Interlandi, Sen Li, Di Wu, Bingjun Xiao, Peng Zhang and Yi Zou. They helped to shape my ideas and research. It has been a true blessing to work with these brilliant people.

In addition, I wish to thank my friends at UCLA VAST lab for supporting me throughout my lengthy graduate school and making the past six years much more enjoyable. I wish to thank the staffs in the computer science department for all their help during my study here. I especially wish to thank Alexandra Luong for taking care of my appointments every month, and Janice Martin-Wheeler for proofreading all my paper submissions.

Beyond UCLA, I wish to thank the fellow researchers at HP Labs for the most rewarding internship experience they have given me. I have been privileged to work with many amazing people there, which has certainly brought out the best in me. I especially wish to thank Kevin Lim who helped mentor me during my times at HPL.

Last but certainly not least, I wish to express my deep appreciation for my parents, Yan and Feng. They have supported me all throughout my life, allowing me to seek my own path, providing me security and guiding me in the moments of confusion. I would

not be where I am today without them.

This work is partially supported by the Center for Domain Specic Computing under the NSF InTrans Award CCF1436827, funding from CDSC industrial partners including Baidu, Fujitsu Labs, Google, Huawei, Intel, IBM Research Almaden, and Mentor Graphics; C-FAR, one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA; grants NSF IIS-1302698 and CNS-1351047; and U54EB020404 awarded by NIH Big Data to Knowledge (BD2K).

Vita

2010	B.S. Electronics & Information Engineering, Xi'an Jiaotong University, China
2010 - 2016	Ph.D. Student, Department of Computer Science, University of California, Los Angeles

Publications

Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong, "Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale", *Symposium on Cloud Computing (SOCC)*, 2016.

Jason Cong, Muhuan Huang, Peichen Pan, Di Wu, and Peng Zhang, "Software Infrastructure for Enabling FPGA-Based Accelerator in Data Centers: Invited Paper", *International Symposium on Low Power Electronics and Design (ISLPED)*, 2016.

Jason Cong, Muhuan Huang, Di Wu, and Cody Hao Yu, "Invited-Heterogeneous Datacenters: Options and Opportunities", *Design Automation Conference (DAC)*, 2016.

Peng Zhang, Muhuan Huang, Bingjun Xiao, Hui Huang, and Jason Cong, "CMOST: a System-Level FPGA Compilation Framework", *Design Automation Conference (DAC)*, 2015.

Muhuan Huang, Kevin Lim, and Jason Cong, "A Scalable High-Performance Customized Priority Queue", *International Conference on Field Programmable Logic and Applications (FPL)*, 2014.

Jason Cong, Muhuan Huang, and Peng Zhang, "Combining Computation and Communication Optimizations in System Synthesis for Streaming Applications", *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2014.

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.

Jason Cong, Milos Ercegovac, Muhuan Huang, Sen Li, and Bingjun Xiao, "Energy-Efficient Computing using Adaptive Table Lookup based on Nonvolatile memories", *International Symposium on Low Power Electronics and Design (ISLPED)*, 2013.

Jason Cong, Muhuan Huang, Bin Liu, Peng Zhang, and Yi Zou, "Combining Module Selection and Replication for Throughput-Driven Streaming Programs", *Conference on Design, Automation and Test in Europe (DATE)*, 2012.

Jason Cong, Karthik Gururaj, Muhuan Huang, Sen Li, Bingjun Xiao, and Yi Zou, "Domain-Specific Processor with 3D Integration for Medical Image Processing", *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2011.

Jason Cong, Muhuan Huang, and Yi Zou, "Accelerating Fluid Registration Algorithm on Multi-FPGA Platforms", *International Conference on Field Programmable Logic and Applications (FPL)*, 2011.

Jason Cong, Muhuan Huang, and Yi Zou, "3D Recursive Gaussian IIR on GPU and FPGAs — A Case for Accelerating Bandwidth-bounded Applications", *Symposium on Application Specific Processors (SASP)*, 2011.

CHAPTER 1

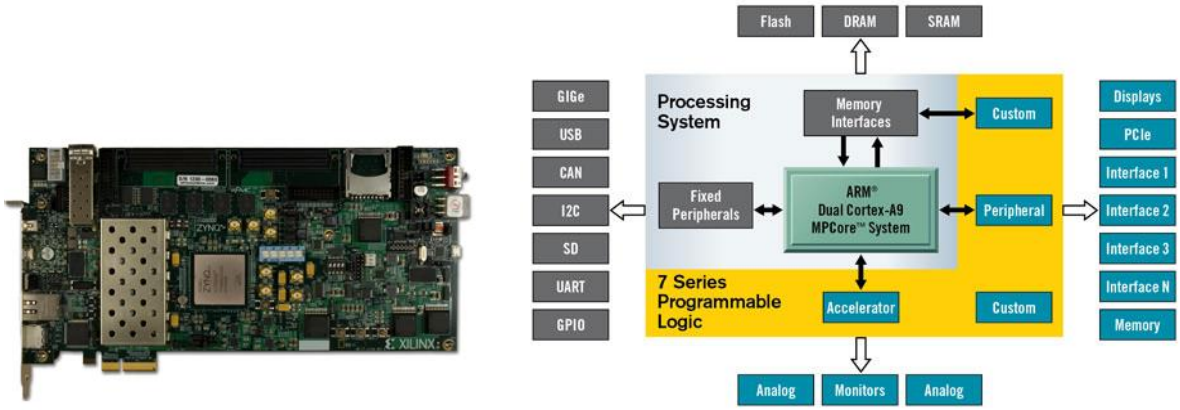
Introduction

The last decade has witnessed tremendous changes in processor and platform designs, shifting from higher frequency processors to multi-core processors, and more recently shifting in favor of accelerator-rich architectures. As pointed out in [CSR11, CGG12], there often exists a significantly large performance gap between an entirely customized solution and a general-purpose one. The difference in the case study of the 128-bit key AES encryption algorithm implies a performance/energy efficiency gap of roughly 3 million. In their study, an investigation of the energy breakdown of the CPU pipeline components reveals that the majority of the energy consumption (i.e., 64%) is attributable to supporting the flexible instruction-oriented model of the general-purpose core, and not for performing actual computations.

Accelerator-rich architectures have two major advantages over traditional general-purpose processing architectures. On one hand, accelerators such as GPUs and FPGAs are massively parallel architectures and thus can provide significant higher processing functionality than CPUs; they can help to meet and even exceed our ever-increasing processing needs. On the other hand, since energy consumption is becoming one of the major limiting factors for scaling up the processing capability of computation platforms (e.g., power-constrained or battery-operated systems-on-chips) and data centers, designers are looking beyond general-purpose processors to leverage accelerators that increase the performance per-watt metric of the applications and reduce the energy consumption of the system.

As a result, accelerators have found their home in many computing architectures.

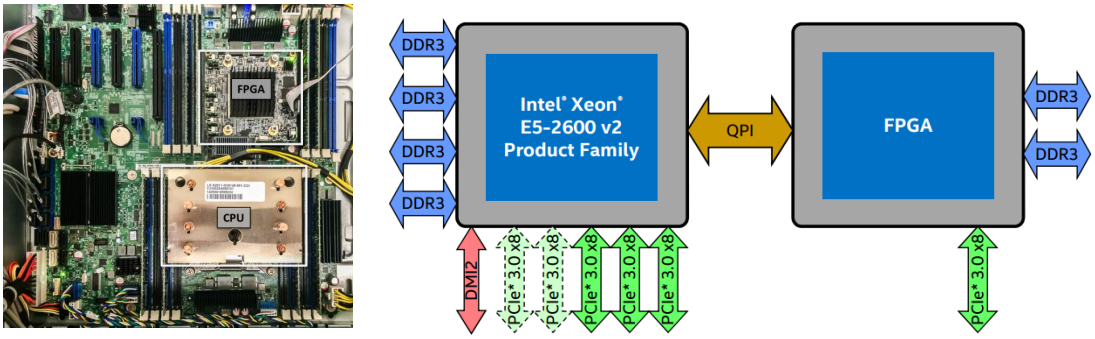
Single-Chip Level Designs: For example, system-on-chip (SoC) FPGAs and SoC GPUs are widely available on the market. Major FPGA vendors, such as Xilinx [xil] and Altera [alt], provide SoC FPGAs which integrate an ARM-based processing system



(a) Zynq board.

(b) Zynq architecture [zyn].

Figure 1.1: Xilinx Zynq-7000 EPP dual-core ARM Cortex-A9 + FPGA.



(a) Xeon + FPGA board.

(b) Xeon + FPGA architecture [int].

Figure 1.2: Intel Xeon + FPGA platform.

and the on-chip programmable logics on a same chip. Figure 1.1 shows an example of the Xilinx SoC that contains a dual-core ARM Coretx-A9 and an FPGA [zyn].

Server Node Level Platforms: Server vendors have been looking beyond general-purpose servers and aim to provide an easy-to-customize server node to boost the performace and power efficiency. For example, since early 2013, HP has been building a line of high-efficiency servers called Moonshot. The servers can be customized to embed GPUs, DSPs or FPGAs, and thus can provide the most resourceful systems with a minimum of space. More recently, Intel has built an Xeon + FPGA platform that links CPU and FPGA together through Quick Path Interconnect (QPI) so that they can share memory. Figure 1.2 demonstrates the Xeon + FPGA platform. A quantitative comparison between the QPI-based CPU-FPGA system and PCIe-based CPU-FPGA system can be

Figure 1.3: Our local cluster with FPGA cards.

found in [CCF16].

Datacenter Level Solutions: Cloud computing vendors (e.g., Amazon Web Service) have incorporated GPUs in their platforms. FPGAs are also becoming popular in the cloud environment. Recently the Bing search engine from Microsoft [PCC14] demonstrated 2X better system throughput when its 1632-node servers are equipped with FPGAs. At UCLA, we also built a local cluster with FPGA boards as shown in Figure 1.3. The cluster has 20 CPU nodes and 8 of them have PCIe integrated FPGA boards. Considering that there is one FPGA board per server, it is not a trivial task to manage all the FPGAs in the system.

Such accelerator-rich architectures, with their recent increased system heterogeneity, bring new challenges in accelerator and data management. These challenges are detailed in the following sections.

1.1 Chip-Level Resource and Data Management Challenges

A large percentage of real-world applications can be classified as streaming applications, where the applications are centered around the notion of *streams* of data. Streaming applications are becoming increasingly important and widespread. Due to the prevalence of streaming applications, there are extensive studies on system modeling and design techniques. Actor-based streaming modeling is developed to capture the intrinsic features of these applications, such as synchronous data flow models [LM87a]. Each actor in such a modeling refers to the computation that is to be performed on the incoming streams of data. Recent studies also provide language support for streaming applications, such as StreamIt [TKA02], Brook [BFH04] and Cg [MGA03]. In this modeling an application is specified as a set of actors connected by communication channels. The parallelism of an application is implicitly specified within the graph model, and can be derived from the data dependencies between the actors, *i.e.*, actors that do not have data dependencies among them can work in parallel. Throughput is an important criterion for evaluating performance of streaming applications. We can implement efficient FIFO-based dedicated and low-cost communication channels on FPGAs to support pipelining and thus improve the system throughput. In addition, we can also design customized computation datapaths for the actors.

How to best utilize the parallelism and regularity of streaming application at design time has recently been a hot research topic. The first common approach to increase the system performance is replicating the actors to exploit the data parallelism. A stateless actor, whose output is fully determined by its input, offers opportunities for data parallelism — replicas of a stateless actor can be executed in parallel. This strategy has been well studied on the multi-core platforms [GTA06, GTK02, KM08, HCK09, LDW06, UGT09, ZBS13]. The general idea is to first perform actor fission and fusion and then allocate the replicas to different processor cores. In the context of FPGA, replication optimization is more complex, because actors have different resource utilizations in terms of look-up tables (LUT) and registers, and they have to share the available on-chip resource. In [HWB09], a heuristic method is proposed which performs maximal replication of all the stateless actors and then iteratively fuses the actors that do not affect the

throughput. In general, the replication technique is an easy way to increase throughput. However by simply replicating the actors, not only are the critical computation datapaths replicated, all the non-critical datapaths and control paths are also replicated, which does not necessarily contribute to improving the system performance. To better utilize the FPGA on-chip resources, one should explore different customized actor implementations with the same functionality, rather than rely on the replication technique only. This is a typical module selection problem, as discussed below.

The second approach to increase the system performance focuses on module selection [ID91, ADC95, ILP98, SWN07, CCX05]. Given different implementations of each actor, one selects the proper implementations to integrate into the system and schedule the actors to meet the system requirement. A more recent work [JHI10] explores different configurations of application specific instruction set processors (ASIPs) under throughput and latency constraints and proposes an ILP-based solution. However, most of the studies mainly focus on module selection while the design space of module replication, which is essentially useful in the case of FPGAs, is not explored.

On the other hand, communication between computation actors is also important for streaming applications in terms of system performance and resource utilization. FIFO is the common communication mechanism used to enable pipelining between actors. FIFO size is one of the key design parameters that needs to be optimized in the system synthesis. It affects the on-chip block RAM (BRAM) utilizations. More importantly, it affects the concurrency of the actor scheduling and thus has a direct impact on system performance. The buffer optimization problem is known to be NP-complete [BML12]. [SGB06] formulates the buffer size constraints directly on the SDF graph. The buffer size requirement on communication channels is imposed on the SDF graph by adding additional backward channels. The number of initial tokens corresponds to the FIFO size. Thus the original application model that does not the consider the FIFO size is transformed into a new model which contains the scheduling constraints imposed by the given FIFO size.

Several approaches have been proposed to optimize the buffer utilization under certain design metrics [GGD02, NG93]. Integer linear programming is used to obtain the minimal buffer size for synchronous data flow(SDF) [GGD02] and homogeneous synchronous data flow (HSDF) [NG93] respectively. But these methods are limited to the maximal

throughput scheduling which is not efficient in exploring the different performance and area trade-off options. Some researchers [GBS05,LGX09] use a model checking technique to efficiently explore the state-space of the SDF graph. But the complexity of these methods is relatively high, especially for the graphs with large amount of state transitions. In addition, they minimize buffer size just for a deadlock-free scheduling without considering actual performance. [SGB06] proposed an efficient way to explore the Pareto-curve of the buffer size and throughput by pruning the design space without losing the Pareto points, where throughput is calculated by an eager simulation of the periodic execution, and buffer size distribution is incrementally explored according to the storage dependencies. A recent work [CZ12] modeled the buffer size directly in the analytic model and proposed an efficient heuristic based on it. But it works only on acyclic SDFs and does not prove optimal or near-optimal in large graphs. In all of this set of work, actor optimizations like module selection and replication are assumed to be a separate preprocessing or post-processing stage. This may lose the optimality in finding the scheduling for both actor and buffer optimization.

Other related work includes [KDV97] where a simulator is adopted to evaluate different dataflow architectures for streaming applications. Our problem is a subset of their's since in our architecture we only consider FIFO interfaces. With our simplified architecture, we are able to quickly decide the selected modules without resorting to a simulator. Outside of streaming context, a computation and communication co-optimization framework is studied in [LCM09]. It combines data-reuse optimization using a scratchpad and loop-level parallelism optimization into a singe framework and formulate it into an integer geometric programming problem. This is a module-level optimization and does not apply to our problem directly.

Learning from the existing work, we observe that system design for streaming applications is still facing two major challenges: (1) System design results are determined by a complicated combination of different design aspects (module selection, module replication, buffer size optimization, scheduling and allocation), but current automation frameworks always take single or limited aspects into considerations. (2) Considering all these comprehensive design objectives and constraints makes the existing algorithms either non-scalable for large and complex designs, or difficult to achieve comparable quality of

results with the optimal solutions.

We will investigate this problem in Chapter 3.

1.2 Node-Level Resource Management Challenges

Accelerators such as FPGAs and GPUs often serve as slave processors on a host CPU node. Typical CPU-acceleration integration methods include a PCIe-based approach and a QPI-based approach. To make use of these slave processors, current solutions require an explicit user program that handles data communication between the host and accelerators, and monitors computation tasks that are offloaded to the accelerators through APIs that are either driver-level APIs or higher-level APIs, such as OpenCL. This is also true in the virtualized environment (such as the virtual machine (VM) or the container environment); the operating system or the hypervisor relies on the user to decide which tasks should be offloaded to the accelerator.

Therefore, it is important to manage both computation resources at runtime to get the best possible performance. When computation tasks are offloaded to the accelerators, the CPU should not be idled, waiting for other tasks to finish. Other computation tasks should be executed on the CPU at the same time in the ideal case. A straightforward integration of CPU and accelerators that simply offloads tasks to accelerators and leaves the CPU idle may achieve only marginal performance gains, despite the high speedup achieved from the accelerator.

1.3 Cluster-Level Resource and Data Management Challenges

At the cluster-level, the current trend in equipping accelerators such as GPUs and FPGAs in commodity data centers also requires new resource and data management strategies.

Accelerators usually only accelerate part of the application's execution, and non-accelerable tasks have to be converted back to the host. Thus, unlike other cluster resources, accelerators are usually underutilized from a tenant's perspective; accelerators should no longer be bundled to a single tenant at a time. Other unique properties of accelerators include: (1) Accelerators are often treated as I/O devices. (2) CPUs can

be treated as a backup resource for accelerators; if accelerators are not available, tasks can be converted back to the host CPU and can take advantage of CPU computation power. (3) Accelerators have predictable performance since they are clean slaves that do not run OS; this feature enables more sophisticated accelerator management based on runtime accelerator status. However, most current cluster resource management—such as YARN [VMD13], Mesos [HKZ11] and Omega [SKA13]—are accelerator-oblivious. They only consider allocating CPU and memory resources. We believe that accelerators should be the first-class citizen in the resource pool, and this requires dedicated management techniques.

Traditional cluster resource management support for CPU and memory-sharing mainly relies on operating system (OS) features. CPU core-sharing is realized by launching multiple computation processes on a core, and it relies on the context switch by OS. Similarly, memory-sharing among multiple processes is managed by OS—when the used memory exceeds the total available memory on the core, memory is swapped out to disks. As current OS does not manage accelerators directly, it is hard to apply these resource management techniques to accelerators.

Concurrently, accelerator-rich architectures offer new opportunities in cluster data management. Data shuffling, which moves the data across all the machines, is the heart of almost all the cluster-scale data processing frameworks, including MapReduce and Spark. "Sorting," which is the major computation-intensive kernel during data shuffling, ought to achieve improved performance and energy-efficiency when being offloaded to accelerators. How to seamlessly incorporate accelerators in the data shuffling stage is yet an open question.

1.4 Dissertation Statement

This dissertation examines and addresses several resource and data management challenges in accelerator-rich architectures.

- At chip-level, we explore the design space of mapping streaming applications onto accelerators. Specifically, we consider FPGAs as the targeted accelerators. A computation and communication co-optimization algorithm is proposed to minimize the

occupied resources while meeting system performance requirements.

- At node-level, we study the runtime to orchestrate task execution on CPU and task execution on accelerators. A dataflow execution model and its corresponding runtime thread allocation policy are proposed.

- At cluster-level, we develop a cluster accelerator manager which is deployed on top of a commodity cluster resource manager. The proposed on-line accelerator allocation allows fine-grained time-sharing of accelerators between multiple tenants and improves cluster throughput.

- Finally, we present our analysis on accelerating data shuffling on accelerator-rich clusters. We demonstrate that although data shuffling is computation-intensive, integrating accelerators into the system does not provide performance gain due to data serialization and deserialization overhead.

1.5 Organization

This dissertation will examine the above-mentioned challenges. The remainder of the dissertation is organized as follows. Chapter 3 presents resource and data management on accelerator-rich systems-on-chips. Chapter 4 presents resource management on accelerator-rich nodes. Chapter 5 presents our resource management in accelerator-rich clusters. Chapter 6 presents our analysis on accelerating data shuffling on accelerator-rich clusters. We conclude and discusses future research opportunities in Chapter 7.

CHAPTER 2

Background

This chapter presents the background for several key topics, including chip-level programming model, cluster-level programming model, node-level execution model, node-level CPU and GPU orchestration, cluster-level resource manager and cluster-level data shuffling.

2.1 Chip-Level Programming Model

In this section we present synchronous data flow graphs and homogeneous synchronous data flow graphs which are commonly used to model streaming applications, and we define system throughput from the graphs.

2.1.1 Synchronous Data Flow Graph

Synchronous data flow graphs (SDFGs) [LM87a] have been traditionally used to model streaming applications. In an SDFG $G(V, E)$, each node $v \in V$ (also called an actor) represents a computation kernel. Each edge $u \rightarrow v$ represents a data communication channel from node u to v. The amount of data tokens consumed and produced by a computation kernel is determined at design time, and is usually referred to as the producer rate and the consumer rate. For example, in Fig. 2.1, the producer rate and consumer rate for actor n_1 is 2 and 1 respectively. When an actor starts its firing/execution, it removes the number of tokens that equals the consumer rate from its input channels. At

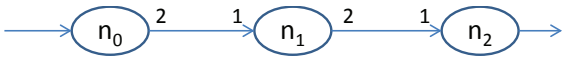


Figure 2.1: An example of SDFGs. Producer/consumer rates are labeled at the beginning/end of the edges.

the end of a firing/execution, the actor produces the number of tokens that equals the producer rate to its output channels. We model the data consumption and production process as an atomic process where the number of tokens on a channel will only be changed at the beginning or at the end of an actor firing. Such a fixed producer rate and consumer rate make it easier to analyze and predict the timing behavior of complex streaming applications, and this is the main reason that SDFGs are used.

The execution of an SDFG is defined in terms of actor firings:

Definition 1 *A **periodic execution schedule** or **iteration** of an SDFG is a set of actor firings such that after all these firings, the SDFG returns to the same state, i.e., the number of tokens on each channel remains the same before or after an iteration.*

The number of actor firings in one periodic execution schedule is called the repetition factor vector q [LM87b]. In Fig. 2.1, actors n_1, n_2 and n_3 fire 1, 2 and 4 times respectively to maintain a data balance between producer and consumer actors. Therefore, the corresponding repetition factor vector q is [1;2;4]. Note that it is possible that the execution of SDFGs results in deadlock or an infinite number of accumulated tokens on the channels. In this case, there does not exist a periodic execution schedule. More detailed analysis on whether there exists a periodic execution schedule given an SDF can be found in [LM87a]. Such a system is called an inconsistent system, and it is beyond the scope of this work.

The *periodic execution schedule* is very important in analyzing system throughput for streaming applications. Streaming applications usually perform computation on a very long data sequence (which is typically assumed to be infinite). It is costly to adopt an irregular scheduling for each of the repeated actor firings. Since SDFG has a periodic behavior in actor firings, it's reasonable to have a periodic scheduling.

Note that in the context of SDF, communication channels are often referred to as buffers. In the context of FPGA implementations, communication channels are often referred to as FIFOs. In this chapter we do not distinguish between these two.

2.1.2 Homogeneous Synchronous Data Flow Graph

An SDFG can be transformed into an equivalent single-rate data flow graph, which is called a homogeneous synchronous data flow graph (HSDFG) [LM87a]. In HSDFG, all

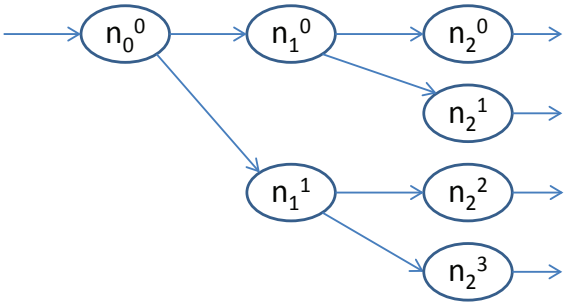


Figure 2.2: The HSDFG converted from SDFG in Fig. 2.1.

the actor firings in one iteration are explicitly enumerated. Fig. 2.2 shows the HSDFG converted from the SDFG in Fig. 2.1. There are two firings of actor n_1 (n_1^0 and n_1^1) and four firings of actor n_2 ($n_2^i, 0 \leq i < 4$). Edges in HSDFG represent data dependency between producers and consumers. For example, n_1^0 and n_1^1 can start to work only after n_0^0 finishes and produces one data to each of them, so there are data dependency edges between them.

We find it easier to start from the HSDFG (instead of the SDFG) when modeling scheduling constraints since all the actor firings have been enumerated in the graph. Thus, given an application that is modeled in an SDFG, we will first transform it into an HSDFG by using the conversion algorithm in [SB00].

2.1.3 Throughput Definition

In this work throughput is viewed as a constraint for streaming applications. Denote n_i^j as the jth firing of actor n_i in the first iteration. We associate each actor firing n_i^j in the scheduling graph with a start firing time t_i^j, which denotes when the firing can start. We also refer to t_i^j as scheduling variables.

Throughput is defined as how often the actor firing n_i^j can start a firing again. Particularly in the scheduling graph, throughput is defined as follows:

$$Throughput = \frac{1}{t'^j_i - t_i^j}, \tag{2.1}$$

where t'^j_i denotes the start firing time of the jth firing of actor t_i in the following iteration.

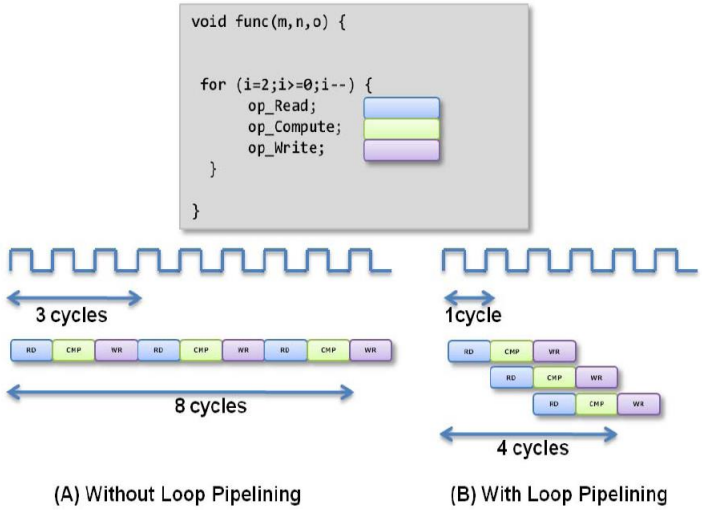


Figure 2.3: Module implementation with and without pipelining. This figure is adopted from [viv].

2.2 Chip-Level Design Space Exploration with High Level Synthesis

2.2.1 Design Space Exploration

The modules in the streaming application typically work on a very large sequence of data. Therefore it is nature to adopt the module replication strategy, where multiple modules of the same functionality work on multiple data in parallel. System designers need to decide the number of replicas of each module.

Besides, system designers may also need to consider the best module implementation. In general, module replication is not an area-efficient design technique for increasing throughput because control logics, which do not directly contribute to throughput, are also duplicated. Moreover, some computing logics might be wasteful after replication. For example, consider a module that contains two add and one multiply operation, while the slowest implementation would use one adder and one multiplier to do the computation. When we duplicate such an implementation, two adders and two multipliers are generated, and one multiplier is wasted.

When mapping a module in streaming programs onto FPGAs, by setting different design goals and deploying different optimization configurations we can generate "func-

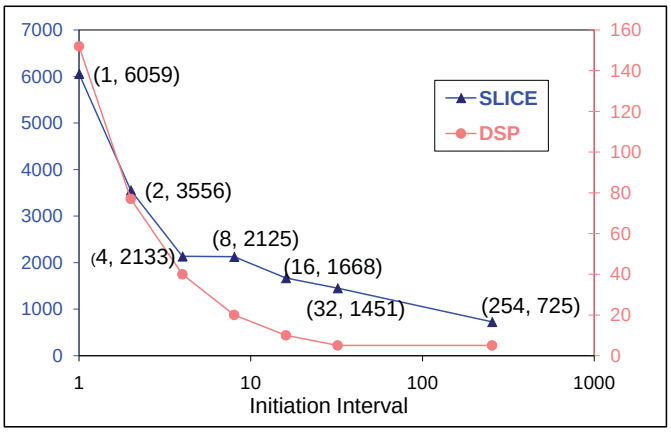


Figure 2.4: Tradeoff between performance and area of one module in benchmark "filterbank."

tionally equivalent" implementations that differ in area and performance. Therefore, we could generate a library of implementations for each module so that design space is enlarged. For example, *initiation interval* (II) is a well-known parameter that trades off between area and throughput. In streaming applications, the initiation interval specifies the number of cycles between the consecutive firing of a module and indicates to what extent the module is pipelined. A pipelined module offers high throughput, but usually at a large area cost because resource sharing opportunities are reduced. Figure 2.3 shows an example of module implementations with and without pipelining.

Figure 2.4 plots the initiation interval and area cost of our FPGA-based implementations for one module in the StreamIt benchmark "filterbank" [Str]. The initiation interval of these design points are 1, 2, 4, 8, 16 and 254, respectively. We can see from the figure that decreasing the initiation interval results in more DSP logics and slices. To achieve a throughput of one firing per cycle, we could either replicate the slowest implementation (with $II = 254$) into 254 copies, or use the implementation with $II = 1$ directly. Obviously the latter option is more area efficient.

It's worthwhile to mention that pipelining does not guarantee that the throughput target can be satisfied, and replication is still needed in cases where a fully pipelined design cannot satisfy the high throughput target. There are also situations when hardware designers choose to use the predefined IP cores to avoid high development cost, and they do not have many choices for different implementations.

2.2.2 High Level Synthesis

Thanks to recent advantages in high-level synthesis (HLS) tools [ZFJ08, CLN11], we are able to design FPGAs using a C/C++ based programming flow. HLS allows user to specify the module optimizations. For example, to enable loop pipelining (Figure 2.3), user only need to insert *"pragma HLS pipeline"* into the code. HLS greatly reduces programming efforts, allowing us to quickly design and evaluate the performance and area consumption of different modules.

2.3 Node-Level CPU-Accelerator Orchestration

Now we present background for node-level CPU execution model and related works on CPU-GPU execution model. Commonly used execution models to exploit data-level parallelism and task-level pipeline parallelism are data-parallel execution model and streaming execution models. In data-parallel execution model, multiple threads are launched in parallel to process the same task but with different data. While in streaming execution model, multiple threads may be launched to process different computation tasks.

2.3.1 Dataflow/Streaming Execution Model

Relevant work includes compilation of new streaming languages such as StreamIt [TKA02] and Brook to multicore architectures [KM08, HCK09, GTA06]. However, these studies do not consider the scenario of offloading computation to accelerators where CPUs may become idle, waiting for the accelerator to finish. Our runtime can make good use of these idled CPU cycles by launching other tasks onto the CPU cores.

2.3.2 CPU and GPU Coordination

Prior works studied the CPU-GPU collaboration for data parallel kernels [KDT12,KKL11, LSP13, LHK09]. These studies present frameworks that can partition the workloads/threads across multiple devices such as CPUs and GPUs. StarPU [KDT12] presents new APIs to offload kernel computations to OpenCL-based GPUs. The work in [KKL11] presents an OpenCL framework/runtime that works on multiple GPUs using a single

compute device image. The work in [LSP13] deals with a single kernel multiple devices system. Scheduling optimizations based on prior profiling information or prediction models are studied as well in [LSP13, LHK09]. However, these studies only explore data-level parallelism and do not consider pipeline parallelism between different tasks.

In this thesis, we will investigate the execution models that consider both data-level parallelism and task-level pipeline parallelism on the heterogeneous CPU-accelerator node.

2.4 Cluster-Level Programming Model and Resource Management

Further expanding the scope of accelerator-rich architectures, we look into resource and data management at cluster-level accelerator-rich architectures. In this section we will review the conventional cluster-level application programming model and discuss how to management cluster resources for cluster tenants.

2.4.1 Cluster-Level Programming Model

MapReduce, originally proposed from Google [DG08], is a programming model and an associated implementation for processing and generating datasets. It inspires much of the initial big data analytics work and serves as a template for open source systems like Apache Hadoop.

In the MapReduce programming model, users specify the computation in terms of a *map* and a *reduce* function. In the map function, each input record is processed to generate a key-value pair. In the reduce function, values associated with the same key are grouped together, and an operation is applied to them to obtain the final results.

map: (k1, v1) $\rightarrow$ *list(k2, v2)*

reduce: (k2, list(v2)) $\rightarrow$ *list(v2)*

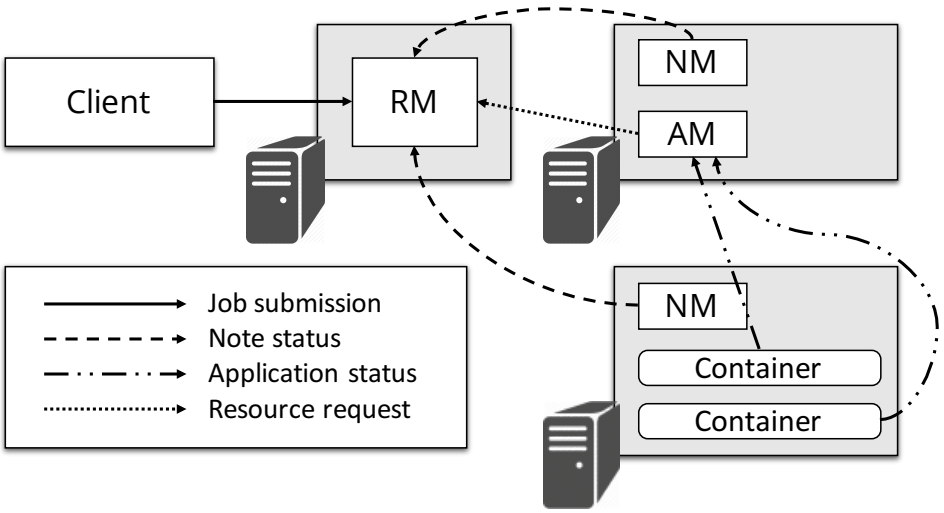


Figure 2.5: Example YARN architecture showing a client submitting jobs to the global resource manager.

2.4.2 Cluster-Level Resource Management

YARN (Yet Another Resource Negotiator) is a widely used cluster resource management layer in the Hadoop system that allocates resources, such as CPU and memory, to multiple big data applications (or jobs). Figure 2.5 shows a high-level view of the YARN architecture. A typical YARN setup would include a single resource manager (RM) and several node manager (NM) installations. Each NM typically manages the resources of a single machine, and periodically reports to the RM, which collects all NM reports and formulates a global view of the cluster resources. The periodic NM reports also provide a basis for monitoring the overall cluster health at the RM, which notifies relevant applications when failures occur.

A YARN job is represented by an application master (AM), which is responsible for orchestrating the job's work on allocated *containers*, i.e., a slice of machine resources (some amount of CPU, RAM, disk, etc.). A client submits an AM package—that includes a shell command and any files (i.e., binary executable configurations) needed to execute the command—to the RM, which then selects a single NM to host the AM. The chosen NM creates a shell environment that includes the file resources, and then executes the given shell command. The NM monitors the containers for resource usage and exit status, which the NM includes in its periodic reports to the RM. At runtime, the AM uses an RPC interface to request containers from the RM, and to ask the NMs that host its

containers to launch a desired program. Returning to Figure 2.5, we see the AM instance running with allocated containers executing a job-specific task.

To manage heterogeneous computing resources in the datacenter and provide placement control, YARN recently introduced a mechanism called *label-based scheduling* [yar]. Administrators can specify labels for each server node and expose the label information to applications. The YARN resource manager then schedules the resource to an application only if the node label matches with the application-specified label. Examples of node labels can be an FPGA or GPU, which indicate that the nodes are equipped with a special hardware platform.

Different from the centralized resource scheduling frameworks (*e.g.,* YARN), Mesos [HKZ11], which originated at Berkeley, is a two-level resource scheduling framework. Its centralized resource manager makes resource offers to its underlying applications, offering one piece of resource to one application at a time. Applications make the decision as whether to accept the resource offer or not. Since Mesos distributes the resource offers without knowing the resource needs of the applications, applications achieve data locality of the resource by rejecting the offers without data locality.

In both YARN and Mesos, applications only see the resources that have been allocated or offered. Omega [SKA13] adopts a completely different strategy by introducing a new share-state resource allocation mechanism and allowing each application to access a view of the overall cluster resource. A lock-free concurrency control is subsequently implemented in their resource manager.

Although structurally different, these cluster resource managers share many common design principles:

- **Data Locality.** Moving computation is cheaper than moving data. This is particularly true when the data size is large. Resource managers should respect applications' data locality and allocate resources close to where the data is located.

- **Multi-Tenancy and High Utilization.** The traditional approach which allocates a physical node to only one tenant at a time results in low cluster utilization. Resource managers should support fine-grained resource sharing among multiple tenants.

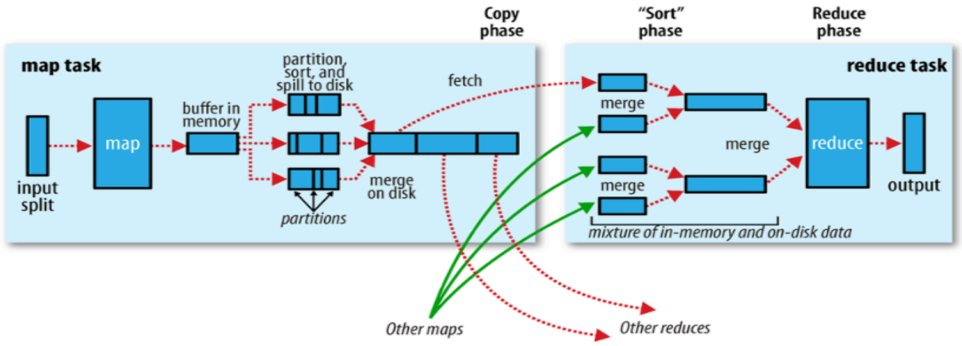


Figure 2.6: Data Sorting and Shuffling in MapReduce. This figure is adopted from [Whi12].

- **Scalability.** Resource allocation should be fast and scalable to increasing cluster sizes. For example, current capacity scheduler in YARN can fill up a decent- sized cluster in less than 100ms. That being said, resource managers should avoid high-complexity resource scheduling algorithms.

2.5 Cluster-Level Data Sort and Shuffle

Data sorting and shuffling is, in many ways, the heart of large-scale distributed computing frameworks, such as MapReduce/Hadoop and Spark. For simplicity, here we use shuffle to refer to this data sorting and data shuffling stage. In MapReduce, shuffle performs the data sorting and data transmission from the mappers to the reducers, guaranteeing that the input to every reducer is sorted by key. It is an area of the codebase where refinement and improvements are continually being made, and is where the "magic" happens [Whi12]. Spark, whose operators are a strict superset of MapReduce, also supports sorting-based data shuffle between consecutive computation stages. In fact, at an earlier stage of Spark development, it adopts a hashing-based shuffle which turns out to be one of the limiting factors of scaling out Spark. Sorting-based shuffle is thus then introduced to Spark. Overall, the data shuffle is an expensive but indispensable operation of many distributed computing frameworks.

Figure. 2.6 provides an illustration of the shuffle stage in MapReduce. At the mapper side, an in-memory buffer collects the output from each map task. When the buffer is almost full (80% by default), a new thread starts to sort the data (using quicksort by

default) and spills the content to the disk. When map tasks finish, a thread performs a merge-sort to merge multiple sorted data on the disk to a single sorted sequence of data. The reducers fetch the data from the mappers and perform a final round of data merging that merges data from the mappers on different nodes. Multiple rounds of merge-sort are performed if the number of mappers is large.

CHAPTER 3

Chip-Level Resource and Data Management

3.1 Introduction

In this chapter we propose a novel system synthesis approach to tackling the resource and data management issues. We formulate the module selection, module replication, buffer size optimization and scheduling simultaneously into one single optimization framework, considering the trade-offs among system throughput, logic and on-chip memory utilization. The proposed framework is shown in Fig. 3.1. It takes the system throughput requirement, application model, and an implementation library as inputs and outputs the module selection/replication results, FIFO size and a feasible schedule. Within the framework, an efficient solving algorithm is presented which achieves both high-quality design (*i.e.* low logic/BRAM costs within the throughput constraints) and scalability.

The remainder of this chapter is organized as follows. Section 3.2 gives a motivation example which shows the benefit of combining computation and communication optimizations in system synthesis. Section 3.3 provides a detailed analytic formulation of the scheduling constraints and objectives for module selection, module replication and

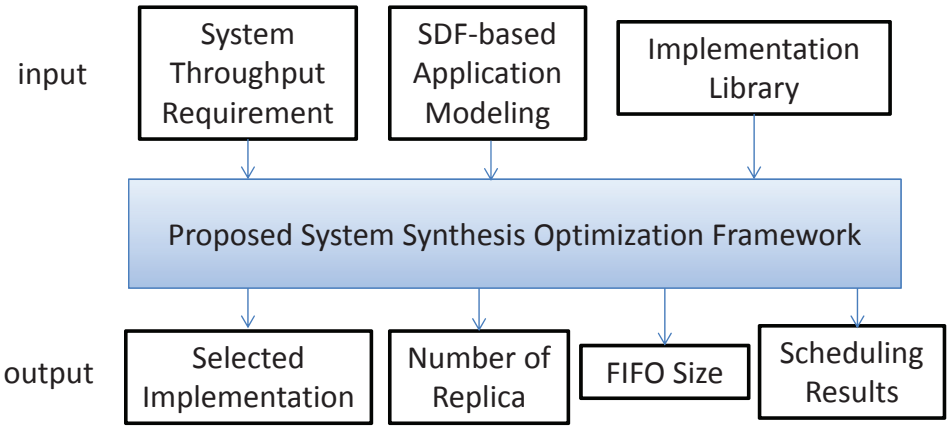


Figure 3.1: The proposed system synthesis framework.

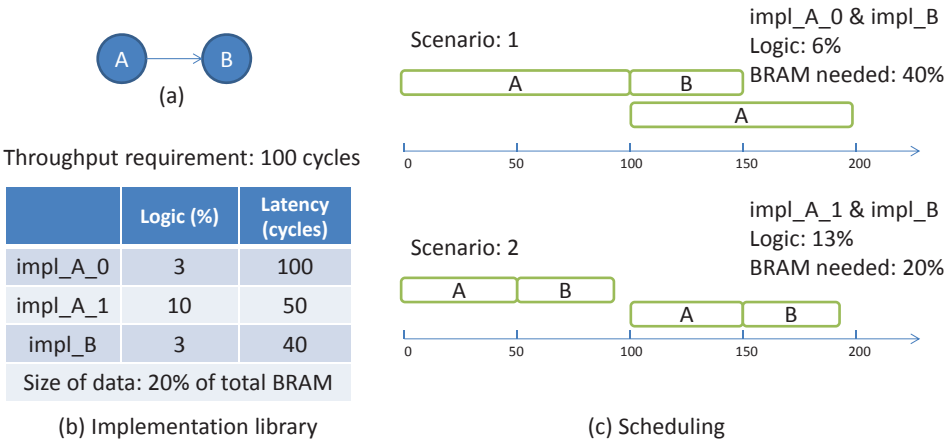


Figure 3.2: A motivating example. In scenario 1, buffer size is not considered during module selection. The implementation with minimum logic is selected. In scenario 2, buffer size is considered together with module selection. In both cases, these are feasible schedules that can meet the system throughput requirement. However in scenario 2 with the consideration of buffer size, the selected implementation can reduces BRAM use by 20%.

buffer size optimizations. Section 3.4 proposes an efficient iterative algorithm to find the solution to the combined optimization problem. Experimental results are shown in Section 3.5, where both design quality and run-time complexity are measured with streaming benchmarks.

3.2 A Motivation Example

Fig. 3.2 gives a motivational example. We show a very simple case in streaming applications in Fig. 3.2(a), where A is the producer actor and B is the consumer actor. A and B are executed repeatedly.

The implementation options are listed in Fig. 3.2(b). Actor A has two implementations, impl_A_0 and impl_A_1, which show the trade-offs between the logic cost and execution latency. Actor B has only one fixed implementation. Each of actor A's executions produce a fixed size of data to the communication channel. We assume the size is 20% of the FPGA on-chip memory. Each of actor B's executions consume the same amount of data from the communication channel. The system throughput target is set to one execution of A and B in every 100 cycles.

In Fig. 3.2(c) scenario 1 , we decouple the computation and communication optimizations by first considering module selection without considering buffer size. The solution with minimal logic utilization is considered to be the optimal one. Then the scheduling that has the minimum buffer size requirement is calculated afterwards. Therefore, in the optimal solution, impl_A_0 and impl_B are selected. Then in this case, we need to double the data buffer size which takes 40% BRAM resource, since a second execution of A has to be overlapped with the first execution of B in order to meet the throughput constraint. And the second execution of A has already started before the first execution of B finishes. In Fig. 3.2(c) scenario 2, we consider buffer size optimization together with module selection. In this case, a slightly large implementation of A is selected which can significantly improve the performance of actor A (at the cost of 7% logic increase compared to scenario 1). Then we do not have to execute A and B in parallel. When the second execution of A starts, the first execution of B has finished and the communication channel is empty. Thus, the needed buffer size is only 20% of total BRAM.

From the example we see that we cannot take computation optimizations (module selction/replication) and communication (buffer size) optimizations as two separate steps. Thus, in this work we propose to combine computation and communication optimizations together during system synthesis.

3.3 System Mapping

With the modeling of application behavior in the HSDF graph, our system mapping problem can be defined as the following:

Given application modeling in an HSDF graph consisting of actor firings n_i^j as nodes and data dependencies between actor firings as edges, the target throughput requirement Thr_{tar}, and implementation options P_n^s for each actor with the logic cost $A(P_n^s)$, execution latency $et(P_n^s)$, and pipeline initial interval $\delta(P_n^s)$, we find the optimized module selection results P_n^{sel} and replication factors rep_n for each actor, the total FIFO size, and the periodic scheduling t_i^j for each actor firing. The BRAM and logic utilization on FPGA is minimized under the given system throughput constraint.

A list of the denotations used in the following sections of the chapter can be found in

Table 3.1: Denotation of variables.

parameters	meaning
n_i^j	jth firing of actor n_i
n'^j_i	jth firing of actor n_i in 2nd iteration
Lib_n	implementation library for actor n
P_n^s	implementation in library Lib_n
$A(P_n^s)$	area cost of P_n^s
$et(P_n^s)$	execution time of P_n^s
$\delta(P_n^s)$	initiation interval of P_n^s
Thr_{tar}	throughput target
variables (unknown)	meaning
t_i^j	start firing time of n_i^j, scheduling variables
t'^j_i	start firing time of n_i^j in 2nd iteration, scheduling variables
b_n^s	if implementation P_n^s is selected
rep_n	number of replicas for each module

Table. 3.1.

In this section we discuss the problem of mapping HSDFGs onto FPGAs. First, we generate an FPGA implementation library for each actor which realizes the functionality of streaming application kernels. Second, given the selected modules, we formulate all the scheduling constraints and delineate them as new edges on the HSDFG. We refer to the HSDFG with scheduling constraints as the scheduling graph. Last, we formally state the whole system synthesis framework, which contains module selection/replication, buffer size optimization and scheduling. An ILP formulation is presented throughout this section to capture the optimal solution. We present a scalable and efficient near-optimal algorithm in Sec. 3.4.

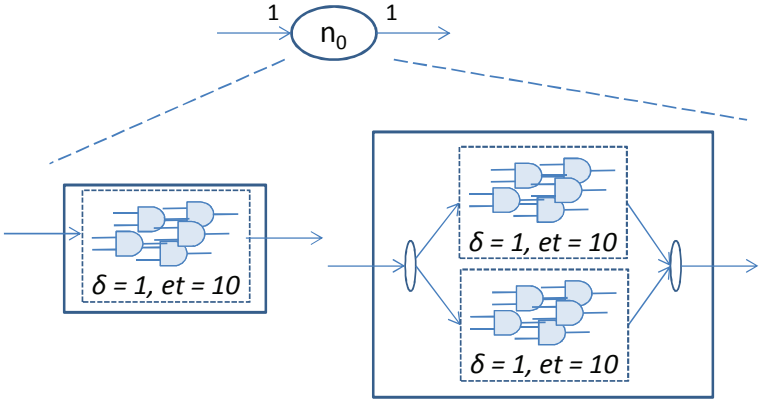


Figure 3.3: An example of module replication. Module throughput can be further improved by duplicating the modules and adding the corresponding split and join logic.

3.3.1 Implementation Library Constraints

In realizing streaming applications onto FPGAs, by setting different optimization configurations we can generate a set of "functionally equivalent" implementations that differ in area and performance. Usually a significant area saving can be achieved by using slower implementations. In terms of performance, hardware implementations can differ in initiation interval (δ) and latency (et). Initiation interval specifies to what extent the hardware module can be pipelined. Latency specifies execution time of the hardware module. It is the time interval between the time point when an actor starts firing and the time point when the actor finishes its firing and produces data tokens to its output channels.

In streaming applications, initiation interval means the number of cycles that the two consecutive firings, which are mapped to the same module, need to be separated by. An initiation interval $\delta = 1$ indicates a fully pipelined module with a new firing initiated every clock cycle. Hardware modules with a smaller δ, and thus with a higher throughput, are usually achieved at a larger area cost because a smaller δ reduces many resource-sharing opportunities. For the same reason, hardware modules with a smaller et require more logic for parallel computing since the logic can not be shared if they are used in parallel.

Module replication techniques can further improve the module throughput even when δ has decreased to 1. Fig. 3.3 shows an example of module replication. The actor consumes one data token and produces one data token on each firing. Assume we already have a module that achieves $\delta = 1$ and $et = 10$ and we replicate it into two copies. Then

from a system point of view, such an implementation consumes and produces two data tokens on each firing in every clock cycle. Note that additional split and join logic is needed here to distribute the data to different modules in a round-robin fashion.

Several commercial C-to-FPGA high-level synthesis tools now provide the ability to synthesize hardware modules with a specified pipeline level and parallelism level. For example, the initiation interval and loop unrolling factor can both be specified for a loop. FPGA synthesis tools can also provide an estimation of area usage, initiation interval and latency of the current synthesized hardware modules. Thus, given a C program for streaming applications, we can quickly generate different hardware implementations by altering the design configurations, and build up a library of implementations. In addition, to save development cost, we can also adopt IPs (*e.g.* FFT) cores provided by FPGA vendors into our implementation library. In this work, granularity of a module is defined by users — it can be a few instructions for the smallest module or multiple loops for a large module.

The implementation alternatives are formulated as the following: For each actor n, the implementation library is Lib_n $(P_n^1,\ P_n^2, ...P_n^{S_n})$, where each implementation P_n^s in Lib_n can perform the functionality of n with area cost $A(P_n^s)$, initiation interval $\delta(P_n^s)$, and execution time $et(P_n^s)$. S_n denotes the number of implementations we have for actor n. P_n^{sel} denotes the selected implementation from the library. A binary variable b_n^s is introduced to denote if P_n^s is selected. Thus, $A(P_n^{sel})$, $\delta(P_n^{sel})$ and $et(P_n^{sel})$ (area costs, initiation interval and latency of the selected modules respectively) can be formulated as the following:

$$
\begin{aligned}
A(P_n^{sel}) &= rep_n \cdot \sum_s b_n^s \cdot A(P_n^s) \\
\delta(P_n^{sel}) &= \sum_s b_n^s \cdot \delta(P_n^s) \\
et(P_n^{sel}) &= \sum_s b_n^s \cdot et(P_n^s) \\
\sum_s b_n^s &= 1,
\end{aligned}
$$

where rep_n denotes the number of replicas for the selected implementation.

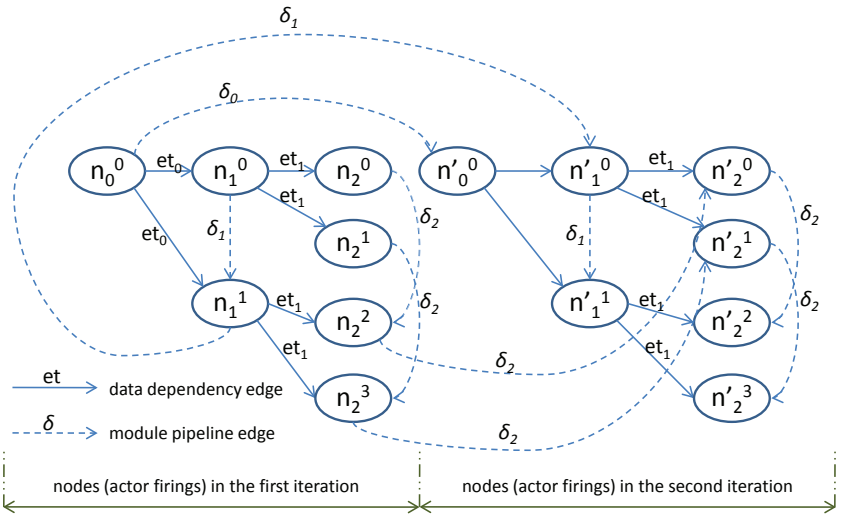


Figure 3.4: Scheduling graph for SDFG in Fig. 2.1. It contains all the actor firings in two iterations. Detailed explanations of the edges can be found in Sec. 3.3.2.

3.3.2 Scheduling Constraints From Computation Modules

In order to model a periodic scheduling, we proposed to depict the throughput constraint on two consecutive SDF iterations. Therefore, we unfold the HSDFG twice to include all the actor firings in two iterations, as shown in Fig. 3.4. We also add edges to delineate the hardware dependencies and communication buffer constraints. We call this type of graph a **scheduling graph**.

Definition 2 *A **scheduling graph** is a graph $G(V, E)$, consisting of a set of actor firings in two iterations and a set of edges including data dependency edges, module pipeline edges and buffer-constrained edges(explained in detail below).*

In the scheduling graph, node n_i^j denotes the jth firing of the actor n_i. We would map each actor firing to a hardware module. Actor firings that perform the same computation can share the same hardware implementation. For example, actor firings n_2^j and n'^j_2 $(j = 0, 1, 2, 3)$ are all performing the same computation task, and thus can be mapped to the same module.

We associate a weight for each edge in the scheduling graph. The weight denotes the number of cycles that the two firings connected by this edge need to be separated by. The edges can be divided into three categories:

Data dependency edges: Edge weight equals execution latency (et) of the hardware

module that the predecessor[1] of this edge is mapped to. These are the edges between the firings of different actors. They reflect the producer-consumer relations where the successor cannot start firing until the predecessor has finished its firing. For example, the weight of the edge between n_1^0 and n_2^0 is the execution time of the hardware module for n_1^0, since n_2^0 cannot start firing until n_1^0 produces the data to the communication channel. We use FIFOs as data communication channels.

Module pipeline edges: Edge weight equals initiation interval (δ) of the hardware module. These are the edges between the firings that are mapped to the same hardware module. They need to be separated by δ cycles since a hardware module can only start to process a new firing every δ cycles. When the number of replicas for an actor is more than one, we adopt the cyclic scheduling policy to assign firings to the hardware module. The cyclic scheduling policy works in a round-robin fashion that always assigns a firing to the first available hardware module. For example, if the number of replicas is two, then all the even-numbered firings are assigned to the first module and all the odd-numbered firings are assigned to the second module. In Fig. 3.4, we assume two copies of hardware modules are selected for actor n_2; thus firings n_2^0, n_2^2, n'^0_2 and n'^2_2 are mapped to one module, and the other firings of n_2 are mapped to the other module. In Fig. 3.4, the corresponding edges are added between n_2^0, n_2^2, n'^0_2 and n'^2_2 with weight equal to δ_2.

The detailed formulations of the module pipeline edges are as follows:

$$t_i^{j_2} - t_i^{j_1} \geq \begin{cases} \delta(P_{n_i}^{sel}), \, if \; j_2 - j_1 = rep_n \\ 0, \quad otherwise \end{cases} \tag{3.1}$$

where $t_i^{j_2}$ and $t_i^{j_1}$ denote the j_2th and j_1th firings of actor n_i.

3.3.3 Scheduling Constraints From Communication Channels

The size of communication channels also impose scheduling constraints. For example, a producer module cannot push new data tokens into an output channel if the channel is full. Similarly, a consumer module cannot start its computation if there is not enough data tokens in the input channel. Therefore furthur delineates the scheduling constraints from the communication channels.

[1]If there exists a directed edge e from u to v, we call u the predecessor, and v the successor of edge e.

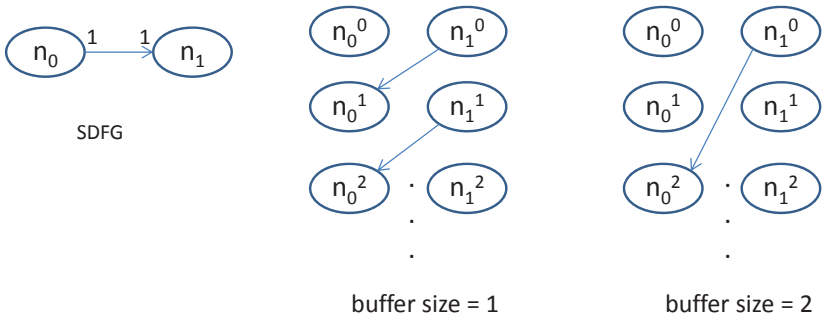


Figure 3.5: An example of buffer-constrained edges. An SDFG is shown on the left. Buffer-constrained edges are shown in the scheduling graph when buffer size is 1 and 2 respectively. The producer rate and consumer rate is 1.

Buffer-constrained edges: Here we model the scheduling constraints imposed by buffer size as buffer-constrained edges. The edge weight of a buffer-constrained edge equals the execution latency (et) of the consumer task. Given a buffer size between the producer actor and consumer actor, we can derive a set of scheduling constraints among the producer and consumer actor firings. Consumer actors should fire in time to consume the data tokens so that the data tokens in the communication channel will not increase in a cumulative manner and exceed the capacity of the buffer. We adopt a conservative model as in [SGB06] where the buffer space to produce output data token is assumed to be available at the beginning of the actor firing.

Given produce actor n_0 and consumer actor n_1, we add an edge from n_1^j (jth firing of n_1) to n_0^i (ith firing of n_0), if the following equation holds:

$$j = \min\{k | (i+1) * p - (k+1) * c \leq buf\}, \tag{3.2}$$

where p is the producer rate and c is the consumer rate. buf is the given buffer size. $(i+1) * p$ is the number of produced data tokens, and $(k+1) * c$ is the consumed data tokens. (It is not $i * p$ since in our model label i starts from 0.) The intuition is that enough data tokens should have been consumed before the ith firing of the producer n_0. Fig. 3.5 shows an example. If the buffer size is only 1, then actor n_0^1 cannot start firing until n_1^0 finishes, which consumes the data in the FIFO. If the buffer size is 2, then actor n_0^1 can always start firing regardless of actor n_1^0. The detailed ILP formulation for buffer-constrained edges is omitted here due to page limitation.

3.3.4 Problem Statement

The design space exploration problem for streaming applications requires selecting implementations from the library for each actor under throughput constraint while minimizing the total area cost. To meet the throughput target, the selected hardware implementations from the libraries can be replicated.

Given a scheduling graph $G(V, E)$ and $n_{i1}^j, n_{i2}^k \in V$, $e(n_{i1}^j \rightarrow n_{i2}^k) \in E$ denotes an edge between the jth firing of actor n_{i1} and kth firing of actor n_{i2}. Based on our discussion in Sec. 3.3.2, weight $w(e)$ of the edge $e(n_{i1}^j \rightarrow n_{i2}^k)$ can be derived as

$$w(e) = \begin{cases} et(P_{n_{i1}}^{sel}), & if\ e\ is\ data\ dependency\ edge \\ \delta(P_{n_{i1}}^{sel}), & if\ e\ is\ module\ pipeline\ edge \\ et(P_{n_{i1}}^{sel}), & if\ e\ is\ buffer-constrained\ edge \end{cases} \tag{3.3}$$

where $P_{n_{i1}}^{sel}$ is the selected implementation from the actor n_{i1}'s library. Although the formulation of the data dependency edge and the buffer-constrained edge look similar, the difference is that a buffer-constrained edge is an edge from a consumer node to the producer node, while a data dependence edge is an edge from a producer node to a consumer node. Our design space exploration problem can be described as the following:

Input: (1) Scheduling graph $G(V, E)$ of the streaming applications, (2) throughput target Thr_{tar}, and (3) the implementation libraries.

Output: (1) Selected implementation P_n^{sel} from the library for each actor n, (2) number of replicas rep_n^{sel}, (3) buffer size for each communication channel, and (4) start firing time t_i^j for each actor firing n_i^j.

The selected implementations should satisfy the following constraints:

Constraints:

$$\forall e(n_{i1}^j \rightarrow n_{i2}^k) \in E,\ t_{i2}^k - t_{i1}^j \geq w(e), \tag{3.4}$$

$$\forall\ n_i^j \in V,\ t'^j_i - t_i^j = c, \tag{3.5}$$

$$c = \frac{1}{Thr_{tar}}. \tag{3.6}$$

Optimization goal:

$$minimize : \frac{BufSize}{TotalBufSize} + \frac{Logic}{TotalLogic} \tag{3.7}$$

Constraint (3.4) ensures the precedence of actor firings. Constraints (3.5) and (3.6) guarantee that the schedule is periodic and should meet the system throughput. Variable denotations are listed in Table 3.1 for references. Buffers can be implemented as on-chip block RAM, and hardware modules are implemented using slices. The area metric we used here is the sum of occupied percentages of buffer and logic. The optimization goal can be extended to other area metrics as well. Note that we do not consider the BRAM utilization of the modules. In fact, we assume that modules can always use distributed RAM instead of BRAM whenever possible. This is easy to achieve in many high-level synthesis tools where resource types are explicitly defined by users.

In Sec. 3.4, we present an efficient iterative exploration algorithm to solve the overall problem.

3.4 Proposed Approach: ST-Syn

Algorithm 1 ST-Syn Overall Algorithm

Step 1: Update the scheduling graph with newly selected hardware modules or newly allocated buffer size (*e.g.,* update the edge weight).

Step 2: Schedulability checking process. Check if there is a feasible schedule that can satisfy the system throughput given the current setting of modules and buffer size. If not, go to Step 3. Otherwise terminate.

Step 3: Module/Buffer size improvement. Go to Step 1.

In this section we present our algorithm called ST-Syn (streaming synthesis), as summarized in Algorithm 1. It is an iterative exploration algorithm that contains schedulability checking and efficient implementation improvement techniques. The algorithm starts from an implementation that utilizes the minimum logic and buffer size. For each computation kernel, the implementation with the smallest area is selected. Number of replicas is set as 1. The buffer size is set to the larger value between the producer rate and the consumer rate, which is the least buffer size required by both the producer actor and the consumer actor to fire. Given such configurations, we try to schedule all the actor firings under the performance requirement. If the schedulability check fails, we try

to improve the implementation by either selecting a faster implementation or increasing the buffer size. We repeat the schedulability checking and implementation improvement process until the system performance can be satisfied.

More specifically, the schedulability checking is formulated as a *system of difference constraints* (SDC) problem. Implementation improvement is formulated as a minimum cut problem in a weighted penalty graph. Both of these problems can be solved in polynomial time using LP relaxation with integer solution guaranteed.

3.4.1 Schedulability Checking

In this section we show that schedulability checking can be formulated as a system of difference constraints, and thus can be done in polynomial time. As discussed in Section. 3.3.2, given selected hardware modules and buffer size, we can add the data dependency edges, module pipeline edges and buffer-constrained edges in the scheduling graph, thus finalizing the graph structure as well as the edge weights. The only unknown variables are the scheduling variables t_i^j. The edges in the scheduling graph represent the scheduling constraints. We can mathematically model the scheduling constraints as a set of *difference constraints*. Using the definition in [CZ06], *difference constraints* are defined as follows:

Definition 3 *An integer difference constraint is a formula in the form of $x - y \leq w$ for integer variables x and y, and constant w.*

The schedulability check problem is to determine whether there is a solution satisfying the constraints (3.4), (3.5) and (3.6). Note that in constraints (3.4), $w(e)$ is now a known value since we already decided the module implementations and buffer size.

Lemma 1 *SDC-based schedulability checking can be solved in polynomial time.*

Proof is based on the results in [CZ06]. Although t_i^j and t'^j_i are integer variables, schedulability check can be solved in polynomial time using linear programming relaxation. This is because the underlying matrix for SDC is a *totally unimodular* matrix [CZ06]. Thus, if there is a feasible solution under linear programming relaxation, it is also a feasible integer solution.

3.4.2 Iterative Improvement

Algorithm 2 Iterative Improvement

Step 1: Identify ϵ-critical paths in the scheduling graph.

Step 2: Associate an area penalty with each edge.

Step 3: Perform a min-cut on the graph.

Algorithm 2 shows our iterative improvement algorithm. When the current settings of hardware modules and buffer size cannot meet the system throughput target, we need to either find a "better" module (a faster module or a larger number of replicas) which can contribute to system performance, or increase the buffer size of the communication channels. We do so by first identifying the bottlenecks in the system. The current system is delineated as a weighted directed scheduling graph, and scheduling bottlenecks can be viewed as critical paths in the graph.

Definition 4 *The **critical path** of the scheduling graph $G(V, E)$ as the longest path among all the paths between node n_i^j and node n'^j_i, $\forall i, j, n_i^j \in V, n'^j_i \in V$.*

The general idea here is that the newly selected modules or buffer size should be able to contribute to the system performance in the sense that the length of the critical path is decreased. For the efficiency of the algorithm, instead of identifying a single critical path, we adopt the network flow theorem to find all the ϵ-critical paths ([SWB88]) in the system.

Definition 5 *Denote the length of the critical path of the scheduling graph is L. The **ϵ-critical paths** of the scheduling graph $G(V, E)$ are the set of paths between node n_i^j and node n'^j_i whose length is $L(1-\epsilon)$,$\forall i, j, n_i^j \in V, n'^j_i \in V$.*

Then for each edge on the ϵ-critical paths, we calculate the area penalty needed to improve this edge. (a) If the edge is a data dependency edge, then we can select a module with a smaller execution latency from the implementation library. (b) If the edge is a module pipeline edge, then we can select a module with a smaller initiation interval or we can increase the number of replicas. (c) If the edge is a buffer-constrained edge, then

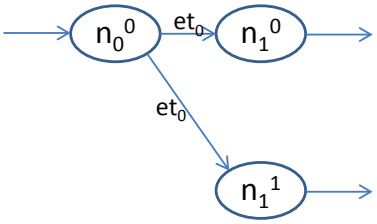


Figure 3.6: An example of part of an ϵ-critical path. An improved hardware module of n_0 will contribute to both paths in the graph.

we can increase the buffer size by $gcd\{p, q\}$ (the greatest common divider between the producer rate and consumer rate). It is shown in [SGB06] that buffer size should be a multiple of $gcd\{p, q\}$ to be used usefully. Area penalty is calculated as the increased percentages of logic or buffer utilization. To assign a fair penalty to each edge, in both case (a) and (b), we require that the newly selected module be able to reduce the critical path by at least Δ. And Δ can be used to adjust the convergence speed of the algorithm.

The ϵ-critical paths and the associated weight (area penalty) on the edges form a weighted directed graph. Denote such a graph as $H(V, E)$. We perform a minimum cut on the graph so that all the critical paths can be improved and the total area penalty is minimum.

The **minimum cut** problem is formulated as follows. Associate each node in the graph $H(V, E)$ with a variable p_i. p_s and p_t denote the variables for the source and sink node. Associate each edge $(i, j) \in E$ with a binary variable d_{ij} to indicate whether this edge is cut or not. Let c_{ij} denote the edge weight. Thus the problem constraints are:

$$\begin{aligned} d_{ij} - p_i + p_j &\geq 0, \quad (i,j) \in E \\ p_s &= 1 \\ p_t &= 0 \\ p_i &\geq 0, \qquad i \in V \\ d_{ij} &\geq 0, \quad (i,j) \in E \end{aligned} \tag{3.8}$$

Note that d_{ij} is not explicitly specified as a binary variable in the constraints. Nevertheless, we will show later that such relaxation can still guarantee that d_{ij} is binary.

The tricky part is how to formulate the optimization goal. It is not simply a summation of $d_{ij} * c_{ij}$. This is because improvement of one hardware module/buffer will have

an effect on multiple edges in the ϵ-critical paths. For example, in Fig. 3.6, assume both edges from n_0^0 to n_1^0 and n_1^1 are on the ϵ-critical path; to decrease the edge weight, a faster module of n_0 will be selected. The area penalty would be the same on both edges. Let us denote it as c. Thus, if a min-cut cuts through both these two edges, then the total area penalty should be c, rather than $2 * c$, since only one module is improved. In this case, the optimization goal can be formulated as $max\{d_{01} * c, d_{02} * c\}$, where d_{01} and d_{02} are the binary variables to denote whether these two edges are cut or not in the min-cut. With this $max\{\}$ formulation, when both edges are considered as cut in the min-cut, the edges only contribute c to the overall area penalty rather than $2 * c$.

Generally, we can divide the edges in $H(V, E)$ into several sets ϕ_k. The edges that require improvement of the same module/buffer are grouped into the same set ϕ_k. Thus, the optimization goal of the min-cut problem is:

$$\sum_k \max_{(i,j)\in\phi_k} d_{ij} * c_{ij}. \tag{3.9}$$

Since the maximum of two convex function is also convex, the above optimization goal is convex and the underlying matrix is *totally unimodular*. Thus the problem can be solved by LP relaxation with integer solution guaranteed [HK10].

Lemma 2 *The minimum cut problem (constraints in (3.8) and objective in (3.9)) can be solved in polynomial time.*

3.4.3 Update Scheduling Graph

When a new implementation or number of replicas is selected, or a new buffer size is allocated, we need to update the scheduling graph: (1) Module pipeline edges are deleted and new edges are added as the number of replicas changes. For example, assume that originally the number of replicas is two, then there is a module pipeline edge between every two actor firings. When the number of replicas changes to 3, there should be such an edge between every three actor firings. (2) Module pipeline edge weights are changed to initiation intervals of the newly selected modules. (3) Data dependency edge weights are changed to the execution time of the newly selected modules of the producer actor. (4) Buffer-constrained edges are updated as buffer size changes. The edge weight is changed to the execution time of the newly selected modules of the consumer actor.

Note that we will add buffer-constrained edges into the scheduling graph after we add all the data dependency and module pipeline edges. The reason is that adding buffer-constrained edges may result in cycles in the scheduling graph. If a cycle is detected, then no feasible schedule exists. In this case, we continue to increase the buffer size. Intuitively, if the buffer size is large enough, no buffer-constrained edges will need to be added into the scheduling graph.

3.4.4 Complexity of ST-Syn

In each iteration we perform a scheduling graph update, schedulability checking, and module improvement. For each kind of edge (data dependency, module pipeline and buffer-constrained), the maximum number of edges is $O(V^2)$, where V is the number of nodes (actor firings in the scheduling graph). Thus, complexity of the scheduling graph update is polynomial to the number of edges in the system. Together with Theorem 1 and Theorem 2, we can obtain the following theorem:

Theorem 1 *Each iteration of ST-Syn takes polynomial time.*

To avoid oscillation during the iterative improvement, we adopt a simple heuristic where a selected hardware candidate will be excluded from the implementation library. A hardware candidate refers to both the selected hardware module and the number of replicas. It means that in the current iteration, if implementation P_n^i is selected with the number of its replicas equal to r, then in any following iterations, P_n^i can no longer be selected with the number of replicas equal to r. However, we still allow P_n^i to be selected if the number of replicas is not equal to r. Thus, the algorithm is guaranteed to converge since there are only limited design points to explore.

3.4.5 Alternative Solution using Integer Linear Programming

We can also formulate the overall problem using an integer linear programming. We need to introduce a set of binary variables to indicate which kernel implementations are selected. Therefore we can formulate the area consumption and module latency. We also need to introduce binary variables to indicate the scheduling order of the modules, based on which we can formulate the buffer size. Integer linear programming provides us the

optimal solution but it is a very time-consuming a process. We will demonstrate more in the experiments.

3.5 Experiments

In this section we discuss two case studies for evaluating our algorithms. The first case is a FIFO-based merge-sort. FIFO-based merge sort is shown to be a high-performance sorting architecture on FPGAs for large sorting problems [KT11]. The design is bounded by communication resources. Thus it is a BRAM dominate design. The second case is MPEG-4 where the logic and BRAM utilizations are comparable under different system performance requirements.

We implement three methods: (1) ILP_Separate, the ILP formulation in [CHL12] which solves the module selection/replication problems without considering buffer size, and then the required buffer size is minimized according to the system scheduling result of the computation optimization. (2) ILP_Buf, an integrated ILP formulation that combines module selection, replication and buffer size optimizations. (3) Our proposed ST-Syn algorithm.

In addition, we compare our iterative exploration with another simple straightforward heuristics for the MPEG-4 decoder to demonstrate the superiority of our proposed algorithm.

3.5.1 Settings

The high-level synthesis tool *Vivado_HLS* from Xilinx (version 2013.2) is used to perform the estimation of performance and resource usage. We argue that detailed logic synthesis in physical design, although providing us with accurate performance and resource usage information for the hardware design, is too time-consuming, and thus not suitable for use in the early stage of hardware design. To generate different hardware modules for each actor, we adopt several design techniques, such as (1) loop unrolling, (2) loop pipelining, (3) array partitioning, and (4) software pipelining. Each implementation has three attributes: initiation interval, execution time, and area cost. Initiation interval and execution time can be retrieved from the $Vivado\_HLS$ synthesis report. The area metric

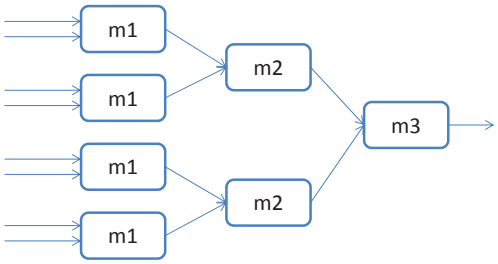


Figure 3.7: An example of merge sort to sort 8 values. $m1$ takes one value from each of its two input channels, reorders the two values and then sends them out to $m2$. $m2$ takes two values from each of its two input channels, merges them into one sorted stream that contains 4 values. $m3$ works in a similar fashion and outputs a sorted stream that contains 8 values.

we use is a combination of the on-chip resources (*i.e.*, FF and LUT for computation, and block RAM for data communication). The logic utilization is estimated as the maximal value between FF utilization and LUT utilization. The area metric used in the ILP objective function is the sum of logic utilization and BRAM utilization. Thus the area cost is reduced whenever BRAM or logic consumption reduces. The target platform in hardware design is Zynq XC7Z020, and the target FPGA clock cycle is set at 100M Hz.

The ILP solver we use is GLPK [GLP]. If the ILP solver fails to achieve the optimal results within two hours, we use the suboptimal result returned from the solver, which is the best solution that it can get in two hours.

3.5.2 FIFO-based Merge Sort

FIFO-based merge sort [KT11] contains a cascaded chain of merge kernels. Each merge kernel merges two sorted streams of data into a single sorted stream. Fig. 3.7 shows an example. All the data produced are first streamed into FIFOs before the consumer starts to process it. Thus the FIFO size increases as the size of sorting problems increases.

Fig. 3.8 shows the area utilization under different throughput settings. Results from ILP_separate, ILP_buf and ST-Syn are all shown in the figure. The following are some resource usage numbers from this set of experiments:

(1) The BRAM utilization (resource for communication channels) is much larger than the logic utilization. Variation of logic utilization (range: 2% to 8%) under different

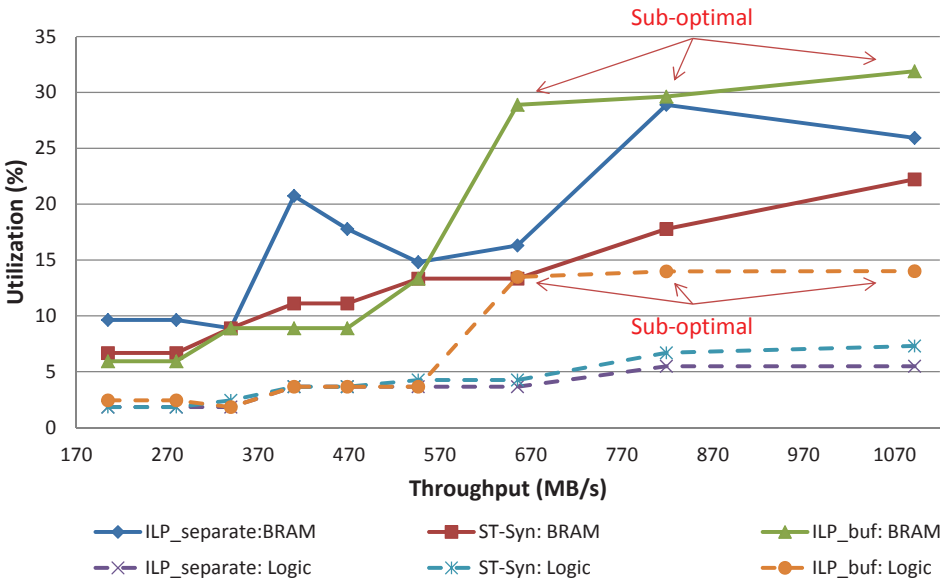


Figure 3.8: FIFO-based merge sort: area utilization (logic & BRAM) under different throughput settings. 16384 values are merged. Runtime of ILP_buf is limited to up to 2 hours, and thus ILP_buf only generates sub-optimal results.

throughput is much smaller compare to the variation of BRAM utilization (range: 5% to 30%). Therefore, it is important to consider buffer size optimizations together with module selection and replication in system synthesis. Moreover, we can see from the gap between ILP_buf:BRAM and ILP_separate:BRAM curves that the BRAM usage overhead of ILP_Separate is 62% compared to ILP_buf. (BRAM overhead is calculated as: (BRAM utilization of ILP_separate - BRAM utilization of ILP_buf)/(BRAM utilization of ILP_buf). Logic overhead is calculated in the same fashion.)

(2) The proposed ST-Syn can achieve near-optimal quality of results. BRAM usage overhead of ST-Syn is only 12% on average. Runtime of ST-Syn is less than 1 second in all cases.

(3) In terms of total area cost (logic + BRAM), the overhead of ILP_separate and ST-Syn are 41.3% and 7.9% compared to ILP_buf respectively.

Moreover, there are several observations that match our expectations:

(4) Logic utilization does not strictly correlate with the system performance requirement. For example, comparing the second point and third point on the ILP_buf curve, we can see that logic utilization actually drops when system throughput increases. This is not surprising since we are optimizing the sum of logic and BRAM utilization, and in

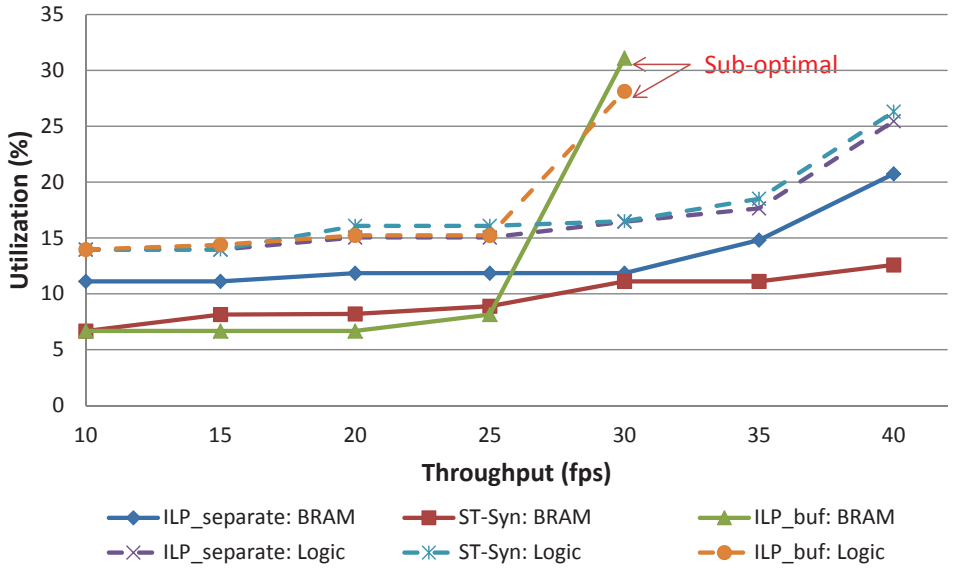


Figure 3.9: MPEG4: area utilization under different throughput settings. Runtime of ILP_buf is limited up to 2 hours. ILP_buf does not return a feasible integer solution at 2 hours when throughput is 35 fps and 40 fps.

this case, the BRAM utilization has significantly increased from the second point to the third point.

(5) BRAM utilization does not strictly correlate with the system performance requirement. The size of BRAM reflects the data balance between the producer and consumer kernels. Intuitively, if the producer produces data too early while the consumer consumes data too late, a large FIFO size is needed. Thus, BRAM size highly depends on the selected implementations, number of replicas and scheduling. And it may not increase as system throughput requirement increases.

3.5.3 MPEG4

We show the area utilization of MPEG4 under different throughput settings in Fig. 3.9. In our experiment, throughput varies from 10 fps (frames/second) to 40 fps. In the case of MPEG4, logic utilization is higher than the BRAM utilization. Compared with ILP_buf (suboptimal results are excluded from the comparison), logic overhead is -8.2% and BRAM overhead is 61.5% in the case of ILP_separate. In the case of ST-Syn, logic overhead is -0.02% and BRAM overhead is only 12.4%. Runtime of ST-Syn is less than 10 seconds in all cases, while ILP_buf takes minutes to hours.

	problem	through-	# of	Runtime (s)			Logic Utilization (%)			BRAM Utilization (%)		
	size	put	variables	ILP_buf	ST-Syn	speedup	ILP_buf	ST-Syn	overhead (%)	ILP_buf	ST-Syn	overhead (%)
sort	4096	250MB/s	154	0.05	0.01	5	1.80	1.80	0.0	4.40	4.40	0.0
	4096	500 MB/s	154	1.05	0.03	35	3.60	3.60	0.0	5.20	5.60	7.7
	8192	500 MB/s	462	177	0.05	3540	3.60	3.60	0.0	6.70	6.70	0.0
	8192	1 GB/s	462	422	0.08	5275	6.02	7.20	19.6	9.26	9.30	0.4
MPEG4		10 fps	3058	3	0.02	150	14.00	14.00	0.0	6.70	6.70	0.0
		15 fps	3058	72	0.1	720	14.40	14.00	-2.8	6.70	8.15	21.6
		20 fps	3058	46	4.1	11	15.20	16.10	5.9	6.70	8.20	22.4
		25 fps	3058	3297	9	366	15.20	16.00	5.3	8.20	8.90	8.5
JPEG		1 fps	878	23	0.05	460	12.64	12.64	0.0	7.41	8.60	16.1
average						1174			3.1			8.5

Figure 3.10: Overall speedup and area overhead.

3.5.4 Overall Speedup and Area Overhead

Fig. 3.10 shows runtime of FIFO-based merge sort and MPEG4, as well as JPEG. We try to cover different designs by varying the problem size and the throughput requirement. In the case of merge sort, two different problem sizes are considered — 4096 and 8192 values are sorted. The number of variables (including intermediate variables) in ILP formulation is also listed. Note that given the same problem size, when throughput changes, the runtime of the ILP solver also varies. When throughput is set high, ST-syn usually runs for more iterations and thus takes a longer time. We compare our ST-Syn with ILP_buf in terms of speedup as well as logic and BRAM overhead. The average logic overhead is 3.1%, and the average BRAM overhead is 8.5%. However, our ST-Syn is 1174x faster than ILP_buf on average.

3.6 Conclusion

Communication and computation optimizations are two central aspects in system-level synthesis. System-level synthesis should consider both aspects in a unified framework rather than decouple them into two processes. In this chapter we investigate an efficient system-level synthesis algorithm for a combined communication and computation optimization problem. More specifically, we provide a complete formulation and solution to deal with the throughput-driven module selection/replication and buffer size optimization problem. We first derive a scheduling graph and then perform an iterative exploration algorithm by formulating the schedulability checking as a system of difference constraints problem and the module improvement as a min-cut problem. The proposed algorithm

runs in polynomial time with little area overhead.

Currently, the complexity of the algorithm mainly comes from the large scheduling graph. In the future, we would like to devise design space exploration algorithms where we can use SDFG directly. Also, our current algorithm relies on the numbers in the high-level synthesis reports. The reported number (latency, initiation interval and area usage) is a conservative estimation. For example, worst-case latency of the program is used in the synthesis report. However, at the run-time, latency of the modules can be data-dependent. Thus, we would also like to investigate algorithms that can handle such a data-dependent variance.

CHAPTER 4

Node-Level CPU-Accelerator Orchestration

4.1 Introduction

With the current ever-growing volume of data, the problem of efficiently processing such big data has attracted a lot of attention from both academia and industry [Whi12,ZCD12, ZCF10]. To fit the data into memory and to leverage multiple cores and servers, today's big data applications tend to distribute the datasets into multiple partitions where partitions can be processed in parallel [Bor08].

Whereas a data-partition approach can accelerate big data applications on multicore CPUs, using FPGA accelerators is a more attractive solution since it helps to address the limited scaling of general-purpose CPUs. Recently, FPGA acceleration for big data applications have stimulated a lot of researches due to FPGAs' low power, high performance and energy efficiency [PCC14, BRH15, CDL13]. However, most prior studies mainly focused on the FPGA accelerator design itself and did not consider efficient CPU and FPGA co-optimization, which we find can be the key to the performance of such applications. In this chapter we aim to answer one key question: *How should the multicore CPU and FPGA coordinate together to optimize the performance of big data application?*

Through our experiments We find that although we get a high speedup on kernel computation by offloading the computation to FPGAs, the overall application speedup we can achieve will be much smaller when comparing with multi-threaded CPU implementation. The major reason is that current application execution model fails to fully utilize the system resources such as CPU cores and I/O. More specifically, when computation is offloaded to the FPGA, the CPU threads wait for the accelerator to finish and thus are idled.

Therefore, we propose a dataflow execution model and an interval-based scheduling

algorithm to effectively orchestrate the computation between multiple CPU cores and the FPGA, which greatly improves the overall system resource utilization. Our experiments show that for pure CPU execution, the dataflow execution model provides a similar performance as the original data-parallel execution model. However dataflow execution model outperforms data-parallel execution model when FPGA is integrated into the system.

The remainder of this chapter is organized as follows. Section 4.2 presents our dataflow execution model and its corresponding runtime. A case study on an in-memory sorting routine is presented in Section 4.3 to validate our approach. A few more case studies are presented in Section 4.4. We leave the hardware design details used throughout this chapter in Section 4.5. Finally we conclude in Section 4.6.

4.2 CPU-FPGA Co-Scheduling

4.2.1 Dataflow Execution Model

Since data is partitioned in big data applications, a common approach to accelerate the application is through data parallel execution where multiple tasks of the same kernel are launched in parallel. However such a execution model often fails exploit all the system resource (CPU, FPGA and IO) since the running tasks are of the same kind and the performance may be bounded by a single resource while other resource is less used or idled.

To make use of the CPU cycles that are saved from the FPGA acceleration and to better utilize IO, we propose to use a dataflow execution model. Each application is divided into several stages. Each stage can have multiple tasks that leverage data-level parallelism and all the stages are connected through in-memory data queues and work in a pipelined fashion. The dataflow execution model here is similar to the actor-based streaming execution model discussed in Chapter 3, where each stage corresponds to each actor in the synchronous dataflow and can process streams of data.

To execute a dataflow program, the number of threads allocated to each stage needs to be decided. Slower stages deserve more CPU threads, while faster stages need fewer CPU threads. Besides the computation complexity of the stage, factors like disk bandwidth

and data format play important roles in determining a stage's performance and thus the efficiency of the entire dataflow. For example, SSDs typically provide a higher bandwidth than HDDs; therefore, if the input data resides on SSD instead of HDD, the performance of the read stage will be improved. Finally if computation is offloaded to FPGA, CPU will be less utilized and thus other CPU-sensitive stages (like sort) may run faster. Therefore, it is nontrivial to determine the best thread allocation for a dataflow program.

In a summary, the thread allocation problem can be formulated as follows: given N threads/cores on the compute platform and given m stages in an application, decide x_{it}, the number of threads allocated for each stage i at any time t, and decide y_{it}, the number of threads allocated for each stage i that shall use FPGAs, so that $y_{it} < x_{it}$ and $\sum_i x_{it} = N$ at any time t and the application can finish in the shortest amount of time. At the same time, the total resource that is used by all the tasks should not exceed the system resource. Denote r_{ik}^c as the amount of the kth resource that is used by a task in stage i when it is executed on CPU, and denote r_{ik}^f as the amount of the kth resource that is used by a task in stage i when it is executed on FPGA. The following constraints should stay true at any time t for each resource k: $\sum_i r_{ik}^c \cdot (x_{it} - y_{it}) + r_{ik}^f * y_{it} = R_k$, where R_k is the total resource of type k.

4.2.2 Proposed Runtime Thread Allocation Strategy

At runtime, we profile the CPU utilization ($util_i$) of stage i every small period of time. It is calculated as the actual thread time spent on each task divided by the total thread time of all tasks in this stage. Time that is spent on reading data from the input queue (dequeue) and writing data to the output queue (enqueue) is not counted into the actual thread time. Therefore, a high *util* represents that a stage is making high use of its allotted CPU resource, while a small *util* represents that a stage might be wasting time on dequeue/enqueue and thus is not making full use of its allotted CPU resource.

Our runtime adaptive thread allocation algorithm monitors the CPU utilization of each stage, and makes adjustments in thread allocation every period of time which moves CPU threads from the stages with a lower *util* to the stages with a higher *util*. Empirically we determine the number of threads that are moved from a faster stage to a slower stage,

δ, as follow:

$$\delta = n_f - \lceil n_f/(\frac{u_f + u_s}{2 \cdot u_f})\rceil, \tag{4.1}$$

where n_f represents the current number of threads allocated to the faster stage, and u_f and u_s represent the CPU utilization of the faster stage and slower stage respectively. u_f is smaller than u_s since the faster stage should have lower CPU utilization. $\frac{u_f+u_s}{2\cdot u_f}$ represents the gap between the CPU utilization of the faster stage and the average CPU utilization of these two stages ($\frac{u_f+u_s}{2}$), based on which we determine the number of reallocated threads (δ).

The interval of thread re-allocation should be long enough so that the current thread allocation policy have taken effect. This is due to the fact that when we decrease the number of threads for a stage, we wait for the allocated threads from the previous iteration to finish instead of killing them. Therefore the CPU utilization statistic that is collected right after thread re-allocation may not represent the effectiveness of the new thread allocation policy. Empirically the interval to sample CPU utilization is approximately equal to the task time and thread reallocation is executed when tens of tasks have finished in the slowest stage.

4.3 A Case Study of In-Memory Samtool Sorting

Throughout this section we use an example of the sort routine in Samtools [sam] to illustrate the common issues in integrating FPGA solutions into multi-threaded CPU computations, present our observations and experiment results.

4.3.1 Samtool In-Memory Sorting

Samtool sorting takes a genomic sequencing file as an input, sorts read alignment by leftmost coordinate or by read name, and finally outputs the sorted read alignments to a compressed file, so that latter processing tools can easily identify the duplicate read alignments in the genome.

Figure 4.1 presents an overview of the algorithm used in Samtool sorting [sam]. First,

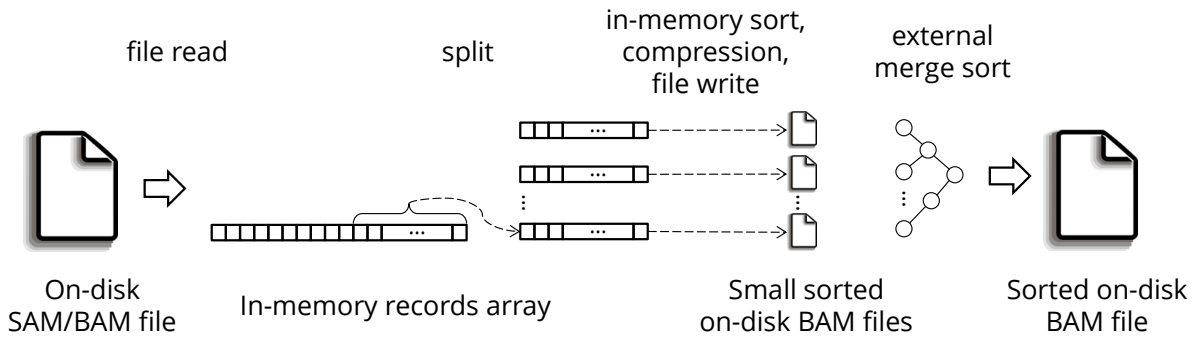


Figure 4.1: An overview of the sort routine in Samtools.

it sequentially reads the on-disk file that records sequencing reads in either normal SAM (Sequence Alignment/Map) text or block-compressed BAM (Binary Alignment/Map) format. Second, it splits the read alignments into multiple partitions, so that each partition can be sorted in memory, compressed, and written to a small temporary sorted file in parallel. Compression is applied at this stage for each data block (e.g., 64KB size) within the partition before writing to the disk so as to save storage space and bandwidth. In addition, a cyclic redundancy check (CRC) code is computed on each original uncompressed data block, which will be used to detect file errors when the compressed file is read in the future. Third, all temporary sorted files will be merged together into a single sorted BAM file using external merge sort, which is mainly disk bound. In this paper we focus on the optimization of the first two stages, sequential read and parallel partition processing, which occupy around 50% execution time of the entire Samtool sorting. We call these two stages *in-memory Samtool sorting*.

4.3.2 Experiment Setup and Initial Profiling

The software in-memory Samtool sorting [sam] runs on a 12-core Intel Xeon CPU E5-2620 (@2.40GHz) with CentOS 7.2. This server has 128GB memory and 500GB SSD. The input data samples used in the experiments are the high-coverage exome samples from the 1000 Genome project.[1] For illustration purposes, we use a 27.6 GB SAM file chopped from the first segment of the third exome sample throughout this paper

[1] Data can be downloaded from:

- http://www.internationalgenome.org/data-portal/sample/NA12878
- http://www.internationalgenome.org/data-portal/sample/NA12892
- http://www.internationalgenome.org/data-portal/sample/HG01500

unless otherwise specified. To generate the input SAM files for Samtool sorting, we use `bwa-mem` [Li13, BWA] to align these input exome samples.

Based on our profiling on the single-thread in-memory Samtool sorting, the compression and CRC algorithms, which are well suited for FPGA acceleration [FKB15, AHS14, Wal07, HLW15], occupy around 45% of the execution time. (More profiling results will be presented in Figure 4.4 in Section 4.3.5.) This motivates us to design an FPGA accelerator for compression and CRC. We design our accelerator with Vivado HLS and SDAccel (v2016.1). The default FPGA board is Xilinx Kintex UltraScale KU115. We also synthesized our design on the Alpha-Data ADM-PCIE-KU3 board to demonstrate portability of our design.

4.3.3 Accelerator Design and Performance

There are already several studies that accelerate compression and CRC on FPGAs [FKB15, AHS14, Wal07, HLW15]. We implement an FPGA compression and CRC accelerator design similar to these works. The major difference from these works is that we design our accelerator in HLS, which is portable and maintainable across Xilinx FPGAs.

Our FPGA accelerator takes a byte array as input. It computes the cyclic redundancy check (CRC) code of the input array and produces a compressed byte array. The produced CRC code is the same as the result from Linux *crc*32() in *zlib.h*. Our FPGA accelerator can process 16 bytes/cycle at 200 MHz, achieving a theoretical peak bandwidth of 3.2 GB/s. Table 4.1 shows the FPGA kernel resource utilization on the Xilinx KU115 board and ADM-PCIE-KU3 board, respectively.

Table 4.1: FPGA resource utilization of compression and cyclic redundancy check accelerator.

	LUTs %	FF %	BRAM %	DSP %	frequency (MHz)
KU115	12.6	4.9	9.0	0	200
KU3	16.6	7.6	13.2	0	200

The measured performance of our accelerator kernel from Xilinx OpenCL runtime is 2.8 GB/s. The gap between our measured throughput and the theoretical throughput

(3.2 GB/s) is because the data transfer between FPGA DRAM and FPGA kernel does not achieve a perfect pipeline initial interval (II) that equals to 1, since each DRAM burst read/write includes non-payload data overhead.

We test the compression ratio of our accelerator under the Calgary Corpus dataset [cal]. We are only able to achieve a compression ratio of 1.73 (geometric mean) across the dataset, lower than the previous work, but still at a comparable level. The reason of a lower compression ratio is mainly due to the history string matching loss when there are hash conflicts to the same dictionary. Unlike RTL designs in [FKB15] where doubled frequency is used for hash table, in HLS we do not have the flexibility of using different clocks in a single design.

We compare our compression throughput and ratio to two recent studies [FKB15, AHS14] and the single-core CPU version in Table 4.2. Although we see room to further optimize our accelerator design (*e.g.,* replacing some modules with RTL designs with doubled frequency), we did not pursue along that direction since its performance is already limited by the CPU-FPGA data transfer bandwidth.

Table 4.2: FPGA accelerator comparison.

Design	theoretical and measured throughput (GB/s)	compression ratio
This work	3.2 / 2.8	1.73
Altera OpenCL [AHS14]	2.8 / -	2.17
Microsoft RTL [FKB15]	5.6 / -	2.09
CPU [FKB15]	- / 0.05	2.62

To the best of our knowledge, this is the first compression (with CRC) design using Xilinx HLS.

4.3.4 FPGA Accelerator Integration with CPU

Before we integrate our FPGA accelerator with the CPU, we first measure the effective CPU-FPGA communication bandwidth, which is an important aspect in system perfor-

mance as observed in [CCF16]. According to our bandwidth tests, we propose a bandwidth optimization technique on OpenCL buffer reuse. Then we present our CPU-FPGA integration and evaluate its performance.

4.3.4.1 CPU-FPGA Communication

Similar to [CCF16], we measure the PCIe read and write bandwidth using OpenCL APIs $clEnqueueReadBuffer()$ and $clEnqueueUnmapMemObject()$, respectively.[2] All these APIs include an additional *memcpy* between the PCIe address space and the user application address space.

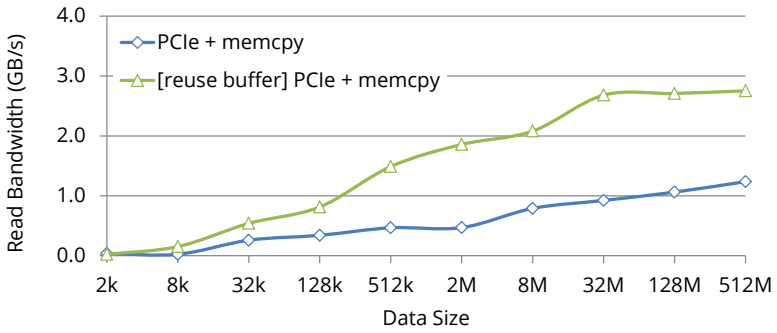
First, we find that the data transfer rate is quite low in the Xilinx SDAccel (v2016.1) environment, up to 2.2GB/s only. Moreover, we find that in the Xilinx OpenCL runtime, reusing those memory objects across accelerator invocations improves the PCIe read and write bandwidth by up to 2x, indicating the need for a buffer reuse mechanism. Figure 4.2 presents our experiment results on effective data transfer bandwidths. Detailed explanations and corresponding solutions are presented below.

1. Although the pure PCIe data transfer rate can reach up to 4-5 GB/s in our platform, when we count in the time of memory copy between application threads and OpenCL runtime which is unavoidable in the application, we are not able to achieve concurrent read/write data transfer rates that are higher than 2.3 GB/s. Given that the throughput of our accelerator is already higher (3.2 GB/s) than 2.3 GB/s, any better accelerator designs will not help performance in the current setup.

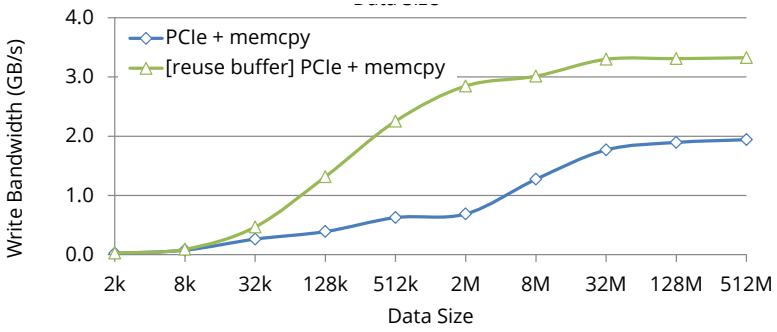
2. The original BAM file is in a block-compressed format, where each uncompressed block has the maximum size of 64 KB. However data transfer rates are below 1.0 GB/s when the data size is 64 KB. To solve this problem, we redefined the maximum size of uncompressed blocks in BAM format to 32 MB; at this size the data transfer rates are much higher.

3. At OpenCL runtime, an OpenCL memory object is first created and allotted to the kernel. Figure 4.2 demonstrates that reusing OpenCL memory objects can increase

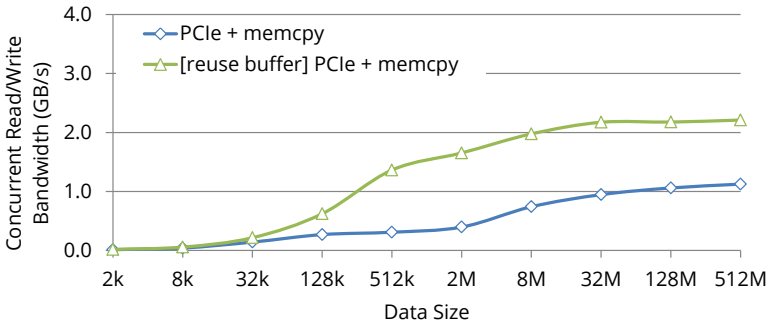
[2]The reason that we do not use $clEnqueueMapBuffer()$ to perform PCIe read is that Xilinx has confirmed that this API has degraded performance on CentOS 7.2 in SDAccel 2016.1.



(a) Read bandwidth.



(b) Write bandwidth



(c) Concurrent read/write bandwidth

Figure 4.2: Effective data transfer bandwidth between CPU and FPGA through PCIe.

the bandwidth between host CPU and FPGAs. We find that the 'PCIe + memcpy' can achieve up to 1.2 GB/s read bandwidth and 1.9 GB/s write bandwidth. The concurrent read and write bandwidth is as slow as 1.1 GB/s. However if we reuse the OpenCL memory object in the consecutive bandwidth tests instead of allocating new memory objects, we observe a significant higher bandwidth. The read bandwidth and write bandwidth can achieve up to 2.8 GB/s and 3.2 GB/s; concurrent read/write bandwidth is 2.2 GB/s. Given this observation, we implement a runtime OpenCL block reuse mechanism, which is presented in Section 4.3.4.2. Note that concurrent read/write bandwidth is slower than our kernel throughput (2.8 GB/s as measured), which becomes the limiting factor for us in achieving the expected kernel performance.

4.3.4.2 FPGA Runtime and Data Management

To handle efficient sharing of the FPGA among multiple application threads, we leverage the Blaze runtime system [HWY16], which is an open-source project that offers accelerator management at node-level and at cluster-level. At node-level, the Blaze runtime system serves as an abstraction layer between the application threads and the underlying FPGAs. This additional layer maintains all information about tasks, kernels, kernel arguments, and device data blocks, enabling more sophisticated management (e.g., task queueing, thread-level fairness, and automatic FPGA reconfiguration) than that of the default OpenCL runtime. We will discuss more on the cluster-level aspect of the Blaze system in Chapter 5.

In the original Blaze, each accelerator invocation implicitly creates new OpenCL memory objects that are used as the input and output of the kernel. In this paper we improve Blaze by including a data management module that provides a runtime OpenCL block reusing mechanism, so as to improve the effective CPU-FPGA communication bandwidth. Our data managment module is explained as follows.

Instead of releasing the OpenCL memory objects from previous runs, we maintain these objects in a lookup table as long as we do not run out of device memory space. New memory object allocations will first perform table lookups to see if there are already allotted objects that are large enough to hold the current ones; failure to find pre-allocated

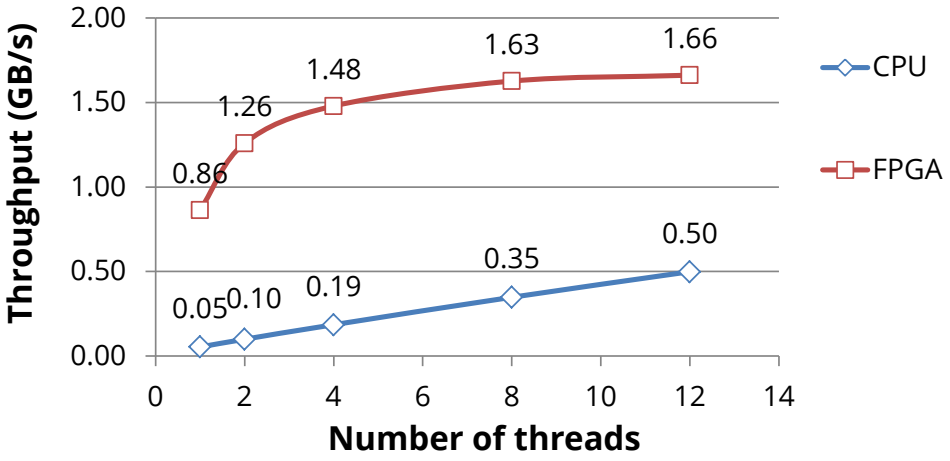


Figure 4.3: Compression and CRC task throughput measured from host CPU side. Measured FPGA performance includes data transfer time between the CPU and the FPGA.

objects results in new memory objects being allocated, and old memory objects being released if we run out of space.

4.3.4.3 Integrated CPU-FPGA Performance

Figure 4.3 presents the compression (with CRC) task performance in the multi-threaded scenario. In this experiment, a total of 553 compression tasks are executed. We can observe that the CPU performance increases almost linearly with the increasing number of threads. Task performance on FPGA is measured from the host CPU side, which includes the data transfer between the host CPU and FPGA DRAM through PCIe. Therefore, the observed single task performance is only 0.86 GB/s, rather than 3.2 GB/s. Communication and computation are overlapped when more threads are launched. So the FPGA task performance increases slightly with the increasing number of threads. The peak FPGA task performance achieved is 1.7 GB/s; it is limited by PCIe communication bandwidth rather than the FPGA kernel computation throughput.

Compared to the CPU compression (with CRC) implementation, our FPGA implementation achieves 17.2x speedup under the single-thread scenario, and achieves 3.3x speedup in the 12-threaded scenario.

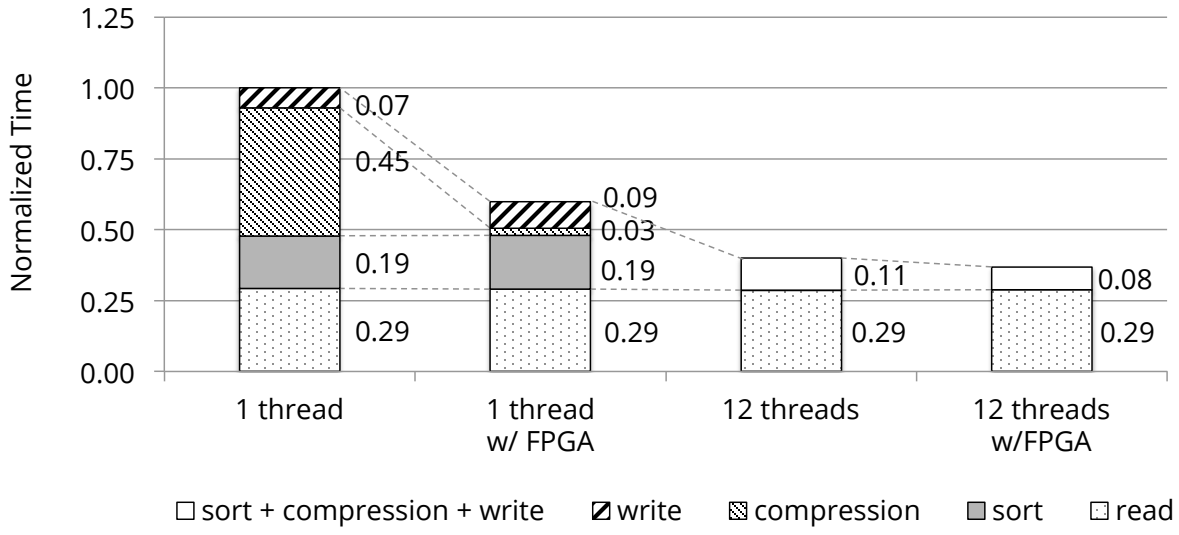


Figure 4.4: Normalized time for in-memory Samtool sorting.

4.3.5 Performance of Samtool Sorting

Finally, Figure 4.4 summarizes the runtime breakdown for in-memory Samtool sorting with and without FPGA acceleration. During the in-memory sorting phase, data is first read into memory. Then parallel threads are launched; each thread sorts a chunk of data, compresses it and writes the data to a file. When multiple threads are used, we do not add extra synchronization among threads after sorting or compression. Therefore, we cannot tell the exact time that is spent on sorting, compression or file write; instead, the total time of these three steps is reported in the figure.

Looking at Figure 4.4, we can see that in the single-thread scenario, we achieve 17.2x speedup on the compression kernel by using the FPGA accelerator, 2.3x speedup on 'sort+compress+write', and 67% overall performance improvement. Note that file write time increases slightly since our FPGA compression ratio is smaller than that of the CPU baseline.

However, comparing the two rightmost columns in Figure 4.4 where 12 threads are used in the CPU baseline, we can only observe 1.4x speedup on 'sort+compress+write'. This is because 'sort' and 'compress' are well-parallelized in this baseline. The application performance is now limited by the sequential read stage and there is a marginal of 8% overall performance improvement by integrating the FPGA accelerator into the system.

To better understand the optimization opportunities in the 12-thread Samtools in-

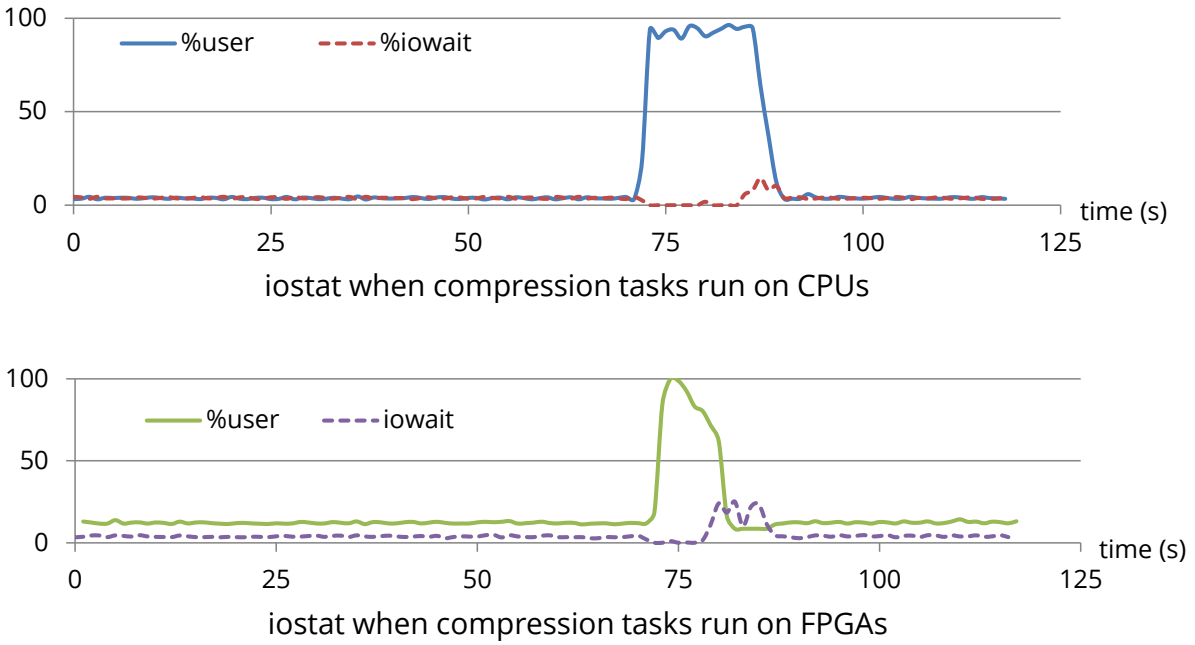


Figure 4.5: System iostat during 12-thread in-memory sorting.

memory sorting with and without FPGAs, we print out the CPU user utilization and iowait from the linux 'iostat' command in Figure 4.5. During the read phase, the CPU utilization is low; it stays below 5% in our CPU baseline, and it stays around 12% when FPGAs are being used. The increased utilization comes from the Blaze runtime system. Peaks in the CPU utilization graphs represent the *sorting + compression* phase, followed by the *write* phase where there is a spike of iowait. When compression is offloaded to FPGA, the CPU utilization quickly drops after the *sorting* phase. Finally, there is little iowait except in the write phase, which indicates that the IO bandwidth could be underutilized for the read, sorting, and compression phases.

Looking at Figure 4.5, we also find that although using the FPGA accelerator does not make a significant improvement on application performance, a large amount of CPU cycles are saved. However the current execution model cannot make good use of these saved CPU cycles and CPU is just wasting time waiting for the accelerator to finish. This motivates us to re-design the Samtool in-memory sorting.

4.3.6 Parallelizing the Read Stage

Following today's trends in big data processing, we first partition our input SAM files into multiple smaller files so that file read can be parallelized and IO bandwidth can be

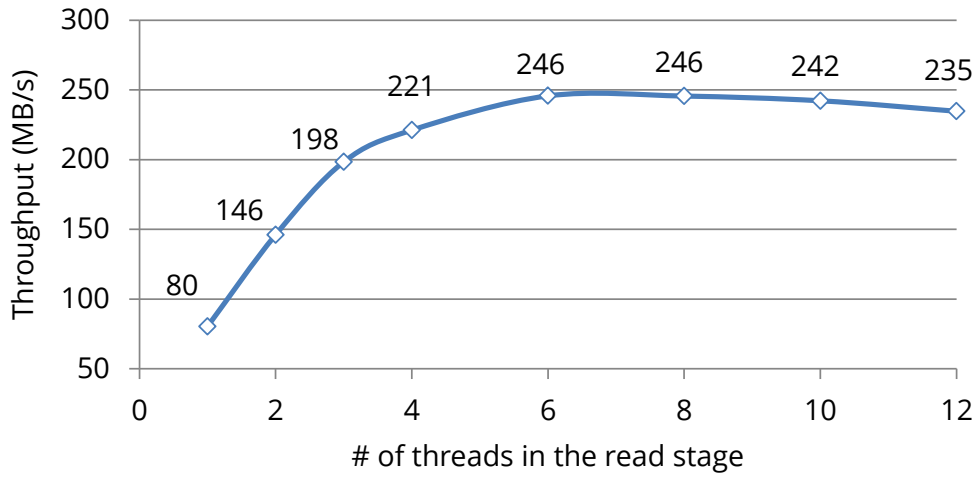


Figure 4.6: SAM file read throughput on SSD.

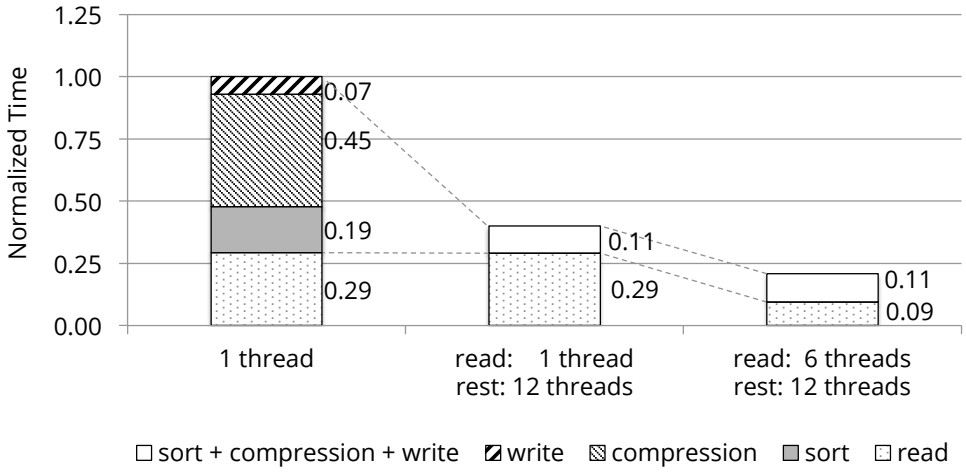


Figure 4.7: Normalized execution time for in-memory Samtool sorting using different number of threads.

better-utilized. Note that in real usage this file splitting process can be easily omitted by modifying `bwa-mem` [Li13, BWA], the computation step right before the Samtool sorting in the genome processing pipeline, to output multiple files instead of one file.

Figure 4.6 presents the parallel SAM file read throughput using different number of threads. We find that in our system, increasing the read threads up to 6 can improve SAM file read throughput; using a thread number beyond 6 does not provide any benefit and can even slow down the overall read throughput, since 6 threads has already reached the IO bandwidth limit.

Figure 4.7 presents the performance of in-memory Samtool sorting using the best configuration on CPU: using 6 threads for read, and 12 threads for latter stages. Compared

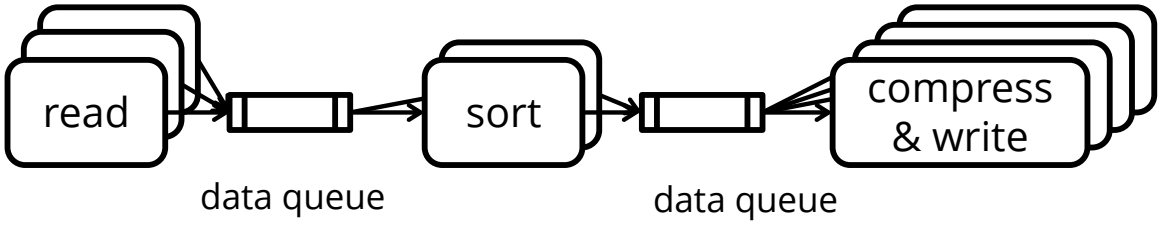


Figure 4.8: A dataflow model for in-memory sorting. In this example, 3, 2 and 4 threads are used to execute *read*, *sort* and *write* stages, respectively. Data between stages are organized using multi-input and multi-output queues.

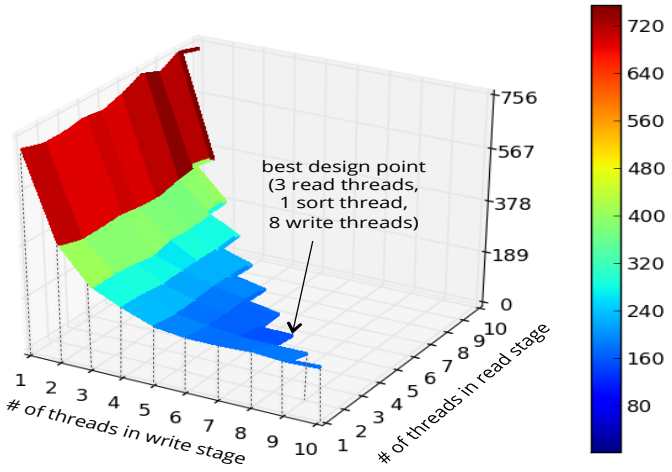
to the original 12-thread version (the middle column), parallelizing the read stage can improve the overall performance by 2x.

4.3.7 Dataflow-Samtools

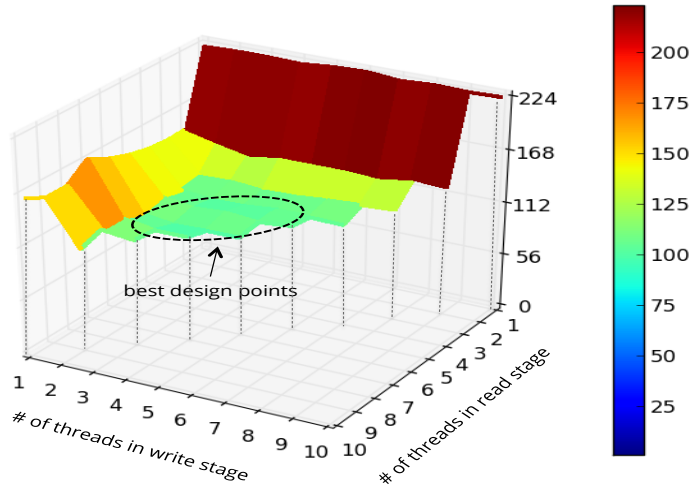
We model Samtool in-memory sorting using the proposed dataflow execution model as shown in Figure 4.8. We divide the entire application into three stages: *read*, *sort* and *compress*+*write* (we denote it as *write* in the rest of this paper). We name our dataflow implementation of Samtool sorting as **Dataflow-Samtool** sorting.

Figure 4.9 presents our design space explorations on thread allocations. It shows the effect of thread allocation on the performance of Dataflow-Samtool sorting. There are a total of 12 threads running, and we distribute them to read, sort and write stages. Overall Figure 4.9b shows a better performance than Figure 4.9a since FPGA accelerators are used. We found that when FPGAs are not used (Figure 4.9a), the number of threads in the write stage largely determines the performance. This is because compression and CRC in the write stage are the most computation-intensive. When FPGAs are in use (Figure 4.9b), the affinity between the number of threads in the write stage and application performance is no longer prominent.

In practice, it is not always possible to explore the whole design space to determine the best thread allocation. It takes over 8 hours to collect the application performances on all design points in Figure 4.9 using our small input data sample of 27.6 GB. This motivates us to design a runtime adaptive thread allocation strategy as described below.



(a) Performance on 12-core CPUs. At the best design point, the application time takes 145s.



(b) Performance on 12-core CPUs with FPGA acceleration on compression (and CRC). There are several design points that achieve similar good performance. The application takes around 100s in these design points.

Figure 4.9: Design space exploration on thread allocation for Dataflow-Samtool sorting. The number of threads in *sort* stage equals to $(12 - \#\ of\ threads\ in\ read\ stage - \#\ of\ threads\ in\ write\ stage)$. Since we need to maintain at least one thread for each stage, the range of x and y axes are $[1,\ 10]$. The z axis shows the execution time in seconds using different colors.

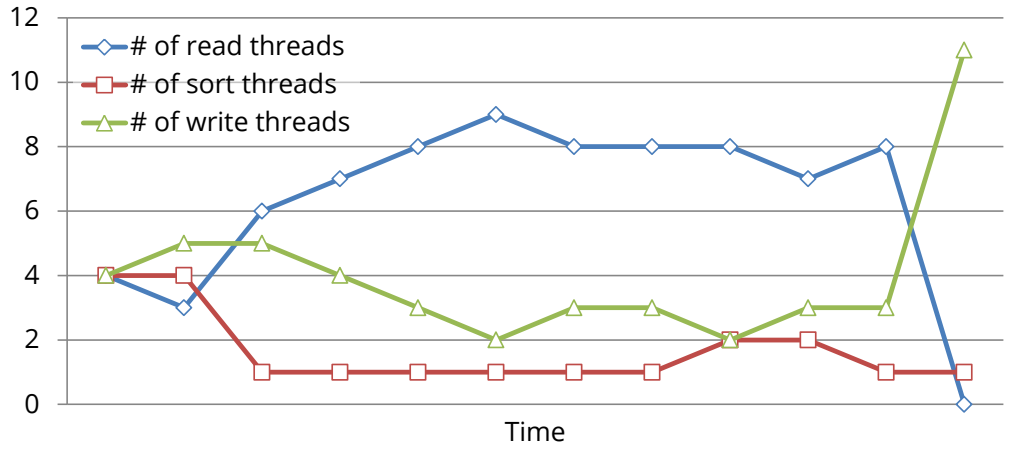


Figure 4.10: Runtime adaptive thread distribution on our CPU-FPGA platform.

4.3.7.1 Evaluation on Runtime Thread Allocation

In our experiments, sorting a 200GB SAM file takes about half an hour. The *util* is updated every 2 seconds and the thread configurations are updated every 20 seconds. Note that for our smaller experimental datasets, which only take a few minutes to sort, the *util* is updated every 0.5 seconds and the thread configurations are updated every 5 seconds. Note that how to decide these parameters for general applications was discussed in 4.2.2.

Figure 4.10 presents our adaptive thread distribution during runtime on the CPU-FPGA co-optimization case. We start the application with an equal distribution of threads among all stages. After several rounds of tuning, we can see that the thread allocation algorithm tends to use about 8 read threads, 1 sort thread and 3 write threads. At the end, when the read stage finishes, all the read threads are allocated to the write stage.

Table 4.3 presents the effectiveness of our runtime adaptive thread allocation. We compare the application execution time using adaptive thread allocation to that of the best and worst static thread allocations. Application execution time using static thread allocation are shown in Figure 4.9 with all thread configurations explored, where the best and worst configuration can lead to a performance difference by 5X or 2.2X on CPU-only and CPU-FPGA platforms, respectively. Compared to the best static thread allocation, we found that the runtime adaptive thread allocation can provide a performance very close (93%) to the best static allocation on the CPU platform. Moreover, it can even

outperform the best static allocation on the CPU-FPGA platform. This is because our runtime thread allocation strategy provides a more flexible thread configuration than static allocations. For example, when a stage finishes, its threads can be reallocated to other stages.

Table 4.3: Comparisons between static thread allocation and adaptive thread allocation on application time (seconds).

Application time (s)	Static best	Static worst	Dynamic
CPU-only	145	734	155
CPU-FPGA	100	220	97

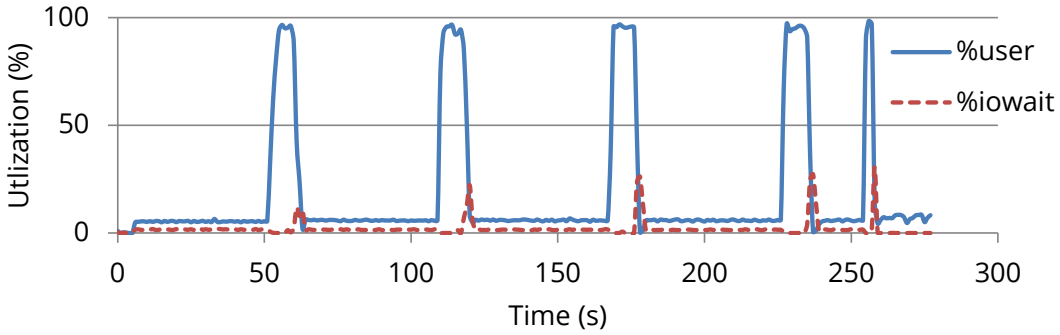
4.3.8 Overall Performance

4.3.8.1 Comparison on System Utilization

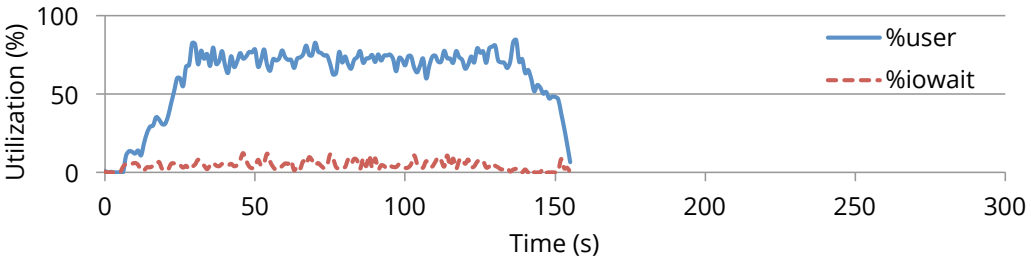
Figure 4.11 presents the overall performance and system metrics comparison on Samtool in-memory sorting using 12-threads. The original Samtool library takes 275s (Figure 4.11a), while our dataflow-Samtool finishes in 155s (Figure 4.11b) and is further improved to 97s when the FPGA is utilized (Figure 4.11c). In Figure 4.11a, different phases such as read and sort, compression and write are obvious; CPU utilization is extremely low during read phases. Note that multiple iterations are needed to process the entire dataset. This application does a poor job of making use of the computation resources.

We rewrite the Samtool library code using a dataflow model, and Figure 4.11b reports an immediate speedup of 1.9x. Our dataflow-Samtool overlaps read, sort and write stages; therefore, the CPU utilization constantly stays around 70% to 80% during the execution.

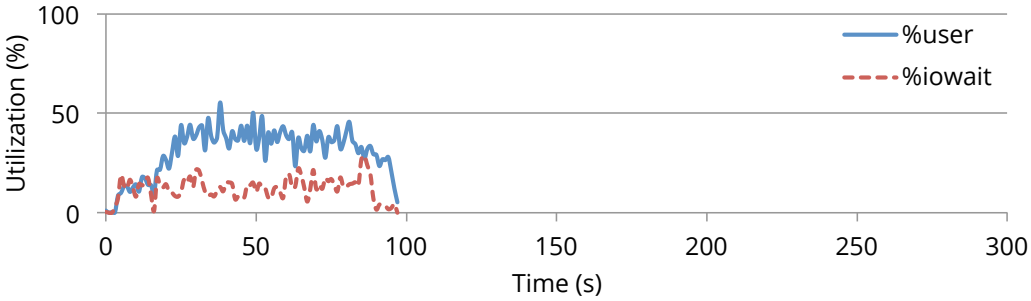
Figure 4.11c shows that by using our FPGA accelerator the application latency experiences further speedup by 1.4x, and in total achieves 2.6x speedup over the original Samtool library implementation. Note that at this point there is limited room for further improvement since the I/O has been the bottleneck in our dataflow-Samtool (there is a lot of iowaits in Figure 4.11c). Adding more SSDs to the system and direct output files



(a) Samtool in-memory sorting on the 12-core CPU. Application latency is 275s.



(b) Dataflow-Samtool in-memory sorting on the 12-core CPU. Application latency is 155s.



(c) Dataflow-Samtool in-memory sorting on the 12-core CPUs and FPGA. Application latency is 97s.

Figure 4.11: System iostat during 12-thread in-memory sorting in Samtool, dataflow-Samtool, and dataflow-Samtool-FPGA.

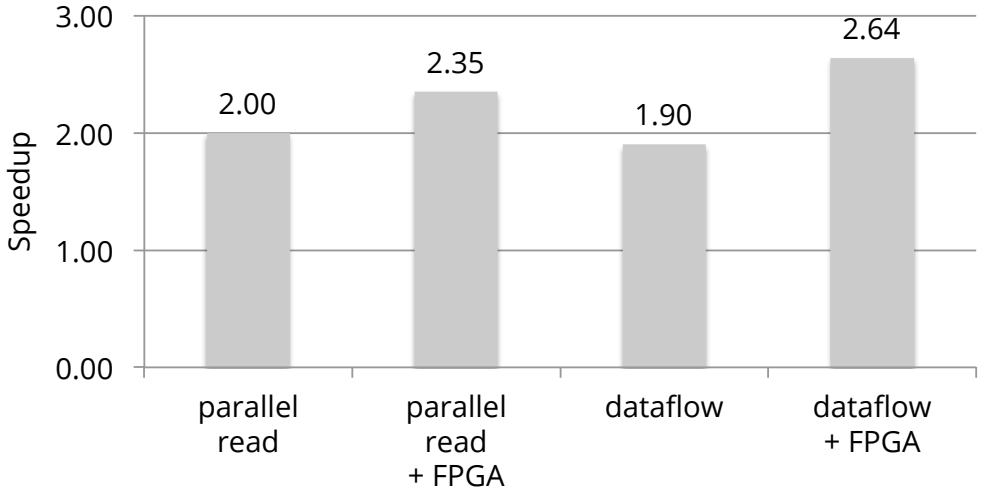


Figure 4.12: Overall speedup of different optimizations over original 12-thread Samtool in-memory sorting.

to different SSDs could alleviate this problem.

4.3.8.2 Comparison of Different Optimizations

We compare the performance of different optimizations over the original 12-thread in-memory Samtool sorting in Figure 4.12:

1. **parallel read** parallelizes the read stage using 6-threads, and then executes the sort+compression+write stage using 12 threads;
2. **parallel read + FPGA** is based on parallel read but uses FPGA to perform compression;
3. **dataflow** is the 12-threaded CPU implementation of Dataflow-Samtool;
4. **dataflow + FPGA** is the 12-threaded CPU implementation of Dataflow-Samtooldatflow-Samtool with FPGA.

There are several observations. First, dataflow does not out-perform parallel read. The major reason is that there is additional memory consumption is maintaining data queues in the dataflow model which slows down the memory system. Thus the dataflow exeuction model is slightly slower than parallel read.

Second, dataflow execution model is more FPGA-friendly than data-parallel execution model (parallel read). Because it can overlap FPGA execution with other CPU intensive tasks, dataflow execution model brings more significant speedup when FPGAs are introduced into the system.

Finally, among all four optimizations, CPU-FPGA co-optimized dataflow-Samtool achieves the best performance, which is 2.64x faster than the original 12-thread Samtool sorting.

4.3.8.3 Performance for Different Datasets

Finally, we present the application performance on large datasets in Table 4.4. On average we achieve 2.6x speedup for genome data in-memory sorting.

Table 4.4: Application execution time for different datasets (minutes).

Data	Input size (GB)	number of records to sort	Samtools CPU	Dataflow Samtools w/ FPGA	speedup
SRR098401	54.5	2.28E+08	10.6	3.8	2.8x
SRR098359	56.7	2.37E+08	10.9	4.2	2.6x
SRR1295423	235.2	4.12E+08	34.3	14.0	2.5x

4.4 More Case Studies

In this section we perform more experiments to evaluate our dataflow execution model. We still use the Samtool in-memory sorting applications but different data format and storage are used in different experiments.

4.4.1 Changing Input format

We change the input format from SAM format to BAM format where data is stored as compressed binary file. This change indicates that input file size is reduced and decompression is needed during read stage. Reading files in BAM format is more computation-

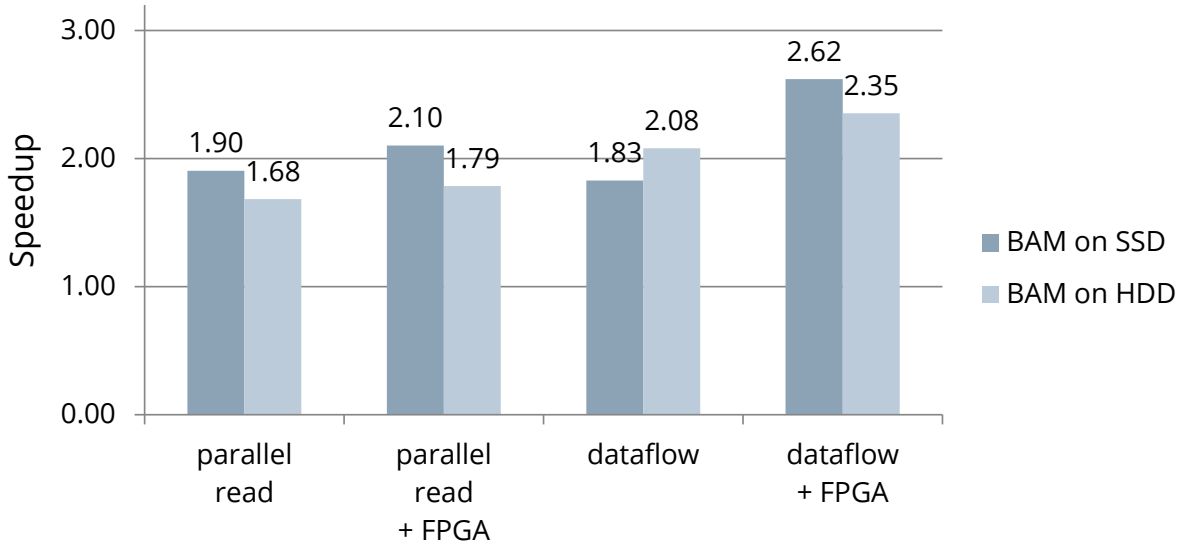


Figure 4.13: Overall speedup of different optimizations over original 12-thread Samtool in-memory sorting using different configurations.

intensive than reading files in SAM format. The results are presented in Figure 4.13 as *BAM on SSD*. In this case *dataflow + FPGA* achieve the best performance which is 2.62x better than the 12-threaded CPU baseline.

4.4.2 Changing Storage Type

We also test the another case where the input reside on HDD rather than SSD. Since parallel read on HDD is slower than parallel read on SSD, the application is more disk-bounded. Therefore overall we observe less speedup by using FPGA accelerators. However among all the execution models, dataflow execution with FPGA accelerator still performs the best (2.35x speedup) as shown in *BAM on HDD* in Figure 4.13.

4.5 Accelerator Designs

Finally we briefly review the our compression and CRC accelerator design that are used throughout this chapter.

There are many studies on accelerating compression on FPGAs due to its importance. For example, Altera has implemented a portable deflate compression algorithm using OpenCL and achieved a compression rate of 16 bytes/cycle at 193 MHz [AHS14]. The best result is achieved in the Microsoft implementation in RTL [FKB15], which achieves

a compression rate of 32 bytes/cycle at 175 MHz. Whereas most of our design is similar to the these designs [AHS14, FKB15] at high level, our work demonstrates that using Xilinx HLS can achieve a compression throughput similar to that implemented using Altera OpenCL [AHS14]. Compared to the RTL design [FKB15], it is more portable and maintainable. Theoretically, if HLS can support multiple asynchronous clocks in the design, we can use double clock frequency for our hash table design as done in [FKB15], and then we will achieve the same or similar compression ratio.

CRC is intrinsically binary polynomial division. Traditional CRC circuits are based on linear feedback shift registers [PZ92, MDM01], where retiming and pipelining have been studied to improve its performance. There are several recent studies on table-based algorithms to calculate CRC [HLW15, Wal07]. We apply the design concept in [HLW15] to our HLS design and achieve the same performance (16 bytes per cycle).

To the best of our knowledge, we are the first to focus on the efficient coordination between the multicore CPU and FPGA using a dataflow execution model.

4.6 Conclusions

In this chapter we present a dataflow execution model so as to better incorporate FPGA accelerators into the system. This model coordinate today's multicore CPU and FPGA together to optimize the performance of big data applications. It combines data-level parallelism on multicore CPU, hardware specialization on FPGA, and pipeline parallelism between CPU cores and FPGA. Accordingly, we developed an adaptive runtime to effectively orchestrate the computation between multiple cores and FPGA.

To validate our approach, we conduct a case study on in-memory Samtool sorting. We integrate an FPGA accelerator for compression and CRC into the in-memory sorting routine. We found that a straightforward integration of this FPGA accelerator into the 12-thread Samtool sorting only achieved a marginal 8% system throughput improvement. With our proposed dataflow execution model, we can achieve an average of 2.6x system performance speedup over the original 12-thread in-memory Samtool sorting.

CHAPTER 5

Cluster-Level Resource Management

5.1 Introduction

Modern big data processing systems, such as Apache Hadoop [had] and Spark [ZCF10], have evolved to an unprecedented scale. As a consequence, cloud service providers, such as Amazon, Google and Microsoft, have expanded their datacenter infrastructures to meet the ever-growing demands for supporting big data applications. However, due to the problem of dark silicon [EBA11], simple CPU core scaling has come to an end, and thus CPU performance and energy efficiency has become one of the primary constraints in scaling such systems. To sustain the continued growth in data and processing methods, cloud providers are seeking new solutions to improve the performance and energy efficiency for their big data workloads.

Among various solutions that harness GPU (graphcs processing unit), FPGA (field-programmable gate array), and ASIC (application-specific integrated circuit) accelerators in a datacenter, the FPGA-enabled datacenter has gained increased attention and is considered one of the most promising approaches. This is because FPGAs provide low power, high energy efficiency and reprogrammability to customize high-performance accelerators. One breakthrough example is that Microsoft has deployed FPGAs into its datacenters to accelerate the Bing search engine with almost 2x throughput improvement while consuming only 10% more power per CPU-FPGA server [PCC14]. Another example is IBM's deployment of FPGAs in its data engine for large NoSQL data stores [BRH15]. Moreover, Intel, with the $16.7 billion acquisition of Altera, is providing closely integrated CPU-FPGA platforms for datacenters [int], and is targeting the production of around 30% of the servers with FPGAs in datacenters by 2020 [har].

With the emerging trend of FPGA-enabled datacenters, one key question is: *How can*

we easily and efficiently deploy FPGA accelerators into state-of-the-art big data computing systems like Apache Spark [ZCF10] and Hadoop YARN [VMD13]? To achieve this goal, both programming abstractions and runtime support are needed to make these existing systems programmable to FPGA accelerators. In this work, we mainly focus on runtime support, particularly resource management. Programming support for FPGA accelerator deployment can be found in [HWY16].

Managing FPGA accelerator resource at cluster-level is non-trivial. It usually takes several seconds to reprogram an FPGA into a different accelerator (with a different functionality). A frequent FPGA reprogramming in a multi-accelerator scenario can significantly degrade the overall system performance. This raises a fundamental question: *Do we manage "the hardware platform itself" or "the logical accelerator (functionality) running on top of the hardware platform" as a resource?*

To address this challenge, we propose the following solutions:

1. Policies for managing logical accelerator functionality —instead of the physical hardware platform itself—as a resource, where better scheduling decisions can be made to optimize the system throughput and energy efficiency.

2. An efficient runtime to share FPGA accelerators in data-centers, where an FaaS framework is implemented to support sharing of accelerators among multiple threads and multiple applications in a single node. Also, an accelerator-centric scheduling is proposed for the global accelerator management to alleviate the FPGA reprogramming overhead for multi-accelerators.

3. A prototype that is compatible with existing ecosystems like Apache Spark with no code changes and YARN with a lightweight patch.

5.2 System Overview

The Blaze runtime system integrates with Hadoop YARN to manage accelerator sharing among multiple applications. As illustrated in Figure 5.1, Blaze includes two levels of accelerator management. A global accelerator manager (GAM) oversees all the accelerator resources in the cluster and distributes them to various user applications. Node

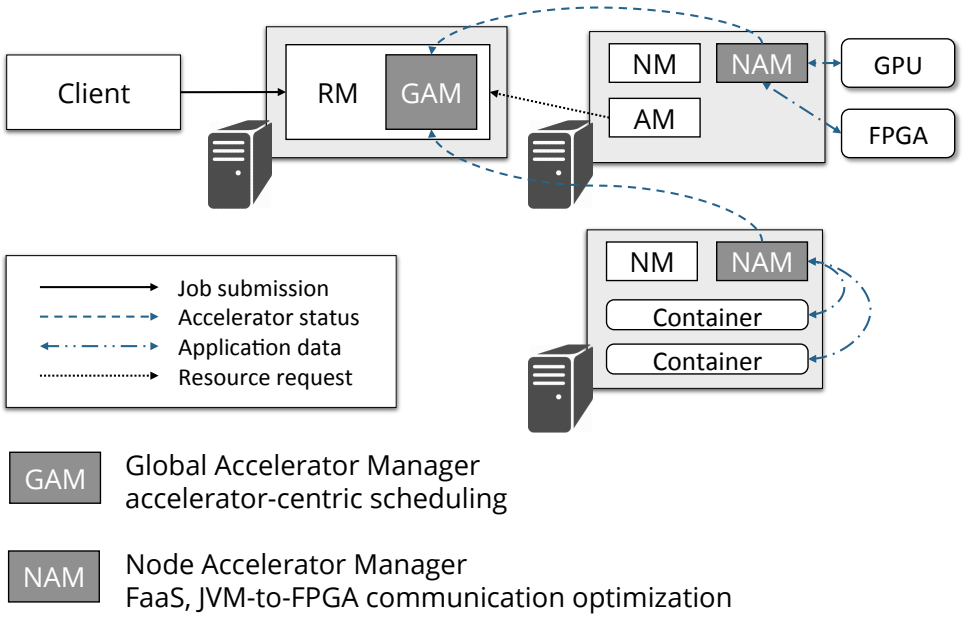


Figure 5.1: Overview of Blaze runtime system.

accelerator managers (NAMs) sit on each cluster node and provide transparent accelerator access to a number of heterogeneous threads from multiple applications. After receiving the accelerator computing resources from GAM, the Spark application begins to offload computation to the accelerators through NAM. NAM monitors the accelerator status, handles JVM-to-FPGA data movement and accelerator task scheduling. NAM also performs a heartbeat protocol with GAM to report the latest accelerator status. In this chapter, we focus on the design of GAM. Details of NAM can be found in [HWY16].

Blaze execution flow. Figure. 5.2 presents the execution flow of the Blaze system. During system setup, the system administrator can register accelerators to NAM through APIs. NAM reports accelerator information to GAM through heartbeat. At runtime, user applications request containers with accelerators from GAM. Finally during application execution time, user applications can invoke accelerators and transfer data to and from accelerators through NAM APIs.

Accelerator-centric scheduling. In order to solve the global application placement problem considering the overwhelming FPGA reprogramming overhead, we propose to manage the logical accelerator functionality, instead of the physical hardware itself, as a resource to reduce such reprogramming overhead. We extend the *label-based scheduling* mechanism in YARN to achieve this goal: instead of configuring node labels as 'FPGA', we propose to use accelerator functionality (e.g., 'KMeans-FPGA', 'Compression-FPGA')

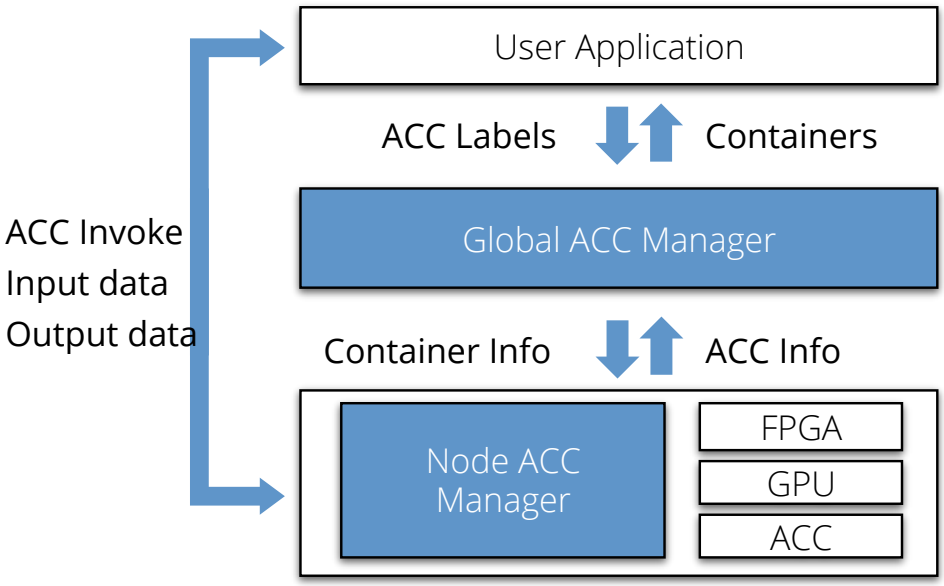


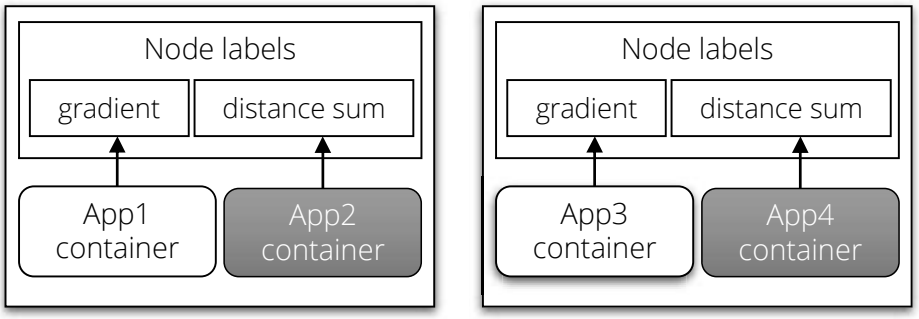
Figure 5.2: Blaze execution flow.

as node labels. This helps us to differentiate applications that are using the FPGA devices to perform different computations. We can delay the scheduling of accelerators with different functionalities onto the same FPGA to avoid reprogramming as much as possible. Different from the current YARN solution, where node labels are configured into YARN's configuration files, node labels in Blaze are configured into NAM through command-line. NAM then reports the accelerator information to GAM through heartbeats, and GAM configures these labels into YARN.

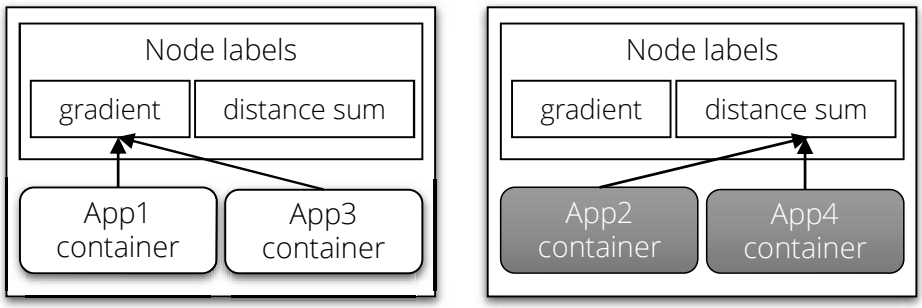
5.3 Global Accelerator Manager

In order to mitigate the FPGA reprogramming overhead, it is better to group the tasks that need the same accelerator to the same set of nodes. The ideal situation is that each cluster node only gets the tasks that are requesting the same accelerator, in which case FPGA reprogramming is not needed. Figure 5.3 illustrates that grouping accelerator tasks can reduce FPGA reprogramming overhead.

By managing logical accelerator functionality as a resource, we propose an accelerator-locality-based delay scheduling policy to dynamically partition the cluster at runtime, avoiding launching mixed FPGA workloads on the same cluster node as much as possible. During accelerator allocation in GAM, we consider the nodes in the following order as



(a) Naive allocation: applications on a node use different accelerators which leads to frequent FPGA reprogramming.



(b) Ideal allocation: applications on the node use the same accelerator and thus there is no FPGA reprogramming

Figure 5.3: Different resource allocation policies. In this example, each cluster node has one FPGA platform and two accelerator implementations, "gradient" and "distance sum". Four applications are submitted to the cluster, requesting different accelerators.

scheduling priorities: 1) the idle nodes that do not have any running containers; 2) the nodes that run similar workloads; 3) the nodes that run a different set of workloads. Specifically, we define an affinity function to describe ith node's affinity to an application as $f_i = \frac{n_{acc}}{n}$, where n_{acc} is the number of containers on this node that use the same logical accelerator (or label), and n is the total number of containers on this node. A node with higher affinity represents a better scheduling candidate. An idle node which has zero running containers has the highest affinity and is considered the best scheduling candidate. GAM tries to honor nodes with higher accelerator affinity by using the so-called delay scheduling.

At runtime, each NAM periodically sends a heartbeat to the GAM, which represents a scheduling opportunity. The GAM scheduler does not simply use the first scheduling

opportunity it receives. Instead, it may skip a few scheduling opportunities and wait a short amount of time for a scheduling opportunity with a better accelerator affinity. In our implementation, we maintain a threshold function for each application, which linearly decreases as the number of missed scheduling opportunities increases. A container is allocated on a node only if the node's accelerator affinity is higher than the threshold function.

5.4 Experiments

5.4.1 Experimental Setup

The experimental platform we use is a local standard CPU cluster with up to 20 nodes, among which 4 nodesare integrated with FPGA cards using PCI-E slots. Each server has dual-socket Intel Xeon E5-2620v3 CPUs with 12 cores in total and 64GB of main memory. The FPGA card is AlphaData ADM-PCIE-7V3, which contains a Xilinx Virtex-7 XC7VX690T-2 FPGA chip and 16GB of on-board DDR3 memory. The FPGA board can be powered by PCI-E alone and consumes around 25W, which makes it deployable into commodity datacenters.

The software framework is based on a community version of Spark 1.5.1 and Hadoop 2.6.0. The accelerator compilation and runtime are provided by the vendor toolkits. For the AlphaData FPGA cards, we use the OpenCL flow provided by the Xilinx SDAccel tool-chain, where the OpenCL kernels will be synthesized into bitstreams to program the FPGA.

We choose two iterative machine learning algorithms, logistic regression and K-means clustering, from Spark machine learning library.

Logistic regression (LR). The baseline LR is the training application implemented by Spark MLlib [mll] with the LBFGS algorithm. The software baseline uses netlib with native BLAS library. The computation kernels we select are the logistic gradients and the loss function calculation. The kernel computation takes about 80% of the total application time.

K-Means clustering (KM). The KM application is also implemented using Spark

MLlib, which uses netlib with native BLAS library. The computation kernel we select is the local sum of center distances calculation. The datasets used in KM are the same as LR, and the percentage of kernel computation time is also similar to LR.

The FPGA accelerators for all applications are designed in-house. The accelerator specifications for LR and KM can be found in [CHW16]. Table 5.1 presents an overview of the accelerator speedup compared to the 12-thread CPU software baseline in terms of throughput improvement. We set `--num-executors` to 1 and `--executor-cores` to 12 in Spark. The input data for LR and KM are based on a variant of the MNIST dataset [mni] with 8 million records, and is sampled such that on average each node will process 2-4GB of data.

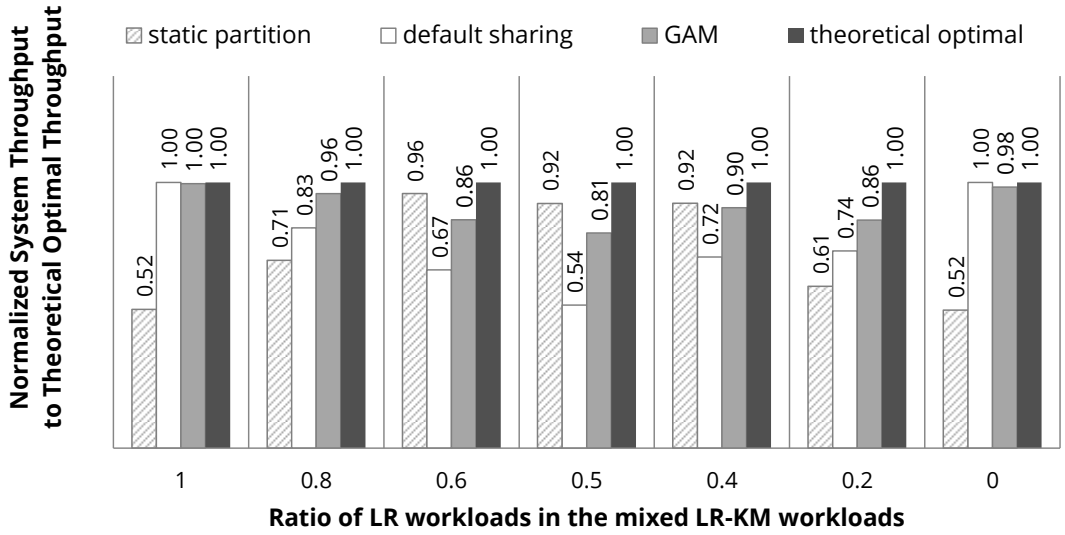
5.4.2 GAM Analysis

To evaluate the effectiveness of GAM's resource allocation policy (i.e., accelerator-centric scheduling), we choose seven sets of workloads on a 4-node CPU-FPGA cluster. Each set contains LR and KM applications of various input data sizes, and the ratio of these two applications varies among different sets of workloads.

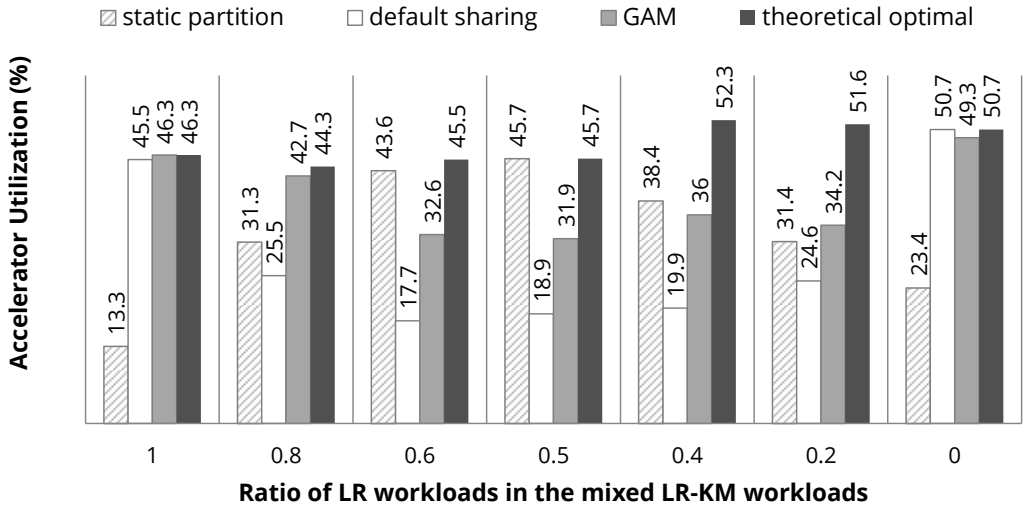
We compare GAM with two baselines: *static-partition* and *default sharing*. In *static partition*, we evenly partition the 4 nodes into two sets: 2 nodes only run LR applications and the other 2 nodes only run KM applications. Therefore, reprogramming never occurs in the experiments. In *default sharing*, all the FPGA nodes can run both LR and KM workloads, and we use the Apache YARN's default resource allocation policy. Our GAM has settings similar to *default sharing*, but uses our accelerator-centric scheduling policy. We also calculate the offline theoretical optimal scheduling results, in which case we assume that the submission time of all the sets of workloads are known beforehand and the submission time are close to each other. GAM can then intentionally delays the workloads

Table 5.1: FPGA accelerator performance profile.

Application	Kernel	Speedup
LR	Gradients	3.4×
KM	DistancesSum	4.3×



(a) System throughput of different workloads. The number is normalized to the offline theoretical optimal results.



(b) Accelerator utilization of different workloads.

Figure 5.4: Normalized system throughput and accelerator utilization of mixed workloads on a CPU-FPGA cluster.

that are using different accelerators and swap in the workloads that are submitted at a later time but use the same accelerators. Therefore, FPGAs are reprogrammed only once on each node given that the number of different accelerators in our experiments is two. A more general formulation that allows for arbitrary number of accelerators and arbitrary submission time can be found in Section 5.5.

Figure 5.4 plots the normalized system throughput to theoretical optimal and accel-

erator utilization. Comparing the baseline *static partition* with *default sharing*, we find that static partition performs better when the cluster is partitioned in a way that the ratio of KM nodes to LR nodes is close to the ratio of KM workloads to LR workloads (*i.e.,* ratio is 0.5), while *default sharing* performs better when the workloads only contain LR or KM applications (*i.e.,* ratio is 1 or 0), since the applications can use all 4 FPGA nodes. However the advantages of *default sharing* decline as the workloads become more heterogeneous due to FPGA reprogramming overhead.

GAM incorporates the best aspects of *static partition* and *default sharing*: it potentially allows applications to use all cluster FPGAs (shown as the accelerator utilization rate in Figure 5.4 (b)). Meanwhile, it reduces FPGA reprogramming overhead by placing similar workloads on the same set of nodes. On average, *static partition* and *default sharing* are 27% and 22% away from the theoretical optimal results, while GAM is only 9% away from the optimal results.

5.5 Offline Scheduling Formulation

In this section, we present an integer linear programming (ILP) formulations of the offline resource and task scheduling problem in the context of YARN and a cluster with FPGA accelerators.

Each application requests a set of containers from YARN and/or global accelerator manager (GAM), and uses the allocated resources to execute its tasks. The applications may arrive at different times. We would like to optimize for the overall latency for this set of applications. Particularly these formulations consider FPGA reprogramming overhead when measuring performance.

5.5.1 Problem Formulations

Symbol Definition

- App_i: Application i.
- $Task_{ij}$: jth task of App_i.

- $Node_k$: Cluster node k.

Problem Parameters

- a_i: The arriving time of App_i.
- nc_i: The number of containers requested by App_i.
- t_{ij}: Execution time of $Task_{ij}$ on FPGAs.
- rep: Time to reprogram an FPGA.
- $e_{iji'j'}$: Dependency exists between $Task_{ij}$ and $Task_{ij'}$.
- $p_{ii'} = 1$, if App_i and $App_{i'}$ uses different FPGA bitstreams; otherwise 0.

Decision Variables

- $m_{ijkp} = 1$, if $Task_{ij}$ is scheduled to $Node_k$ and from a global point of view, it is the pth task executed on $Node_k$; otherwise 0.
- s_i: Launch time of App_i.

Intermediate Variables

- f_i: The completion time of App_i.
- L: The maximal latency of all applications.
- u_{ij}: The start time of $Task_{ij}$.
- v_{ij}: The completion time of $Task_{ij}$.
- $b_{ij} = 1$, if executing $Task_{ij}$ requires FPGA reprogramming; otherwise 0.
- $o_{iji'j'} = 1$, if (1) $Task_{ij}$ and $Task_{i'j'}$ are executed on the same node, (2) they occupy consecutive execution slots with $Task_{ij}$ follows $Task_{i'j'}$, and (3) App_i and $App_{i'}$ uses different bitstreams; otherwise 0.
- $d_{ik} = 1$, if there exists at least one task from App_i that is executed on $Node_k$; otherwise 0.

Now we present the problem formulation as follows.

Optimization goal:

$$\min L. \tag{5.1}$$

Constraints:

- A task in only mapped to one execution slot:

$$\sum_{k,p} m_{ijkp} = 1,\ \forall i, j. \tag{5.2}$$

- Each slot only has one task:

$$\sum_{i,j} m_{ijkp} = 1,\ \forall k, p. \tag{5.3}$$

- Application launches after its arriving time:

$$s_i \geq a_i,\ \forall i. \tag{5.4}$$

- Tasks start after the application launches:

$$u_{ij} \geq s_i,\ \forall i, j. \tag{5.5}$$

- Definition of task completion time:

$$v_{ij} = u_{ij} + t_{i,j} + rep * b_{ij},\ \forall i, j \tag{5.6}$$

- Definition of completion time of App_i:

$$f_i \geq v_{ij},\ \forall i, j. \tag{5.7}$$

- Definition of maximal latency of all applications:

$$L \geq f_i.\ \forall i. \tag{5.8}$$

- Dependency between tasks:

$$u_{ij} \leq v_{ij'}, \forall e_{iji'j'}. \tag{5.9}$$

- Definition of whether a task requires reprogramming (b_{ij}):

$$b_{ij} = \sum_{i'j'} o_{iji'j'}, \forall i, j, i' \neq i, j'. \tag{5.10}$$

- Definition of $o_{iji'j'}$:

$$o_{iji'j'} = \sum_{k,p} m_{ijkp} \cdot m_{i'j'k(p-1)} \cdot p_{ii'},\ \forall i, j, i' \neq i, j'. \tag{5.11}$$

  Transforming the above formula into an linear programming is presented in the next section.

- Definition of d_{ik}:

$$d_{ik} = \begin{cases} 1, & \text{if } \sum_{j,p} m_{ijkp} \geq 1 \\ 0, & \text{otherwise} \end{cases},\ \forall i, k. \tag{5.12}$$

  Transforming the above formula into an linear programming is presented in the next section.

- Tasks are executed on the requested containers:

$$\sum_{k} d_{ik} = nc_i,\ \forall i. \tag{5.13}$$

5.5.2 ILP formulations

All the formulations in the above sections are in the linear form except for formula (5.11) and (5.12). We continue to discuss how to reformulate these two formulas into a linear form.

Reformulate formula (5.11) Formula (5.11) can be transformed into an ILP form by applying the following theorem:

Lemma 3 *Given binary variables x, a and b, function $x = ab$ is equivalent to the following set of functions in the linear form:*

$$\begin{cases} x \leq a, \\ x \leq b, \\ a + b - x \leq 1. \end{cases} \tag{5.14}$$

Reformulate formula (5.12) Formula (5.12) can be transformed into an ILP form by applying the following theorem:

Lemma 4 *Given a binary variable x, and a non-negative integer variable y, formulation*

$$x = \begin{cases} 1, & \text{if } y \geq 1 \\ 0, & \text{otherwise} \end{cases} \tag{5.15}$$

is equivalent to the following set of functions in the linear form:

$$\begin{cases} y \leq M \cdot x, \\ x \leq y, \end{cases} \tag{5.16}$$

where M is a very large positive integer, or the maximum value of y.

Theorem 2 *Formula 5.1 to formula 5.13 is an integer linear programming program.*

5.5.3 Complexity Analysis

Denote that the total number of tasks as N, the number of cluster nodes as M, and the number of applications as A. The total number of decision variables is $N^2M + A$. The total number of intermediate variables is $A + 1 + 3N + N^2 + MA$.

Denote that the number of task dependency edge e is E. Then the total number of constraints (from formula 5.1 to formula 5.13) is $4N + 2A + E + MN + 2N^2 + MA$.

5.5.4 Discussions

The formulations presented in this write-up aim to show that it is possible to formulate certain accelerator resource and task scheduling problem into an integer linear programming problem although it is not practical to call an ILP solution at runtime. Now we further examine some assumptions in our formulations and explain how to extend them to more general problems.

1. We assume that each node only has one FPGA board and each FPGA board only supports one execution kernel at a time. This assumption is usually true in today's server configuration. On the other hand, it is trivial to extend the current formulation to support multiple FPGA boards on the same node.

2. We assume all the tasks are executed on the FPGAs. We intentionally ignore the non-accelerable tasks in an application and do not consider migrating an accelerable task onto the CPU. The reason is that it is extremely difficult to formulate the execution time of a CPU task, which may involve disk and network accesses, in a multi-tenant scenario on today's multi-core CPU platforms.

3. We assume all the tasks in an application use the same FPGA bitstream. However it is trivial to extend the current formulation to support multi-bitstream in an application. We just need to change the parameter $p_{ii'}$ into $p_{iji'j'}$ and adjust the formulations correspondingly.

4. We assume the number of tasks is known in advance. This assumption, however, is not always true. In many iterative applications, such as machine learning applications, number of iterations depends on the convergence conditions and thus is not known beforehand.

5. This formulation provides an offline solution. Online resource scheduling and task scheduling (at YARN level, task scheduler level and node accelerator manager level) lack the information about future tasks and future applications to achieve a global optimal solution.

5.5.5 Related Work on Offline Resource and Task Scheduling

In the task and resource scheduling area, the concepts that are similar to FPGA reprogramming is context switch and preemption. Context switch is an important mechanism to share a CPU processor among multiple threads. However only a limited number of works studied this problem, which might due to the reason that in general-purpose systems, context switch cost is not significant. [DCC07] shows that average context switch overheads in Linux on an ARM platform is only 0.17% - 0.25%. Nevertheless in certain systems, such as real-time systems, tasks exhibit a very high context switch cost due to the latency time of the I/O devices [ERC95]. An analytic model is presented in [ERC95] to calculate the number of preemptions in scheduling a set of periodic tasks using 'Rate Monotonic (RM)' policy and 'Earliest Deadline First (EDF)' policy. It shows that EDF generally incur less context switching overhead than RM. [LMK99] proposes a 'Limited

Preemptive Scheduling' that limits preemptions to execution points with small cache-related preemption costs. [WS99, SW00] introduces a concept of preemption-threshold into 'Rate Monotonic' scheduling where a task can only be preempted if its preemption-threshold is lower than the other task's priority.

The overhead of preemption has also been studied in the dynamic voltage scaling (DVS) scenario, where voltage and clock frequency are dynamically adjusted to reduce power and energy consumption of the system. Preemption in DVS is more frequent and have negatively impact on the system energy consumption. [KSY02] shows that DVS scheduling algorithms can incur up to 500% task preemption comparing to non-DVS scheduling algorithms. [KKM04] explains that task preemption may increase the energy consumption on the memory subsystems and long lifetime of preempted tasks may increase energy consumption in system devices. A delayed-preemption, which tries to postpone preemption by delaying the activation of a higher-priority task is proposed to reduce the negative impact of DVS algorithms.

The works above mainly focuses on task scheduling how to reduce preemption overhead on embedded system or single node system. While in our problem, we also need to address the problem as which set of cluster nodes should be used to execute the jobs. Therefore our work is more general comparing to these prior works.

At cluster level, there are a limited number of works that consider preemption in resource allocation. [CDD14] considers to reduce the number of resource reallocation when reserving resource for a job. The problem is described in a MILP problem and the authors propose to use heuristic algorithm to solve the problem. Different from their work, we consider particularly FPGA tasks in our formulation and we do not allow multiple tasks to use the same FPGA at a time. We are able to formulate the problem into an ILP problem.

5.6 Conclusions

In this chapter we propose to manage the logical accelerator functionality as a resource instead of the physical hardware platform itself. Using this new concept, we are able to extend Hadoop YARN with an accelerator-centric scheduling policy that better man-

ages global accelerator resources and mitigates the FPGA reprogramming overhead. We demonstrate that our accelerator-centric scheduling achieves close to optimal system throughput. In addition, we present an ILP formulation of offline resource and task scheduling which can be used to evaluate the gap between optimal allocation and online scheduling algorithms.

CHAPTER 6

Cluster-Level Data Shuffling

6.1 Introduction

Data sorting and shuffling is, in many ways, the heart of large-scale distributed computing frameworks, such as MapReduce/Hadoop and Spark. For simplicity, here we use shuffle to refer to this data sorting and data shuffling stage. In MapReduce, shuffle performs the data sorting and data transmission from the mappers to the reducers, guaranteeing that the input to every reducer is sorted by key. It is an area of the codebase where refinement and improvements are continually being made and is where the "magic" happens [Whi12].

Spark, whose operators are a strict superset of MapReduce, also supports sorting-based data shuffle between consecutive computation stages. In fact, at an earlier stage of Spark development, it adopted a hashing-based shuffle which turns out to be one of the limiting factors of scaling out Spark. Sorting-based shuffle is thus then introduced to Spark. Overall, the data shuffle is an expensive but indispensable operation of many distributed computing frameworks.

In this chapter we use the benchmark Terasort to evaluate the performance of cluster-level data shuffling. Terasort is a popular benchmark that measures the amount of time it takes to sort one terabyte of randomly distributed data. In Terasort, each data record has 100 bytes, including a 10-bytes key and 90-bytes of data. The sorting routine reorganizes the data according to the key value. We perform extensive experiments on different implementations of Terasort and study whether accelerators can improve the sorting performance.

6.2 Experiment Setup and Initial Profiling

To study the Tersort performance, we set up a local 8-node Xeon (E5-2620 @2.40 GHz) cluster; each node has 64 GB memory and 24 cores. We leverage the latest software framework in the community. We use the latest Hadoop 2.7.3 to build our distributed file system and use the latest Spark 2.0.1 as the computation framework. The Terasort benchmark originates from SparkBench 2.0 [spa]. Nevertheless we made significant changes to the source code to achieve a better performance. We configure the Spark local working directory to SSD, which exploits the best I/O performance and leaves more stress on CPU processing capability. We clear system file cache before each application starts so as to guarantee that data is fetched from disk instead of the cached in-memory copy.

6.2.1 Data Shuffling in Spark

The major difference between Spark shuffling and MapReduce shuffling is the time to start the reduce tasks. In MapReduce, reduce tasks can start after some of the map tasks have finished; while Spark, as a more general computation framework, chooses to adopt a stage-by-stage scheduling policy where shuffle operations are considered as boundaries of the stages. Therefore, reduce tasks do not start until all the map tasks have finished.

We believe that the task scheduling policy in MapReduce is probably more suitable for Terasort applications. Nevertheless, in this work, we focus on the Spark shuffling performance since Spark is gaining popularity in recent years (as demonstrated in Figure 6.1).

6.2.2 Initial Terasort Profiling

Listing 1 shows our Spark Terasort implementation. *dataset* is the input data that is read from HDFS. The Spark operator $repartitionAndSortWithinPartition$ first repartitions the dataset based on the provided partitioner and then sorts by the key. Figure 6.2 shows the RDD transformation of the code in Listing 1. Table. 6.1 lists the Spark configurations.

In this experiment we sort 64 GB of data on a 8-node Xeon cluster. The application

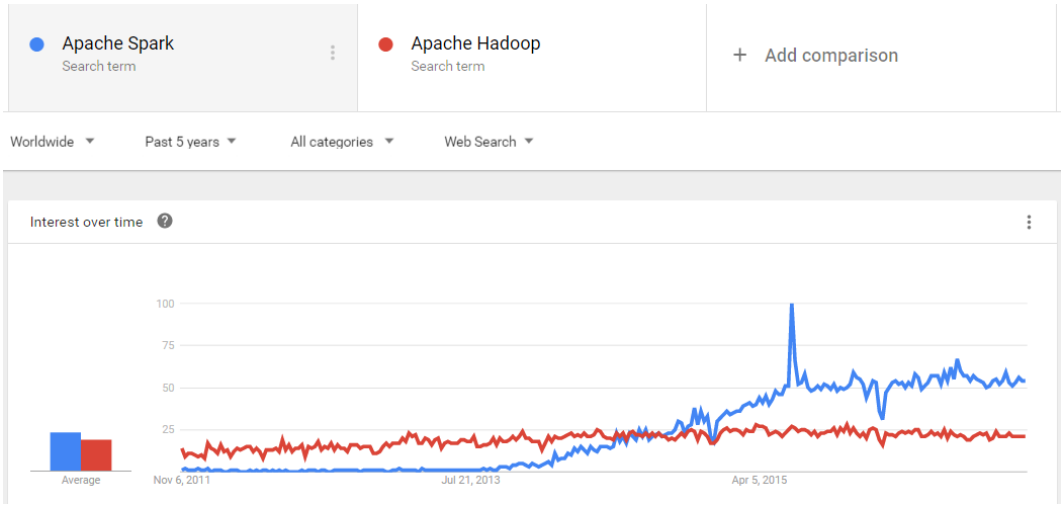


Figure 6.1: Google Trend for Apache Spark and Apache Hadoop. Data was collected from www.google.com/trends on November 6, 2016.

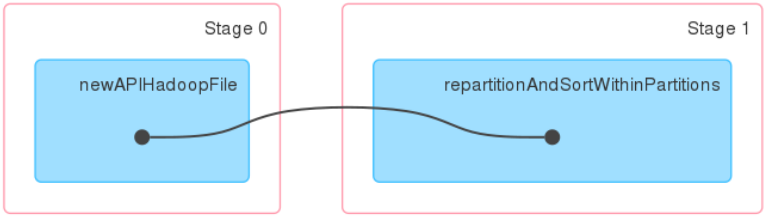


Figure 6.2: RDD computation lineage for the code in Listing 1.

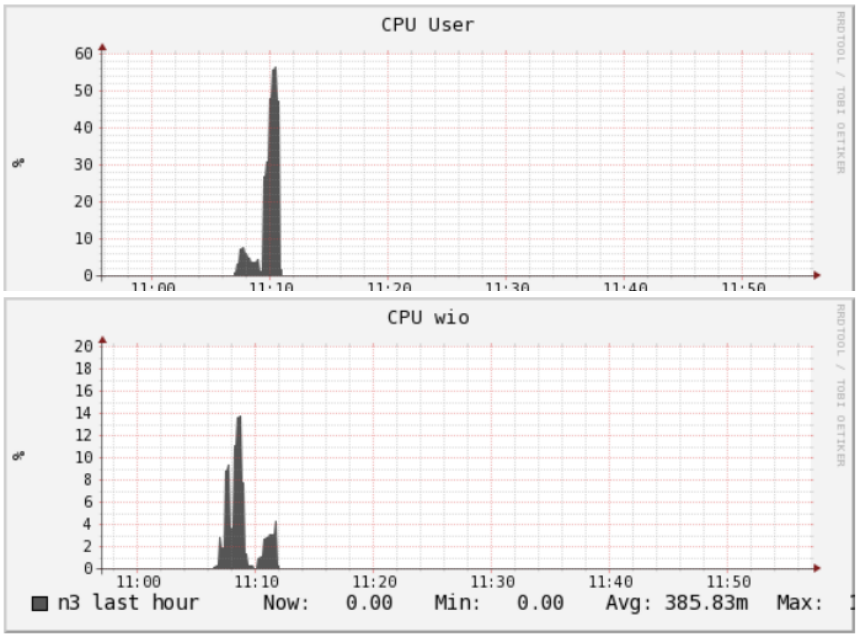


Figure 6.3: System metrics of a node when running 64 GB Spark Terasort on 8-node cluster.

Listing 1 Spark Terasort (scala)

```scala
val dataset = context.newAPIHadoopFile[
    Array[Byte], Array[Byte]](inputFile)
val sorted = dataset.repartitionAndSortWithinPartitions(
    new TeraSortPartitioner(dataset.partitions.size))
sorted.saveAsNewAPIHadoopFile[TeraOutputFormat](outputFile)
```

Table 6.1: Spark configurations in Terasort.

spark.executor.memory	60g
spark.serializer	org.apache.spark.serializer.KryoSerializer
spark.rdd.compress	false
spark.io.compression.codec	lzf
SPARK_WORKER_CORES	24

takes 3.6 minutes to finish. The *newAPIHadoopFile* stage and *repartitionAndSortWithinPartitions* take 2.3 minutes and 1.3 minutes to finish respectively. Figure 6.3 shows the system metrics monitored from Ganglia monitoring tools. The application starts at around 11:08 and ends at around 11:12.

From the figure 6.3 we can easily identify stage 0 and stage 1. Stage 0 has higher wait IO and thus is more I/O intensive, while stage 1 is more CPU-intensive. In stage 1, CPU utilization reaches 60%. Therefore, we should focus on using accelerators in stage 1 instead of stage 0 to improve Terasort performance.

6.2.3 Acceleration Opportunities

Shuffle phase can variably stress the CPU, memory, disk and network capabilities. It can be more CPU-bounded when the latest hardware is used for disk and network.

Generally there are two approaches to accelerate Spark shuffling from the system perspective, 1) modifying Spark user program, and 2) modifying Spark internal code, *e.g.,* incorporating customized shuffle code. Modifying Spark user program has the benefit of not affecting other Spark users while modifying Spark internal code opens up more optimization flexibility. In this chapter, we will exam the first option, modifying Spark

user program, and leave the study on second option as future work.

6.2.4 Sorting TeraFormat Records in C++ and Java

After we profile the Spark Terasort application, we now start to evaluate whether a sort accelerator can improve the performance. Therefore, we first implement a sorting routine in C++, which is demonstrated to provide better performance in sorting Teraformat data than that of Java. We will evaluate if such a sorting routine is beneficial to the Spark Terasort application in the next section.

Note that although we do not have a GPU-based or an FPGA-based implementation of sorting routines, many insights obtained from using a C++ sorting routine stay true when we begin to consider using GPU/FPGA accelerators. This is due to the fact that accelerators are peripheral devices, and thus, using a GPU/FPGA sorter from JVM and using a C++ sorter from JVM share the same procedure, *i.e.,* offloading computation from JVM to native.

The following experiments show the performance of our C++ TeraFormat sorter and compare it with a Java TeraFormat sorter. Each TeraFormat record includes a 10-byte key and a 90-byte value. The records are sorted according to their keys. We use the generic sort function from the C++ standard library and the default sort function from the *java.util.Arrays* respectively.

Figure 6.4 shows the execution time to sort TeraFormat records and the speedup of the C++ sorting routine over the Java sorting routine. This figure demonstrates that the C++ program consistently outperforms the Java program. When the data size is small, *e.g.,* 12.5 MB, C++ sorter outperforms the Java sorter by 6.6x. When the data size is larger, *e.g.,* larger then 400 MB, the C++ sorter still achieves a 1.6x speedup over the Java sorter.

In the remainder of this chapter, we will use this C++ sorting routine as an sorting accelerator to evaluate the performance of integrating accelerators into Spark shuffling.

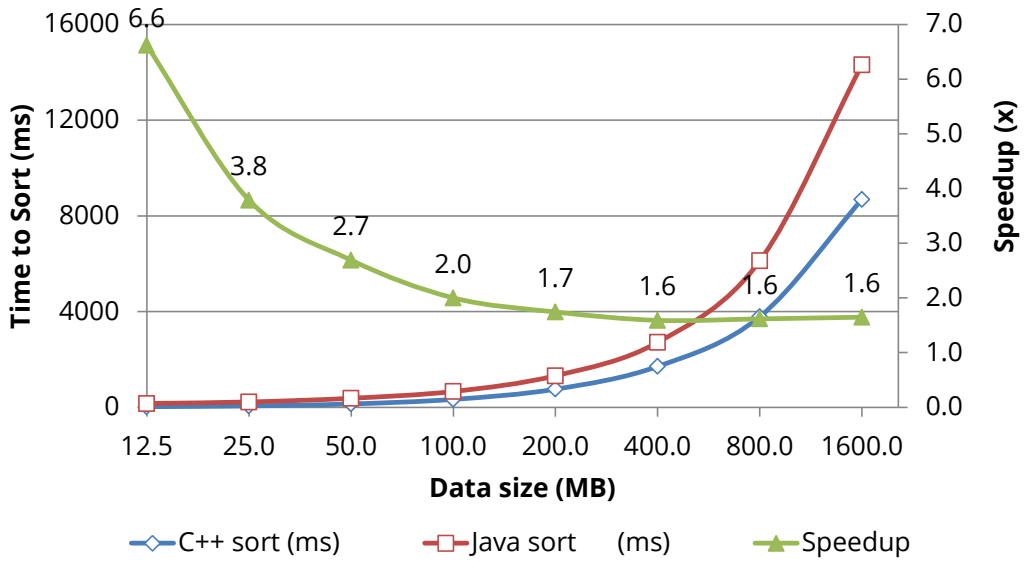


Figure 6.4: Performance comparison on sorting routines written in C++ and Java. Speedup of C++ sorting over Java sorting is also measured.

6.3 Using Accelerators in Spark Shuffling

In this section we evaluate the approach of moving the sorting routine from Java virtual machine (JVM) to native by using a native C++ sorter instead of the default sorter in JVM. We will demonstrate that although the C++ sorter is 1.6x faster than the Java sorter, moving the sorting routine from JVM to native incurs significant overhead in today's Spark shuffling implementation, canceling out the benefit of the C++ sorting routine.

6.3.1 Terasort with Customized Java/Scala Sorting Routines

Since the Spark operator *repartitionAndSortWithinPartitions* (in Listing 1) does not take a customized sorting routing over a data array, we change the code to incorporate the user-defined sorting routing. Listing 2 shows the Spark code that can leverage a user-defined sorting routine. We use the *partitionBy* operator to first partition the data, and each partition will use our own sorting routine, *Sort* , in *mapPartitions*. Figure 6.5 shows the RDD transformation of the code in Listing 2.

An example of a customized sorting routine is shown in Listing 3. It traverses through the RDD iterator, transforms it into an array and then sorts the array. We denoted this set of experiments as **Terasort-C0**.

Listing 2 Spark Terasort using customized sorting routine.

```
1 val dataset = context.newAPIHadoopFile[
2     Array[Byte], Array[Byte]](inputFile)
3 val partitioned = dataset.partitionBy(new
4     TeraSortPartitioner(dataset.partitions.size))
5     .persist(StorageLevel.MEMORY_ONLY)
6 val sorted = partitioned.mapPartitions(
7     Sort, preservesPartitioning = true)
8 sorted.saveAsNewAPIHadoopFile[TeraOutputFormat](outputFile)
```

Listing 3 A customized sorter written in Scala.

```
1 def Sort (iter: Iterator[(Array[Byte], Array[Byte])]) :
2     Iterator[{Array[Byte], Array[Byte]}] = {
3   iter.toArray.sorted.iterator
4 }
```

Now we present the Terasort performance with the customized Scala sorter (Listing 3). The total time of sorting 64 GB on our 8-node cluster is 3.7 minutes, where stage 0 takes 2.3 minutes and stage 1 takes 1.4 minutes. Comparing Terasort-C0 with the original Terasort that takes 3.6 minutes to finish, this 0.1 minute overhead comes from the data copying from *Iterator* to *Collection*. Spark permits the users to access the RDD partitions through iterator, a utility for traversing over the elements. In order to sort the data, we have to gather the elements into a collection and sort them therein. This gathering step consumes additional CPU cycles and incurs memory footprint. The average memory consumption on each node rises from 32 GB (original Terasort) to 40 GB (Terasort-C0). Note that if a much larger data (i.e., 100 GB) is sorted and memory is all used up, the slow down of Terasort-C0 can be more significant due to memory swapping.

6.3.2 Terasort with Customized C++ Sorting Routines

Now we evaluate the performance of integrating the C++ sorter into the Spark Terasort application. We modify the sorter implementation in Listing 3 to leverage a native C++

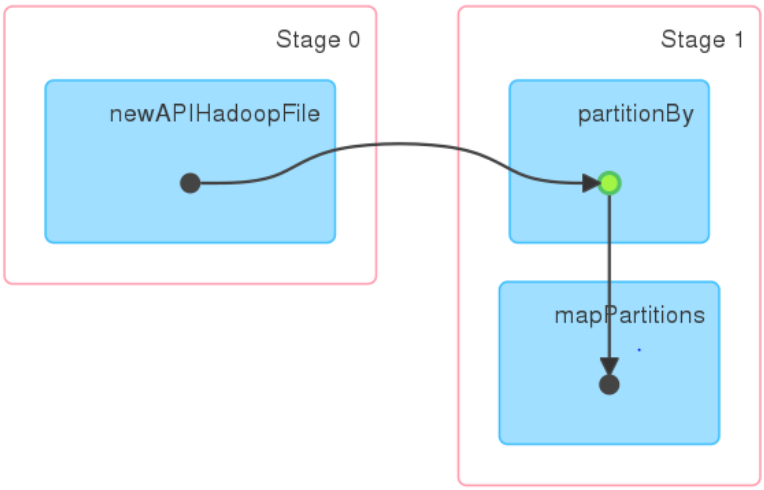


Figure 6.5: RDD computation lineage for the code in listing 2.

sorter. Java native interface (JNI) is used to transfer data from JVM to native. Note that only the keys, rather than both the key and the values, of the key-value pairs need to be transferred; this helps to reduce the JNI overhead. We denote this set of experiments that use the C++ sorted as **Terasort-C1**.

The application now takes 4.5 minutes to finish, which is 1.25x slower than the original Terasort (3.6 minutes). Memory usage reaches to 64 GB on each cluster node which indicates a larger memory footprint of using a customized C++ sorting routine. Note that a large memory footprint leads to higher JVM garbage collection workload and thus can slow down the application. We observe that in Terasort-C1, the actual time that is spent on sorter is only 20% to 50% of each *MapPartitions* task's lifetime.

6.3.3 Reducing Memory Footprint

Now we study whether using a more space-efficient data storage format can delivery better performance.

We modify the RDD persist level in Listing 2 line 5 from MEMORY_ONLY to MEMORY_AND_DISK_SER. By using MEMORY_ONLY, Spark stores the RDD as deserialized Java objects in JVM, while by using MEMORY_AND_DISK_SER, Spark stores the RDD as serialized Java objects in memory and disk. Serialized objects are generally more space-efficient but more CPU-intensive to read and write due to the additional deserialization overhead.

We denote the experiments that adopts the above modification as **Terasort-C2** and **Terasort-C3**, where Terasort-C2 uses the scala sorter and Terasort-C3 uses the native C++ sorter. Our experiment results show that using serialized Java objects helps reduce memory consumption. However, data serialization and deserialization consume more CPU cycles and thus slow down the program.

Table 6.2 summarizes the performance of all the benchmarks.

6.3.4 Performance Summary

Table 6.2: Performance of sorting 100GB TeraFormat records.

Benchmark	persist method	sorter	Total time (minutes)	slow down w.r.t original	ave. memory used on each node
original Terasort	deserial.	Scala	3.6	-	32 GB
Terasort-C0	deserial.	customized Scala (Listing 3)	3.7	1.03x	40 GB
Terasort-C1	deserial.	customized C++	4.5	1.25x	64 GB
Terasort-C2	serial.	customized Scala (Listing 3)	7.7	2.14x	34 GB
Terasort-C3	serial.	customized C++	7.9	2.19x	50 GB

There are two major observations:

- JNI overhead. Comparing Terasort-C0 with Terasort-C1, we find that offloading sorting routine from JVM to native through JNI affects the performance, although the native sorting routine is faster than the Java sorting routine.
- Data serialization overhead. Data serialization is necessary when a large amount of

data needs to be sorted. Comparing Terasort-C0 with Terasort-C2 and comparing Terasort-C1 with Terasort-C3, we find that although serialized data is more space-efficient, data serialization and deserialization procedure significantly slow down the program.

The above analysis on overheads of integrating accelerators into JVM indicates that using accelerators to improve the sorting routine in Spark Terasort is not promising.

6.4 Conclusions

In this chapter we perform extensive experiments to understand cluster-level data shuffling, and evaluate the opportunities in improving data shuffling using accelerators. We find that data shuffling is both CPU- and memory-intensive. Thus, we propose to use a customized sorting routine with the goal of offloading the computation to accelerators to alleviate CPU workload. However, we find that using a customized sorting routine incurs significant overhead in data traversing, serialization and deserialization, which slows down the Terasort application by up to 2.2x. This demonstrates that accelerating shuffling with a customized sorting routine through application-level coding is not promising.

CHAPTER 7

Conclusions and Future Directions

Accelerators are becoming increasingly popular in modern computing, addressing the demand for high-throughput and low-power computation. This thesis discusses several important aspects of accelerator-rich architectures, including accelerator resource management at chip-level, CPU-accelerator orchestration at node-level, resource allocation at cluster-level, and data shuffling management at cluster level. We believe that as future computing platforms move towards accelerator-centric solutions, the proposed methodologies of resource and data management in this thesis can be used in future systems and can inspire more innovations. Emerging acceleration-rich architectures open up a number of rich new research areas in resource and data management. Some of the possible avenues for future research on this topic are discussed below:

- **Job-Level Task Management** This thesis covers topics on cluster-level resource management and node-level resource management. A missing piece is to investigate job-level task management, where a job gets resources from a cluster resource manager and decides how to allocate its tasks to the allotted resources. A task scheduler needs to predict a task's performance on different computation resources so as to make scheduling decisions. This task allocation is non-trivial in accelerator-rich clusters since some nodes may be equipped with accelerators and some nodes may not. Moreover, accelerators are shared by multiple tenants.

- **Cluster-Level Data Shuffling** We investigate using accelerators to accelerate in-memory sorting during the data shuffling stage by modifying application code. Another direction to make use of the accelerator is through modifying Spark internals and using customized Spark shuffling through the pluggable interface for shuffles that is provided by Spark. Finally one can make use of the accelerators during the network transfer stage. Since there is a large amount of data that needs

to be transferred during the shuffling stage, compression and decompression accelerators should be able to reduce network load.

References

[ADC95] Imtiaz Ahmad, Muhammad K Dhodhi, and CYR Chen. "Integrated scheduling, allocation and module selection for design-space exploration in high-level synthesis." *IEE Proceedings-Computers and Digital Techniques*, **142**(1):65–71, 1995.

[AHS14] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. "Gzip on a Chip: High Performance Lossless Data Compression on FPGAs Using OpenCL." In *Proceedings of the International Workshop on OpenCL 2013 & 2014*. ACM, 2014.

[alt] "Altera." `http://www.altera.com`. Accessed: 2014-11-4.

[BFH04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. "Brook for GPUs: stream computing on graphics hardware." In *ACM Transactions on Graphics (TOG)*, volume 23, pp. 777–786. ACM, 2004.

[BML12] Shuvra S Bhattacharyya, Praveen K Murthy, and Edward A Lee. *Software synthesis from dataflow graphs*, volume 360. Springer Science & Business Media, 2012.

[Bor08] Dhruba Borthakur. "HDFS architecture guide." *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, p. 39, 2008.

[BRH15] Brad Brech, Juan Rubio, and Michael Hollinger. "IBM Data Engine for NoSQL - Power Systems Edition." Technical report, IBM Systems Group, 2015.

[BWA] "Burrow-Wheeler Aligner for pairwise alignment between DNA sequences." `https://github.com/lh3/bwa`. Accessed: 2016-09-25.

[cal] "Calgary Corpus Dataset." `http://corpus.canterbury.ac.nz/descriptions/#calgary`. Accessed: 2016-09-25.

[CCF16] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. "A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms." In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pp. 109:1–109:6, New York, NY, USA, 2016. ACM.

[CCX05] Deming Chen, Jason Cong, and Junjuan Xu. "Optimal module and voltage assignment for low-power." In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pp. 850–855. ACM, 2005.

[CDD14] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. "Reservation-based Scheduling: If You'Re Late Don'T Blame Us!" In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pp. 2:1–2:14, New York, NY, USA, 2014. ACM.

[CDL13] Eric S Chung, John D Davis, and Jaewon Lee. "Linqits: Big data on little clients." In *ACM SIGARCH Computer Architecture News*. ACM, 2013.

[CGG12] 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, pp. 843–849, New York, NY, USA, 2012. ACM.

[CHL12] Jason Cong, Muhuan Huang, Bin Liu, Peng Zhang, and Yi Zou. "Combining module selection and replication for throughput-driven streaming programs." In *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 1018 –1023, march 2012.

[CHW16] Jason Cong, Muhuan Huang, Di Wu, and Cody Hao Yu. "Heterogeneous Datacenters: Options and Opportunities." In *Proceedings of the 53nd Annual Design Automation Conference*. ACM, 2016.

[CLN11] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees 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, 2011.

[CSR11] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. "Customizable domain-specific computing." *IEEE Design and Test of Computers*, **28**(2):6–15, 2011.

[CZ06] Jason Cong and Zhiru Zhang. "An efficient and versatile scheduling algorithm based on SDC formulation." In *Proceedings of the 43rd annual Design Automation Conference*, pp. 433–438. ACM, 2006.

[CZ12] Yuankai Chen and Hai Zhou. "Buffer minimization in pipelined SDF scheduling on multi-core platforms." In *17th Asia and South Pacific Design Automation Conference*, pp. 127–132. IEEE, 2012.

[DCC07] Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell. "Context Switch Overheads for Linux on ARM Platforms." In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *Commun. ACM*, **51**(1):107–113, January 2008.

[EBA11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark silicon and the end of multicore scaling." In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pp. 365–376. IEEE, 2011.

[ERC95] Juan Echague, Ismael Ripoll, and Alfons Crespo. "Hard real-time preemptively scheduling with high context switch cost." In *Real-Time Systems, 1995. Proceedings., Seventh Euromicro Workshop on*, pp. 184–190. IEEE, 1995.

[FKB15] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. "A scalable high-bandwidth architecture for lossless compression on fpgas." In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pp. 52–59. IEEE, 2015.

[GBS05] Marc Geilen, Twan Basten, and Sander Stuijk. "Minimising buffer requirements of synchronous dataflow graphs with model checking." In *Proceedings of the 42nd annual Design Automation Conference*, pp. 819–824. ACM, 2005.

[GGD02] Ramaswamy Govindarajan, Guang R Gao, and Palash Desai. "Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks." *Journal of VLSI signal processing systems for signal, image and video technology*, **31**(3):207–229, 2002.

[GLP] "GNU Linear Programming Kit." http://www.gnu.org/software/glpk.

[GTA06] Michael I Gordon, William Thies, and Saman Amarasinghe. "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs." *ACM SIGOPS Operating Systems Review*, **40**(5):151–162, 2006.

[GTK02] Michael I Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S Meli, Andrew A Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, et al. "A stream compiler for communication-exposed architectures." In *ACM SIGPLAN Notices*, volume 37, pp. 291–303. ACM, 2002.

[had] "Apache Hadoop." `https://hadoop.apache.org`. Accessed: 2016-05-24.

[har] "Intel to Start Shipping Xeons With FPGAs in Early 2016." `http://www.eweek.com/servers/intel-to-start-shipping-xeons-with-fpgas-in-early-2016.html`. Accessed: 2016-05-17.

[HCK09] Amir H Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. "Flextream: Adaptive compilation of streaming applications for heterogeneous architectures." In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pp. 214–223. IEEE, 2009.

[HK10] Alan J Hoffman and Joseph B Kruskal. "Integral boundary points of convex polyhedra." In *50 Years of Integer Programming 1958-2008*, pp. 49–76. Springer, 2010.

[HKZ11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." In *NSDI*, volume 11, pp. 295–308, 2011.

[HLW15] Yuanhong Huo, Xiaoyang Li, Wei Wang, and Dake Liu. "High performance table-based architecture for parallel CRC calculation." In *The 21st IEEE International Workshop on Local and Metropolitan Area Networks*, pp. 1–6. IEEE, 2015.

[HWB09] Andrei Hagiescu, Weng-Fai Wong, David F. Bacon, and Rodric Rabbah. "A computing origami: Folding streams in FPGAs." In *Proceedings of the 46th Annual Design Automation Conference*, pp. 282 –287, 2009.

[HWY16] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. "Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale." In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pp. 456–469, New York, NY, USA, 2016. ACM.

[ID91] Masaki Ishikawa and Giovanni De Micheli. "A module selection algorithm for high-level synthesis." In *Circuits and Systems, 1991., IEEE International Sympoisum on*, pp. 1777–1780. IEEE, 1991.

[ILP98] Kazuhito Ito, Lori E Lucke, and Keshab K Parhi. "ILP-based cost-optimal DSP synthesis with module selection and data format conversion." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **6**(4):582–594, 1998.

[int] "Intel Xeon + FPGA Platform for the Data Center." `http://reconfigurablecomputing4themasses.net/files/2.2%20PK.pdf`. Accessed: 2016-10-20.

[JHI10] Haris Javaid, Xin He, Aleksander Ignjatovic, and Sri Parameswaran. "Optimal synthesis of latency and throughput constrained pipelined MPSoCs targeting streaming applications." In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pp. 75–84. IEEE, 2010.

[KDT12] Christoph Kessler, Usman Dastgeer, Samuel Thibault, Raymond Namyst, Andrew Richards, Uwe Dolinsky, Siegfried Benkner, Jesper Larsson Träff, and Sabri Pllana. "Programmability and performance portability aspects of heterogeneous multi-/manycore systems." In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1403–1408. IEEE, 2012.

[KDV97] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter Van Der Wolf. "An approach for quantitative analysis of application-specific dataflow architectures." In *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*, pp. 338–349. IEEE, 1997.

[KKL11] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. "Achieving a single compute device image in OpenCL for multiple GPUs." *ACM SIGPLAN Notices*, **46**(8):277–288, 2011.

[KKM04] Woonseok Kim, Jihong Kim, and Sang Lyul Min. "Preemption-aware dynamic voltage scaling in hard real-time systems." In *Low Power Electronics and Design, 2004. ISLPED '04. Proceedings of the 2004 International Symposium on*, pp. 393–398, Aug 2004.

[KM08] Manjunath Kudlur and Scott Mahlke. "Orchestrating the Execution of Stream Programs on Multicore Platforms." In *PLDI*, PLDI '08. ACM, 2008.

[KSY02] Woonseok Kim, Dongkun Shin, Han-Saem Yun, Jihong Kim, and Sang Lyul Min. "Performance comparison of dynamic voltage scaling algorithms for hard real-time systems." In *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*, pp. 219–228, 2002.

[KT11] Dirk Koch and Jim Torresen. "FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting." In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 45–54. ACM, 2011.

[LCM09] Qiang Liu, George A Constantinides, Konstantinos Masselos, and Peter YK Cheung. "Combining data reuse with data-level parallelization for FPGA-targeted hardware compilation: a geometric programming framework." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **28**(3):305–315, 2009.

[LDW06] Shih-wei Liao, Zhaohui Du, Gansha Wu, and Guei-Yuan Lueh. "Data and computation transformations for Brook streaming applications on multiprocessors." In *International Symposium on Code Generation and Optimization (CGO'06)*, pp. 12–pp. IEEE, 2006.

[LGX09] Weichen Liu, Zonghua Gu, Jiang Xu, Yu Wang, and Mingxuan Yuan. "An efficient technique for analysis of minimal buffer requirements of synchronous dataflow graphs with model checking." In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pp. 61–70. ACM, 2009.

[LHK09] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping." In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 45–55. IEEE, 2009.

[Li13] Heng Li. "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM." *arXiv preprint arXiv:1303.3997*, 2013.

[LM87a] Edward A Lee and David G Messerschmitt. "Synchronous data flow." *Proceedings of the IEEE*, **75**(9):1235–1245, 1987.

[LM87b] Edward Ashford Lee and David G Messerschmitt. "Static scheduling of synchronous data flow programs for digital signal processing." *IEEE Transactions on computers*, **100**(1):24–35, 1987.

[LMK99] Sheayun Lee, Sang Lyul Min, Chong Sang Kim, Chang-Gun Lee, and Minsuk Lee. "Cache-Conscious Limited Preemptive Scheduling." *Real-Time Syst.*, **17**(2-3):257–282, December 1999.

[LSP13] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. "Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems." In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pp. 245–256. IEEE Press, 2013.

[MDM01] Fabrice Monteiro, Abbas Dandache, A M'sir, and Bernard Lepley. "A fast CRC implementation on FPGA using a pipelined architecture for the polynomial division." In *Electronics, Circuits and Systems, 2001. ICECS 2001. The 8th IEEE International Conference on*, volume 3, pp. 1231–1234. IEEE, 2001.

[MGA03] William R Mark, R Steven Glanville, Kurt Akeley, and Mark J Kilgard. "Cg: A system for programming graphics hardware in a C-like language." In *ACM Transactions on Graphics (TOG)*, volume 22, pp. 896–907. ACM, 2003.

[mll] "Spark MLlib." `http://spark.apache.org/mllib/`. Accessed: 2016-05-24.

[mni] "The MNIST database of handwritten digits." `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#mnist8m`. Accessed: 2016-05-24.

[NG93] Qi Ning and Guang R. Gao. "A novel framework of register allocation for software pipelining." In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL, pp. 29–42. ACM, 1993.

[PCC14] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. "A reconfigurable fabric for accelerating large-scale datacenter services." In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pp. 13–24. IEEE, 2014.

[PZ92] T-B Pei and Charles Zukowski. "High-speed parallel CRC circuits in VLSI." *IEEE Transactions on Communications*, **40**(4):653–657, 1992.

[sam] "Samtools." `http://www.htslib.org/doc/samtools.html`. Accessed: 2016-09-25.

[SB00] S. Sriram and S. Bhattacharyya. "Embedded Multiprocessors Scheduling and Synchronoization." In *Marcel Dekker, Inc., New York*, 2000.

[SGB06] Sander Stuijk, Marc Geilen, and Twan Basten. "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs." In *Proceedings of the 43rd annual Design Automation Conference*, pp. 899–904. ACM, 2006.

[SKA13] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. "Omega: flexible, scalable schedulers for large compute clusters." In *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 351–364. ACM, 2013.

[spa] "Spark Bench." `https://github.com/SparkTC/spark-bench`. Accessed: 2016-10-20.

[Str] "StreamIt benchmarks." `http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml`. Accessed: 2014-10-23.

[SW00] Manas Saksena and Yun Wang. "Scalable real-time system design using preemption thresholds." In *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pp. 25–34. IEEE, 2000.

[SWB88] Kanwar Jit Singh, Albert R Wang, Robert K Brayton, and Alberto Sangiovanni-Vincentelli. "Timing optimization of combinational logic." In *Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on*, pp. 282–285. IEEE, 1988.

[SWN07] Welson Sun, Michael J Wirthlin, and Stephen Neuendorffer. "FPGA pipeline synthesis design exploration using module selection and resource sharing." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **26**(2):254–265, 2007.

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe. "StreamIt: A language for streaming applications." In *International Conference on Compiler Construction*, pp. 179–196. Springer, 2002.

[UGT09] Abhishek Udupa, R Govindarajan, and Matthew J Thazhuthaveetil. "Synergistic execution of stream programs on multicores with accelerators." In *ACM Sigplan Notices*, volume 44, pp. 99–108. ACM, 2009.

[viv] "Vivado Design Suite User Guide." `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf`. Accessed: 2014-11-4.

[VMD13] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. "Apache Hadoop YARN: Yet another resource negotiator." In *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5. ACM, 2013.

[Wal07] Mathys Walma. "Pipelined cyclic redundancy check (CRC) calculation." In *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on*, pp. 365–370. IEEE, 2007.

[Whi12] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[WS99] Yun Wang and Manas Saksena. "Scheduling Fixed-Priority Tasks with Preemption Threshold." In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, RTCSA '99, pp. 328–, Washington, DC, USA, 1999. IEEE Computer Society.

[xil] "Xilinx." `http://www.xilinx.com`. Accessed: 2016-10-30.

[yar] "Allow for (admin) labels on nodes and resource-requests." `https://issues.apache.org/jira/browse/YARN-796`. Accessed: 2016-10-20.

[ZBS13] Jiali Teddy Zhai, Mohamed A Bamakhrama, and Todor Stefanov. "Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems." In *Proceedings of the 50th Annual Design Automation Conference*, p. 170. ACM, 2013.

[ZCD12] 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, pp. 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[ZCF10] 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*, pp. 10–10, 2010.

[ZFJ08] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. "AutoPilot: A platform-based ESL synthesis system." In *High-Level Synthesis*. Springer, 2008.

[zyn] "Zynq-7000 All Programmable SoC." `http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`. Accessed: 2016-10-30.