A Study on Acceleration Methods of Data Center Applications with Reconfigurable Hardware

Eric Shun Fukuda

A thesis presented for the degree of
Doctor of Engineering

Graduate School of Information Science and Technology
Hokkaido University
Japan

March, 2015

A Study on Acceleration Methods of Data Center Applications with Reconfigurable Hardware

Eric Shun Fukuda

Abstract

This study discusses the use of reconfigurable hardware, especially in data centers. Reconfigurable hardware is a promising technology for overcoming the difficulties face by general-purpose processors, which have been central to computer science for decades. Enabling reconfigurable technology to be used by a variety of people, including software developers, will have a great impact on pushing current computing to a new era. The recent enthusiasm for cloud computing indicates the increasing significance of data centers, and improving the performance of data centers in computation speed and energy efficiency has a great impact on the entire computer industry.

Although computers have progressed enormously since their invention, physical restriction is beginning to prevent them from achieving higher computation speed and energy efficiency. These two matrices, computation speed and energy efficiency, are critical in data centers that handle gigantic data traffic from all over the world. As the amount of data that data centers handle is growing even more, computer developers need different ways of improving computers. Among the several alternatives that have been proposed, reconfigurable hardware is one of the most suitable choices, and this is what we try to use in this study. Reconfigurable hardware promises higher performance and energy efficiency in many application domains, and several research groups have been trying to deploy it in data centers.

Meanwhile, reconfigurable hardware is used mainly by embedded or networking device vendors, and curious end users often simply enjoy trying them with reconfigurable boards purchased on their own. This is because it is difficult to obtain higher performance with inexpensive reconfigurable

boards, which an individual can buy, than with a commercially available general-purpose processor, and reconfigurable boards that can exceed the performance of commercial general-purpose processor are too expensive for end users to buy individually. By deploying reconfigurable hardware and making it available to the public, the cost of utilizing reconfigurable hardware will go down and thus more end-users will be eager to use it. More applications such as databases that require very low latency will be built on reconfigurable hardware as well as general-purpose processors. Such a computing system, which uses several general-purpose processors and reconfigurable devices or heterogeneous systems, is too complicated for end users' everyday use. As a consequence, to make the system look simple for the end users, it will become more transparent: they will eventually not notice which kind of computing device they are using.

However, there are many problems that must be solved before such a cloud system can be widely used. Among them, we focus on two major problems in this study: developing applications with reconfigurable hardware is more difficult compared to software applications, thus, the cost of developing the application is higher; and a method for using reconfigurable hardware in data centers is not established.

In order to solve the first problem, we assess where the difficulties lie in the state of the art design method that uses high-level synthesis tools for developing hardware accelerated application systems, and propose a method to overcome the difficulties. For solving the other problem, we propose a new technique of using reconfigurable hardware in data centers.

Contents

1 Introduction **7**

1.1 Background 7

1.1.1 Problems of General Purpose Processors 7

1.1.2 Applications on Hardware 8

1.1.3 Growing Popularity of Reconfigurable Hardware 9

1.1.4 Reconfigurable Hardware in Data Centers 11

1.1.5 Open Issues of Reconfigurable Hardware in Data Centers 13

1.1.6 Future of Reconfigurable Hardware in Data Centers . . 18

1.2 Organization 20

References 22

2 Acceleration by HLS **24**

2.1 Introduction 24

2.2 Stream Processing 25

2.3 Related Work 26

2.4 DRP: The Evaluation Platform 28

2.5 Window Join on DRP 29

2.5.1 Evaluation Strategy 30

2.5.2 Step 1: Pure Software Code 31

2.5.3 Step 2: Sliding Window Buffer 32

2.5.4 Step 3: Parallel Output Buffer 33

2.5.5 Step 4: Match Table 35

2.5.6 Step 5: Chunk Data Prefetching 36

2.5.7 Step 6: Parallel Table Lookup 37

2.5.8 Step 7: Loop Folding (Pipelining) 38

2.5.9 Step 8: Low Match Rate Optimization 43
2.6 Discussion 43
2.6.1 Optimization Overview 43
2.6.2 Performance 45
2.6.3 Lessons learned 46
2.6.4 Adaptive Stream Processing 49
2.7 Summary 49
References 51

3 Acceleration by a Hybrid Approach of HLS and RTL 53
3.1 Introduction 53
3.2 Related Work 55
3.3 SQL-Based Stream Processing Language 57
3.4 Our Approach 57
3.4.1 Generalization of Queries into Abstract Hardware . . . 58
3.4.2 Mapping Abstract Hardware to C Code 59
3.4.3 Shallow Hardware Optimization in C Code 59
3.5 Evaluation 60
3.6 Discussion 61
3.7 Conclusion 62
References 67

4 Acceleration by I/O Caching 69
4.1 Introduction 69
4.2 Key-value Stores in Data Centers 70
4.3 Background 72
4.3.1 Memcached 72
4.3.2 Related Work 76
4.4 Concept of NIC Cache 77
4.5 Cache Simulation 78
4.5.1 Testing Tool 79
4.5.2 Simulation Results 83
4.6 Hardware Design 84
4.6.1 Experimental Conditions 88

4.6.2 Latency 89
4.6.3 Throughput 90
4.7 Cache Size Maximization 90
4.8 Discussion and Future Work 94
4.9 Conclusion 95
References 96

5 Conclusion **98**

Acknowledgement **103**

List of Publications **104**

List of Figures

1.1 Simplified architecture of a typical reconfigurable hardware. . 9
1.2 Replacing general purpose processors in data centers with reconfigurable hardware. 11
1.3 Variations of organization of servers with reconfigurable hardware. 15
1.4 Two architectures of servers for reconfigurable hardware and general purpose processors. 17

2.1 Compilation flow. 27
2.2 DRP hardware overview. 28
2.3 Join and window join. 30
2.4 Synthesized hardware (Step 1). 32
2.5 Synthesized hardware (Step 2). 34
2.6 Expected hardware (Step 3). 36
2.7 Synthesized hardware (Step 4). 38
2.8 Timing chart of the folding. 39
2.9 Synthesized hardware (Step 8). 41
2.10 Throughput improvement. 45
2.11 Match rate dependency (Step 7 vs 8). 48

3.1 Comparison of conventional and our approaches. 55
3.2 Example queries and schema of incoming stream. 56
3.3 C function calls for nested queries. 64
3.4 Query module architecture and its code. 65
3.5 Throughput comparison between naive C written without hardware development knowledge and parsed C. 66

4.1 The operation of memcached. 73
4.2 The image of the proposed method. 74
4.3 Connection of software modules. 78
4.4 Key access distribution for the first 5,000 requests. 79
4.5 Appearance interval of same keys for all workloads. 80
4.6 Cumulative ratio of keys. 81
4.7 Miss rate for GET requests with various replacement policies. 82
4.8 Miss rates for small cache sizes for Workload A with FIFO, LRU, and LFU cache algorithms. 85
4.9 Miss rates with read-allocate and read-no-allocate for Workload C. 86
4.10 NIC cache architecture. 87
4.11 Correspondence of the hash table and the value storage. . . . 87
4.12 FPGA NIC mounted on a memcached server. 88
4.13 Throughput of the system with various hit rates. 91
4.14 Latency improvement with various associativity and cache capacity. 92
4.15 Miss rates with constant block memory size. 93

List of Tables

2.1 Performance metrics throughout the optimization process. . . 42

3.1 Usage of input bandwidth. 62

4.1 Memcached commands 73
4.2 Description of YCSB workloads. 75
4.3 Design specification of FPGA 89
4.4 Latencies of the system. 89
4.5 Throughputs of the system based on RTL simulation. 90

Chapter 1

Introduction

1.1 Background

1.1.1 Problems of General Purpose Processors

Since its arrival in the 1940s, the computer has always been seeking higher computation speed. This objective has been achieved through the replacement of vacuum tubes with silicon transistors, miniaturization of the silicon process, and parallelization of processors. As general-purpose processors based on von Neumann Architecture improved both from manufacturing processes and processor architecture aspects, algorithms that assume such architectures have been widely studied and improved.

After the use of computers spread widely in society, the demand for higher energy efficiency increased. Today, many people have multiple computing devices, and data centers of web service providers have tens of thousands of servers of more. Thus, reducing the power consumption or improving the energy efficiency of computers is critical for solving the ever-growing energy problem. The performance of processors has improved continuously by miniaturization of the silicon process, though Moore's law, which predicts that processor performance improvement will end in the next decade or so.

Among many solutions to this issue, using hardware dedicated for specific applications is a prospective alternative. One such hardware that is already in wide use is the Fast Fourier Transform (FFT) hardware accelerator [1]. In contrast to general-purpose processors, which have not been developed for a specific application, application-specific hardware achieves higher performance and energy efficiency in exchange for generality. However, developing application-specific hardware requires more in time and costs than developing a software application that runs on a general-purpose processor. Solving this cost issue is essential for application-specific hardware to be broadly used.

1.1.2 Applications on Hardware

Dedicated hardware for a specific application achieves higher performance and energy efficiency than general-purpose processors because of the following reasons:

- Application-specific hardware gains high parallelism that is specific to the application.
- Application-specific hardware implements the instruction as hardware circuits whereas general-purpose processors fetch instructions from the memory and control the data path according to them, which leads to increased energy consumption.

However, implementing a dedicated hardware costs more than implementing software applications on a general-purpose processor in terms of time and expense for the following reasons:

- Software development and hardware development require different skills, and thus it is difficult for software engineers, who often develop the algorithm, to design hardware.
- It is necessary to extract high parallelism from the application in order to obtain high performance on dedicated hardware, and it is difficult to extract high parallelism simply by converting the software algorithm to hardware because software algorithms are intended to run on a general-purpose processor that supposes serial processing.
- Although software can be executed immediately after writing or modifying the source code, hardware requires silicon chip manufacturing which costs millions or tens of millions of dollars.

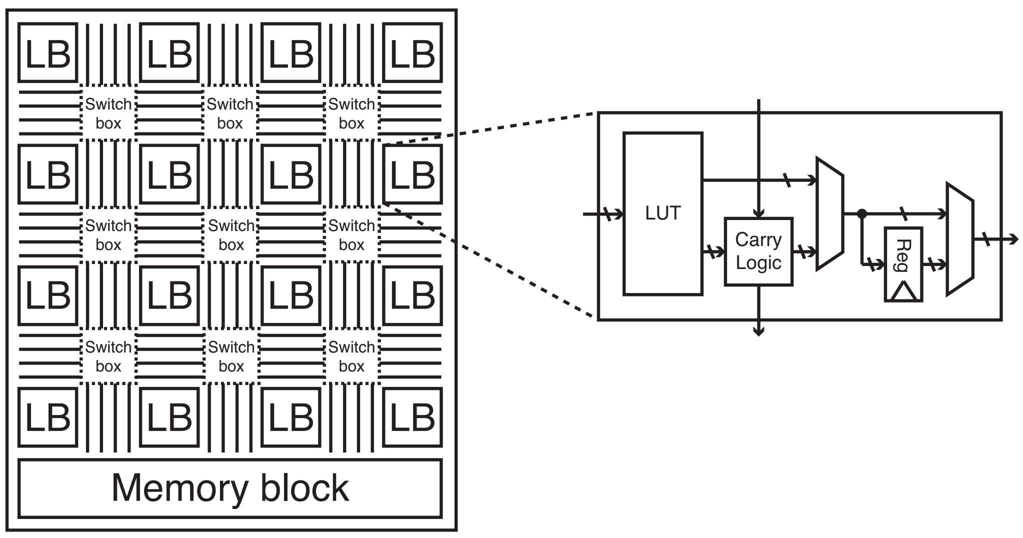


Figure 1.1: Simplified architecture of a typical reconfigurable hardware.

Such advantages and disadvantages are significant especially for application specific integrate circuits (ASICs) and even growing as the silicon process continues to miniaturize. Recently, the increase of financial cost of manufacturing silicon chips limits the opportunities to develop application specific hardware.

1.1.3 Growing Popularity of Reconfigurable Hardware

As a solution for this situation, reconfigurable hardware is attracting wide attention. Reconfigurable hardware is a hardware chip in which a hardware developer can design and modify the internal circuit after production. Generally, on a reconfigurable chip, there are logic blocks (LBs) aligned in a grid (Fig. 1.1). Each LB has a lookup table (LUT), a carry logic, and a register. A user can program the LUT in an LB so that it can have an arbitrary combinational logic. (Some reconfigurable architectures use an arithmetic logic unit (ALU) instead of LUTs.) Programming the multiplexer can configure the connections among these elements within an LB. There are wires running vertically and horizontally in the grid throughout the chip to which the LBs are connected, and the wires to be connected in the switch box can be programmed. At the intersections of the wires, there are switch boxes that can be programmed as to which vertical and horizontal wires should be connected. In addition, some memory blocks are inserted on the chip whose wire connections can be also programmed. By programming these programmable elements, an arbitrary circuit can be implemented on a reconfigurable hardware chip as long as the resource required for the application does not exceed what is available on the chip.

The circuit is usually described with Hardware Description Language (HDL) such as Verilog HDL or VHDL. The hardware description is converted to information of the LUTs, multiplexers (MUXes), and switch boxes' configuration by the tool that the reconfigurable hardware vendors provide. (This configuration file is called a bit file.) The bit file is then downloaded to the chip's configuration memory, and the chip acquires the intended behavior. Examples of reconfigurable hardware chips include field-programmable gate arrays (FPGAs) [2], programmable logic devices (PLDs) [3], coarse grained reconfigurable architecture (CGRA) [4], and a dynamically reconfigurable processor (DRP) [5], etc.

The switches that configure the on-chip interconnection and redundant wirings of reconfigurable hardware become the overhead and therefore, the performance is lower than for ASICs. This limitation made reconfigurable hardware used mainly for pre-manufacturing testing of ASICs until recently because it was difficult to implement a high performance application on reconfigurable hardware. However, as the miniaturization of the silicon process continued, reconfigurable hardware became capable of implementing sufficiently practicable circuits. Now that the cost for making ASICs has risen, there are more opportunities for reconfigurable hardware to provide a better way of accelerating an application even if the performance is lower than ASIC.

The progress of hardware development tools is another force that promotes the dedicated hardware use. (This is not only for reconfigurable hardware but also for ASICs.) Algorithms that search the hardware design space are continuously improving the performance of hardware synthesized fro HDL descriptions. Designing dedicated hardware is becoming less of an obstacle.

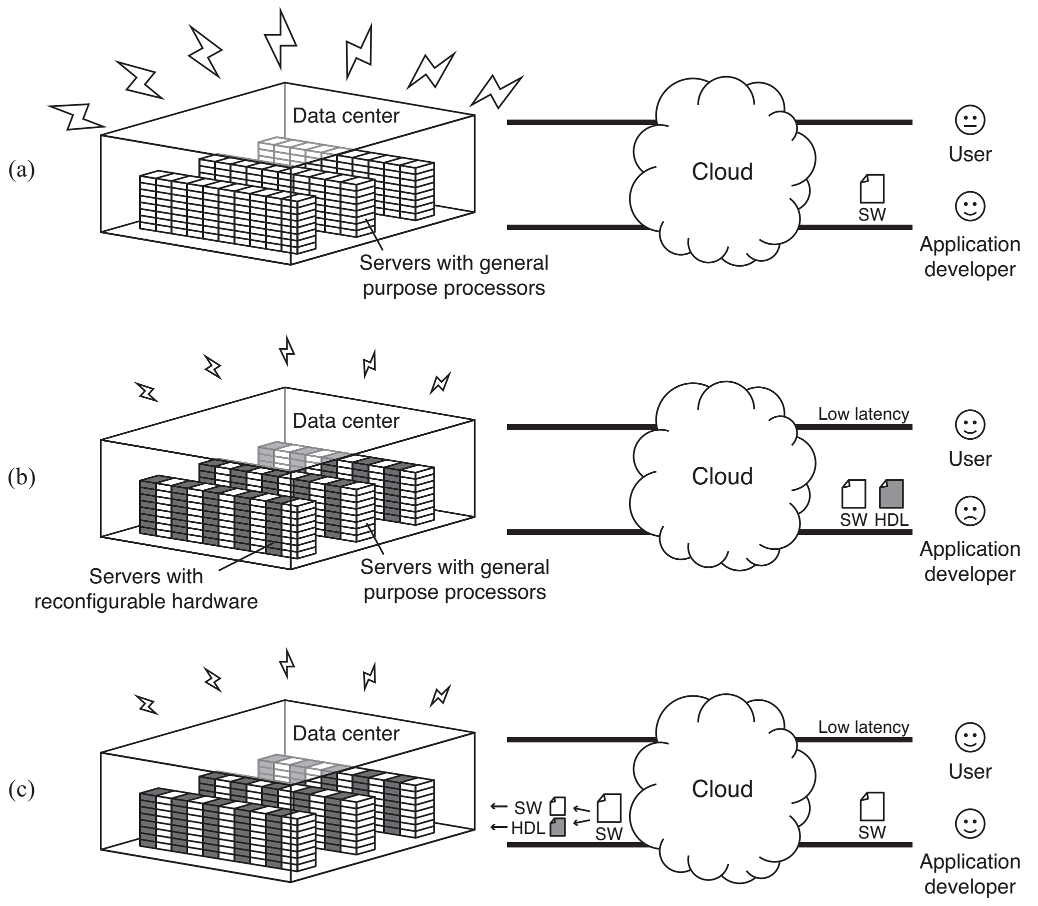


Figure 1.2: Replacing general purpose processors in data centers with reconfigurable hardware.

1.1.4 Reconfigurable Hardware in Data Centers

Many web service providers such as Google, Amazon, Facebook, and Microsoft, run data centers that have over tens of thousands of server computers (Fig. 1.2a). Running so many computers in parallel enables the service to deal with an enormous amount of data at an enormous data rate. Moreover, the widespread use of cloud computing is moving the processes that were done in computers of individuals to data center computers and aggregating data that people and companies kept by on their own to web service providers' data centers.

A natural result of this trend is that the energy consumption of data centers is becoming a growing issue. According to a report, one data center consumes as much as power as one thermal power plant can generate [6]. Companies that run data centers are eager to reduce the power consumption by improving the energy efficiency of servers and the efficiency of air conditioning of data centers.

Dedicated hardware can solve the problem that data centers are facing. As mentioned previously, dedicated hardware executes tasks faster and is more efficient than general-purpose hardware, reducing computing resources in number of processors and energy. In addition, because data centers aggregate the tasks that were scattered to individuals' and companies' computers all around the world, dedicated hardware can be more effective.

Reconfigurable hardware suits the usage in data centers for the same reasons that apply to ASICs; however, there is a specific reason that reconfigurable hardware is better than ASICs in data centers: Reconfigurable hardware can modify its functionality whenever necessary, which enables web services powered by dedicated hardware to be updated more frequently than using the more costly ASICs.

Recently, several studies have been actually trying to deploy reconfigurable hardware to data centers (Fig. 1.2b). One such work is by Microsoft Research [7]. In this work, Putnam and his group accelerated Microsoft's search engine, Bing, with FPGAs. In their system, most of the search engine is executed on general-purpose processors. However, the process called ranking, which ranks possible search results and is a costly portion of a search engine, is offloaded to FPGAs and accelerated. The experiment using 1,632 FPGA servers showed that the throughput doubled while keeping the latency the same as software. Microsoft is plans to carry out this system in production and to apply it to other applications.

A professional hardware development team that is particularly allowed to access the facilities in the data center performs this work. However, another study aims to open the reconfigurable hardware in the data center to the public. The goal is to enable end users to use FPGA resources as much as they want anytime by allocating virtual FPGA resources in the same manner as software virtual machines in Infrastructure as a Service (IaaS)

[8]. This system successfully accelerated the application-level load balancer; however, further evaluation is needed. If this system is effective for various applications, it should soon become popular with many end users because the cost of the reconfigurable hardware can be shared among those who use the same hardware. Whereas every user must buy FPGA boards, costing hundreds or thousands of dollars without such a cloud system, this system will be much less expensive.

1.1.5 Open Issues of Reconfigurable Hardware in Data Centers

However, there are some issues that still must be solved in order to use reconfigurable hardware in data centers.

Hardware Development for Software Engineers

One issues is the difficulty for software engineers to develop application-specific hardware. Although reconfigurable hardware provides high performance and power efficiency to application developers with competitive cost, the difficulties for software developers to design dedicated hardware prevents reconfigurable hardware from being widely used by software developers (Fig. 1.2b). The reasons are as follows:

1. Compared to software programming languages such as C, Java and Python that are used for developing applications that run on general-purpose processors, HDLs such as Verilog HDL and VHDL that are used for developing dedicated hardware use lower abstraction for describing the functionalities, and thus they are difficult for people to understand.

2. Dedicated hardware gains processing speed by parallelizing the application where general-purpose processors processes data basically serially, and various techniques are required to extract maximum parallelism without violating data dependencies.

3. Although general-purpose processors are based on von Neumann Architecture, dedicated hardware does not have a specific architecture, and this leads to broad design search space.

4. Because there are limited numbers of computing elements and memory elements on a reconfigurable hardware chip, the application developer should design the circuit so that it uses the elements in a balanced manner.

Recently, a technology called High-Level Synthesis (HLS), which synthesizes hardware circuits from codes that were written in software programming languages such as C and Java, is becoming widely used among hardware developers. Although HLS has been studied for decades, it was not until recently that they could be in practical use and become bundled to development tools provided by reconfigurable hardware vendors [9]. HLS relieved the difficulty caused by reason 1 above and reduced the time for developing hardware.

Although the remaining three difficulties have also been relieved by HLS to some extent, they still remain as challenges. Examples of possible solutions for reasons 2 and 3 are:

- to synthesize highly parallelized dedicated hardware from instruction sequence for general-purpose processors.

- Synthesize hardware from application-specific description language.

- Generate a code from software code that is easy for HLS tools to synthesize hardware.

In this study, we first clarify what is being a hurdle for software developers to develop hardware when using an HLS tool. On the basis of the perception, we propose a method based on the third method listed previously.

Reason 4 does not become an issue as long as the circuit we want to implement is small enough that it consumes only a fraction of available resources. However, when the resource usage is critical for the performance, the possible solutions include

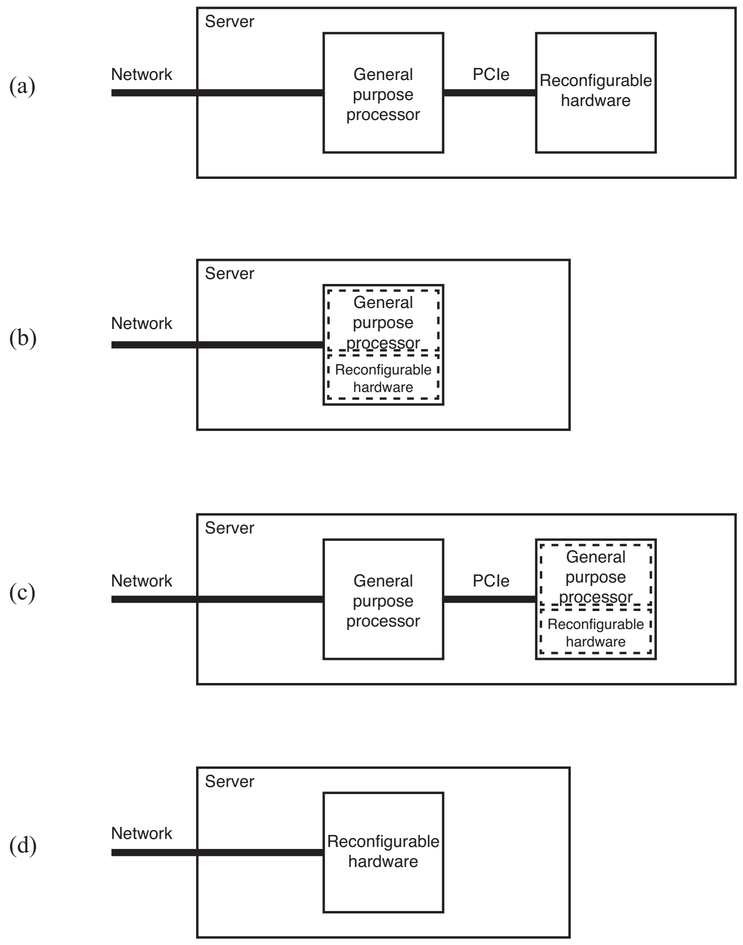


Figure 1.3: Variations of organization of servers with reconfigurable hardware.

- offloading only the critical function to a dedicated hardware and using a general-purpose processor for everything else
- dividing the circuit into pieces and distributing them to several reconfigurable hardware chips

In this study, we assume that the circuit fits on a single chip. Although the second approach is actively studied in the community, we do not cover it in this study.

Architecture of Reconfigurable-hardware-powered Data Centers

Another problem of using reconfigurable hardware in data centers is that the methodology of reconfigurable hardware's usage for data centers is not established yet. Reconfigurable hardware has been used in various forms in computer systems:

- mounting a PCI Express board with reconfigurable hardware on a server with a general-purpose processor (Fig. 1.3a)
- using a reconfigurable architecture core that is integrated with a general-purpose processor core on the same chip (Fig. 1.3b), or using such a chip mounted on a server with a general-purpose processor (Fig. 1.3c);
- using a server that has a reconfigurable hardware device and not a general-purpose processor (Fig. 1.3d).

On using reconfigurable hardware, we should be very careful to select the architecture of the system because the performance varies significantly when reconfigurable hardware is involved in the system.

Here, we will classify computer applications in two categories: real-time and non-real-time. In data centers, real-time application examples include data retrieval and stream processing, which often require low latency, and non-real-time application examples include indexing for search engines and analyzing data stored in databases that often require high throughput.

For applications that require high throughput, placement of reconfigurable hardware in the data center does not have a significant effect on the performance: what matters is whether the data source and hardware are connected with a high-bandwidth connection. Hence, an extension board with reconfigurable hardware connected to a server with a general purpose-processor via PCI Express will fulfill the requirements as long as the bandwidth of the PCI Express is sufficient. However, for applications that require low latency, it works better if the reconfigurable hardware is placed near the network inside the server, avoiding the extra latency that occurs when placing a general-purpose processor in between the network and the reconfigurable hardware.

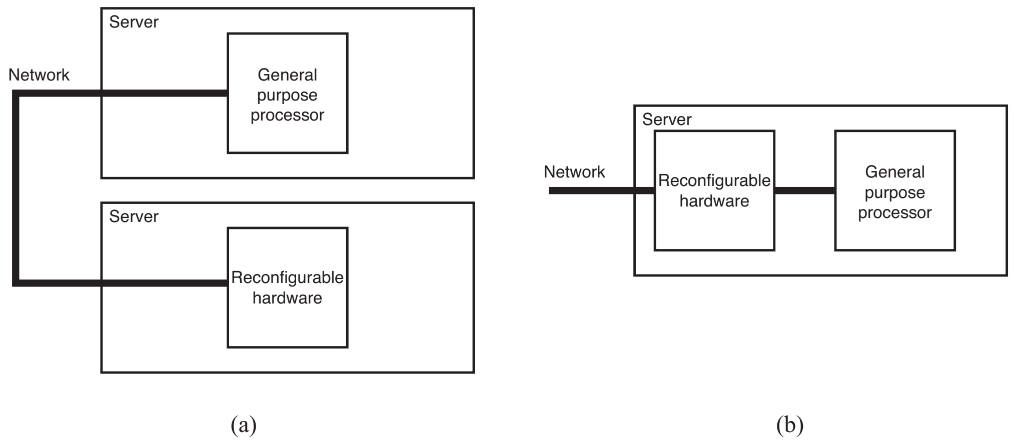


Figure 1.4: Two architectures of servers for reconfigurable hardware and general purpose processors.

An example of a server computer that has the network and reconfigurable hardware directly connected and a general-purpose processor is not placed in between is Maxeler Technology's MPC-X series [10]. However, this server does not have a general-purpose processor. Recently, using reconfigurable hardware and general-purpose hardware together is becoming more popular because reconfigurable hardware and general-purpose processors are good at different types of applications and the development of such a heterogeneous system is becoming easier and more efficient because of recent advances of hardware development techniques. In addition, heterogeneous systems are anticipated to be used more widely in data centers.

There are two types of architecture for using reconfigurable hardware and a general-purpose processor in combination while connecting the network directly to the reconfigurable hardware. One way is to use a server that has only reconfigurable hardware such as the aforementioned MPC-X, and connect it over a network to a regular server that has a general-purpose processor (Fig. 1.4a). The latency of networks is decreasing every year. Nevertheless, it is better not to have such latency. The other way is to place reconfigurable hardware at the network interface of the server that has a general-purpose processor (Fig. 1.4b). This architecture enables the application developers to use both the reconfigurable hardware and general-purpose processor together with a small latency between the two devices.

Examples of works that use such architecture are research on High Frequency Trade (HFT) by NEC [11] and on Memcached by Xilinx [12]. In these systems, the FPGA at the network interface does the process that should be done at a low latency, and the general-purpose processor that is connected to the FPGA via PCI Express does the computation that uses the output of the FPGA or that the FPGA is not good at. The usage of the general-purpose processors in these works were to do only a fraction of the computation that the application requires or to do completely different computation between the FPGA and the general-purpose processor.

In Chapter 4, we propose a novel method of taking advantage of the architecture that places the reconfigurable hardware at the network interface of a server computer that has a general-purpose processor. In this method, the general-purpose processor and the reconfigurable hardware do basically the same computation; however, the reconfigurable hardware does only the functions that are frequently called with the frequently accessed data, and the general-purpose processor handles the rest of them. This method enables the reconfigurable hardware to process the data that comes in from the network at a low latency, and at the same time, leaves the general-purpose processor to do a variety of computations that are used less frequently. It also reduces the cost of developing the hardware system of the application because only a part of the application that is frequently used has to be implemented in hardware.

1.1.6 Future of Reconfigurable Hardware in Data Centers

Research continues on deploying and using reconfigurable hardware in data centers. Methods of using reconfigurable hardware over the Internet will develop approximately in three steps. First, we will use computer boards with reconfigurable hardware and a network connection. A reconfigurable hardware board that does not require mounting on a computer via PCI Express and are capable of programming from remote over the Internet will be shared among many hardware developers also software developers that are curious about developing hardware. Although some reconfigurable hardware boards are very expensive, the developer using the board is not necessarily occupying the board all the time. Thus, sharing the board will reduce its non-active period, reducing the cost for each developer. Even if reconfigurable hardware becomes available in cloud systems, this kind of form of using reconfigurable hardware will remain in the embedded system industry.

Next, reconfigurable hardware will be available for end users by allocating its resources through IaaS. This method will enable an end users to use multiple reconfigurable hardware as a computing fabric, or multiple end users to share a single reconfigurable hardware device. At this step, the developer using this method should still need the mindset and skills for developing a hardware system regardless of using RTL or HLS (Fig. 1.2b).

In the final step, the user will not notice whether the person is running a program on reconfigurable hardware or a general-purpose processor (Fig. 1.2c). This is due to the advances in hardware development technology. The compiler and the program execution environment will run the software program partially on the general-purpose processor and partially on the reconfigurable hardware while determining which is more efficient. Of course, the end user will have the option to optimize the part that is executed on reconfigurable hardware so that it will be faster and more efficient. However, as the hardware offloading technology matures, the cost of manually optimizing the hardware will come to a point that it will not compensate for the benefit of the performance improvement. This is the same situation as software compilers in that it took over the manual optimization as its compilation technique progressed.

In prospect of such future, the work we provide in this study is significant in the following perspective:

- Hardware development technology for software developers is the most important factor for introducing reconfigurable hardware to data centers. As we will go over in Chapter 2, knowing today's latest hardware development technology and analyzing the difficulties that remain will shed light as to what should happen next, which should be done regularly because the fastest and most meaningful way to advance the technology is to practice with the latest technology.

- As reconfigurable hardware goes transparent in data centers, the application running on the hardware must be written entirely in software. Extracting high performance from reconfigurable hardware with a software program is a challenging task. The work introduced in Chapter 3 achieves this goal with several kinds of reconfigurable architectures theoretically.

- When reconfigurable hardware becomes transparent in data centers, functions of applications should be executed on various computing devices such as reconfigurable hardware, general-purpose processors, and graphic processing units (GPUs), either statically or dynamically so that the application will be executed in the most efficient way. The work discussed in Chapter 4 attempts to execute a part of the application running on a general-purpose processor on a reconfigurable device.

1.2 Organization

In Chapter 2, we analyze the difficulties when a software developer tries to develop a dedicated hardware with an HLS tool, which is the most straight forward approach for a software developer to develop hardware. In particular, we implement an operation called *window join*, an element operator of stream processing, on DRP with *CyberWorkBench* (CWB). As a result of our step-by-step implementation, we will show what kind of knowledge a software developer would need to develop hardware.

In Chapter 3, we propose a novel method for software programmers to develop application specific hardware with reconfigurable hardware. We take StreamSQL, a description language specific to stream processing, as an example for our method and propose a parser that converts StreamSQL queries to C code intended to be synthesized to hardware configuration with an HLS tool. This method enables application developers to develop hardware by StreamSQL without having any knowledge of hardware development.

In Chapter 4, we propose a method of taking advantage of reconfigurable hardware in data centers. We accelerate a server that runs `memcached`, an in-memory key-value store, by caching its functionalities and data to its network interface card (NIC) that is equipped with an FPGA and DRAM. This method does not require any modification to the software memcached; therefore, memcached servers that are already in operation can be enhanced.

Finally, in Chapter 5, we will summarize our work and discuss what should be done in the future.

Bibliography

[1] Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Palecezny. Very fast fourier transform algorithms hardware for implementation. *IEEE Transactions on Computers*, C-28:333–341, 1979.

[2] R.H. Freeman. Configurable electrical circuit having configurable logic elements and configurable interconnects, September 26 1989. US Patent 4,870,302.

[3] G. D. Electrically programmable logic circuits, June 18 1974. US Patent 3,818,452.

[4] Reiner Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, pages 564–570. ACM, 2001.

[5] Masato Motomura. A dynamically reconfigurable processor architecture. *Microprocessor Forum*, 2002.

[6] Jonathan G Koomey. Worldwide electricity used in data centers. *Environmental Research Letters*, 3(3), 2008.

[7] 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*, pages 13–24. IEEE, 2014.

[8] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 109–116. IEEE, 2014.

[9] The open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/opencl/`.

[10] Maxeler: Mpc-x series. `https://www.maxeler.com/products/mpc-xseries/`.

[11] Hiroaki Inoue, Takashi Takenaka, and Masato Motomura. 20Gbps C-based complex event processing. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[12] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bär, and Zsolt István. Achieving 10gbps line-rate key-value stores with fpgas. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, pages 1–6, 2013.

Chapter 2

Acceleration by HLS

2.1 Introduction

In Chapter 1, we mentioned that it is difficult for software engineers to develop hardware. If so, what specifically is difficult? Even though HLS has made it easier for developers to design hardware with software programming languages, difficulties still remain. In this chapter, we analyze the existing difficulties through a development process of an application specific hardware with C language and an HLS tool. Furthermore, we clarify what software developers should know when they design hardware. This kind of knowledge has not been explicitly clarified because hardware developers learn it empirically. However, this knowledge should give directions to what should be improved in order to make it easier for software engineers to develop hardware.

The organization of this chapter is as follows. In Section 2.2, we explain the significance and issues in hardware stream processing. In Section 2.3, we introduce related works of developing a hardware stream processor with software. After introducing the reconfigurable hardware platform that we used in our case study in Section 2.4, we look into the details of our steps of the development and optimization of window join that we implemented on DRP with HLS tool. Then, in Section 2.6, we evaluate the performance of our window join specific hardware and clarify the viewpoints that software developer should need in order to develop a hardware system. Furthermore, we point out the effectiveness of dynamically reconfiguration feature of DRP to stream processing. Finally, we will summarize our work in Section Section 2.7.

2.2 Stream Processing

Stream processing [1] is attracting considerable attention as an important computation paradigm in the era of big data and cloud computing. In contrast to conventional database processing, stream processing handles numerous real-time data streams delivered through network at real-time throughput, which is conventionally realized by distributed processing on parallel servers. Although these approaches have been remarkably successful in handling the dynamic nature of incoming streams [2], there still exists increasing demand for even higher throughput. Moreover, the urgent need to reduce the ever-rising power consumption in data centers [3] indicates the limitation in relying upon such a power-hungry *scale-out* approach alone.

In view of this situation, hardware-oriented acceleration of stream processing using field-programmable gate array (FPGA) has been actively studied [4–6]. In essence, hardware customized to a given problem can achieve much higher throughput/power than a software solution that runs on general-purpose hardware. However, such hardware solutions typically have two major drawbacks: (1) they have limited in-field flexibility and (2) software engineers find it difficult to design them. As *adaptive query* becomes an important notion in stream processing, issue (1) needs to be addressed seriously. Since stream processing requires considerable effort at the algorithm development level, owing to which this field employs mostly software engineers, issue (2) is also a fundamental problem. Actually, solving these problems on an FPGA-based framework has been an active research topic in recent years [4, 7].

The dynamically reconfigurable processor (DRP) [8] may serve as another good foundation to overcome these drawbacks from a different perspective. As will be presented in Sect. 2.4, the DRP has rich in-field flexibility and a software-friendly design environment (*i.e.,* a fairly mature C-level design tool). Since the DRP has been applied mostly in the domain of image processing applications [9] and is used by engineers with hardware knowledge, it is an interesting challenge to evaluate the performance of a DRP architecture and its design tool in solving the above-mentioned issues in stream processing.

On the basis of this observation, we carried out an experimental step-by-step implementation of adaptive stream processing on the DRP by considering *window join*, a simple but extensively studied important operation in stream processing, as a case study. Our goal in this study is to find answers to the following questions using the DRP evaluation platform:

- How and where a software source code should be modified to make it an optimized source code for hardware synthesis
- In doing so, how a software engineer should be knowledgeable and skilled in terms of hardware design
- How a state-of-the-art high-level design tool can hide hardware design details from the software programmer
- How performance varies according to different modifications
- How dynamic reconfiguration helps achieve adaptiveness of the solution

The evaluation was performed in a sequential manner, so that further insights could be obtained.

2.3 Related Work

Recently, several studies were conducted on hardware-accelerated stream processing. One such work uses C programming and a high-level synthesis tool [4]. This study expects the developer using this system to write simple functions *e.g.*, arithmetic operations, aggregation, etc., and to sequence them in a regular expression. Circuits synthesized from the functions evaluate the data stream, and if the sequence of return values satisfies the given regular expression, the system asserts a signal. After its high performance being

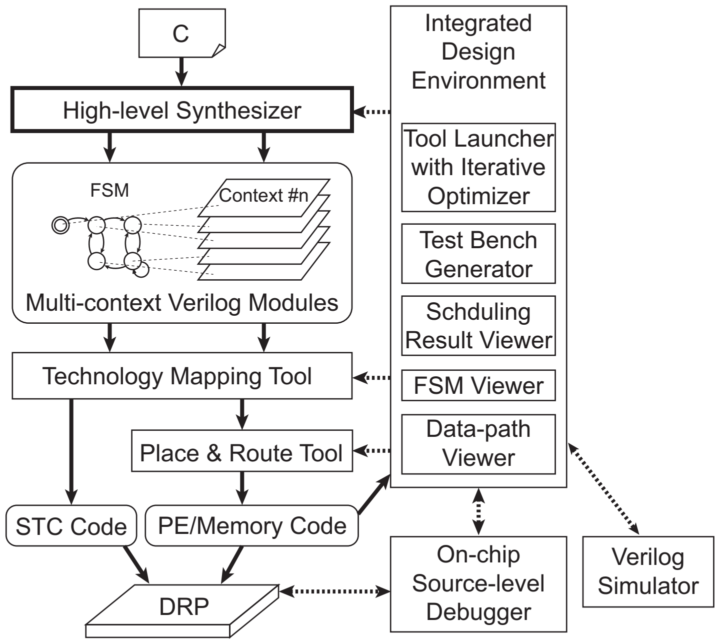


Figure 2.1: Compilation flow.

proven on an FPGA, this system has been enhanced in order to be dynamically reconfigurable, and thus queries can be modified while the system is in operation [7].

While these studies used C to describe the system discussed above, Mueller *et al.* proposed *Glacier*, a hardware synthesis system that maps SQL queries to hardware circuits [5]. The circuit consists of tiny circuit elements that correspond to stream processing operators. Mueller *et al.* demonstrated that the stream processor designed by this system and implemented on an FPGA runs faster at lower energy than a CPU. Based on this system, a dynamically reconfigurable stream processor that can switch its composition in a single clock cycle has been proposed [10].

These studies generally indicate that providing a domain-specific design framework on top of a general FPGA design platform is effective for accommodating stream processing systems in two ways: The synthesized circuits will be fast enough without requiring hardware optimizations by developers, and the system will be easier to be dynamically reconfigurable by limiting the circuits' degree of freedom. Despite lacking such a domain-specific design

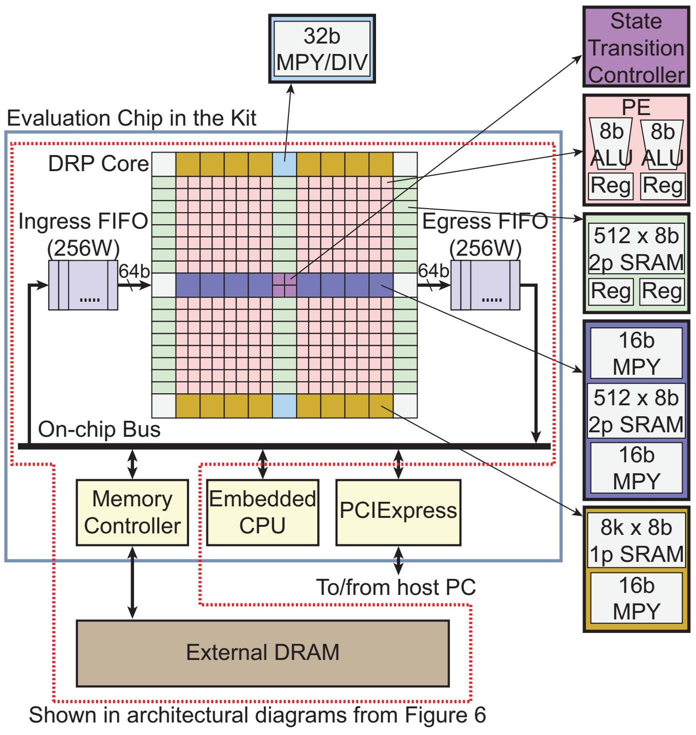


Figure 2.2: DRP hardware overview.

framework, DRP is inherently dynamically reconfigurable and has a C-based design tool, and hence it can be a platform for developing a stream processing system. To the best of our knowledge, ours is the first study to have evaluated such a system.

2.4 DRP: The Evaluation Platform

The dynamically reconfigurable processor (DRP) first presented in 2003 [8] features an array of small processing elements and block memories onto which user programs are compiled and mapped as hardware configurations (coarse-grained reconfigurable architecture). It features one-cycle dynamic reconfiguration; in other words, hardware configuration can switch every cycle, which is governed by a finite state machine (FSM) extracted from the user program. Essentially, reconfiguration takes place when a state transition occurs in the

FSM [11].

There exists an integrated design tool for DRP based on high-level synthesis technology whose compilation flow is depicted in Fig. 2.1 [12]. It first synthesizes input source code in C into an FSM and a set of *hardware contexts*, where each context is associated with each state. FSM is compiled into state transition controller (STC, explained later) code, and the hardware contexts are mapped onto the PE/memory array. Basically, by spatially mapping each primitive operation (each data structure) onto a different PE (internal memory), inherent parallelism in given input source code is naturally extracted in the resulting customized hardware. In addition, the tool has powerful automatic optimization capabilities for enhancing the inherent parallelism, thereby shortening the critical chain of operations, and reducing the PE/memory usage, such as speculative conditional branch execution and balanced operation tree mapping. Moreover, the tool supports various programmer-controlled optimization capabilities such as *loop unrolling*, *loop folding*, and *loop merging* for extracting additional parallelism [11]. Overall, the tool efficiently extracts/enhances parallelism as long as data and control dependencies are maintained.

In this study, we consider a publicly available DRP evaluation kit [9]. Figure 2.2 shows the configuration of the evaluation chip used in the kit, where DRP is integrated as an on-chip IP core. The DRP core in this particular implementation has 256 PEs, 48 two-port block memories (512 B each), 16 KB one-port block memories (8 KB each), etc. (see Fig. 2.2). The DRP core is sandwiched between ingress and egress FIFOs ($64\,\text{b}\times256\,\text{W}$ each) that are connected to an on-chip bus (this configuration is called an STP Engine in [9].) The chip is mounted on a PCI Express card (called DRP Express) for handy evaluation using off-the-shelf PCs, on which the GUI-feature-rich tool runs.

2.5 Window Join on DRP

As is well known, *join* is one of the most important operators in relational data processing [1]. The basic idea of join is to combine tuples from two tables if their key fields match and create a new table (Fig. 2.3 (a)). While

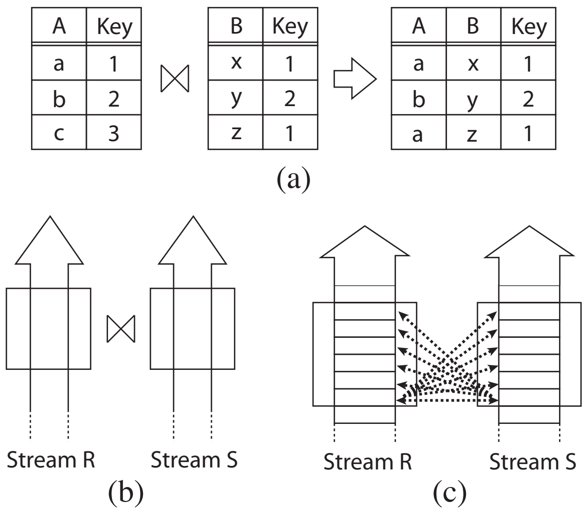


Figure 2.3: Join and window join.

the number of tuples in tables is limited, that of real-time streams is infinite; therefore, an equivalent operator in stream processing introduces a *sliding window* to limit the tuples on which it joins, *i.e.,* window join (Fig. 2.3 (b)). Since streams continuously flow through the window, window join has only to compare the newly entered tuple in the $R(S)$ stream with those in the $S(R)$ stream's window (Fig. 2.3 (c)).

2.5.1 Evaluation Strategy

Though the algorithm is simple, the window size in a realistic stream processing application tends to become very large (*e.g.,* tens of thousands), and hence, it is difficult to conduct comparisons in parallel on monolithic hardware. In view of this problem, the handshake join concept was proposed [6] and examined [13], where the window is partitioned into a series of sub-windows on which window join operators are executed in parallel. In this way, a huge window join can be divided and distributed among a number of different FPGA chips. In our window join evaluation on DRP, we adopt this concept overall, and focus on designing an efficient window join operator for each sub-window. In other words, we design a system that deals with only one sub-window on DRP and we will refer to this sub-window as simply a

Algorithm 1 Step 1 algorithm.

```
1: loop
2:   $register_R \Leftarrow ingressFIFO \Leftarrow r_t$
3:   for $i \Leftarrow 0$ to $N - 1$ do
4:     $register_S \Leftarrow ingressFIFO \Leftarrow s_{t-i}$
5:     if $regisger_S = register_R$ then
6:       $O \Leftarrow egressFIFO \Leftarrow \{register_S, register_R\}$
7:     end if
8:   end for
9:   (Repeat line 2 to 8 here reversing $R$ and $S$, and $r$ and $s$.)
10:  $t \Leftarrow t + 1$
11: end loop
```

"window" hereafter in this chapter.

Our window join case study begins with pure software code. Following our analysis of the inefficiency of synthesized hardware architectures, we gradually introduce optimization/modification to the source code until a fairly optimized code for hardware is arrived at. By tracing the code transitions carefully, we believe we can resolve the issues raised in Sect. 2.2.

We applied the following design parameters in the evaluation, considering the limited hardware resource on the DRP evaluation chip: (1) The size of a window (N) is set to 16 tuples and (2) each tuple in R and S streams is 32 b, with a 16 b *key* field and a 16 b *value* field. Here, *key* is randomly synthesized so that we can modulate the match rates. Setting of these parameters will be discussed in Sect. 2.6.

2.5.2 Step 1: Pure Software Code

We begin with a very simple C source code appropriate for software engineers. The code substitutes tuples $r(s)$ in streams $R(S)$ to $register_{R(S)}$ every time they are compared (Algorithm 1). $Register_{R(S)}$ declared in the source code is allocated in the DRP core, which results in the hardware architecture shown in Fig. 2.4. This architecture is clearly inefficient because the same tuples are moved from an external DRAM into registers inside the DRP core

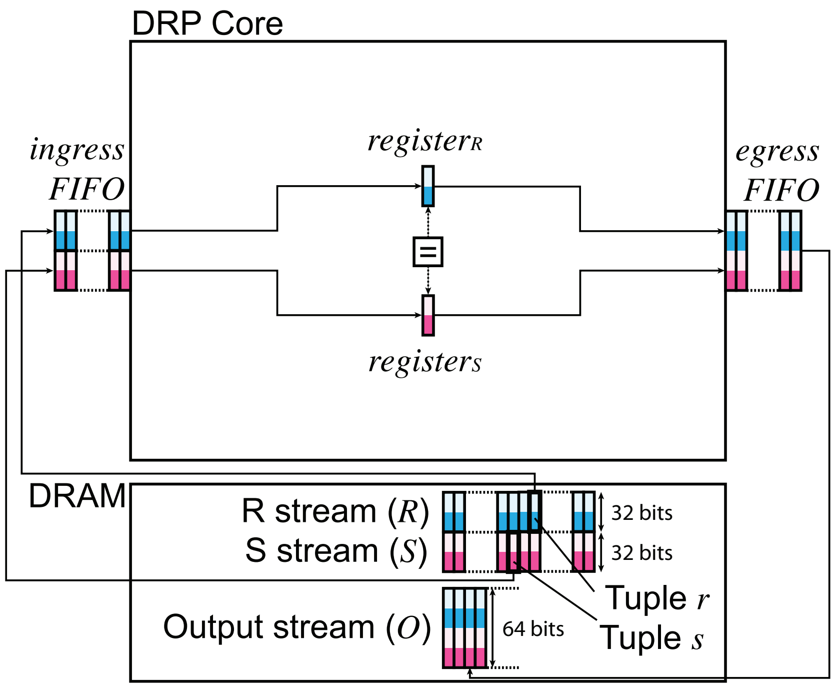


Figure 2.4: Synthesized hardware (Step 1).

repeatedly. The window join throughput (measured in terms of the incoming tuple rate) is 6.7 Mbps (Table 2.1). Table 2.1 summarizes the throughput and other performance metrics discussed further in Sect. 2.6.

2.5.3 Step 2: Sliding Window Buffer

In Step 1, each tuple is read from the DRAM several times. Since external memory access has a large delay, reducing the number of accesses should improve the window join throughput. We will reduce the number of DRAM accesses by buffering the tuples that belong to the window inside the DRP. This observation makes us to declare arrays (w_R and w_S) in the code and compare newly introduced tuples (w_{R0} and w_{S0}) with the remaining tuples in w_S and w_R (Algorithm 2). The resultant architecture shown in Fig. 2.5 (note that the shift registers for w_R and w_S are synthesized) shows 17-fold

Algorithm 2 Step 2 algorithm.

```
1: loop
2:   for $i \Leftarrow N-1$ to 1 do
3:     $w_{Ri} \Leftarrow w_{R(i-1)}$
4:     $w_{Si} \Leftarrow w_{S(i-1)}$
5:   end for
6:   $w_{R0,S0} \Leftarrow ingressFIFO \Leftarrow \{r_t, s_t\}$
7:   for $i \Leftarrow 0$ to $N-1$ do
8:     if $w_{R0} = w_{Si}$ then
9:       $O \Leftarrow egressFIFO \Leftarrow \{w_{R0}, w_{Si}\}$
10:    end if
11:    if $w_{S0} = w_{Ri}$ then
12:      $O \Leftarrow egressFIFO \Leftarrow \{w_{Ri}, w_{S0}\}$
13:    end if
14:  end for
15:  $t \Leftarrow t+1$
16: end loop
```

throughput improvement.

2.5.4 Step 3: Parallel Output Buffer

The $2N$ comparisons in Algorithm 2 could have been executed in parallel by allocating $2N$ comparators inside the DRP core since there is no data and control dependency among them. However, the synthesized hardware (Fig. 2.5) fails to exploit this parallelism. The reason is *egressFIFO*, which can take in one data item at a time, while each of the $2N$ comparisons may produce an output tuple.

A solution to this problem is to declare $2N$ *output buffers* in the source code (Algorithm 3), so that each of the potential matches can possess a buffer to store the result. Hence, the tool can safely parallelize the $2N$ comparisons. $compare_1(\cdot)$ does this $2N$ comparisons, concatenates the tuples whose keys matched, and returns them. One of the parallel *buffers*, on the other hand, will get the opportunity to write a tuple into *egressFIFO* ($pop(\cdot)$) every main

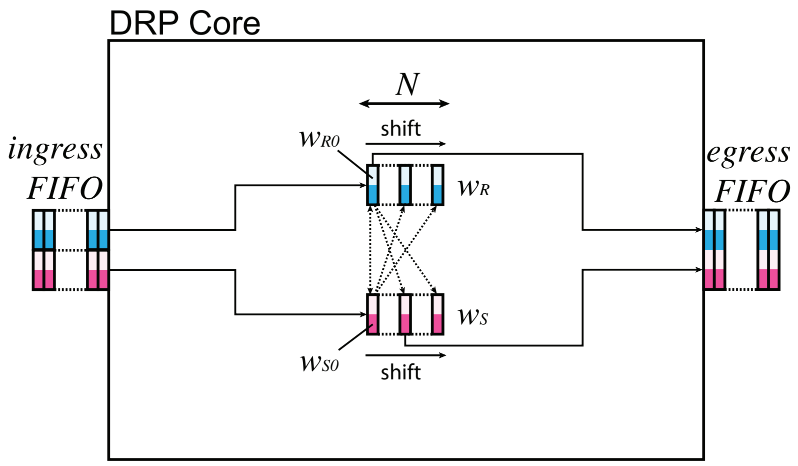


Figure 2.5: Synthesized hardware (Step 2). Although the tuples still comes from the external DRAM, it will not be shown in this figure nor the subsequent figures in order to save space.

loop cycle (this is realized by the **for** loop that starts at line 3 and ends at line 8). Since the *buffer*s are implemented as stacks, we use *push* and *pop* for accessing them. The idea is essentially to decouple parallel comparisons from serial *egressFIFO* access, which is feasible as long as the bandwidth at *egressFIFO* is lower than that at *ingressFIFO*, *i.e.,* a low match rate. The match rate issue will be discussed in Sect. 2.5.9 and 2.6.1.

We expected the tool to synthesize an architecture that achieves drastic performance improvement, as shown in Fig. 2.6. We assigned a directive (not mentioned in Algorithm 3, however, for better visibility) so that the buffers ($buffer_{1\sim 2N}$) will be realized by registers. The throughput, however, actually reduced to 8 Mbps due to the comparisons not being done in a single cycle. This is because the parallel output buffer required more registers than what is available in a single context. When a resource shortage takes place, the synthesis tool divides the context into several contexts so that the resource usage of each context does not exceed the limit.

Algorithm 3 Step 3 algorithm.

```
1: loop
2:   for $i \Leftarrow 1$ to $2N$ do
3:     for $k \Leftarrow N-1$ to $1$ do
4:       $w_{Rk} \Leftarrow w_{R(k-1)}$
5:       $w_{Sk} \Leftarrow w_{S(k-1)}$
6:     end for
7:     $w_{R0,S0} \Leftarrow \mathit{ingressFIFO} \Leftarrow \{r_t, s_t\}$
8:     $\mathit{buffer}_{1\sim 2N} \Leftarrow_{push} \mathit{compare}_1(w_R, w_S)$
9:     $O \Leftarrow \mathit{egressFIFO} \Leftarrow \mathit{pop}(\mathit{buffer}_i)$
10:    $t \Leftarrow t+1$
11:  end for
12: end loop
```

2.5.5 Step 4: Match Table

Since registers and block memories are observed to be critical resources in parallelizing an window join operation, it becomes crucial to minimize buffering requirements. Generally speaking, registers can be accessed with more parallelism, but are more expensive than block memories. Therefore, it is important to examine the usage of registers relatively more carefully. This observation leads us to re-write the code, as shown in Algorithm 4. Here, $w_{R(S)}$ receives only the *key* fields, while whole tuples go to $\mathit{storage}_{R(S)}$. The function $\mathit{compare}_2(\cdot)$ does the $2N$ comparisons between w_R and w_S and returns match results that is encoded to 32-bit-width vector. This bit vector is stored to *table*, which in turn points the location of matched tuples in $\mathit{storage}_{R(S)}$. The function $\mathit{output}_1(\cdot)$ then returns the matched tuples to *egressFIFO* by acquiring appropriate tuples, which are found by decoding the bit-vector, from $\mathit{storage}_{R(S)}$. Variable l in the figure represents the difference in time between the matched tuples. It is used for encoding and decoding the location of them in the *storage*s.

The tool synthesized the hardware architecture shown in Fig. 2.7. Note here that registers are used only for *keys* that need to be compared in parallel, and buffering requirement is reduced drastically by avoiding redundantly

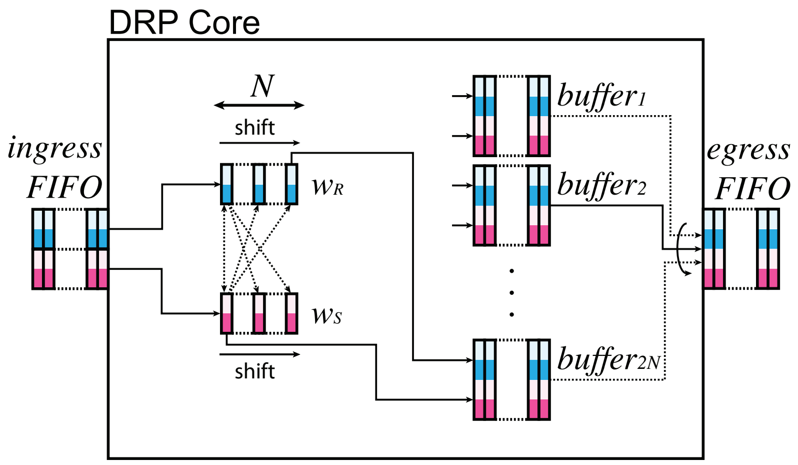


Figure 2.6: Expected hardware (Step 3).

duplicating tuples to multiple output buffers (L is set to the length of *ingressFIFO*, which is 256). The throughput recovers to 103 Mbps, since now it can conduct $2N$ comparisons in parallel.

The entities in the figures are realized by registers if it is depicted with single lines, and block memories if depicted with double lines. Moreover, arrays in the codes can be synthesized either by registers or block memories, and the programmer can specify which to use in the code by writing directives in the form of comments.

2.5.6 Step 5: Chunk Data Prefetching

Now that the register/memory shortage is solved and comparisons can be fully parallelized, the next obvious performance bottleneck is the stream input to the DRP core. Thus far, source codes brought R and S tuples one at a time, incurring idle time in waiting for the arrival of the tuples. A solution is simply to read tuples in *chunks* so that *ingressFIFO* almost always has enough input tuples to join. This is realized by calling a DRP specific API which orders the Memory Controller to burst access the external DRAM, taking data size for its parameter. For this purpose, we set the size of each chunk to that of *ingressFIFO*. As a result, the throughput increases threefold than obtained in the previous step.

Algorithm 4 Step 4 algorithm.

```
1: loop
2:   $optr \Leftarrow 1$
3:   for $iptr \Leftarrow 1$ to $L$ do
4:     for $i \Leftarrow N-1$ to 1 do
5:       $w_{Ri} \Leftarrow w_{R(i-1)}$
6:       $w_{Si} \Leftarrow w_{S(i-1)}$
7:     end for
8:     $w_{R0,S0}, storage_{R,S}[iptr] \Leftarrow ingressFIFO \Leftarrow \{r_t, s_t\}$
9:     $table[iptr] \Leftarrow compare_2(w_R, w_S)$
10:    $O \Leftarrow egressFIFO \Leftarrow output_1(table[optr], storage_{R,S})$
11:    if $table[optr]$ has no positive bits then
12:      $optr \Leftarrow optr + 1$
13:    end if
14:    $t \Leftarrow t + 1$
15:  end for
16: end loop
```

2.5.7 Step 6: Parallel Table Lookup

In Steps 4 and 5, there are 32 bit-slots in a single *table* entry (a bit-vector). Though the bit-vector occupies little space, decoding it is tedious for a coarse-grained architecture such as DRP. Actually, it becomes a critical path that limits the maximum operation frequency. To resolve this problem, we split the table into four 8 b tables that can be handled relatively fast. This modification enables us to execute $compare_2(\cdot)$ and $output_1(\cdot)$ in Algorithm 4 as four parallel 8 b operations that decrease the circuit delay.

In Step 5, we were treating the 32-bit column as a single variable. Therefore, the circuit became a five ($= \log_2 32$) stages' binary tree of `OR` gates that aggregates the comparison results. Each of the comparison results is shifted before being aggregated at minimum zero bits and at maximum 31 bits uniquely according to the location where the match was found. Since DRP has a 8-bit architecture, each `OR` gate and shifter are realized by combining four 8-bit PEs. Combining multiple PEs and realizing a PE with a larger

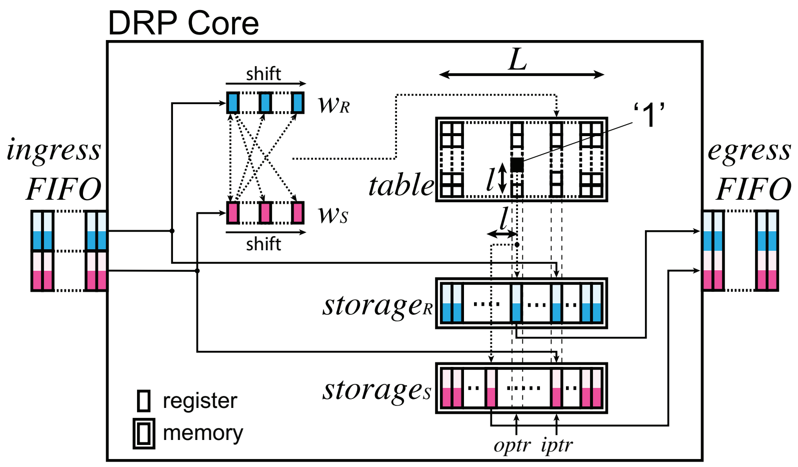


Figure 2.7: Synthesized hardware (Step 4).

bit width automatically is one of the HLS tool's functionalities. Overall, there was delay of six gate levels.

On the other hand, the idea of the circuit in this step is basically the same as the previous step, except that an column is treated as an 8-bit variable. Therefore, there will be one shifter and only three ($= \log_2 8$) stages' binary tree of `OR` gates. Of course, this logic will be parallelized with four ($= 32/8$) identical paths, but the circuit delay will be the same. Hence there was a delay of four gate levels. Note that each gate has less delay than a gate used in Step 5, because the gate in this step does not combine multiple PEs and used as a larger gate with wider bit width.

However, since the tool divided these circuits into several contexts, the actual inter-register delay of these circuits are not so different. Instead, the number of contexts of these circuits was four in Step 5, and two in Step 6. This is the major cause of the difference between the throughput of the two architectures. As a result, the throughput improved by 83%.

2.5.8 Step 7: Loop Folding (Pipelining)

As explained in Sect. 2.4, the tool features a *loop folding* option, which essentially allows each iteration of the loop to start as early as possible, overlapping with previous iterations, as far as data/control dependencies

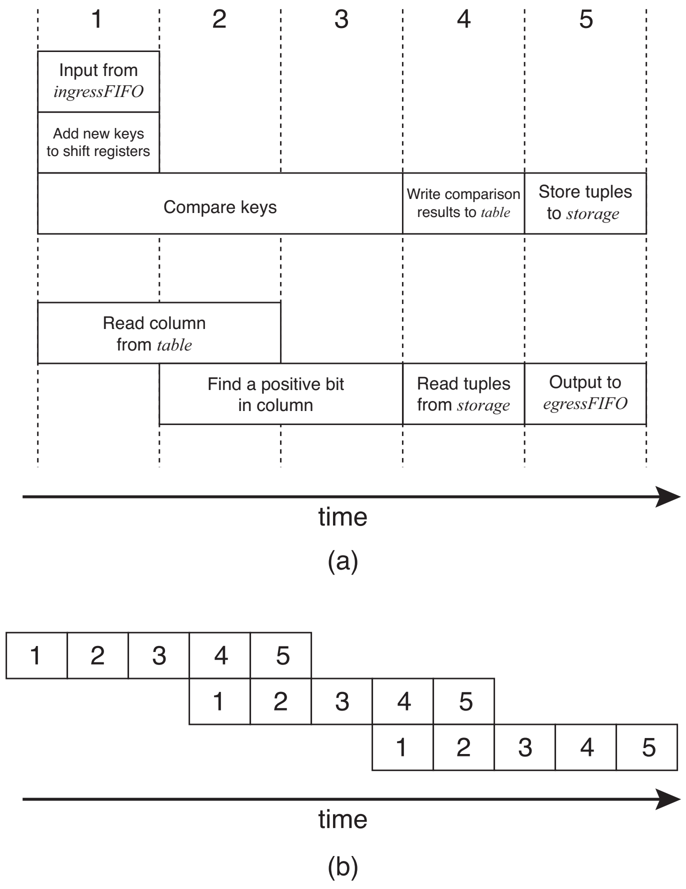

1
2
3
4
5
Input from *ingressFIFO*
Add new keys to shift registers
Compare keys
Write comparison results to *table*
Store tuples to *storage*
Read column from *table*
Find a positive bit in column
Read tuples from *storage*
Output to *egressFIFO*
time
(a)
time
(b)

Figure 2.8: Timing chart of the folding.

allow. Though this is a powerful high-level synthesis feature that can extract additional parallelism, there are several conditions that should be met by the source code, *e.g.*, the code should have no inner-loops. Another tool option, *loop unrolling*, is also exploited to meet this condition here.

Figure 2.8a shows the timing of the functionalities in the main loop (each iteration takes five cycles). The tool generated a circuit that folds this loop, initiating iterations every three cycles (Fig. 2.8b). The number of cycles per main loop initiation improved to three, which was four in the previous step. The reason why the number of cycles in the main loop increased is because the tool coordinated the states by dividing them in order to maintain data dependencies. As a result, the throughput improvement at this step was

Algorithm 5 Step 8 algorithm.

```
1:  loop
2:     for $iptr \Leftarrow 1$ to $L$ do
3:        for $i \Leftarrow N-1$ to 1 do
4:           $w_{Ri} \Leftarrow w_{R(i-1)}$
5:           $w_{Si} \Leftarrow w_{S(i-1)}$
6:        end for
7:        $w_{R0,S0}, storage_{R,S}[iptr] \Leftarrow ingressFIFO \Leftarrow \{r_t, s_t\}$
8:        $buf_{1\sim 2N} \Leftarrow_{push} compare_3(w_R, w_S)$
9:        $t \Leftarrow t+1$
10:    end for
11:    for $i \Leftarrow 1$ to $2N$ do
12:       while $buf_i$ is not empty do
13:          $O \Leftarrow egressFIFO \Leftarrow output_2(pop(buf_i), storage_{R,S})$
14:       end while
15:    end for
16: end loop
```

87%.

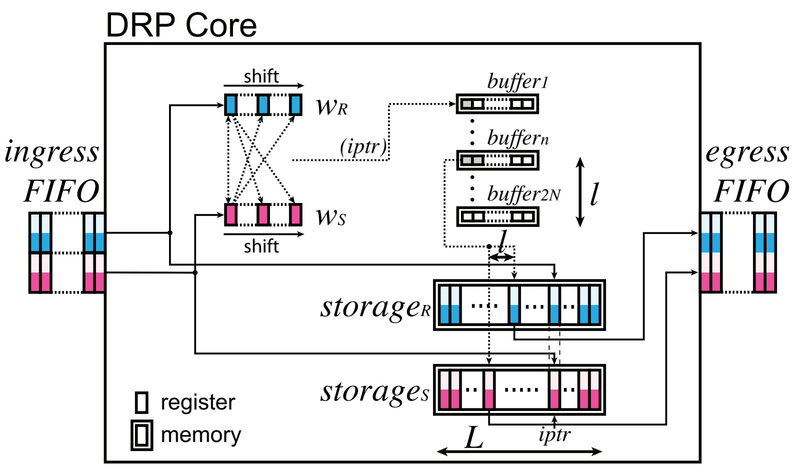


Figure 2.9: Synthesized hardware (Step 8).

Table 2.1: Performance metrics throughout the optimization process.

	Thrpt		Freq	States	Idle	Resources				
#	[Mbps]	States	[MHz]	/tuple	[%]	ALU	2pM	1pM	Reg	Remarks
1	6.8	19	53	98.6	80.2	28	0	0	88	Pure software code
2	113	105	55	18.8	38.8	119	0	0	586	Sliding window buffer
3	8.6	48	49	349.0	3.2	553	0	0	2020	Parallel output buffer
4	103	23	53	14.3	56.6	149	12	0	235	Match table
5	311	32	37	7.4	3.2	109	12	0	326	Chunk data prefetching
6	571	38	40	4.3	5.4	114	12	0	255	Parallel table lookup
7	1068	32	56	3.1	7.7	215	16	0	372	Loop folding (pipelining)
8	1469	113	38	1.4	14.7	320	32	16	315	Low match rate opt.

2.5.9 Step 8: Low Match Rate Optimization

Though not explained explicitly, Steps 4 through 7 do not rely on the low match rate assumption unlike Step 3: when the match rate is high and $storage_{R(S)}$ becomes full ($iptr = L$), $output_2(\cdot)$ runs until $optr$ reaches L and $storage_{R(S)}$ becomes empty. However, if a low match rate is expected between the two streams, a different optimization strategy can be considered; we consider separating the $output_2(\cdot)$, which runs scarcely, from the main loop so that the loop can run faster.

In order to realize the architecture shown in Fig. 2.9, we rewrite the source code as in Algorithm 5. Unlike the source codes until Step 7, now it has two separate **for** loops. In addition, as a minor improvement, we replace *table* with parallel *buffers* (stacks) that hold indices of matched tuples in $storage_{R(S)}$. As introduced in Sect. 4.5, the variable l here represents the difference in time of matched tuples and used for encoding the location of them in the storages. The function $compare_3(\cdot)$ does the $2N$ comparisons between w_R and w_S and pushes the value of $iptr$ to the *buffer*s which correspond to where the match was found. The function $output_2(\cdot)$ pops one of the *buffer*s and retrieves the location of matched tuples in the *storage*s. In the synthesized architecture (Fig. 2.9), the first loop is successfully folded by the tool and is executed in a single cycle. The throughput is improved by 38%.

2.6 Discussion

2.6.1 Optimization Overview

Table 2.1 summarizes the performance metrics throughout the optimization process. The reported input throughput ("Thrpt") was measured by setting the match rate to 0.1%. "States" indicates the total number of states in the synthesized FSM, while "States/Tuple" is the number of dynamic states per incoming tuple. Since the DRP can run one state per clock cycle, "Freq" (clock frequency) divided by "States/Tuple" determines the throughput if there is no "Idle" time. Here, "Idle" is due to an empty *ingressFIFO* and/or a full *egressFIFO*. Hence, code optimization is performed to improve these three metrics.

As shown in Table 2.1, Step 1 clearly suffers from excessive "idle" cycles. This is because a certain number of idle cycles occurs each time the DRP Core accesses the DRAM, and the system brings the same tuples several times. As we explained in Sect. 2.5.3, we reduced the numbers of DRAM access in Step 2. You can see the percentage of idle cycles reduced just by reducing the number of DRAM accesses. However, not being able to compare the keys in the shift registers in parallel, Step 2 still takes about 19 states on average per tuple.

Table 2.1 also shows that the parallel *buffer*s we employed in Step 3 required 2020 registers whereas only 512 are available. Moreover, the number of ALUs (553) was also exceeding the limit (512). This resource shortage resulted in context division which further led to throughput reduction. In Step 4, the architecture which uses block memories reduced the number of required ALUs and registers. You can see in Table 2.1 that two port memories are newly used and the number of ALUs and registers (149 and 235 respectively) reduced. The *table* and the *storage*s were realized by two port block memories so that the input, comparison and output processes do not interfere with each other.

The reason that the percentage of idle cycles increased in Step 4 is that the number of idle cycles caused by accessing the DRAM did not change although the total processing time decreased. Likewise, the reason the percentage of the idle cycles increased after hitting the bottom in Step 5 is that there is a minimum amount of idle cycles that occurs at the beginning and the end of the entire process which cannot be eliminated even by burst accessing the DRAM. Therefore, as the number of the entire processing cycles reduced, the ratio of the idle cycles increased.

In Step 8, we detached the output operations (checking the *buffer*s, selecting the matched tuples in the *storage*s, and outputting the tuple to *egress-FIFO*) from the main loop. As a result, the number of states in the main loop (line 2 to 5 in Algorithm 5) reduced and therefore achieved almost 1 state/tuple. On the other hand, the output operations became more complex and the number of states for the operations increased. (Note that the number of states described in Table 2.1 includes all the states regardless of the

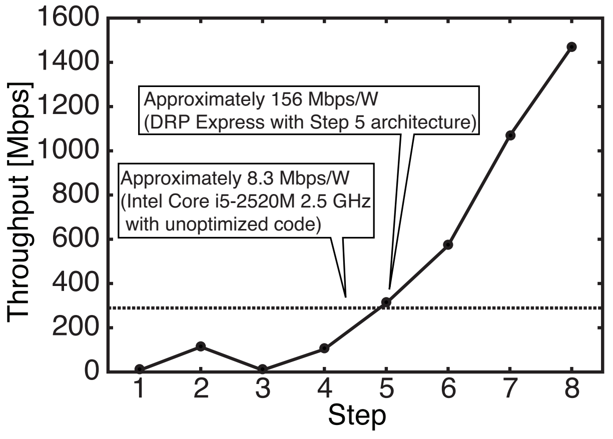


Figure 2.10: Throughput improvement.

state being a part of the main loop or not.) However, when the match rate of the tuples between the two streams is very low, the output operations run scarcely. Therefore, states/tuple decreased as a whole in spite of the overall number of states increased.

Since *storage*s no longer needs to be accessed in parallel in Step 8, the *storage*s use one port block memories. For the same reason, the *buffer*s could be realized by using one port block memories, but two port block memories were used. The reason for this is that although there had to be 32 *buffer*s in parallel, there were only 16 one port block memories in the DRP Core. Therefore, although the *buffer*s do not have to be accessed in parallel, they are implemented with two port block memories, which are more sufficient.

2.6.2 Performance

Figure 3.5 highlights the throughput improvement. Step 8 is 216 times faster than Step 1. Looking at this result, we can say that we have good control of DRP only with C codes. For reference, we evaluated a straight forward (unoptimized) window join code written in C running on Core i5-2520M (2.5 GHz). The horizontal line in Fig. 3.5 indicates the throughput of the CPU code, which was 290 Mbps. You can see that the throughput of DRP does not exceed the CPU until Step 5. Before Step 5, we had made several optimizations that require hardware development knowledge. For example, in Step 3, we employed parallel buffers in order to make the comparisons in the window in parallel. In Step 4, we tried to detour the lack of registers by utilizing block memories. This means that the programmer should have some hardware knowledge to make a DRP code run faster than a primitive CPU code.

We will not discuss the maximum throughput of DRP and CPU, since we have not fully optimize the CPU code. However, comparing Step 5 and the CPU, which have almost the same throughput, throughput/power of DRP was about 19 times higher than that of the CPU. We assumed that their power consumption to be their Max TDP, 2 W for DRP and 35 W for CPU, not having facilities to measure power consumption.

Our work focuses not on competing with window join systems implemented on other reconfigurable hardware, but on illustrating how software developers use C-based HLS. However, for your reference, the throughput of handshake join algorithm implemented on an FPGA [13] was a little over 1 [M tuples/s] whereas ours (Step 7 architecture) was approximately 12.5 [M tuples/s] when the match rate between two streams are 10%. The relatively higher throughput of our work is due to small tuple size (32 bit width), and small window size (16 tuples). The tuple size was 64 bit width and the window size was 512 in [13], which will enormously reduce the throughput of our system if adopted.

2.6.3 Lessons learned

As discussed in Sect. 2.2, a key for more widespread adoption of reconfigurable hardware acceleration is to increase its availability to software engineers. From the in-depth examination of the window join optimization process, we would like to propose the following "Five Awarenesses" as a must-have for a stream processing programmer trying to exploit reconfigurable hardware acceleration.

- **I/O Awareness:** Since I/O tends to be a throughput bottleneck when designing a high-throughput stream processing system, a programmer should find a way to maximize I/O usage in order to maximize the throughput. For example, in Step 2, we reduced the number of accesses to the same tuples in the streams, which improved the throughput by 17 folds. We also reduced the number of idling cycles by prefetching the tuples, and improved the throughput by 200%. Suitable buffering leads to maximizing I/O throughput and helps relieving the bottlenecks in the system.

- **Buffer Awareness:** Buffers are useful for dividing the system into input, intermediate and output processes. As we mentioned above, by doing this, we can focus on relieving the bottlenecks in each processes. For example, in Step 2, we reduced the DRAM access by buffering the input tuples. In Step 4, $w_{R,S}$ and $storage_{R,S}$ isolate the input and output processes from the comparison process. Therefore, we just had to focus on increasing the parallelism and reducing the iteration interval of the main loop. The programmers should also be aware that FIFOs are not the only buffering method; they should find the appropriate buffering method for the application.

- **Resource Amount Awareness:** The programmer should reconsider the buffering architecture when the resources are lacking. This is because input, intermediate and output processes share the hardware resources although they are isolated by buffers and the programmer can focus on each of them separately. In order to gain parallelism in Step 3, we tried to buffer the output tuples with parallel buffers. However, we had to reconsider the architecture since the ALUs and registers were not enough to implement the parallel buffers. Creating a suitable architecture that the resources are used in good balance enables programmers to focus on optimizing each process.

- **Loop Awareness:** After dividing the processes into input, intermediate and output with well balanced resource usage, and the input and the output processes are no longer the bottlenecks, the loops in the intermediate process becomes the bottleneck. One of the common and powerful techniques of improving the parallelism of the loops is to fold them, as we have done in Step 7. However, to fold the loops, some requirements should be met. For example, the inner loops should be

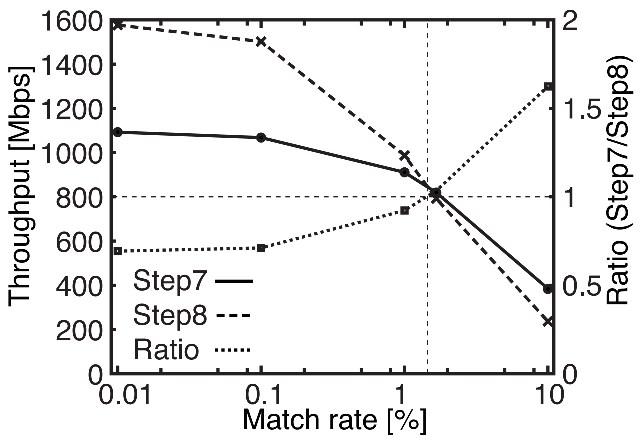


Figure 2.11: Match rate dependency (Step 7 vs 8).

unrolled, which can be done by assigning a directive to the loop, and data dependency should be resolved by suitably choosing registers, one port block memories or two port block memories for storing the data. Improving the parallelism in the main loop requires laborious work but greatly improves the performance of the system.

- **Resource Type Awareness:** Knowledge about the underlying resource, *e.g.*, ALU and internal memory granularity, also helps optimizing the source code to fit the underlying hardware platform. For example, we were able to reduce the delay of the circuit by dividing the *table* into four parallelized partial tables utilizing the DRP's 8-bit architecture. In the later steps of the optimization, especially the reducing phase of loop iteration interval, reducing the delay by considering the resource characteristics eventually leads to reducing the number or states in the main loop. Although resource type is scarcely considered when writing software, programmers should be aware of when developing a hardware.

To avoid confusion, it is important to ensure that all the window join codes are written in C, and that the tool can hide hardware details such as FSM synthesis, ALU mapping, register/memory mapping, place and routing, and I/O allocation. Nevertheless, the above "Five Awarenesses" are important for the optimization process. The question is whether the tool will become smart enough in the future such that these awarenesses will not be required. To the best of our knowledge on window join, this is impossible, since those steps involve architectural creation in addition to technical optimization (the latter should become more transparent to programmers as the tools become smarter).

In spite of expecting the tools to create new architectures, we believe that what we really need is to establish a stream-oriented programming model, an abstraction layer to a programmer, that allows the "Five Awarenesses" to be naturally accomplished. Hence, our future work is to 1) investigate more stream processing examples on the DRP and other reconfigurable architectures such as FPGA to validate the above proposal, and to 2) establish a programming model to cover essential features of stream processing applications.

2.6.4 Adaptive Stream Processing

Figure 2.11 compares the throughput of Steps 7 and 8, the latter being optimized for a low match rate. For a match rate more than 1.5%, Step 7 performs better than Step 8. The ratio of Step 7 to Step 8 is 0.69 at 0.1% and 1.62 at 10%. Since the DRP core can reconfigure its configuration cycle-to-cycle, it can instantly switch between Steps 8 and 7 when, for example, the match rate between two input streams dynamically changes between low and high states. As this example demonstrates, dynamic reconfiguration can be utilized for adaptive stream processing, which will be examined using more realistic stream processing applications in future.

2.7 Summary

The objectives of our work in this chapter were to clarify the difficulties of developing a dedicated hardware with a software programming language and an HLS tool, to find the difficulties for software developers to design hardware, and to define the viewpoint that software developers should have in order to develop hardware. To achieve these objectives, we picked stream processing, especially window join which is a costly operator of stream processing, as a case study and implemented it on DRP, which is suitable to stream processing due to its dynamic reconfigurability.

Through repeated experiments, we conclude that 1) a state-of-the-art high-level synthesis tool is sufficiently powerful for writing all source code in C, while hiding hardware details from the programmer on one hand, and extracting inherent parallelism from the source code on the other, 2) a throughput improvement of two orders of magnitude can be achieved by code optimization, 3) "Five Awarenesses" (I/O, Buffer, Resource Amount and Type, and Loop) are critical for a software engineer to carry out optimization, and additionally, 4) the dynamic reconfiguration feature is attractive for coping with the dynamically changing nature (adaptiveness) of stream processing.

Our analysis on window join demonstrated that establishing a programming model that provides an abstraction layer to the software engineer is essentially important for wider adoption of reconfigurable hardware acceleration. As future work, we will examine our proposal in further detail with more applications and on wider range of platforms including commercial FPGA.

Bibliography

[1] Arasu, A., Babu, S. and Widom, J.: The CQL continuous query language: semantic foundations and query execution, *The VLDB Journal*, Vol. 15, No. 2 (2006).

[2] Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S. and Doo, M.: SPADE: the system s declarative stream processing engine, *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (2008).

[3] Koomey, J. G.: Worldwide electricity used in data centers, *Environmental Research Letters*, Vol. 3, No. 3 (2008).

[4] Inoue, H., Takenaka, T. and Motomura, M.: 20Gbps C-Based Complex Event Processing, *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications (FPL)* (2011).

[5] Müller, R., Teubner, J. and Alonso, G.: Streams on Wires - A Query Compiler for FPGAs, *Proceedings of the VLDB Endowment*, Vol. 2, No. 1 (2009).

[6] Teubner, J. and Mueller, R.: How soccer players would do stream joins, *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (2011).

[7] Takagi, M., Takenaka, T. and Inoue, H.: DYNAMIC QUERY SWITCHING FOR COMPLEX EVENT PROCESSING ON FPGAS, *Proceedings of the 2012 22nd International Conference on Field Programmable Logic and Applications (FPL)* (2012).

[8] Motomura, M.: A dynamically reconfigurable processor architecture, *Microprocessor Forum* (2002).

[9] Motomura, M.: STP Engine, a C-based Programmable HW Core featuring Massively Parallel and Reconfigurable PE Array: Its Architecture, Tool, and System Implications, *Proceedings of the COOL Chips XII* (2009).

[10] Miyoshi, T., Kawashima, H., Terada, Y. and Yoshinaga, T.: A Coarse Grain Reconfigurable Processor Architecture for Stream Processing Engine, *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications (FPL)* (2011).

[11] Toi, T., Nakamura, N., Kato, Y., Awashima, T., Wakabayashi, K. and Jing, L.: High-level synthesis challenges and solutions for a dynamically reconfigurable processor, *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design (ICCAD)* (2006).

[12] Toi, T., Awashima, T., Motomura, M. and Amano, H.: Time and Space-multiplexed Compilation Challenge for Dynamically Reconfigurable Processors, *IEEE International Midwest Symposium on Circuits and Systems* (2011).

[13] Oge, Y., Miyoshi, T., Kawashima, H. and Yoshinaga, T.: Design and Implementation of a Merging Network Architecture for Handshake Join Operator on FPGA, *2012 IEEE 6th International Symposium on Embedded Multicore SoCs (MCSoC)* (2012).

Chapter 3

Acceleration by a Hybrid Approach of HLS and RTL

3.1 Introduction

In the previous chapter, we analyzed the difficulties that software developers encounter when developing application specific hardware. As a result, we found there are five perspectives that they should know. Several methods that have been proposed to eliminate difficulties by virtualizing some of the perspectives. CoRAM [1] is one such method. CoRAM alleviates hardware development cost by virtualizing the I/O to external DRAM and processors from the core logic. This method diminishes the I/O awareness from the five awarenesses that we proposed in the previous chapter. However, there four awarenesses remain. The key for eliminating several awarenesses at the same time is to suppose a specific application field to accelerate.

In this chapter, we propose a method that simplifies the development and use of dedicated hardware by a slightly different approach from this approach that tries to eliminate the awarenesses that are necessary for hardware development. More specifically, we propose a system that converts StreamSQL, a stream processing specific language, to hardware. This system takes two steps. In the first step, we parse the query written in StreamSQL to C code that intended for high-level synthesis, and the second, the C code compiles to hardware configuration by an HLS tool. In this way, we enable software engineers to develop hardware with software and relive the cost of porting the system to various hardware platforms.

A system that synthesizes hardware from StreamSQL has already been developed [7]. Although this approach restricts the application of product hardware, the developer needs less knowledge about hardware development compared to HLS. Because this approach is intended to compile SQL queries directly to hardware configurations, however, developing such a compiler would take much effort to support various reconfigurable hardware.

In order to overcome these difficulties of HLS and SQL-to-hardware compiler, a method that uses both of these in combination has been proposed [8]. This approach first uses a parser that converts an SQL query to an HLS code written in C, and then uses an HLS tool that compiles the code to a hardware configuration. Although this method was proposed as a part of a larger system that uses an FPGA, it successfully reduced the workload of development of SQL-to-hardware compiler by giving over the hardware configuration process to the HLS tool, rather than doing it manually.

Because the HLS tool undertakes the hardware specific configuration, this method should be able to be applied to other hardware. Therefore, in this chapter, we try to apply this method to another reconfigurable hardware, dynamically reconfigurable processor (DRP), and focus on compiling basic SQL queries that were used in [7] as our primary evaluation.

Our contributions in this chapter are as follows:

- Evaluating how well the SQL-to-C parser extracts DRP's potential
- Verifying whether the SQL-to-C parser provides an SQL-to-hardware compiler with portability to DRP by using it in combination with an HLS tool
- Pointing out what should developers of an SQL-to-C parser should be aware of when porting the parser to another environment

The organization of this chapter is as follows. In Section 3.2, we introduce other works that try to generate a hardware configuration from SQL queries. Before we explain our method, we briefly introduce StreamSQL, which we use as our description language of the product of our system in Section 3.3. Then,

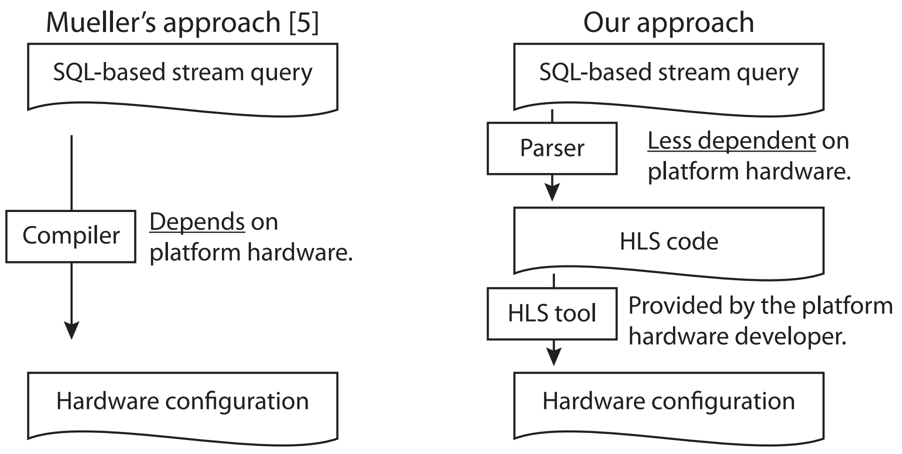


Figure 3.1: Comparison of conventional and our approaches.

we provide details of the proposed method in Section 3.4 and evaluate it in Section 3.5. After we discuss the results in Section 3.6, we will summarize our work in Section 3.7.

3.2 Related Work

Mueller et al. proposed a system called *Glacier*, which compiles SQL-based stream queries to high-throughput hardware configurations [7]. This work took five basic queries (four of which, Q_1 to Q_4, are listed in Fig. 3.2) as application examples, and used FPGA as its hardware platform. It essentially proposed how to map each of the SQL primitives to a corresponding hardware template, and then connect them as they are specified in queries provided to the system. This idea was extended in many ways and became the basis of several related works such as [5].

One study, however, has proposed an advanced framework for compiling SQL based continuous query with user-defined C/C++ functions. The system realizes 20Gbps bit stream processing on the FPGA [3] and exploits HLS not only for compiling the user-defined C/C++ functions but also for the C codes that are parsed from standard SQL queries by their original parser. Our work tries to apply this parsing method to DRP, jointly using the HLS

```
         CREATE INPUT STREAM Trades (
           Seqnr  int,         -- sequence number
Schema:    Symbol string(4),   -- valor symbol
           Price  int,         -- stock price
           Volume int)         -- trade volume

         SELECT Price,Volume
  Q1:      FROM Trades
          WHERE Symbol="UBSN"
           INTO UBSTrades

         SELECT Price,Volume
  Q2:      FROM Trades
          WHERE Symbol="UBSN" AND Volume>100000
           INTO LargeUBSTrades

         SELECT count() AS Number
           FROM Trades [SIZE 600 ADVANCE 60 TIME]
  Q3:     WHERE Symbol="UBSN"
           INTO NumUBSTrades

         SELECT wsum(Price,[.5,.25,.125,.125] AS Wprice
           FROM (SELECT * FROM Trades
  Q4:                WHERE Symbol="UBSN")
                [SIZE 4 ADVANCE 1 TUPLES]
           INTO WeightedUBSTrades
```

Figure 3.2: Example queries and schema of incoming stream.

tool bundled to it.

3.3 SQL-Based Stream Processing Language

In our study, we use SQL-based stream processing language as an application description language. Fig. 3.2 shows some example queries. The schema in Fig. 3.2 specifies the fields that the stream named “Trades” has. The tuples from the stream are processed according to the queries (Fig. 3.2, Q_1 to Q_4) which consist of the following clauses:

- `SELECT` clause specifies the fields of the outgoing stream.
- `FROM` clause specifies the input stream.
- `WHERE` clause specifies the conditions for selecting the tuples.
- `INTO` clause specifies the name of the outgoing stream.

Additionally, some queries have aggregation functions in their `SELECT` clauses (Q_3 and Q_4 in Fig. 3.2). Aggregation functions calculate some measures from a certain range of tuples in the stream, which are specified by a window written in the `FROM` clause with its size and sliding interval (e.g. Q_3 counts the number of tuples whose symbol is “UBSN” within 600 s, and Q_4 calculates the weighted sum of the stock prices from the previous four tuples).

3.4 Our Approach

Generally, a hardware development procedure can be divided into two stages; the first is to fixes the processing architecture which involves specifying the I/O or where to pipeline, and the second is to arranges the wires, registers, and memories so that the circuit meets the requirements such as delay or resource amount.

As in [8] the parser converts SQL to HLS code written in C, and then the generated HLS code is compiled to hardware configuration by HLS tool (Fig. 3.1). In other words, the compilation process is divided into two stages:

1. Parser: specifies the architecture that is suited to SQL-based stream processing

2. HLS tool: synthesizes and optimizes hardware configurations

In this study, we use an existing HLS tool that is bundled with the DRP; therefore, hereafter in this chapter, we look only into the parser.

3.4.1 Generalization of Queries into Abstract Hardware

We first generalize the query structure to abstract hardware modules in order to enable any queries to be converted into hardware configurations.

First of all, whether or not there are any subqueries inside the query is determined. To find a subquery, we look inside the `FROM` clause. If there is a subquery (Fig. 3.3a), there is going to be two query modules between the input and output (Fig. 3.3b); otherwise, there will be only one query module.

The query module is described in Fig. 3.4a. It consists of three modules;, selection, slide timing, and aggregation. However, if the query does not use aggregation, the query module only has the selection module. The output of the selection module is sent out as an output tuple in such case (e.g. Q_1 and Q_2). The selection module asserts the "selected" signal if the tuple satisfies the conditions specified in `WHERE` clause (e.g. `Symbol="UBSN" AND Volume>100000`, in Q_2), or otherwise negate it. The slide-timing module monitors the incoming tuples and notifies the aggregation module of the slide timing by asserting the "slide" signal. The slide timing can be detected by counting the valid tuples, when tuple-based windowing is used, or by monitoring the timestamp of the incoming tuples, when time-based windowing is used.

The aggregation module consists of three components, a shift register (`w[0]` to `w[N-1]`), an aggregation unit, and a pre-calculation unit (Fig. 3.4b). The shift register holds the values that the aggregation unit uses. The number of registers is determined by dividing the window size by the sliding interval. When the registers receive the "slide" signal from the slide-timing module, each register sends its content value to the next register. The aggregation unit collects the values from the shift register and outputs the aggregation result. The pre-calculation unit updates the content of `w[0]` whenever there is a valid input tuple. What the pre-calculation unit does depends on the aggregation specified in the query. For example, when `count(·)` is specified (Q_3), the pre-calculation unit increments the value in `w[0]`, initializing it by zero when the "slide" signal is asserted. The aggregation unit, then, collects the values in the registers and adds them up.

3.4.2 Mapping Abstract Hardware to C Code

In the generated C code, the main function first calls the function that corresponds to the most innermost query (Fig. 3.3c), giving the input tuple that was received in `receive_tuple(·)`. The query function calls the query function that corresponds to the next inner query in the return statement after some operations to the query. The outermost query function returns the tuple itself after some operations to it. Finally, when the returned tuple reaches the main function, it becomes an output (`send_tuple(·)`).

The functionalities of the query modules are mapped to C code as shown in Fig. 3.4c. The operations such as slide timing module checking the timing and asserting the "slide" signal, or selection module checking the conditions specified in the `WHERE` clause of the query and asserting the "selected" signal, are mapped as conditions in **if** statements. This is because whether the operations in aggregation units are executed depends on these conditions, and therefore, the operations that depend on these conditions are executed under the corresponding **if** statements. If the query does not require windowing, the operations of the slide-timing module and aggregation module are omitted in the code.

3.4.3 Shallow Hardware Optimization in C Code

As shown in Fig. 3.3, there is a pipelining directive just before the while loop in the main function. Thanks to the HLS tool, all a parser developer has to do in order to pipeline the loop is to write this directive. However, before we pipeline the loop, some preparations must be made to the code. First, loops cannot have inner loops. We use another kind of directive to unroll the inner loops. This directive is also specified right before the loop, and the HLS tool will automatically unroll the loop. The other preparation for the loop is to design the hardware architecture, which is already done in the abstract hardware, so that it can be efficiently pipelined. In our code, there is a “valid” flag in the tuple with the intention of doing this. The “valid” flag enables the hardware to receive and send tuples at a constant speed, which leads to making the pipelined loop efficient.

Another optimization done in the code is to burst access the memory. The DRP evaluation kit we used does not have a network interface, it has to send and receive tuples to and from the external DRAM. The HLS tool that we used has a featured function (which is hidden in `receive_tuple(·)` and `send_tuple(·)`) to do this. This feature requires a predictive numbers of input and output tuples; therefore, the optimization that we made in order to efficiently pipeline also profits from this optimization. Note that this is the only optimization we made that is specific to DRP.

3.5 Evaluation

We compared the performance of the codes that were directly written in HLS C, and the codes generated by our parser from Q_1 to Q_4 (Fig. 3.5). We did not optimize the codes directly written in HLS C because optimization would be difficult for software engineers who generally do not have hardware development skills. Because the intended users of our parser are those who do not have hardware development knowledge, it is fairer to compare with non-optimized HLS codes.

Synthesis was done by the *DRP tool* which is a development tool suite bundled with the DRP evaluation kit. The synthesis tool included in the *DRP tool* is based on CWB which was developed by NEC [10]. The generated C codes are fully capable of synthesizing into hardware configurations unless the resource usage exceeds what is available on DRP [9]. DRP can be driven at various clock speeds and the DRP tool has a functionality to search the optimal clock speed for the application. Each result shown in Fig. 3.5 was measured at such clock speed.

For reference, we measured the throughput of the Intel Core i5-2520M processor (2.5GHz) running Q_1 (horizontal line in Fig. 3.5). The figure shows that the throughput of HLS C codes written without hardware development knowledge is about the same as that of the CPU. When using our parser, the throughput was more than twice as fast as the CPU. However, when we consider the power efficiency, assuming that the DRP consumes 500 mW and CPU 5 W, the DRP was 24 times more efficient than the CPU.

As long as the queries consist of `SELECT`, `FROM`, `WHERE`, and `INTO` clauses and the hardware resources are sufficient, the DRP's throughput will remain as high as shown in Fig. 3.5 because the system can be pipelined as shown in Fig. 3.3. However, when a query contains `GROUP BY` or `JOIN` clauses, which are outside the scope of this work, the throughput tends to go down vastly. [7] suffers from reduction of throughput when dealing with the `GROUP BY` clause because it requires a CAM to implement a grouping functionality. `JOIN` is a very difficult operation that many works have been seeking its efficient implementation in hardware [2, 5, 7]. Building translation functionalities of these clauses into our system is an important part of our future work.

3.6 Discussion

One metric that signifies the usage of a parser, besides the absolute performance of the resulting hardware, is how well the hardware's potential is extracted. To evaluate such significance, we measured the bandwidth usage of the generated hardware. Table I shows how efficiently the input bandwidth of DRP was used. Input bandwidth is the most critical limitation that constrains the overall throughput. Therefore, the input bandwidth usage can be a good barometer for evaluating how efficiently the hardware is used. According to the table, over 90% of the potential of DRP was extracted by using the parser, whereas only 50% of the potential was extracted without it.

The cost-consuming and hardware-dependent low-level operations such as wiring or scheduling were delegated to the HLS tool. There were only three optimizations that had to be done by the parser that involve hardware development knowledge: 1) specifying which loop to be pipelined, 2) enabling the memory burst accessing option, and 3) providing the basic architecture in

Table 3.1: Usage of input bandwidth.

Query	Hand written C	SQL to C Parser
Q_1	49.6%	90.8%
Q_2	50.4%	91.7%
Q_3	49.8%	96.5%
Q_4	33.5%	96.8%

order for the pipelining and memory burst accessing to be effective. Among these, the burst memory accessing option was the only optimization that was hardware specific in the architectural level. This means that the parser can be used in various hardware architectures as long as the HLS tool provides the abstracting function for controlling the I/O. Because a controller of the I/O is one of the most difficult component to design in lower levels and therefore requires hardware development knowledge and highly depends on the hardware architecture, it can be said that the parser is providing a good portability.

3.7 Conclusion

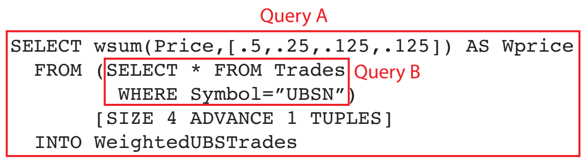
In this chapter, we showed that the concept of SQL-to-C HLS based compiler, proposed in [8] using an FPGA, was effective on DRP which features a dynamically reconfigurable architecture. The HLS C code was compiled to the hardware configuration by a state-of-the-art proprietary HLS tool that is customized to DRP. The results of the evaluation show that the parser enables software engineers to develop stream processing hardware that is 24 times more power efficient than a common CPU, or twice as fast as directly written HLS C code without any hardware development knowledge.

The SQL-to-C parser provided portability to the SQL-to-hardware compiler to work on DRP, by delegating the cost-consuming low-level optimization of DRP to the HLS tool, and using the high level function for controlling the I/O, which is dependent on DRP.

The optimizations except the I/O done by the parser do not depend on

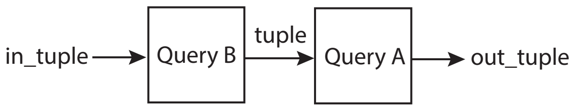
DRP at the architectural level as long as the synthesized hardware configuration does not violate the hardware limitation. Therefore, the only aspect of DRP that a developer of a SQL-to-C parser should be aware of is the I/O. This tendency can be generalized to various kinds of reconfigurable hardware.

The limitation of this work is that the sample queries we evaluated were rather simple. We will improve our parser and consider the larger or more complex queries that include grouping functionality.


Query A
Query B

```
SELECT wsum(Price,[.5,.25,.125,.125]) AS Wprice
  FROM (SELECT * FROM Trades
         WHERE Symbol="UBSN")
       [SIZE 4 ADVANCE 1 TUPLES]
  INTO WeightedUBSTrades
```

(a)

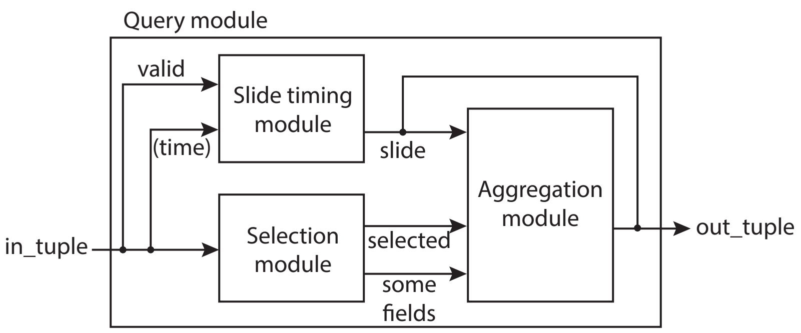


(b)

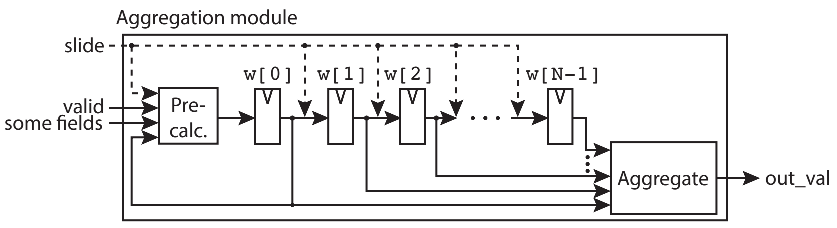
```
Trades queryA (Trades in_tuple)
{
  tuple = some_query_processingA(in_tuple);
  return tuple;
}

Trades queryB (Trades in_tuple)
{
  tuple = some_query_processingB(in_tuple);
  return queryA(tuple);
}

void main()
{
  /* Pipeline loop */
  while (1) {
    Trades in_tuple = receive_tuple();
    out_tuple = queryB(in_tuple);
    send_tuple(tuple)
  }
}
```

(c)

Figure 3.3: C function calls for nested queries.



(a)



(b)

```
<Tuple type> queryX (<Tuple type> in_tuple)
{
  if (in_tuple.valid) {
    if (sliede=window_slide_check(valid[,time])) {
      out_val = aggregate(w);
      slide_window(w);
    }
    if (selected=select()) {
      w[0] = pre_calc(w[0],slide,<some fields>);
    }
  }
  out_valid = <slide|selected>  ···(*)
  out_tuple = pack(out_valid,out_tuple);
  return <out_tuple|queryY(out_tuple)>;
}
```

🟦 Done in slide timing module.
🟩 Done in selection module.
🟥 Done in aggregation module.
(*) <u>`slide`</u> is selected if windowing is required, otherwise, <u>`selected`</u> is selected.

(c)

Figure 3.4: Query module architecture and its code.

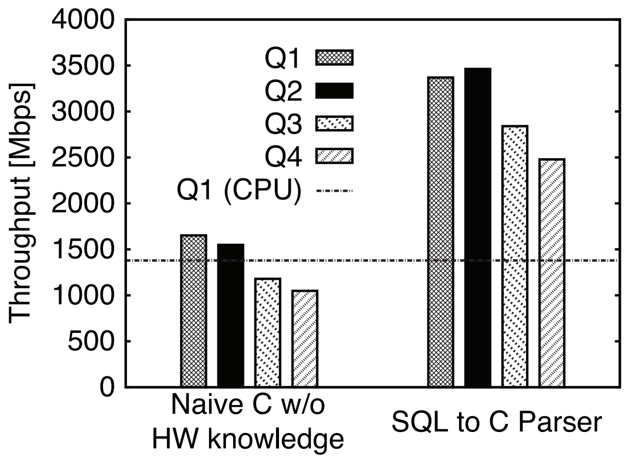


Figure 3.5: Throughput comparison between naive C written without hardware development knowledge and parsed C.

Bibliography

[1] Eric S Chung, James C Hoe, and Ken Mai. Coram: an in-fabric memory architecture for fpga-based computing. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 97–106. ACM, 2011.

[2] E. S. Fukuda, H. Kawashima, H. Inoue, T. Fujii, K. Furuta, T. Asai, and M. Motomura. C-based adaptive stream processing on dynamically reconfigurable hardware: a case study on window join. In *Proceedings of the 9th international conference on Reconfigurable Computing: architectures, tools, and applications (ARC)*, 2013.

[3] H. Inoue, T. Takenaka, and M. Motomura. 20Gbps C-based complex event processing. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[4] M. Kitsuregawa. Challenge for info-plosion. In *Proceedings of the 18th international conference on Algorithmic Learning Theory (ALT)*, 2007.

[5] T. Miyoshi, H. Kawashima, Y. Terada, and T. Yoshinaga. A coarse grain reconfigurable processor architecture for stream processing engine. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[6] M. Motomura. A dynamically reconfigurable processor architecture. *Microprocessor Forum*, 2002.

[7] R. Mueller, J. Teubner, and G. Alonso. Streams on wires - a query compiler for FPGAs. *Proceedings of the VLDB Endowment*, Vol. 2, No. 1, 2009.

[8] T. Takenaka, M. Takagi, and H. Inoue. A scalable complex event processing framework for combination of SQL-based continuous queries and C/C++ functions. In *Proceedings of the 2012 22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012.

[9] T. Toi, T. Awashima, M. Motomura, and H. Amano. Time and space-multiplexed compilation challenge for dynamically reconfigurable processors. In *IEEE International Midwest Symposium on Circuits and Systems*, 2011.

[10] K. Wakabayashi and B. C. Schafer. *"All-in-C" Behavioral Synthesis and Verification with CyberWorkBench*, pp. 113–127. Springer Netherlands, 2008.

Chapter 4

Acceleration by I/O Caching

4.1 Introduction

In Chapter 2 and 3, we discussed the issue of making reconfigurable hardware easier to be utilized by software programmers. In this chapter, we discuss the other issue involved in reconfigurable hardware when deploying it in data centers: the architecture of a data center with reconfigurable hardware is not fully established yet. As we discussed in Chapter 1, where to place the reconfigurable hardware in a data center is less significant for applications that require high throughput in comparison with applications that require low latency. For running low latency applications in data centers with reconfigurable hardware, it is better to place the reconfigurable hardware at the network interface of the server computers. Using the host general purpose processor as necessary will enable the system to be applicable to versatile applications.

Based on this architecture, we propose a new way of using reconfigurable hardware in data centers. Other works that uses reconfigurable hardware placed at the network interface tries to run different applications or functionalities between the reconfigurable hardware and the host general purpose processor. Unlike them, our method accelerates the application running on the host general purpose processor by taking advantage of functionalities and data locality of the application and caching them at the reconfigurable hardware.

We chose *memcached* as an example application of our method. Memcached has been widely accepted as a technology to improve response speed of web servers by caching data on DRAMs in distributed servers. Because of its importance, acceleration of memcached has been studied on various platforms. Among them, FPGA looks the most attractive platform to run memcached, and several research groups have tried to obtain much higher performance than that of CPU out of it. Difficulties encountered there, however, is how to manage large-sized memory (gigabytes of DRAMs) from memcached hardware built in an FPGA. Some groups are trying to solve this problem by using an embedded CPU for memory allocation and another group is employing an SSD. Unlike other approaches that try to replace memcached itself on FPGAs, our approach augments the software memcached running on the host CPU by caching its data and some operations at the FPGA-equipped network interface card (NIC) mounted on the server. The locality of memcached data enables the FPGA NIC to have a fairly high hit rate with a smaller memory. In this chapter, we describe the architecture of the proposed NIC cache, and evaluate the effectiveness with a standard key-value store (KVS) benchmarking tool. Our evaluation shows that our system is effective if the workload has temporal locality but does not handle workloads well without such characteristic. We further propose methods to overcome this problem and evaluate them. As a result, we estimate that the latency improved by up to 3.5 times over software memcached running on a high performance CPU.

4.2 Key-value Stores in Data Centers

Web service providers that have tremendous amounts of users and other information are eager to facilitate new technologies that enable their servers to handle more data traffic. One such technology employed by many web service providers is key-value stores (KVSs). A KVS holds data (values) with keys uniquely assigned (key-value pairs; KVPs), and sends them out as the data (value) is requested with the corresponding key. For its speed of finding the requested data in contrast to traditional relational database management systems (RDBMSs), many web service providers are now using

KVS databases such as DynamoDB at Amazon [1], BigTable at Google [2], memcached at Facebook [3], and many others. Memcached [5] is a technology that reduces the latency of data retrieval by storing KVPs in distributed servers' memories instead of fetching from the hard drives of database servers. Its simple data structure and computation have led to its wide adoption by various web service providers.

Memcached is used not only by Facebook, but also by a number of major web service providers such as Wikipedia and YouTube [5]. According to Facebook's research on their own memcached workloads, they use hundreds of memcached servers [3, 4]. In view of such extensive use, improving the memcached performance would have a large impact on web services' response. In fact, researchers have investigated the suitability of various hardware platforms for running memcached, from multiple low power CPUs [6–8] to many-core processors [9] and FPGAs [10]. Meanwhile, FPGA-based memcached systems are outperforming high performance CPUs such as Intel® Xeon® by an order of magnitude [11].

Although these efforts have improved the performance of memcached, major challenges remain. One such challenge is that it is difficult for FPGAs to efficiently manage a large memory size. Memcached servers usually have a few dozen gigabytes of memory, and such a memory space is too large for an FPGA to efficiently manage [12]. One group is trying to handle large memory size by utilizing a CPU core that is embedded in the FPGA [11]. The FPGA invokes the CPU to allocate or reallocate some blocks in the memory and stores data there. Another group employed an SSD to enlarge the memory space using a DRAM as a cache [13].

In this chapter, we propose a method that makes possible a low latency hardware memcached system with less memory than others require. Our method caches the subset of data stored in software memcached running on the host CPU at the network interface card (NIC) equipped with an FPGA and a DRAM memory. When the server receives a request from a client, the NIC tries to retrieve the data within the DRAM and sends it back if the data is found. If not, the NIC passes the request to the host CPU and the CPU executes the usual memcached operation. Since memcached data has locality, the NIC requires only a fraction of the amount of memory that the host server has. Furthermore, the commands the NIC cache does not support can be delegated to the host CPU; therefore only the frequently used memcached commands have to be supported on the NIC.

The contributions of this chapter are as follows:

- We proposed a method that reduces the delay of memcached by caching the memcached data at the NIC mounted on the server.
- We explained how the subset of memcached functionalities should be implemented on the FPGA equipped NIC in order to maintain data consistency between the NIC and the host CPU.
- We verified the improvement of the performance with a standard evaluation tool which is capable of evaluating various KVSs.
- We apply least frequently used (LFU) cache replacement algorithm that has slightly higher hit rate than least recently used (LRU) algorithm for workloads with Zipfian distribution. However LRU is a better choice when taking the hardware implementation cost into account.
- We analyze the effect of our system in relation to the workloads' characteristics in detail.
- We propose a method that enables a large size of cache on the NIC with small amount of block memory on the NIC's FPGA.

Although we focus on proving the effectiveness of our NIC caching architecture with memcached, it is important to note that this architecture can be applied to many other server applications that require lower latency as long as the data has temporal locality.

4.3 Background

4.3.1 Memcached

Memcached is a kind of KVS database that caches data on memories of distributed servers in the form of key-value pairs. As Fig. 4.1 shows, memcached servers store the subset of data stored in the database servers, which

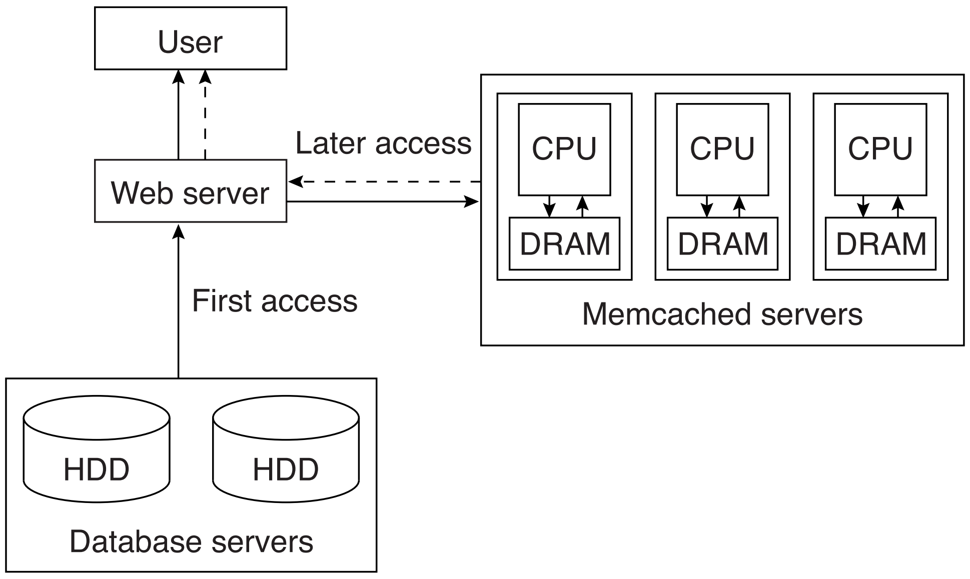


Figure 4.1: The operation of memcached.

usually use hard drives, in order to allow faster data access from the web server. Memcached servers often have a few dozen gigabytes of memory each and run in a cluster of several hundred servers. Data are not stored in the memcached server at the beginning, and the web server has to get the data

Table 4.1: Memcached commands

Command	Operation
SET	Store a KVP.
ADD	Store a KVP if the key is not found.
REPLACE	Replace a KVP if the key is found.
APPEND	Append data to a stored value.
PREPEND	Prepend data to a stored value.
CAS	Overwrite a value if the KVP is unchanged since last reference.
GET	Retrieves a value with a key.
GETS	Get a CAS identifier while retrieving a value with a key.
DELETE	Removes an KVP.
INCR/DECR	Increment or decrement a value.
STATS	Get an report of the memcached server statistics.

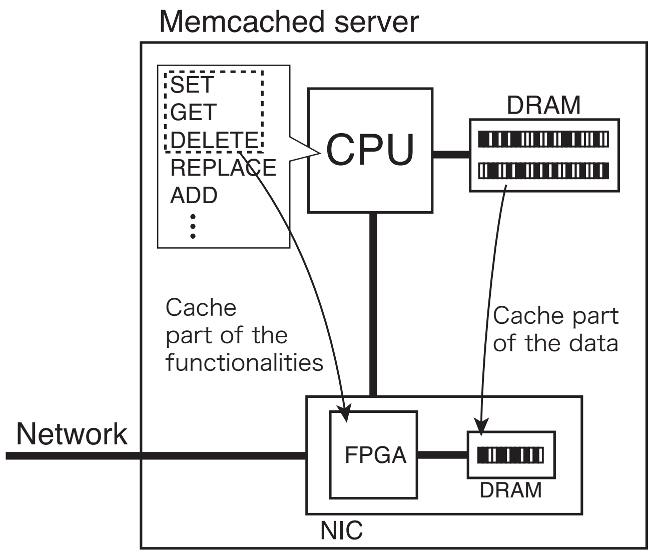


Figure 4.2: The image of the proposed method.

from the database servers. The web server sends the data back to the user and also sends a SET request with a paired key (250 bytes or smaller) and value (1 MB or smaller) to memcached to store the data. When the web server needs the same data later, it sends a GET request with the key to the memcached server, and the memcached server returns the value to the web server. Data that are not accessed frequently on the memcached server are evicted when the capacity is full. If the web server sends a GET request for data that has been already evicted, the memcached server notifies the web server that a cache miss has occurred. The web server will then check the database server and SET the data to the memcached server again.

Table 4.1 is a list of memcached commands. GET, SET and DELETE are the commands that are mainly used, and GET is the most frequently used command among them. According to a paper that reports the details of the memcached workloads of Facebook [3], the ratio of GET, SET and DELETE is 30:1:14 (exact ratio of DELETE not being provided in the paper, we estimated it visually from the chart). Therefore, the investigations we look through in Section 4.3.2 usually support only the GET, SET and DELETE commands.

Table 4.2: Description of YCSB workloads. (Originally shown in [15].)

Workload	**Operations**	**Record selection**	**Application example**
A–Update heavy	Read: 50% Update: 50%	Zipfian	Session store recording recent actions in a user session
B–Read heavy	Read: 95% Update: 5%	Zipfian	Photo tagging; add a tag is an update, but most operations are read tags
C–Read only	Read: 100%	Zipfian	User profile cache, where profiles are constructed elsewhere (e.g. Hadoop)
D–Read latest	Read: 95% Insert: 5%	Latest	User status updates; people want read the latest statuses
E–Short ranges	Scan: 95% Insert: 5%	Zipfian/Uniform	Threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id)

4.3.2 Related Work

Berezecki et al. evaluated the performance of memcached running on Tilera's TILEPro64 processor, which can allocate computations to its 64 cores [9]. Examining several configurations of cores running operations such as Linux kernel, network operations and others, the throughput per watt attained a maximum 2.4-fold increase over Xeon. However, the latency remained the same or worsened slightly from Xeon's 200 - 300 µs to TILEPro64's 200 - 400 µs.

Chalamalasetti et al.'s work was the first to try to utilize FPGA for accelerating memcached [10]. The system mainly consists of two parts: a network processing part and a memcached application part. The network processing part extracts memcached data from incoming packets and gives them to the memcached application part, and also does the reverse. Receiving the data from the network processing part, the memcached application part calculates hashes from the keys in order to determine the memory address at which the KVPs are stored and writes to or read from the memory. The performance of memcached improved dramatically in this scheme: throughput per watt attained 4.3-fold over Xeon and the latency became 2.4 to 12 µs.

Blott's group further improved the performance of memcached running on an FPGA by improving the UDP offload engine and adopting dataflow architecture [11]. They achieved over 15-fold higher throughput per watt than a Xeon and a latency of 3.5 to 4.5 µs. This work also features a CPU for allocating the memory. Their system stores the key in the block RAM and value in the external DRAM. The system we propose in this chapter is different from this: we store only the tag in the block RAM and store the key and the value to the external DRAM. This enables us to store more KVPs in the external DRAM, and as we propose in Section 4.7, this will further be extended to hash table compression method.

Another approach was proposed by two groups almost coincidently [7, 8]. Through dynamic analysis of memcached codes, they found that instruction cache misses or low branch prediction success rates caused by the frequent call of the network protocol stack, kernel and some library codes was the bottleneck. Their approach to get rid of this bottleneck was to replace the network process and some of the memcached process (GET request handling) software codes with hardware and integrate it into an SoC with a CPU core. This method was evaluated on an FPGA that had an embedded CPU core and yielded 2.3 to 6.1-fold higher throughput per watt than a Xeon. Our method is close to [7] and [8] in the sense that we execute part of the memcached process on hardware. However, we do not share the memory between the memcached hardware and the CPU, and thus the memory control of our method is simpler.

To gain a larger storage size on hardware memcached, Tanaka and Kozyrakis employed a solid state drive (SSD) in their FPGA based system [13]. Their approach is to store KVPs in the SSD on the FPGA board, using the DRAM on the same board as a cache. They achieved 14-fold higher throughput, 5- to 60-fold low latency and 12-fold higher throughput per watt than a Xeon.

Recently, a commercial memcached appliance that can be used in practice has been developed [14]. This appliance achieved 9.7-fold higher throughput than a Xeon by using a CPU and multiple FPGAs while the latency was 500 µs to 1 ms, which is larger than for a Xeon. Its throughput per watt has not been publicly announced.

4.4 Concept of NIC Cache

The basic idea of our method is to cache part of memcached server's data and functionalities to the NIC mounted on the same computer. According to Facebook's investigation into their own memcached workloads, there is some locality of access to their data [3]. On top of that, Facebook's investigation also indicates that among all memcached commands, SET, GET and DELETE account for most of the requests. This means that reducing the processing latency of only frequently accessed data should have a large impact on the web server's performance. The nearest place to the web server in the server computer on which memcached is running is the network interface. Therefore we try to efficiently reduce the latency by caching frequently used data and functionalities (SET, GET and DELETE) at the NIC and leaving the less frequently used data and functionalities to be handled by the host CPU. The NIC we assume to use has a fast connection to the network (several tens of Gbps), an FPGA, gigabytes of memory, and a fast connection to the host CPU (Fig. 4.2).

We assume our system to behave as follows. However, this is an example of adopting the FPGA NIC for memcached, and the behavior can be changed and adapted to various applications.

SET: The NIC stores the KVP to its DRAM and sends back a reply notifying the web server whether the KVP was properly stored. If a KVP already stored in the DRAM becomes evicted, a SET request with the evicted KVP is sent to the host CPU (write-back, write-no-allocate).

GET: If the key in the request is found in the NIC, the NIC returns a reply message with the corresponding value to the web server. Otherwise, the NIC sends the request to the host CPU, and the CPU searches for the key and returns it to the NIC. After the KVP is cached to its DRAM, it is sent back to the web server (read-allocate). If the key was not found at the CPU, it returns a reply notifying the web server that the key did not exist.

DELETE: If the key in the request is found in the NIC, it is invalidated and the request is sent to the host CPU. The CPU invalidates the data and returns a reply to the web server notifying that the KVP was successfully deleted.

4.5 Cache Simulation

In this section, we evaluate the NIC cache concept over software simulation in order to estimate its effectiveness. We implemented a cache simulator that behaves as mentioned in Section 4.4. Test workloads were generated with *Yahoo! Cloud Serving Benchmark* (YCSB), a standard benchmarking tool for KVS [15]. YCSB, cache simulator and memcached was placed as shown in Fig. 4.3. Requests generated by YCSB are sent to the cache simulator

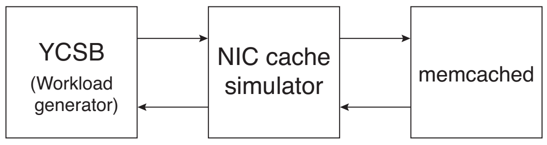


Figure 4.3: Connection of software modules.

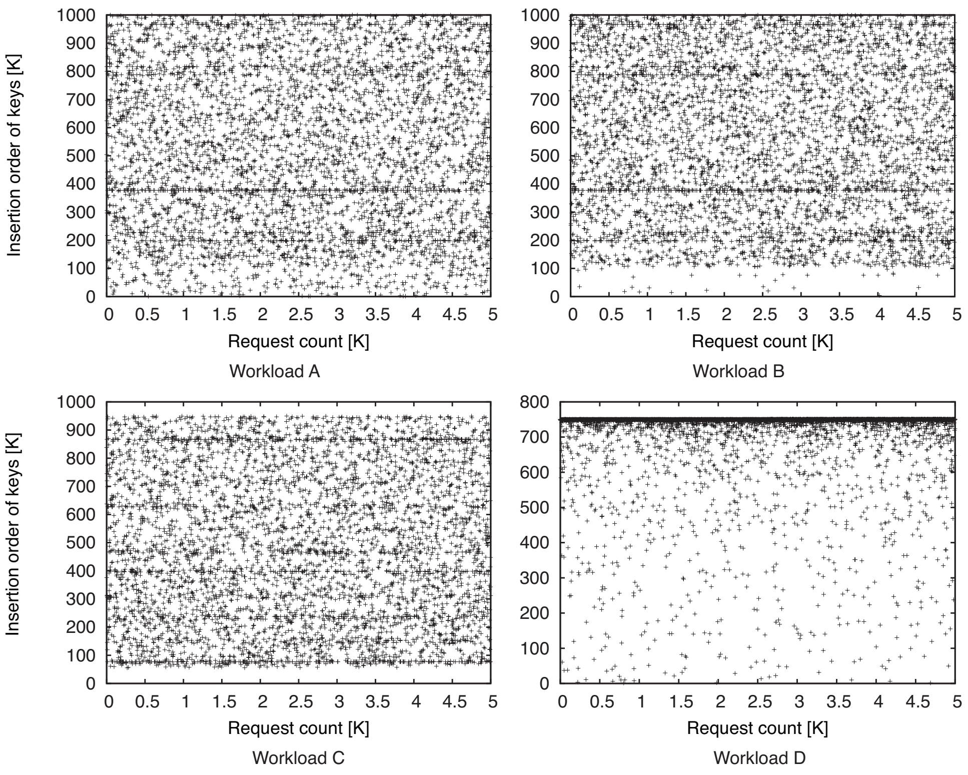


Figure 4.4: Key access distribution for the first 5,000 requests.

and the cache simulator processes the requests as described in Section 4.4, backed up by real memcached.

4.5.1 Testing Tool

YCSB provides workloads that simulate various KVS use cases. Table 4.2, quoted from [15], shows the characteristics of the workloads. Each workload is characterized by the ratio of commands and the record selection distribution. YCSB has load phase, which sends SET requests for all the keys for warm up, and transaction phase, which sends requests with the characteristics given in Table 4.2. All the results provided in this chapter are measured during the transaction phase.

Read, update and insert operations in the table correspond to memcached's GET, REPLACE and ADD commands respectively. In our experiment, however, we use SET for both update and insert operations. The

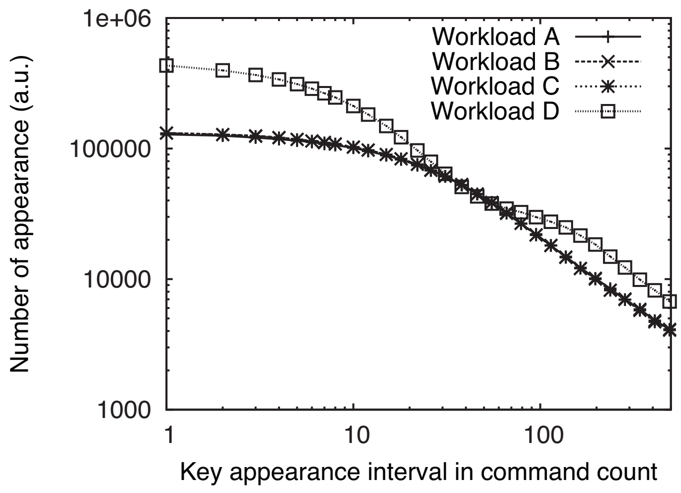


Figure 4.5: Appearance interval of same keys for all workloads.

difference between SET and update and insert is that update (REPLACE) and insert (ADD) check whether or not the data is already stored, and decide to store the data accordingly. Since we have to access the memory before we know whether the same key is stored, we used SET in place of REPLACE and ADD. The delay will be almost the same because checking whether the data is stored or not can be done in parallel with other operations. Regarding Workload E, we do not use it because memcached does not support the scan operation. Thus we use Workload A to D for our evaluation.

There are two record selection distributions: Zipfian and Latest. Zipfian is a distribution in which certain records are popular independent of their insertion order. An intuitive example is Wikipedia, where certain entries such as "Moore's Law" or "Transistor" are frequently viewed even though they were created years ago. On the other hand, Latest is a distribution in which the records added recently are the most popular ones. An example of Latest selection is Facebook's user updates where people mainly view their friend's recent posts.

Fig. 4.4 shows the access to each key for the first 5,000 requests. The x-axis is the number of requests, which approximately represents the time, while the y-axis shows the keys ordered according to their first appearance. You can see some stripes which are the popular keys in the Zipfian distribu-

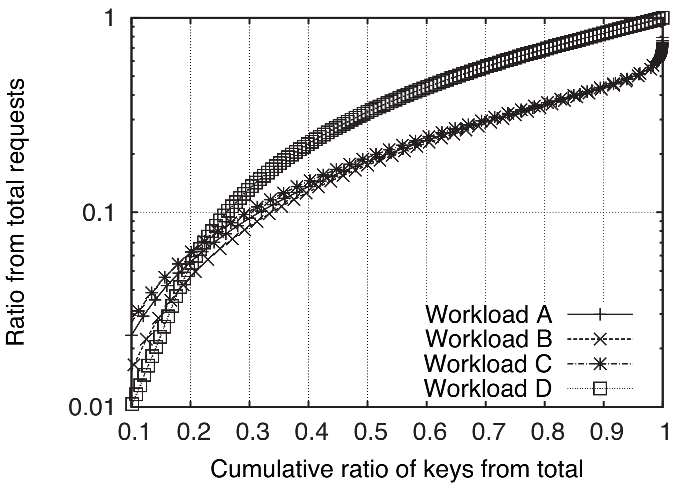
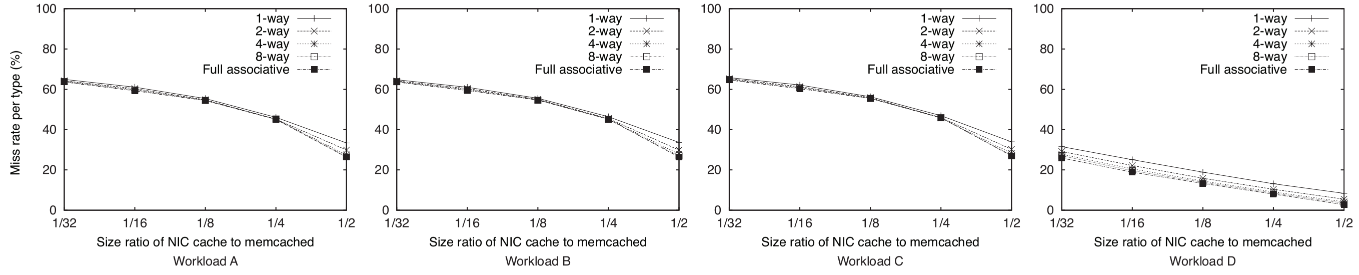


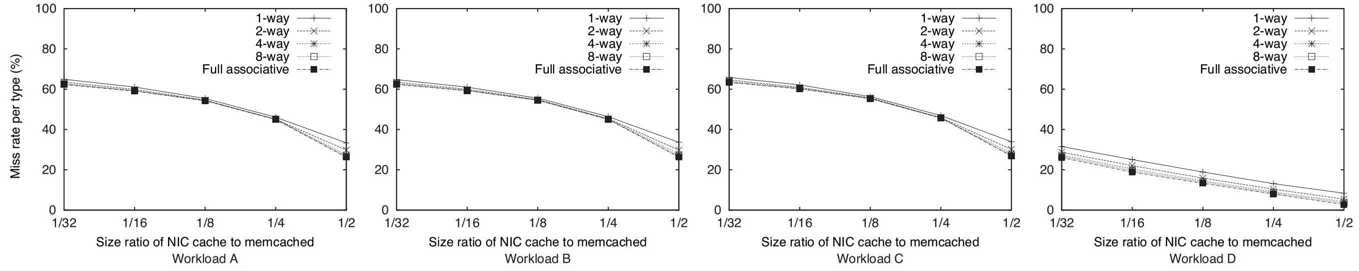
Figure 4.6: Cumulative ratio of keys.

tion (Workload A to C), and the key that appeared the last (the top in the chart) is intensively requested in Latest distribution (Workload D).

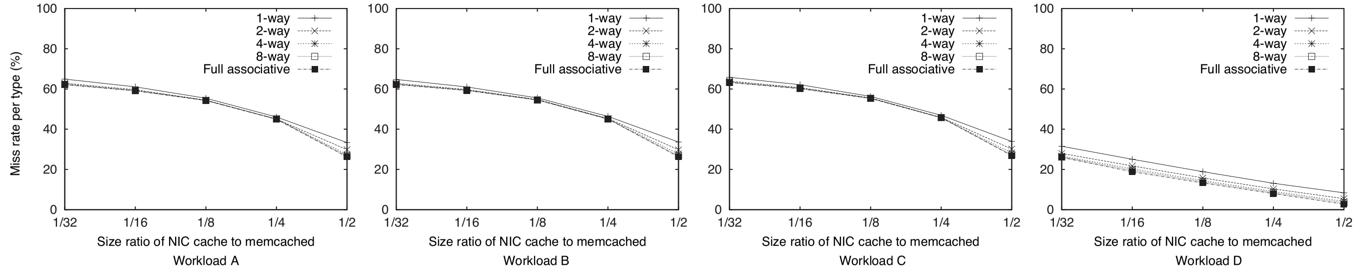
Fig. 4.5 shows that Workload D is unique in terms of the key appearance interval. The x-axis denotes the appearance interval of the same keys while the y-axis denotes the total number of each appearance interval. This figure signifies that Workload D has relatively less intervals between the same keys compared to Workload A to C.



(a) Miss rate for GET requests with FIFO replacement policy.



(b) Miss rate for GET requests with LRU replacement policy.



(c) Miss rate for GET requests with LFU replacement policy.

Figure 4.7: Miss rate for GET requests with various replacement policies.

Fig. 4.6 shows the appearance of keys in the form of cumulative distribution function (CDF). The x-axis is the ratio of the keys from the total keys, arranged in ascending order of their appearance ratio from total requests. The y-axis is the cumulative ratio of keys from total requests. Fig. 4.6 indicates that Workload D differs from Workload A to C also in the characteristic of key appearance. In the figure, the right ends of Workload A to C's graphs becomes almost vertical. This means that there is a large gap between the appearance probabilities of the popular keys, which make the dense areas in Fig. 4.4, and that of the rest of the keys.

4.5.2 Simulation Results

Fig. 4.7a, b and c show the miss rates for GET requests at the NIC cache with various associativity and capacity for FIFO, least recently used (LRU) and least frequently used (LFU) replacement algorithms respectively. The x-axis is the relative ratio of the NIC cache capacity to the memcached capacity running on the host CPU. We set the memcached capacity to 512MB and evaluated the cache size parameter from 1/32 to 1/2. The actual cache size we implemented, which we will discuss in Section 4.6, was 128 MB and this is 1/4 of the memcached capacity. (Rather than the absolute cache size, the ratio of the cache size to the host memcached size determines the miss rate.)

Apparently, the difference in miss rates among the three algorithms is very small. For Workload A to C, the miss rates are a few percent less with LRU and LRU than with FIFO when the capacity of the NIC cache is small. You can also see that Workloads A to C, which use the Zipfian distribution, have similar curves, while Workload D with Latest distribution have linearly decreasing miss rates as the NIC cache's capacity increases. (For better visibility of this, Fig. 4.8 features the miss rates for small cache sizes for Workload A with the three different algorithms.) As we mentioned earlier, workloads with Zipfian distribution have specific keys that are popular independently from when the key has recently been called. This makes us think that LFU, which tries to leave the popular keys in the cache, is a good solution for reducing the cache miss rate. However, LFU had larger, while the difference was still very small, effect than LRU only when the cache size was small (1/32 or 1/16 of memcached) and the cache associativity was relatively small (2-way or 4-way). This ineffectiveness comes from the very few number of popular keys in Workload A to C: The LFU has effect only when the ratio of the popular keys is relatively large in the cache.

We also carried out an experiment with a read-no-allocate policy, which means that the NIC does not cache the KVP on receiving the GET reply from the host CPU. This policy has the advantage of keeping the data consistency between the NIC cache and the host CPU easier. If a GET miss for a certain key occurs at the NIC and the subsequent request is a SET for the same key, the KVP set by the SET request at the NIC can be overwritten by the GET reply for the GET miss from the host CPU. This problem can be avoided in either by two ways: sending request from the NIC to the host CPU synchronously, or employing a read-no-allocate policy. Since synchronous requests can lead to an increase in average latency, employing a read-no-allocate policy can be beneficial if the miss rate at the NIC does not increase.

We found that the miss rate increased by less than a few percent for Workload A, B and D. For Workload C, however, the miss rate increased by more than ten percent (Fig. 4.9). This degradation comes from the command mix of Workload C. Unlike Workload A, B and D, Workload C does not send SET requests, so once a popular key is evicted from the NIC during the load phase, it cannot store it again in the transaction phase, and thus the miss rate rises.

4.6 Hardware Design

To prove the proposed method works correctly, we designed and implemented the system on an FPGA NIC. Note that the system described below is meant for making sure that the method we proposed above works under a simple one-to-one connection between the server and the client.

Although the cache has a relatively low hit rate for Workload C, as our initial implementation, we implemented the system with a read-no-allocate policy for its simple implementation. Fig. 4.10 shows the architecture of the NIC cache. The circuit implemented in the FPGA consists of five parts as

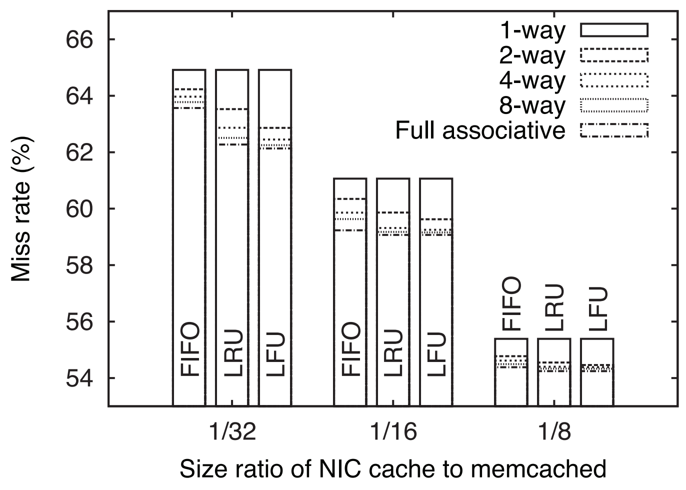


Figure 4.8: Miss rates for small cache sizes for Workload A with FIFO, LRU, and LFU cache algorithms.

described below:

Incoming packet handler: Non-memcached packets received from the network side are sent to the CPU without any operations so that the CPU could run not only memcached but also other server applications. On receipt of a memcached packet, the command, the key and the value are extracted from the packet and sent to the memory controller, hash calculator and the hash table. If the command is a GET and the memory controller returns a miss, the packet is sent to the CPU. If the memory controller returns a hit for a GET command, the packet is discarded. If the command is a SET or a DELETE, the packet is sent to the host CPU regardless of hit or miss returned from the hash table.

Outgoing packet handler: Outgoing packet handler does three things. First, it creates a packet in reply to a GET request using the key and the value given from the memory controller. Second, it receives memcached or other packets from the host CPU. Finally, it merges the packets from the two data sources (memory controller and the host CPU) and sends them out to the network. As mentioned in the beginning of this section, in our initial implementation, we do not cache data from the reply packets so as to simplify our implementation. Improving this behavior is a part of our future work.

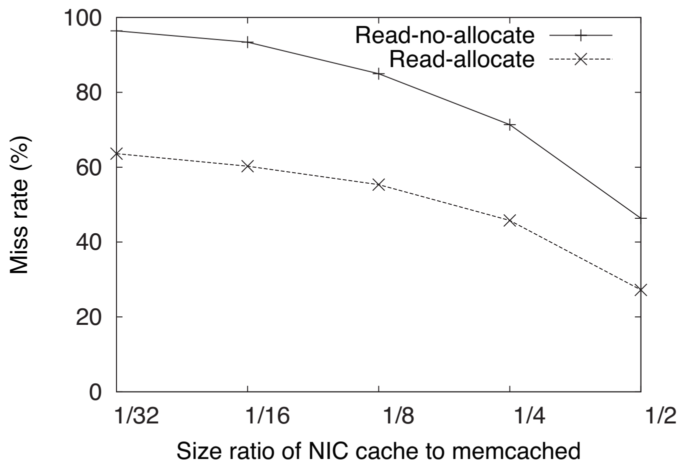


Figure 4.9: Miss rates with read-allocate and read-no-allocate for Workload C.

Hash calculator: Hash calculator receives a key from the incoming packet handler and calculates a hash with Jenkins's lookup3 function [16]. It produces a 32-bit hash from the key.

Hash table: Hash table manages where in the DRAM memory to store the KVP. More detailed structure is given in Fig. 4.11. The top 15 bits of the hash given from the hash calculator becomes the index of the hash table, and the lower 17 bits are written to the empty entry in the row, pointed to by the index, as a tag. The table is 8-way associative with a pseudo LRU replacement algorithm. Although LFU have a slightly better hit rate for small size and low associativity cache, we chose to implement pseudo LRU due to its lower implementation cost. The address of the memory is retrieved uniquely from the column and the row where the tag is stored. The key and the value are stored at the location on the memory where the address points. Memcached originally supports variable sizes of keys and values, but since YCSB supports fixed key and value sizes by default, we use fixed sizes. According to Facebook's investigation, key sizes are mostly less than 50 bytes and value sizes are less than a few hundred bytes. Therefore, we set the key size and the value size to 64 bytes and 448 bytes respectively to keep our hardware implementation of memory addressing simple by setting the size of the KVP to 512 bytes, which is a power of two. If the command given

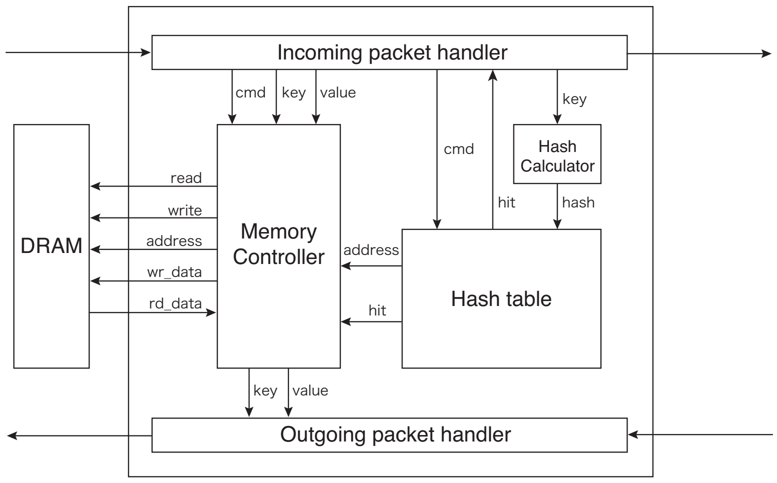


Figure 4.10: NIC cache architecture.

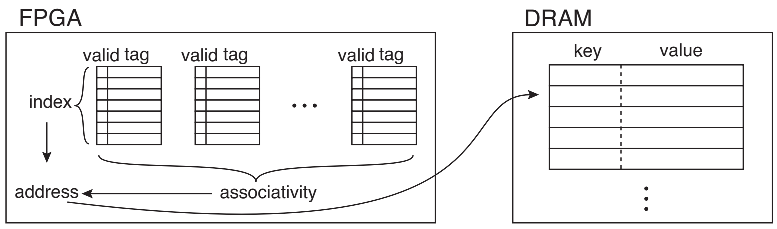


Figure 4.11: Correspondence of the hash table and the value storage.

from the incoming packet handler is a SET, the hash table stores the tag in a certain entry, setting its valid flag. If the command is a GET, the hash value is looked up in the hash table and hit/miss information is sent to the memory controller. Both in the case of SET and GET, the calculated address is sent to the memory controller. DELETE invalidates the valid flag if the hash value stored in the entry matches the hash value given from the hash calculator. value to the memory at the address given from the hash table. If the command is a DELETE, it does nothing.

Memory controller: The memory controller receives the command, the key and the value from the incoming packet handler, and also receives the address and the hit/miss information from the hash table. If the command is a GET, it sends a read signal and the address to the memory. Then the two keys from the incoming packet handler and the memory are compared to see whether they match. Since identical hash values can be generated from

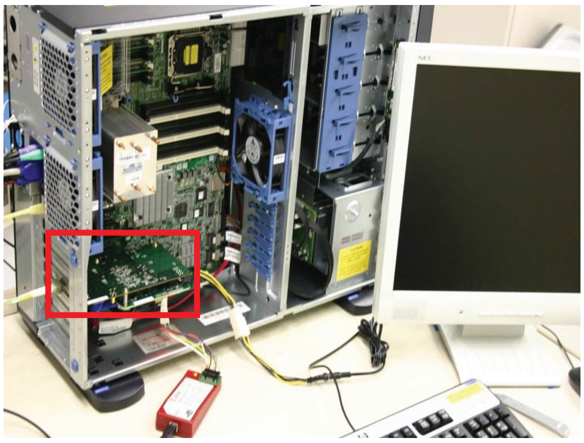


Figure 4.12: FPGA NIC mounted on a memcached server.

different key strings, the judgment of hit/miss at the hash table is uncertain. The keys should be checked here so as to make sure they are really identical. Provided that the keys match, the memory controller sends the key and the value to the outgoing packet handler; otherwise it does nothing. If the command is a SET, it writes the key and the value to the memory at the address given from the hash table. If the command is a DELETE, it does nothing. The memory controller also has a cache inside, which reduces the latency of external DRAM access.

4.6.1 Experimental Conditions

We used UDP protocol for the communication between the computer that runs YCSB and the computer that has the FPGA NIC and runs memcached. The two servers were connected with the 10 Gbps interconnect. Although memcached supports both TCP and UDP protocols, to make the packet offloading simple, we used UDP.

Our proprietary platform board consists of two 10 Gbps network interfaces, a Virtex-5 LX330T FPGA, a 1 GB DDR2 SDRAM memory and a PCI Express (Gen1 x8) interface. The host CPU is Intel Xeon E5-1620. Fig. 4.12 depicts the FPGA NIC mounted on a memcached server. How efficiently we can use the memory on the NIC depends on how large a hash table we can implement in the FPGA's block RAMs. Table 4.3 shows the resource usage.

Table 4.3: Design specification of FPGA

Number of used block RAM and FIFO	238 / 324 (86%)
Number of used slice LUTs	60314 / 207360 (29%)
Number of used lice Registers	64505 / 207360 (31%)

Table 4.4: Latencies of the system.

Network	9 µs
Reply from NIC (NIC cache hit)	29 µs
Reply from host CPU (SET)	87 µs

4.6.2 Latency

First of all, we confirmed that our system works for all Workloads A to D. Then we evaluated the latency of our system in three ways: First, to estimate the network latency, we implemented a system on the FPGA of the NIC that returns the request immediately after receiving it from the network. Next, we implemented the system described in Section 4.6, sent GET requests for the same key for several times, and got the minimum average. Finally, we sent SET requests with different keys several times and got the average latency. All the requests were sent from the server connected to the FPGA NIC with a 10 Gbps interconnect. The results are shown in Table 4.4. According to this table, we can estimate that the latency of the NIC cache was 20 µs (29 µs - 9 µs) and the latency of the host CPU was 78 µs (87 µs - 9 µs)

Based on the minimum latencies and the hit rates, we estimated the maximum improvement of our system for GET requests compared to using only the CPU (Fig. 4.14). The estimation was done with the following formula.

$$87\mu s/(hit\_rate \times 20\mu s + miss\_rate \times 87\mu s) \tag{4.1}$$

For workload A and B (Zipfian distribution), the latency improved at a maximum by about two times, and for Workload D (Latest distribution), the latency improved at a maximum by 3.5 times. Since the system was implemented with a read-no-allocate policy, the improvement of the latency of Workload C (Zipfian distribution) was a little less than for Workload A and B.

Table 4.5: Throughputs of the system based on RTL simulation.

Command	Throughput (ops/sec)
SET	191,424
GET (100% hit)	399,680-958,772
GET (100% miss)	138,465-165,480

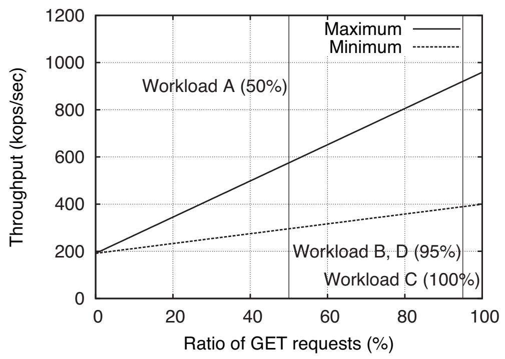
4.6.3 Throughput

Next, we evaluated the throughput of our system. We used RTL simulation to estimate the throughput. We gave a SET-only workload, a GET-only (100% hit) workload, and a GET-only (100% miss) workload as input. Table 4.5 shows the results. The numbers have certain ranges whose lower bound corresponds to 0% hit at the memory controller cache and upper bound to 100% hit. Since our DRAM access module in the memory controller is not optimised, the throughput is relatively slow compared to other works [11].

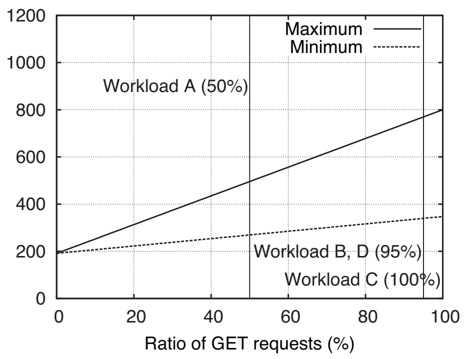
Based on these numbers, Fig. 4.13 estimates the throughput with different miss rates at the NIC cache. The vertical lines indicate the command mixes of the workloads; therefore the crossing points show the actual throughputs. In each graph, the throughput increases as the ratio of GET requests to the whole (SET and GET) requests increases. The graphs that are labeled "maximum" are the results for when the cache in the memory controller had 100% hit, and the "minimum" are for 0% hit. The actual throughput will be between the maximum and the minimum depending on the hit rate at the memory controller cache. For example, Fig. 4.13d shows that the throughput for workload A with cache size of 1/16 of the host DRAM (the miss rate is around 60% according to Fig. 4.7b) is between 210,000 to 350,000 ops/sec.

4.7 Cache Size Maximization

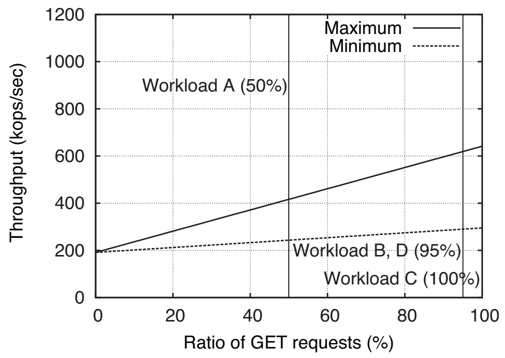
The cache size of our system is determined by the number of the entries in the hash table implemented on the block memories in the FPGA. In other words, the size of the available block memories can become a bottleneck if a larger cache size is required. In fact, as shown in Table 4.3, we have already used 86% of the block memories for having 128 MB cache, so it is difficult to



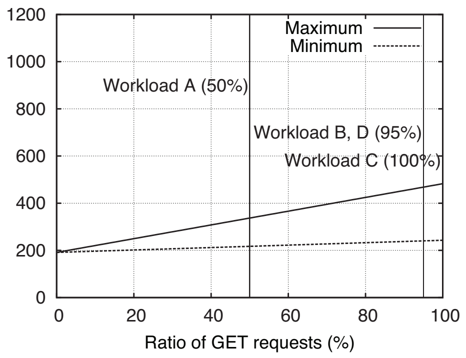
(a) Throughput for 0% miss rate



(b) Throughput for 20% miss rate



(c) Throughput for 40% miss rate



(d) Throughput for 60% miss rate

Figure 4.13: Throughput of the system with various hit rates.

have a larger cache size. In this section, we consider and evaluate a method that enlarges the cache size with a limited amount of block memories by narrowing the tag width.

In our system, we store tags, which are the lower 17 bits of the hash value calculated from the key, instead of storing the key itself in the hash table. (Along with the 17-bit tag, the cache uses 1-bit valid bit and 1-bit MRU bit. MRU bit is used for implementing pseudo-LRU. Throughout this chapter, we do not include the valid bit and the MRU bit in the term "tag.") Therefore, on a SET, a certain KVP in the cache can be overwritten by another KVP which has a different key but has the same index and tag. On a GET however, the system ensures that it will not return a KVP that was not requested by checking whether the key in the retrieved KVP from the DRAM matches the requested key. Conversely, it is possible to reduce the block memory usage by making the width of the tag smaller as long as

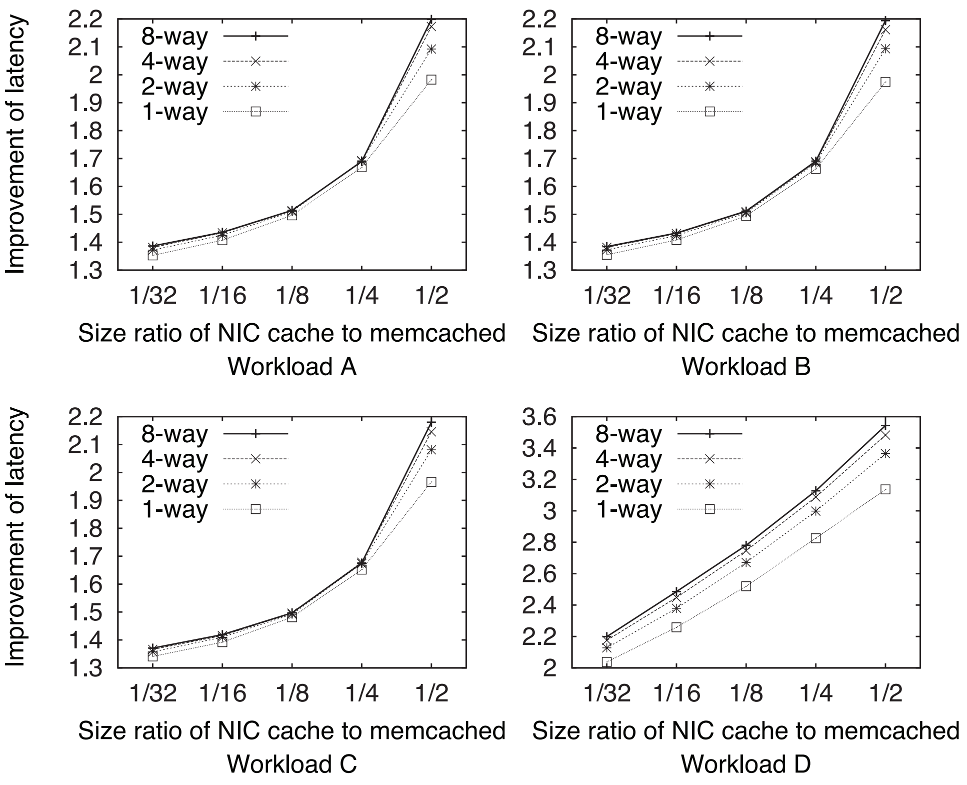
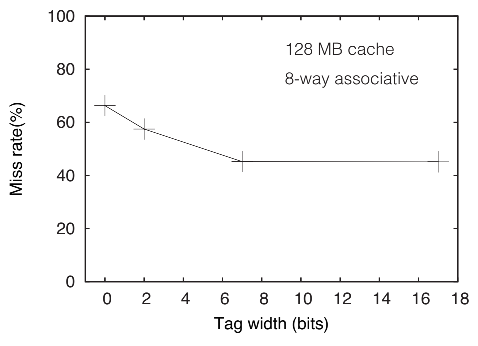


Figure 4.14: Latency improvement with various associativity and cache capacity.

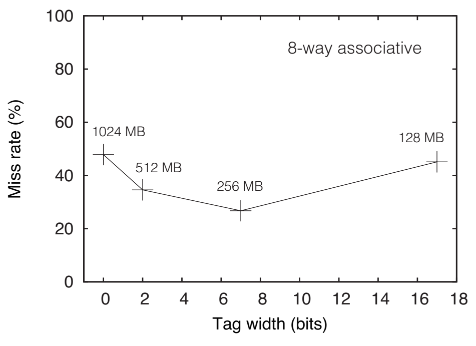
the retrieved key is checked. For such purpose, we investigated the relations between the tag width and the miss rate.

First, we investigated the relation between the tag width and the miss rate. Fig. 4.15a shows the miss rate with tag widths of 0, 2, 7 and 17. (As we mentioned above, these numbers do not include the valid bit and the MRU bit.) We used Workload A and 8-way associative hash table for this evaluation. The figure shows that there is little difference in miss rates between the cases of 17-bit and 7-bit tags. As the tag width gets further narrower towards the left, the miss rate becomes larger due to the increase of chances of overwriting the keys with different keys.

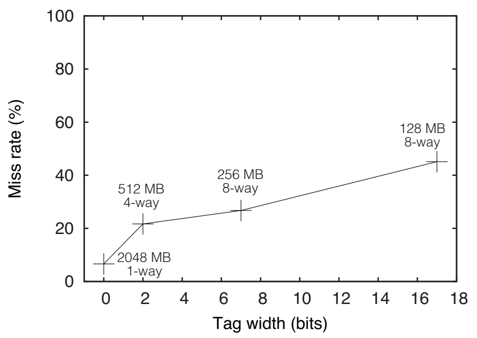
Next, we evaluated the the effect of narrowing the tag and enlarging the cache size with constant block memory size (Fig. 4.15b). We used Workload A and 8-way associative hash table also in this evaluation. As the tag width



(a) Miss rates with variable tag width, constant cache size, and constant associativity.



(b) Miss rates with variable tag width, variable cache size, and constant associativity.



(c) Miss rates with variable tag width, variable cache size, and variable associativity.

Figure 4.15: Miss rates with constant block memory size.

gets narrower, there are more room in the block memories for increasing the hash table's index size, and therefore the cache size can be increased. When the tag is narrowed from 17 bits to 7 bits, the miss rate decreases because the cache size increases while the chance of KVPs being overwritten does not increase. For smaller tag widths, however, the negative effect of overwriting the key becomes larger than the positive effect of larger cache sizes, and result in an increase of miss rate.

The block memories can further be exploited. Fig. 4.15c shows the hit rates of when the block memories are fully exploited for each tag width. Although if the hash table is 8-way associative, only four different tags can be stored in a single row when the tag width is two ($4 = 2^2$). Instead, we reduced the associativity to four and doubled the number of the index when the tag width is two. In the case of 0-bit-width tags, in other words, in the case of no tags, not only the tags but also the MRU bits are no longer necessary since the associativity is virtually 1-way. The hash table can then consist only of valid bits. Therefore the cache size can be twice as large as that of when the MRU bit still exist. Although the chances of keys overwritten by different keys increase, the effect of increase in cache size outraces such effect and therefore the miss rate decreases when the tag width becomes smaller. Throughout these experiments, it can be said that the cache size can be increased without worsening the miss rate even though the tag width is narrowed.

4.8 Discussion and Future Work

In order to keep the implementation simple and avoid data inconsistency between the NIC cache and memcached, we decided to employ a read-no-allocate policy. As a consequence, this leads to a decrease in the hit rate for Workload C, which has only GET requests. However, employing read-allocate instead will lead to a drop of NIC cache's average latency. Finding a solution to keep data consistency and high performance at the same time is one of the largest tasks remaining.

Another limitation in this work is that YCSB, the benchmarking tool we used, uses fixed sized keys and values for evaluation. If the web server sends a SET request to our system with variable key and value sizes, while we have fixed sized space to store the KVP as described in this chapter, we have to ignore the request at the NIC and leave it to the CPU to handle. This will lead to a decrease in the hit rate of GET requests and thus the performance of the system will degrade. To overcome this problem, we should employ a method to accept any key and value sizes with an efficient memory allocation technique.

Although there is only 128 MB cache on our NIC, it can be expanded in two ways. First, As we discussed in Section 4.7, the cache size can be doubled without increasing the miss rate by narrowing the tag width. (The cache size is proportional to the size of the hash table.) Second, some of the recent FPGAs like Vertex UltraScale have more than ten times of block RAMs than the one we used has. Altogether, there can be 20 times larger cache (2.5 GB) than the current size (128 MB) on the NIC. In this case, our evaluation considers 5 to 80 GB cache.

Compared to CPU caches, our NIC cache has higher miss rates. We found out that the NIC cache does not show high hit rate with FIFO or LRU for workloads with Zipfian key distribution. Therefore in this we tried LFU, an cache replacement algorithm that leaves the popular keys in the cache, expecting the hit rate to improve. However, LFU had almost no effect. This is because the popular keys in the YCSB's workloads with Zipfian distribution is so few that they could remain in the cache even with other cache replacement algorithms. Our next goal is to improve the hit rates for workloads with Zipfian distribution.

4.9 Conclusion

In this chapter, we proposed a method to accelerate low latency data center applications by caching the frequently used functionalities and the data running on the host general purpose processor at the reconfigurable hardware placed at the server's network interface. In particular, we proposed a method to improve the latency of memcached by caching its data at the NIC and replying to the client immediately from the NIC when the requested data is found. The evaluation was done with a common KVS evaluation tool, YCSB.

With the cache parameters determined through software simulation, the hardware evaluation showed that our method improves the latency by up to 3.5-fold for GET requests for keys with the Latest distribution compared to a Xeon. Our further investigation showed that the size of the block RAM on the FPGA is less likely to become the bottleneck of the cache size if the tag width of the cache is narrowed. We simplified our method by fixing the sizes of the key and the value, and the hit rate might drop if we adopt variable key and value sizes. We will try improving the hit rate by employing a better cache algorithm and by utilizing the DRAM with an efficient memory allocation method.

Furthermore, we believe that our approach to improve the performance of the application by caching the data at the NIC is applicable to other applications as well. We will try generalizing our method as a new computation architecture.

Bibliography

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of 21st ACM Symposium on Operating Systems Principles*, 2007, pp. 205–220.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, 2008.

[3] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Characterizing facebook's memcached workload," *IEEE Internet Computing*, vol. 99, pp. 41–49, 2014.

[4] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, 2013, pp. 385–398.

[5] `http://memcached.org/`.

[6] W. Lang, J. M. Patel, and S. Shankar, "Wimpy node clusters: What about non-wimpy workloads?" in *Proceedings of the 6th International Workshop on Data Management on New Hardware*, 2010, pp. 47–55.

[7] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing soc accelerators for mem-

cached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 36–47.

[8] M. Lavasani, H. Angepat, and D. Chiou, "An fpga-based in-line accelerator for memcached," *IEEE Computer Architecture Letters*, vol. 99, pp. 1–4, 2013.

[9] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele, "Many-core key-value store," in *Proceedings of the 2nd International Green Computing Conference and Workshops*, 2011, pp. 1–8.

[10] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An fpga memcached appliance," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2013, pp. 245–254.

[11] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István, "Achieving 10gbps line-rate key-value stores with fpgas," in *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013, pp. 1–6.

[12] A. Wiggins and J. Langston, "Enhancing the scalability of memcached," `http://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached-0`.

[13] S. Tanaka and C. Kozyrakis, "High performance hardware-accelerated flash key-value store," 2014, presented in the 5th Annual Non-Volatile Memories Workshop.

[14] "Convey computer memcached appliance," `http://www.conveycomputer.com/files/1813/7998/4963/CONV-13-046_MCD_Datasheet.pdf`.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010, pp. 143–154.

[16] B. Jenkins, "Lookup3.c, for hash table lookup," 2006, `http://burtleburtle.net/bob/c/lookup3.c`.

Chapter 5

Conclusion

Many web services such as search engines, social network, e-mail, and shopping are produced every day and they have become essential to our daily life. As a consequence, however, the energy consumption of data centers, which are the very basic infrastructure of web services, has become a serious issue. The miniaturization of the silicon process, which has been a major factor in improving of processor performance and thus a major factor of web services' improvements, is predicted to be approaching to its end, and it is becoming difficult to expect continued performance improvement. Although the ASIC can solve the energy consumption issue and the performance issue simultaneously, the ASIC does not compensate the cost due to the web service's fast development speed. One possible solution is to use reconfigurable hardware. However, reconfigurable hardware has two issues in data centers: it is difficult for software developers to use, and its usage and architecture in the data center are not firmly established.

In this study, we aimed at revealing the difficulties of a software engineer's developing application-specific hardware, proposing a method to develop and use hardware to avoid such difficulties and proposing a method and server architecture that can take advantage of reconfigurable hardware in data centers. In order to achieve such objectives, we first clarified the necessary perspectives for software developers to design hardware with a state-of-the-art design method through the process of developing a stream processing hardware with a high-level synthesis tool. Then, on the basis of this perspective, we developed a simplified system that enables software developers to design stream processing specific hardware without any hardware development knowledge. Furthermore, we proposed a computing method and architecture that take advantage of reconfigurable hardware and data center application characteristics. In the rest of this chapter, we summarize the results of our work.

In Chapter 2, we developed a processor that does window join, which is an operator of stream processing, by synthesizing hardware from a source code written in C with a high-level synthesis tool. By analyzing the difficulties we faced during the development process, we revealed five awarenesses that software developers should have

- I/O
- Buffering
- Resource amount
- Loop
- Resource type

The window join processor that we developed achieved 19x higher energy efficiency than software.

In this work, we divided the window into small sizes and concentrated on a single subwindow. This approach is applicable to large windows by using the same hardware massively, and effective for reconfigurable hardware deployed in cloud systems. However, because the implemented circuit is optimized to a certain data size and the available resources on DRP, implementing the same application to different hardware will have to be re-optimized to the target hardware. Inventing a method that is capable of synthesizing circuits to various sizes of data and reconfigurable hardware is highly beneficial for realizing the previously mentioned final step. For building such a method, the five awarenesses we proposed would be a useful policy.

In Chapter 3, we proposed a method that enables software engineers to develop hardware by using a parser that converts StreamSQL to HLS-intended C code, and proved that it is effective in practical use through measurements of the system we implemented on DRP. The parser has the following characteristics:

- It converts StreamSQL queries via a hardware framework that corresponds to its logical structure.
- It achieves high performance by having a hardware structure that takes advantage of the data-flow type processing nature that the StreamSQL has.
- The generated C code is optimized to the target hardware and software developers do not have to be aware of the hardware.

The performance of the resulting hardware showed more than twice as high throughput than software and extracted over 90% of the I/O bandwidth of the chip.

This work was meaningful in that it reduces the software programmer's cost for developing reconfigurable hardware based systems. Choosing the stream version of SQL, which is heavily used in data centers, makes the impact of the work even larger. However, the developer of the parser that converts SQL queries to C code still needs knowledge of hardware development. This cost should decrease as the HLS technology progresses or by using virtualization technology of reconfigurable hardware. If the parser imports such technologies and combines with a technology that offloads the complex functionalities that were not supported, the parser will be edible for making reconfigurable hardware transparent in data centers.

In Chapter 4, we proposed a system that enhances the unmodified database software system by supporting the general-purpose processor with a dedicated hardware that is placed in the I/O path of the general-purpose processor. We took memcached, an in-memory key-value store, as an example of this method. The system has the following characteristics:

- The hardware implemented on the FPGA is simplified by supporting only the most frequently used commands among all memcached commands.
- Placing the hardware on the network interface reduces the latency of the memcached request compared to the usual system.

- Taking advantage of the temporal locality of the memcached data, caching only a part of the memcached data enhances the performance.

The implemented system yielded 10 times smaller latency than only the software memcached.

This work contributes to the data center architecture to seamlessly use general-purpose processors and reconfigurable hardware and to make reconfigurable hardware use transparent. To enable end users to use this architecture, which caches the general-purpose processor's frequently used functionalities and data to the reconfigurable hardware placed at the network interface of the server, on IaaS where end users have the freedom to allocate computing resources on demand, more improvement is needed. Functionalities such as moving recently used functions or processes at the general-purpose processor to the reconfigurable hardware or dynamically changing the characteristics of the data that is cached at the network interface will be required. For this improvement, taking in the achievements of research on hardware development technology is crucial. In addition, processing methods or programming methods that intend to be moved between the general-purpose processor and the reconfigurable hardware will be needed.

Data centers were mainly used for storing various data originally; thus, there is a high demand for making database applications faster and more efficient. In fact, there are many Internet services based on SQL that are provided to the end users by network service provider companies. In such a sense, the methodologies that were proposed in this thesis, i.e. the hardware generating method from stream SQL and the acceleration method of key-value store, which is the basis of recent large-scale non-relational databases, have a large impact on database hardware application research.

However, the recent trend of opening up the computing resources in data centers to end users as IaaS is accelerating the diversification of applications running on data centers. There are more cases in which non-database applications such as image processing, data analysis, pattern recognition, and machine learning, are being executed on data centers. Those applications are often incapable of describing with SQL. In fact, some database researchers are trying to enhance the current SQL grammar so that various non-SQL functionalities can be embedded in SQL queries. It is not easy to establish a general methodology that can convert such diverse applications to efficient hardware. As shown in Chapter 2, developers need hardware development knowledge to develop efficient hardware even with HLS. To solve this problem, researchers are trying to narrow the hardware design space by making a template for each application domain or using different programming paradigms, such as functional programming, from the procedural programming on which many current HLS tools are based.

The rapid advancement of hardware acceleration of database applications is due to the large number of researchers and developers that are eager to accelerate database applications in data centers. The method we showed in Chapter 3, the hardware development method from stream SQL, is based on such past works. However, since non-database applications in data centers are diverse, creating a general method for hardware accelerating those applications will be more difficult. The more actively such research is done, the more rapidly it will advance. Research on hardware acceleration of individual application and research on a general hardware acceleration method for diverse applications will mutually enhance each other. The most efficient way to produce such an ascending spiral is to open the data center equipped with reconfigurable hardware to the public. Such a platform will involve not only researchers but also more developers in the industry in hardware acceleration research and accelerate the advancement speed.

Acknowledgements

I want to thank Professor Masato Motomura and Professor Tetsuya Asai at Hokkaido University for his advice during the writing of this thesis and pursuing my Ph.D. research. I am also deeply grateful to Professor Hiroki Arimura, Professor Eiichi Sano and Professor Junichi Motohisa at Hokkaido University for their advice and fruitful discussions.

Dr. Hiroaki Inoue at NEC Corporation, Dr. Takashi Takenaka at NEC Corporation, Professor Hideyuki Kawashima at University of Tsukuba, Mr. Taro Fujii at Renesas Electronics, and Koichiro Furuta at Renesas Electronics provided me with a lot of suggestions on the research and hardware development techniques.

I also would like to thank the members of my research team, Dahoo Kim, Tsunaki Sadahisa, and Kasho Yamamoto for supporting my research by developing the software and hardware components of the system we proposed.

List of Publications

1. Journal Papers

1. E.S. Fukuda, H. Inoue, T. Takenaka, D. Kim, T. Sadahisa, T. Asai, and M. Motomura, “Enhancing memcached by caching its data and functionalities at network interface,” Journal of Information Processing, vol. 23, no. 2 (2015).

2. E.S. Fukuda, H. Kawashima, H. Inoue, T. Asai, and M. Motomura, “C-based de- sign of window join for dynamically reconfigurable hardware,” Journal of Computer Science and Engineering, vol. 20, no. 2, pp. 1-9 (2013).

2. International Conferences

1. E.S. Fukuda, H. Inoue, T. Takenaka, D. Kim, T. Sadahisa, T. Asai, and M. Motomura, “Achieving higher performance of memcached by caching at network interface,” The 2014 International Conference on Field Programmable Technology (FPT), Shanghai, China (Dec. 10-12, 2014).

2. E.S. Fukuda, H. Inoue, T. Takenaka, D. Kim, T. Sadahisa, T. Asai, and M. Motomura, “Caching memcached at reconfigurable network interface,” The 24th International Conference on Field Programmable Logic and Applications (FPL), Munich, Germany (Sep. 2-4, 2014).

3. E.S. Fukuda, T. Takenaka, H. Inoue, H. Kawashima, T. Asai, and M. Motomura, “High level synthesis with stream query to C parser: Elimi-

nating hardware development difficulties for software developers," The 18th Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI), pp. 310-315, Sapporo, Japan (Oct. 21-22, 2013).

4. E.S. Fukuda, H. Kawashima, H. Inoue, T. Asai, and M. Motomura, "Exploiting hardware reconfigurability on window join," The 2013 International Conference on High Performance Computing and Simulation (HPCS), Helsinki, Finland (Jul. 1-5, 2013).

5. E.S. Fukuda, H. Kawashima, H. Inoue, T. Fujii, K. Furuta, T. Asai, and M. Motomura, "C-based adaptive stream processing on dynamically reconfigurable hardware: window join case study," The 9th International Symposium on Applied Reconfigurable Computing (ARC), Los Angeles, U.S.A. (Mar. 25-27, 2013)

3. Domestic Conferences (in Japanese)

1. 福田 エリック駿, 定久 紀基, 井上 浩明, 竹中 崇, 浅井 哲也, 本村 真人, "二重キャッシングによる Memcached 高速化の提案," 電子情報通信学会 リコンフィギャラブルシステム研究会, 慶応義塾大学 (日吉), 2014 年 1 月 28-29 日.

2. 福田 エリック駿, 川島 英之, 井上 浩明, 藤井 太郎, 古田 浩一朗, 浅井 哲也, 本村 真人, "リコンフィギュラブルハードウェアを用いた高速ストリーム処理の一検討," 電子情報通信学会 リコンフィギャラブルシステム研究会, 北陸先端科学技術大学院大学 (能美), 2013 年 9 月 18-19 日.

3. 福田 エリック駿, 川島 英之, 井上 浩明, 浅井 哲也, 本村 真人, "C 言語による動的リコンフィギュラブルハードウェアへの Window Join の実装," 電子情報通信学会 情報ネットワーク研究会, 福井大学 (福井), 2013 年 6 月 20-21 日.

4. Journal Papers (Co-authored)

1. D. Kim, I. Hida, E.S. Fukuda, T. Asai, and M. Motomura, "Reducing power and energy consumption of nonvolatile microcontrollers with transparent on-chip instruction cache," Circuits and Systems, vol. 5, no. 11, pp. 253-264 (2014).

5. International Conferences (Co-authored)

1. K. Yamamoto, E.S. Fukuda, T. Asai, and M. Motomura, "An accelerator for frequent Itemset mining from data stream with parallel item tree," The 19th Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI), Yilan, Taiwan (Mar. 16-17, 2015).

2. D. Kim, I. Hida, E.S. Fukuda, T. Asai, and M. Motomura, "A study of transparent on-chip instruction cache for NV microcontrollers," The 7th International Conference on Advances in Circuits, Electronics and Micro-electronics (CENICS), Lisbon, Portugal (Nov. 16-20, 2014).

3. D. Kim, E.S. Fukuda, T. Sadahisa, T. Asai, and M. Motomura, "Hardware architecture for accelerating key-value retrieval implemented on FPGA," The 3rd Japan-Korea Joint Workshop on Complex Communication Sciences (JKCCS), Busan, Korea (Oct. 27-28, 2014).

6. Domestic Conferences (Co-authored, in Japanese)

1. 定久 紀基, 山本 佳生, 金 多厚, 福田 エリック駿, 浅井 哲也, 本村 真人, "Locality-Sensitive Hashing のスケーラブルなハードウェアアーキテクチャの FPGA 実装," 電子情報通信学会総合大会, 立命館大学びわこ・くさつキャンパス (草津), 2015 年 3 月 10-13 日.

2. 定久 紀基, 山本 佳生, 金 多厚, 福田 エリック駿, 浅井 哲也, 本村 真人, "類似検索を行う Locality-Sensitive Hashing のスケーラブルなハードウェアアーキテクチャ," 電子情報通信学会集積回路研究会・コンピュー

タシステム研究会合同 平成 26 年度若手研究会, 機械振興会館 (東京),
2014 年 12 月 1-2 日.